SORMA: Interoperating Distributed Robotics Hardware

Jörg A. Walter

Department of Computer Science · University of Bielefeld · D-33615 Bielefeld
Email: walter@techfak.uni-bielefeld.de · http://www.techfak.uni-bielefeld.de/~walter/

Abstract: This paper introduces the **Service Object Request Management Architecture** ("SORMA"), its design issues, and its concepts. It is a software framework for rapid development of object-oriented software modules and their integration into stand-alone and distributed applications. SORMA provides an intelligent "object-bus" for inter-operating and sharing distributed computing and robotics hardware.

We investigate the question, why too many valuable hardware and software systems and components are a "one-of-a-kind" product which do not find economical re-use. For instance, due to the short life-time of single-usage code, extensive, *robust*, and verbose exception handling – a prerequisite of incremental work – is often sacrificed . We analyze the *"costs of communication"* between the component's builder and all its users, who build solutions.

We propose to pay much more attention to interactive exploration frameworks, which support rapid, qualified information gain on context-situated, efficient applicability. SORMA demonstrates how to reach self-explaining, built-in interactivity which does not impair the component's real-time efficiency.

Keywords: *Service Object Request Management Architecture; shared, distributed resources; economy of reusing components; communication cost; built-in interactivity; time-optimal and protected invocation.*

1 Introduction

The experience of building and operating robot-vision labs [5, 10, 12] shows, that a substantial amount of effort easily dissipates in adaption of software components to application specific needs. A lot of interesting ideas and application code is developed, but too often those remain as "one-of-a-kind" pieces. Descriptions like short life-time, little use and re-use, little contribution to later projects and other peoples similar problems, are not exceptional for (too) many valuable, fine components.

Why are many software as well as expensive hardware components not better re-used and not better shared among working teams? We emphasis the term *"communication cost"* as a very useful view point and discuss in the Sec. 2 the *economy of re-use*. We are not going to speculate about monetary figures on the over-all economic potential of making components better re-usable (they are large) – instead we investigate this question starting with the component, its users and its maker.

In Sec. 3 we discuss ingredients for a software framework, which forms a suitable, standardized form for developing more sustainable components, which achieve easier *re-usability*, and give the basis for *incremental work*. Sec. 4 presents some essential ideas of "SORMA". Sec. 5 reports on the experiences we made with this software framework within the context of a university robotics laboratory.

2 Economy of Reusing and Sharing Software and Hardware Components

The economics of reusable components are dominated by the *cost of communication* between *component builder* (*"abstractor"*, e.g. the programmer of a software library) and the *solution builders*, who integrates several components into new applications (also called the *"elaborators"*) [2].

Reducing Communication Costs. A prerequisite for a successful reuse and integration is to *communicate* the intented purpose of the component, which problem it solves, when and how it is applicable. Here, to communicate means, to *make known* how to apply the component correctly and efficiently (Webster). The sender is the maker, the "abstractor", the receivers are the countable number n of users, the "elaborators".,

How can we reduce the time necessary to understand how to reuse a component? If the overall time it takes all n elaborators to figure out the reusable component - is an significant issue, this question must be seriously considered.

The Usual Communication Mechanisms are tutorials, reference manuals, source code comments, – and of course, verbal explanation and advice. Clear structuring of documents is a classical requirement. Very helpful are well organized on-line documentation systems with search capabilities on indexes and full text. Electronic access is inexpensive (to use and to update), concurrently available to many people and places, and very rapid for searching.

The abstact communicated information is – to a certain extent – always prone to be out-of-date, wrong, or incomplete (to a particularly larger extent if we want/need to share already early, non-perfect versions). Furthermore, this communication involves the process of encoding of a fact using natural language, which often leaves space for more than one unique interpretation (special programming languages exist for this reason).

In cases where these ambiguities become obvious, one can clarify them, e.g. by asking an expert (if available). Misinterpretations are the worst case. Typically, they are very costly and can render huge amounts of time and money useless.

Make Known by Exploration. The *direct exploration* of the object is a further mechanism of communicating the component's intention, its constraints, and usage. This *trial-and-error* procedure is a fundamental form of learning, it can be unsupervised or guided by some teacher. Despite its effectiveness, and probably due to its playful appearance, exploration is usually not perceived as a serious and canonical communication mechanism by its own.

The efficiency of exploration depends on the *costs and benefits* of each trial and possible errors. This has several constituents: Usually, primary *test cost* factors are the *time* to set up a test program (or better interactive test suite), the time to perform, and to evaluate the tests (including all needed compile-link-load-run loops; not to forget the recommended clean-up of possible temporary hacks). The potential *error costs* depend on the system fault tolerance, the time and effort to recover, and the real costs (ranging from a graceful fault signal, an abort, an irrecoverable system error, to a robot going wild). The *benefit* of an occured error is the information gained from the system (learn value) and depends on how informative the error messages are and how understandable the response is to the tester (e.g. "#0042" versus what-why-where). Of course, the benefit of a successful test is, beside the gained experience, the verification of the tested component's properties.

This analysis emphasizes the extra value of easily available, rapidly operating, user- and reuser-friendly experimentation frameworks. Asking the component itself, to explore and also to clarify any other ambiguity becomes than inexpensive.

3 Design Issues for a Software Framework

In this section, we address several points, which appear valuable for reducing communications costs, but not widely recognized in the field of robotics.

Pattern. For guiding software reuse some researchers proposed the method of developing software "patterns" . A *pattern* is a repeated conceptual paradigm which shortens the communication such, that it replaces *learning* by *re-cognition* when the pattern is *re*-encountered. (A good pattern example is the "Save" and the "Save-As" button, you might find, when you are working with a new graphical document editor. If you instantly perceive the proper meaning of these buttons, you saved the time otherwise required for studying the manual.) A successful pattern allows more rapid understanding by standardization, better solution turn-around times, and thus improves the component's economy.

Object-Oriented Programming ("OOP") is an important idea of gaining better re-use value for software components (e.g. [3, 8]). An object is a piece of code that owns private data and provides service through methods (=procedures). It is a run-time instance of a *class* which describes the behavior of a set of alike objects. By the properties *inheritance* and *polymorphism* OOP formalizes the idea of a pattern. Furthermore, the concept of *encapsulation*, providing only procedural access to private resources, promotes easy re-usage. Since the implementation details are hidden behind the published interface, unexpected interferences when integrating components can be significantly reduced.

Context-Oriented Configuration. The abstractor, in building a reusable piece, is solving a whole set of future problems. For instance, most reusable software results from the experience of being an elaborator several times, then having a flash of insight that solves a number of elaborators' problems once and for all. But often there is an inherent solution conflict between the level of abstraction and generality

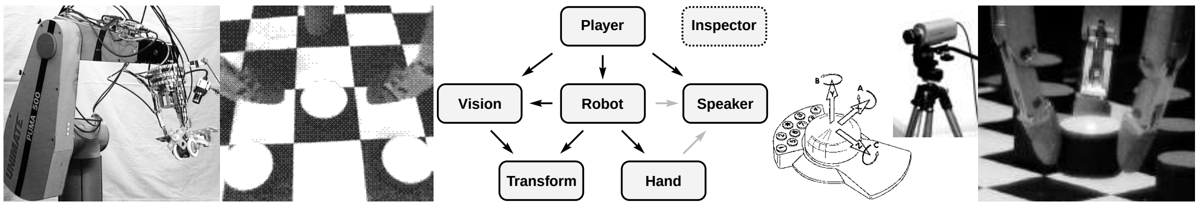

Fig. 1: Puma robot manipulator with 3-fingered TUM hand in the Checkers player scenario.

versus the level of suitability for a particular problem context. One way to make a component more powerful is to formulate the desired context and the appropriate behavior in a set of configuration parameters. In order to keep the required communication and learning effort low, methods for efficient and simple usage are important. In cases of larger numbers of context parameters, the proper organization of consistent sets is a significant burden and should be solved in a systematic and convenient way. By *hiding information*, the (re-)user can focus on the essential parameters for his particular context.

Concurrent Team Development includes the *early sharing* of components and is (in certain phases) related with the problem of integrating also pre-mature and rapidly changing code versions. From our experience, three items are worth mentioning: *(i)* high-level Unix operating systems offer superior (inherent not only attached) *multi-user and network capabilities* (tcp/ip, nfs, amd, remote login, X11, etc.). They provide the solid ground for sharing distributed resources via standard networks. Furthermore, numerous, high-quality, standard tools are everywhere available (e.g. teams of size larger than one might find aid with the “Concurrent Version System”, CVS); *(ii)* strict OO encapsulation and modularization simplifies ongoing and independent component development; *(iii)* run-time encapsulation of application components using *protected invocation*.

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.

Migrating Components. In order to allow early integration of complex components, it is extremely valuable to have the opportunity to quickly isolate any application part by secure fire walls. Easy component *migration* needs support for organizing and re-organizing process and communication structures. Additionally, the program code must be able to use the very same call syntax for local and remote component communication.

Error Messaging Support when Objects Migrate. When software components are used in both, an local and a distributed fashion, the abstractor, the elaborators, and the user need extra support. If it is not available, the programmers get frustrated: what to do with the message in case the call is local or remote? Send to whom and how? Conventional solutions, like sending messages to the standard error channel are unreliable, since they might stay unseen in some other terminal. Therefore, it is very desireable to use a unified and compact exception message generation interfaces, delegating the appropriate handling to the infrastructure.

Summary of Design Issues. We found following points important:

- support for an easily available, rapidly operating exploration framework for each component
  - clear-text access to *all* configuration and *all* usage options
  - *built-in interactivity* to provide a life-long test and exploration suite
  - built-in documentation – synopsis and (help) option menu
  - detailed, understandable error messages
- sharing distributed resources
  - *interoperating* computing as well as special robotics resources
  - maximize *re-usability* and *portability* across all our available (Unix) operating systems
  - efficient configuration support
  - *interactive* command-line and graphical user-interface tools (GUI)
  - *scripting* capabilities (interpreter mode)
  - *real-time* capabilities for all time-critical parts, while keeping interactive support (!)
- improving *robustness* and *fault tolerance* by

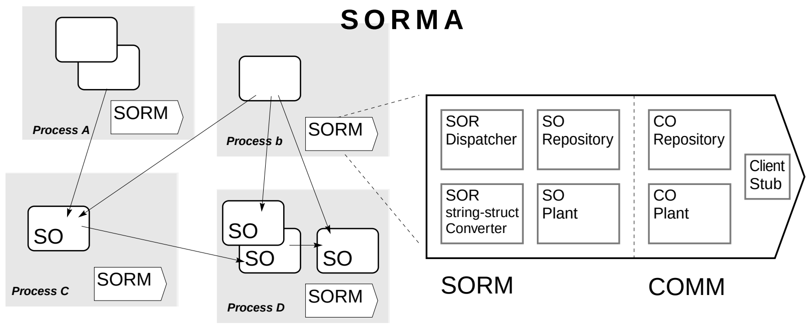

Fig. 2: ***Service Object Request Management Architecture*** (**SORMA**) is a framework and infrastructure to connect to dynamically created Service Objects (SO) via *time-optimal* intra-process and *protected* inter-process communication. *(Left:)* Various software components communicate in four processes running on one or several hosts. *(Right:)* Local and remote requests to SOs are managed by the SORM and COMM.

– strong support on exception message transport and handling (important for network transparency)
– self-managed process communication mechanisms with resume capabilities, also for complex state machines (e.g. robot manipulators)

Usually, these design goals are considered conflicting. The next section introduces the ***Service Object Request Management Architecture*** ("**SORMA**"), and describes some of the concepts which contribute to solve the above stated goals. For a more detailed account see [11].

4 SORMA Concepts

Central to SORMA is the notion of a "service", which offers a certain functionality, e.g. abstract computing, providing low- or high-level access to hardware devices including sensors and actuators. A service is realized in form of a *Service Object* ("**SO**"), an instance in the OOP sense (see Sec. 3). Each SORMA process has a *Service Object Request Manager* ("SORM"), which is responsible for *(i)* dynamic constructing (building), *(ii)* maintaining, and *(iii)* sharing of SO. This has to be managed in a time and memory efficient manner.

Fig. 2 illustrates several properties: SORMA facilitates easy SO-requests between several SO – within a single process, as well as across process and machine boundaries. A SO can be shared employing a RPC-TCP Internet Protocol. Then requests from distributed callers are served by a "daemon" in an hierarchical *client–server* model. Remote requests are handled by the optional COMM, the "Communication Object Management Module" using a proxy-SO, as explained below. A SORMA process may provide itself several *service "classes"*, each enabling to instantiate one or more SOs, which we also call *service flavors*, see below.

The next paragraphs describe the SO interface, the local and remote instantiation, and available interactive tools SCOTT (Service COmmunicaTion Tools). But first, let us advertise one immediate advantage of complying to the SORMA interface idea.

SO Interface Incentives. SORMA supports the reuse of written and compiled code in four main process configurations depicted in Fig. 3. The only prerequisite is, that the software interface follows the idea of a SORMA service class. *(i)* a regular self-contained, stand-alone *program* with local (fast intra-process) SO requests; *(ii)* by linking the SORMA "server stub" a network *daemon* is created, which can be interoperated by other SORMA clients, including the standard inspector tool "scott"; *(iii)* interactive local (and remote) SO requests 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.

Actually, by the help of the SCOTT standard tools (see below), the functionally of type *(iii)* and *(iv)* are already available when configuring the service in one single *(ii)* deamon. A daemon might be not sufficient, when tight *real-time requirements* occur. Then the *protected invocation* of a remote SO might take to much time and the alternative, direct code linking (type *(i)* Fig. 3) is advantageous. The direct, intra-process invocation of a SO is ***time-optimal*** to the extent that it costs about the extra time of only one subroutine call. And, as already indicated, SORMA offers the *identical call syntax* for both invocation paradigms. This symmetry feature facilitates the flexible migration of service tasks from and to other processes.

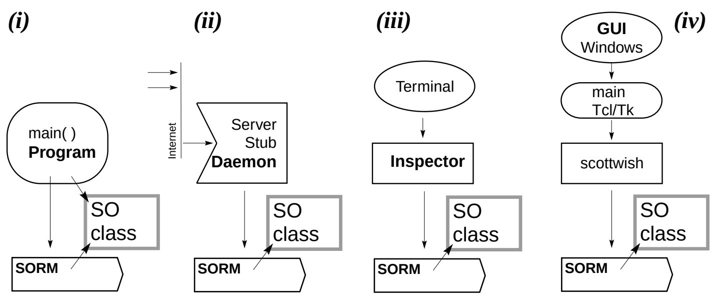

Fig. 3: SORMA instant re-use scenarios. *(i)* a regular *program* with fast intra-process SO requests; *(ii)* a network *daemon* offers service to other processes via the internet; *(iii)* command-line access to *all* SO methods and options; *(iv)* GUIs by SORM-Tcl/Tk coupling.

SO Interface — a Compact Pattern. How can SORMA combine the support level for real-time, interactivity, and re-use? One clue is the interface design of the service object. It is standardized and small – which allows extended support for the abstractor, the elaborator, as well as the final user.

SO — Methods: Beside the OO standard constructor (new) and destructure (free) methods, the main procedural interfaces are subsumed in two major methods: *EXEC* and *CTRL*. Compared to a device driver interface, the *EXEC* method does the main job, like *read*() and *write*() for data exchange to a disk drive. The method can be designed to operate most rapidly – any special configuration when initializing the object (here SO or device). For reconfiguring special properties (e.g. attributes, conditional behaviors) there is another interface. The *CTRL* method is conceptually comparable to the *ioctl*() call, but it is not likewise cryptic. E.g. *CTRL* is text-oriented and has a mandatory help and self-explaining dump option.

Two methods allow two separate parser mechanisms to optimize different design goals, usually mixtures involving ease-of-use, comfort, information hiding, fault-tolerance, parser effort, execution time, etc. A very time-critical service will abandon an *EXEC* parser and may instead specialize itself by dynamic binding of the desired *EXEC* method (run-time binding and re-binding upon *CTRL* calls).

SO — General Data Structure: All methods share the same data structure for input and output. Sending a non-empty message (method request or reply) to another process involves three phases: the message gets packed in some standard format, introduced into a transmission medium, finally, the packet gets received at the target and unpacked. For three reasons SORMA employs one single standard data structure for the message transports: *(i)* standard packing/unpacking routines (XDR) are static and pre-compiled (speed); *(ii)* it simplifies text-conversion support (efficient translation mechanisms) to and from clear-text messages, and *(iii)* it simplifies their usage – since it is a pattern.

We found the following combination of data types a good compromise: a *float* tuple (real, for continuous values; a tuple is a vector or 1D array of dynamic length) an *int*eger tuple (e.g. flags, counters, discrete signals), a text string (for human-machine communication; *char** null-terminated) and a *char*acter tuple (also “opaque” or “any”, for e.g. image data or arbitrary complex data structures packed elsewise, e.g. [4, NDR]). A result message makes use of an additional return value, indicating success or a fault id, and potentially a verbose human readable clear-text debug message for notification, warnings and errors. Receiver and sender id complete the transport data structure.

We found the described data type choice lean – but general enough to fit all our essential requirements. Herewith intra-process communication is organized as a time-optimal call-by-reference – while remote method calls can employ standard and rapid packer routines. Before the remote invocation mechanism, the next section explains the local instantiation.

Service Flavors by Name. One key concept of SORMA is the service object instantiation on the basis of a unique name. The name consists of possibly three parts, e.g. “*foo.a@daemon*”: *(i)* the class name “foo”; *(ii)* the (optional) SO flavor specialization “foo.a”; *(iii)* the optional “@daemon” network address, if it is present, it indicates a remote service which is then resolved via the proxy mechanism to the SO name “foo.a” – then local at the daemon side, see below (for options on opaque network addressing see [11]).

At creation time, a local name (like “foo.a” without “@” part) gets *expanded into a list of CTRL arguments* – text tokens, which are interpreted by the service's *CTRL* method (see Fig. 4). This provides a general hook for *arbitrary configuration and state transition calls*. Different names lead to the instan-

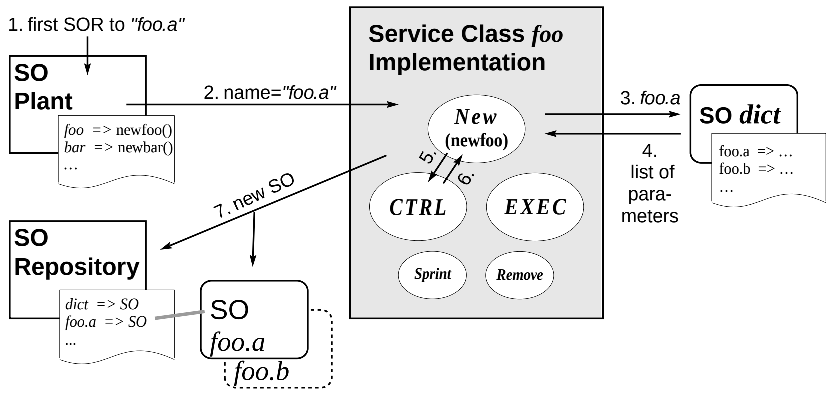

Fig. 4: The local service object instantiation and specialization procedure when the SORM is called for the name "$foo.a$" the first time. The registered class "foo" constructor function gets invoked "newFoo("foo.a")", which retrieves the "recipe" from a dictionary service. The "recipe" is a token list, denoting a sequence of state transitions. It gets parsed by the object's *CTRL* method and specializes the desired service object flavor.

tiation of different service objects according to their "recipe". Since the resulting SOs belong to the same service class but exhibit modified characteristics (attributes, configuration, properties etc.), we call them also *"service flavors"*.

The flavor name concept solves the conflict between the three (before) opposed goals of: *(i)* serving hardware by *shared* server access, *(ii)* which is *easy-configurable*, *(iii)* but still *state-free* (as far as possible). Because the script is associated with the service object name, the COMM can efficiently manage the requested connections. The SO names, which are part of all initial SO reference calls, are not only sufficient to build, but also - in case of failing daemon - to re-establish the connections to the appropriate service flavors, served by a restarted SORMA daemon.

Beyond this robustness advantage, the name-to-script scheme is a versatile configuration and default mechanism. SORMA offers a central *dictionary* facility for expanding SO names. It offers various options – the most straight forward loads entries from a plain text file (lines plus commenting notes). Of course, also this service is implemented as SO, which has the immediate advantage, that dictionaries in daemons can be remotely edited – during runtime.

Proxy SO. Fig. 5 illustrates how the connection to a remote service SO is created. The first request involves the dynamic construction and caching of the remote target SO (12), a proxy SO (3), and a Connection Object ("CO",6). The upper scheme displays several steps: Since the client SORM fails to retrieve 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 (5), if not already available (4). The SO request

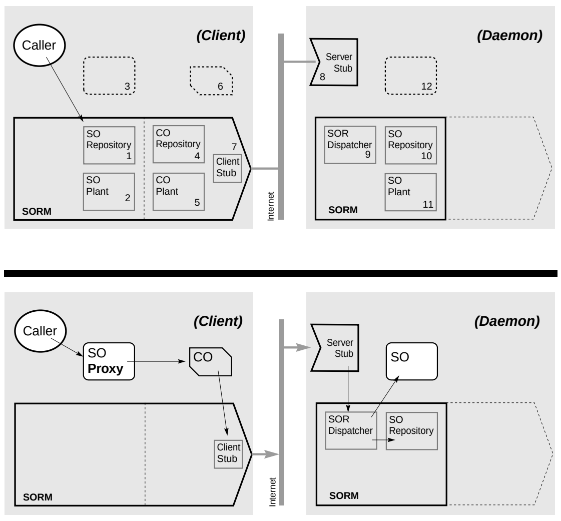

Fig. 5: Creation and later usage of a remote Service Object within the daemon process (rigth side) by a local *Proxy* SO at the caller side, see text.

reaches the daemon SORM on the right side via TCP Internet protocol and stubs 7 and 8. There, a SO-reference-by-name mechanism induces the instantiation of the desired SO.

The lower half of Fig. 5 depicts how the cached objects accelerate all further calls. Requests to the remote SO look identical to local SO-requests – since they *are local* requests to the ***SO-proxy***. The SO-proxy acts as substitution of the remote SO and *completely mirrors* its interface, including the input, potential exception, and output messages. 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.

Exception Message Support. To standardize exception message generation, SORMA has one universal, compact procedure, which knows about verbose, warning, and error messages. The transport back to the requester works likewise for local and remote method calls – in a cumulative way – also across process and machine boundaries. Human readable clear-text messages, but also unique identification codes (for error handlers) supplemented with context information (source code location) are standard.

Text Conversion turned out to be a very important ingredient for reducing communication costs with SORMA. 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.

Interactive Service COmmunicaTion Tools (SCOTT). As mentioned before, two standard tool executables are readily available to remotely access all (possible) SO in the network. *(i)* A *command line* oriented inspector "scott" offers test-suite features like line completion, editing, and recent history matching. Furthermore, it invites to program service requests by simple *scripts*. *(ii)* Any desired SO method can be invoked via a mouse-click on a button, or a slider etc. The "scottwish" is a SORMA enhanced Tcl/Tk shell and facilitates *graphical user interfaces* (GUI), simply by writing short scripts, see Fig. 6.

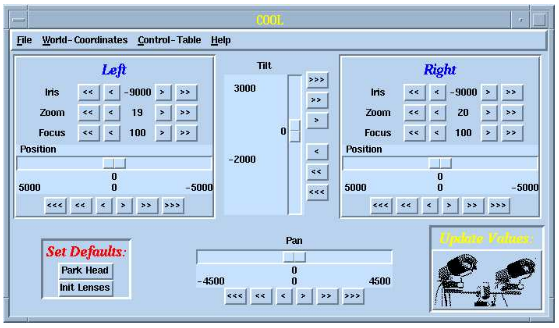

Fig. 6: Example of a graphical user interface (GUI, by R. Rae), produced by the "scottwish" and a Tk script. The active stereo camera head can be interoperated by mouse clicks and drags

5 Experiences with SORMA

The first implementation example was a checkers player system, developed with a team of students. Due to the limited space, we must refer any details to the description in [11]. In this hybrid application the distributed object infrastructure was in the foreground. Complying to the international rule of checkers, a robot manipulator (PUMA), equipped with a hydraulic dextrous hand (TUM) and supported by an image processing system, is empowered to play this board game against a human player. The task was split in several service components grouped to six main processes running on two (or 3, 4) Unix workstations (see Fig. 1). The GUI enhanced symbolic player process interoperates all other hardware components by SO requests, which themselves call other daemons for service (e.g. the token transfer SO carry involves the robot, hand, vision etc.). During operation all daemons can be interactively inspected.

Here the real-time demands were *soft* (best effort) – and the achieved SORMA interprocess communication times are more than sufficient to meet the desired inter-activeness, including speech generation (between 1.5 and $4.0 \cdot 10^{-3}$s mean time for request plus reply communication with a daemon; across our computer park).

The real-time capabilities were demonstrated in a 3 D robot tracking application, combining vision and force senses and a rapid learning PSOM network [11]. This *hard real-time* task required strict deadlines with zero time fault tolerance. Thanks to the RCCL/RCI package we conveniently achieve synchronous robot control employing a standard SUN SparcStation [6, 10]. One standard SORMA client-server CO asynchronously connected the vision analysis and the robot host. All other communication used the SORMA time-optimal intraprocess invocation (about $8 \cdot 10^{-6}$s elapsed call overhead time) and further interprocess communications services via shared memory (about $3 \cdot 10^{-5}$s).

In contrast to other robot control software frameworks like Chimera [9] or Psyche [7], SORMA does not attempt to be a real-time operating system (OS) itself (e.g. no scheduler). Chimera and Smart [1] focus on multi-processor VME-bus systems which are hosted by a Unix workstation. SORMA offers to interoperate also software components served by high-performance workstations across the network - using the standard interface.

Currently we care about full portability and interoperability across the following operating systems: Aix, Iris4d, Irix5, Irix6, NextStep, OSF1, Linux, Solaris, and SunOS (which gives also the reason to still hesitate about relying on P-threads implementations, which are yet not available on *all* platforms above).

The central interest is to provide a framework for effective building of components and rapid assembly to distributed solutions. It is interesting to note, that despite the fact that SORMA was independently developed, it shares many ideas formulated in large industrial initiatives to build intelligent distributed object infrastructures, in particular CORBA by OMG [8]. Actually, the "Common Object Request Broker Architecture" recently inspired the revised terminology within S**OR**MA. Instead of CORBA's goal of serving many vendors' interests with multiple protocols, we are working with complex robotics hardware, requiring specially adapted solutions. Here SORMA emphasizes in particular interactive, built-in exploration and usage support, robustness, and real-time efficiency.

Furthermore, SORMA supports early resource sharing and re-use of components: providing better services will invite to re-usage, which gives incentive for better services – thus a positive feedback loop can emerge. We believe, this helps to improve the service's economy.

References

[1] Robert J. Anderson. SMART: A modular architecture for robotics and teleoperation. In *Proc. IEEE Robotics and Automation, Georgia*, volume 2, pages 416–421, 1993.

[2] Kent Beck. Patterns and software development – adding value to reusable software. *Dr. Dobb's Journal*, February:18–21, 1994.

[3] G. Booch. *Object-Oriented Design With Applications*. Benjamin/Cummings, Redwood City, CA, 1991.

[4] Gernot Fink, Nils Jungclaus, Helge Ritter, and Gerhard Sagerer. A communication framework for heterogeneous distributed pattern analysis. In V. L. Narasimhan, editor, *International Conference on Algorithms and Applications for Parallel Processing*, pages 881–890, Brisbane, Australia, 1995. IEEE.

[5] Enno Littmann, Andrea Meyering, Jörg Walter, Thomas Wengerek, and Helge Ritter. Neural networks for robotics. In K. Schuster, editor, *Applications of Neural Networks*, pages 79–103. VCH Verlag Weinheim, 1992.

[6] J. Lloyd and V. Hayward. Multi-RCCL user's guide. Technical report, McGill Reseach Center for Intelligent Maschines, McGill University, Montréal, April 1992.

[7] Brian Marsh, Chris Brown, Thomas LeBlanc, Michael Scott, Tim Becker, Cesar Quiroz, Prakash Das, and Jonas Karlsson. The rochester checkers player: Multimodel parallel programming for animate vision. *IEEE Computer*, 2:12–19, Feb 1992.

[8] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1994.

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

[10] Jörg Walter and Helge Ritter. The NI robotics laboratory. Technical Report SFB360-TR-96-4, 1996.

[11] Jörg Walter and Helge Ritter. Service Object Request Management Architecture: SORMA concepts and examples. Technical Report SFB360-TR-96-3, Universität Bielefeld, D-33615 Bielefeld, http://www.techfak.uni-bielefeld.de/~walter/pub/, 1996.

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

Acknowledgments:
The experience using and extending NST, the "Neural network Simulation Tool" developed by Helge Ritter, had important influence in the time efficient design of the SO interface. Many thanks to H. Ritter, C. Dücker, J. Jockusch, N. Jungclaus, and G. Menkhaus for proofreading an earlier version of this paper. The checker player aplication came into play with the valuable contributions of C. D., H. Holzgraefe, R. Kaatz, M. Krause, D. Selle, B. Sieker and P. Ziemeck. The tracker application was joint work with C.D. and G. Heidemann. The depicted SORMA-GUI for the active stereo camera head was designed by R. Rae. This work was supported by the ministry for research and education of NRW.