The Software Framework SmartSoft for Implementing Sensorimotor Systems[*]

Christian Schlegel and Robert Wörz

Research Institute for Applied Knowledge Processing (FAW)
Helmholtzstraße 16, D-89081 Ulm (Germany)
{schlegel, woerz}@faw.uni-ulm.de

Abstract

This paper presents the software framework SmartSoft *to implement sensorimotor systems.* SmartSoft *not only contains software components to support a modularized implementation but also structural rules and templates. These ensure that the implementation of modules is conforming with an overall multilayer system architecture. Since the provided structures allow the exact specification of the external behavior of modules in terms of interfaces and dependencies, the interaction with a symbolic task execution layer is explicitly supported.* SmartSoft *significantly eases the implementation and integration of new modules into a complex sensorimotor system which for example provides the opportunity even to compare and reuse different modules on a mobile platform.* SmartSoft *has already proven its usefulness within the collaborative research center (SFB 527) "Integration of Symbolic and Subsymbolic Information Processing in Adaptive Sensorimotor Systems".*

1 Introduction

The implementation of a complex sensorimotor system not only requires efficient components but also a concept for the overall integration. Within the system architecture various requirements of particular components have to be considered. The bigger the complexity of a system, the more important is its architecture. But the system architecture also needs an adequate support by a specific software framework. Besides structural rules, this framework has to provide assistance for creating a consistent implementation of various and different components. It also has to enable and to support an implementation according to the system architecture. Furthermore, it has to address specific challenges resulting from a distributed development of software components.

Despite the fact that the software framework plays a central role within a consistent integration, it is frequently neglected. As a result many systems are often built in a less structured way. They can be modified only with large effort. Although there are a lot of implementations of complex sensorimotor systems available, only very few supporting software frameworks exist. A key question is how to organize a software framework for sensorimotor systems.

The improvements in object oriented design and programming offer various approaches which can support the methodical design and implementation. So called middleware facilitates for instance the access to distributed objects. By using design patterns [5] for frequently required structures it can be assured that consistent and approved solutions are commonly used. Even though the implementation of complex systems is supported by the availability of corresponding tools, for almost every new application still the question has to be answered how to model and structure the application to take advantage of modern design principles and tools.

This is the central point where the framework SmartSoft assists the implementation of sensorimotor systems. It provides normally required structures which ensure that implemented components fit together. Critical building blocks are e.g. communication and synchronisation mechanisms. The provided software patterns are designed to address the specific requirements of sensorimotor systems using modern software components and principles. By using SmartSoft, in particular the question is answered *how* to implement components so that they can be easily in-

[*]This work is part of project C3 (http://www.uni-ulm.de/SMART/Projects/c3.html) of the SFB 527 sponsored by the Deutsche Forschungsgemeinschaft

tegrated, ported or extended. Thus mechanisms for the communication of modules, the parallel execution of tasks and the configuration of the modules' internal control flow are provided. Using an object oriented client-server architecture, which encapsulates the communication, enables the separation of implementations of algorithms from communication mechanisms. This allows to add a new functionality or to test an alternative approach without rebuilding the software of the robot and disburdens the user from integration challenges by ensuring the overall consistency. Within the SFB 527, it is the basis for structuring and implementing the *Demonstrator*, which is a B21 platform (figure 1) of RWI/ISI.

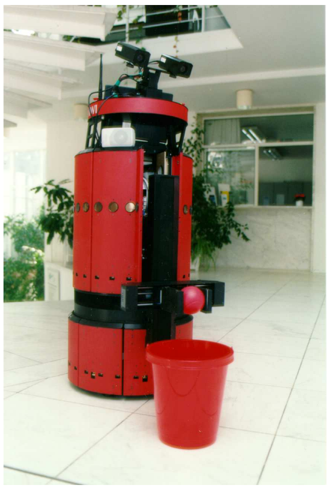

Figure 1: The Demonstrator

First chapter 2 shows some aspects of the underlying system architecture of the robot. Chapter 3 illustrates the basic structures provided by the framework. Chapter 4 explains some technical details of the implementation of the software and chapter 5 provides a conclusion and details future work.

2 Architecture and Requirements

2.1 System Architecture of the Robot

In addition to common requirements for a software framework for complex systems, specific requirements out of the field of robotics have to be considered. The mechanisms supported by SMARTSOFT are derived from the multilayer architecture shown in figure 2. This architecture is a proven model for cognitive robots [2, 4, 6]. Another overview on architectures can be found in [1]. In the following only parts relevant to SMARTSOFT are explained.

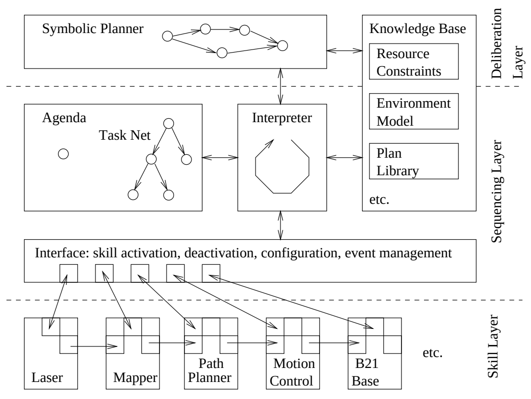

Figure 2: The System Architecture

A functionality of a module in the *skill layer* is called a *primitive skill*. One or more of them can be combined by the *sequencing layer* to form different *behaviors*. Behaviors implement control loops with a close coupling between sensors and actors. A module of the skill layer gets data as a client and serves as a server for its results. The parameter set of a module active at a given time is set from outside the module and can be changed dynamically at runtime. A module providing skills has to be structured in such a way that its processing time requirements comply with the overall timing constraints of the behaviors wherein the module is used. Additionally, a module must be able to detect and signal a malfunction of itself [7]. Thus other modules can intervene to cope with that situation.

Typical modules of the skill layer are the *path planner* and the *mapper*. The behavior *drive to region* e.g. combines the primitive skills *build actual map*, *find path* and *drive to subgoal without collision* of the modules *mapper*, *planner* and *motion control*.

The coordination and configuration is controlled by the *sequencing layer*. According to the actual situation, it selects the behaviors which can be used to achieve the current goals. The sequencing layer only works on discrete states which are used to synchronize the real world execution with the symbolic description of the expected execution progress.

The modules of the *deliberation layer* contain time-consuming algorithms, e.g. symbolic action planning. A typical characteristic of such a module is that during processing a query on that layer, several state changes on the sequencing layer may take place.

2.2 Task Execution

An important aspect is the interaction between the different layers. The key components of the interface between the skill layer and the sequencing layer are *configurations* and *events*. Events fire if specific conditions become true and report e.g. the successful completion of a task. They are processed on the sequencing layer by *wait-for*-constructs in task nets. Figure 3 shows a part of the declarative representation of a behavior, which is composed of different skills. The syntax is similar to the RAP-system [4]. *Constraints* have to be true throughout the execution and the *configuration* describes the setup procedure. More complex behaviors are represented in task nets using ordering constraints like *parallel-or* and *sequence* [10]. The agenda based execution of task nets not only considers the various constraints of a behavior but also uses the declarative description of configurations to generate the appropriate setup sequence based on uniform module interfaces.

```
(define-rap (follow ?object)
  (method
    (constraints (and
        (laser-module globalstate active)
        (vision-module globalstate active)
      ...))
    (configuration (
        (cdl-module goal-source vision)
        (cdl-module stuck-event continuous ?e1)
      ...))
    (task-net
      (parallel
        (t1 (wait-for ?e1 ...) :proceed)
        (t2 (wait-for ?e1 ...) :finish)
```

Figure 3: Part of a Behavior Representation

2.3 Requirements and Solutions

The supported concepts of the framework cover the distributed implementation as different modules, the communication between modules and the internal structure of a module. Technically, each *module* is a *separate process*. Using the framework, different variants of implementations are minimized and distributed development is supported. Some of the main features will be discussed in the next sections.

The division into several distinct modules is supported by a rigorous client-server architecture. Spreading of modules over several computers is transparent at the implementation level since all communication mechanisms are provided as standardized templates. Furthermore, all server services of a module are summarized into a single server object for the client side. If one module wants to access another module, it has to include that server object. To access a service, the client calls a method of its local server object. The access to a server object is in no way different to calling a method of any object. Transparently for the user, even complex communication patterns between different modules (and therefore between different processes) can be started by a local method call. Using parallel threads, the communication of a module is decoupled from the computation of an algorithm.

Another important feature of the communication concerns the interaction between modules. To be able to describe the relationship between different modules and their dependencies, the supported communication variants are limited to a few patterns. Using only standardized communication patterns, the external behavior of a module is completely described by the used patterns. Therefore a module behaves externally always the same independent of the internal implementation. Thus a module can be viewed as a black box which can be easily exchanged with alternative implementations. This is the basic key to bridge the gap between a declarative description of behaviors, their configurations and compatibility constraints and skills as building blocks for behaviors.

The combination of several skills into different behaviors presumes a configuration interface of a module. The patterns of the framework have to support e.g. the deactivation of a module at the skill layer even when the module waits in a blocking call for an answer. The abstract view of the framework has to hide this mechanism which results in a reduction of the effort needed to implement a module. Furthermore these patterns increase the reliability of the implementation by using approved and tested patterns.

Realtime aspects are *not* in the main focus. Timer functions provide mechanisms to regularly invoke methods. The rigorous thread based implementation where no polling is used ensures adequate reactiveness.

3 The Framework SmartSoft

3.1 The Basic Structure of a Module

The basic structure of a module is shown in figure 4. The configuration class manages the internal module state including the activation, deactivation and selection of the various states. It must be included in every module. The functionality of a module is implemented in the user part using independent threads derived from a task class. The communication is handled in a separate thread and is therefore transparent to the user. This results in an ensured reaction of the module to external requests.

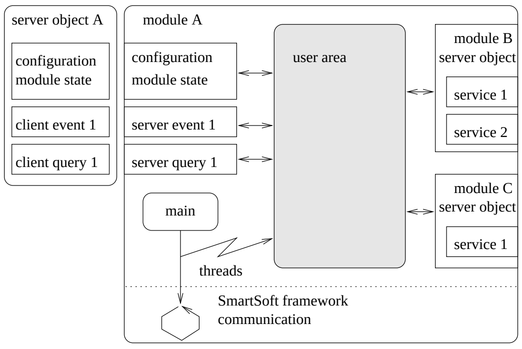

Figure 4: The Basic Structure of a Module

The external interface of a module is implemented with predefined primitives, each containing a server and a client side. These primitives provide complete communication patterns (e.g. for executing a query) and they need only be parameterized with the desired data types. For providing a service the server side of the primitives has to be used. The corresponding client sides of all implemented services of a module are combined into one single server object for the client. If a client wants to use a service of a module it has to include the appropriate server object and has to link the module's client library. Services are used by calling methods of the server object. Since the server object is automatically generated from standardized communication patterns by inheritance, it provides a much more structured representation of the server functionality as could be done by simple library calls. Simply swapping a module's server object class within a client allows to access an alternative implementation as long as the same interface is used. Figure 4 shows a module A. This module includes the server object of module B, which hides two service primitives, and module C, which hides one primitive. Therefore module A can use the services of module B and C. The server object of module A, which is used to get access to the services of module A, contains three primitives: the configuration class for the module state, an event and a query.

Server objects needed by a module have to be included only once. The synchronisation of concurrent accesses of several user threads within the module are handled internally and transparently by the primitive itself. This significantly simplifies the implementation of complex module structures using several threads.

3.2 The Communication Primitives

This section explains the available communication primitives. Figures 5 and 6 show communication patterns initiated by the client. Figure 8 introduces a communication pattern which is initiated by the server itself. The layer called *System* is implemented by the framework, the methods denoted with *Server* and *Client* are visible to the user. The variables A, B etc. are the communicated data objects. These have to be defined by the user while instantiating the primitives.

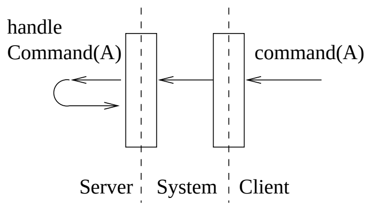

Figure 5: The Primitive *Command*

The primitive *Command* is used to implement an unidirectional communication. The object A is sent from the client side to the server, which evaluates the object using the command handler. This handler is called at the server side once a command object is received. The handler evaluates the object either by itself or puts it into a queue which is handled by another thread of the module. When instantiating this template not only the communicated object has to be provided, but also a handler method at the server side. An example for this primitive is setting the actual driving speed.

In contrast to the previous primitive, the communication primitives *Command with Status* and *Query* return an object from the server. The two patterns only differ in the names of their methods and the returned object which is used as a status in case of *Command with Status*. Figure 7 shows the timing of a query. The request object A must contain all information for the specification and the processing of the query. On the server side an incoming request is processed by a query handler which typically forwards the request to separate processing threads. These can send the result B to the appropriate clients using the unique query identifier *Id*, which is internally used to assign the correct request-result pairs. On the client side the query method is used for a blocking query which returns when the result has arrived. The other two methods are used for asynchronously sending a request and waiting for the result later. The concurrent execution of several queries within several threads and the correct assignment of the incoming results are handled completely within the communication primitives.

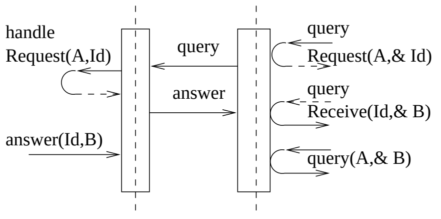

Figure 6: The Primitive *Query*

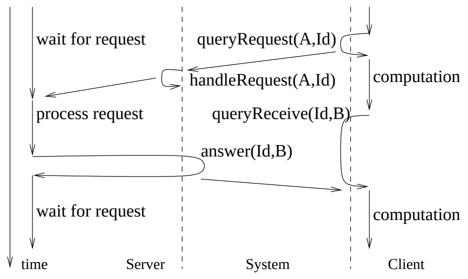

Figure 7: The Timing of a *Query*

The primitives introduced so far are suitable for a client which actively wants to send information to or get information from a server. Whereas these primitives provide communication mechanisms initiated from the client side (pull service), the next primitive shown in figure 8 is initiated from the server side (push service). It enables a server to actively provide new data to subscribed clients.

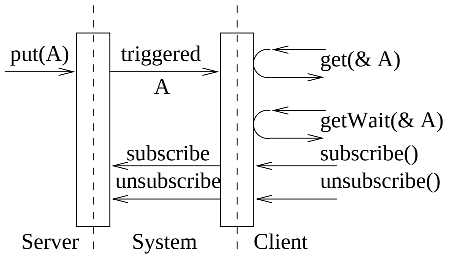

Figure 8: The Primitive *Autoupdate Newest*

The primitive *Autoupdate Newest* sends the object A to all subscribed clients as soon as new data A is provided at the server side by the put method. A client has access to the latest data (*get*) or can wait via a blocking call for the next incoming new data (*getWait*).

The *Autoupdate Timed* differs only with respect to the point of time when new data is sent to subscribed clients. The server provides new data at arbitrary times, but the data is sent to the clients in prespecified intervals. A client can select an individual interval (in multiples of the server interval) to reduce the communication overhead.

3.3 The Events

A special communication service is provided by *Events*. They are used to asynchronously send information from a server to subscribed clients to inform them about a specific condition becoming true. Therefore events are extensively used to synchronize continuously working modules with state-based discrete control structures. For example, an event can be used to notify that the path planner can no longer find a path to the goal region. Every event can be activated with individual parameters and it fires if the event condition is true considering the corresponding parameters. By using an unique identifier for every event activation, the same event can be activated several times simultaneously with different parameters.

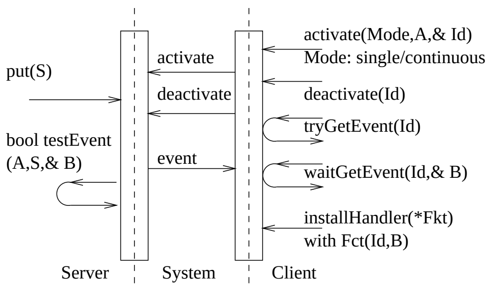

Figure 9: The Primitive *Event*

The interface of the event primitive is shown in figure 9. A client can activate an event with specific parameters A, can deactivate an event, can wait for the arrival of a specific event (blocking) or can install a handler at the client side which is called as soon as the event fires. The server provides new data S via the put method to the event object, which tests all parameters A of event activations under the actual data S using the *testEvent* funtion. If a test for a parameter set is satisfied the corresponding event fires. In the case of a *single* event it fires only once for each parameter set. A *continuous* event fires each time new data is provided via the put method as long as the event condition still holds.

3.4 The Configuration Class

Since not all modules have to (or can) be active at the same time and should be activated and parameterized according to the current task, a configuration class is provided (figure 10). It prevents transient states by delaying requested state changes until all concerned activities within a module have released that particular state. New state locks within a module are not granted as long as an external request is unprocessed. A dedicated *neutral* state inhibits all activities within a module and can be externally forced. In this case even all blocking calls within the module are canceled.

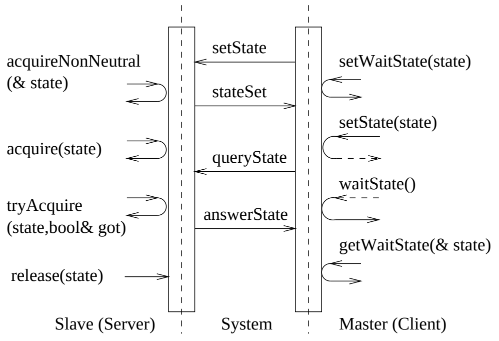

Figure 10: The Configuration Class

The *master* controls a module from outside and therefore corresponds to the client side. Activities within user threads depend on the availability of specific states which can be checked using the acquire methods. Each active state allows to specify an unlimited set of substates which are activated as soon as the corresponding main state is entered. This allows to easily switch on and off different processing options within a module. The clear structure and semantic of the activation mechanism is a necessary prerequisite to allow an uniform module representation within higher layers of the system architecture. Furthermore, it relieves implementing a module from considering complex interferences with other modules.

3.5 Example of a Module Interface

Figure 11 shows the interface of the self-localization module. This module uses the laser scanner and accesses the robot base for position updates. The module itself contains an interface for selecting a reference map using the command primitive to provide a filename as command object C. In addition, an event fires if the position uncertainty exceeds a threshold. The maximum allowed error is specified as parameter A during event activation. The object B sent by the event contains the robot position and the actual error matrix for a detailed description of the situation which triggered the event.

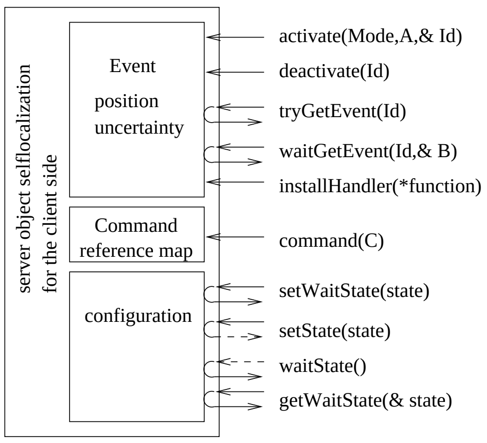

Figure 11: The Example Interface

4 The Implementation

Figure 12 shows an overview of the structure of the SMARTSOFT framework. Besides the primitives the application programmer interface contains also predefined module structures. A module is implemented as process where the user activities are handled in several threads based on the task class of the framework. The communication as well as the synchronization activities of the framework within a module are handled in a separate thread independent of the user threads of a module. Main parts of the implementation of SMARTSOFT are based on the freely available software package ACE [11]. This ensures portability among different platforms.

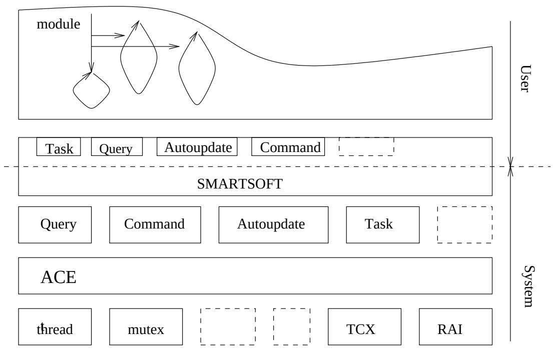

Figure 12: The Structure of the Framework

Throughout the whole framework no polling is used. This results in a very efficient usage of system resources. At the moment, the low level communication routines, which are invisible to the user, are built on top of the software package TCX [3]. Despite the overhead for flattening data structures for communication purposes, no significant overhead is produced by SmartSoft itself since it extensively uses e.g. condition variables. The framework itself is implemented in C++ under Linux and offers also an interface to LISP modules. If the framework should be used with other robots not compatible to the RWI/ISI B21, only modules providing interface services to the hardware of the robot have to be reimplemented.

5 Summary and Conclusions

The SmartSoft framework has already shown its applicability with the implementation of various modules of the SFB demonstrator. Currently, modules for path planning, motion control, collision avoidance, self-localization, video based object recognition and person following, speech input and speech output and symbolic task planning and task coordination are available. In addition, SmartSoft modules to provide comfortable access to the robot's hardware have been implemented. Among others these include server modules for the laser scanner, the robot base, the sonars, the video system, the pan tilt unit and the B21 manipulator.

The application programmer interface provided by SmartSoft allows a modular implementation of components for sensorimotor systems. In particular, the abstraction level achieved by providing templates for often needed structures goes beyond a simple object oriented design. For example there is a complete set of communication patterns which offer more than the raw communication by for example hiding synchronization and concurrency mechanisms in a user transparent way.

An important feature of the framework is the object oriented implementation and representation of the services of a module. Thus the implementation of algorithms can be done without considering any communication mechanisms. Additionally, there are design patterns for frequently needed structures. A consistent external interface including a structured representation of server services for clients is forced. By providing events and configuration mechanisms for modules, symbolic coordination and task execution mechanisms are supported. Through an abstract user interface and the usage of modern programming paradigms, the effort to adjust a system based on SmartSoft to new developments and needs is reduced. One of the next upgrades will use CORBA [8] as underlying communication system. A comprehensive description of the usage of SmartSoft can be found in [9].

Acknowledgments

The authors thank all members of the SFB 527 for valuable hints. These ensure that the framework becomes a universal tool for implementing complex sensorimotor systems.

References

[1] Special issue on integrated architectures for robot control and programming. *IJRR*, 17(4), 1998.

[2] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1), 1997.

[3] C. Fedor. *TCX - An Interprocess Communication System for Building Robotic Architectures: Programmer's Guide to Version 10.xx.* Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993.

[4] R. J. Firby. Adaptive execution in complex dynamic worlds. Technical Report 672, Department of Computer Science, Yale University, 1989.

[5] E. Gamma, R. Helm, and R. Johnson. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1997.

[6] D. Kortenkamp, R. P. Bonasso, and R. Murphy. *Artificial Intelligence and Mobile Robots - Case Studies of Successful Robot Systems.* AAAI Press and The MIT Press, Menlo Park, CA, 1998.

[7] F. Noreils. Integrating error recovery in a mobile robot control system. In *Int. Conf. on Robotics and Automation*, 1990.

[8] OMG. CORBA/IIOP 2.2 specification. http://www.omg.org/corba/.

[9] C. Schlegel and R. Wörz. SmartSoft - *Introduction.* FAW Ulm, Subproject C3, SFB 527.

[10] C. Schlegel and R. Wörz. Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework smartsoft. In *Eurobot*, Zürich, Schweiz, September 1999. (To appear).

[11] D. Schmidt. ACE - Adaptive Communication Environment. http://www.cs.wustl.edu/ ˜schmidt/ACE.html.