Technical Report SFB360–TR–96–3

Service Object Request Management Architecture

SORMA

Concepts and Examples

Jörg Walter and Helge Ritter

September 27, 1996

SFB-360 · Project D4 · Introductory Report of the
Arbeitsgruppe Neuroinformatik
Technische Fakultät
Universität Bielefeld
D–33615 Bielefeld

Please send any comments, remarks or feedback via
Email: walter@techfak.uni-bielefeld.de
Tel: 0521-106-6064 · Fax: +49-521-106-6011
For further information see:
http://www.techfak.uni-bielefeld.de/~walter/

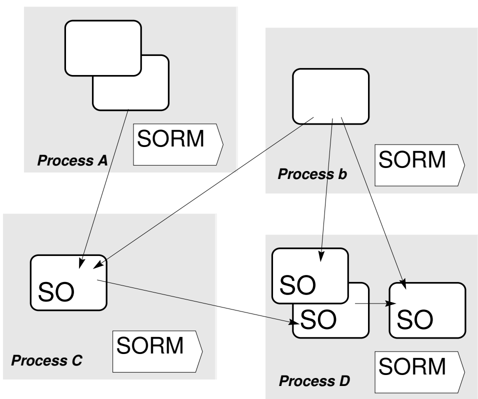
Abstract:

We report on **SORMA**, the ***Service Object Request Management Architecture***, its concept, developement, and implementation examples. SORMA provides an intelligent *"object-bus"* for distributed computing and inter-operation of robotics hardware. It is a software framework for rapid development of object-oriented software modules and their integration into stand-alone and distributed applications.

SORMA was designed to meet the requirements arising from a large set of specialized robotics components in a university research institution (see also SFB360-TR-96-4). Our experiences show, *(i)* that many robotics experiments and applications typically have been a "one-of-a-kind" process, where often the software was developed from scratch, even though much of the code is similar to code written for other applications; *(ii)* due to the short life-time of single-usage code, extensive, robust, and verbose exception handling is often sacrificed; *(iii)* early sharing and integration of several complex application components, concurently developed by a team of programmers, needs strong tool support.

SORMA encourages the re-use of components by interactive test, exploration, and usage throughout the life span of a software component. At the same time this user-friendlyness does not impair its real-time capabilities.

After describing the SORMA concept, we report on two hybrid integration examples: The "Bielefeld Robot Checkers Player" and a real-time 3 D tracking application combining vision and force commands.



SORMA

Contents

Chapter 1

A Software Architecture: SORMA

Hardware in “White Boxes”

The experience of several years of building and operating robot-vision labs[1] showed, that a substantial amount of effort easily dissipates in adaption of software components to application specific needs. A lot of ideas and pieces of source code are generated but often they remain a “one-of-a-kind” products and are not economically re-used. They need a suitable, standardized form in order to be sustainable, achieve practical re-usability, and give the basis for incremental work.

For general computing and special support for artificial neural networks a powerful software framework NST was developed in our connectionist research group during the recent years (see below). The next section describes the important experiences gained and introduces the non-standard requirements of a large collection of complex hardware components. Our robotics laboratory comprises among other, an industrial 6 DOF Puma robot manipulator, a hydraulically driven dextrous multi-fingered robot hand, force-torque sensors, tactile fingertip sensors, various active and passive vision systems, as described in more detail in (Walter and Ritter 1996b). Connecting and *inter-operating* these components within a general purpose high-performance Unix work-station environment in a suitable, convienient, and efficient manner was a central aim for the development of SORMA .
The next section explains the problems, that had to be addressed and shows some historic routes.

[1] see e.g. Walter, Martinetz, and Schulten 1991; Walter 1991; Walter and Schulten 1993; Littmann, Meyering, Walter, Wengerek, and Ritter 1992; Walter and Ritter 1996b

1.1 Experiences that led to the SORMA Design

The *Neural network Simulation Tool* (NST) is a software framework developed by Ritter (1995, 1996) for constructing neural network systems from a library of object-oriented software components called *units*. Each NST-unit type implements a particular functionality ranging from single neuron types to neuron layer and entire networks, including also advanced networks types such as the PSOM network (Walter 1996). Additional units implement functions for mathematical, file, and image operations as well as graphical 2 D and 3 D data visualization, and many more. The units have input and outputs (pointer to *float* typed "pins", grouped to "connectors") that allow to connect them with other units and form "circuits" that combine the functions of several units in the desired way (see also Tab. 1.1). This leads to a versatile configurability when the units are instantiated and combines this with a very fast operation of the circuits, once everything is wired up.

The question how to connect the Puma robot to a NST-program that runs on another, faster graphic workstation led to the development of two NST-units: *(i)* a Puma-NST-unit with an interface for position command information etc. and *(ii)* a special NST "rpc-unit", which could mirror the interface of a remote NST-unit as a *proxy*, using *remote procedure calls* for the communication with the remote NST-unit. The remote NST-unit is wrapped in a specially configured process and can run on any other Unix computer in the network.

The NST "rpc-unit" Client-Server Concept

This architecture follows the *client-server* paradigm, where a general resource is made available by a process, the so-called *server*. The server provides a pre-compiled set of remote callable functions to one ore more callers by listening to socket calls bound at a specific pre-compiled and registered rpc program number. The calling *client* process (running on the same, or a different host in the network) calls the remote server procedure similar to a subroutine call. The arguments (structure) are packed-up, sent over the net, unpacked, the server function is called, the results are packed, back-transported and unpacked. All this finds a certain amount of support by the (Sun) RPC, the *Remote Procedure Call* protocol, bases on TCP/IP (Transport Control Protocol / Internet Protocol), see e.g. Bloomer (1992).

The NST-rpc proxy unit solved some of the common problems occuring when using plain rpc calls, including: *(i)* overhead in building a connection; *(ii)* the procedure call table and the interface definition is static (fixed at compile time). This allowed different kinds of NST-unit types to be wrapped and compiled into a server, which follows the idea of dynamic instantiation of multiple remote units of various kinds. On the client side they could be connected as if they were local (see also the proxy object in Fig. 1.6).

It actually worked fine – as long as nothing went wrong – but nevertheless, debugging could become hairy and some problems remained unsatisfying. In the following section, we analyze these difficulties further.

1.2 What is Special in Robotics?

As described before, the Unix process controlling the Puma robot is a special dual-threaded process. One thread is the planning level and may take its time, the other, the robot low-level control task (thread) has *hard real-time* constraints: *(i)* hard dead-lines (available time <10 ms, non-graceful with no time fault tolerance) with *(ii)* certain system calls restriction (e.g. allocation of virtual memory is here forbidden, since it might require memory paging to disk). It might not be obvious to non-experts, but

synchronous robot control is an ***ongoing decision making process*** .

Involved *risks* are as real as the actuation happens in real-world. This makes a huge difference between *real-world actuation* and virtual reality *simulations* (or pure perception and cognition tasks). These decisions are potentially fatal to man and machine. They require a lot more care and consideration whether an uncertain information is acceptable or not.

"*Safety first*" translates to "*in dubio pro stop-and-exit*". This works great and makes experimentation really safe, simple, and reliable. Safety circuits everywhere, in form of hardware (stoppers, brakes, power electronics, force sensible table etc.) and software (consistency test, checksums, range checks etc.) are distributed in many layers and silently help to watch for trouble.

Particularly helpful are those watchdog circuits, which are able to *tell* what went wrong before they engage into one of the shutdown procedures. This includes three aspects:

(i) *generating* a meaningful signal (containing information);

(ii) *message transfer* to the user (that he becomes aware of the event);

(iii) *presented* in a form that the human *can quickly learn* about the problem – with decent effort (the error code 42 or a core dump may contain the hint of the day).

Aspect *(i)* concerns the circuit author, he will probably be nice and generate a signal – if *(ii)* the *system architecture* does provide transport *and* he can expect that the user/operator has interest in learning about the problem *(iii)* (and the sales fellow doesn't veto).

When designing a more general, reusable software interface for operating a robot, the following aspects need particular attention: *(i)* a helpful robot interface needs extensive consistency checks on a high level of abstraction, in order to *generate* helpful information on possible command mistakes. Lower-level safety switches are too silent and non-recoverable. *(ii)* The *transport* and *(iii)* delivery of exception messages must be assured. *(iv)* Murphy's law works once-in-a-while and *(v)* added consistency checks *(i)* is imperfect - particularly in the phase when new code gets developed. In the last two cases the Unix robot control process will terminate, which has the consequence that the state of this process is lost. Any program for gathering data or making long-time tests should be prepared not to loose many valuable data etc. One solution is to make the (single process) user program *restartable* from a back-up state previously saved. The other solution is to split the task into a (save) user program and the robot serving process. The Unix operation system (OS) is employed to set up *firewalls* against program crashes (in the robot process). When using RPC this works across the Internet (client–server concept, see glossary p. 60).

The really hard problem starts later, after a possible "fire" is over. Assuming the problem cause could be solved, the robot serving process is started again, and the robot starts an initial motion into the park position waiting since for new client requests. We need the ability to *resume* the over-all task, which means in the client–server architecture to *re-connect*. One part is the *management of the rpc connection* (see below), the other how to proceed with the task execution. The robot and its control task is a rather *complex state machine*. To resume operation in the desired way the expected robot state has to be reached and then the desired operation executed.

Two principal designs appear: *(i)* the interface allows state transition calls, analogous to any robot control language; *(ii)* a complete state information is coded into the software interface.

The consequences for *(i)* are that *each* remote request has to manage the restart case by its own. The resume case includes requesting the correct sequence of state transition calls.

Our first implementation of a remote callable NST unit was a type *(ii)* design. The basic robot configuration was done in a text file, for setting up the more complicated things, (transformations, safety boundaries etc.) the rest was set up in the call interface - restricted to arrays of *float*[2].

The effect could be called *premature code freezing*. Because any expansion of the state and operation options changed the interface definition, it rendered previ-

[2]The NST-unit interface restriction to the type *float* is lifted in the newest NST and NEO releases (Ritter 1996).

ous code too easily useless.

Last not least, research and development in the robotic domain involves a significant amount of *interactive testing*. Automatic test programs are not recommendable in an early development phase (risk). Here we find a gap between the easy-to-memorize symbolic variable and procedure names, which we learned to program - and the run-time optimized coding by numbers. (e.g., floats and opcodes). Therefore, we like to have understandable clear-text test-suites in order to *(i)* resist the decay of the code authors decryption capabilities and *(ii)* to share their value with other users. *(iii)* They should allow to overwrite previous configurations in order to interactively tune parameters - without the need to edit mature configuration files (versioning problem).

1.3 Particular Design Issues for a Software Infrastructure

The above reported experiences were extremely valuable for the complete re-design of a software structure aiming at:

- *rapid* building of applications by assembling components
- rapid development of better *re-usable* building blocks
- wrapping hardware in "*white boxes*"[3], which means that the box can be extended, specialized, and inspected.
- *interoperating* complex hardware components
- distributed computing with a helpful, intelligent infrastructure

Despite the fact that the development started rather spontaneous, it lead to a more general architecture, called SORMA discussed below. It turned out, that it fulfills many generally put up demands for a state-of-the-art *distributed object infrastructure* as those are emerging from industrial software standardization efforts, e.g. driven by OMG (Object Management Group, CORBA) or Microsoft (COM/OLE). SORMA is not multivendor oriented, but takes a lot of care to be 100 % *multiplatform compatible* on all our Unix architectures (see Walter and Ritter 1996b). It can not talk to Winword, but it is *fast* and well tailored to serve the special demands of a robotics laboratory.

[3]The term *white box* and *glass box* is coined by Ivar Jacobsen (Jacobsen and et al 1992) and pictures very nicely the contrast to a closed *black box*.

1.3.1 Ingredients for an Object

The Object Oriented Programming (OOP) paradigm defined an software *object* and its characteristics the following way (see Booch 1991; Orfali et al. 1994): it is a piece of code that owns private data and provides services through methods (=procedures). A *class* is a template ($\neq$ C++) that describes the behavior of a set of alike objects. To be more precise, an object is a run-time instance of a class. The three main properties of an objects are: *(i)* The objects gives only procedural access to the private resources, *encapsulated* by the object. Methods and public instance data are part of the interface, published by the object. *(ii) Polymorphism* means, that the same method can do different things, depending on the class that implements it. *(iii) Inheritance* is the mechanism how sub-classes can be derived from existing parent classes. Data structures and methods can be added or overwritten.

1.3.2 NST and Time-optimal Invocation

How does NST implement this ideas? Despite NST is programmed in the language C, it has all principal features of an strong object-oriented approach. The inheritance property finds classical OOP support only on the first (sub-)class level, i.e. for different NST-unit types. Any further *specialization* can be done at run-time, depending on the arguments given to the instantiation method.

This is a rather interesting and powerful feature: it allows a bit more than the classical *late-binding* of the object oriented programming (OOP) paradigm. Binding refers to the linking of the software interface between two objects. Late, or dynamic binding means, the interface is determined when the message is sent, in contrast to static binding, which is fixed at compile time.

Name of	**NST**	**Unix OS Kernel**	**SORMA**
class	unit type	device driver	service class
boot call	–	init()	register-service-class
object	unit	e.g. kernel inode	service object (SO)
create call	create_unit(args)	*open*(args)	create(name) [or –]
job	*exec*() & *adapt*()	*read/write*(args)	*exec*(args) [or text SOR]
control call	*ctrl*(int)	*ioctrl*(args)	*ctrl*(args) [or text SOR]
misc.	*load/save*(file)		*dbx*(args)
free	*remove*	*release*|*close*	remove

Table 1.1: Comparison of names and methods in three environments: NST, a classic Unix kernel, and SORMA . Despite they are not written in a typical OOP languages like C++, they share the principal concepts of OOP.

Tab. 1.1 lists methods, which each NST-unit class must implement (or inherit). It shows, that the two main functional methods *exec* and *adapt* have no arguments. All required arguments must be given at the public data interface (shared values at "connectors" and their "pins") – or at creation time. E.g. an image processing NST-unit object can operate on a monochrome or a RGB color image representation. The operating behavior is configured at instantiation time and leads to the construction of an object with one or three "connectors" (per image IO). This is a form of *late-binding* or "*run-time compilation*" since the circuit description code not only creates and connects objects, it also dynamically can *specialize* the (subsequently static) interfaces and methods.

The special power of NST is the versatile configurability at unit construction time and the specialized **time-optimal execution** of the "circuits", once everything is wired up. This concept of shifting as many of possible preparing tasks to the first call, can significantly accelerate the execution of the main functionality. How significant this effect is, depends on the amount of particular overhead time, the repetition rate, and the time-criticality of the execution task. In the real-time domain of robotics these cases are given – supporting this feature opens the domain of time-critical sub-tasks.

1.3.3 A Small Interface to the "Service Object"

We tried to combine the good sides of both worlds, the real-time capable and the smart object oriented interactive world. The found compromise is named by its purpose: the *service object*. It shall serve the need to wrap hardware resources in white boxes – a special kind of object. The interface definition is a trade off between flexibility and expandability versus small and strict interface definitions where good interactive support becomes feasible.

Tab. 1.1 lists, next to NST, the interface methods of a classical Unix device driver (Beck et al. 1994). As a run-time optimized interface is shows strong similarities. All three share the idea of separating procedural requests into the designated "*execution*" functionality and another request call, controlling the resource (see column "job" and "control" in Tab. 1.1).

For SORMA it appeared very attractive to foresee *two separate levels of parsing*. One possibly complex and detailed parsing mechanism (clear text - with more "muscles") and a second call interface (*exec*) which can be designed, e.g. to meet real-time requirements (no parser) or any desired level of comfort (see Sec. 3.3). The "CTRL" method should implement a textual parsing of the sent string message in analogy to the list of arguments (`argv`) passed at Unix program invocation. This is in contrast to the "*EXEC*" method, which is unrestricted in its interface design. The advantage and further details will be clarified later.

The *"DBX" method* is an extra inspection method for accessing meta-information from the service object, e.g. version, compile time, verbosity flag, audit trail of "CTRL" method requests. Additionally, the class implementation must provide one function to give a clear-text "dump" of the complete particular data structure encapsulated by the object instance. All other methods are inherited.

As *public data interface* the service object offers a data transport structure. We choose to implement a strong interactive support for a particularly small structure, which is a set of the most important data types, i.e. *string, float[], int[], any[]* (*[]* means 1 D array or vector). Since it includes an "*any*" block (untyped or "opaque" typed, for e.g. image data) is can be converted to arbitrary complex structures, suitable support can be found in XDR - or NDR-libraries (Bloomer 1992; Fink et al. 1995)). Together with context information (from, to, call/return code, and event message "tdbx" string) this transport structure is used in a bi-directional way (see Sec. 1.4.1).
Before we want to proceed in explaining more details in the design of the service by itself, we next introduce the general framework.

1.4 Introduction to the SORMA Architecture

What is the advantage of using an object? The essential argument is, "the code gets re-usable". Fine – but in a university environment, not too many long-time software programmers are around, who experienced and internalized the need of re-usable code. Many things are done and re-done again. Even when code is intended to be reused, often only some fragments survived. And, too many good ideas get thrashed when the authors leave.

Therefore, we need - beside a decent object structure, a good argument for convincing a non-acquainted user (programmer), to comply to a standard interface structure. The cost-benefit view shows always the initial effort spent to learn the rules and usage. Here SORMA offers the *incentive* of immediate supporte for a ready service object class in four scenarios, shown in Fig. 1.1. After source code compilation a class implementation can be linked to:

(i) a program (stand-alone or mixed client);

(ii) a server stub which produces a SORMA *server* - in the following called by the synonym "***daemon***" (in order to better discriminate between *server process*=daemon and *service*=functionality in a *service class* and its instances, the *service objects*.) This requires the little effort of a inserting a few lines into a configuration file and a makefile (copy-paste-modify);

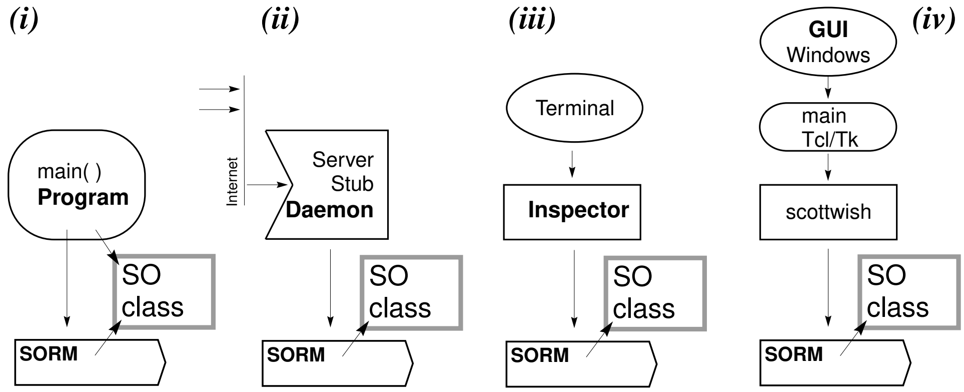


Figure 1.1: SORMA services instantly find support in four process configurations:
(i) Stand-alone process, which is a regular **program** with local calls;
(ii) server or **daemon** process for shared network access from other application clients and standard SCOTT inspectors;
(iii) interactive local (and remote) service object requests (SOR) are conveniently facilitated in the line oriented **inspector** mode.
(iv) The SORM-Tcl/Tk coupling allows easy configuration of an application specific **GUI** (Graphical User Interface) to communicate to local (and remote) services. Note, the standard tool executables "scott" and "scottwish" (self type *(iii)* and *(iv)*) are readily available to remotely test and use a type *(ii)* configured service. Therefore, in many cases one single daemon configuration is entirely sufficient (for short self tests, a type *(iii)* inspector is a run-time option to each daemon, by default only a lean version without "scott's" extended command line editing, completion, and history features).

(iii) command-line oriented *inspector*, giving full access to all built-in (and remote, daemon-provided) services. This feature offers immediate test-suite comfort, see Fig. 1.1 and Sec. 1.8.1.

(iv) graphic user interfaces (GUI) are designed by Tcl/Tk scripts, which allow easy request to built-in (and remote) services by simple mouse clicks on screen buttons, sliders etc.

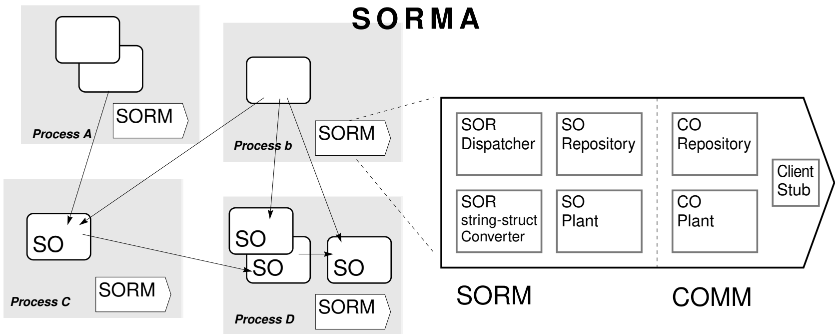


Figure 1.2: The core of **SORMA** , the *Service Object Request Management Architecture* is SORM and COMM, with development support by SCOTT (Service COmmunicaTion Tool). The depicted SORM comprises several components, which are kept in this iconic arrangement for detailed explanation in the following pictures.

1.4.1 Some SORMA Vocabulary:

Service Object Request Management Architecture: SORMA is a framework and infrastructure to communicate to dynamically created Service Objects (SO) upon SORs (see below). This structure is also called “object bus” or also distributed “field bus”) See Fig. 1.2.

Service Object Request Manager: The ***SORM*** is responsible for efficient *(i)* building, *(ii)* maintaining, and *(iii)* sharing of multiple Service Objects (SO) – efficient w.r.t. to *(i)* time and *(ii)* memory.

Connection Object Management Module: The ***COMM*** communicates requests to external, distributed SO served by daemons - on the same host or elsewhere in the Internet network (it is a module, since a stand-alone process may not need it.)

Service COmmunicaTion Tool ***SCOTT*** offers development and usage tool support for services, which are configured as SORMA daemons (see Sec. 1.8).

Service Object Request: A textual service object request ***(SOR)*** is a single string, containing three main parts:

$$"SO_{name} \quad TOKEN_{method} \quad body"$$

where the selector $TOKEN_{method} \in$ {`EXEC`, `CTRL`, `DBX`} uniquely determines the called method type (see Sec. 1.3.3. The request $body$ translates to the optional variables in the transport structure – segmented by reserved tokens:

"[`TMSG`] [$text$] [`FMSG` n $f_1 \ldots f_n$] [`IMSG` m $i_1 \ldots i_m$] [`TDBX` $message$]"

Return values can be converted back to a text string, the '$TOKEN_{method}$' is then replaced by the token pair '`RET` $i_{returnCode}$' (identifying the success or exception identifier). This facilitates bi-directional conversion of the transport structure (excluding the untyped *any[·]* message).

1.4.2 SORMA Instantiation and Invocation

A sequence of illustrations shall explain: how service objects are instantiated and invoked; what the two principal request techniques are: sending a string message request or by direct call-by-reference (previously gained by a reference-by-name request); the remote service object request management is illustrated, and an example given, how two SORMA clients share resources on two daemons. Please consult Fig. 1.2 - Fig. 1.7 and the supplementing explanations.

The SORM Manager itself allows introspection by implementing a special SO with a common interface. By this means, the SORM can be instructed and queried about state and capabilities. SORMA network agents become possible.

1.4.3 Service Object Names are Unique

Each Service Object (SO) is associated with a unique name. To facilitate the SO-creation and repository mechanism the full SO-name is constructed in such a way, that the service class sc and possible remote daemon address can be quickly extracted:

#	Syntax	Example	SO type
L	$sc[.flavor]$	$foo.a$	local SO name
R_l	$sc[.flavor]@host{:}num$	$foo.a@druide{:}11$	remote daemon explicit
R_a	$sc[.flavor]@hostAlias$	$foo.a@d1$	remote daemon alias
R_d	$SC[.flavor]$	$FOO.a$	remote daemon default

Local SO-names (L: with lowercase class name and without remote extension) identify a locally served SO - this instantiation is described in the next section.

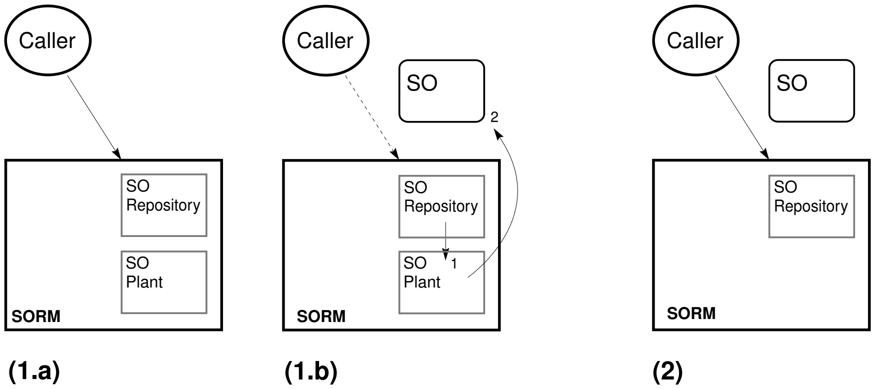


Figure 1.3: The SORM builds and maintains Service Objects (SO) by unique names. Here the *get-SO-by-name* call is illustarted.
(Left two:) If the repository does not hold the requested SO-reference, the plant is engaged to dynamically instantiate the service object (see also Fig. 1.8).
(Right:) Further requests to the same name will be referred to the same SO (shared access).

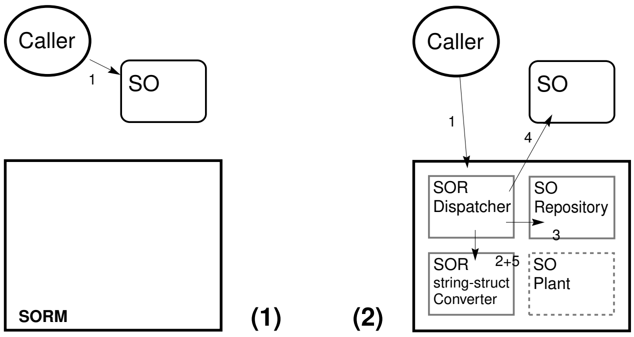


Figure 1.4: SORMA offers two alternative ways of calling a service object (SO): *call-by-reference* and *request-by-name*.
(Left:) call-by-reference is the **time optimal invocation** of a SO and is, beside a debug hock, essentially as fast as a sub-routine call, since all arguments are referenced as well. It requires a previous get-SO-reference-by-name call (Fig. 1.3) to receive the SO-handle.
(Right:) request-by-name is a **textual, one-step invocation**. The *Service Object Request* (SOR) is a message containing the SO-name, request type, and all arguments. The text message is send to the dispatcher (`strDispatch()`, 1), gets converted (2) and send to the SO (4) found in the repository (3). If the SO does not already exist, the SO-plant will create it (not shown 3.1, 3.2 = step 1, 2 in Fig. 1.3). Finally, all results of the SO-call (4) are converted into a text message (5), which is suitable to human readers as well as text oriented languages (e.g. Tcl).

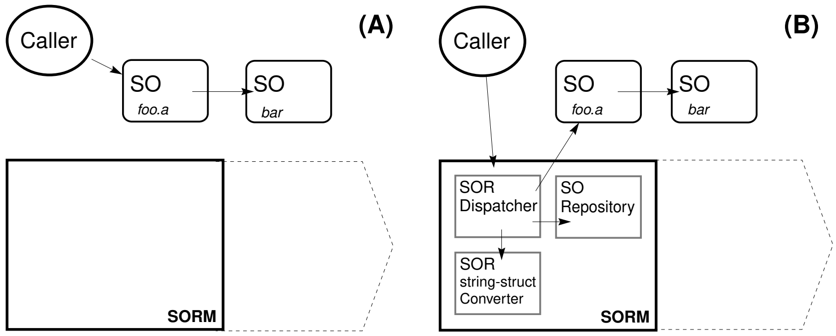


Figure 1.5: Hierachical Service Object call. A SO calls a sub-service, *(A, left:)* by reference or any mixed: by reference and by name *(B, right)*. Each SO carries a unique name. The dashed lines indicate the optional COMM module, which facilitates the remote SO invocation, see Fig. 1.6.

The last three types are variants of a SO request to a (R_l, R_a, R_d) remote daemon. The connection object management (COMM) needs the information where to locate the daemon in the Internet $host$ and which port number num the process has registered. The SORM at the daemon will process then the (there local) request.

The latter two address specifications - by alias (R_a) and by default (R_d) - are opaque network addresses. These daemon locations need not to be known at programming time. They are resolved at connection create time (honering also the location name "*self*"). They are the hooks for a central, or de-central *"object bus" configuration mechanism*. Name resolving could be used, e.g. for load balancing and object trading (see discussion). Currently we delegate this task to the individually configurable *dictionary* service object, described next.

1.4.4 Common facility: The Dictionary SO

One common service class is the dictionary. Currently, one special object instance, named dict, is employed for *(i)* daemon name resolution and *(ii)* as part of the service instantiation procedure, as described below.

The dictionary is essentially a service to table look-up a single index word and expand it to a sequence of words.

$$indexWord \rightarrow \ list\ of\ words$$

The table can be filled in various ways: *(i)* entries from process environment variables; *(ii)* explicitly; *(iii)* loading from one (or more) text file with the simple line format "$indexWord \quad list\ of\ words$". It facilitates *recursive definition* of entries by index word substitution (marked by keyword or special character, at load time).

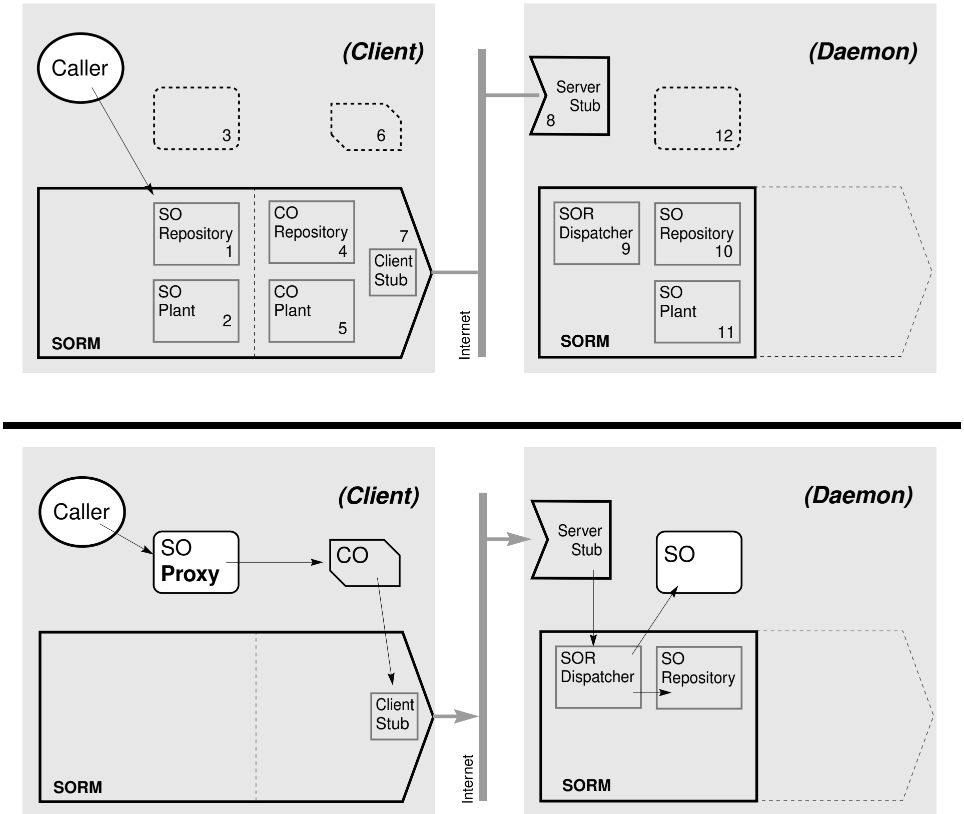


Figure 1.6: Creation and later usage of a remote Service Object by Proxy SO.
(Upper scheme:) The first get-SO-reference-by-name call to a remote Service Object (SO) leads to a number of steps. Since the client SORM fails retrieving the requested SO in the SO-repository (1), the SO-plant (2) instantiates (3) a new proxy SO. The necessary daemon communication is handled by the *Connection Object*, CO (6), which is built, if not already available (4,5). The SO request reaches the daemon SORM on the right side via TCP Internet protocol and stubs 7 and 8. There, the get-SO-reference-by-name induces the instantiation of the desired SO (10, 11, 12 as in Fig. 1.3)
(Lower scheme:) Requests to the remote service object look identical to local SO-requests – since they *are local* requests to the **SO-proxy**. The SO-proxy **completely mirrors** the remote SO w.r.t. functionality, exception messages, input, and output. The proxy does *not* mirror irrecoverable daemon failures, which facilitates **protected invocation**. Instead, its connection object and the COMM, the *Connection Object Management Module* would signal the problem and try to *semi-autonomously reconnect* to a newly started daemon.

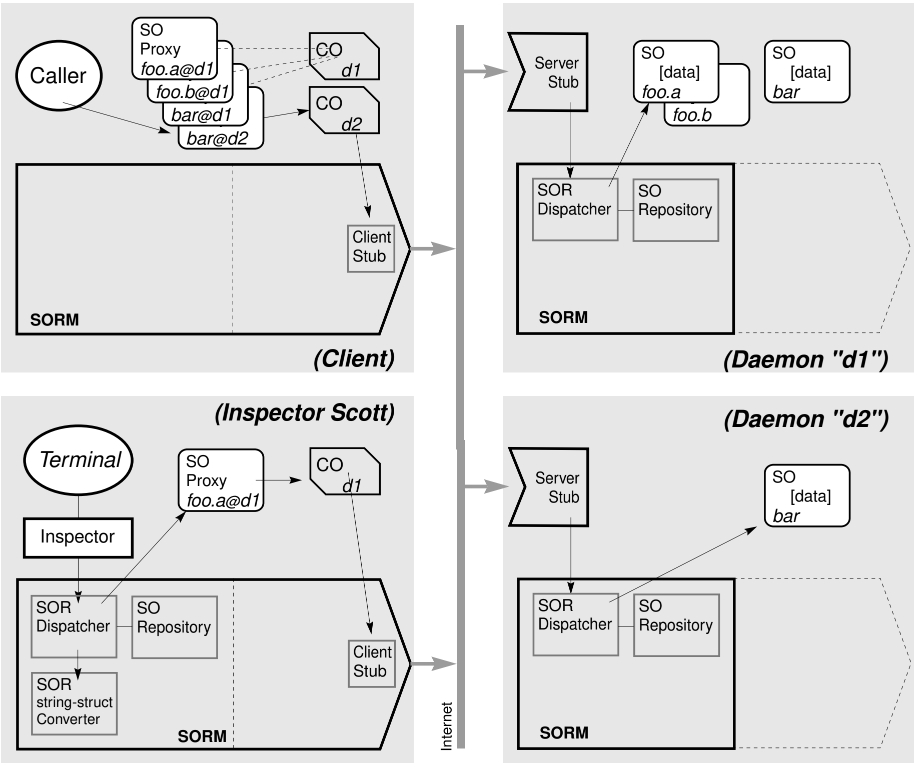


Figure 1.7: The SORMA (Service Object Request Architecture) allows *protected and shared usage* of remote services, distributed in the network. Here, an example with two processes (left side) concurrently requesting remote Service Objects (SO) on two separate daemons ("d1" , "d2" , right side). *(upper left:)* The client processes has already built four proxy SOs using two connection objects (CO). The request to the SO proxy with name "`bar@d2`" is routed to the SO "`bar`" at daemon "d2" on the *lower right*. The connection object maintenance is responsibility of the COMM (Connection Object Management Module) and more efficient by bundling communication requests, normally per daemon. *(Lower left)* The SCOTT inspector grants interactive service object requests to all existing (and possibly creatable) remote SO. E.g., the request "`foo.a@d1 CTRL -help -dump`" (typed to the terminal) gets routed to "`foo.a`" at daemon "d1" *(upper right)* and will return the on-line help and a report on all data, encapsulated as indicated in this particular service object (both methods, called by "`-help`" and "`-dump`", are mandatory to all service class implementations).

The load file structure is un-complicated and allows to take care on *good readability* of entries. Therefore, it can cope with comment lines (also part line comments after #), long multi-line definitions (backslash marked), and quoting of words containing white spaces.

Since the dict is a regular service object, it can be contacted via a textual request, which allows querying, adding, removing, reloading, saving etc. of the dictionary entries. Consequently, the SORMA architecture facilitates to remotely edit the dictionary of another network daemon – at run-time. The next section will underline the potential of this feature.

1.5 SO Plant: *"Service Flavors"* on Demand

Combining: "Shared" + "Easy Configurable" + "State-free"

One of the key design points of SORMA is the concept of service object instantiation on the basis of a *unique* name. The name gets *expanded into a list of object configuration and state transition calls*. Different names lead to the instantiation of different service objects. Since the resulting SOs belong to the same service class but usually exhibit modified characteristics, we call them also **"service flavors"**.

This name-to-configuration-instruction-list mechanism can be considered as a form of *built-in scripting*. It solves the conflict between the three (before) conflicting goals of *(i)* serving hardware by *shared* server access, *(ii)* which is *easy-configurable*, *(iii)* but still *state-free*. The script is associated with the service object name, which is part of all initial SO reference calls, and textual SORs (see Fig. 1.4). For remote SO calls, it allows the COMM (Communication Object Management Module) to efficiently take care about not only building, but also about re-establishing connections to the desired service flavors, served by other SORMA daemons. The following example illustrates, how the SO name determines a particular service flavor specialization.

As indicated in Fig. 1.3 and detailed for the request example "$foo.a$" in Fig. 1.8, the SO-plant calls the registered SO class constructor function (class name foo) by the full local service name. The common dictionary service resolves it and supplies the desired list of words. Those should be parsed by the "CTRL" method implemented by the object class.
Assume the dictionary service contains following entries:

foo.a → -breakfast -where kitchen -tempo presto
foo.b → -breakfast -where garden -freshJuice -tempo adagio
bar → -tea 1.5 -ice crushed -lemon 0.3

and the service "CTRL" method knows what to do with those tokens. Requesting

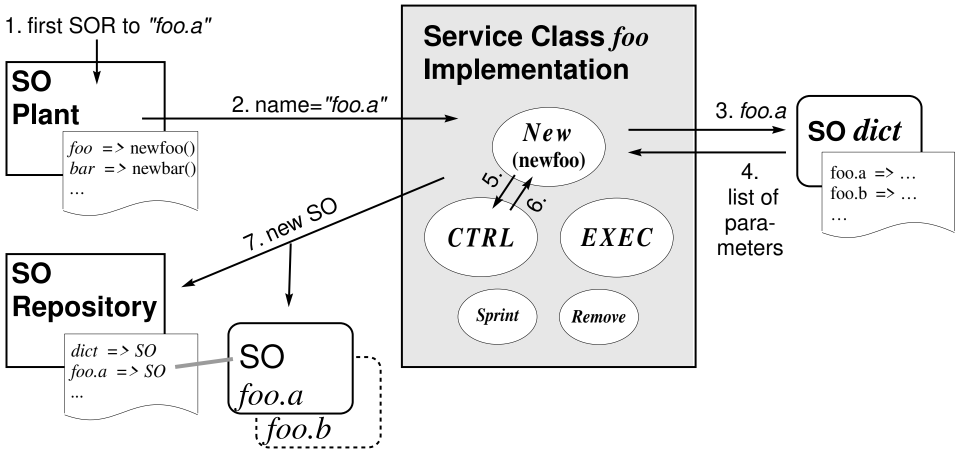


Figure 1.8: The service object instantiation and specialization procedure when the SORM is called for the name "$foo.a$" the first time (step 1b-2 in Fig. 1.3). The registered class "foo" constructor function gets invoked "newFoo("foo.a")", which retrieves the "recipe" from the dictionary service. The "recipe" is a token list, denoting a sequence of state transitions. It gets parsed by the object's "CTRL" method and specializes the desired service object flavor.

a service object instance $foo.a$ will create an object, which has already parsed the token list "-breakfast -where kitchen -tempo presto".

The semantics of these tokens is subject to the interface definition of the implementing class, here "foo". There are many ways to suite special needs of an application. To give a brief idea consider these opportunities: *(i)* when $foo.a$ is called it is clear that the tempo presto is requested. In case of re-connecting to the process, everything is said, a "*EXEC*" call for $foo.a$ is fast. (*state memory*) *(ii)* The $foo.a$ requester can change his mind and request a "CTRL" call "-tempo piano", affecting all future calls to $foo.a$ (*state change*); *(iii)* Alternatively, on special occasions, the client calls for an other favorite menu $foo.b$, then $foo.a$ and $foo.b$ can be used interleaving, without affecting each other (*parallel states*); *(iv)* If there are multiple clients, private object instances can be generated by duplicating the recipe of the service with a unique flavor name (*non-shared*). *(v)* If desired, *resource locking* may be implemented by the class creation method.

The flavor instantiation procedure by names and stored parameter sets can be associated with the process of "cooking": *"Expert cooks"* (supplied by the class implementation) prepare object flavors on demand (SOR) following the *recipe* kept in the *cookbook* (dynamic table by dictionary).

In particular, the following aspects appear valuable:

- The dictionary offers a compact format of defining a set of *reentrant states* – a feature which *facilitates a robust, failure-tolerant* state machine server on the required level of granularity (state-free robot serving, see Sec. 1.2).

The initial parsing procedure can be used

- for *configuration* of the SO
- for *method specialization* of the SO (reversible and irreversible, e.g. select different execution methods)
- for *time-optimized invocation*. The two-step service usage allows data preparation (pre-allocation and pre-computing of data structures etc.)
- as a powerful and convinient *default mechanism* to initialize the internal data structure of the service object flavor (multi-purpose, multi-user, with recursive composition)

The NST-concept of "run-time compilation" clearly appears, but here, the concept is more soft – the interface definition may pre-compile - and re-compile - its structure but can also operate as pure interpreter. The, service designer may gradually "shift" between these two concepts.
Furthermore,

- keeping a large variety of flavors in stock is a matter of inserting suitable "notes" in the "cookbook" file (with the recursive word substitution in the dictionary and a overwriting parser, sometimes storing just the differences is enough). It is *memory efficient*, the service object is created on demand;
- the "CTRL" parser mechanism is *double useful*: after the initial service configuration the service may be *re-configured* by re-using the same parser structure. This facilitates clear-text configuration and property modification at run-time;
- the name of a *sub*-service object can be part of the configuration. This strategy allows to *program-by-name-picking-a-la-carte*: by exchanging a single sub-service name, the associated service configuration and "behavior" is changed. This includes swapping of flavors as well as service classes (requiring equivalent interfaces);

1.6 Error Propagation

As already mentioned before, the general need for useful debugging and error information - generated *and* signaled back to the user - becomes a necessity in a distributed operating system. They can guide to the needle in the heap of hay. Unfortunately, "software hay" sometimes exhibit emergent sharpness when smaller, smooth looking heaps are put together. Therefore, the value of decent error reporting increases progressively with application complexity. In a development environment, this point can hardly be overestimated. From experience we'd like to generalize and recommend: *Never judge software robustness by the absence of error messages – better check how meaningful generated warning messages are.*

When the software is used in a distributed way, the programmer needs extra support. If it is not available, the programmer gets frustrated: what to do with the message in case the call is local or remote? Send to whom and how? Messages sent to the standard error channel are unreliable, they may stay invisible and get lost.

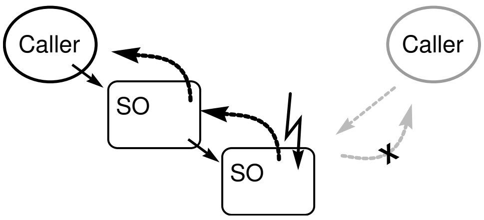


Figure 1.9: All occuring error, or any other message, generated by a service, is propagated back to the original caller. This is achieved in a cumulative way and across process and machine boundaries.

It is therefore important to enforce an error handling procedure. We found the following useful:

- unified procedure - identical for locally and remotely called services;
- clear-text messages (human readable), but also unique error and warning codes (symbolic processing), context information (source file, line);
- discriminate three message classes: *(i)* verbose, *(ii)* warning, and *(iii)* error messages (determined by error code);
- accumulate messages and send them back to the calling user - also across process and machine boundaries (see Fig. 1.6);
- programmer-friendly: easy-to-define a prototypical error message (definition may be placed in the header file or anywhere in the source code, they are semi-automatically integrated to a central table of errors); easy-to-invoke (single macro) with optional specialization and text additions.

1.7 Remote Service Calls and Parallel Processing

Remote service requests in SORMA are based on the remote procedure call, which is intended to behave like a local request (proxy). The disadvantage is, that the calling client process blocks until the daemon call returns. This means, there is only one (jumping) token of activity in a net of such connected processes (not talking about anyway concurrent processing in various special hardware devices - which we actually do want to interoperate (see Walter and Ritter 1996b). In a single processor machine, this is perfect, since the operating system scheduler will take care, that computing time is not wasted.

But still, in order to harvest the power of elsewhere potentially available computing resources we must organize *asynchronous interprocess communication*. Several canonical ways are possible: *(i)* Queuing requests and results (by message passing with or without shared memory); *(ii)* non-blocking rpc request and callback from server; *(iii)* deferred synchronous request (call twice); *(iv)* giving up on the idea of a called procedure using, e.g. stream based architectures (see also (Fink et al. 1996));

1.7.1 Threads

Splitting into sub-tasks, one for central processing and others for asynchronous remote communication management also need inter sub-task communication and synchronization. Modern approaches simplify this by offering *threads*. Threads are sometimes called “light weight” processes, sharing the same process memory space and other resources. They can not magically accelerate any computing performance (except in multi-processor systems supporting parallel thread dispatch), and they can not solve the possible necessity to wait for communication IO (input/output). What a thread library does offer, is programming support for performing process relevant operations during IO-wait states. This simplifies organization of multiple blocking IO connections. Within one process several sub-tasks are distributed to independently scheduled threads. Since they share the same resources, inter-task communication is simple, fast, but non-protected. Efficient programming with threads, requires detailed provisions to keep system consistency when resources and functions are utilized “simultaneously” by parallel threads (“thread-safe” programming).

As in any parallel computation system, possibilities for socalled *race conditions*, *dead-locks*, and *starvation* have to be carefully considered. Debugging of these run-time failures become very soon very complex and difficult (combinatorial explosion of cases). Furthermore, Schrödinger's problem for observing a quantum-mechanical system re-appears: here the debugging disturbs the real-time performance of the observed system. Here, the interaction frequency scale and the granularity of parallel sub-systems play an important role.

Since, the efficiency of sub-task switching on blocking IO channels involves tight cooperation with the OS kernel, the peculiarities of the different vendor's OS severely delay the standardization. We did not base the SORMA implementations on threads, simply because *portable* threads are not yet available on *all* our Unix operation systems, which we want to see cooperating: SunOS, Solaris, Iris4d, Irix5, Irix6, OSF1, Aix, NextStep, and Linux. E.g. the Posix "P-threads" standard is very recently released thus we can expect more and more (fully) reliable implementations without constantly floating definitions as seen is the past. manger

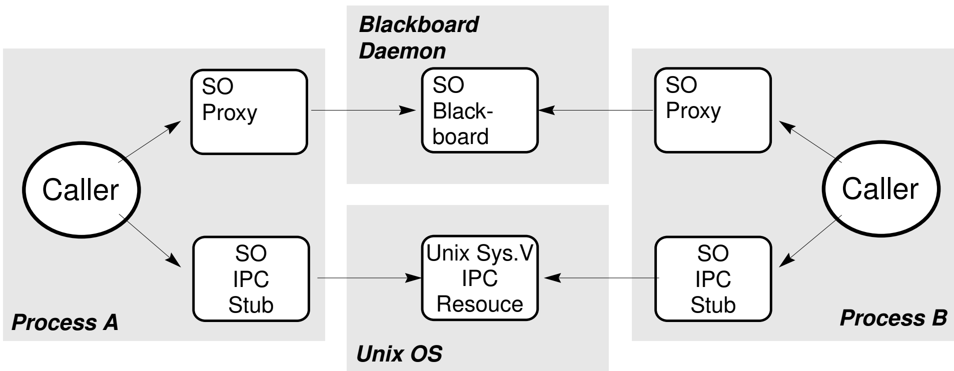


Figure 1.10: SCOTT facilitates Inter-Process Communication (IPC) by shared access to *(upper branch)* a mediating third daemon serving as black-board and *(lower branch)* general Unix IPC resources. Specialized IPC service objects simplify the common access by structuring the Unix resources and configuring access keys. Note, that IPC resources are bound to the local machine – however, SORMA yields network access by employing a local "scott" daemon on the target host via the proxy mechanism.

Fig. 1.10 shows the principal SORMA solutions for asynchronous Inter-Process Communication (IPC) for regular "heavy-weight" processes. A third SORMA daemon can be employed for doing a central blackboard service for multiple callers. For processes running on the same host, the Unix kernel implements (at least) three IPC paradigms: *(i)* semaphore service (flags with atomic read-set operations) *(ii)* message queuing, and *(iii)* shared memory access.

By offering specialized service classes, the usual "heavy" set-up effort can be turned to easy configurable service objects. These IPC stubs take care on agreeing on access keys, and structure the raw resource in a specialized manner.

Since they are SORMA services, they are immediately available to remote clients, when a suitable daemon is started (e.g. "scott"). By wrapping the host specific IPC resources in a "white" SORMA box it can be accessed from anywhere in the network.

1.7.2 IPC Example: Cyclic Snapshot Buffer in Shared Memory

One example of the SORMA IPC-stub concept is the service class implementation (*shmem_w*) working on the System V shared memory resource. Here, the emphasis is the rapid (time-optimal) one-way transport of "snapshot data". Sec. 2.2 will report on a real-time application which uses this building block in several places (Fig. 2.7).

The shared memory segment gets pre-structured depending on the specification of the writer service object. This structure information is stored in the memory segment header and available to any reader-SO.

Here, the concept is a *snapshot multi-buffering*, which means a certain number of blocks (snapshots) find place in the (pre-allocated = fast) shared memory segment. They get written in cyclic order. This allows a configurable amount of asynchrony of process, which desire to communicate either *all*, the most recent, or the *recent history* of snapshots (note, that this includes single-, double-, triple-, and multi-buffering).

Snapshot data consists of: *(i)* a data segment (of type *float[]* and type *any[]*) with (configurable) constant size; *(ii)* a counter, and a *(iii)* time stamp is automatically appended, when a fixed sized record (snapshot) is written to shared memory. This allow any reader to determine the age of each record.

The IPC-stub service classes for writing and the class for reading offer a certain amount of error checking and interactive monitoring (clear text formatting of snapshop sequences etc.)

Besides the purpose of interprocess communication this service turned out to be very valuable for a *plug-in* type *process monitoring* as well as for off-line *"flight recorder"* analysis. When using a time optimized exec-method variant (non-checking), the SO is fast enough to be used for debugging real-time code (signaling passed source code lines) – a task which is otherwise, at least in our particular case, rather difficult.

1.7.3 Real-Time and Invocation Performance

Real-time task are tasks with time constraints being part of their specification. *Hard real-time* tasks have to fulfill specific dead-lines, while *soft real-time* tasks require "best-effort" to meet the performance specification. A further main consideration is the *time-fault tolerance* of a system. E.g. a processor overload situation might compromise the keyboard response time (soft real-time) or might delay an emergency-stop signal to a moving robot manipulator (hard real-time). Usually, Unix system are suitable only for hard real-time tasks on a time scale larger than 100 msec. Special effort is necessary to employ a Unix workstation for robot real-time control (see Walter and Ritter 1996b) as required for the tracking application described in the following chapter.

Depending on the time-criticality of an application the speed of communication can play a crucial role. Here SORMA has the big advantage to offer *service migration* for the purpose of load balancing and in particular to improve performance by choosing time-optimal invocation. As pointed out earlier, this can be as simple as re-linking program code and redefining service locations in the dictionary file (opaque addresses, see Sec. 1.4.3).

#	**Invocation type**	min-max	unit
F	function call	0.11 – 0.45	μsec
TO	execSvc(@self)	0.25 – 1.5	μsec
P_{AA}	execSvc(@localhost)	0.97 – 3.8	msec
P_{AB}	execSvc(@otherhost)	1.5 – 4.0	msec
S	strDispatch("nop@self EXEC")	0.08 – 0.56	msec
S_{+10f}	strDispatch("nop@self EXEC FMSG 10 ..")	0.20 – 2.0	msec
M	execSvc(shmem$_w$)	12 – 98	μsec

Table 1.2: Average invocation time performance (on a set of hosts). Note, the speed of the time-optimal invocation (TO) is close to the function call speed (F) and is more than three orders of magnitude faster than the protected invocation (μsec and msec).

We measured the range of mean request-reply communication overhead times on and between several Unix workstations (including processor configurations with Sun Sparc 2 & 20, DEC-Alpha 4x21064, SGI Indigo R4400 & 2xR8000, Intel Pentium 90 & 2x160 MHz). The recorded time ranges are average real times taken with various processor types and load levels.

Table 1.2 shows, that the speed of the time-optimal invocation (TO) is close to the function call speed (F) and more than three orders of magnitude faster than the protected invocation (P). Those interprocess communications within one machine (P_{AA}) and across the internet (P_{AB}) account for a latency in the order of three milliseconds for the entire round-trip. Using the textual SO request method `strDispatch` (S) requires for the plain string conversion service a fraction of a millisecond, which significantly increases with the number of converted values (S_{+10f}, here in both directions). The swiftness of the IPC-stup for writing to a shared memory segment (cyclic multi-buffer, see Sec. 1.7.2) is measured with a few 10 microseconds. Here a non-checking, accelerated execution alternative exists, which reduces the IPC service call time to about 65%.

1.8 SCOTT : Service COmmunicaTion Tools

As pointed out before, strong *interactive support* is one important goal to enhance over-all software reuse efficiency by reducing the effort spend to figure out the component. It is associated with:

- interactive testing of the *entire* interface
- easy configuration (*tuning*) of limits and parameters etc.
- *interoperation* of remote objects ("white hardware boxes")
- *inspection* of configuration and properties
- error reports and warning reports (readable)
- ease of use – invites to re-use
  - non-cryptic, semantic messaging
  - readability of results
  - offering also high level of abstraction
  - on-line help (implemented by each service class)

As Service COmmunicaTion Tools (SCOTT) two readily configured executables "scott" and "scottwish" are available.

1.8.1 "scott" : the Inspector

"scott" is the standard SORMA "*inspector*", already indicated in Fig. 1.1. This command-line interface is similar to a shell (standard input) and allows to send textual service object request to the SORM. This is just enough to

- *interoperate all remote daemons* and their (possible) service objects in the SORMA architecture;
- scripting (e.g., for extended test sequences, little applications)

For convenience, "scott" offers shell features, like line editing, history saving and matching, command line completion, etc.

1.8.2 "scottwish"

The SORMA textual SO request technique invites to use services by scripts called from buttons, sliders etc. The Tcl/Tk package facilitates graphical user interfaces (GUI) in a very simple way. As shown in Fig. 1.1 a short Tk-script calling the ready "scottwish" will be enough. Fig. 1.8.2 shows an example where mouse clicks on widgets interoperate an active stereo camera head and its lenses.

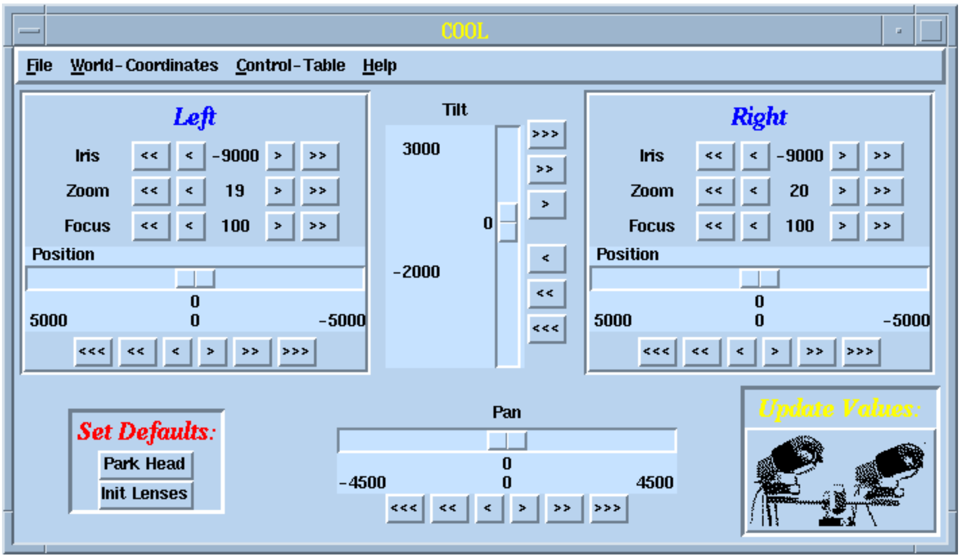


Figure 1.11: Example of a graphical user interface (GUI), produced by the "scottwish" and a Tk script (Kubisch 1995).
(Left:) The active stereo camera head offers 4 DOF for head gaze control (neck pan, tilt, and vergence) plus 3 DOF in each motor lens system (focus, aperture, zoom), interfaced by SORMA services, served by a single daemon process (type *(ii)* configuration in Fig. 1.1). Thus the actice vision head can be interoperated by mouse clicks and drags (using the GUI above), by the command-line inspector "scott", or directly by any application.

1.9 SORMA – Bridges

1.9.1 SORMA – NST Interactions and Graphical Programming

Other systems may benefit from the SORMA features by bridges. Since the SORMA development started, NST was extended to support strings and was further supplemented by **NEO**. NEO is a graphical interface, which allows to visually program applications in an icon drag-and-drop fashion. The data flow is specified by line drawing with the mouse. By NEO's visual programming language NST circuits with unit elements can be drawn, wrapped-up in container units, which subsequently become themself regular, connectable NST unit objects. Starting from a large number of powerful neural network, computing, and graphical animation units increasingly complex modules can be arranged and interactively tested and used.

By means of a general SORMA-by-NST unit *all* network accesible SORMA services (deamons) are usable (e.g. sensors systems) and can be included and graphically wired up in NST/NEO circuits.

Vice versa, in a second step, any of the avilable or created NST units can be packaged and exported to the network via a NST-by-SORMA daemon. In this way the high-performance computing facilities can be easily integrated to large applications. Sec. 2.1.3 will give an example on using a PSOM network service as a central, shared SORMA resource.

1.9.2 Exporting SORMA services via the DACS Bridges

G. Fink and N. Jungclaus (1995) developed the *"Distributed Applications Communication System"* ("**DACS**"), which offers a set of advanced socket based inter-process/inter-application communication protocols. Building on the message passing paradigm, DACS offers processes bi-directional communication procedures like synchronous and asynchronous rpc, as well as by *demand streams*. Demand streams are a new interesting method and allow processes to *subscribe* to the results of other processes. Each subscribed process receives his private result copy after the newly produced data structure is available. The intention is to let several processors contribute their best by parallel working on a stream of e.g. sensory data (like speech data, see Fink et al. 1996).

Data structures are dynamically packed and unpacked by NDR (Network Data Representation) library functions. This allows, in contrast to static, pre-compiled XDR (see Glossary) – standard network rpc library routines (Bloomer 1992) conversion routines, to pack/unpack arbitrary complex structures in a lisp-like, type tagged list representation.

Similar to the PVM (Parallel Virtual Machine) concept, DACS employs a (for the entire network) central communication daemon for dynamic maintenance of the communication structure. This is a trade-off between centralized control allowing immediate communication re-configuration – and the extra communication overhead, which becomes substantial in applications with high-communication frequency and/or with real-time constraints[4].

Since DACS employs advanced, but not (yet) fully portable threads (see Sec. 1.7), SORMA has difficulties to interoperate DACS applications. On the other way around, the bridge is very simple – services in SORMA are *fully usable*, also via DACS.

1.10 SORMA Features

The *Service Object Request Management Architecture*, SORMA is a distributed object oriented programming approach implementing an "object bus", suitable to serve the special needs occuring in the robotics domain. It emphasizes:

- OO "White boxes" for hardware – offer a high level of abstraction, without loosing access to internal affairs (in contrast to a black box).
- Time-optimal invocation – allows full real-time efficiency required in the domain of robotics.
- Protected invocation – allows interprocess service requests from process to process also across the network. No server faults are transmitted to the caller in an uncontrollable way.
- SO migration – both invocation schemes are fully exchangable (same syntax).
- Interoperability of distributed objects – all operations at local service are fully network transparent. The proxy mechanism mirrors also possibly generated event messages.
- Dynamic service object instantiation by name – service objects are created (by class name) and specialized by "flavor recipe", which is done by parsing a list of state transition requests. The same parsing mechanism is later available, e.g. for interactive tuning.

[4] Fink et al. (1995) report 3.9 msec for the best DACS communicated case where the caller, the callee, and the central daemon are running as three processes on the same host. Cummunication takes 9.1 msec for the worst case, when three different hosts are involved. This compares to the much wider SORMA span of 0.25 μsec – 4.0 msec, see Sec. 1.7.3 (Tab. reftab:SormaTimes, p. 27).

- Robustness – protected invocation and SORMA self-managed re-connectivity to newly started daemons. Resume capabilities for complex state machines (e.g. robot) is facilitated by coding re-configuration information in flavor names.
- Easy-to-configure (both, on object and SORMA layer) – dynamically at run-time and persistent by editing a dictionary file.
- Easy-to-use: service naming and SCOTT interactive support – immediate test-suite, scripting, and GUI; Requesting service by names hides all configuration details behind semantic wording.
- Easy-to-fix – support extensive and detailed clear-text error and warning messaging, for rapid error spotting (also remote) - also giving non-experts a chance.
- Easy-to-learn – the small primary interface ("*EXEC/CTRL/DBX*") in conjunction with on-line help and readable commented dictionary entries make it easy to explore new service classes and their behavior (plug-and-play)
- Easy-to-share – SORMA services are simple to share via daemon configuration, which is an incentive for better interface designs by early beta-testing.
- Concurrency support – shared hardware service by multiple users (private flavor states) and concurrent application development through robustness.

Chapter 2

Hybrid Integration Examples

The previous chapter explained the SORMA concept, this part reports on two demanding integration examples:

(i) the Bielefeld Checkers player demonstrates how a complex task can be successfully be composed in a distributed, team development, computing and robotics architecture;

(ii) the second application emphasizes real-time capabilities. A rapid learning network (Parameterized Self-Organizing Map, PSOM) is used for 3 D visual target *tracking* in combination with force following and interactive user commands;

2.1 Wiring Service Objects to Play Checkers

The first example is an *autonomous checkers player*, developed together with a group of students. The initial idea was to exercise all major robot-vision lab components in a complex real-time task including human interaction. The decision to play the board game of checkers, less driven by the comparable predecessor from Rochester (see discussion), but rather by the incidental availability of a well working symbolic game algorithm.

Particularly in a research environment, the very useful strategy *"Divide and Conquer"* is prominent. But when complex task should be performed in reality and should be repeatable, a further requirement gains importance: all "conquered" parts *must work harmonically and reliably together* (a challenge, which is more standard in the industrial domain).

Thus, the task of dividing the complex real-time task into manageable, parallel programmable software objects is coupled with the problem of making the phase of joining the parts to a phase of excitement and not of nightmares. The checkers player project was a major impetus for the development of SORMA far beyond the purpose of a communication library which conveniently wraps low-level library calls. After all, SORMA proved its value for developing easy-to-test, easy-to-integrate, network-transparent, sustainable code in a group of programmers with very different programming experience.

2.1.1 Why Checkers?

The task of an autonomous robot-vision system, which is able to play checkers with humans involves quite a number of fundamental issues in the field of sensor based robotics. Among them are:

- A hybrid system concept combining components with neural networks, artificial intelligence, and conventional engineering solutions;
- Perception action loops, closed on several levels.

It comprises:

- an AI task planning level for analyzing the game situation, strategic move planning, task decomposition and control;
- image acquisition, processing, and high-level interpretation;
- semi-autonomous task execution and validation by the robot arm-hand system with sensor integration for guarded motion and visual servoing;
- man - machine interaction in two communication situations: *(i)* the game played by the pair human player $\leftrightarrow$ system and *(ii)* the system $\leftrightarrow$ operator interaction concerning game supervision, safety, and error and exception signaling;
- robust fault-tolerant communication between distributed software and with complex hardware.

2.1.2 The General Plot

Fig. 2.1 shows a picture of the scene: a checkers board with wooden, colored tokens (red and yellow) is placed somewhere on the table, the Puma robot behind. The human player is asked to choose a valid move, while the board is surveyed by the end-effector camera from an upper position. As soon as the human move is considered final (non-changing new board configuration) it is validated and either accepted or rejected with an appropriate spoken voice remark. Once the human

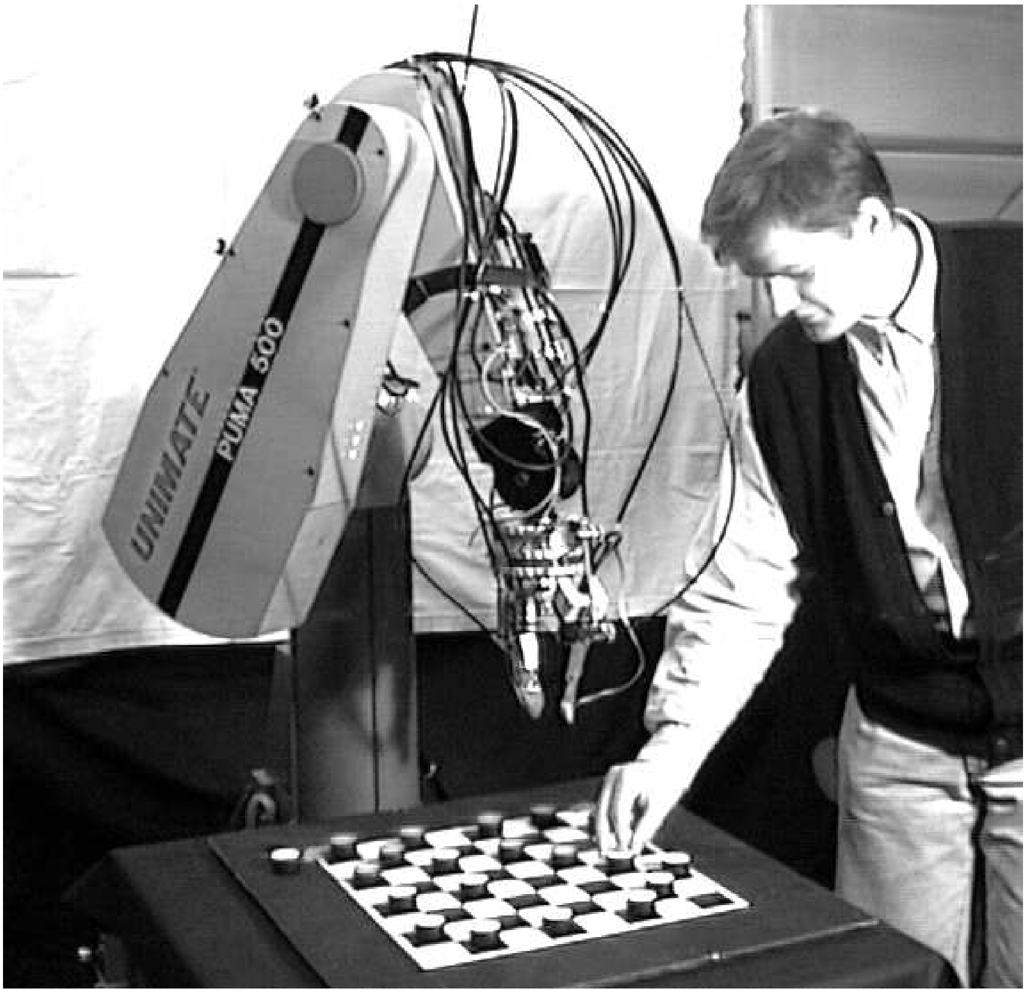


Figure 2.1: The checkers game set up with the board somewhere on the table and the Puma robot playing against the human player.

move is legal, a response is planned by a symbolic game playing algorithm. Since the international rules of checkers are obeyed, the response may be a simple move, a capture, multiple captures, a crowning, or an extended long-move (privilege of the queen). The latter ones must be decomposed in a sequence of sub-tasks (single move primitives) before the robot-vision system is engaged to execute the response. The loop goes on until the game is over, or the human decides to stop or restart a new game.

2.1.3 Divide, Conquer, and Connect

Following the idea of protected communication (see Sec. 1.4), we distributed the task on several daemons processes as sketched in Fig. 2.2. Their names indicate the main purposes, the arrows display the directions of occurring service (SO) request.[1]

The "Player"

The player process is the master of all the other daemons. Therefore it is responsible for following the plot described above. Its particular tasks are the following: *(i)* it comprises the artificial intelligence component able to analyze the symbolic game situation, validate human moves and plans its reaction. This part is programmed in the C language for quickly generating and evaluating trees of legal moves scenarios. The result is a valid move, possibly including an action sequence, e.g., additionally removing captured stones to the heap. *(ii)* The subservice calls are supervised, error messaged are evaluated for autonomous reaction or proper telling the user or operator about the exception (this part is written in the Tcl language). *(iii)* A Tcl/Tk graphical user interface welcomes the user and offers several extra configuration options (mostly for developing purpose, e.g., a symbolic board visualization for comparing internal representation, or dry runs in simulator mode; acoustic verbosity levels, see below.)

Notifying (Non-Localized) Humans: the "Speaker" Daemon

When the human interacts with a machine, important messages should be presented in such a way, that the human's attention is drawn. Usually, the user is sitting in front of some display, where the notification schemes ranges from a beep signal, a textual message in a terminal to a (sticky foreground) pop-up window which must

[1]The checkers application would not exist in the presented form, without the important contributions of Christof Dücker (robot manipulator, vision), Gunther Heidemann (vision 2nd), Hartmut Holzgraefe (robot hand), Michael Krause (player, GUI), Dirk Selle (game), Bernd Sieker (player), and Patrick Ziemeck (vision 1st).

be confirmed before it disappears again. But this is unreliable when the human is far from the relevant screen, for example he is staying at the table expecting something to happen. The natural solution is to communicate it *acoustically*.

The "speaker" daemon offers a central network service to speak out recorded comments, alert messages (e.g., "power-up the robot") and message sequences (e.g., "…at" "D" "4") in a spoken form (in this application a single speaker, placed at the robot "talks", fed by SGI workstation.)

This way any applications can profitably use the named SOR without worrying about the incompatibilities of different computer brands. Writing applications with voice messaging become portable across the computer platforms by a single function call, like `'strDispatch("FORK.sound EXEC alertPowerUp.snd");'` (see also Fig. 1.4). The name indicates the general class of network-transparent, asynchronous, parallel service calls. Additionally, the number of non-blocking execution (fork-calls) of parallel processes can be limited, here in order to avoid scrambled messages.

To complement the spoken *"what"* message by demonstrated *"where"* information in the workspace, the laser pointer device in the hand can be engaged to *point at the spot* and blink. A separate laser service class takes care of these needs (in the checkers example, this method is activated in case a grasped stone got lost, which is so rare that the "laser" daemon is not drawn in Fig. 2.2.)

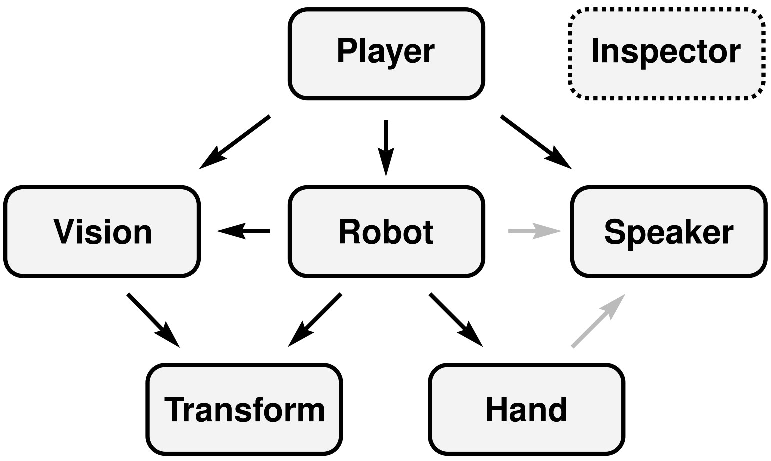


Figure 2.2: **The Checker Players Architecture.** The hierarchical client - server architecture in the checkers player task with usually six Unix processes, distributed on three computers. The arrows indicates the SORMA service call directions. The "Player" process is the master of the daemons below. All relevant messages are back-propagated. Optionally, an standard SCOTT "Inspector" may monitor and call any of the daemon services – during a running applications.

The Central "Transform" Daemon

The Checkers scenario needs three different board representations: *(i)* abstract board coordinates [*A3*=(1,3)] *(ii)* camera image coordinates [pixels], and *(iii)* robot world coordinates [mm]. The transformation between these three coordinate systems depends on the actual board location on the table. This problem is analogous to a auto-associative image completion task, which is described in Walter and Ritter (1996a). Here we apply a neural network approach called *Parameterized Self-Organizing Map* (PSOM), see Walter (1996) Based on the identification of a pair of violet board markers, the correct transformation can be adapted. The associative completion feature of the PSOM is utilized to perform the required coordinate transformations. The PSOM algorithm is implemented a NST unit type, wrapped as SORMA service class, and configured here as a separate "transform" daemon (see Sec. 1.1 and Sec. 1.9.1). This allows to send transformation requests to a central resource, here a single neural network module.

The second PSOM task of the "transform" daemon is to perform visual Jacobian matrix operations in order to check and correct the robot position before grasping objects (pre-share postion, see Sec. 2.1.4).

High-level Perception: The "Vision" Daemon

The "vision" daemon serves as a specialized "sensor agent" for high-level image interpretations. Based on the input of the end-effector color camera, three services are used: *(i)* findBrdMarker find the board marker pair (called once at start-up and when the board seems relocated). *(ii)* getBrdSit get the current symbolic board situation in abstract board coordinates together with a detailed estimation of a confidence level for each field. *(iii)* Before grasping, the relative position between the object and the end-effector is determined by the getDisk service object, in order to correct the pre-share postion, see Sec. 2.1.4.

The first version of the vision daemon ran on the Androx system and has recently been replaced by a (one order of magnitude) faster version working on the Datacube (see Walter and Ritter 1996b). In both cases a look-up-table based color segmentation, followed by a connectivity analysis is performed. A competitive winner-takes-all RBF (radial basis function) approach assigns and certifies the color specific blobs (center and size) to the image field locations previously obtained from the "transform" server. All other required parameters can be tuned interactively and stored in the SORMA dictionary (the larger color look-up tables are stored as name references to extra files).

The Datacube system allows *extremely rapid color segmentation* by coding the color information in a representation of the HSV space (hue, saturation, and value; 6+5+5=16 bit) and allowing direct identification in a 64 k look-up table. It captures the training result of a RBF network supplied with hand-labeled images of the checkers board. A competitive clustering algorithm allocates the radial basis functions in the HSV space. For this application a single Heavyside function turned out sufficient. This reveals a great potential for more refined learning networks to enlarge and adapt the tolerance range in ambient light conditions. This implementation is a good example for *compiling the essence of a learning method* into an extremely rapid, specialized hardware.

The current SO interface levels consist of highly-specialized checkers calls, the lower level building blocks are not accessible yet. This is quite perfect in the checkers player application but reveals that the level of interfaces should offer various entry levels in order to be better re-usable in other contexts. An example of a more versatile multi-level access is shown in the functionalities of the robot server.

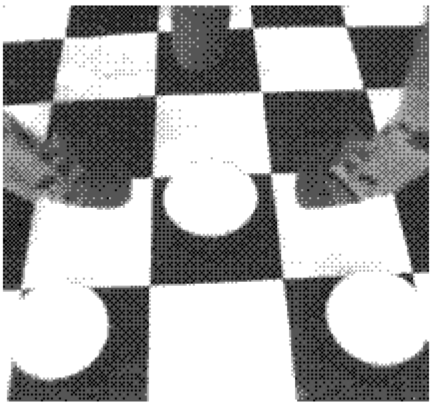


Figure 2.3: Images obtained from the end-effector camera are the basis for correction movements to the desired preshape grasp position. The PSOM network is applied to implements a rapid learning procedure for the non-linear, visual Jacobian transformation.

Action: The "Robot" Daemon

The "robot" daemon offers a variety of parameterized services, including the following: *(i)* The basic service pino (derived from the robot's nickname "pinocchio") offers an interface to fundamental and structural parameters (e.g., control task interval, enabling force torque sensor reading and gravity compensation, enable writing state records to shared memory buffers, and activating other service control tasks; for a fuller account see Dücker 1995.)

(ii) The mvpos service class offers position controlled motion commands. Those can be absolute positions, named or explicit, or relative to a reference pose, given in various representations (joint angles, generalized position, relative transformation matrices etc.) and with configurable speed and interpolation modes (joint or Cartesian). The service can listen to relative displacement commands passed by a specific shared memory segment (see Sec. 2.2). Furthermore, robot status information can be accessed.

(iii) The wspc class defines services to *guard* current motion, see (Walter and Ritter 1996b) These watchdog functions define sensory pattern matchers. E.g., currently, geometry (workspace="wspc") and force patterns are configurable. The configuration "recipes" can be collected and stored in the dictionary with conveniently retrievable names (e.g., "wspc.fragile", "wspc.noTable"). Note, that any particular guard service after dynamical instantiation can be interactively re-configured. Activation is done by the higher level motion commands like the following:

(iv) push and carry are complex movement primitives, which integrate several action and perception components as sub-services. push includes guarded approach to, contact with, and compliant pushing of an object with one straightened finger. All control law parameters, sub-service names etc. are hidden in the dictionary, only initial and final transfer positions are arguments to the EXEC call. For the checkers application the carry is central for moving the checkers tokens (single or queen stacks), facilitating capture moves, see Sec. 2.1.4

The "Hand" Daemon

The manus service, here wrapped as a stand-alone daemon on the host *"druide"*("manus" daemon), provides interface to the TUM-hand. Absolute and relative motions become interactively available as *(i)* per joint, *(ii)* per finger or as *(iii)* coordinated fingertip motions. Hand postures can be called by names and *(iv)* trajectory sequences can be directly executed. Three-finger (tripod) grasp and release operations include automatic sensor verification and additional (configurable) signaling. E.g. if the hand is requested to grasp the object and the fingers cannot find the expected counter forces (in place), the manus service may directly invoke the voice service to notify about this problem. It might be interesting to note that this way the desired degree of user-friendliness and autonomy can be dynamically configured. All (user-unfriendly) low-level *"manus"* internal control law parameters etc., located on the subordinated VME controller (see Walter and Ritter 1996b), are interactively accessible (read+write), but normally hidden.

2.1.4 High-level Task-Directed Action Services

As pointed out by Brunner et al. (1995)a (1995)b, the integration of sensor-based task primitives (*"elemental moves"*,"EM") into a hierarchical framework is an important step to reach a move generalized planning and assembly plan generation. The transfer of a stone seems to be a good example to explain how task-directed programming can be approached within the SORMA concept.

The carry service performs a multi-sensor based pick-and-place operation, provided by the "robot" daemon. As Fig. 2.4 illustrates, the composed carry service splits into action sub-tasks involving the “hand” , the “transform” , the “vision” , and in exception cases, the “speaker” and “laser” daemons. The carry service is called with the initial and final target location argument in abstract board coordinates (3 D, including the height, for manipulating space saving stone heaps). The “transform” daemon (+T) converts them into the required robot world coordinates. The robot arm moves into an approach position above the stone. At the same time, the fingers of the hand are reshaped in a pose suitable to the inter-stone distance, which is concurrently handled by the manus service.

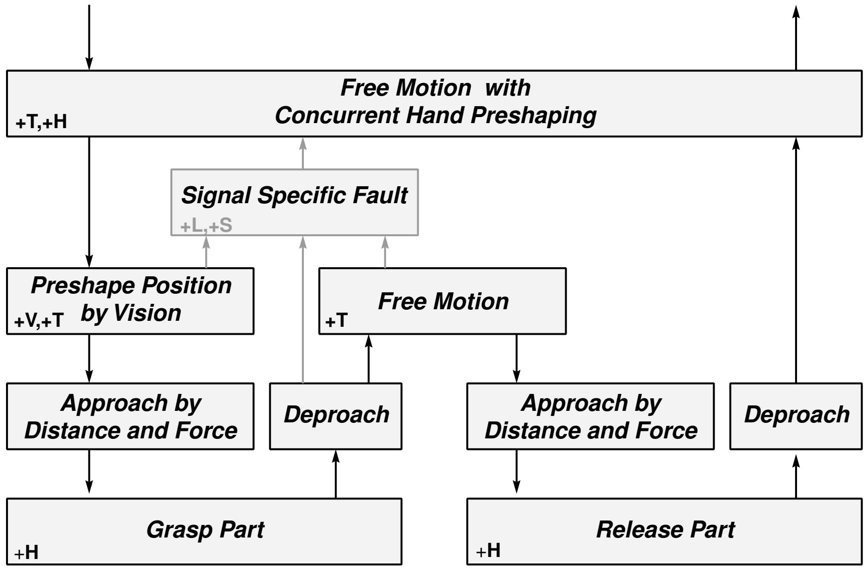


Figure 2.4: The robot “carry” task for relocating an object, commanded in symbolic coordinates. The “+”-marked initials indicate the sub-services requested for performing the sub-goals (**T**ransform, **H**and, **V**ision, **L**aser, and **S**peaker daemon). The arrows illustrate the state transitions controlled by sub-task specific conditions and sensory feedback.

In the next step (Preshape-Position-by-Vision), the position above the stone is verified and corrected using one image from the end-effector (see Fig. 2.1.3). The image acquisition and interpretation is encapsulated in the “vision” (+V) service getDisk and its stone center information converted to a correction of the horizontal robot position, again by a suitable PSOM call to the “transform” (+T) daemon.

The approach speed in the vicinity of the grasped object is configured slower than the free space motion and is additionally guarded by continuous force checking. Now, the object gets embraced by a symmetric tripod grasp of the hand, lifted (deproach), and transferred (arch shaped path) to the next approach position. The sub-task for approaching and deproaching are re-used to carefully place and release the stone before moving back to the next (park or task) position. The visual and the force sensor are used to verify the expected task execution. If inconsistencies are detected, they will be signaled back to the caller (the player) and additionally (if desired) directly to the human via the "speaker" and "laser" service (+S,+L) as described above.

Figure 2.5: Free Motion transfer phase of the object.

2.1.5 Discussion

One of the main advantage of SORMA is the *interactive access to* all *relevant configuration parameters*, but at the same time hiding them behind a single compact call. For example the complete carry service call is definition such, that only the execution relevant arguments, initial and final position, are given. Modified instances are useful to gain special non-default configuration. The following example demonstrates how such a second instance is dynamically defined, created, and used with one of the required task specification parameters changed.

```
dict CTRL -add carry.slow '-SUBST carry -transferSpeed 0.2'
carry.slow EXEC FMSG 6   3 4 0   4 5 0
carry EXEC FMSG 6   4 5 0   3 4 0
```

(i) The first line instructs the dictionary to add a new entry with the name `carry.slow`. *(ii)* With the second line, a new service object, named `carry.slow`, is dynamically instantiated with the default parameterization list of `carry`, plus one extra overwriting speed value, as stored now in the dictionary. The carry operation is then executed (`EXEC`) *slowly* from field *C4* to *D5* and *(iii)* back in nominal transfer speed. This shows that alternatively to appending parameters to arguments lists, the modified services can reflect their execution "behavior" in an intuitive, user-friendly name. Both service flavors are then readily available, details are present but hidden. Note that the first line can be omitted after inserting one permanent line in the primary dictionary file.

This code can be directly typed in the on-line interpreter mode of the "robot" daemon or in any on-line inspector, calling the serving "robot" daemon remotely. In the latter case, the service names must get the appropriate remote address extension (e.g., `carry@druide:0x14`, or dictionary abbreviated, `carry@pinoServ`, or dictionary default address `CARRY`, see Sec. 1.4.3). The same lines can be called one-to-one as text oriented `strDispatch()` calls in any client program (C and Tcl/Tk script), or identically by using the vectorial interface directly (here 6 floats "`FMSG 6...`", see Fig. 1.4).

Note, that the point is not that every parameter should find external access, but that SORMA parsing structure and support makes is particularly *easy* to *create* and *maintain* interactive access to them. This way, all parameters found in the initial test phase sustain this access, even when used as sub-sub-service, throughout the service's lifetime.

Sub-service calls are configurable by the sub-services level or a complete swap by exchange of the name (e.g., `wspc` versus `wspc.noTable`). This is an efficient environment to tune and program controllers, sensory configurations, and conditional parameter sets in an object-oriented manner.

We have developed a *robust system* to play the checkers game in a distributed and hierarchical client - server architecture. SORMA supports the system robustness in a number of ways:

- Sensor based task qualification after execution and
- several layers of closed perception–action loops add to a system, able to cope with a number of exceptions in an intelligent task-oriented (situated) manner, with different levels of autonomy (non, semi, full autonomous reaction).
- The system supports systematic error and warning message propagation (code and text) back to the original caller (network transparent). A qualified message is the best a service can offer, in case the service object cannot react autonomously itself upon the exception. Clear text messages facilitate the robust usage of services by non-expert users.
- The system services communication structure offers rapid local and (identical) protected remote service calls.
- The system is able to cope with failed and restarted daemons. This feature is of greatest value for concurrent application building in a team of programmers. It is much less important as soon as the entire complex product is mature and performs stably.

Furthermore, rapid learning by demonstration finds a good example in the checkers scenario. In particular the rapid construction of the required mapping from misaligned objects to correcting robot movement for reaching the correct hand preshape position. Based on a set of only 4 training examples the sensor-action mapping is learned. During the training phase the robot is located on a horizontal 2×2 grid, centered at the correct nominal grasping pose. The complete non-linear mapping is learned by a PSOM network. This implies the knowledge of the Jacobian matrix and its inverse over the learning domain and facilitates a precise one-step correcting move, which is an advantage to approaches which construct a local linear mapping at *one single point*. E.g. (Castano and Hutchinson 1994) and (Brunner et al. 1994) determine the inverse sensor Jacobian matrix also by a number of test-moves of the robot arm. By a overall comparable training procedure the PSOM approach offers to learn the non-linear approximation task accurately over an extended domain, in contrast to linearized approximation in a single point (Jacobian matrix).

How does the system compares to the Rochester Checkers Player? Marsh et al. (1992) aimed at demonstrating the advantage of the animate vision paradigm by decomposing the problem in function modules. These are programmed in different programming models and run on a 32-node BBN Butterfly Plus parallel processor. The developed operating system *Psyche* allows various interprocess communication and synchronization schemes between the Motorola 68020 processor nodes. Psyche and SORMA share the idea of software modules with a set of interface functions to grant procedural access to the code and data encapsulation by the module. Furthermore, they share the idea of offering time optimized and safety optimized protected function calls to other modules. They mostly differ in the level of communication, while SORMA can cope with various general purpose Unix operating systems, Psyche is itself an multi-processor operating system for a VME-based shared memory machine. The outside world communication is channeled via a Sun workstation and a serial port on each node. The VME-bus hosts in parallel a Datacube MaxVideo 20 system, see also (Walter and Ritter 1996b).
The Rochester checkers player engages a VAL controlled Puma 761 robot, carrying a camera platform and a passively compliant checkers-pushing tool.

In spite of the parallel implementation of the board image interpretation, the checkers player, and the move planner, the Rochester system did also follow the sequential nature of the game. Due to the interference of the image analysis and robot action , caused by the camera-in-hand configuration, the high-level procedural sequence *perception-cognition-action-perception* was not left. Therefore, the hierarchical call structure in Fig. 2.2 seems an equivalently suitable solution. Here, concurrent action executions are reserved for particularly suitable cases (see Fig. 2.4).

The advantage is the gain in a clear error propagation structure, which enormously simplifies error spotting in a complex, distributed application. Note, this is a development *tool* advantage, which naturally becomes invisible as soon as the application matures to playback performance.

In the next section, we will present an application aiming at the contrary, a fast concurrent action – perception coupling.

2.2 Visual and Force Tracker

2.2.1 Motivation

The purpose of the tracker application is twofold: *(i)* to test the usefulness of the PSOM approach and the SORMA concept in a demanding *real-time application*; *(ii)* The checkers application brought up the question of *increasing the human - machine interaction dynamics and autonomy*. What will happen, if the human fools the robot system and takes away the stone? (The checkers system will expose the unexpected fault loudly.) Can we make the robot following the human hand teasing the robot with the desired object? Can we make the robot pick a part from a non-resting, moving human hand? This is a basis requirement for, and therefore a step towards a helping "third (robot) hand", which is able to dynamically take-over object or tools presented by a human, able to cope with unknown, drifting handshake positions.

The standard *passive tracking* problem is concerned about locating the image of a moving object, seen by a stationary camera system, e.g., (Allen et al. 1993). *Active tracking* (animate vision paradigm, see Ballard 1991) aims at keeping the target centered in the image of an active camera. It is more demanding, since, additional to the target, the ego-motion of the system effects the seen camera image. Delays in sensory processing time become relevant, and can be compensated by *predictive control* (see below).

Active vision approaches divide in two main categories: *(i)* gaze direction control was studied for one (Murray and Basu 1994) (2 D, azimuth and elevation angle) or two stationary cameras (Coombs and Brown 1993; Daniilidis et al. 1995).Those active camera tracking systems are interesting, e.g, as automatic cameraman in surveillance, security, and video-telephone systems.

(ii) Here, we are interested in moving the camera attached at the robot hand in full 3 D space. The idea of an *active vision-in-hand* configuration was studied before, mostly for static target objects, e.g., by Gengenbach (1994), van der Smagt (1995), and Brunner, Arbter, Hirzinger, and Koeppe (1995) and in case of 2 D moving targets by Papanikolopoulos, Khosla, and Kanade (1993). A zero-gravity 3 D ballistic tracker was successfully demonstrated by the ROTEX experiment (Hirzinger et al. 1994).

Additionally, we want to combine *(i)* the *visual guiding* by an object seen by

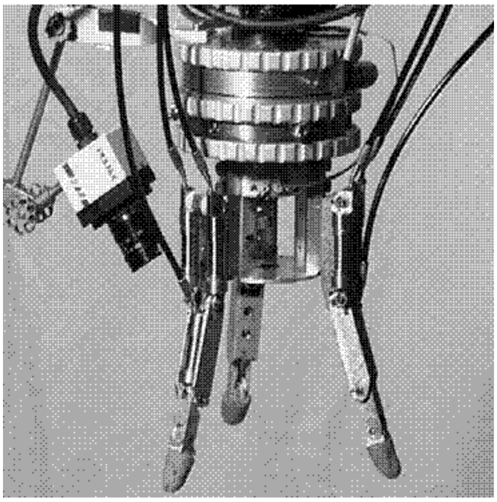


Figure 2.6: The camera-in-hand end-effector configuration. Between the arm and the hydraulic hand, the cylinder shaped device can measure the currently exerted (6 D) force and torque values. The three finger modules are mounted here symmetrically at the 12 sided regular prism base. On the left side, the color video camera looks at the tracked object from an end-effector fixed position.

the end-effector camera with the idea of *force guiding the robot hand manually* in two ways: *(ii)* directly at the robot hand and *(iii)* remotely, by *teleoperating* the robot with the 3 D SpaceMouse. In both cases, the actual forces, exerted by the human are measured and converted to modify its current position.

This form of integrated force control solves at the same time the problem of possible collisions in a very graceful and practical way. In those cases, the visually commanded input are counterbalanced by the reactive forces of the environment. The result is a form of "virtual spring" located at the visually determined target position, which we find an attractive compromise.

In spite of this graceful reaction, care must be taken to discriminate the desired target from undesired artifacts, in order to avoid unintentional robot behavior. As redundant and independent safety layers, we implemented a *(i)* force overload measuring in the wrist (separate software layer, stop motion and exit process) and *(ii)* we armed the experimentation table with force sensors at the legs (FSR sensors, independent hardware circuitry for powering off the Puma).

2.2.2 The Tracking Task

To simplify the vision pre-processing stage in the first experiments, we choose a monocular, model-based approach to gain the 3 D target position information. In particular, we mounted a ping-pong ball at a stick and equipped it with an central illumination. The task is to follow the object in a nominal relative position, which is suitable for a later step, the interception in an object-depend grasp pose.

The general scheme of the tracking application in shown in Fig. 2.7. It comprises several main components:

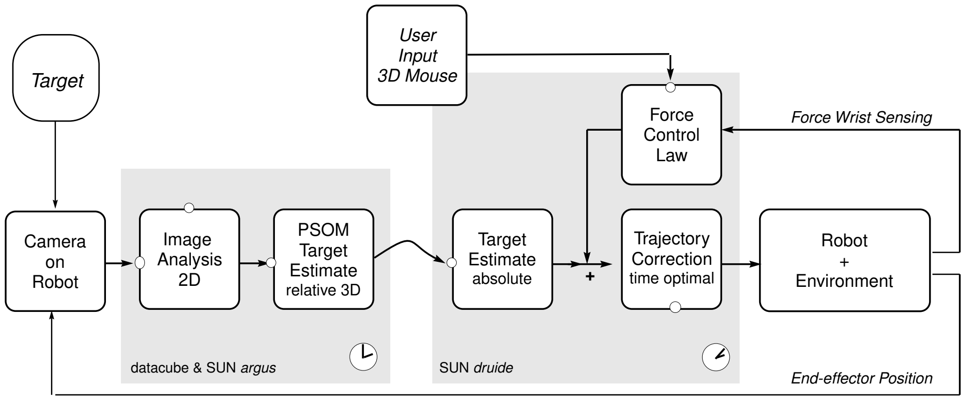


Figure 2.7: Real-time tracking with three inputs: *(i)* predictive visual servoing with asynchronous image preprocessing and reconstruction of the relative nominal target position with a PSOM; *(ii)* manually guiding the robot using the wrist force-torque sensor; *(iii)* teleoperation by the 3 D SpaceMouse. The task is distributed between the vision host *"argus"* and the robot host *"druide"*(see Walter and Ritter 1996b). The interprocess communication uses several shared memory SORMA service objects (Sec. 1.7.2; indicated by little circular ports to the block icons).

2.2.3 2 D Image Analysis

The wide-angle camera is fixated at the robot end-effector (see Fig. 2.6) and detects the ball, when visible in the view field. While emitting light, the object is easy to detect by its relative brightness to the background. The Datacube system, hosted by the *"argus"* workstation, grabs a monochrome image of size 500×500 pixels and performs a binary threshold operation, followed by a connectivity analysis.

The connectivity analysis is a very convenient library call, which generates a tree of connected pixel areas (blobs) and enclosed holes. The advantage is a high-level *object verification* for the price of longer processing time, dependent on the found blobs structures. With the given set-up, finding the largest blob is sufficient to determine the centroid of the binarized ball in the image, as well as the pixel area, the radius and eccentricity of the best fitting ellipsis. Later on the blob size parameters give input to the relative depth estimation (by PSOM), which makes them a safety relevant information. Therefore, the values are matched with expectation patterns, in order to avoid misinterpretation of artifacts. We verify, e.g., that the object is round (eccentricity), its size is within a suitable range, and last not least, the size estimation is not fouled by a blob clipped at the image margins.

This image analysis process is running continuously on the host *"argus"* and communicates its (raw) results via a SORMA IPC snapshot buffer (cyclic shared memory, see Sec. 1.7.2). A second service object instance facilitates asynchronous reception of preprocessing parameters, which can be interactively tuned by a convenient GUI interface.

2.2.4 Learning 3 D relative Target Estimation

After validating the 2 D image parameters, the learned 3 D model of the target object is used to gain the Cartesian distance to the desired object pose in robot tool coordinates. Here, the binarized image centroid, area and mean radius are employed for this reconstruction, which is draws on the perspective distortion of the camera.

As shown in (Walter 1996), a PSOM can favorably learn this transformation by a small set of training examples. The target is fixated, and watched by the robot-vision system from a $3{\times}3{\times}3$ grid of position around the desired nominal pose. By recording the relative robot commands together with the image preprocessing results, the camera mapping is learned. This implies learning of the entire model influenced by the target shape and nominal pose parameters, as well as the camera parameters, thus as focal length, relative position and orientation to the robot coordinates.

The PSOM network is used as SORMA service object module within a pure client process. This facilitates the access of shared memory IPC communication modules even across machine boundaries. Its input is read from a segment shared with the image processing task and the output, the relative target estimation in world coordinate, is transfered to a standard service on the other host *"druide"*. Time lag measurement are discussed below.

2.2.5 Sensor processing times

Since both, the target and the robot are moving in space, the relative target information must be related to the time when sensory information was gained. As soon as a new relative target estimation is accessible at the robot control task, the visual sensor information has a substantial and varying age τ, as illustrated in Fig. 2.8. The processing times depends primarily on the number and sizes of found image blobs (110 ± 20 ms) meanwhile several robot control cycles are producing robot trajectory set-points and may move significantly.

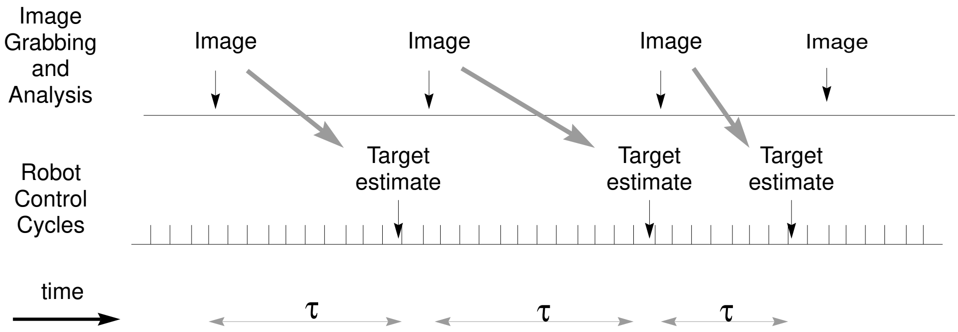


Figure 2.8: Asynchronous camera image grabbing and processing with varying duration require a target state prediction, taking the sensor information age into account. Clock differences on the time-stamping host computers are accounted for.

Therefore sensory information receives a *time-stamp*, when the image grabbing is completed. When evaluating the time-stamp differences, here, the additional problem of different wall clocks needs consideration (clock icons in Fig. 2.8). We solve this problem by repeatedly estimating the clock difference during the data transport from *"argus"* to *"druide"* and producing such a suitable time base correction.

2.2.6 Asynchronous Absolute Target Prediction

The asynchronous nature of the visual sensory input recommends a predictive target state model, because real robot arm control requires a method to operate at fast servo rates (15 ms) - regardless of whether new information of object positions are available. Furthermore, we have to take sensor noise and the pre-processing delays into account. In a similar way as in (Allen et al. 1993) and (Daniilidis et al. 1995) we employ a α-β-γ stationary linear Kalman filter (Bar-Shalom and Fortmann 1988; Grewal and Andrews 1993), in order to estimate the state s of the target. In world coordinates we denote its position $\mathbf{o}$, velocity $\mathbf{v}$, and model the target motion with constant acceleration $\mathbf{a}$ in time t, equal to step k with duration Δt:

$$\mathbf{s} = (\mathbf{o}^T, \mathbf{v}^T, \mathbf{a}^T)^T \tag{2.1}$$

$$\mathbf{s}(k+1) = \Phi\, \mathbf{s}(k) \qquad \text{with} \qquad \Phi = \begin{pmatrix} \mathbf{1}_3 & \Delta t \mathbf{1}_3 & (\Delta t^2/2)\mathbf{1}_3 \\ 0_3 & \mathbf{1}_3 & \Delta t \mathbf{1}_3 \\ 0_3 & 0_3 & \mathbf{1}_3 \end{pmatrix} \tag{2.2}$$

Each time t ($\hat{=}$ step k) a new visual sensor input is available, the state s at the time $t - \tau$ ($\hat{=}$ step $k'_i = k - \tau/\Delta t$) the sensory input originates, is recursively estimated from the previous target state $\mathrm{s}(k'_{i-1})$ and the absolute target position $\mathbf{o}(k'_i)$ which is gained by using history records of previous robot positions feedbacks. Since these are periodically written in a cyclic shared memory buffer (SORMA service, see Sec. 1.7.2) the appropriate record can be determined from the measured sensor time delay $k - \tau/\Delta t$. The new estimation $\tilde{\mathbf{s}}(k_i')$ influences the target model according to the Kalman gains.

$$\mathbf{s}^{new}(k_i') = \mathbf{s}^{old}(k_i') + (\mathbf{1}_9 - \mathrm{diag}(\alpha\mathbf{1}_3, \beta\mathbf{1}_3, \gamma\mathbf{1}_3))(\tilde{\mathbf{s}}(k_i') - \mathbf{s}^{old}(k_i')) \qquad (2.3)$$

The gain coefficient α, β, and γ can be derived as a function of the target maneuvering index. The maneuvering index λ describes the ratio of plant noise covariance and measurement noise covariance. The higher the maneuvering index, the more reactive the system response to visual input changes. Vice versa, a lower maneuvering index reflects a high confidence, that the target model describes a smooth motion.

Additional to the Kalman filtering, the motion estimation $\mathbf{v}$ and $\mathbf{a}$ is damped for low amplitude motions as an additional stabilization for a quasi-stationary hand-held object.

2.2.7 Active and Reactive Force Input

The other two force input channels are pre-processed in a modular, isotropic way. The 3 D vectorial inputs are low-pass filtered and radial squashed by a piecewise linear function, comprising a null-radius filter, a scaling, and a clipping filter. The parameters can be configured individually for the wrist force, and the 3 D mouse input (SORMA dictionary). The latter input is fed with the help a separate Space-Mouse daemon, which handles the serial line communication and writes the result to a SORMA shared memory segment (max 60 Hz).

2.2.8 Trajectory Correction

As indicated in Fig. 2.7, the three commands are combined to modify the robots current trajectory. To allow fast robot tracking motions, the remaining distance to target interception must be divided into differential path segments per control cycle such, that the robot reaches the target (stopping criteria) *(i)* with essentially zero velocity (locking-on), and *(ii)* within minimal time. This goal conflict is regulated by specifying speed and acceleration limits. We implemented a trajectory generator with trapezoidal shaped velocity profiles, taking the maximum allowed accelerations and de-accelerations into account. A similar approach was taken for 2 D simulations and described in detail in (van der Smagt 1995).

For the safe operation of the robot, care is taken to check sensory data integrity (checksumming) and actuality. *(i)* The FTS input is synchronous, and *(ii)* the

SpaceMouse will be called back by the daemon, if a non-zero input is not updated within a certain time span.

(iii) The predictive nature of the visual motion commands requires a method to deal with "target-disappeared" events. (These can occur when the robot gets teased too wildly, and the configured accelerations do not allow the robot to keep the object in full sight, or, the image analysis can not identify a unique and expected object structure, e.g. due to occlusion). Here, we propose an *expiring mechanism*, to perform a smooth stopping behavior of the current motion. After a configurable expiration time without information update (typ. 0.5 s), the target state model is adapted such, that the current motion decays exponentially.

2.2.9 Discussion

We have developed a system for tracking moving objects, relying on real-time estimation of 3 D relative pose parameters from monocular images. The underlying model including target object structure and the particular perspective geometry are rapidly learned by a PSOM network. The system is able to cope with inherent noise and inaccuracy of the visual sensor by applying parameterized filters that smooth and predict the objects position seen by the camera mounted at the robot hand. Additionally force compliant motions and remote control with a 3 D mouse are integrated.

The robot control system is able to cope with the inherent mismatch between the vision sampling rate and the servo update rate. However, the bandwidth of primarily the visual input is limiting. It allows to specialize the filter parameterization for *rapid, saccade like movements*, and alternatively, for slow and *smooth pursuit* motions. This opens the interesting question, how to find suitable characteristics to determine the humans intentional context. Then, we can employ the context-oriented learning scheme (see Walter 1996) together with context optimized parameter sets.

The combination of direct force, vision and remote control shows a variety of possible rapid human - robot system interactions. Further research should be carried out to explore the feasibility of a more general semantics of man - machine interfaces, integrating vision, touch (force) mediated gestures. Those "hands-on" gestures could encode interaction-related inputs, e.g. for commanding task goals and controlling modalities like machine (and service) behaviors.

Chapter 3

Summary and Discussion

In the following we want to summarize and discuss the SORMA architecture. The following major issues were particularly important for the design of SORMA:

3.1 Design Issues

- support of *team development* by *protected invocation* of services;
- *shared network access* to resources;
- remote service access (protected invocation): full network transparency by a *proxy service objects* mechanism;
- time-optimal invocation for *real-time* tasks. Alternatively to the inter-process, protected invocation, the very same compiled code can be used by fast intra-process communication mechanism – with identical interfaces, which gives
- support for *service migration* (symmetric intra-process & inter-process communication, on local and remote network host)
- improving *robustness*
  - by strong support on exception message transport and handling (important for network transparency)
  - resume capabilities for complex state machines (e.g. robot) are facilitated by encoding detailed re-configuration information in service "flavor" names;
  - self-managed re-connection capabilities to newly started partner processes (a particularly helpful feature during concurrent application development in a team);

- support for the economic *re-use* of software (service modules) by:
  - strong dynamic configuration and (run-time) re-configuration support;
  - interactive testing, exploration (learning), and usage;
  - built-in and on-line help (life-long);
  - standard command-line inspector (SCOTT) with clear text interface to all options;
  - simple way to create customized GUIs (graphic user interfaces, with Tcl/Tk *cottwish*);
  - scripting capability (interpreted);
- our heterogeneous Unix workstation environment requires 100% portability and compatibility between various operating systems AIX, Alpha, Iris4d, Irix5, Linux, NeXTstep, Solaris, SunSO (now – not in the future);
- all robotics hardware devices are interfaced by specialized service object instances, serving as special "device driver". By configuring these as network accessible "daemons", one can easily constitute a multi-master *field-bus* based on standard Unix workstations;

3.2 Language Mapping

As already mentioned before, the SORMA interface structure has many roots in NST and is also programmed in the language C. This has is pros and cons: The C compiler allows as much object orientation we want, but we loose automatic deep-leveled object pointer handling and automatic multi-generation class inheritance. This encourages to design more flat structures and save unnecessary operations.

From the object oriented programming (OOP) point of view, this is only the half way to the tempting world of arbitrary complex, aggregated object designs. This is somehow true, but doing this exercise without a C++ compiler, gives us a better control on the time performance: *(i)* on a short time scale for time critical tasks SORMA encourages to use "run-time binding" upon *CTRL* commands instead of the classical ways of "early binding" (fast, but static) or "late binding" (flexible, but slower); *(ii)* on a longer time scale C++ repeatedly exhibited floating language definitions, which results in unsatisfying design instabilities. Clearly, in the long run it would be advantageous, to have a standard C++ (o.s.) language implementation with enhanced compiler support for the OOP aspects of the SORMA concept.

3.3 Interface

A key idea of SORMA is its compact SO interface design. It is the basis of an extended and easy-to-use interactive support – conceptually as well as by tools. At the same time it accomplishes this without impairing the real-time capabilities of the service.

The IPC-sub service class is an excellent example on the advantages of splitting the method dispatching in two levels (*EXEC* and *CTRL* method). It gives space to an intelligent (self-checking) and convenient (clear text) object interface and a second execution level parser, which can be designed as fast as desired.

Which types of parser mechanisms are used (for the *EXEC* and *CTRL* methods) depends on the design goals and the implementers preference. The goals are usually mixtures involving ease-of-use, comfort, information hiding, fault-tolerance, parser effort, execution time, etc. The implementation palette reaches from: *(i)* no-parsing (fastest); *(ii)* case switch; *(iii)* loops with token matching (template standard), and *(iv)* language parser (context-free grammar with bison).

Text Conversion turned out to be a very important ingredient for reducing communication costs with SORMA (Walter and Ritter 1996c). It provides easy-to-use, unrestricted interactive access to each SO interface. Separated by reserved word tokens, the transport structure can be string converted in a bi-directional way (excluding the unspecified “any” tuple). By this verbatim translation, interactive exploration is quite more authentic as usual, ready-made test or demonstration programs. Applicability and limits of a component can be investigated in a very quick and efficient manner. The results can be 1:1 transfered to program code – always with the option to chose between protected and time-optimal invocation. Here, SORMA combines the advantages of interpreted interfaces (like Mathematica, Maple) and the benefit of real-time efficiency achieved by its SO-interface.

3.4 CORBA

When comparing SORMA with industrial distributed OOP approaches it appears that the SO concept matches most goals and attributes characterizing a state-of-the-art OOP infrastructure. In particular, CORBA is seen as todays most advanced industrial standard of an infrastructure to distribute and connect objects across networks, applications, languages, tools and operating systems. CORBA stands for *“Common Object Request Broker Architecture”* and is an ongoing development process of the Object Management Group (OGM), comprising about 400 companies, including Sun, IBM, HP (OMG 1995; Stal 1995).

Actually, CORBA and SORMA do share many concepts (which indeed gave recently the inspiration to the terminology OR within S**OR**MA). The *Object Re-*

quest Broker (ORB) is responsible for the transmission of object requests and exchange of data. Supported communication paradigms are synchronous rpc, non-returning (one-way) rpc, as well as deferred synchronous requests. The remotely invoked objects are called *Universal Networked Objects* (UNO). A *Dynamical Skeleton Interface* (DSI) denotes, what we would call the "proxy objects" on the caller side. The transported data types are defined in an xdr-like language IDL (Interface Definition Language). Objects are initialized static or dynamic (Dynamic Invocation Interface DII). Using two *half-bridges* on either side ($\sim$ stubs in Fig. 1.6), two ORBs communicate by an *Inter-ORB* protocol, which is a tree of possible and mandatory mechanisms to ship data between software products of different vendors (IIO, GIOP, ESIOP, etc.)

Criteria for a Supercomponent

Orfali et al. (1994) give a interesting concept list of "*added smarts*", which a component should provide, in order to deserve the term "*supercomponent*". Many of the 16 items find already implementation and support in the SORMA framework:

- Security – access control is not implemented yet; (non-secured) client identification and audit trails to CTRL call are provided;
- Licensing and metering – not considered yet;
- Versioning – CVS (Concurrent Version System) revision and compile information is run-time accessible via the DBX method for each service class (inherited by template);
- Life cycle management – creation, aliasing, and destruction of a service object is responsibility of the SOR-Manager. The SORM is on-line accessible by its SO-interface (via himself);
- Support for open tool palette – visual drag-and-drop assembly techniques are available with NST/NEO (see Sec. 1.9.1);
- Event notification – in a synchronous SOR scheme returning exception messages is well supported; Semi-autonomous processes may employ common event signaling services (see Sec. 2.1.3)
- Configuration and property management, scripting, metadata and introspection, ease-of-use, semantic messaging – well supported as discussed before;
- Persistence – the state of a service object is usually loaded as defined by dictionary entries. Saving of interactively modified state configuration normally

involves user text editing (help by clear-text dumping of internal states). The class may implement direct saving.

- Transaction control and locking – means serialized shared access, transaction with two step commitments. no direct support yet (indirect by IPC stubs);
- Relationship – SORMA supports the usual flat organization of SO by dynamic creation and association with other components. Additionally, local objects can be owned by others (bypassing the repository and name binding), see also Fig. 1.5
- Self-testing – should be done at service object creation time;
- Self-installing – *(i)* a service class must be registered at boot time to inform the SO-plant about it. *(ii)* In a distributed network a service gets available when the daemon is fired-up. This can be delegated to the standard Unix "inetd" network daemon, triggered by a request to the port-mapper. The daemon can be queried about its capabilities and state, currently no central registry is defined. The "object bus" gets configured via file (see Sec. 1.4.3)

SORMA extends this concept list by

- real-time efficiency (time-optimal invocation, symmetric local calls and pre-specialization)
- robustness
- interactivity

in order to serve our specific demand in the domain of robotics.

In contrast to other robot control software frameworks like *Chimera* (Stewart et al. 1992) or *Psyche* Marsh et al. (1992), SORMA does not attempt to serve as a real-time operating system (OS) itself (e.g. it has no scheduler) Since we want to have full portability also to high performance graphic workstations, which do have their own OS, this attempt would not carry far.

Instead, the SORMA communication infrastructure facilitates to interoperate hardware via the standard network, just like a *decentralized multi-master Unix field-bus system* with a high level of portability, flexibility and re-usability.

3.5 Future Work

An interesting line of future work arises from the idea of trading service object request. At a central or decentralized market place, tasks are negotiated and traded between multiple agents. This ansatz builds on the ability of matching and ranking requests and resources on the base of e.g. attributes, prerequisites, availability, latency times, and prices. For example, a very simple application is trading event signal, i.e. specific messages and / or non-specific signals, like e.g. flash-light, beep, text, speech production. More complex resource and task bidding and trading concerns the actuation and the information acquisition using different resources, e.g. visual and / or force and / or tactile sensory exploration procedures.

SORMA's interprocess communication (IPC) pays attention to full platform independence and supports currently two main paradigms: *(i)* standard socket based rpc protocol and the time-optimal path via *(ii)* System V shared memory, which can be used as *global* shared memory across the network. We plan to extend this towards *(i)* asynchronous rpc, *(ii)* network transparent message passing, *(iii)* remote semaphores, and *(iv)*, better DACS integration. Furthermore, we seek *(v)* direct integration of the inter-bus S-bus / VME-bus communication, in order to increase flexibility for the embedded controller in the dedicated robotics VME-system (see Walter and Ritter 1996b).

Appendix

A: Textual Messages and the Dynamic String Concept

The SORMA concept requires good textual message support in two domains: *(i)* the text conversion of the message transport data structure; *(ii)* the error propagation concept cares about understandable, detailed debug and exception messaging (intra- as well as inter-process communication). The efficient string handling, which supports cumulation of arbitrary short and long string messages is an important requirement.

Here arises a data type conflict. A very fast and compact representation is the standard string *char**, where the termination of the text is indicated by a null-character. All library function dealing with strings employ this notion. It does not signal the length of the allocated memory. The usual remedy is to estimate the maximal used string length and add a decent safety margin when programming code using text. When unknown many text messages get appended, this procedure is insufficient. The allocated memory and the string length must be reconsidered and, in case, more memory space re-allocated before the strings are concatenated.

E.g. the textual oriented language Tcl solves this by defining a new data type incorporating the allocated length and the buffer pointer. But this step requires, that some – or all – the string processing function interfaces become redefined and must be relearned. Regular strings must be converted to the particular structure, before they are compatible.

Here, we suggest the *dynamic string concept*. It is type-equivalent to a regular string. Therefore, all function interfaces expecting a string are compatible without any extra effort.

The critical operation is the catenation of two strings. The fastest way is the appending of the second to the first string, if there is enough pre-allocated memory left. Any re-allocation costs extra time (copy, free) and should be avoided. Therefore *dynamic string concept* knows about a reserved free space for the catenation, named `strDynCat()`.

The key idea is to encode the (minimal) allocated memory length in the length of the string. The *dynamic string concept* defines a set of allowed memory lengths. The `strDynCat()` the catenation operation makes sure, that the minimal memory size is taken (from the set), which can contain the result string. From the string length of the target dynamic string the left free space can be inferred. For SORMA the chosen set of foreseen memory sizes are the multiples of a constant (nl_0). For other applications a exponential growth model might be adequate ($2^n l_0$).

The dynamic string catenation functionality (`strDynCat`) is an essential building block for efficiently solving the incremental text message handling with arbitrary sizes.

Glossary

Architecture: "the art or science of building, including design, construction, and often decorations; the character or style of building" (Webster)

Client – Server Model: The client process calls a procedure remotely at a server process located somewhere in the network (see also RPC).

CO: Connection Object in SORMA, handled by the COMM

COMM: optional Connection Object Management Module in SORMA, see Sec. 1.4

CORBA: Common Object Request Management Broker Architecture, see Sec. 3.4

DACS: Distributed Application Communication System, see Sec. 1.9.2

Daemon: a program which waits to take a particular action, e.g. upon a client request (see also client server model)

DLR: Deutsche Gesellschaft für Luft- und Raumfahrt; FTS by the group of Prof. Hirzinger DLR Oberpfaffenhofen

***EXEC/CTRL/DBX*:** indicate the execution, control, and debug method requests to a SO, see Sec. 1.4.3

Field-bus: Using a single bus (containing a few wires) several distributed hardware devices are connected, which are spread out in the "field" (also factory, work cell, room, car etc.) It drastically reduces the numbers of required wires to connect remote sensors, switches, actuators, lights, with their control and monitoring systems. This safes costs and effort for installation and and maintenance (cable failures are easy to detect) and offers intelligent reconfiguration of the distributed system: new devices are hooked onto the bus – the rest can be done by software.

FTS: 6 D Force Torque Sensor for measuring exerted forces (3D) and moments (3D) between to mechanical parts. Here, in particular at the DLR wrist sensor between the robot arm and multi-fingered TUM hand, more details in (Walter and Ritter 1996b)

GUI: Graphical User Interface with buttons, sliders and also complex widgets on a computer screen

IP: the Internet Protocol is the network layer of TCP/IP. It provides the routing of packets in the Internet, but needs higher level layers for reliable communication (retransmission etc., see TCP)

IPC: Inter-Process Communication using e.g. TCP/IP or Unix System V standard resources, i.e. messages, semaphores, and shared memory, see Sec. 1.7

Migration support for SOs means, that changing the location of the service performing process is possible with low effort (local time-optimal, remote protected, load balancing for several machines etc.), see also protected and time-optimal invocation

NDR: Network Data Representation, alternative to XDR, e.g. by DACS or DCE (Trademark Distributed Computing Environment by Open System Foundation, OSF)

NST: Network Simulation Tool, see Sec. 1.1

OMG: Object Management Group, steering group for CORBA

OO: Object Oriented, see OOP

OOP: Object Oriented Programming, see Sec. 1.3.1

OS: the Operating System of a computer system

Protected invocation means, to invoke a component (method call) by a mechanism, that a potential failure does not affect the caller in an uncontrollable way. For instance, memory allocation bugs can exhibit very unpredictable effects (cmp. time-optimal invocation)

Proxy SO: A service object which acts as a substitute for a local SO. It performs by remote service invocation and by mirroring the interface including occuring exception messages, see CO and Fig. 1.6.

PSOM Parameterized Self-Organizing Map, see (Walter 1996)

RCCL/RCI: Robot Command C Library and Robot Control Interface for synchronous control via the SunOS workstation, more details in (Walter and Ritter 1996b)

Real-time task is a task with time constraint specification, see Sec. 1.7.3

RPC: Remote Procedure Call, a standard *TCP/IP* protocol following the *client / server* model.

SCOTT: Service COmmunicaTion Tool executable for interactive test, exploration and usage of services in SORMA. In particular the command-line oriented "scott" inspector and the *Tcl/Tk* oriented "scottwish" for easy generation of GUIs.

Server: a system, here in particular a process, which provides service to clients, see *client/server* and *RPC*. A server process is a *daemon* (here used throughout, in order to distinguish server, service, and *SO*)

Service: a particular functionality provided by a piece of software, which may or may not interface to special hardware devices or any other component. In *SORMA*, the service is encapsulated in the SO:

SO: Service Object within *SORMA*, see Sec. 1.4

SOR: Service Object Request message in textual form (string converted transport date structure)

SORM: Service Object Request Manager, see Sec. 1.4

SORMA: Service Object Request Management Architecture, see Sec. 1.4

Stub: program code which handles the communication with lower network layers, see also *RPC*, *TCP/IP*, and Fig. 1.6

Tcl/Tk: A text-oriented, embeddable Command Language and Tool Kit (Ousterhout 1994)

TCP + UDP: Transmission Control Protocol (TCP) provides a session-based, reliable service for the delivery of sequenced packets across the internet. In contrast the User Datagram Protocol (UDP) delivers datagrams fast but unreliable.

Threads: sometimes called *light-weight processes*, see Sec. 1.7

Time-optimal invocation means, to invoke a component with minimal overhead and a speed which is close to a regular function call (cmp. protected invocation), see Sec. 1.7.3

TUM: Technische Universität München (TUM robot hand by the group of Prof. Pfeiffer)

XDR: eXternal Data Representation for the machine independent encoding and transmission of data across the network.

Acknowledgments

Many thanks to the following contributors: Christof Dücker (robot arm control and vision services), Gunther Heidemann (Datacube vision procedures), Hartmut Holzgraefe (robot hand, PowerGlove, and SpaceMouse services), Ján Jockusch (tactile fingertip sensor services and GUI), Nils Jungclaus (DACS bridge), Rüdiger Kaatz (electronics), Michael Krause (Checkers player and TCl/Tk GUI), Robert Kubisch (active camera head services and GUI), Dirk Selle (hand kinematics and Checkers symbolic game algorithm), Bernd Sieker (voice samples, Tcl, video), Patrick Ziemeck (Androx vision procedures).
This work was supported by the ministry for research and education of NRW.

Bibliography

Allen, P., A. Timcenko, B. Yoshimi, and P. Michelman (1993). Automated tracking and grasping of a moving object with a robotic hand–eye system. *IEEE Trans. Robotics and Automation 9*(2), 152–165. [stationary stereo camera].

Ballard, D. (1991). Animate vision. *Artificial Intelligence 48*(1), 57–86.

Bar-Shalom, Y. and T. Fortmann (1988). *Tracking and Data Association*. Academic Press, NY.

Beck, M., H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner (1994). *Linux Kernel Programmierung*. Addison Wesley.

Bloomer, J. (1992). *Power Programming with RPC*. O'Reilly & Associates.

Booch, G. (1991). *Object-Oriented Design With Applications*. Benjamin/Cummings, Redwood City, CA.

Brunner, B., K. Arbter, and G. Hirzinger (1994, September). Task directed programming of sensor based robots. In *Intelligent Robots and Systems (IROS-94)*, pp. 1081–1087.

Brunner, B., K. Arbter, G. Hirzinger, and R. Koeppe (1995). Programming robots via learning by showing in a virtual environment. In *Virtual Reality World '95, Stuttgart, Feb 21-23*.

Brunner, B., K. Landzettel, B.-M. Steinmetz, and G. Hirzinger (1995). Telesensorprogramming - a task-directed programming approach for sensor-based space robots. In *Int. Conf. on Advanced Robotics (ICAR), Sant Feliu de Guixols, Spain, Sept 20-22*.

Castano, A. and S. Hutchinson (1994). Visual compliance: Task-directed visual servo control. *IEEE Transactions on Robotics and Automation 10*(3), 334–341.

Coombs, D. and C. Brown (1993). Real-time binocular smooth persuit. *Int. J. of Computer Vision 11*(2), 147–164.

Daniilidis, K., C. Krauss, M. Hansen, and G. Sommer (1995). Real time tracking of moving objects with an active camera. Technical report, CS Dept.

Universität Kiel. Nr. 9509.

Dücker, C. (1995). Parametrisierte Bewegungsprimitive für ein Roboter–Kraft/Momenten–Sensor Handsystem. Diplomarbeit, Technische Fakultät, Universität Bielefeld.

Fink, G., N. Jungclaus, H. Ritter, and G. Sagerer (1995). A communication framework for heterogeneous distributed pattern analysis. In V. L. Narasimhan (Ed.), *International Conference on Algorithms and Applications for Parallel Processing*, Brisbane, Australia, pp. 881–890. IEEE.

Fink, G. A., N. Jungclaus, F. Kummert, H. Ritter, and G. Sagerer (1996). A distributed system for integrated speech and image understanding. In *International Symposium on Artifical Intelligence*, Cancun, Mexico, pp. submitted.

Gengenbach, V. (1994). *Einsatz von Rückkopplungen in der Bildauswertung bei einem Hand-Auge-System zur automatischen Demontage*. DISKI 72. Infix, Sankt Augustin.

Grewal, M. and A. Andrews (1993). *Kalman Filtering*. Prentice Hall.

Hirzinger, G., B. Brunner, J. Dietrich, and J. Heindl (1994). ROTEX – the first remotely conrolled robot in space. In *Intern. Conf. on Robotics and Automation (San Diego)*, pp. 2604–2611. IEEE.

Jacobsen, I. and et al (1992). *Object Oriented Software Engineering*. ACM press and Addison-Wesley.

Kubisch, R. (1995). Aktives Sehen mittels eines binokularen Kamerakopfes: Ein Ansatz auf der Grundlage neuronaler Netze. Diplomarbeit, Technische Fakultät, Universität Bielefeld.

Littmann, E., A. Meyering, J. Walter, T. Wengerek, and H. Ritter (1992). Neural networks for robotics. In K. Schuster (Ed.), *Applications of Neural Networks*, pp. 79–103. VCH Verlag Weinheim.

Marsh, B., C. Brown, T. LeBlanc, M. Scott, T. Becker, C. Quiroz, P. Das, and J. Karlsson (1992, Feb). The rochester checkers player: Multimodel parallel programming for animate vision. *IEEE Computer 2*, 12–19.

Murray, D. and A. Basu (1994). Motion tracking with an active camera. *IEEE Trans. on Pattern Analysis And Machine Intelligence 16*(5), 449–459.

OMG (1995, July). The common object request broker: Architecture and specification. Specification revision 2.0, Object Management Group, http://www.omg.org/.

Orfali, R., D. Harkey, and J. Edwards (1994). *The Essential Distributed Objects Survival Guide*. John Wiley & Sons.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.

Papanikolopoulos, N., P. Khosla, and T. Kanade (1993). Visual tracking of a moving target by a camera on a robot. *IEEE Trans. Robotics and Automation RA-9*(1), 14–31.

Ritter, H. (1995). NST tutorial, NST reference manual, NST in 8+ϵ pages. Institut für Neuroinformatik, Universität Bielefeld.

Ritter, H. (1996). Neural network Simulation Tool (nst). Technical Report SFB360-TR-96-5, Universität Bielefeld, D-33615 Bielefeld. (in preparation).

Stal, M. (1995). Der Zug rollt weiter. *iX 5*, 160–168.

Stewart, D. B., D. E. Schmitz, and P. K. Khosla (1992). The Chimera II real-time operating system for advanced sensor-based control applications. *IEEE Trans. on System, Man, and Cybernetics 22*(6), 1282–1295.

van der Smagt, P. (1995). *Visual Guidance using Neural Networks*. Ph. D. thesis, University of Amsterdam.

Walter, J. (1991). Visuo-motorische Koordination eines Industrieroboters und Vorhersage chaotischer Zeitserien: Zwei Anwendungen selbstlernenden neuronalen Algorithmen. Diplomarbeit, Physik Department der Technische Universität München.

Walter, J., T. Martinetz, and K. Schulten (1991, June). Industrial robot learns visuo-motor coordination by means of the "neural-gas" network. In *Proc. Int. Conf. Artificial Neural Networks (ICANN), Espoo Finland*, Volume 1, pp. 357–364. Elsevier, New York.

Walter, J. and H. Ritter (1996a). Associative completion and investment learning using PSOMs. In M. C. v.d. S. W. v. J. Vorbrüggen, and B. Sendhoff (Eds.), *Artificial Neural Networks – Proc. Int. Conf. ICANN 96, July Bochum*, Lecture Notes in Computer Science 1112, pp. 157–164. Springer.

Walter, J. and H. Ritter (1996b). The ni robotics laboratory. Technical Report SFB360-TR-96-4, TF-AG-NI, Universität Bielefeld, D-33615 Bielefeld.

Walter, J. and H. Ritter (1996c). SORMA: Interoperating distributed robotics hardware. In *Proc. Int. Conf. on Robotics and Automation (ICRA-97)*, pp. (submitted).

Walter, J. and K. Schulten (1993). Implementation of self-organizing neural networks for visuo-motor control of an industrial robot. *IEEE Transactions in Neural Networks 4*(1), 86–95.

Walter, J. A. (1996). *Rapid Learning in Robotics*. , Technische Fakultät, Universität Bielefeld. http://www.techfak.uni-bielefeld.de/∼walter/pub/.