NASA/ESA Conference on Adaptive Hardware and Systems

UbiManager: A Software Tool for Managing Ubichips

Yann Thoma, Andres Upegui
REDS - HEIG-VD
Yverdon-les-Bains, Switzerland
Email: yann.thoma@heig-vd.ch, andres.upegui@heig-vd.ch

***Abstract*— This paper introduces the *UbiManager*, a tool for managing the ubichip reconfigurable circuit. The ubichip is a custom reconfigurable electronic device for implementing circuits featuring bio-inspired mechanisms like growth, learning, and evolution. The ubichip has been developed in the framework of Perplexus, a European project that aims to develop a scalable hardware platform made of bio-inspired custom reconfigurable devices for simulating large-scale complex systems. In this paper, we present the software tool used for designing, simulating, emulating, debugging, configuring, and monitoring the systems to be implemented in the ubichip. This paper also presents the dissemination plans of the *UbiManager*, that consist in a web platform allowing researchers to access the hardware platform from any remote base station.**

I. Introduction

The Perplexus project [1] aims to develop a scalable hardware platform made of custom reconfigurable devices endowed with bio-inspired capabilities. This platform will enable the simulation of large-scale complex systems and the study of emergent complex behaviors in a virtually unbounded wireless network of computing modules.

The Perplexus platform will consist thus in a scalable network of ubidules (UBIquitous computing moDULES) equipped with wireless communication capabilities and rich sensory elements [2]. The platform is modular for allowing the application developer to customize his platform set-up. In this way the application developer can easily build his system setup by selecting what to plug to the ubidule from a set of peripherals. These peripherals can be different communication interfaces (wifi, bluetooth), sensors, actuators, cameras, or flash memories. This modularity is guaranteed by the use of standard interfaces such as USB.

At the heart of these ubidules, we use a ubichip (Ubidule Bio-Inspired CHIP)[3], a custom reconfigurable electronic device capable of implementing bio-inspired mechanisms such as growth, learning, and evolution. These bio-inspired mechanisms will be possible thanks to reconfigurability mechanisms like dynamic routing, distributed self-reconfiguration, and a simplified connectivity. Such an infrastructure will provide several advantages compared to classical software simulations: speed-up, an inherent real-time interaction with the environment, self-organization capabilities, simulation in the presence of uncertainty, and distributed multi-scale simulations.

One of the most critical problems faced by custom reconfigurable devices is the absence of design tools. These devices are typically developed under projects running for a few years and the development of tools are rarely among the priorities of the project, making the device very difficult to use. In the Perplexus project, we began both - the design tool and the ubichip architecture - almost in parallel from the beginning of the project, what have allowed us to develop a synergy among both.

In this paper we present the *UbiManager*, the tool currently used for design, simulation, emulation, debugging, configuration, and monitoring of circuits on the ubichip. For this, in sections II and III we introduce the ubidule and the ubichip respectively. Section IV describes the *UbiManager* tool, its capabilities, and the design interface. Then, section V describes the utilization of the *UbiManager* for simulation, debugging, and monitoring purposes. Afterward, section VII describes the web interface that allows to share the use of these ubichips. Finally, section VIII concludes.

II. Ubidule

The ubidule is an electronic board that mainly contains an ARM processor and a ubichip. The ARM processor acts like the ubichip manager, allowing the ubichip configuration and a close interaction with it. Currently, we are using an Xscale PXA270 running Linux, and it is accessible through Ethernet, Wi-fi, or Bluetooth.

A ubidule should be able to closely interact with its environment. In order to keep the platform general enough, no sensor or actuator has been directly added to the board, but instead, five USB ports and a microSD card slot allow us to connect such devices. These ports provide the possibility to plug different communication interfaces like Wi-fi, Bluetooth, or Zigbee. They allow also to dispose of a large choice of USB sensors and actuators developed by different manufacturers, ranging from simple analogue temperature or light sensors to more sophisticated devices as accelerometers, RFID reader, mice, and servomotors.

The ubidule offers also the possibility of driving an LCD-touchscreen that can be directly plugged onto the board. As it will be explained later, this LCD-touchscreen is managed by a graphical application running on the ARM processor, and it can be used for configuring the ubichip, as well as to observe its state.

At this time, we have a first ubidule prototype (figure 1) [2], and includes a Spartan-3 5000 for prototyping the ubichip.

978-0-7695-3166-3/08 $25.00 © 2008 IEEE
DOI 10.1109/AHS.2008.39

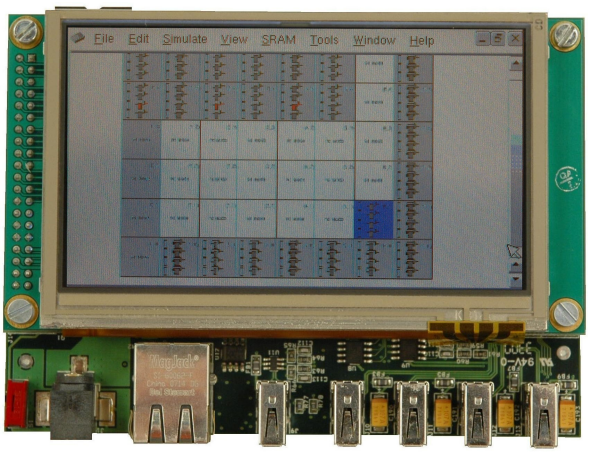

Fig. 1. Picture of a ubidule board

The Xilinx FPGA [4] allows for the emulation of a simplified version of the ubichip, as well as for testing the entire system.

III. UBICHIP

The heart of the ubidule contains a ubichip. This reconfigurable circuit is a new kind of FPGA that implements special features like self-replication, dynamic routing, and a SIMD architecture. In this section, we will briefly describe the ubichip, but the interested reader can find more details in [3].

A ubichip is mainly composed of three reconfigurable layers. The first one is an array of ubicells, the reconfigurable logic elements used for computation purposes. A ubicell is composed of four 4-input look-up tables (LUT) and four flip-flops (DFFs). These ubicells can be configured in different modes like counter, FSM, shift-register, 64-bit LFSR, adder, subtractor, etc. A ubicell can also be configured as a simple 4-bit processor, allowing the merging of n ubicells for creating $4n$-bit processors. By using this configuration mode, a processor array can be used as a SIMD array of processors, with a hardwired on-chip sequencer being responsible for the multi-processor management.

The second layer contains dynamic routing units that permit the ubicells to dynamically connect to any part of the circuit. Based on identifiers and a concept of sources and targets trying to reach a correspondent with the same ID, it looks quite similar to the system described in [5], while having enhancements on different points (cf. [3]). This layer allow thus to create and destroy connections in a dynamic way, allowing the implemented system to change its topology on run-time.

Finally, the third layer is made of self-reconfiguration units that allow a part of the circuit to self-replicate somewhere else on the chip, without any external intervention. This mechanism allows also to destroy a section of the circuit. This truly new feature can be very useful for cellular systems such as neural networks with changing topologies. A neuron can, for instance, decide to duplicate when it has a high level of activity, or it can destroy itself after a long period of inactivity in order to leave resourced for other neurons. More details about this mechanism can be found in [6].

IV. UBIMANAGER

A new circuit with fancy reconfigurable features is nice, but without design tools that allow the creation, simulation, and debugging of the implemented circuits, it is nothing but a toy for engineers, requiring a huge effort for implementing the simplest circuit. Design tools constitute thus a critical issue when using any type of programmable device, being it a processor or an FPGA.

The typical design flow for an FPGA-based design consists on an initial translation from an HDL description into a netlist, and then from a netlist into a description of the circuit configuration bits. However, the structure of the ubichip is quite different from those of standard FPGAs that are composed of LUTs and DFFs connected to switch matrices. The 4-LUT cell of the ubichip, as well as its non-regular routing organization did not allow us to exploit existing HDL synthesizers. Therefore, we conceived a graphical tool in order to allow the development of Perplexus applications.

UbiManager (cf. figure 2) is the main tool for using the Perplexus platform. It allows to graphically configure each one of the three reconfigurable layers of a ubichip, and then displays the system state during a simulation or the execution on a ubidule at run-time. It can run on a base-station or on an embedded platform as the ubidule.

The software can open multiple documents at the same time, and copy-paste commands allow to easily reuse design parts. Moreover, drag-and-drop is also implemented. Both copy-paste and drag-and-drop permit to include an image of the current selected units in any software that deals with images, such as OpenOffice or Powerpoint. An auto-updater has also been integrated to let the user always run the latest version.

A. Views of the configurable layers

Figure 2 shows the ubicell layer. One can select a ubicell by clicking on it, in order to configure it using a set of

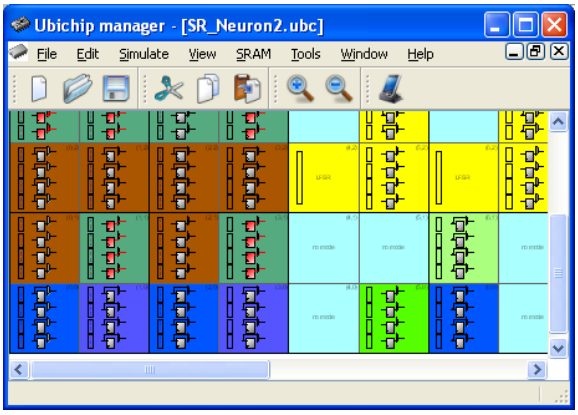

Fig. 2. UbiManager graphical interface showing ubicells

configuration options through a user-friendly menu. Then, one can visualize some of the features of the configured circuit. For instance, in the figure, one can differentiate the ubicells configured as independent 4-LUTs in combinatorial mode, and in registered mode, as well as those configured as 64-bit LFSR. The same view can also be used to visualize the system state, as will be described in the next section.

Figure 3 illustrates the dynamic routing layer. The dynamic routing units are configurable as sources or targets, and then the underlying system implemented on the ubicell layer can trigger a path creation or destroy existing paths. The configuration and visualization of these dynamic routing units are possible thanks to this second view.

Finally, figure 4 shows the self-replication layer configuration. As described in [6], a self-replicating system requires a morphological description describing a construction path. This path allows the further construction of a daughter cell, just by following the morphological description defined by a set of building flags. The screenshot shown in figure 4 depicts the construction path that allows a system to be replicated in the platform. This view allows thus to edit such construction path and to visualize the state of a system being self-replicated.

B. UbiAssembler

As presented in section III, a ubichip reconfigurable array can implement a SIMD architecture, where a ubicell acts like a 4-bit processor. In that case, a $4n$-bit processor can be obtained by merging n ubicells. The SIMD array is controlled by a centralized sequencer that executes a program previously loaded in the ubidule SRAM. A special assembler has been developed for this sequencer, as well as a tool for translating it to the binary format. Assembler files can be open within UbiManager, in a text editor with specific keywords highlighted.

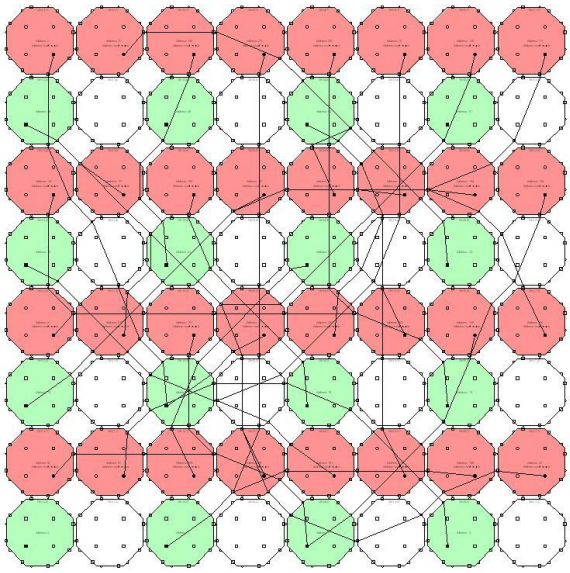

Fig. 3. UbiManager graphical interface showing routing units

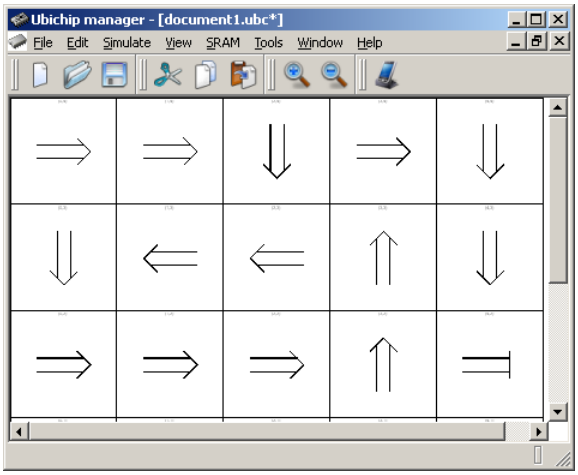

Fig. 4. UbiManager graphical interface showing self-replicating units

For running a simulation, or an experiment on a ubidule, if the project requires the SIMD mode, then the assembler file associated with the project is compiled, and sent to the SRAM. In order to ease the debugging of an application, the internal state of the sequencer can be observed during a run. The state of every register is displayed in a windows like the one shown in figure 5.

In conclusion, the *UbiManager* offers a set of interactive menus that allows to select the different types of logic units, to edit their configuration by selecting among the provided configuration modes, and to visualize some of the configured features in each unit as well as the connections created on the static and dynamic routing. As previously explained,

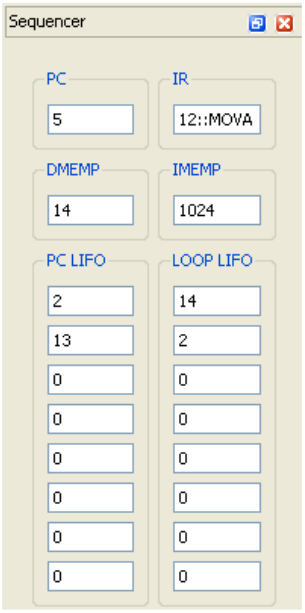

Fig. 5. UbiManager graphical interface showing the sequencer state

there is, at this time, no automatic tools that can generate a configuration file based on an HDL description, because of the complexity of the ubicell. However, this complexity is due to the coarse grained architecture, which let standard functions such as 4-bit counters be very easy to implement, even only with a graphical interface.

V. Simulation/Emulation

After the design phase, the *UbiManager* allows to simulate a system in order to test its functionality. As the ubichip has been described in VHDL, the simulation is performed with Modelsim. A standard Modelsim simulation allows to force signals and to observe results in a waveform window or to use a test bench that must be previously written. While very useful for traditional designs, this interface is too restrictive for our application. A third approach for managing Modelsim simulations is from an external software description. For this, Modelsim supplies a C library that permits to interface a VHDL simulation with some C/C++ code, through the so called Foreign Language Interface (FLI). In this way, it is possible to let some C functions be called when certain user-defined signals change their values. The C/C++ code can then interact with the simulation by setting signals values and by retrieving information about signals in the design.

For this purpose, we wrote a Dynamically Linked Library (DLL) loaded at the starting point of a simulation, which supplies a function called on every falling edge of the clock. This DLL, called *UbiFli*, instantiates a TCP/IP server that waits for connections. The *UbiManager* that started the simulation can connect to this server, and then interact with the simulation. The main advantage of using TCP/IP communication relies in the fact that the same protocol is used to access both, the simulated circuit on Modelsim and a real ubichip on a ubidule.

From the point of view of a *UbiManager*, a simulation or a ubidule supply the same functionality. Moreover, several *UbiManagers* can be simultaneously connected to the same simulation/ubidule, and interact with it. It is therefore possible to have several users observing the same behavior from anywhere at the same time.

When connections are established, *UbiFli* waits for commands. These commands are:

- *Run 1 clock step*
- *Run n clock steps*
- *Run n clock steps with state recovery at each clock cycle*
- *Run as long as no break command is sent*
- *Break a run*
- *Reset the system*
- *Reconfigure the system*

At the end of a command execution, *UbiFli* sends the state of the system back to all the connected *UbiManagers*. This state contains the state of every ubicell, routing unit, and self-reconfiguration unit. When a *UbiManager* gets the state of the system, it updates its graphical display accordingly, letting the user observe the simulation result.

Figure 6 illustrates a simulation scenario. The following steps detail the starting tasks of a simulation.

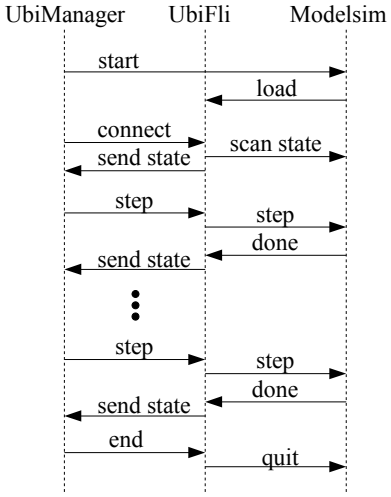

Fig. 6. Simulation scenario

1) (UbiManager) Starts simulation
2) (UbiManager) Generate configuration files
3) (UbiManager) Starts Modelsim
4) (Modelsim) Loads *UbiFli*
5) (UbiFli) starts the TCP/IP server
6) (UbiManager) Connects to the *UbiFli* server
7) (UbiManager) Do step by step (for instance)
8) (UbiManager) Sends command to *UbiFli*
9) (UbiFli) Sends back the system state

A simulation can be observed/managed by more than one UbiManager, and so acts a ubidule. The ARM runs a TCP/IP server, called *UbiServer*, that is responsible to access the ubichip. This server responds to the same commands understood by *UbiFli*. Figure 7 illustrates different communication schemes between a simulation, a ubidule, and four *UbiManagers*. The arrows represent a TCP/IP connection, and the *UbiManagers* are run on 3 different PCs.

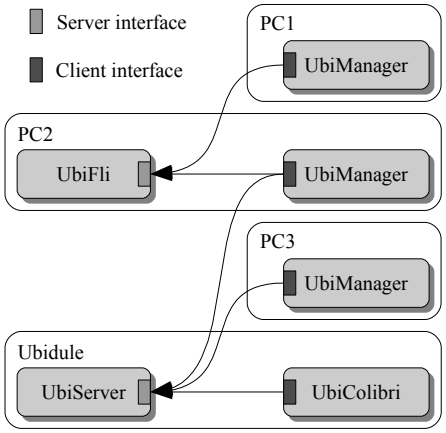

Fig. 7. Communication scheme between different components

A. Plugins

At first, the *UbiManager* was written for the design and simulation of a system implemented for a ubichip reconfigurable array. It rapidly turned out that some applications needed to interact with the circuit, without human intervention. For instance, an intrinsic evolutive hardware system [7] requires a genetic algorithm [8] to run, load configurations into the chip, and retrieve information about the system execution. Therefore a system of plugins has been designed, letting the user write a C++ plugin in order to interact with the simulation. A plugin contains a function called every time the state of the system changes. This function can get the state of the system, and can modify it on the fly.

For instance, a plugin can be responsible for monitoring the routing array. By analyzing the state of the array every clock cycle, it can generate logs or store intermediate configurations in a file in order to allow, for instance, a post-simulation analysis of the routing process.

A plugin can manage the simulation and act on the configuration bits. For this purpose , a configuration access library has been written, and provides a similar functionality as Jbits [9]. Jbits is a Java API that allows to access the configuration bitstream of Xilinx FPGAs. In a similar manner as Jbits, our configuration library offers the possibility of dynamically and partially accessing the configuration bits of the device at a higher abstraction level. In this way one can easily modify the content of a set of registers containing a parameter, a seed for a pseudo-random number generator, or any building block.

The installation of the *UbiManager* comes with some sample plugins that can be easily modified by developers to act accordingly to their needs.

An important feature of plugins is that they can be loaded by different parts of the application:

- *UbiManager*: If loaded by the *UbiManager*, the plugin receives the state of the system only when the *UbiManager* retrieves this state from the simulation. It means that the user has to exploit the *"run n clock cycles with updates"* in order to let the plugin interact at each clock cycle.
- *UbiFli*: As the *"run n clock cycles with updates"* forces a lot of TCP/IP communication, and so a loss in performance, the plugin can be directly loaded by *UbiFli*. In that case, a command like *"run n clock cycle"* will call the plugin on every clock cycle, without the need of using the TCP/IP connection. With this option, a real gain in performance is observed. However, no graphical interface can be used in this mode, and so a plugin that requires some user interaction should not be loaded by *UbiFli*.
- *UbiServer*: The plugins are written using the Qt environment, and can therefore be compiled under Linux. *UbiServer* can then load the plugin, avoiding the sending of the system state through Wireless. If the application allows for the use of this mode, it can offer a huge gain in term of performance, since the wireless communication represents a real bottleneck in the system.
- *UbiColibri*: *UbiManager* can run directly on the Ubidule. It can then load plugins if required, as soon as the plugins are compiled under Linux.

Figure 8 shows the different uses of a plugin after being compiled for the correct target system.

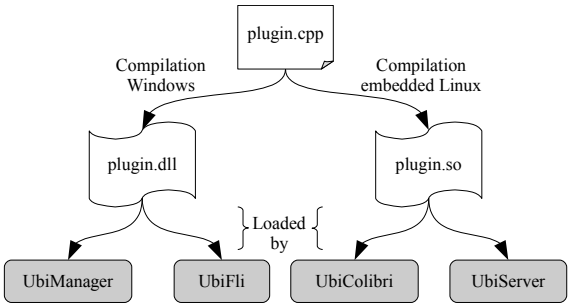

Fig. 8. Plugins usage

B. External application

Plugins, except if loaded by the simulation or by the *UbiServer*, require the *UbiManager* to be run by the user. For some applications, it would be more convenient to have a stand-alone program taking care of the interaction with the simulated/emulated system.

As the *UbiManager* can connect to a Modelsim simulation or a *UbiServer*, an interface has been designed in order to keep a single type of communication between the components. This interface, in the form of a C++ class, can very easily be integrated in any new software. An application such as one using a genetic algorithm could then be implemented with an executable instead of a plugin, if needed.

It is also interesting to note that, as every execution passes through a server (*UbiFli* or *UbiServer*), and a server supporting several connections, it is possible to launch an execution with *UbiManager*, start an application that will connect to this execution and then interact with it. The *UbiManager* will receive the state updates, and let a user visually observe the system state.

VI. Application Examples

Up to now, the *UbiManager* has been used as design tool on different applications that have allowed to exploit the special reconfigurability features of the ubichip. The dynamic routing has been used in the implementation of two types of systems featuring changing topologies: ontogenetic neural networks and evolutionary graph models with dynamic topology. The self-replication mechanism has been used in a particle swarm optimizer with adaptable size. These three systems where successfully implemented thanks to the *UbiManager* design and simulation interface.

The first application example is presented in [10] and [11]. These papers present implementations of neural circuits able to grow by physically creating and destroying synaptic connections depending on neural activity. The described models include a synaptogenetic mechanism that allows to create

connections, and a synaptic elimination mechanism that allows to prune the network. Both, synaptic creation and pruning are performed following an activity-driven approach: the more active neurons have a higher probability of being highly connected, and less active synapses may be destroyed. The created network is compared with a randomly created network as the one depicted in figure 3. These networks exploit the dynamic routing mechanism in order to implement the dynamically created and destroyed synapses.

A second application, also exploiting the dynamic routing, is the implementation of evolutionary graph models with dynamic topologies [12]. The models consider two graphs: an interaction graph and an imitation graph. After interaction, agents in our model revise their strategies by an imitation rule taking into account the distribution of payoffs and the proportion of behaviors in their imitation neighborhoods. The model of [12] explores a particular instance of this general model in which the interaction graph is a static cycle and the imitation graph is a directed, dynamic small-world network constructed over this cycle. With this setup, they try to model the fact that individuals often compare and imitate others with whom they do not interact, and that this imitation social network is in general more dynamic than the interaction graph.

Finally, the third application is a particle swarm optimizer with adaptable size [13] that exploits the self-replication capabilities of the ubichip. This implementation considers a PSO algorithm where the number of particles changes during the swarm life-time. A particle can decide to self-replicate or to self-destroy depending on the performance exhibited by the swarm as depicted in figure 9. If the swarm performance is not good enough the swarm should grow in size and if it is good enough some particles may be removed in order to leave reconfigurable resources for other applications. This system results promising for optimizing functions that change over time and where the size of the swarm may increase and decrease in order to adapt the swarm to the computational needs of a changing environment.

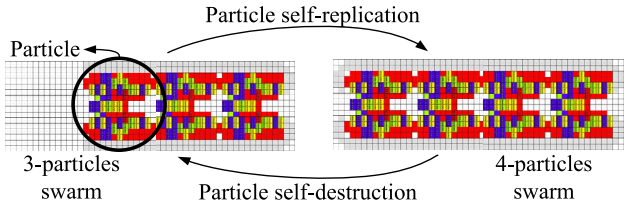

Fig. 9. Particle swarm optimizer with adaptable size

These three applications have used the self and dynamic configuration capabilities of the ubichip, but they have also used plugins for automatically modifying algorithm parameters. For instance, the initialization of the agents' state and the payoff matrix in the evolutionary graph application is performed by a plugin that partially reconfigures the device, and then recovers the agents' state after each agent's update. These applications are the firsts to be implemented on the ubichip substrate, and their implementation would be just unfeasible without the *UbiManager*.

VII. Worldwide usage

One of the goals of the Perplexus project is to make the platform available to anyone who would like to test/use its new features. The plan is to build around 100 ubidules before the end of the project. In order to fully exploit these ubidules, a web platform is intended to let a user have a look at the state of the ubidules population.

PHP scripts running on the server side are accessed both by the ubidules and by the user's web browsers [1]. An active ubidule runs a *UbiServer*, and accesses one of the scripts every 10 minutes to indicate it is ready for an application run. While running an application, the ubidule connects also to the main web server every 10 minutes, letting the server know whether the ubidule is running. Both requests from *UbiServer* allows the PHP page to update a status file in which information about the ubidules state is stored. At the end of an application run, the ubidule informs the server that it is ready and active, waiting for a new application. Each ubidule is identified by its IP address, and is marked as active, running, or disconnected. In order to fully detect disconnected ubidules, the date of the last access from *UbiServer* is also stored.

When a user asks for the list of available ubidules, the main web page only has to check the status file to detect which one is alive, under exploitation, or disconnected. If one is available, then the user can request its use and directly connect the *UbiManager* to the ubidule, by copying its IP address. The *UbiManager* also provides the capability of asking the ubidules list to the web server. In that case, there is no need of copying an IP address from the web page, everything being controlled by the *UbiManager*. In this way, a developer that has designed a system and tested it through simulation, can rapidly test it on the real hardware, without the need of having a board on his desk. Figure 10 shows a standard usage scenario, featuring a ubidule, an instance of *UbiManager*, and a web server.

The basic functionality of the distributed platform, currently, does not manage a potential abusive user keeping control on one or several ubidules. The first implementation assumes fair users, but a second version will include strategies to detect such behaviors and block them.

VIII. Conclusions

The Perplexus ubidule is a reconfigurable hardware platform for the simulation of complex systems. As such system requires design and verification tools to deal with (from a developer point of view), we have created a set of tools to ease its use.

The *UbiManager* allows a developer to create a system using a graphical user interface, and to simulate it. During a simulation, every flip-flop of the system can be graphically observed. C++ plugins can also be written in order to automatically interact with the simulation by retrieving the flip-flops state and by changing it if required.

[1]The website can be attained through: http://www.perplexus.org

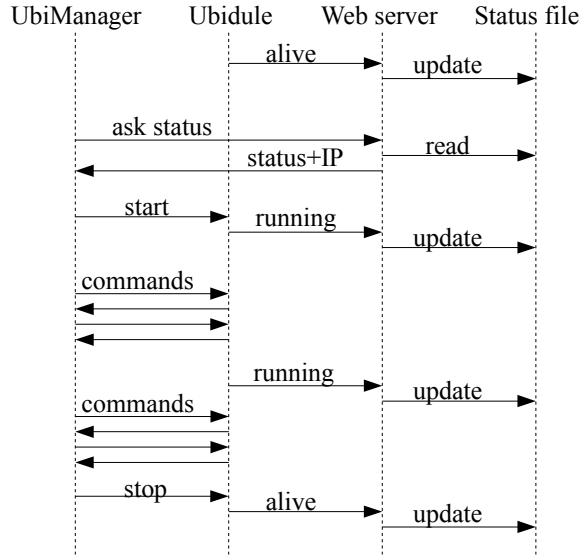

Fig. 10. Scenario of a system test on a ubidule, started from *UbiManager*

Real test on the platform can also be launched from the *UbiManager*, a TCP interface allowing for a wireless communication with the hardware. As there will be around 100 boards at the end of the project, a centralized web site permits a user to monitor which ubidule is active, and which of those are running an application. The *UbiManager* can then connect to a free one in order to test a design.

The population of ubidules, being distributed, is therefore accessible by anyone from all over the world. It makes Perplexus an excellent prototyping platform for complex systems.

ACKNOWLEDGMENT

This project is funded by the Future and Emerging Technologies programme IST-STREP of the European Community, under grant IST-034632 (PERPLEXUS). The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

REFERENCES

[1] E. Sanchez, A. Perez-Uribe, A. Upegui, Y. Thoma, J. Moreno, A. Villa, H. Volken, A. Napieralski, G. Sassatelli, and E. Lavarec, "PERPLEXUS: Pervasive computing framework for modeling complex virtually-unbounded systems," in *AHS 2007 - Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, T. Arslan, A. Stoica, M. Suess, D. Keymeulen, T. Higuchi, R. Zebulum, and A. T. Erdogan, Eds. Los Alamitos, CA, USA: IEEE Computer Society, aug 2007, pp. 600–605.

[2] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. Moreno, J. Madrenas, and G. Sassatelli, "The perplexus bio-inspired hardware platform: A flexible and modular approach," *Knowledge-Based & Intelligent Engineering Systems Journal*, 2008. To appear.

[3] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. Moreno, and J. Madrenas, "The Perplexus bio-inspired reconfigurable circuit," in *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, T. Arslan, A. Stoica, M. Suess, D. Keymeulen, T. Higuchi, R. Zebulum, and A. T. Erdogan, Eds. Los Alamitos, CA, USA: IEEE Computer Society, aug 2007, pp. 600–605.

[4] S. Brown, R. Francis, J. Rose, and Z. Vranesic, *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[5] Y. Thoma and E. Sanchez, "An adaptive FPGA and its distributed routing," in *Proc. ReCoSoc '05 Reconfigurable Communication-centric SoC*, Montpellier - France, Jun. 2005, pp. 43–51.

[6] Y. Thoma, A. Upegui, A. Perez-Uribe, and E. Sanchez, "Self-replication mechanism by means of self-reconfiguration," in *20th International Conference on Architecture of Computing Systems 2007 (ARCS '07), Workshop proceedings*, M. Platzner, K.-E. Grosspietsch, C. Hochberger, and A. Koch, Eds. VDE Verlag, mar 2007, pp. 105–112.

[7] J. Torresen, "Possibilities and limitations of applying evolvable hardware to real-world applications," in *FPL 2000*, ser. Lecture Notes in Computer Science, R. Hartenstein and H. Grünbacher, Eds., vol. 1896. Berlin Heidelberg: Springer-Verlag, 2000, pp. 230–239.

[8] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: The University of Michigan Press, 1975.

[9] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A java-based interface for reconfigurable computing," *Proc. of the Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[10] A. Upegui, Y. Thoma, A. Perez-Uribe, and E. Sanchez, "Dynamic routing on the ubichip: Toward synaptogenetic neural networks," in *AHS 2008 - Proceedings of the 3rd NASA/ESA Conference on Adaptive Hardware and Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2008.

[11] A. Upegui, A. Perez-Uribe, Y. Thoma, and E. Sanchez, "Neural development on the ubichip by means of dynamic routing mechanisms," in *ICES 2008 - Proceedings of the 8th International Conference on Evolvable Systems*. LNCS, Springer Verlag, 2008. Submitted.

[12] J. Peña, J. Peña, and A. Upegui, "Evolutionary graph models with dynamic topologies on the ubichip," in *ICES 2008 - Proceedings of the 8th International Conference on Evolvable Systems*. LNCS, Springer Verlag, 2008. Submitted.

[13] D. Bertizzolo, "Optimisation par essaim de particules auto-réplicatifs," Diploma project report, HEIG-VD, Switzerland, 2008.