Uniform treatment of hardware- and software components

Luka Lednicki, Jan Carlson

Mälardalen Research and Technology Centre
PO Box 883, SE-721 23 Västerås, Sweden
+46 21 {10 15 45, 15 17 22}
{luka.lednicki, jan.carlson}@mdh.se

Mario Žagar

University of Zagreb
Faculty of Electrical Engineering and Computing
Unska 3, HR-10000 Zagreb, Croatia
mario.zagar@fer.hr

ABSTRACT

One of the challenges in development of embedded systems is to cope both with hardware and software components. Often is their integration cumbersome due to their incompatibilities, different specifications and different approaches in their development. In this paper we present a component-based technology we have developed for building distributed systems consisting of both embedded hardware devices and software written in high-level programming languages. To obtain a uniform view on hardware and software we use Universal Plug and Play (UPnP) technology for the communication between these parts of the system. Our technology consists of a component model that allows us to treat UPnP devices as components, and a run-time framework that supports this component model when the system is deployed. To evaluate the principles we have developed a prototype tool that implements the technology and demonstrated a feasibility of the approach.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Languages

Keywords

Embedded systems, UPnP, Component-based development

1. INTRODUCTION

With the continuous advancement of embedded computers their usage grows rapidly. Examples of that can be seen in environmental and industrial monitoring and control, telecommunication, smart houses, and many other domains.

Standard development models have difficulties keeping up with such complex systems, and as a result the development becomes too costly and time consuming, or the produced systems suffer from poor reliability and predictability. In search for a better development process, component-based development (CBD) emerges as a possible solution. It encourages reuse of once developed software or hardware components, and allows some properties of the system to be predicted in the early stages of development, based on the properties of the components it consists of. The use of such approach could greatly reduce development time and make the final product more robust, reliable and efficient.

On the other hand, the increase of available resources makes it possible for embedded devices to implement advanced middleware to communicate with other elements of the system. Although the usage of such middleware may take up more resources than the actual core functionality, in many cases the benefits it provides to the development process outweigh the cost of more powerful hardware. One technology that can be used as middleware is Universal Plug and Play (UPnP).

In this paper we propose a solution for building distributed systems consisting of both hardware and software in a component-based manner using UPnP technology. For this purpose a new component model called UComp has been developed, along with tools for developing and deploying systems built on that model.

The rest of the paper is organised as follows: In Section 2 we introduce UPnP protocol and CBD approach. Section 3 describes our solution for combining hardware- and software components. In Section 4 we present the UComp component model, and Section 5 describes the UComp run-time framework. In Section 6 we describe the tool for visual development UComp systems. In Section 7 we discuss some of the characteristics of our component technology and provide an overview of related work. Section 8 concludes the paper and states the possibilities for future work.

2. BACKGROUND

Before presenting the proposed component model, we give an overview of the UPnP protocol and a short introduction to CBD.

2.1 Universal Plug and Play

UPnP is open standard that provides means for discovery, description and cooperation of different devices using standard TCP/IP network protocols [15]. Its name is derived from Plug-and-Play (PnP), a technology that allows seamless connecting of peripheral devices to a personal computer. UPnP takes that concept and applies it to any device connected to a computer network. To do this, it leverages well established protocols and technologies like IP, TCP, UDP, XML and SOAP.

The two main types of entities that the UPnP architecture defines are *devices* and *control points*. Devices are entities of a UPnP network that provide services. Each service can define an arbitrary number of actions that are used to control the device, and one or more state variables which model the state of the device. Control points are clients to the services that the devices define in that they invoke actions defined by the services and/or monitor the values of their state variables.

UPnP networking is divided in six steps: *addressing*, *discovery*, *description*, *control*, *eventing* and *presentation*.

Through *addressing*, a UPnP device acquires a valid network address. Both managed and unmanaged networks are supported by the standard. If a DHCP server is present on the network it assigns an IP address to the device, otherwise the device uses the Auto-IP protocol to obtain a unique and valid address.

A device that has acquired a network address proceeds with the next step: *discovery*. In this step the device advertises its presence to control points that are connected to the network, using multicast UDP messages in which it states its name, type and location on the network. To acknowledge its presence on the network, a UPnP device repeats this step periodically. Apart from passively collecting advertisement messages from the devices, control points can search the network for available devices, a certain type of devices or services, or a specific device using its unique ID.

Once the control point discovers a device of interest on the network it can initiate the *description* step to gain information about it. It does this by requesting the XML description contained within the device. The description contains detailed information about the device and a list of the services it implements, together with the location of the service description XML for each service. A service description lists all state variables and actions, and defines input and output arguments for each action.

With the information about devices, the control point can start invoking those actions in the *control* step. Actions are invoked by sending SOAP messages containing action name and input arguments to the device. The device then responds with either a message containing output arguments or an error message that contains the code and description of the error.

Parallel to the control, *eventing* step can take place. Eventing enables the devices to notify control points when a state variable of one of its service changes value. To receive such notifications the control point has to subscribe to the events of the service.

The last step that the UPnP device architecture defines is *presentation*. It enables devices to present their functionality through a web page, but this step is not relevant to this paper and will not be described further.

2.2 Component-based development (CBD)

In CBD, systems are built from well-defined *components*. At run-time they are deployed to a *component framework* which supports them and manages their resources. To assure that the components can be deployed they need to conform to a specific *component model*. Component models define how a component interacts with the component framework and with other components in the system.

Components are self-sufficient functional units that communicate with their environment only through well-defined interfaces, which makes them very suitable for reuse. Explicit connections, the result of limiting components to interact only through their interfaces, make system built using CBD easier to analyze and maintain.

Reuse of existing components can further be facilitated by creating component repositories. Once a new component is developed it should be put in such a repository. The repository would then provide means for system developers to browse available components and use them in the development of a system.

General purpose component models like COM [11], JavaBeans [14], .NET [6], EJB [7] and CORBA [3] are already widely used in development of desktop-, web- and distributed applications. The success of such models motivated research in applying the component-based approach in development of embedded systems. However, the resource restrictions faced by most embedded devices make general purpose component models unsuitable for such systems. On the other hand, those restrictions also make the analyzability and predictability that CBD provides even more beneficial. Because of that, new component models are being developed to satisfy both the needs and the constraints of embedded systems. Examples of such models are SaveCCM [1], Koala [16] and Rubus [2].

3. COMBINING HARDWARE- AND SOFTWARE COMPONENTS

While the general purpose component-based technologies provide solutions for high level applications (for example desktop or web applications), component technologies for embedded systems are mostly limited to the resource-constrained systems. A problem arises when trying to connect the two in complex systems consisting of both high level software components and low level embedded components that are closely connected to the hardware. In such systems there is a need for a uniform way of handling both high (software) and low level (hardware) components.

As an example we will take a simple greenhouse temperature monitoring system. It consists of a sensor that monitors the temperature inside a greenhouse, a display showing the current temperature, an alarm that should sound if the temperature exceeds 35°C, and a button that is used to acknowledge an alarm and reset it. The display and reset button is realised as a Java application running on a personal computer.

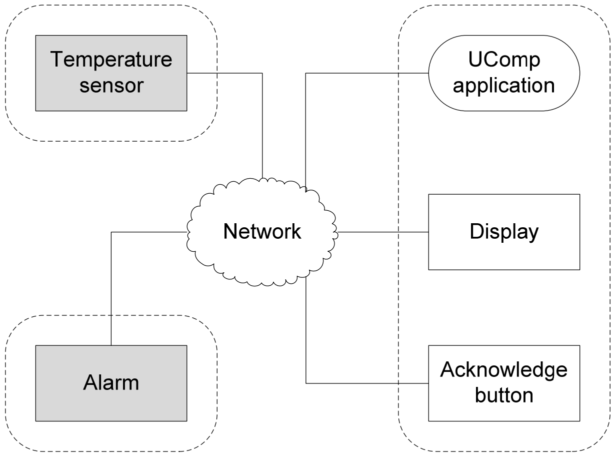

Figure 1: Example system, monitoring the temperature of a greenhouse.

An overview of the system is given in Figure 1. Embedded hardware devices are shown as gray rectangles, while the software applications are shown as white rectangles. In addition, the UComp application that will support the system is shown. Dashed lines separate different physical nodes.

Although this simple system would be easy to implement without using CBD principles, the effort needed for development and maintenance would rise drastically with the increase of system complexity. UComp, the component-based technology that we propose in this paper, addresses that need. Using UPnP to connect and describe the components, we created a component model in which there is no distinction between software and hardware components. At the same time the footprint of the middleware layer is kept at a level which is acceptable for resource-limited embedded devices. UComp also allows for a simple way of creating component repositories.

To support the development we have created a tool for visual development of UComp systems (*UComp Developer*) and a tool for deploying them (*UComp Deployer*) to any Java-enabled platform.

Our solution consists of an application that uses a UPnP control point to communicate with UPnP devices connected to the network. For each of the devices it generates a set of one or more components that are bound to the device and represent its actions and state variables. These components are then presented to the developer who can use them for building a system. We also enable use of components that are not bound to any UPnP device. Instead, the functionality of those components is completely defined by software. The temperature sensor, alarm, display and the alarm acknowledgement button shown in Figure 1 are realised as UPnP devices. The UComp application shown in the figure treats those devices as components and provides the desired functionality of the system.

By having the components available at run-time, systems built in this way can be extremely flexible because any modification of the system can be done while the system is running and the embedded devices are deployed.

3.1 Benefits of using UPnP

Defining architecture and protocols, but not their implementation, makes UPnP platform, language and media independent. To a control point there is no difference between a Java or .NET application acting as a UPnP device and a micro-controller using UPnP device stack.

The use of standard protocols allows UPnP to be used in existing computer networks with little or no modifications. Also, the possibility to use the Internet to connect devices and control points allows the developer to build systems containing devices distributed over large geographical distances in an inexpensive way.

Another benefit of using standards as HTTP and XML is that UPnP is easily extendable. A device vendor can add new information to the device description or control and eventing messages without breaking the UPnP standard.

4. THE UCOMP COMPONENT MODEL

To enable smooth deployment and interaction of distributed components in a system, a new component model was defined. In it, two different component types were defined: *UPnP components* and *software components*. UPnP components wrap around UPnP devices and present their functionalities in a component-based manner. Software components are not associated with any UPnP device. Instead, their functionality is fully implemented in Java code.

Component interfaces consist of input and output *ports*. Ports can be viewed as access points to the component, through which components exchange data and control (triggering) signals. System execution follows the pipes and filter pattern. Data and triggering signals from an output port of one component can be directed to input ports of one or more components.

The UComp component model is loosely based on SaveCCM. Although there are some differences that arise from the different domains and purposes of the two models, while developing the UComp model we wanted to allow for UComp systems to be easily transformed into SaveCCM, and vice versa. Such transformation would give users the ability to use tools developed for SaveCCM to verify and analyze UComp systems, and UComp could provide a way to implement and deploy systems designed in SaveCCM.

4.1 UPnP components

UPnP components represent actions and events of UPnP devices. In respect to that, there are two types of UPnP components: *UPnP action components* (or just *action components*) and *UPnP event components* (*event components* from now on). A single UPnP device corresponds to a set of UPnP components: one event component for each service that the device provides, and one action component for each action defined by a service.

The input- and output ports of UPnP components are generated according to the arguments of the device's actions (for action components) or the state variables of its services (for event components). In addition to these ports, every UPnP component has a Boolean output port named *connected*. This port is set to true if the device is connected to the network (accessible by the control point) and event subscriptions are accepted in case of event components. This information can be very useful in distributed systems where a connection between the distributed components is not reliable. In the case that one or more components are temporarily unavailable, a warning can be signalled, and their functionality can be rerouted to a backup system.

4.1.1 Action components

Action components represent actions of UPnP devices. Every action component is bound to a specific device by its Unique Device Name (UDN), a specific service of that device by the service ID, and in the end to a specific action of that service by the action name.

The ports of action components are generated according to the arguments of the action. For every input argument of the action an input port is added to the component and for every output argument of the action an output port is added, taking into account the data types of each argument. The names of the ports are equal to the names of the arguments. Action components also have an additional input port named *trigger* that accepts any data type. This port can be used for additional triggering, as well as triggering of components whose actions don't have any input arguments.

When a UPnP action component is triggered, values of its input ports (with exception of the "trigger" port) are stored and transformed into input arguments for the UPnP action. Then, a control message is sent to the device to invoke the action. In the end, output arguments are parsed from the result message and their values used to set the values of the output ports.

The action components in our greenhouse temperature monitoring systems are the temperature display and the alarm. To display the temperature we will use the *SetLine* action of the display UPnP device. The action has one input argument named *text* and no output arguments. Thus, the corresponding action component (shown in Figure 2) will have two input ports, *text* and *trigger*, and the *connected* output port.

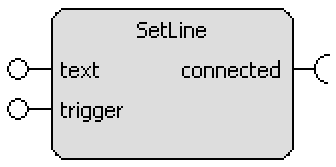

Figure 2: Action component generated for the SetLine action of the display UPnP device.

Event components handle the event notifications generated by UPnP devices. Every event component is bound to a specific device by its UDN and the service ID. When the system is started event component instructs the UPnP control point to subscribe to events of the service they are bound to. Components confirm that subscription in regular intervals, and in the case of loss of subscription, send re-subscription requests.

Ports of event components are generated using the state variable tables defined by the UPnP services. For each evented state variable of the service, an output port is created with the same name as the state variable.

When the control point receives event notification from the service, the new values of state variables are used to set the values of output ports of the component.

In the temperature monitoring example we use event components to obtain the temperature from the temperature sensor and to monitor the state of the alarm acknowledgment button. The event component that would correspond to the *tempSensor* service provided by the sensor UPnP device is shown in Figure 3. It has no input ports, and two output ports: *temperature*, matching the temperature state variable of the service, and the *connected* port that signals if the device is available on the network.

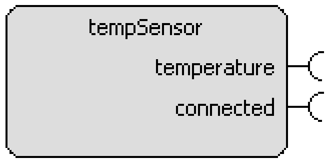

Figure 3: Event component representing state variables of the tempSensor service of the temperature sensor UPnP device.

4.2 Software components

Software components are not associated with any UPnP device; instead their functionality is fully implemented in Java. Some of the roles of software components are to process the data received from, or sent to, UPnP components, manipulate the execution of components (e.g., generation of periodical triggers), data flow control (using switches) and definition of constants. Their function can vary from very simple (for example addition of two numbers, logical operations, extraction of a substring from string) to complex data processing. Having simple functions available as components (together with use of simple data types in component interfaces) makes it unnecessary to write any glue-code when connecting the components and thus enabling easier development.

In our example this types of components are used to compare the temperature read by the sensor components to the temperature limit, and to control the alarm state.

Software components are stored as Java class files. This makes the creation of a component repository fairly simple. For a new component to be available for development and deployment, it only needs to be copied to adequate directory of the file system.

4.3 Ports

Ports are the access points of a component, through which it sends and receives data and triggering signals. They are defined by their names and data types. The names of all input ports and names of all output ports of a component must be unique (although an input port can have the same name as an output port).

4.3.1 Connections between ports

One output port can be connected to multiple input ports, but an input port can be connected to only one output port. Whenever a component sets new data to one of its output ports, the port automatically sends the data and triggering signals to all input ports connected to it. Data is also transferred from an output port to an input port when a connection between the two is made, thus providing better behaviour of the system during run-time modification. The data is always transferred by value, and not by reference. Both input and output ports buffer the last data that was set to them. Ports can also be reset, making the port signal that there is no data available.

4.3.2 Data types

Every port defines a data type for the data it handles. In addition, input ports can define other data types they can accept and cast into their base data type. Although ports could use any Java class for their data type, only five types are currently implemented: *Boolean*, *Integer*, *Double* and *String*. These types are chosen to cover data types defined for UPnP arguments and state variables. A port can also be configured to handle no data, in which case it is used for triggering purposes only.

4.3.3 Triggering of components

When an input port receives a signal from the output port it is connected to, it becomes active.

Every input port has an attribute called *trigger type*. This attribute defines how the state of the port affects the triggering of component execution. Although all components define default trigger types for their input ports, the developer of the system can change that type at any time to achieve the desired system behaviour. There are three types of triggers for input ports:

- Trigger. A component is triggered if all trigger input ports are active.
- Priority trigger. A component is triggered if any of its priority trigger input port is active.

- Data. If port's trigger type is set to data, it is only used to receive data, and does not affect the triggering of the component.

By combining these three trigger types, complex triggering patterns or feedback-loops can be achieved.

The graphical representation of output ports and all input port types can be seen in Figure 4. The figure shows an instance of *Component A* having input ports *a* (data port), *b* (trigger port) and *c* (priority trigger port), and an output port *out*.

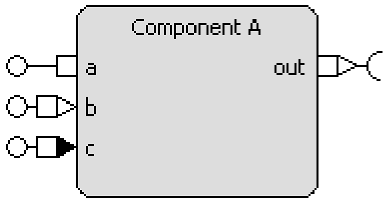

Figure 4: Graphical representation of a component and its ports.

4.4 Component execution

Initially, all components in the system are in an idle state waiting to be activated for execution. Activation can be caused either by the triggering signals received at the input ports of the component, or by its internal events. Once activated the component starts its read-execute-write sequence: First, the component reads all values from its input ports and stores them internally, and then it executes its functionality. Finally, the component updates the values of its output ports.

By looking at the way they are executed, two types of components can be distinguished: *passive* and *active* components. Action components and most software components are passive, meaning that they execute only when they are triggered by signals received from other components, while event components and some software components are active and thus may start their execution by an internal event.

Execution of passive components is done by a part of the framework called the *Executor*. The Executor manages a queue of components that need to be executed and runs a thread that does the actual execution of these components. When a component is triggered, it adds itself to the queue of the Executor object. The execution thread waits until there is at least one component waiting to be executed. Then, it takes a component from the queue and calls its execute method. At the end of a component's execute method all input triggers are reset.

The execution of active components starts by an internal event. In the case of event components, it starts when a UPnP event notification is received by the component. Although the source of this event is in fact external to the system, it is viewed as internal to the component because it was not generated by any interaction with other components. Active components are executed in a separate thread than the passive components, defined by either the UPnP control point (in case of event components) or the components themselves (in case of active software components).

4.5 Example

Figure 5 shows the graphical representation of the greenhouse temperature monitoring system developed using UComp. The system consists of *tempSensor* (temperature sensor UPnP device) and *ButtonPanel* (acknowledgment button UPnP device) event components, *SetLine* (display UPnP device) and *SetAlarmState* (alarm UPnP device) action components and *constant 35*, *Comparator* and *SR* software components. The temperature value from the *tempSensor* is outputted directly to the display and to the *Comparator*, where it is compared with the constant *35.0*. If the temperature is greater than 35.0, the comparator activates the *s* (set) port of *SR* (set/reset flip-flop) that stores the alarm state. The acknowledgment button is connected to the *r* (reset) port of *SR*. The output of *SR* is then connected to the input of the *SetAlarmState* action component.

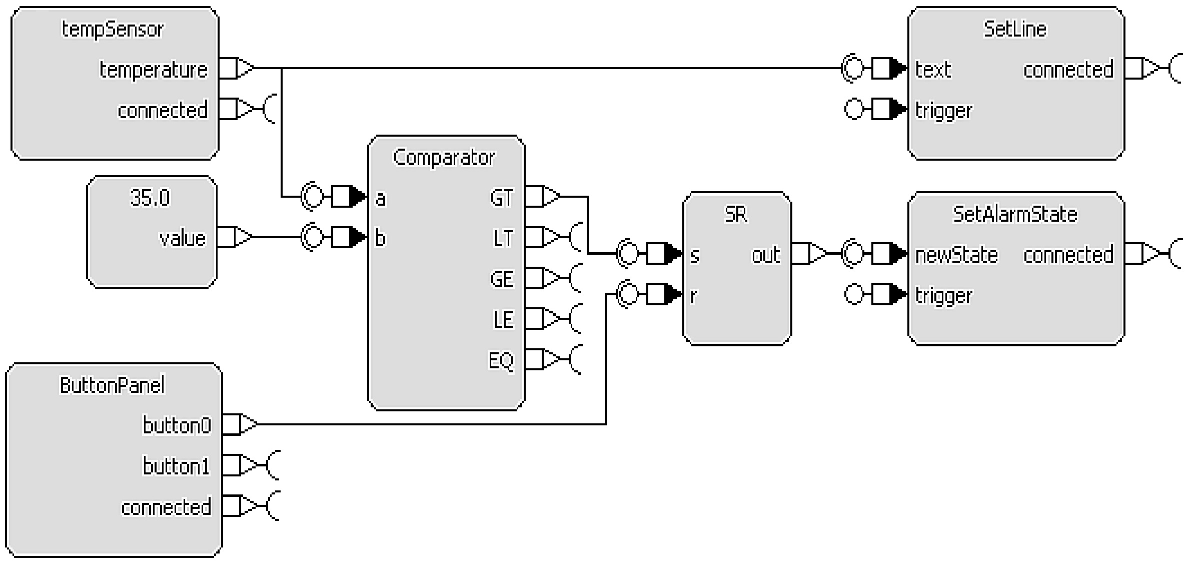

Figure 5: Graphical representation of the temperature monitoring system developed using UComp. The image is a screen-shot of the development panel in the UComp Developer tool.

5. THE UCOMP RUN-TIME FRAMEWORK

The UComp architecture (shown in Figure 6) is conceived as a Java application that controls UPnP devices available on the network, processes their data, and relays data between them. The application communicates with the devices through a single UPnP control point implemented by the CyberLink UPnP stack [4]. The functionality of the system is defined by the components it uses and the connections between those components. This centralized architecture has a number of benefits:

- Data received from a device can be processed by the application before it is forwarded to other devices, making the systems much more flexible and eliminating the need to change the code of the devices to adapt them to the needs of the developed system.
- Embedded devices do not need to implement UPnP control points. These devices have limited memory capacity and processing capabilities. Having to implement the control point stack would significantly decrease their performance.
- Run-time modification of systems is much easier. System's behaviour can be modified by simple changes in the interconnection of components (or by changing the components themselves) in the central application. No change in the behaviour of the devices is needed. If devices were to communicate directly to one another, means for changing their configuration at run-time would have to be devised. Such functionality would mean that standard UPnP devices could not be used. Also, it would take up a portion of device's resources.

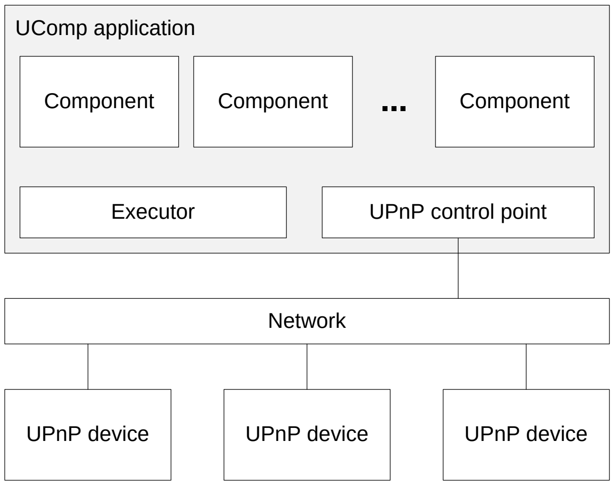

Figure 6: The UComp architecture.

6. DEVELOPMENT TOOL

For building UComp systems, we created a visual development tool named *UComp Developer*. It enables browsing available components, visual representation of components on a development panel, modifying connections between them, setting their properties and the properties of their ports, and starting and stopping the execution of the developed system. Systems developed with this tool can be saved, or restored from, XML files.

The system is developed in the development panel. In it, all components that the system consists of are graphically represented. The graphical representation also shows all input and output ports of the components, together with connections between those ports.

Available UPnP and software components are presented in a tree structure. The tree consists of two main sub-trees: one for the UPnP components and one for software components. The first one is populated by UPnP components representing all devices that are currently available on the network. They are further grouped by the device and the service they are bound to. To generate the software component sub-tree, the application scans the file system (more precisely the Java classpath) for all Java classes that extend the *SoftwareComponent* class. Both the UPnP component sub-tree and the software component sub-tree can be refreshed while the *UComp Developer* application is running.

7. DISCUSSION AND RELATED WORK

While testing the systems built with UComp some false disappearance of devices were detected. They were caused by the use of unreliable UDP protocol in UPnP advertising. In case too many UDP messages were lost on the network, the control point concluded that a device is no longer connected to the network. This undesired behaviour could be eliminated by modifying control points to use UPnP control messages to test for the existence of the devices that are about to expire.

Another problem that arose was long execution time of UPnP action invocations. In our experiments with the example temperature monitoring system, using Rabbit RCM2200 microcontrollers with a custom built generic UPnP device stack as embedded devices, it varied between 100 and 350ms. The SOAP protocol that is used in the invocations is somewhat complex when applied on embedded devices. A solution for this would be to extend UPnP with an additional control protocol for use with the embedded devices. Such protocol could coexist with the standard UPnP control protocol and be used when both the device and control point supports it. A solution for a better control protocol using representational state transfer (REST) approach is given in [9].

In the process of development, two different types of component executors were investigated. The first one started a new Java thread for the execution of each component, while the second sequentially executed all components in the same thread. As the time needed to start a new thread for each execution surpassed the time spent on the execution of the code in many simple software components, we have decided to use sequential execution for the software components.

Use of CBD in developing embedded systems has been explored in component models such as SaveCCM [1], COMDES-II [8], Rubus [2] and Koala [16]. However, most of these models do not specifically aim to solve the problem of connecting hardware with software. In addition, they focus their component-based approach on the design-time, loosing the benefits of components at run-time.

An alternative standard for connecting embedded devices and software used in industry is OPC [10]. It uses Microsoft's COM [11] and DCOM [5] technologies for communication between OPC servers (embedded devices) and OPC clients. At the time

OPC does not provide means for controlling devices in form of executing commands.

UPnP was also explored as a middleware for robot development in [12] and [13]. The work describes benefits of using UPnP over real-time CORBA (TAO) in such an embedded environment. The work also introduces extensions to standard UPnP protocols that allow UPnP to better accommodate the needs of a robot SDK.

8. CONCLUSION AND FUTURE WORK

In this paper we have proposed a simple component-based technology for developing systems containing both embedded hardware and high level software applications. This was achieved by using UPnP architecture as middleware for discovering components, describing them and managing connections between them. In our component model we have achieves a uniform way of looking at hardware and software components. To further improve the development process, we have created a tool that enables browsing of available components and visual composition of systems. We have demonstrated how this technology could be applied on a simple temperature monitoring system.

As future work, system design could be enhanced by providing a UPnP component repository in the development tool. This could easily be achieved by storing UPnP device descriptions to files in a well-organised directory structure.

The component model could further be improved by including functional and non-functional properties in UPnP device description. Attributes specifying the same properties could also be added to software components. Using those properties we could do a detailed analysis of the system both at the design and run time.

To increase the performance of the system, the UPnP protocols could be extended to better fit the needs of embedded devices and systems. This would include improving the UPnP discovery and defining a light-weight control protocol.

9. ACKNOWLEDGEMENT

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and the Unity Through Knowledge Fund supported by the Croatian Government and the World Bank via the DICES project.

10. REFERENCES

[1] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE approach to component-based development of vehicular systems. Journal of Systems and Software, 80(5):655–667, May 2007

[2] Arcticus Systems, Rubus Software Components, Available from www.arcticus-systems.com

[3] F. Bolton. Pure CORBA. Sams, 2001

[4] S. Konno, Cyberlink for Java, http://www.cybergarage.org/

[5] Microsoft, .DCOM Technical Overview, http://msdn.microsoft.com

[6] Microsoft, .NET, http://www.microsoft.com/net/

[7] R. Monson-Haefel. Enterprise JavaBeans. O’Reilly and Associates, 2001

[8] X. Ke, . Sierszecki, C. Angelov, COMDES-II: A component-Based Framework for Generative Development of Distributed Real-Time Control Systems, RTCSA, pages 199-208, 2007

[9] J. Newmarch, A RESTful approach: clean UPnP without SOAP, Consumer Communications and Networking Conference, pages 134-138, January 2005

[10] OPC Foundation, .OPC, OLE for Process Control,. Report v1.0, OPC Standards Collection, 1998, http://opcfoundation.org.

[11] D. Rogerson. Inside COM. Microsoft Press, 1997

[12] Sang Chul Ahn, Jung-Woo Lee, Ki-Woong Lim, Heedong Ko, Yong-Moo Kwon, Hyoung-Gon Kim, UPnP Approach for Robot Middleware, Proceedings of the 2005 IEEE International Conference on Robotics and Automation, pages 1959-1963, April 2005

[13] Sang Chul Ahn, Jung-Woo Lee, Ki-Woong Lim, Heedong Ko, Yong-Moo Kwon, Hyoung-Gon Kim, UPnP SDK for Robot Development, SICE-ICASE, pages 363-368, October 2006

[14] Sun Microsystems, JavaBeans Specification, http://java.sun.com/beans/

[15] UPnP Forum, UPnP Device Architecture 1.0, http://www.upnp.org/resources/documents/

[16] R. van Ommering, F. van der Linden, and J. Kramer, The Koala component model for consumer electronics software. In IEEE Computer, pages 78–85, IEEE, March 2000