Ikaros: Building Cognitive Models for Robots

Christian Balkenius, Jan Morén, Birger Johansson and Magnus Johnsson

Abstract— The Ikaros project started in 2001 with the aim of developing an open infrastructure for system-level brain modeling. The system has developed into a general tool for cognitive modeling as well as robot control. Here we describe the main parts of the Ikaros system and how it has been used to implement various cognitive systems and to control a number of different robots ranging from robot arms and hands to active vision systems and mobile robots.

I. INTRODUCTION

The goal of the Ikaros project is to develop an open infrastructure for system level modelling of the brain including databases of experimental data, computational models and functional brain data. The infrastructure supports a seamless transition from a pure modelling and simulation set-up to real-time control systems for robots running on one or several computers in a single or multiple threads. Computational models are built by connecting individual modules that implement a specific brain model or algorithm into larger systems.

The system makes heavy use of the emerging standards for Internet based information such as XML and makes all part of the system accessible through an open web-based interface. We believe that this project has the potential to radically change the way system level modeling of the brain is performed in the future by defining standard benchmarks for brain models and substantially increase the gain from cooperative research between groups.

A system like Ikaros can not operate in a vaccuum. Instead, the goal is to allow Ikaros to easily work with as many external sources of information as possible. There is simply too many types of information that need to be used by the system and without taking an inclusive approach, the task of adapting information and models becomes too great. The only viable solution is to integrate Ikaros with other similar endeavors whenever possible. This inclusive approach means that we want to offer a large corpus of experimental data from cognitive experiments for use with Ikaros, but we also strive to make it easy to adapt other experimental data for use within the system.

Inclusivness also means making development a transparent and straightforward process. As part of the standard infrastructure, Ikaros already contains a sizable number of standard modules that are useful in a broad range of cognitive models. The infrastructure also contain modules that allow for an

C. Balkenius, B. Johansson and M. Johnsson are with Lund Univeristy Cognitive Science, Kungshuset, Lundagård, SE-222 22 Lund, Sweden. christian.balkenius@lucs.lu.se

Jan Morén is with Knowledge Creating Communication Research Center, NICT, 2-2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan jan.moren@gmail.com

easy interface with various types of hardware such as video cameras and robots. For example, there are easy interfaces to the various standards for video capture and video files, for audio processing as well as for robot control through a set of drivers for different hardware systems.

The goal of the infrastructure specification is to be minimally demanding for anyone developing an Ikaros module. It should be possible to learn to use it in a few minutes while still providing support for very complex architectures. In the following sections we describe the different parts of the Ikaros system and the choices that have been made when designing the different components.

II. SYSTEM-LEVEL MODELS

The core concept of system-level modeling is the module which corresponds to a part of a model. A module can have a number of inputs and outputs and encapsulates a particular algorithm (Fig. 1). This does not mean that cognitive models built using Ikaros must adhere to a modular view of cognition. Instead, a system-level approach to cognitive modeling acknowledges that different cognitive components interact in many ways and it is one of the strengths of the approach that it explicitly shows these interactions as connections between modules. A module in Ikaros is thus not a statement about locality or impenetrability, it is only an acknowledgement that a system is constructed from several components, and these components or modules have different properties.

In general, to design a system-level model it is necessary to answer four questions:

What are the components of the system? This entails answering at what level the model should be described. Are the components individual neurons or brain regions, or are they some form of abstract description of functional components without direct relation to the brain? There is no single correct answer to these questions; it depends on the model being implemented.

What are the relations between the components? Are they parallel systems with little interaction, or are they tightly coupled? Are they all at the same descriptive level or are some components subparts of others? Is the system heterogeneous or hierarchical?

Which function is performed by each component? How can the functions be described as mathematical functions or as algorithms? Ikaros supports systems built from standard modules that implement elementary mathematical functions as well as modules that are hand coded from scratch.

What information is transmitted between the components and how is it coded? The question of coding is the most important for a system-level model and the only one where

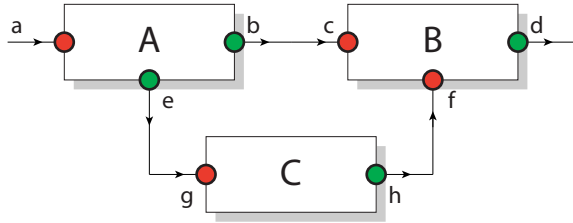FIG. 1: *A module with one input and one output.*

FIG. 2: *A small system with three modules A, B, C with connections between them.*

Ikaros puts any major constraints on the possible models. In Ikaros, all inputs and outputs are coded as matrices of floats. This limits the possible models in several ways that make it more likely that different models can be interconnected. Although Ikaros puts no constraints on the interpretation of the matrices, this type of structure is best used for coding in terms of numerical values, either directly or using some form of distributed code.

In Ikaros, the components are specified using an XML-based language which also describes the relation between the components. The function in each component is described either using standard modules or by writing new simulation code. The transfer of information between components is implicit in the coding of the different modules.

III. DESCRIBING MODELS

Fig. 1 shows a simple module. This module has a single input through which it receives input data and a single output through which it sends its output data. The input is read in discrete time and the module also generates new output at discrete intervals.

Modules can be connected together to form systems (Fig. 2). This network of modules is what makes up a model in Ikaros. Here, the model consists of three modules A, B and C. Module A has one input (a) and two outputs (b and e). Module B has two inputs (c and f) and a single output (d). Finally, module C has one input (g) and one output (h). The complete model has the single input a and the single output d.

One of the greatest strengths of Ikaros is its ability to handle large complicated cognitive models consisting of many interacting subcomponents. To allow the specification of such architectures, an XML-based description language has been developed [6]. This language has three main components: the module, the group and the connection.

A module element describes an instance of a particular Ikaros module and sets its parameters. These parameters are handled to the constructor function of the module as described below. The only two required attributes are *class* and *name* that decides what code the module will run and how it will be referred.

```
<module
    class = "MyClass"
    name = "MyModule"
    alpha = "3"
    beta = "0.1"
/>
```

A connection between two modules is specified in a connection element:

```
<connection
    sourcemodule = "Thalamus"
    source = "Output"
    targetmodule = "Amygdala"
    target = "Input"
/>
```

Finally, it is possible to group modules and connection in to larger structures. The following example corresponds to the structure shown in Fig. 3 and Fig. 4. It defines a group (or new module) called X with an input x and an output y. The group consists of three modules A, B and C which have multiple connections between them. The input x is connected to the input a of module A and the output y receives data from output d of module B.

Groups can also be given inputs and outputs to let them function as new modules or be read from external files and be used as call descriptions. A specification of these features is however beyond the current description.

IV. THE SIMULATION SYSTEM

Currently, the main part of Ikaros is the simulation system which consists of a platform independent simulation kernel together with a large set of modules that implements different functions and models.

A. Design Criteria

There were a number of important considerations in the choice of the simulation structure. The first was that it should be platform independent. There are two reasons for this. The first is that it was expected that the system would be required to run on different architectures. The second, and more important reason was that the we did not want to depend on one particular compiler or operating system. It is well known that code is only portable once it has been ported. By simultaneously developing for several operating systems, it would be almost guaranteed that Ikaros would be reasonably portable. We have consequently strived to comply with the relevant standards as much as possible. These includes ANSI C++, POSIX and BSD sockets. A related choice was to depend on as few external libraries as possible. Although the current version of Ikaros uses external libraries for sockets, timing, threads and mathematical operations, it can still be run in a minimal version that only uses a small set of standard C++ libraries.

The second main design choice was to use a discrete-time model for simulation. Although this is the normal operation for most neural network simulators, there are some notable exception. However, to allow the easy integration of different
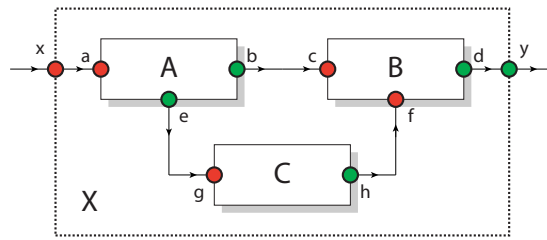
FIG. 3: *A group consisting of three modules. The group is externally considered as a module named X with one input x and one output y. These inputs and outputs are internally connected to input a of module A and output d of module B.*

```
<group name = "X">
   <input name = "x" targetmodule = "A" target = "a" />
   <output name = "y" sourcemodule = "B" target = "d" />
   <module name="A" ... />
   <module name="B"  ... />
   <module name="C"  ... />
   <connection sourcemodule= "A" source = "b" targetmodule = "B" target= "c" />
   <connection sourcemodule= "A" source = "e" targetmodule = "C" target= "g"/>
   <connection sourcemodule= "C" source = "h" targetmodule = "B" target= "f" />
 </group>
```

FIG. 4: *Example of a group of modules with its own input and output. The graphical representation of this system is shown in Fig. 3*

types of algorithms, it was decided that a discrete time simulator would be most useful. It is hard to imagine how many algorithms could be adapted to a continuous time framework. In most cases, this choice does not limit the possible models that can be designed since it only relates to the times when different modules communicate and not their internal structure.

Another consideration was that to make the system attractive it should be as easy as possible to use many different types of programming styles. As a consequence, we decided to only use standard C data structures such as integers and matrices of floats. The use of doubles was decided against on grounds of efficiency and the lack of support for doubles in most vector co-processors.

B. Module Interface

All inputs and output of modules are represented as arrays or matrices of floats and the sizes of these matrices are represented by integers. The sizes of all data structures used by Ikaros are calculated during startup and can not be changed during execution. This restriction only applies for the data moved between modules; for internal data used in modules there are no restrictions at all. The actual code in a module can use any coding style as long as the inputs and outputs are in the right format - indeed, it is entirely feasible to embed or interface with an interpreter in a module for a completely different language transparent to Ikaros itself. Since Ikaros itself is written in C++, either C like or C++ like coding styles can be used as long at it is wrapped in a C++ class. Although the inputs and outputs are part of the Ikaros kernel data structures, the modules themselves does not know about this. Instead, they can magically assume that the input matrices are always filled with the required data.

This design decision has made it easy to incorporate code not specifically written for Ikaros as long as it is reasonably clean. For example, the main function of a trivial module that would only copy its input to its output may look like this:

```
MyModule::Tick()
{
    for(int i=0; i<size; i++)
        output[i] = input[i];
}
```

The point here is that this code looks like any C++ code and there is nothing Ikaros specific with it. When this function is called, the array input will contain the input to the module and after execution, Ikaros takes care of the result in the array output.

It was also considered fundamental that simulations using Ikaros would not be slower than simulations made in a dedicated system. Conceptually, all modules in Ikaros run concurrently and synchronously. This mode of operation was selected because it is the only possibility when it is necessary that execution order is well defined, which is the case for many algorithms. Because of the synchronous operation, there will be a delay of exactly one time step (or tick) between the production of an output from a module and the time when it can be used by another module. In most cases, this extra copying step is necessary anyway and does not usually incur any extra execution cost.

Since this overhead is not always desired however, version 0.8.0 introduced zero-delay connection between modules. Using this type of connections, there is no delay at all between the production of an output and its use by other modules. Instead, the second module refers directly to the

memory where the first module has produced its output. To make the result well defined, zero-delay connections are only allowed within subsets of the complete module networks that form directed acyclic graphs. That this condition is fulfilled is checked during start-up when all modules are sorted according to their position in the graph. With zero-delay connections, the input to the system can in principle be processed in a single time step regardless of the number of modules that the information passes on its way to the output. In this case, the execution overhead is negligible.

The kernel also includes a small set of libraries that hides system specific code for sockets, timing, threads and serial communication. In addition there are utility libraries for memory management, XML processing and mathematical functions. In most cases, the programmers need not know about any of these libraries to use Ikaros.

C. Kernel Start-Up

The kernel is responsible for the creation of the network and its modules at startup, the scheduling during system execution, and the propagation of data between modules. Fig. 5 shows the main component of the running Ikaros system.

Detailed knowledge of the kernel operation is not at all necessary or even recommended for use of Ikaros. Knowing why and in what order things are started do however make it easier to understand the design decisions made. This section can be skimmed lightly without any loss of understanding.

The most important aspect of the kernel is the creation sequence that occurs when the system starts up. This happens in six steps:

a) Class Registration: When the Ikaros program starts, it first registers all code for the modules contained in the system. This initialization step builds a data structure that contains pointers to a creator function for each module type and binds it to a module class name.

b) Module Creation: When the initialization has finished, the kernel reads the supplied control file in XML-format, which specifies the modules to activate and gives them instance names and other parameters. One instance of each module specified is created for every occurrence of that module in the control file. A module can thus have multiple instantiations with different parameters. When each module is created, it registers its inputs and outputs in the kernel to allow them to be connected in the next step. At this stage, the individual modules also gain access to any additional parameters set in the control file for that particular module.

c) Connections: When all modules have been created, the kernel continues to read the control file and make the specified connections between modules.

d) Size Calculations: Most input and outputs have dynamical sizes that are set during start-up. For example, if the input of a module is connected to the output of another module that produces a 4x4 matrix, the input of the second module will adapt to this and set the size of its outputs accordingly. There can be any relation between the size of an input and the size of an output.
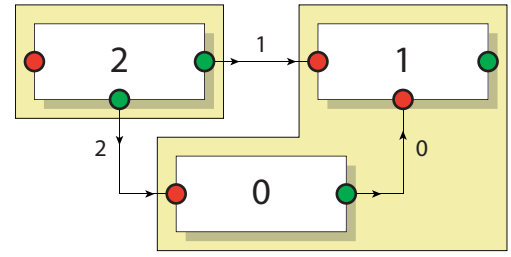

FIG. 6: *The order of execution of three modules. The numbers on the connections indicate the delay in the connections. The numbers on the modules indicate the order in which they should be executed. The two shaded areas correspond to two thread groups.*

For example, the output from the module could be set to have the double size of the input or some other more complex relation. Since there can be a number of cyclical relations between different modules, the calculation of output sizes is performed iteratively until all sizes have been established. If there are cyclical dependencies, these will be found during this stage and an error message will be produced.

e) Sorting the Modules: All modules are sorted in two ways (Fig. 6). The modules are partitioned into different sets that each contains a directed acyclic graphs (DAG) of modules with zero-delay connections between them and only delayed connections to any other modules. Each of these sets can be run in a separate thread and is called a thread group. A topological sort is performed on the groups according to their positions in the DAG which defines a partial order relation on the modules. For modules that have zero-delay connections between them, this order is used to make sure that a module that produces data that another module will use is always executed before that other module.

f) Module Initialization: When all modules have been connected, the initialization phase starts. At this stage, the size of the input that each module will receive is known and each module is allowed to create any additional storage that it needs and initialize variables. To do this, the kernel calls an initialization function for each of the created modules.

D. Kernel Operation

The scheduling mechanism of the Ikaros kernel is responsible for calling the code of each module instance once during each discrete time step (or tick).

In the simplest case, the scheduling consists of calling the tick function for each module in the order in which they were sorted during initialization. When Ikaros runs in threaded mode, each thread group is handled separately in this way. In threaded mode, there is no communication between modules in different DAGs during this time which greatly simplifies the operation of the kernel.

In a second step, the data propagation function is called to copy data from outputs to the inputs of the modules. Data propagation is done simultaneously for all modules. The output for each module is copied to the input to which it is connected. The propagation process is also responsible for the simple data translation that is made by the system and
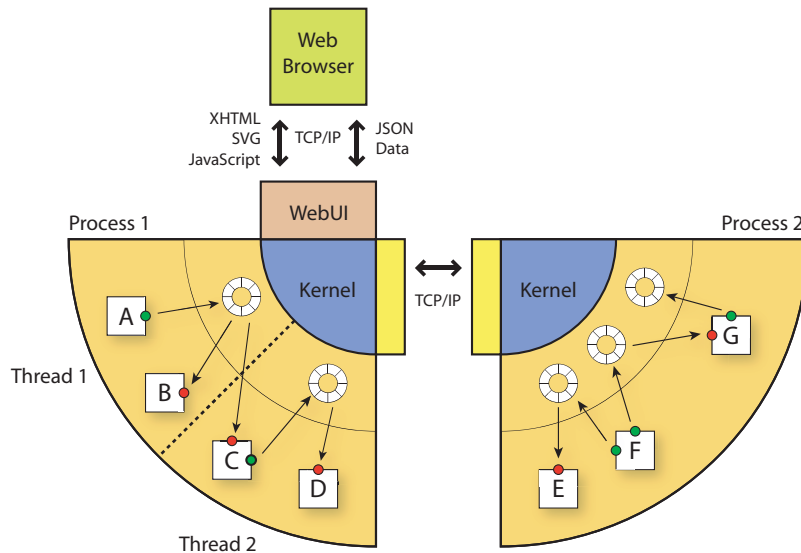
FIG. 5: *The Ikaros kernel. The kernel starts a number of threads where a number of modules (A-G) are executed. The modules communicates through a set of circular buffers that correspond to outputs from the modules. The kernel can also communicate with other Ikaros processes running on the same or on a different processor or computer. In addition, the kernel communicates with an optional graphical user interface client running in a web browser.*

concatenation in the case when several outputs are connected to the same input. In addition, this stage delays the data on connections when this is set in the connection.

Finally, the kernel handles timing when Ikaros runs in real-time mode. In this case, the kernel makes sure that the execution of the tick did not take longer than allowed and waits for the appropriate moment to start the next tick.

E. Anatomy of a Module

Every module in Ikaros implements five functions. For a module named MyModule, the following functions are be defined and called in the following order:

MyModule() The creator function registers all the inputs and outputs of a module. It also gains access to all parameters of this instance of the module from the control file.

SetSizes() This optional function is called repeatedly during start-up to calculate the sizes of dynamic outputs based on the sizes if the inputs to the module.

Init() The init function is called after kernel initialization and lets the module gain access to its inputs and outputs. This is also were any internal data structures are allocated.

Tick() The tick function is where the actual work is being done by the module. It is called repeatedly during the execution of a module and should calculate new outputs based on its inputs (See example in section 3.1).

~MyModule() This optional function deletes any module specific memory that has been allocated in Init() and performs other clean-up that may be necessary.

A template for new modules is available as part of Ikaros. This template is named MyModule and a new module can easily be added to Ikaros by simply renaming the template.

V. STANDARD MODULES

Ikaros contains a large number of standard modules. These can be divided into a number of categories.

IO Modules: There is a set of modules that read data from different file formats, for example text data or different media files. Other modules are used to communicate with external devices such as cameras or robots.

Utility Modules: To simplify the design of models, there are also a large number of utility modules for simple mathematical operations. This includes vector and matrix operations and standard mathematical functions. Other utility modules are used to collect data or statistics or to control an experiment. A few utility modules are used to generate input such as the function generator.

Image Processing Modules: Another set of modules implement standard image processing functions. There are modules to transform the colors in an image, modules that scale images in different ways or performs other spatial transforms. To apply different image processing operators there is a module for convolution, but also modules for specific operators such as the Sobel operator and parametrically defined Gabor filters. There are also several modules that performs edge detection. A few vision modules are more complex and implements a saliency map or an attention focusing mechanism.

Environment Modules: To allow simulation of an agent in an environment, there are a number of modules that implements simple environments. The GridWorld module implements a two-dimensional environment consisting of a grid with obstacles together with an agent that can navigate in it while being controlled by other Ikaros modules. There is also a variant where the simulated robot can move continuously over the grid. This module also simulates a 2D visual field using a ray casting algorithm. Another module simulates an arm with arbitrary geometry.

Other Modules: The standard modules also include a few neural network algorithms and some general learning

algorithms.

VI. REAL-TIME EXECUTION

When Ikaros is used to control robots it is necessary that the precise timing of input and output can be controlled. To accomplish this the kernel has functions to time the execution of each tick. When Ikaros starts up it sets it time-base to the required interval and tries to time the ticks to this time-base. It internally controls that it is able to keep up with the desired speed and will report delays in the execution.

Obviously, the accuracy of the timing will depend on the underlying operating system. The real-time functionality is based on POSIX.4 [21], but since Ikaros is currently not running on real-time operating systems, any other process can in principle interfere with real-time execution. In practice, it is possible to get less than 1 ms resolution on the operating systems we have tested.

An important factor that contributes to real-time performance is the ability to run Ikaros in multi-threaded mode [10]. In this mode, the kernel tries to run every module in a separate thread. When there are zero-delay connections between a set of modules, the kernel will automatically put these in the same thread.

In thread mode, each module can be set to run at different time intervals. For example, a slow visual processing module may run 5 times per second while a faster motor control module can be allowed to run 100 times per second. This feature is very useful for robotic control where some loops need to run at high speed while others are much heavier.

VII. A GRAPHICAL USER INTERFACE

To monitor ongoing simulations, Ikaros has a graphical user interface. Like the modules and connections, this user interface is specified using XML. This XML specification is read by the Ikaros kernel which starts up an integrated web-server which allows standard web browsers to act as graphical clients. The browser gets a set of JavaScript routines from Ikaros that are run in the browser that implements the graphical user interface [9]. The actual drawing is made using SVG [8]. The choice of JavaScript+SVG was based on the fact that this would make the system truly platform independent.

For communication with the sever, the interface uses JavaScript Object Notation (JSON). Although we initially planned to use XML for this communication, JSON turned out to be much simpler to use since it can be natively parsed by JavaScript using the eval function.

Unfortunately, few browsers initially supported SVG and we made the choice to only actively support FireFox. The first version of Ikaros that used this graphical user interface was released a few days before the first version of FireFox to include native SVG rendering (version 1.5). Today, several other browsers support SVG and JavaScript in the required way including Safari and Opera.

Currently, Ikaros has support for graphical objects such as bar graphs, different forms of 2D and 3D plots, images, grids and vector fields. The graphical client can easily be extended with new graphical objects by writing JavaScript code for the drawing of the new object.

One limitation of this solution is that it is not as fast as using a dedicated program for the client. However, we felt that this solution has several advantages. First of all, it means the whole system becomes totally platform independent. But also, and perhaps more importantly, it enables us to transparently monitor and control a running simulation remotely, independent of what system the simulator and the client is running, and we can do so with a simulation running in another room or across two continents with no loss of functionality.

If fast, concurrent representation is important, the very open-ended structure of an Ikaros module enables users to simply write a graphical module that includes the toolkit or other representational system of their choice and display data sent to the module from there. Likewise, a module that receives user interaction can change the behavior of other modules in the system accordingly by defining a "command channel" that sends data to other modules via the same mechanism as ordinary data. Ikaros does not care how data is interpreted within modules after all.

VIII. VALIDATING MODELS

To automatically validate a model against relevant data, for example, neurobiological databases, the specification of a module can include the *models* attribute. For example, a module that claims to model the amygdala could be describes in the following way:

```
<module
    class = "MyClass"
    name = "MyModule"
    models = "Amygdala"
/>
```

This information could be used to match the graph made up of the modules in an Ikaros model to connectivity data found in neurobiological databases. Some first attempts towards such as system have been taken [11]. More recently, we also interfaced the Ikaros validation system with the CoCoMac database.

IX. EXPERIMENT DATABASE

In our earlier studies of classical conditioning we have developed an extensive database of the design and results of conditioning experiments. The development of this database started in 1996 and now contains approximately 200 different experiments. The database is stored in a way that allows the experimental descriptions to be used as input to computer simulations of learning by classical conditioning.

Unfortunately, this database was stored in a form that is not easy to access unless the previous simulator developed at LUCS is used. It also has the limitation that it only covers classical conditioning and not other learning paradigms. As a part of the Ikaros project, we want to extend the experiment database by adding more experiment types and by translating the database to a more accessible format.

In the future, we will add experiment description for other learning paradigms besides classical conditioning. This includes operant conditioning experiment as well as more cognitively oriented experiments. The goal is to cover all experiment types that are regularly used with animals and humans. We estimate that the final database will include approximately 1000 experiments.

The entry for each experiment will include all information that is necessary to reproduce the experimental conditions in a simulator or a real experiment. This includes detailed data of the stimuli used, the apparatus, the exact timing etc. It will be important to differentiate between the part of the experiment description that contains the logic of the experiment and features such as timing and spatial location that are often not essential. This will allow modelers to adapt experiments to their needs in much the same way that an experiment developed for one species has to be changed to fit another. The database will also contain experiment descriptions in narrative form and pointers to external databases such as Medline and BIOSIS when appropriate.

To allow easy access to the experiment database, it will be coded in the XML format that is widely used for on-line data. The choice of XML for the database is natural since it allows for an evolving and continually expanding database structure. It can also be used to mediate the transfer of information from other already existing databases. Apart from translating the already existing database to this format, we will also develop tools that can be used to encode and visualize experiments through a web-based interface.

X. DISCUSSION

During the last few years, Ikaros has been used to build a number of cognitive models and to control many different robots. This has to date resulted in over 40 scientific publications. For example, for cognitive modeling, it has been used in several models of cognitive development and the modeling developmental disorders [3], [7], plasticity in the somatosensory cortex [14] and to study different forms of learning [5] and emotion [20], [4]. A lot of the work on Ikaros has involved visual processing, for example models of visual contour processing [19] and models of visual attention [1], [2].

We have used Ikaros to control a number of different robotic hands built at Lund University Cognitive Science to investigate haptic perception [15], [16], [17], [18]. The hands have different sensors and different degrees of freedom and are all controlled by different neural network based architectures. In another line of research, we have looked at anticipation and navigation in mobile robots including the e-puck and the BoeBot [12], [13]. Here, Ikaros is used to implement very different models that are more classical in the sense that they use potential fields or planning approaches.

The approach in Ikaros to be minimally demanding regarding the types of architectures that can be built and the types of programming styles that can be used has proved to be very successful. It is also clear that many of the design choices made initially were sound and has contributed to the usefulness of the system. Unlike most other frameworks, Ikaros do not force the user into one theoretical model or into using extensive libraries even though such support is offered. This has made it easy for users of diverse backgrounds to quickly learn to use the system.

On the other hand, there are certain restrictions that limits for what systems Ikaros is useful. Some of these constraints certainly makes Ikaros less useful for some systems, in particular architectures that mainly relies on symbolic processing rather than numerical computation. We believe that for a tool to be useful, it is necessary that it is adapted for specific tasks and this inevitably makes it less useful for other tasks. For Ikaros, it was important that it could be used for real-time processing and for robot control, which makes it different from many other framework for more biologically based modeling. We also wanted Ikaros to run on almost any hardware which is the reason behind many of the design choices.

In summary, Ikaros has proven to be a very useful tool for building cognitive systems models and for robot control. It has evolved into a mature and stable system and has currently been adopted by several research groups within the cognitive sciences.

XI. ACKNOWLEDGEMENTS

REFERENCES

[1] C. Balkenius. Cognitive processes in contextual cueing. In F. Schmalhofer, R. M. Young, and G. Katz, editors, *Proceedings of the European Cognitive Science Conference 2003*, pages 43–47. Lawrence Erlbaum Associates, Mahwah, NJ, 2003.

[2] C. Balkenius, K. Åström, and A. P. Eriksson. Learning in visual attention. In *ICPR '04 workshop on learning for adaptable visual systems (LAVS)*. 2004.

[3] C. Balkenius and P. Björne. Toward a robot model of attention-deficit hyperactivity disorder (adhd). In C. Balkenius, J. Zlatev, H. Kozima, K. Dautenhahn, and C. Breazeal, editors, *Proceedings of the First International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*, volume 85 of *Lund University Cognitive Studies*. 2001.

[4] C. Balkenius and J. Morén. Emotional learning: A computational model of the amygdala. *Cybernetics and Systems*, 32(6):611–636, 2000.

[5] C. Balkenius and S. Winberg. Cognitive modeling with context sensitive reinforcement learning. In *Proceedings of AILS '04*. Dept. of Computer Science, Lund, 2004.

[6] Christian Balkenius, Birger Johansson, and Jan Moren. *Ikaros Control File Specification*. http://www.ikaros-project.org/2007/IKC10-20070601/, 2007.

[7] P. Björne and C. Balkenius. A model of attentional impairments in autism: First steps toward a computational theory. *Cognitive Systems Research*, 6(3):193–204, 2005.

[8] J. David Eisenberg. *SVG Essentials*. O'Reilly, 2002.

[9] David Flanagan. *JavaScript: the definitive guide*. O'Reilly, fourth edition, 2002.

[10] Bill O. Gallmeister. *POSIX.4—programming for the real world*. O'Reilly, 1995.

[11] M. Gustafsson and C. Balkenius. Using semantic web techniques for validation of cognitive models against neuroscientific data. In *Proceedings of AILS '04*. Dept. of Computer Science, Lund, 2004.

[12] B. Johansson. Elastic template matching in outdoor environments. Master's thesis, Lund Univeristy Cognitive Science, Lund, 2004.

[13] B. Johansson and C. Balkenius. An experimental study of anticipation in simple robot navigation. In M. et al Butz, editor, *Anticipatory Behavior in Adaptive Learning Systems: From Brains to Individual and Social Behavior*. Springer, 2007.

[14] M. Johnsson. Cortical plasticity: A model of somatosensory cortex. Master's thesis, Lund Univeristy Cognitive Science, 2004.

[15] M. Johnsson and C. Balkenius. Experiments with artificial haptic perception in a robotic hand. *Journal of Intelligent and Fuzzy Systems*, 17(4):377–385, 2006.

[16] M. Johnsson and C. Balkenius. LUCS haptic hand II. Technical Report 9, LUCS Minor, 2006.

[17] M. Johnsson and C. Balkenius. Neural network models of haptic shape perception. *Robotics and Autonomous System*, 22:720–727, 2007.

[18] M. Johnsson and C. Balkenius. Associating som representations of haptic submodalities. In *Proceedings of TAROS 2008*. Edinburgh, UK, 2008.

[19] Stefan Karlsson. Monocular depth from occluding edges. Master's thesis, Department of Mathematics, Lund Institute of Technology, 2004.

[20] J. Morén. *Emotion and Learning - A Computational Model of the Amygdala*. Lund University Cognitive Studies, 2002.

[21] Bradford Nichols, Bick Buttlar, and Jackie Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.