Journal of Software Engineering for Robotics 2(1), September 2011, 20-39
ISSN: 2035-3928

Generic middleware support for coordinating robot software components: The Task-State-Pattern

Ingo Lütkebohle[1,*] Roland Philippsen[2] Vijay Pradeep[3] Eitan Marder-Eppstein[3] Sven Wachsmuth[1]
[1] Applied Informatics Group, Bielefeld University, Germany
[2] Intelligent Systems Laboratory, Halmstad University, Sweden.
[3] Willow Garage Inc, Menlo Park, CA, USA

Abstract—Robot software systems are (again) reaching levels of size and complexity that makes them difficult to construct, evolve, and maintain. One current issue is that systems are increasingly built to perform many different tasks in parallel, each of which must be coordinated and monitored to achieve a goal. If all components were to require different interfaces, system complexity would rapidly grow. General interfaces partially exist on the conceptual level, but their implementations are typically strongly linked to particular architectural proposals, thus reducing re-use and comparability.
This paper presents an architecture-agnostic design pattern for the coordination-related component interaction. It results in a simple and clean component interface to invoke specific functionality, monitor task progress, and update the goals of running tasks. It provides an abstract coordination interface with high observability for the development of coordination and architecture. It thus provides value to all stakeholders in the design and implementation of robot software systems: component developers, coordination developers, and system architects.
We trace the convergence of concepts and approaches from early coordination systems and through various abstraction proposals. Recently, two very similar realizations were developed independently by the authors. This paper presents the underlying insights and practical experience as a generic software engineering method which we named the *Task-State-Pattern*. We describe the functionality it provides to component developers and detail the technical steps necessary to implement it in a distributed event-based toolkit for specific application domains. We provide empirical evidence for the relevance and utility of our approach by presenting case studies and discussing how the proposed pattern leads to a flexible system structure with reduced integration effort.

Index Terms—coordination, software architecture, design pattern, event-based systems, component-based software, robotics.

1 Introduction

Robot software architecture strives to contain the growing complexity of robot software systems by providing established structures for development, guiding developers towards solutions that don't "just" work, but are robust, maintainable and facilitate assembly from existing building blocks [24]. An important strategy towards this goal is to identify unifying abstractions, that enable treating groups of components similarly. This paper refines and generalizes such an abstraction: Robot tasks as a unifying concept for initiating and monitoring a robot's actions.

Essentially, the robot task concept represents the *state* of doing anything (navigating, manipulating, interacting, etc.) by a general finite state-machine, having states like "requested", "running", "completed", "updated", "aborted", and so on. This may sound straightforward, and it is, but it is markedly different from many robot software implementations, which communicate *only* in task-specific ways (e.g. "effector at pos x"). In contrast, adding generalized states simplifies architectures, and also makes team-based system construction more efficient.

One complex aspect is the asynchronous communication necessary for multi-step processes, which may explain why it has mostly been used in high-level control packages. Fortunately, it can also be realized in an architecture-independent toolkit, like other communications middleware. In the first part of the paper, we describe the benefits of the approach for component developers, assuming such a toolkit. We focus on how adopting the pattern early achieves evolvability, and makes components re-usable with systems small and large.

In the second part, we present details on implementing such toolkits, based on a software design pattern we have generalized and refined from existing coordination toolkits:

Regular paper – *Manuscript received January 28th, 2011; revised October 4th, 2011.*

- *Authors retain copyright to their papers and grant JOSER unlimited rights to publish the paper electronically and in hard copy. Use of the article is permitted as long as the author(s) and the journal are properly acknowledged.*

www.joser.org - © 2011 by I. Lütkebohle, R. Philippsen, V. Pradeep, E. Marder-Eppstein, S. Wachsmuth

The task-state pattern. This helps potential toolkit developers by providing a proven recipe for implementation.

Finally, we have evaluated the architectural and teamwork related benefits of applying the approach on a number of case-studies following the empirical software engineering approach [23, 17, 40], to be presented in the third part of the paper.

1.1 Motivation

As a short example, consider an autonomous robot that assists a human by handing over a cup from the table. This requires a fair bit of manipulation (perceiving the goal, moving a manipulator, grasping) and Human-Robot-Interaction (specifying the goal, maybe aborting the action, reporting on robot activity). Many diverse ways for creating and executing an action plan for such tasks can be found in the literature (cf. section 2), but most, if not all, of them involve combining a number of more basic activities.

What we're interested in here is a closer look at the complexity of such a combination, particularly the communication necessary. Specifically, using basic activities requires deciding when (and if) to start something, whether progress is being made, and determining if it is successful or failed.

This requires knowledge about what is being done, and often also how. For example, whether the cup has been grasped depends on stable contact of the effector (what), which could be determined by haptics (how). For arm motion, success may be tracked by measuring joint angles (different how).

This diversity is a fact of (robotics) life, but it becomes a problem when such information is needed throughout the system, e.g. to provide feedback on the robot's actions, handle a variety of errors in a general manner, or, in general, for all high-level control. In particular, system integration and teamwork are both less efficient in the face of growing diversity.

Therefore, our *primary goal* is to provide detailed information about ongoing activities without requiring detailed knowledge of component implementation throughout the system. In other words, we are interested in a common abstraction that achieves polymorphism: A shared interface that handles functionally different components in a unified manner. This interface is the task-state pattern.

A unified interface improves decoupling, because it allows applications to be developed against the common parts only. However, a potential problem is that it can reduce flexibility, because components must not deviate from the proscribed procedure. Therefore, *secondary goals* are, firstly, separation of concerns, to enable combining task-specific and task-independent aspects cleanly, and secondly, evolvability, in particular to allow changing the common task-state machine in a defined and maintainable manner.

1.2 Core Concepts and Terminology

For brevity, this section omits references which can be found in section 2.

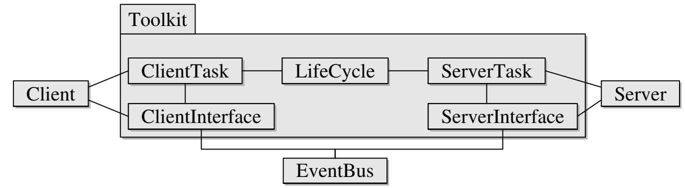


Fig. 1: Components involved in a task.

A software pattern is a reusable recipe for solving a specific recurring software engineering problem [19, 11]. The **Task State Pattern (TSP)** addresses coordination for robotics in distributed event-based systems (DEBS) and is comprised of three elements: a **terminology** which unifies a large body of related work, a **coordination model** which gets specialized to application domains, and an **implementation recipe** for domain-specific TSP toolkits.

Coordination is the act of regulating diverse elements such that the overall operation results in useful behavior. For robot software, the elements being regulated are **components**, e.g. for perception, motor control, planning, etc. Technically, coordination connects the control flow of related components with each other and with the system.

In the TSP, a **task** is defined by a **goal**, which gets specified by a **client** component and then sent to the **server** component which contains the implementation for achieving it. A task **toolkit** is a software framework for implementing such servers and clients, and also for system integration. Its two main ingredients are a **life-cycle** and a **communication service**. The life-cycle is a finite state machine that defines the stages a task can go through. The communication service hides middleware integration from component developers, with utilities for specifying goals, querying or updating the life-cycle, and managing event notifications.

The toolkit has two user-visible facets, one for the client and one for the server. Each facet has a functional interface and a **task object** which encapsulates the specification and state of a given task.

1.3 The Task State Pattern in a Nutshell

In order to give an overview of the TSP and how it addresses coordination of complex robot software systems, we present how a user, e.g. a component developer or system integrator, would typically work with it. For the sake of brevity, we have omitted justifications and references, and we assume that we can rely on an existing toolkit. These details will be given in later sections.

Figure 1 gives bird's eye overview of the structural relationships between the server, client, and toolkit. The life-cycle model is part of the toolkit, so here we treat it as given. Figures 2 and 10 provide life-cycle examples.

When using the TSP to add new functionality to a system, the first step is to **define task-specific structures** that will

be exchanged between the server and its clients: namely the **goal**, the **intermediate result** (if supported by the life-cycle), and the **final result**. Note that the toolkit will attach these structures to generic life-cycle data, but neither the client nor the server need to be concerned with this. Some or all of the specific structures may be empty, but typically there will be at least something in the goal definition, and most likely some form of final result.

At runtime, the order of operations is determined by specific circumstances in the client and the server. Here, we sketch a typical case to clarify the interplay.

To start a new task, **the client instantiates a goal** which is then **sent to the server** via the toolkit's client interface, resulting in a **new client task object**. Instead of blocking, **the client monitors task progress** by inspecting the life-cycle and any intermediate or final results as they arrive: the toolkit updates the task object accordingly and notifies the client via callback methods. Through the task object, the client can also update or cancel a running task.

After advertising its capabilities, **the server asynchronously listens** for goal requests using a callback mechanism provided by the server interface of the toolkit. When a goal is received, a **new server task object** is created and the server can **accept or reject** it. Either way, the server does not block. While working on the task, the task object is used to **send state changes** and intermediate results. Goal updates or cancel requests from the client will arrive via callbacks. The server sends a **final result** when the goal is achieved.

There is room for variation between systems that rely on the TSP. For instance, intermediate results are optional. Also, this account is limited to a strictly pairwise coordination example, but the TSP is designed to seamlessly support more flexible schemes as well. These and other details will be presented in sections 3, 4, and 7.

2 Foundations

The Task-State Pattern straddles areas of many (sometimes competing) architectural proposals, both regarding classical robot (control) architectures, as well as regarding distributed system organization. We will now shortly introduce the architectural foundations relating to our work, to provide a clear view of where we stand, what the pattern covers, and how it connects to systems.

This section is complemented by one that discusses the immediately related work of coordination interfaces and their history (cf. section 5). We defer that discussion until after the introduction of the pattern, however, to present it in context.

2.1 Coordination and Separation of Concerns

Due to the mentioned intersection with many proposals, we emphasize the separation of concerns, to clearly demarcate our contribution. In this, we follow a distinction imported from the software engineering literature (cf. [25, 34]).

The separation of concerns follows the well-known strategy to reduce system complexity by breaking up the problem into parts that can be solved individually. On the architecture level, a good summary is given by [39], who define coordination, as well as its complements communication, computation, and configuration as follows:

- Communication describes *how* something is communicated, for example, whether it occurs through RPC or message passing.
- Computation realizes the behavior and thus *what* is communicated.
- Configuration defines the interaction structure, i.e. who is communicating with whom.
- Coordination defines patterns of interaction, or *when* something is communicated.

We have already used a term for one of the concerns – "coordination" – frequently, in its everyday meaning. For roboticists, this is a relatively new choice – previously, the term "high-level control" has often been used to mean the same thing. However, that term was found to be ambiguous, and work in software engineering for robotics has proposed to use the general software engineering terminology, as outlined above, instead [9].

We follow this proposal and contribute to its realization by providing an interface that separates details related to computation (e.g. task-specific goal information) from those for coordination (e.g. start and completion information).

2.2 The "task" Term

The task concept is commonly found in work on coordination, such as [44]. Usually, coordination is linked (at least implicitly) to a particular architectural approach. As a term, tasks feature very prominently e.g. in the "executive[1]" of most 3-Layer architectures [20], and it can also be applied to behavior-oriented and hybrid architectures [8, 30, 15] (some shun the term while employing the concept).

The TSP makes it easier to experiment with different approaches, by focussing on the commonalities needed to enable them, instead of on their differences.

2.3 Coordination in Robotics

Robotic systems typically use coordination for sequencing actions, managing shared resources, task selection, and the like. Originally, two opposing styles have been proposed. The much earlier *sense-think-act* style continuously repeats a strict sequence of perception, planning, and acting: planning determines the action sequence and takes resource constraints into account, thus covering coordination. In the *behavioral* style, action components are concurrently running and reacting to inputs. Here, coordination is split into two phases: behaviors

1. Somewhat counter-intuitive, in 3-layer architectures, the "executive" does not perform actions itself, but organizes their execution.

become active when certain conditions hold, and any occurring conflicts are resolved with arbitration methods such as priority schemes (subsumption architecture [8]) or by combining output requests (motor schemas [6]).

The relative merit of these styles has been the subject of intense discussion (see Kortenkamp and Simmons [24], section 8.2); meanwhile, the respective advantages of behaviors (fast reaction to local stimuli) and planning (optimizing longer-term activities) led to combined approaches. The ATLANTIS [20] architecture with its three hierarchical layers (controller, sequencer, deliberator) was typical for its time, with a rather behavioral style at the bottom and behavior-selection and coordination on top.

The authors of the CLARAty robotic software framework [49] point to considerable remaining debate about the assignment of functionality between planning and execution, the two upper layers of typical 3-Layer system models; they also note that such models obscure internal hierarchies and sub-systems.

We think that despite much research, a fundamental question remains: how to construct systems in a way that facilitates reuse and reconfigurability, and thus leads to robust extensible systems? Our stance is to leave aside such unresolved matters and consider that robots need a number of functionalities which form feedback loops with the environment. These loops exist at (and span) various levels of abstraction, and we need a way to coordinate them. We have thus formulated the Task State Pattern as a coordination approach which is agnostic of architectural concerns.

2.4 Distributed Event-Based Systems

Component-oriented software engineering is becoming accepted in robotics [24, 10, 9, 47, 48, 33, 31]. Its focus is mainly on reuse and flexibility, primarily through modularization. Clear interface specifications are a key aspect for achieving this, based on the fundamental tenet that only what is hidden can be changed without risk [36, 35].

However, the communication method chosen can make change harder. For instance, remote-procedure call (RPC) communication specifies the target of the communication at the originator, and not just in the abstract, but usually by giving a specific name of server and method. Furthermore, RPC communication is one-to-one, which does not allow other components to monitor it easily.

In contrast, distributed event-based systems (DEBS) [32] particularly emphasize decoupling between concurrently running components: events are asynchronous and have no explicit target; instead, they are (logically) broadcast with metadata to allow filtering. Thus, a potential server component can chose the messages it can handle, and other components can monitor them, too. This facilitates replacement of components, and loosely coupled coordination.

DEBS can be realized both through publish-subscribe [16], and through a shared (usually hierarchical) event-bus. The choice determines some of the run-time characteristics, but it does not affect the basic capabilities that we are using here. In the descriptions in the following we usually specify event-bus communication, but the pattern has also been used (including by the authors) in publish/subscribe systems, and even, albeit with some loss of generality, in RPC-based communication.

3 Using Task-States in Components

While the core of how to implement the pattern is relevant mostly for toolkit developers (cf. section 4), we will shortly introduce its use by component developers. Besides guidance for these developers, we hope that this will also make them aware of the advantages of using the pattern, and ask for it in frameworks that do not offer it yet[2].

3.1 An example life-cycle

Component developers should be aware of the task life-cycle used in their system, so that they know which states and transitions are possible. A simple life-cycle is shown in figure 2. It has only two real states: An "initiated" state, which is held when the client requested a task, but the server has not yet acknowledged, and a "running" state, when the server is executing the task. The end-states are also relevant, one for successful ("done"), and one for unsuccessful execution ("cancelled"). Note that unsuccessful execution can also be caused by the server rejecting task execution, e.g. when it is busy with something else.

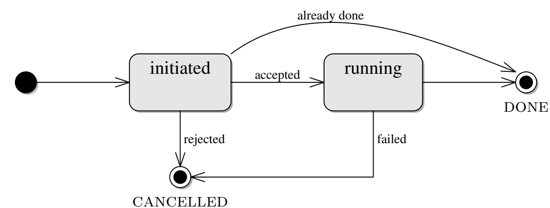


Fig. 2: A basic life-cycle example. The two terminal states could be fused, but then users would be required to keep track of the event which lead to it.

There are essentially two ways for interacting with the life-cycle: Inspecting the task-object and registering for event notifications. The latter is most appropriate if the delay between a state-change and the reaction to it should be short. We expect that this will be usually true for servers.

3.2 Communication

So far, in component-based robotics, client and servers exchange task-specific messages that encode the goal to be performed, and the current state of actions. To this, using

2. Currently available open-source toolkits are *i)* ROS Actionlib, http://www.ros.org/wiki/actionlib, and *ii)* the XCF Task Toolkit, http://opensource.cit-ec.de/projects/xtt

the task-state pattern adds task-independent state-information. Managing this information is usually the responsibility of a dedicated task toolkit. This toolkit takes the place of the communication layer (cf. section 4.1).

Interaction between client and server then entails in the client sending goals together with a task-independent transition event (e.g. "initiate"). The server responds with a decision on execution (e.g. "accepted" or "rejected"). Both client and server may update the task-specific representation at the same time.

Because client and server are running independently of each other, situations may occur where both of them attempt to change the task at the same time. It is the toolkit's responsibility to resolve such cases (cf. section 4.5.1).

3.3 Using the Pattern as a Client

When the result of the task should be used immediately, the procedure is roughly as follows: i) Submit the task to the client interface, which returns a task object. ii) wait either for finish, or for the next transition. iii) check whether the the task has been successful, and react accordingly.

Compared to message-passing, this procedure will provide explicit feedback on whether the task request has been executed or not. Compared to remote-procedure calls, it can provide explicit feedback both at the beginning and at the end of task execution, by waiting for the next transition in step 2, instead of just for completion.

Furthermore, more detailed life-cycles, such as the one in figure 10 (page 28), can provide information about, and interaction with tasks in more versatile ways, while keeping the same coherent interface. Updating goals and canceling tasks are two often-used functions a more general life-cycle can provide in a consistent manner.

Last, but not least, instead of waiting, clients may also ask to be notified of state-changes whenever they occur. This facilitates managing several tasks at the same time, and can also be convenient when only some transitions necessitate a reaction (e.g. failures).

3.4 Using the Pattern as a Server

Creating a server for using task-state based reporting is fairly straightforward: Primarily, a server must map from its internal view of the task's progress to the abstract state-machine, and report the transitions. Depending on the life-cycle, this may either be together with its usual messages, or in addition to them. For example, when it starts acting upon a task, the server should publish an appropriate event (in the example life-cycle, this would be "accepted"). The same applies when the task is completed.

When advanced life-cycles are used, the server implementors face more work, e.g. to support updating and cancellation. Fortunately, because of the general state-machine, the toolkits can support default implementations (rejecting the requests, or using cancel-then-restart for updates). In this way, server implementors can start using a basic implementation, then transparently add more capabilities.

4 Implementing the Pattern in a Toolkit

Having discussed what the pattern is good for, and how it can be used, this section provides guidelines for implementation. This is of interest for toolkit developers, and also provides a general overview about the boundary between the toolkit and other components.

Patterns are rarely implemented in exactly the same way, so we will first describe the most typical implementation of the pattern, and then discuss alternatives. Furthermore, common pitfalls and appropriate remedies will be discussed.

4.1 Premises and Design Influences

We consider implementations for distributed systems, assume that task execution is fundamentally asynchronous, that multiple tasks may occur in parallel, and that each of them can have multiple state changes and/or results. Therefore, we assume that communication is using *asynchronous event notifications*.

The suggested implementation of the pattern is as a *toolkit layer*, i.e. a library for re-use in all components of a system. Apart from re-use, we advocate this to ensure safety and robustness: Our experience is that communication between asynchronously running components has many corner cases which are often not covered until testing reveals them. A toolkit can combine experience from many systems and cases, to provide a safe and robust behavior. Using it in a layered structure, as shown in figure 3, prevents the user's from bypassing it.

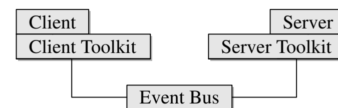


Fig. 3: General structure: Distributed with toolkit layers

The second most important design goal is a gradual adoption curve for developers. Developers generally prefer the simplest solution and while the pattern's life-cycle provides evolvability of the overall system, it may, at first, appear more complex than needed. Therefore, the toolkit should afford developers the choice to have their components support only a minimal subset, with opportunities for gradual growth.

4.2 Summary of implementation steps

As a guide to the remainder of this section, we first provide a short summary which can also serve as check-list.

4.2.1 *Specify the* life-cycle model

The life-cycle is a finite-state machine that contains all states and their possible transitions for tracking the task's progress. More details on choosing a good state-machine will be given in subsection 4.6. For implementing it, we recommend using a configurable model, to aid evolving the life-cycle.

In addition to the finite-state-machine, it should be specified which transitions may be invoked by the client, which by the server, and which by both. This facilitates checking allowable transitions by the toolkit (see next step).

4.2.2 *Specify client and server task objects*

The task objects contain the task representation, and the current instance of the life-cycle. See figure 1 and section 4.4.1 for minimal attributes needed.

The task objects should be distinct for client and server, to check that only allowable transitions are invoked.

4.2.3 *Implement a demultiplexer for notifications*

The (de-)multiplexer maps from incoming notifications to the corresponding task object, and transmits outgoing notifications. See section 4.5 for its responsibilities. A typical implementation strategy is the asynchronous completion token pattern (ACT) [41, p.261–284].

4.2.4 *Implement notification for client and server*

Information about the state of ongoing changes should be available as asynchronous notifications. This saves users from having to periodically poll state objects. The Observer pattern is a typical implementation choice [19]. For servers, observers are also used to initiate new tasks. See section 4.4 for the interfaces.

4.2.5 *Implement a client interface*

The client interface is the point where new tasks are submitted by clients (cf. section 4.4). It cooperates with the demultiplexer to send notifications out, and also manages what the client observes.

4.2.6 *Implement an abstract base class for servers*

The server processing has a fairly regular structure and is amenable to support through an abstract base class. Specifically, it can map incoming events to dedicated functions, and provide default implementations for optional parts of the life-cycle. See section 4.4.2 for details.

4.3 Advantages and Responsibilities

The toolkit's function is to communicate, and keep track of, changes in the state of a task, including start and end, between two or more (distributed) components. This is non-trivial because components are running independently, all communication incurs a transmission delay, and thus two component's view of a task may diverge. It is the toolkit's responsibility to handle these cases in a robust manner.

Conventional middleware is not sufficient here, because it has no concept of the relation between messages. In contrast, the task-state-pattern knows the life-cycle and can use it to recover from errors.

A secondary issue is that a single client/server pair may be engaged in more than just one task at the same time. Therefore, incoming notifications must be associated with a task, and the application side notified of changes as they occur.

Finally, a task toolkit should prevent components from attempting state changes that are not valid according to the life-cycle. If such illegal transitions are still received, either because of bugs or due to network errors, the toolkit must recover from them.

4.4 Structure

We distinguish between user-visible and toolkit-internal objects. The user-visible objects are: i) an interface for clients to, firstly, submit new tasks, and, secondly, receive *change events*, which we call the *client service*, ii) a *server interface* to be implemented, iii) an object to represent, and modify, ongoing *client tasks*, and iv) the same for the server side, a *server task* object. The toolkit-internal objects are a) the life-cycle model, and b) a demultiplexer object that associates incoming messages with their tasks. See figure 4 for an overview.

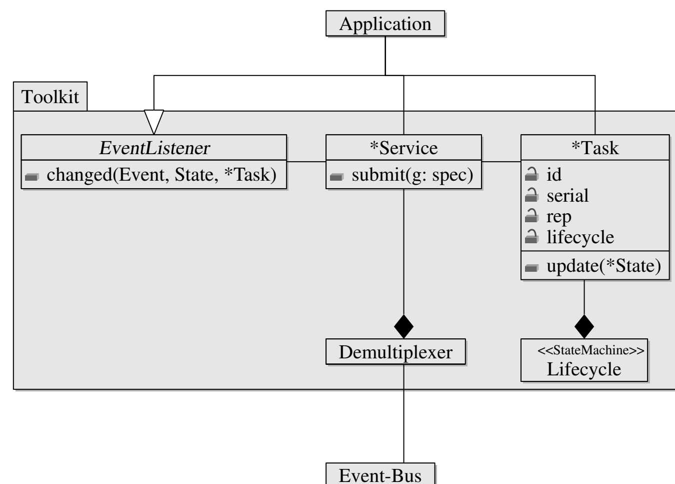


Fig. 4: Minimal toolkit structure for clients. The server side is similar, but combines Service and Listener into an abstract base class.

The life-cycle describes all states a task can have, but client and server differ in how they are allowed to modify it. To prevent errors, the task objects restrict possible changes: The client task object accepts only those modifications that a task client may perform, and rejects the others, and vice versa for the server side (cf. figure 5).

Fig. 5: Partitioned life-cycle access

4.4.1 Task Object and Notification Structure

The attributes of the task object (cf. figure 4) are essential for the added functionality that the toolkit provides, and have to be included in all task-related notifications.

id — An identifyer for the task. It has to be unique at least per server, and ideally globally, to preclude potential confusion.

serial — A number that is incremented on each update and used to detected missed updates. Also note the *originator* field of task messages, described below.

rep — A representation of the task to be performed in detail, including goal, parameters, and the current state (in task-specific terms). This field is specific to the type of the task and is not directly modified by the task toolkit, just updated from notifications.

In addition to these fields, it must be possible to determine which component sent a notification, to check for overlaps (compare section 4.5.1). Usually, this is available from the middleware, but if not, an additional *originator* field must be included in notifications. It specifies whether an update has been sent by the client or by the server.

4.4.2 Designing for Gradual Adoption

When creating a new server implementation, or when moving a simpler server to a more advanced life-cycle, the server implementor has to support additional features. It may be desirable to postpone this effort to a later stage, e.g. for testing, or to rapidly provide basic support before realizing more features. It may also be that a few servers cannot support all parts of the life-cycle, for example, a safety critical task may not allow cancellation.

For such cases, we have found it useful to provide an abstract base class for the server, which provides default functionality for the unsupported cases. See figure 6 for an example abstract server interface.

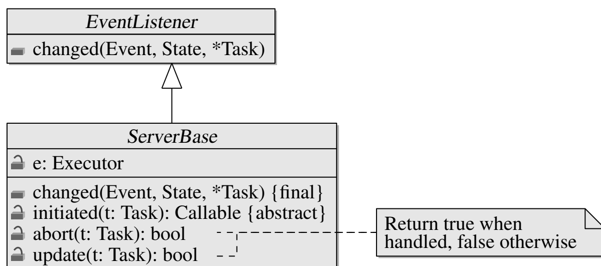


Fig. 6: Methods for example abstract server base class. The method "initiated" has to be implemented, the others provide defaults and can be overriden as necessary.

4.5 Dynamics

The dynamics of the task toolkit are fairly simple, primarily a pipeline, with the toolkit interjecting between client or server and the middleware. On the sending side, the toolkit checks that changes are valid and rejects them if not. The receiving side is slightly more complex, because, firstly, it has to associate notifications with tasks, and secondly, it must detect and handle message overlap.

Figure 7 shows an example sequence for the sending side. It should be noted that this sequence is less fixed than the structure. While the pattern requires a number of objects, their interaction has more flexibility. For example, we show the client service organizing the various other objects, but a different distribution of responsibilities is certainly possible, e.g. with the client task creating the life-cycle. We found that, in our cases, the sequence shown minimizes associations between the objects, but the importance of that design goal is up to the implementors judgment.

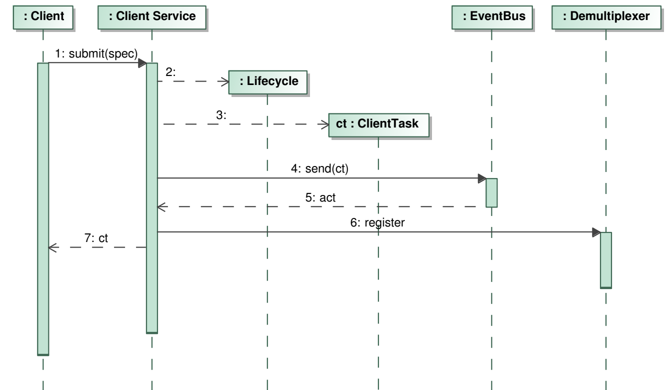


Fig. 7: Invocation sequence for submitting a new task.

When receiving a notifications from the event-bus, the process is essentially reversed, as shown in figure 8. Notifications that are not found by the demultiplexer – which means they have been created by other clients – are ignored.

4.5.1 Detecting and recovering concurrent updates

Not shown so far is the handling of message overlap. Such overlap can occur for several reasons. Firstly, client and server may both attempt to change the state of a task at approximately the same time, but transmission delays cause their changes to overlap. This case is shown in figure 9, where, globally, "update 1" is first and "update 2" is second, but locally (at the end points), the notification for "update 1" may arrive after "update 2" has been sent.

An easy method to detect such cases is the use of a serial number, as specified for the task objects before. Prior to sending a notification, the serial is always incremented by one. Upon receiving a new notification, its serial is compared to the local serial, with the following three possibilities:

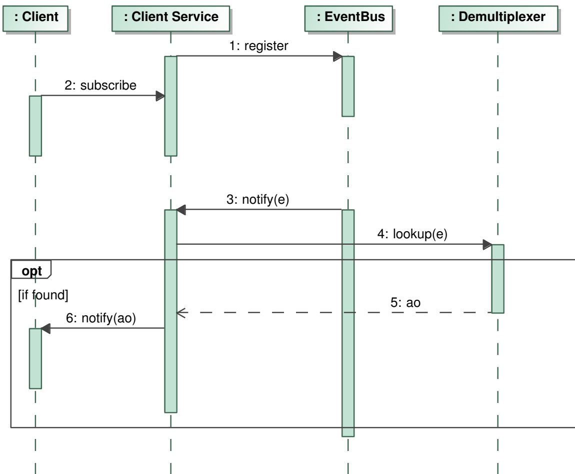


Fig. 8: Invocation sequence for notifying listeners.

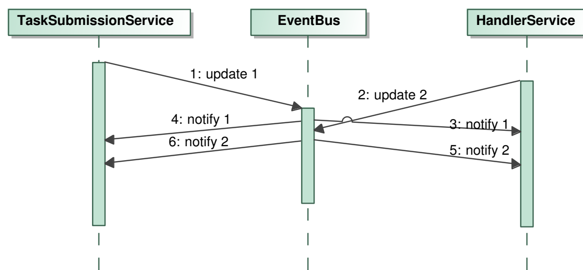


Fig. 9: Example of message overlap due to network delay

1) If the serial is higher than the local one, the notification is a normal update and processing continues.
2) If the serial is equal to the local one, it means that both parties have sent at the same time. This calls for a recovery procedure (see below).
3) If the serial is lower than the local one, it means that the remote component has missed several notifications by the local component. This is usually a serious error and should only occur in case of network problems. Recovery may be attempted, but unless the communication problems are fixed, may not be successful.

Fortunately, case two is recoverable through checking the life-cycle for allowed sequences. A simple means to do so is to use what we call the "server dominant" approach. In this, firstly, the server always determines the current state, because it is the component executing it. Secondly, if the life-cycle allows it, the server applies the client's update after its own and sends an additional notification to that effect. If the life-cycle does not permit the client's update after the server's, it is ignored. The client always accepts the notifications from the server as the correct state, even when the local state would not ordinarily allow the transition.

A drawback of the "server dominant" approach is that clients must be prepared to receive a notification for unexpected target states. This is a common requirement, but if it is undesirable, alternative approaches may be preferred. See section 4.7 for a short discussion.

For case three, we recommend reporting an error. While additional checks may determine that the original problem has ceased and operation could continue, this is not well-placed within the communications infrastructure (which the toolkit is part of). Higher-level mechanisms should be employed to determine the viability of continued operation, and re-start tasks, if possible.

4.6 An Example Life-Cycle

The life-cycle directly determines the distinctions that can be made when tracking tasks. Because having a start and an end is central to the task concept, and execution may always fail, we consider the basic life-cycle introduced earlier to be the core of what all life-cycles must support (cf. figure 2). However, we expect that most users will want more features.

The necessary distinctions are dependent on the system the toolkit is designed for, but here we will present a life-cycle that has served us well, and that we consider to be a good candidate for most systems. See figure 10 for a visualization. In addition to the basics, it also supports updating the goal during execution, aborting tasks, and delivering intermediate results. Its states and transitions have the following meaning:

- INITIATED Initial state for newly published tasks.
  - ACCEPTED Execution is commencing.
  - REJECTED No action will be taken.
- RUNNING Execution is ongoing.
  - UPDATE Goal has changed
  - RESULT_AVAILABLE Intermediate result added
  - COMPLETED Goal reached
  - CANCEL Execution stop requested
  - FAILED Goal could not be reached
- UPDATE REQUESTED New goal available
  - ACCEPT Target is new goal.
  - REJECT Previous goal will remain target.
- CANCEL REQUESTED Aborting execution is requested
  - CANCEL FAILED Aborting not possible.
  - ABORTED Execution aborted.
- CANCELLED Execution stopped without reaching goal.
- DONE Goal reached, execution stopped.

One design choice has been made in the general life-cycle model: it returns to the running state when an update fails. That means it will continue to act on its previous target. It would also be conceivable to attempt an abort in such a case and the model might need to be enhanced to encompass this. The argument for the current choice is that the client is generally best positioned to decide whether the failure is severe enough to warrant aborting the action. For a further discussion of the trade-offs involved in life-cycle design, see section 7.2.

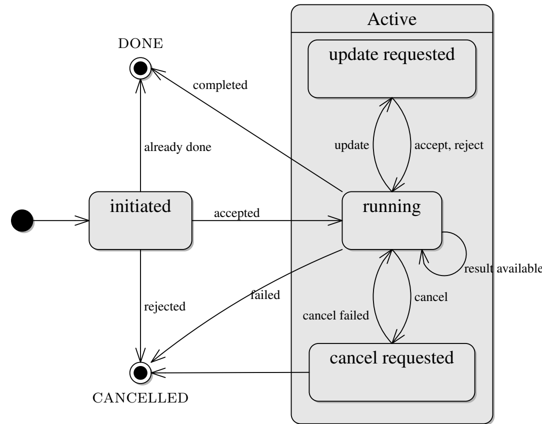


Fig. 10: A fairly general life-cycle. Note that we show "cancelled" and "done" as two states purely for convenience in determining the cause of termination. A general "terminated" meta-state is assumed to determine inactivity. Also note that the life-cycle is a user-defined extension point, so other choices are possible.

4.7 Alternatives

The implementation guidelines described so far attempt to strike a balance between simplicity and efficiency. However, other implementational choices are certainly possible and this section reports some different choices we have seen so far.

4.7.1 Representing the task

Figure 4 has specified a single "rep" attribute that contains the entire current state of the task, both goal specification and results. The tacit assumption here is that the entire representation is communicated every time the state changes. Depending on how large the representation becomes, and how often it changes, this may cause communication overhead. Furthermore, it requires that the representation is modified during operation. Again depending on the complexity of doing so, this may be inefficient.

An alternative approach is to *communicate and store only updates*. This is appropriate when the representation is updated much more often than it is read. The drawback of this choice is that it makes reconstructing the current state more difficult, and it also makes observation harder, because an observer has to follow all messages to reconstruct the state.

4.7.2 Conflict detection and resolution

As mentioned in section 4.5.1, the "server dominant" model for overlap resolution requires that client must be able to accept whatever state the server reports, even if that state would be unreachable from the last state they have requested. This is not normally an issue, but may be considered confusing.

Furthermore, the "server dominant" model treats all overlaps in the same manner. Users may want to have more say in treating such cases, to make case-by-case decisions. While this somewhat contradicts the value of the pattern to generalize handling, we would not want to rule out the necessity entirely.

Therefore, we'll shortly present two alternatives to conflict resolution that we have observed in implementations.

- *Central broker.* Instead of communicating directly between client and server, all communication may be enforced to pass through a central broker that resolves conflicts. From a communication perspective this introduces a single point of failure. Furthermore, a central conflict resolution strategy may be hard to specify. Nevertheless, such an approach can provide predictability.
- *Delayed Transition Evaluation.* all transitions are executed with a fixed delay that is larger than the maximum possible communication delay. This strategy may be used when real-time transports are available that guarantee a known maximum transmission time. This is not recommended for best-effort communication protocols, since they could always exceed the expected maximum.

5 History & Known Uses

Historically, while state-machines have been used for coordination in many systems, including robotics, at first these were not abstract but action-specific. For example, the Intelligent Machine Architecture [26] uses distinct state names for each activity (also cf. 5.1). While to a human the chosen state-names already suggest a commonality, this is not made explicit.

A similar situation is present in automation, where IEC 61499 [2] (the standard for distributed systems in this area), specifies "execution control charts" (ECC) which, while not abstract, are often used in this way (cf. [22, chapter 9]). This applies to systems built using IEC 61131-3 [3, 1].

The utility of abstraction then seems to have first emerged for error-handling, a common cross-cutting concern. Again, [26] already alludes to this, and a particularly direct example is the progression from Simmons' TCA [44], which is task-specific, to his TDL [43], which adds an abstract life-cycle, and emphasizes its utility for handling errors in a loosely coupled manner. This is also one of the approaches known to us (the other being Hanheide et al. [21]) where an earlier approach for fine-grained coordination was succeeded by an abstract life-cycle-based interface later on.

At the same time abstract state-machines have already been used for another area: Tracking the overall state of a component, such as whether it is initialized correctly, available to receive requests, and so on. Examples of such component-level state-machines include OROCOS [46, figure 2.5] and the OMG's RT-component model, as used by OpenRTM [5, figure 6]. While, arguably, the global component run-state is more limited and predictable than task-execution states, this also suggests the utility of abstraction once systems contain more than one instance of a particular type.

To trace the outlined emergence of the pattern for abstract task-state machines in more detail, we first look at the literature and analyze the approaches known to us. Section 6 complements this analysis with empirical data to judge suitability.

5.1 Intelligent Machine Architecture (IMA)

The IMA [26] coordinates low-level tasks, such as blob detection and robot limb motion, using Petri-Nets. The states in these Petri-Nets are action-specific, but have obvious life-cycle relations. For example, the blob-detection action has states START_FIND_SPOT and END_FIND_SPOT. Similarly, the visual controller has states VC_READY, VC_START, VC_DONE and so on.

The Petri-Net structure of this approach is also an example of a hierarchical model. The higher levels (such as the visual controller) contain conflict-resolution states (READY, AVAILABLE, etc.), whereas the lower levels do not have such states and just use start and end states. Error handling is similarly organized in a 3-level hierarchy:

> *Errors are handled first by the Coordinator, and are passed up to the Dispatcher only when a local strategy is not adequate to resolve the condition. In some instances the Dispatcher must turn over error resolution to the Organization Level where operations may be replanned [26, section 5, paragraph 4].*

This architecture, however, does not exploit the similarities in these models and thus its coordination mechanism is not action-independent. Additionally, the coordination hierarchy, while explicitly designed, is not modeled but implicitly contained in the Petri-Net structure.

Information on the implementation of the IMA is scarce, but it is mentioned that message-passing is used for communication, which has very similar semantics to an event-bus. In particular, it also uses asynchronous communication. The main difference is that messages are addressed to a known recipient, instead of being selected from the event-cloud by the handler, as in the Task-State pattern.

5.2 Task Control Architecture (TCA)

Similarly to the IMA, the TCA [44] uses fine-grained models encoded as so-called task-trees. This architecture does not yet exhibit abstract states, but already utilizes an external execution monitor:

> *In particular, monitors and exception handlers can be added without modifying existing components. [...] a module can associate an exception handler with a class of messages handled by another module, so that whenever a message is issued and added to the task tree, the exception handler is automatically added to that node [44, section IV, paragraph 2].*

Notably, the above quote mentions classes of messages, which suggests a relatively close dependence of exception handlers on the monitored tasks.

TCA uses the IPC messaging middleware [45], which supports both publish/subscribe as well as client-server models. It is not specified which of the two is used in the TCA.

5.3 Task Description Language (TDL)

The TDL [43] is the successor of the TCA and one of the earliest examples of abstract life-cycle model based coordination. In this architecture, all tasks are coordinated using the abstract states DISABLED, ENABLED, ACTIVE and COMPLETED. It also introduces the concept of an expansion, which describes the state of an entire task sub-tree using the same set of states.

The TDL keeps the approach of an external exception handler, but also introduces abstract states called SUCCEEDED and FAILED to attach it to tasks, thus again moving towards an abstract life-cycle model.

TDL is a language compiled to C++ code. The resulting code uses the the Task Control Management (TCM) library as the service-level API. Like TCA before it, TCM can use the IPC message-passing middleware. Additionally, it supports Real-Time Inc's DDS publish/subscribe middleware, which suggests that the TCM itself is also based on a publish/subscribe model.

5.4 LAAS Architecture and GenoM Framework

The *Generator of Modules* GenoM [18] is a tool for specification and integration of distributed functional modules and a centralized decisional layer in a hybrid robot architecture [13]. A centrally coordinated generic life cycle model allows module-independent monitoring. Two types of events are supported: control events come from outside the module and are handled by a module's control level, whereas execution events stem from its execution level.

The control level implementation and the structure for the execution level are generated from a formal C-like module definition. User implementations of the execution level are contained in non-interruptible *codels* which essentially serve the role of hook-based life-cycle event dispatching. Concurrency is guaranteed by basing execution events on codel return values.

A module's activities are parameterized and activated through asynchronous requests implemented via named ring buffers. An intermediate reply is sent upon entering the EXEC state, and a final execution report is sent upon termination. Intermediate feedback is read from posters (named shared memory structures) generated from the specifications. Runtime-configurable intra-module data flow is also (mostly) based on posters.

GenoM provides a tool for incremental development of formally verifiable modules that can easily leverage real-time mechanisms. Preemptability of activities is a fundamental part

of the framework: module specifications can state which activities are incompatible with each other and, with a little help from user-implemented interrupt hooks, the auto-generated control layer will properly interrupt entire activity trees based on these conflict specifications.

5.5 PLEXIL

PLEXIL is not a coordination system as such, but a proposal for a standard interchange format, based on the concept of a tree containing processing nodes. It is included in this discussion because coordination systems are needed to execute the specified plan, and because it contain a common, abstract life-cycle for all nodes in the plan tree. There is also a reference implementation available (called "universal executive"). While not specified as such, a centralized coordination component is implied.

The life-cycle itself has many states similar to ones seen so far. Furthermore, several additional transitions reference changes to *other* nodes in the tree, for example `Ancestor_end_condition` or `All_children_waiting_or_finished`. These are essentially events on the coordination, not the task level, but there are provisions for automatically propagating these coordination-level events to changes in a node's state.

In relation to the pattern, while it would not be incompatible, PLEXIL follows the model where mapping from task-specific states to abstract states occurs in a central component, not in the servers. To this end, it comes with much more flexible condition checking facilities.

5.6 DESIRE Architecture

A non-event based implementation of the pattern is exemplified by the DESIRE service robotics architecture [38]. Its life-cycle model consist of RUNNING, FINISHED, COMMANDABORT and COMMANDFATALERROR states, with the last state signifying an unrecoverable error. Errors are further distinguished as ABORT, CONTRACTVIOLATION (indicating an erronous task specification) and FATALERROR (indicating an internal error).

While the DESIRE life-cycle model does contain a state indicating an unacceptable action specification, it is still primarily a model that can only support a centrally coordinated system, because no request states are modeled. Correspondingly, the DESIRE system includes a planning component, which is integrated at the service-level.

Instead of event-based communication, the DESIRE architecture uses the RPC interaction style of a CORBA-based object-oriented middleware. Communication between requesting and handling components is mediated using an execution control component, which realizes the service-level and allows asynchronous communication between submitter and handler.

5.7 Active Memory Architecture

The Active Memory Architecture is a coordination approach built on an active database coupled with an event-bus. Coordination first used Petri-Nets, which later got replaced by a finite-state model. The architecture was originally developed for a cognitive vision system [51, 50] and later transferred to human-robot-interaction [21].

The earlier, Petri-Net based coordination realized a dedicated fine-grained execution monitor based on temporal monitoring of the actions, managing recovery actions if errors were detected. It did not explicitly model the action life-cycle, because actions were continuously repeated as part of the vision-processing cycle anyway.

The later, finite-state approach added an explicit request-acknowledgement life-cycle for action monitoring. Actions in this context were fairly coarse-grained, e.g. "acquire a map of the room". Depending on the nature of the sensor used, these larger activities could either map to a single action (e.g. analyzing a single view from a 360°degree camera) or multiple sub-actions (e.g. activation of a 180°laser scanner plus rotation of the platform to acquire a 360°view). To accommodate both types of handlers without changes to the clients, a model with several optional states was used, similarly to the one in figure 10. In this approach, coordination was decentralized and co-located with the participating components.

5.8 XCF Task Toolkit (XTT)

The XTT is a service-level coordination implementation in Java, initially based on the life-cycle-model of the active memory architecture approach. It was developed in order to experiment with the effect of changes at the level of model, interface, and implementation [27].

XTT uses a layered architecture which embeds the service-level into participating components, but hides the middleware communication from the application-level and also enforces adherence to the life-cycle model. Conflict detection uses serial numbers and recovery employs the dominant handler approach. On the API level, XTT is somewhat unique in that it offers both the common Observer interfaces, as well as synchronous adapters, with a submitter-side Future[3] and a handler-side Callable[4] interface.

As the name suggests, XTT is built on the event-based middleware XCF, in particular, the Active Memory-based event-bus [51]. It hides the communication mechanism, but exposes the data-transport classes of that toolkit, in conjunction with an application-level conversion registry.

5.9 ROS actionlib

Actionlib [29] is a C++ toolkit for action execution based on ROS, an open-source meta-operating system for robots,

3. A Future represents the result of an asynchronous computation.
4. A Callable is a task that returns a result and may throw an exception.

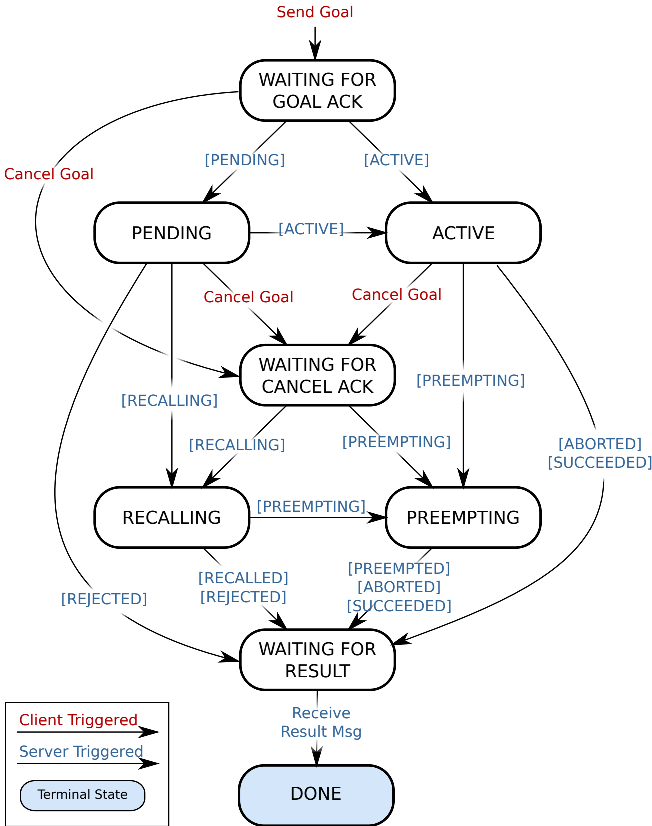


Fig. 11: ROS actionlib client state machine

developed as a joint effort between Willow Garage and the robotics community.

Actionlib comprises a service-level toolkit for task state coordination and, most notably, it maintains two separate state machines: one for the server (handler) and one for the client (submitter). This provides a consistent and elegant implementation of the dominant-handler strategy for recovering from state-change conflicts, albeit with somewhat different terminology, because the states describe the combination of component and task-state. Apart from that, it is equivalent in features to the general life-cycle shown (cf. figure 10).

In addition, the state-machine has to cope with the fact that underlying ROS middleware is not entirely reliable – in case of buffer-overruns, notifications may be dropped. Therefore, it is is more permissive than the state-machines presented before. The result is depicted in figure 11.

Other features of actionlib include id-based and temporal references. The latter makes it easy to cancel e.g. all ongoing actions that have been started before a given time. The action server supports both callback- and queue-based invocation.

One historical fact about the actionlib is that it has originally been intended as the interface layer between a general planner and action components. As such, some of its terminology (e.g. "recall" and "pre-empt") is influenced by that of the planner. The actionlib authors acknowledge that the client state machine terminology is not optimal, but the structure is sound. The client states were chosen to facilitate tracking the server's states, as opposed to being designed to provide feedback about the task's progress. Thus, in practice, many actionlib users use a thin wrapper around the action client (provided by the toolkit and called "simple" action client) that provides the desired feedback about task progress.

6 Case Studies and Results

While the first forms of the task-state pattern can be traced back at least 30 years (cf. section 5), empirical studies of its application have been restricted to building proof-of-concept systems. Furthermore, these have been only described on the functional level, that is, what they are capable of doing, not on the architecture, i.e. how (or how well) they are internally structured. Since then, the growing complexity of robotic software systems has made architecture more prominent.

Therefore, in this section, we will present a number of case studies with a particular focus on the architectural qualities of the resulting systems. As outlined in the introduction, we believe that the pattern *(i)* increases the reusability of components, *(ii)* aids long-term maintenance by reducing the coupling and dependencies between implementations, and *(iii)* a toolkit-based implementation results in simpler components.

Moreover, some of the case studies are taken from an iteratively developed system, created collaboratively by experts from various domains. By comparing the changes between iterations, we can observe *(iv)* the pattern's effects on team productivity.

6.1 Quantifying Savings from Toolkit Use

It is generally believed that increased use of existing libraries can improve re-use and reduce both (novel) code-size and complexity of components. However, realizing these potential savings requires not only a good application programming interface, but also that the functionality the library offers is indeed separable from the core functionality of the component. To establish that the task-state pattern can be separated from core component functionality in this way, we have first measured implementation effort for ad-hoc implementations, and then compared that to the effort required for using a toolkit.

6.1.1 Estimation of Direct Implementation Effort

Our first data comes from a service robotics context, where several components are coordinated using the task-state pattern. In the so-called "Cogniron Home Tour scenario" [7, 21], a human tutor shows a mobile robot around their apartment. The robot follows the human's path and acquires sensory signatures for the locations pointed out and labeled by the user. Human-Robot-Interaction (HRI) here is multi-modal, with speech the primary modality. In the scenario, there is close interaction between the navigation/following components and

the speech dialog system. For example, when the tutor asks the robot to follow, receipt of the command and ability to move are confirmed, based on state-changes in the "following" task. Even more importantly, a failure on the follow task (e.g. when losing track of the human, or when there is navigation trouble) will also be communicated verbally.

This system uses the task-state pattern, but every component has its own implementation of it. This led to inconsistencies in implementation, for example, not all of them support all parts of the life-cycle. In itself, this would not be an issue, but unsupported transactions were not refused, but silently ignored. This alone would have suggested that a unifying implementation is useful.

To get an idea of the implementation effort for task-management, we examined the code of the "following" component. The component itself consists of 1593 lines of C++ code[5], of which 241 lines, or 15.1%, have been dedicated to state management. This is already a fairly high percentage for something which can be classified as mostly communication, but could also be caused by the fact that it is only a small to medium-sized component.

More tellingly, however, the class responsible for task management has been involved in 29% of all changes made to the component, *even though the life-cycle did not change*. Instead, these changes had been necessary due to coupling with code that manipulates goal specifications, which is related but functionally different, and should thus be decoupled in order to minimize development effort and the risk of regressions.

6.1.2 Simplification relative to RPC

The previous data suggests that a toolkit can aid pattern implementation. However, it could also be asked how such a toolkit compares to communication using remote-procedure call (RPC) interfaces, for which very good middleware already exists. We examined this on another component from the previously mentioned "Home-Tour" system, which had so far been directly using an RPC interface from the underlying middleware (XCF [51]): Text-to-Speech Synthesis (TTS).

The TTS component is responsible for taking a text string, synthesizing it into audio (based on the the *Mary* TTS library [42]), and playing it back. The RPC interface was very simple, which provided a functional motivation for using the task-state pattern. Firstly, when multiple requests are received, they are played back in order – aborting was not supported. Secondly, the original RPC call used to invoke it either returns immediately, or after playback has completed, with no provisions for intermediate feedback (such as reporting syllable onset times) during playback.

For our case study, a task toolkit implementation has been provided by one of the authors, and a student assistant has been tasked with adding it to the TTS component, while removing the original RPC interface. To prevent an implementation bias

5. Using David A. Wheeler's SLOC-count

metric	before	after
total N° classes	14	10
N° referencing classes	5	5
total N° methods	71	76
N° referencing methods	12	14
N° referenced classes	9	8
N° referenced methods	11	8

TABLE 1: Dependencies on the middleware (before) and the task toolkit (after) in the TTS component.

class name	WMC	CBO	RFC	LCOM	Ca	NPM
MaryXCF	19	21	80	**101**	4	6
MaryConnector	15	13	60	**5**	3	9
MaryTS	7	12	26	**17**	1	3

TABLE 2: Object oriented design metrics before and after the use of the task-management library. `MaryXCF` is the original component, renamed to `MaryConnector` afterwards, and `MaryTS` is its task-based implementation.

towards simplicity, the student was told only the functional reasons for this addition, not the architectural ones.

We then compared the number of dependencies of the component to the RPC middleware (for the original implementation), respectively the task toolkit (for the new one). The results are summarized in table 1, and we can see that no substantial difference exists. If anything, the toolkit implementation causes fewer dependencies.

To substantiate this further, we computed the Chidamber-Kemerer Java Metrics (CKJM) [14], which measure source code complexity. See table 2 (WMC measures methods per class weighted by their complexity; CBO measures coupling between objects (methods of one class use e.g. methods or instance variables of another class); RFC is the response for a class (how many methods can potentially be executed in response to a message); LCOM measures lack of cohesion in methods (the number of method pairs which do not share instance variables minus the ones sharing at least one variable); Ca measures afferent couplings (how many other classes use a given class); NPM is the number of public methods. See also http://www.spinellis.gr/sw/ckjm/doc/metric.html).

It can be seen that the change has, firstly, increased cohesion, and, secondly, reduced external coupling. While not all metrics can be simply summed, it is apparent the coupling between objects (CBO) has roughly split between the two new classes and similarly for the response for a class (RFC) and afferent couplings (Ca), reducing the per-class coupling. The dominating change, however, is in the cohesion metric (LCOM), highlighted in the table: the methods in the two new classes are much more cohesive than before.

This demonstrates that the component's structure has improved: The couplings to middleware and task toolkit are lower than previously to the middleware alone. Cohesion is now also

higher, showing the better separation of concerns.

Furthermore, the task toolkit is simpler to use than the conceptually less powerful RPC middleware. While this might be surprising at first, it can be explained by the fact that the life-cycle captures similarities between task-using components which are not generally available. Therefore, a task toolkit can move more functionality into the toolkit, which component developers using regular RPC middleware would have to provide themselves.

6.2 Low Coupling with Tight Integration

The issue of coupling concerns us very much, because, on the one hand, low coupling is desirable to make systems changeable, but on the other hand, many *functional* advances in recent robotic systems seem to cause more coupling. For example, in section 6.1.1 above, we described a fairly close exchange of information, where a component in the navigation sub-system ("following") keeps one in the interaction sub-system ("dialog") informed about its state throughout action execution. This coupling is necessary, so that the dialog can inform the user about what the system is trying to do. Fortunately, because it occurs through state-transition notifications in the common, generic life-cycle, the components are still fairly loosely coupled.

In the following, we will present several case studies that examine this in more detail. We will show how the pattern provides both abstraction and enough information for coordination. Furthermore, we will also highlight how the pattern has enabled efficient team-work during development.

6.2.1 Background on the System Studied

In contrast to the two small studies presented earlier, the case studies in this section are based on a fully realized system, the "Curious Robot" scenario [27, 37, 28]. It is an interactive robot system that learns to name and grasp objects from a human tutor. It contains both a full manipulator setup with the usual trajectory/grasp planning and execution components, as well as an anthropomorphic robot, which serves as the interface robot. See figure 12 for an image of the setup.

The focus of the system is on the mixed-initative [4] speech dialog, which also motivates the description as "curious": Both the robot and the human can take initiative in the interaction, with the robot asking for object labels and grasps, and the human commenting on the robot's action execution.

Architecturally, the main issues have been to enable the dialog to both describe what the robot is doing, and allow the human tutor to interact with ongoing robot actions, e.g. for correction of grasp type and position. This integration is based on the task state pattern. Furthermore, the interface robot can produce non-verbal cues, such as orienting its gaze to indicate attentional focus, which is also based on observation of the currently active task.

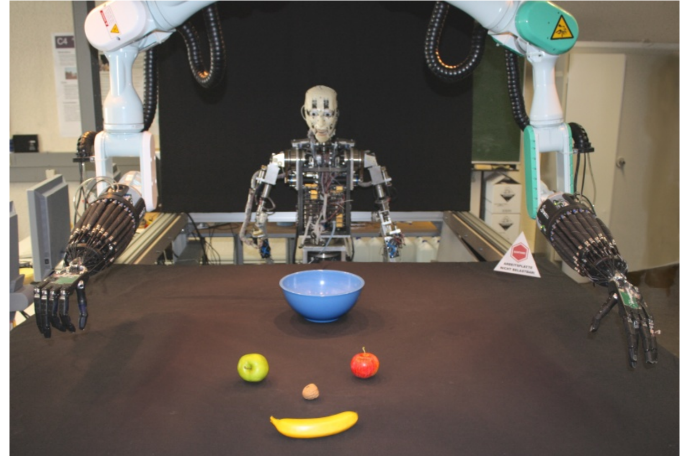


Fig. 12: Curious Robot Setup (from [27])

6.2.2 Study Design

The work presented in this section has been carried out as an active design study [40] during the toolkit's use, which means that the study's results guided the toolkit's further development. The overall system underwent three development iterations during the study period.

We have collected data by examining the state-transition notifications between components. UML sequence diagram notation is used to visualize the resulting message flow. By comparing the transition type with what actually happens in the system, we can see how the pattern is used, and whether the usage is appropriate.

6.2.3 Case 1: Three task types, one pattern

The first study has been taken from the first iteration of the system and thus represents the initial integration.

In figure 13, we see three different tasks: i) System initiative to ask for a label (1-5), ii) moving the arm and hand in a pointing motion (7, 8, 10) and back (18, 20), and iii) text-to-speech (7, 9, 11, 19, 21). In one of these (label query, 12-16) the task is rejected.

The main message of this case study is that the coordination component ("dialog manager") is able to coordinate several different tasks, as well as take the system-initiatives's request, using the same method: An implementation of the pattern.

Apart from this, the task sequences are unsurprising, and the mapping from action states to state-transitions is straightforward. Perhaps indicative of the early stage of integration, none of them modifies an ongoing task, they all use the simple life-cycle (cf. figure 2).

6.2.4 Case 2: Generic feedback and updates of tasks

The second study examines a tighter interaction, by modifying an ongoing task. The functional motivation is that users often comment during execution of a task and that if the integration is good enough to maintain this relationship, interpretation

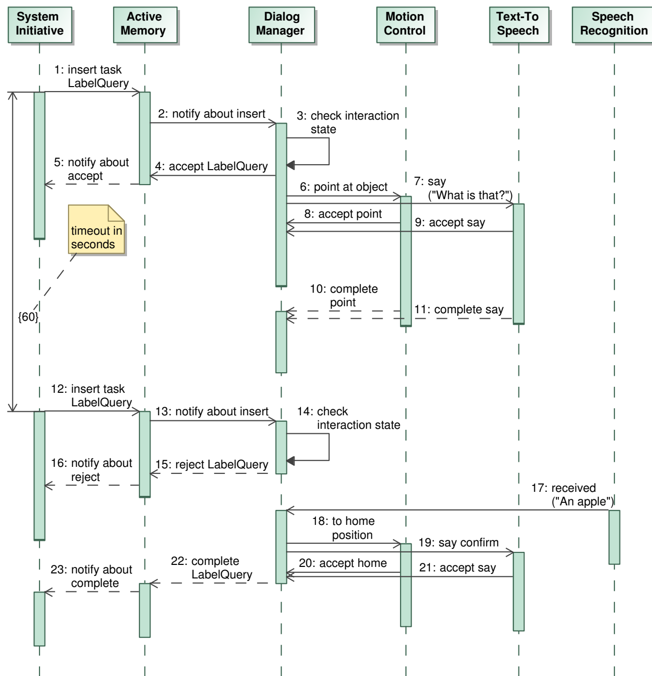


Fig. 13: *Curious Robot* example label query interaction. All life-lines shown represent independent components which communicate through the middleware. The Active Memory is the event-bus component.

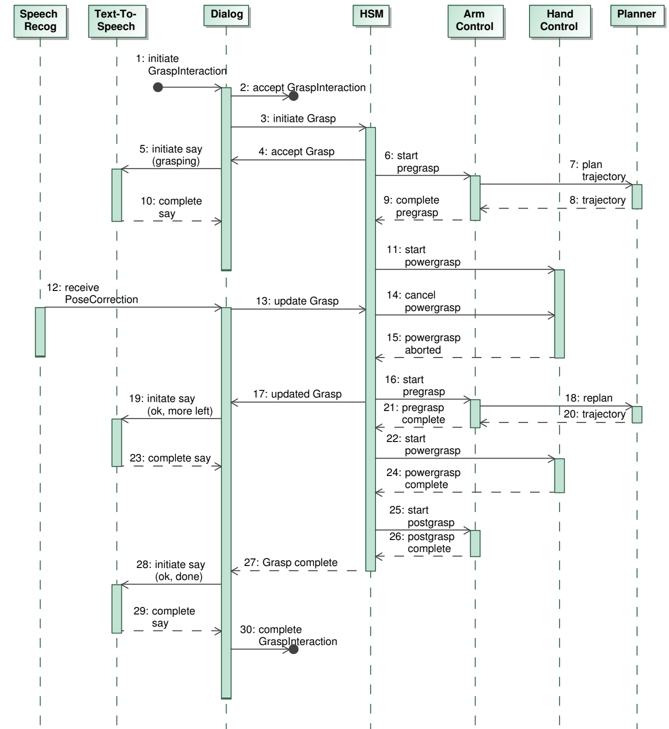


Fig. 14: *Curious Robot* example grasping with user corrections. All life-lines represent independent components, except for Trajectory Planner which is contained with Arm Control.

of the comment is simplified. In other words, the context should be kept, and as we can see from figure 14, the UPDATE transition of the extended life-cycle does so.

That said, the two components being integrated here (dialog and grasping) are from very different sub-systems and should not be overly dependent. This is achieved here through a coordination component for each sub-system, the “dialog manager” on the one side, and the “Hierarchical State Machine” (HSM) on the other. These components represent facades for their respective sub-system.

To provide a point of comparison, we now consider what would happen if only remote-procedure calls (RPCs) were available. Using RPCs, only one piece of information can be transmitted per call and direction, which would be the original grasping goal. Therefore, changing the goal requires another call, and it must specify that it is an update. In the general case, it cannot be assumed that the dialog is the only client, so updates must reference the previous goal somehow. This requires feedback, so the first call must have returned immediately, with an id for reference. As a consequence, information about completion must be communicated through a different channel, either through polling, or by having the client register a callback with the server. One could argue that this could also be performed through a publisher, but then we are already in the realm of asynchronous event notifications, with all the associated implementation difficulties (cf. section 4.5.1). In effect then, the RPC implementation would have introduced an id for referencing tasks, two different calls for modification, and a third method, or a callback for completion information. This is essentially a re-implementation of what the pattern provides by other means, specific to this particular type of task, and with more setup overhead (registering a callback).

In contrast, the pattern’s implementation provides a consistent interface for all of these functions, and adding the update transition is a straightforward extension of the life-cycle.

6.2.5 Case 3: Loosely coupled extension by observation

Ideally, it is possible to add functionality to a system without having to change anything that already exists. In reality, such cases are rare, but this case study represents one of them. One of the principles underlying the the task-state pattern – observability – has been particularly important in achieving it.

The feature addition has been gaze feedback: Gaze, the direction of the eyes, is an important social cue during interaction and communicates attention to a particular point in the environment. In the Curious Robot system, the focus of attention is known: It is either the human, during direct interaction, or the object, during grasping. What was lacking is to orient the interface robot’s head and eyes towards it.

The task-state pattern publishes all state-changes to an event-bus, where they are visible to all interested components

in a system. In this way, the current focus of attention can be determined from a) the presence, and b) the state of tasks relating either to grasping, or interaction.

Figure 15 depicts the case when an interaction task (“Label-Query”) is initiated. First, the dialog receives this task (2) and accepts it (3). The acceptance state is then also transmitted to the gaze feedback module (4). In this simple manner, gaze feedback can be added solely based on information already available in the system.

This case also highlights the importance of intermediate feedback: The gaze should only change when an interaction task is actually accepted, not every time it is requested. Thus, it is not only observability as such – which could also be realized by other means – but also that there is a generic state.

However, it must also be said that the coupling is stronger in one direction than in the other, because the gaze feedback component interprets more than just the state: It also interprets the task-specific parts of the notification, as these contain the position. Thus, data coupling is still present in the gaze feedback component (but only there, for this function).

In the Curious Robot system, we have further reduced data coupling by using a simple, generic representation for all regions that can be attended to. Thus, the dialog uses this representation for the position of the face of the human it is interacting with, and the grasp module uses it to represent the position of the object being grasped. The task-state pattern is also useful without such a generic representation, but it certainly benefits from its availability.

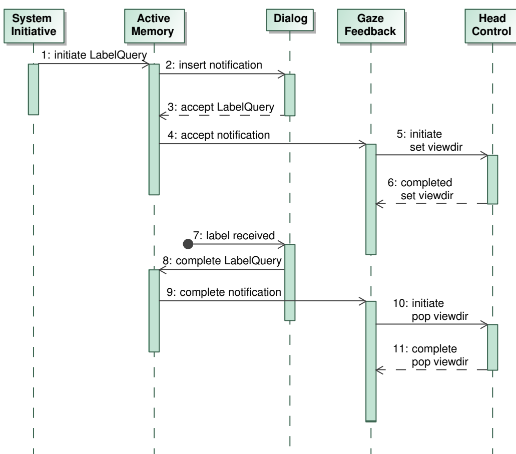


Fig. 15: *Curious Robot* example gaze feedback of dialog state.

6.3 Effects on Team-Work

So far, we have discussed the case studies mainly with respect to the reduction of complexity and coupling. However, they also allow insights into the team-work needed during system construction, which we will now summarize.

The most obvious effects are noticeable in the full system case studies described in section 6.2. The primary effect has been **reduced need for developer communication** due to the generalization provided. This is visible in all of the cases: For the first, the application of the same interaction pattern for several task types saves time on repeated application. While the task-specific parts must still be specified, the pattern pre-specifies the generic parts, and also *when* messages are sent. The second case demonstrates that the pattern can provide functionality which would otherwise require a much larger, specific interface. Finally, in the third case, the pattern enabled one developer to add and test functionality without having to consult the other developers at all.

A second effect is **greater independence during development**, due to graceful degradation when functionality is not available. This is most visible in the second case, where the life-cycle has been extended. The extension itself has been made in the toolkit, which defaults to canceling and re-starting a task, if the server does not support updates, or refusing the update, when canceling is not supported either. Therefore, client and server could add the necessary functionality independently, reducing work scheduling constraints.

Last, but not least, the observability of task states has enabled a new set of analysis tools, visualizing active tasks, and task results. This has provided helpful insight into the operation of the overall system, e.g. to pin-point problems more easily.

7 Discussion and Outlook

So far, we have discussed the pattern mainly with respect to its historical roots and exemplary cases. In this section, we take a broader perspective, and touch upon some general design decisions, and the relevance of the pattern in the larger context of robot software system design and construction. We also take this opportunity to point out where we still see room for improvement or further investigation.

7.1 Effects on the System Development Cycle

The goals we pursued have been motivated by experience from the typical system development cycles, particularly the often enormous amount of “elbow grease” expended during system integration. While the reasons for such situations are many, it is generally accepted that integration benefits from *shared knowledge* between developers. Thus, we believe that identifying commonalities across the various platforms, middlewares, and programming approaches is an important step in this direction. The pattern represents such a commonality: It is re-usable knowledge that can bridge the conceptual gaps between developers and thus facilitate integration.

Furthermore, if shared knowledge can be implemented as a building block independent of specific middlewares or architectures, its application becomes much easier and wider reaching. This is in contrast to approaches which only work in specific eco-systems of surrounding software. That the pattern passes this test has been shown by the fact that there are

now independent toolkits available, and that these toolkits have been used to realize diverse system structures. That said, the existing toolkits are not universally applicable. Thus, one interesting open question would be whether the approach can be usefully applied to most or even all robot software systems.

A more immediate benefit from having a toolkit is that it assists during the development cycle by providing a readily usable life-cycle model, and reducing the burden for implementing non-blocking clients and servers. Furthermore, by encapsulating the communication, it can prevent many common mistakes.

Apart from the technical help for integration and coordination, in our development and use of the TSP we have experienced a boost for the communication between the humans involved in building applications: It helps component designers and implementers establish clear technical documentation of the interactions, and system architects can rely on a unified coordination model. Team discussions for life-cycle design also help to establish a shared view of coordination for a particular domain, and allows all developers to perceive their contribution in a wider system context.

7.2 Life-Cycle Constraints and Design

The common life-cycle is a crucial aspect of the patterns implementation, and determines the capabilities of the resulting system. It must be appropriate, or adoption will suffer. At the moment, the only direct experience with designing life-cycles for this particular purpose stems from general state-machine design advice (e.g. [12, 1.2.4]) and the experiences of the present authors, which is limited to certain systems.

That said, it is certainly promising that the implementations by the presents authors, which were conceived completely independently, have arrived at almost the same state machine structure, despite differences in terminology, and despite having made some different choices in other areas. Compare figure 10, which depicts XTT's state-machine and figure 11, from ROS's actionlib. Apart from differences in naming, the structure is very similar. The actionlib machine has a few more transitions, but this is primarily due to the fact that the underlying transport is assumed to be unreliable, whereas XTT assumes a reliable transport.

We surmise that this similarity is due to the essential constraints of the task concept: There is always a start and an end, and the outcome can, essentially, be success or failure. Within these categories more fine-grained distinctions may be made, but at the moment, both our implementations leave these to the task-specific part of the representation. In terms of interaction, stopping a task prematurely (by both client and server) and updating the goal seem to be the essential interactions, leading to respective transitions. Last, but not least, both implementations allow providing intermediate results, though this is so far used only by few components.

Where the state-machines differ (slightly) is in how they treat updates. The XTT state-machine has a dedicated state for this, whereas ROS's actionlib handles it by sending a new goal. This difference has little effect on the interaction between client and server – they always see all messages and can reconstruct an update from the temporal succession. However, it complicates things slightly for observers, especially those which start listening after the task has already started: They need to reconstruct goal and state explicitly, from a number of messages, which is impossible when not all of them are available. XTT, in contrast, has always placed more of a focus on observation, and so maintains a single representation of the current task state, including information that an update has occurred. The drawback of XTT's choice is that the work to maintain this representation may not always be needed, the advantage that it is easier to observe. Neither of these is more powerful in principle, they just differ in how easy certain things are.

7.3 Potential Liabilities

Of course, there are also cases where the pattern is inappropriate, or uses which are inefficient. In particular, while we have not encountered such cases, we can think of the following symptomatic aspects:

Mismatch to action life-time: for very short-lived tasks, the communication overhead of multiple state change notification messages may be unwarranted. Event-notifications over standard TCP channels can quickly accumulate latencies, and RPC-based middlewares are even worse.

Overcomplicated life-cycle: the attempt to provide a single life-cycle that encompasses all possibilities may result in an automaton that is too complicated for most components. In this case, if a more suitable life-cycle cannot be found through redesign, other coordination methods should be explored.

Tendency to discretize: the explicit states of the life-cycle model may cause a discretization of tasks whose state would better be represented with continuous values. Obvious cases, such as dynamical systems, are unlikely to use the TSP, but less clear-cut cases could suffer from inappropriate use of the pattern.

8 Conclusion

In the introduction, we have motivated our approach by the desire to provide an **abstraction** for diverse tasks and have introduced the "task-state pattern" as a proven concept to realize such an abstraction. This pattern has been described at a detailed level, suitable for direct implementation.

The pattern's utility has been demonstrated through historical analysis, independent implementation, and a number of real-world case studies. These show that diverse tasks of a robot can be invoked, observed and manipulated through a common interface (cf. 6.2.3). We have also shown how a more versatile life-cycle can enable more powerful interaction (cf. 6.2.4), and have introduced means to adopt such life-cycles in a gradual fashion (cf. 4.4.2).

Another goal has been the **separation of concerns**. We have demonstrated this both on the level of individual task servers (cf. 6.1.2), as well as on the overall architectural level, between sub-systems (cf. 6.2).

However, as of now, we do not possess empirical data on this quality for components other than servers. In particular, the task representations always contain task-specific contents, and we have not yet studied how this affects the separation of concerns. The fact that some coordination components interacted with multiple different types of tasks suggests that it is easily possible, but this would certainly warrant further studies to gain a more accurate picture.

The desired **evolvability** has been provided at the application programming interface level of the toolkits (cf. 4.4.2), for which we could also show a simplifying effect (cf 6.1.2). Evolvability at the system levels has been demonstrated through several iterations and reduced coupling (particularly 6.2.5).

For future work, an obvious next step is to explore further application domains. While our case studies already contain several systems and their evolution, they certainly cannot encompass the full spectrum of robot applications. An exhaustive quantiative and qualitative examination of all domains would be unpractical, however, so we also consider it important to explore the potential for simplifying patterns in other areas of robot software architecture, as a stepping stone towards better comparability. A second avenue towards better empirical comparison that we are currently examining are more automatable measures for architectural qualities, based on the additional information provided by the pattern.

In summary, we have given a recipe for realizing a proven coordination interface for robotics that is applicable to a variety of architectures and systems, and have grounded it both analytically and empirically.

References

[1] Programmable controllers - part 3: Programming languages. International Standard IEC 61131-3, International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland, 2003. 28

[2] Function blocks - part 1: Architecture. International Standard IEC 61499-1, International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland, 2005. 28

[3] Introduction into IEC 61131-3 Programming Languages. URL http://www.plcopen.org/pages/tc1_standards/iec_61131_3/. Accessed 21st September 2011. 28

[4] James F. Allen. Mixed-initiative interaction. *IEEE Intelligent Systems*, 14(5):14–23, 1999. ISSN 1541-1672. doi: http://dx.doi.org/10.1109/5254.796083. 33

[5] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. RT-component object model in RT-Middleware – Distributed component middleware for RT (robot technology). pages 457–462, June 2005. doi: 10.1109/CIRA.2005.1554319. 28

[6] Ronald C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, 8(4):92–112, August 1989. doi: 10.1177/027836498900800406. URL http://dx.doi.org/10.1177/027836498900800406. 23

[7] Olaf Booij, Ben Kröse, Julia Peltason, Thorsten Spexard, and Marc Hanheide. Moving from augmented to interactive mapping. In *Robotics: Science and Systems Conference*, Zurich, 2008. 31

[8] Rodney Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, January 1986. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1087032. 22, 23

[9] Davide Brugali and Patrizia Scandurra. Component-based robotic engineering (part i) [tutorial]. *Robotics & Automation Magazine, IEEE*, 16(4):84–96, December 2009. doi: 10.1109/MRA.2009.934837. URL http://dx.doi.org/10.1109/MRA.2009.934837. 22, 23

[10] Davide Brugali, Alex Brooks, Anthony Cowley, Carle Côté, Antonio Domínguez-Brito, Dominic Létourneau, Françis Michaud, and Christian Schlegel. Trends in component-based robotics. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*, chapter 8, pages 135–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-68949-2. doi: 10.1007/978-3-540-68951-5_8. URL http://dx.doi.org/10.1007/978-3-540-68951-5_8. 23

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 1996. 21

[12] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008. ISBN 978-0-387-33332-8. URL http://www.springerlink.com/content/978-0-387-68612-7#section=259931\&page=5\&locus=0. 36

[13] Raja Chatila. Deliberation and reactivity in autonomous mobile robots. *Robotics and Autonomous Systems*, 16(2–4):197–211, 1995. 29

[14] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. 32

[15] Jonathan H. Connell. Sss: a hybrid architecture applied to robot navigation. pages 2719–2724 vol.3, May 1992. doi: 10.1109/ROBOT.1992.219995. URL http://dx.doi.org/10.1109/ROBOT.1992.219995. 22

[16] Patrick T. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL http://dx.doi.org/10.1145/857076.857078. 23

[17] Norman Fenton. Viewpoint article: Conducting and presenting empirical software engineering. *Empirical Software Engineering*, 6(3):195–200, September 2001. ISSN 13823256. doi: 10.1023/A:1011449731678. URL http://dx.doi.org/10.1023/A:1011449731678. 21

[18] Sara Fleury, Matthieu Herrb, and Raja Chatila. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 1997. 29

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. 21, 25

[20] Erann Gat. *Three-layer architectures*, pages 195–210. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-61137-6. 22, 23

[21] Marc Hanheide and Gerhard Sagerer. Active memory-based interaction strategies for learning-enabling behaviors. In *International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Munich, 01/08/2008 2008. 28, 30, 31

[22] Karl Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010. ISBN 978-3642120145. 28

[23] Barbara A. Kitchenham, Shari L. Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled E. Emam, and Jarrett Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1027796. URL http://dx.doi.org/10.1109/TSE.2002.1027796. 21

[24] David Kortenkamp and Reid Simmons. *Robotic System Architectures and Programming*, chapter 8, pages 187–206. Springer-Verlag, 2008. 20, 23

[25] J. Kramer. Configuration programming-a framework for the development of distributable systems. pages 374–384, 1990. doi: 10.1109/CMPEUR.1990.113648. URL http://dx.doi.org/10.1109/CMPEUR.1990.113648. 22

[26] D. R. Lefebvre and George N Saridis. A computer architecture for in-

telligent machines. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 3, pages 2745–2750, May 1992. doi: 10.1109/ROBOT.1992.219991. URL http://dx.doi.org/10.1109/ROBOT.1992.219991. 28, 29

[27] Ingo Lütkebohle, Julia Peltason, Lars Schillingmann, Christof Elbrechter, Britta Wrede, Sven Wachsmuth, and Robert Haschke. The curious robot - structuring interactive robot learning. In *International Conference on Robotics and Automation*, Kobe, Japan, May 2009. Robotics and Automation Society, IEEE. 30, 33

[28] Ingo Lütkebohle, Julia Peltason, Lars Schillingmann, Christof Elbrechter, Sven Wachsmuth, Britta Wrede, and Robert Haschke. A mixed-initiative approach to interactive robot tutoring. In *Towards Service Robots for Everyday Environments*. Springer, 2011. in press. 33

[29] Eitan Marder-Eppstein and Vijay Pradeep. ROS actionlib package documentation, 2009. URL http://www.ros.org/wiki/actionlib. 30

[30] Maja J Mataric. Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3): 304–312, June 1992. ISSN 1042296X. doi: 10.1109/70.143349. URL http://dx.doi.org/10.1109/70.143349. 22

[31] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3 (1):43–48, 2006. ISSN 1729-8806. 23

[32] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer, 1st edition, July 2006. ISBN 3540326510. 23

[33] Anders Orebäck. *A Component Framework for Autonomous Mobile Robots*. PhD thesis, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden, 2004. 23

[34] George A. Papadopoulos and Farhad Arbab. *Coordination Models and Languages*, volume 46 of *Advances in Computers*, pages 330–400. Academic Press, 1998. ISBN 0-12-012146-8. 22

[35] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355602.361309. 23

[36] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL http://dx.doi.org/10.1145/361598.361623. 23

[37] Julia Peltason, Frederic H.K. Siepmann, Thorsten P. Spexard, Britta Wrede, Marc Hanheide, and Elin A. Topp. Mixed-initiative in human augmented mapping. In *International Conference on Robotics and Automation*, Kobe, Japan, 14/05/2009 2009. IEEE, IEEE. 33

[38] Paul G. Plöger, Kai Pervölz, Christoph Mies, Patrick Eyerich, Michael Brenner, and Bernhard Nebel. The DESIRE service robotics initiative. *KI - Zeitschrift Künstliche Intelligenz*, 22(4):29–32, 2008. ISSN 0933-1875. 30

[39] Matthias Radestock and Susan Eisenbach. Coordination in evolving systems. In *TreDS '96: Proceedings of the International Workshop on Trends in Distributed Systems*, pages 162–176, London, UK, 1996. Springer-Verlag. ISBN 3-540-61842-2. URL http://portal.acm.org/citation.cfm?id=695228. 22

[40] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. ISSN 1382-3256. doi: 10.1007/s10664-008-9102-8. URL http://dx.doi.org/10.1007/s10664-008-9102-8. 21, 33

[41] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschman. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 2000. 25

[42] Marc Schröder and Jürgen Trouvain. The German Text-to-Speech Synthesis System MARY: A Tool for Research, Development and Teaching. *International Journal of Speech Technology*, 6(4):365–377, October 2003. ISSN 13812416. doi: 10.1023/A:1025708916924. URL http://dx.doi.org/10.1023/A:1025708916924. 32

[43] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proc. of Conference on Intelligent Robotics and Systems*, 1998. 28, 29

[44] R. G. Simmons. Structured control for autonomous robots. *Robotics and Automation, IEEE Transactions on*, 10(1):34–43, February 1994. doi: 10.1109/70.285583. URL http://dx.doi.org/10.1109/70.285583. 22, 28, 29

[45] Reid Simmons and Dale James. *Inter Process Communication (IPC) – A reference manual.* Robotics Institute, Carnegie Mellon University, Pittsburg, PA, USA, 2001. URL http://www.cs.cmu.edu/~ipc/. The manual is for version 2.6. In April 2010, the IPC software is at version 2.8.5. 29

[46] P. Soetens. RTT: Real-Time Toolkit. http://people.mech.kuleuven.be/~orocos/pub/stable/documentation/rtt/current/doc-xml/orocos-components-manual.html. Accessed 12th July 2010. 28

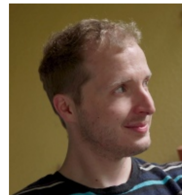
[47] D. B. Stewart and P. K. Khosla. Rapid development of robotic applications using component-based real-time software. *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, 1:465+, 1995. doi: 10.1109/IROS.1995.525837. URL http://dx.doi.org/10.1109/IROS.1995.525837. 23

[48] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497, December 2002. doi: 10.1109/TRA.2002.802930. URL http://dx.doi.org/10.1109/TRA.2002.802930. 23

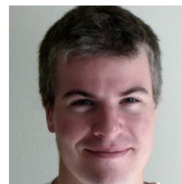
[49] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, pages 1/121–1/132 vol.1, 2001. doi: 10.1109/AERO.2001.931701. URL http://dx.doi.org/10.1109/AERO.2001.931701. 23

[50] Sven Wachsmuth, Sebastian Wrede, and Marc Hanheide. Coordinating interactive vision behaviors for cognitive assistance. *Computer Vision and Image Understanding*, 108:135–149, 2007. 30

[51] Sebastian Wrede, Marc Hanheide, Sven Wachsmuth, and Gerhard Sagerer. Integration and coordination in a cognitive vision system. In *International Conference on Computer Vision Systems (ICVS)*, St. Johns University, Manhattan, New York City, USA, 2006. IEEE. URL files/papers/Wrede2006-IAC.pdf. International Conference on Computer Vision Systems 2006. 30, 32



Ingo Lütkebohle is a researcher at the Applied Informatics Group, Bielefeld University, Germany. He received his Ph.D. from Bielefeld University in 2010, for work on coordinating and rapid prototyping of components for robot software. His main research interest is the advancement of empirical software engineering methods for principled engineering of robot software. On the side, he dabbles in Human-Robot-Interaction, event-based middleware, and the construction of anthropomorphic robots. Ingo is a strong supporter of Open Source Software, and open standards in robotics. He is a founding member of the Robot Engineering Task Force, and a member of IEEE, ACM, and GI e.V. Contact him at iluetkeb@techfak.uni-bielefeld.de



Roland Philippsen is Assistant Professor in the Intelligent Systems Lab at Halmstad University in Sweden. He received his Ph.D. from EPFL, Switzerland, in 2005, for work on mobile robot path planning and obstacle avoidance. His expertise lies in interweaving reasoning, planning, and control. Some of the real-world robotics project he has contributed to are: tour-guide robots at the Swiss Expo.02, educational robots for the EPFL student contest 2001-2004, robotic actors for a theater play in 2005, a Cognitive Robot Companion in 2006, the PR2 of Willow Garage Inc, and the Meka robot of HCRL at UT Austin. Roland always pushes for robust open source software in this area. He believes that autonomous robots can improve the lives of all humanity, provided we openly share knowledge and implementations.

Vijay Pradeep carried out the work in this paper while a Systems Engineer at Willow Garage. His primary research focus is in robot calibration and is interested in understanding how both offline and in-the-loop calibration techniques could allow simpler, less accurate robots to perform difficult manipulation tasks. He is also a co-author of the ROS actionlib library. Vijay received his M.S. in Mechanical Engineering from Stanford, focusing in Control Systems. While at Stanford, Vijay worked on the Starmac quadrotor project, where he focused mostly on the hardware design and software architecture for the testbed along with path planning algorithms for these dynamic vehicles.

Eitan Marder-Eppstein is a Software Engineer at Willow Garage. His current research interests include: autonomous navigation, mapping, and exploration for mobile robots; multi-robot coordination; and the software structure of robot action primitives. Eitan spends a good portion of his time maintaining Willow Garage's Open Source Navigation Stack, as well as helping out with various other software engineering tasks. He is also co-author of the ROS actionlib library. Eitan received both his M.S. and B.S degrees in Computer Science from Washington University in St. Louis.

Sven Wachsmuth is holding a lecturer position at Bielefeld University and is currently heading the Central Lab Facilities of the Center of Excellence Cognitive Interaction Technology (CITEC). He received the Ph.D. degree in computer science from Bielefeld University in 2001. In 2003, he spent a sabbatical year at the Computer Science Department of the University of Toronto. His research interests are in Human-Robot Interaction, especially looking at high-level computer vision problems, and system integration and evaluation aspects. He has been a member of the RoboCup@Home organization committee 2010-2011 and is member of the technical committee in 2012.