UNIVERSITY OF ZAGREB
MÄLARDALEN UNIVERSITY

COMPONENT-BASED DEVELOPMENT FOR SOFTWARE AND HARDWARE COMPONENTS

Luka Lednicki

Zagreb, May 2008

Table of Contents

1 Introduction

Today we are witnesses to growing integration of computers into our environment. Such embedded devices often have the possibility to connect to TCP/IP networks which are standard in today's computing. This allows the devices to be connected in an easy and economic way. Distributed systems built in such a way can range from devices connected by a local network to devices scattered over the globe communicating via the Internet. One of the technologies that can be used for connecting network-enabled devices is *Universal Plug and Play* (UPnP). It allows distributed devices to be discovered, described, controlled and their state monitored.

Rapid development of embedded computers leads to increasing complexity of embedded systems. Standard development models have difficulties keeping up with such complex systems: their development becomes too costly and time consuming, or produced systems are unreliable and unpredictable. In search for a better development process, *component-based development* (CBD) asserts itself as an obvious choice. It is a concept well proven and widely used in electronic engineering. It is also increasingly used in software engineering. Use of CBD in building embedded systems is still in its early stages, mainly because of the strict constraints this systems face.

The aim of this thesis is to create a simple component model for developing distributed embedded systems using UPnP technology and explore the characteristics of that model.

2 Universal Plug and Play

Universal Plug and Play (UPnP) is a technology that defines an architecture for seamlessly connecting different network devices.

The term UPnP is derived from *Plug and Play* (PnP), a technology that enables dynamically connecting peripherals to a personal computer, without requiring reconfiguration or manual installation of device drivers. UPnP takes the idea of PnP and generalizes it (therefore the 'Universal' in the name) to enable cooperation of any two devices connected to a computer network without any manual configuration by the user. The technology enables network devices to be discovered, describe their capabilities, be controlled and exchange information with other network devices. All this is done by defining a set of protocols for communication between devices, leaving the networking media independent. UPnP devices can be implemented using any programming language and on any operating system.

The development of UPnP architecture is controlled by *UPnP Forum*. UPnP Forum is an industry initiative designed to enable simple and robust connectivity among consumer electronics, intelligent appliances and mobile devices from many different vendors. It consists of more than eight hundred vendors.

2.1 UPnP protocol stack

The UPnP protocol stack is based on well known and standardized Internet protocols. This allows UPnP to be used on today's standard TCP/IP networks. The advantage of this is that there is no need to building separate network infrastructure for connecting the devices. It also enables UPnP technology to run on many media that support IP.

Using these protocols also makes building distributed systems composed of UPnP devices much easier because they can communicate over the Internet. Although multicast messages used in UPnP discovery will most likely be blocked by Internet routers, the problem can be solved using hardware or software to create a Virtual Private Network (VPN) between distributed sites.

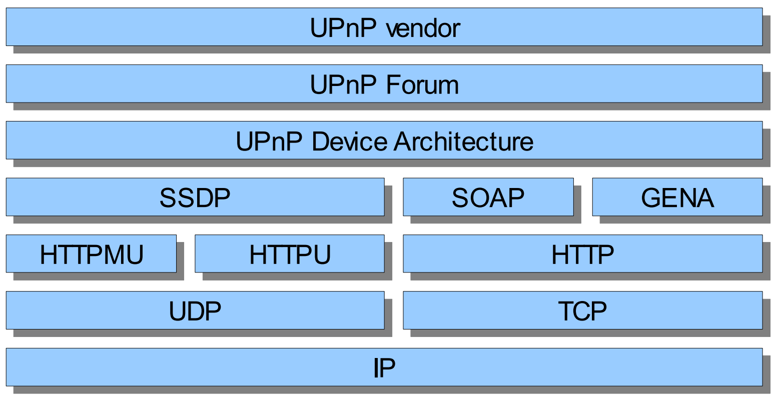

Figure 2.1: UPnP protocol stack [1].

Because all the networking protocols are based on HTTP and XML, UPnP can easily be extended by adding new information, without affecting its standard behavior.

2.2 Elements of UPnP architecture

Two general classifications of entities are defined by UPnP architecture: *controlled devices* (or simply "devices") and *control points*. A UPnP entity can also be a combination of the two, it can implement both a device and a control point.

2.2.1 Devices

A device is a component of UPnP network that provides services. A device functions in the role of a server, responding to requests from control points.

Because UPnP architecture defines protocols for communication between UPnP entities, and not their implementation (API), any network device or application that adheres to UPnP device protocol is also a UPnP device. Consequence of this is that, for an example, UPnP-enabled network router and a desktop application that implements UPnP device stack are equally worth UPnP devices, and can be discovered, described, and controlled in the same way.

2.2.2 Services

Every UPnP device provides one or more services. A service represents a set of functionalities of a device. Every service is defined by its actions and a table of state variables.

An action is a command that the service responds to. Actions can have input and output arguments.

State variable table lists the variables that model the state of a service.

2.2.3 Control points

A control point functions in the role of a client. It discovers UPnP devices on the network and utilizes their services.

When a control point discovers a device it can:

- request the device description,
- request a description of a service provided by the device,
- control the device by executing its actions,
- subscribe to the events of a service provided by the device and receive notifications when the state of the service changes.

2.3 Steps of UPnP networking

UPnP networking is divided in six steps [1].

- **Step 0, addressing**, step in which the device acquires an IP address.
- **Step 1, discovery,** in which a control point discovers devices that are available on the network.
- **Step 2, description**, in which the control point learns more about a device and its capabilities.
- **Step 3, control**, enables the control points to control the devices it discovers.
- **Step 4, eventing**, enables the devices to provide information about their state to control points.
- **Step 5, presentation,** can be used to provide the control point with a HTML page which can be used to control and/or view the state of the device.

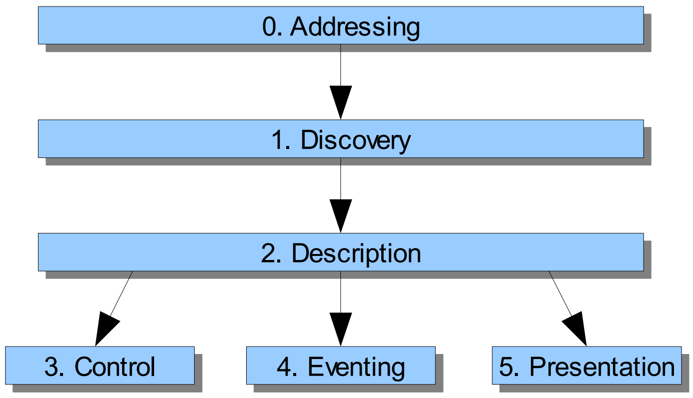

Figure 2.2: Steps of UPnP networking [6].

2.3.1 Addressing

Addressing is Step 0 of UPnP networking [1]. Through addressing, devices get a network address.

The foundation of UPnP networking is IP addressing. Each UPnP device which does not itself implement DHCP server must have a DHCP client and search for a DHCP server when the device is connected to the network (if the device itself implements DHCP server, it may allocate itself an address from the pool it controls). If the DHCP server is available (the network is managed) the device must use the IP address assigned to it. If no DHCP server is available (the network is unmanaged) the device must use automatic IP addressing (*Auto-IP*) to obtain an address. A device that has auto-configured IP address must periodically check for existence of a DHCP server.

Once a device has a valid IP address for the network, it can be located and referenced on that network through that address. There may be situations where the end user needs to locate and identify a device. In this situations, a friendly name for the device is much easier for a human to use than an IP address. If a UPnP device chooses to provide a host name to a DHCP server and register with a DNS server, the device should either ensure that the requested host name is unique or provide a means for the user to change the host name. Most often, UPnP devices do not provide a host name, but provide URLs using literal (numeric) IP addresses.

2.3.2 Discovery

Discovery is step 1 of UPnP networking [1]. For discovery, UPnP uses protocol know as SSDP (Simple Service Discovery Protocol).

When a device is added to the network, SSDP allows that device to advertise itself, any embedded devices it contains and its services to control points on the network. It also periodically re-advertises itself to confirm its presence on the network.

Similarly, when a control point is added to the network, the discovery protocol allows that control point to search for devices of interest on that network. The control point can search for all devices on the network, a specific device defined by its Unique Device Name, just a specified device type or just a specified service type.

In all cases, the device sends a discovery message containing few, essential specifics about the device or one of its services, e.g., its type, identifier, and a pointer to more detailed information.

To send messages, HTTPMU protocol is used, which stands for “HTTP over UDP (multicast)”. HTTP messages are sent over UDP protocol to multicast address and port 239.255.255.250:1900. Any network devices listening for that multicast address and port receives the messages.

Using multicast UDP makes the discovery protocol very simple, but it also has its drawbacks. UDP does not guarantee reliability in the way that TCP does. UDP messages (datagrams) may go missing without notice. This is why UPnP recommends sending each discovery message more than once.

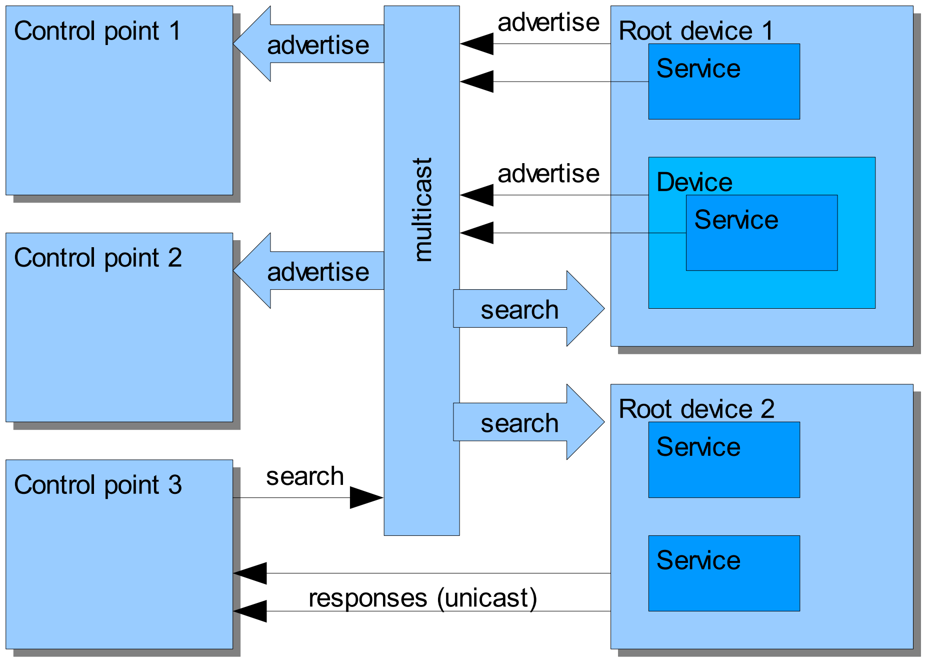

Figure 2.3: UPnP messaging during discovery [1].

2.3.3 Description

Description is step 2 of UPnP networking [1].

After a control point has discovered a device, the control point still knows very little about the device. For control point to learn more about the device and its capabilities or to interact with the device, it must retrieve the device's description from the URL provided by the device in the discovery message.

Device description

The description for a device is expressed in XML and includes vendor-specific, manufacturer information like the model name and number, serial number, manufacturer name, URLs to vendor-specific Web sites, etc. The description also includes a list of any embedded devices or services, as well as URLs for control, eventing and presentation. Every service device offers has its separate description. URLs to that descriptions are given in the service list of device description.

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>base URL for all relative URLs</URLBase>
  <device>
    <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
    <friendlyName>short user-friendly title</friendlyName>
    <manufacturer>manufacturer name</manufacturer>
    <manufacturerURL>URL to manufacturer site</manufacturerURL>
    <modelDescription>long user-friendly title</modelDescription>
    <modelName>model name</modelName>
    <modelNumber>model number</modelNumber>
    <modelURL>URL to model site</modelURL>
    <serialNumber>manufacturer's serial number</serialNumber>
    <UDN>uuid:UUID</UDN>
    <UPC>Universal Product Code</UPC>
    <iconList>
      <icon>
        <mimetype>image/format</mimetype>
        <width>horizontal pixels</width>
        <height>vertical pixels</height>
        <depth>color depth</depth>
        <url>URL to icon</url>
      </icon>
      XML to declare other icons, if any, go here
    </iconList>
    <serviceList>
      <service>
        <serviceType>urn:schemas-upnp-
org:service:serviceType:version</serviceType>
        <serviceId>urn:upnp-org:serviceId:serviceID</serviceId>
        <SCPDURL>URL to service description</SCPDURL>
        <controlURL>URL for control</controlURL>
        <eventSubURL>URL for eventing</eventSubURL>
      </service>
      Declarations for other services defined by a UPnP Forum working
committee (if any) go here
      Declarations for other services added by UPnP vendor (if any) go here
    </serviceList>
    <deviceList>
      Description of embedded devices defined by a UPnP Forum working
committee (if any) go here
      Description of embedded devices added by UPnP vendor (if any) go here
    </deviceList>
    <presentationURL>URL for presentation</presentationURL>
  </device>
</root>
```

Code 2.1: Device description template [1].

Service description

For each service, the description includes a list of actions the service responds to, input and output arguments for each action, and a list of state variables. These variables model the state of the service at run time, and are described in terms of their data type, range and event characteristics.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <actionList>
    <action>
      <name>actionName</name>
      <argumentList>
        <argument>
          <name>formalParameterName</name>
          <direction>in xor out</direction>
          <retval />
          <relatedStateVariable>stateVariableName</relatedStateVariable>
        </argument>
       Declarations for other arguments defined by UPnP Forum working
committee (if any) go here
      </argumentList>
    </action>
    Declarations for other actions defined by UPnP Forum working committee
(if any) go here
    Declarations for other actions added by UPnP vendor (if any) go here
  </actionList>
  <serviceStateTable>
    <stateVariable sendEvents="yes">
      <name>variableName</name>
      <dataType>variable data type</dataType>
      <defaultValue>default value</defaultValue>
      <allowedValueList>
        <allowedValue>enumerated value</allowedValue>
        Other allowed values defined by UPnP Forum working committee (if
any) go here
      </allowedValueList>
    </stateVariable>
    <stateVariable sendEvents="yes">
      <name>variableName</name>
      <dataType>variable data type</dataType>
      <defaultValue>default value</defaultValue>
      <allowedValueRange>
        <minimum>minimum value</minimum>
        <maximum>maximum value</maximum>
        <step>increment value</step>
      </allowedValueRange>
    </stateVariable>
    Declarations for other state variables defined by UPnP Forum working
committee (if any) go here
    Declarations for other state variables added by UPnP vendor (if any) go
here
  </serviceStateTable>
</scpd>
```

Code 2.2: Service description template [1].

2.3.4 Control

Control is step 3 of UPnP networking [1]. Through control, control points can send actions to device's services and query the services state variables.

Invoking actions

To invoke an action to a device's service, control point needs to send control message to the control URL of the service. Control messages are expressed in XML using Simple Object Access Protocol (SOAP). In the message, action name and values for the input arguments of the action are sent to the service. As the result of the action, service responds with a message containing output arguments of the action. If the action failed, a message containing error code and description is returned.

Query for state variables

Using control, control points can also query service's state variables. When control point sends a query message to the service specifying a state variable, service returns a message with the current value of that variable.

2.3.5 Eventing

Eventing is step 4 of UPnP networking [1]. Eventing enables control points to receive notifications when a state variable changes its value. Eventing messages are built upon General Event Notification Architecture (GENA) and XML.

Subscription

If a control point wants to be notified when service's state variables are changed, it needs to subscribe to the events of that service. It does that by sending subscription message to the service's event notification URL (which is defined in the device description). Service then returns a message with information whether the subscription was accepted or not, and a subscription ID along with the expiration time if the subscription was accepted. If the subscription was accepted, service also sends initial event notification with values of all evented state variables. Control point needs to send re-subscription message before the expiration time to renew subscription or the service will cancel the subscription upon expiration.

If a control point does not want to receive notification messages it needs to send an “unsubscribe” message to the service's event notification URL or wait while the subscription expires.

Event notification

Whenever an evented state variable changes its value, service sends notification message containing the new value to all subscribed control points. Whether a state variable is evented or not is defined in the state variable list of the service description.

All notification messages contain their sequence number so the control point can detect if any notification message is missing.

2.3.6 Presentation

Presentation is step 5 of UPnP networking [1].

If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser, and depending on the capabilities of the page, allow a user to control the device and/or view device status. The degree to which each of these can be accomplished depends on the specific capabilities of the presentation page and device.

2.4 Problems with UPnP architecture

Although UPnP has done much to enable simple connectivity of devices, there are still some open issues, which can present mild or severe problems, depending on the configuration and purpose of the system.

Among the biggest of these issues are:

- lack of authentication,
- use of UDP multicast,
- protocol complexity.

2.4.1 Lack of authentication

The UPnP architecture does not implement any authentication [7]. It assumes that all devices connected to the network are trustworthy and are not infected with any malicious software. This problem can be resolved by implementing custom authentication mechanisms, or the standardized Device Security Service on the devices, but since these protocols are quite complex, very little devices do this. Routers and firewalls running the UPnP Internet Gateway Device (IGD) protocol (allowing retrieval of the external IP address, enumeration of the existing port mappings, and adding/removing port mappings) are vulnerable to attacks since the framers of the protocol omitted to add any standard authentication method.

2.4.2 Use of UDP multicast

In step 1 of UPnP networking, discovery, UDP multicast messages are used by the devices to advertise themselves to control points, and by control point to search for devices connected to the network. UDP is a connectionless protocol, and does not guarantee reliability like TCP does. Datagrams may arrive out of order, appear duplicated or go missing without notice. The last can cause problems in UPnP networking. If notification message advertising a device disappears, control point may not detect that a new device is connected to the network, or conclude that a device is no longer available. Similarly, if a device misses a control point's search message, it will not respond to that message, causing the control point not to discover that device. There is no solution to this problem, and only improvement can be done by, as the UPnP architecture suggests, sending messages over UDP more than once.

2.4.3 Protocol complexity

To achieve openness and better standardization, UPnP uses standard Internet protocols and protocols based on XML. Although these can be viewed as lightweight for modern personal computers, they are still pretty complex for small and embedded devices. Since these devices are of biggest interest to the UPnP networking, that can represent a problem.

Tests of the implementation of UPnP device stack were made on RCM2200 network microcontroler with a CPU clock rate of 22.1 MHz, 256 KB of flash memory and 128 KB of ram memory. The stack took nearly 150 KB of the flash memory. Average of execution time of action invocations (from the time the invocation was sent to the time the response was received by the control point) was between 100 and 250 ms.

3 Component-based development

Component-based development (CBD) is a concept well known (almost inevitable) and proven in development of hardware systems. It is based on developing complex systems out of smaller, well defined components. Although the use of CBD principles in developing software has advanced in the last decade, *Component-based software engineering* (CBSE) still has a lot of room for improvement.

Main goals of CBD are [8] [9]:

- reuse of components, thus shortening time-to-market of new systems,
- making the systems easier to maintain and upgrade by making their components easily replaceable and deployable,
- making the system development easier and more reliable by predicting system properties from the properties of its components.

3.1 *Basic concepts*

In CBD systems are built by combining *components* using their *interfaces*. To connect interfaces of two components, *contracts* of those interfaces must be satisfied. To ensure that the components can be deployed to a *component framework* that will support them at run-time, and that they can interact with each other, *component models* are used.

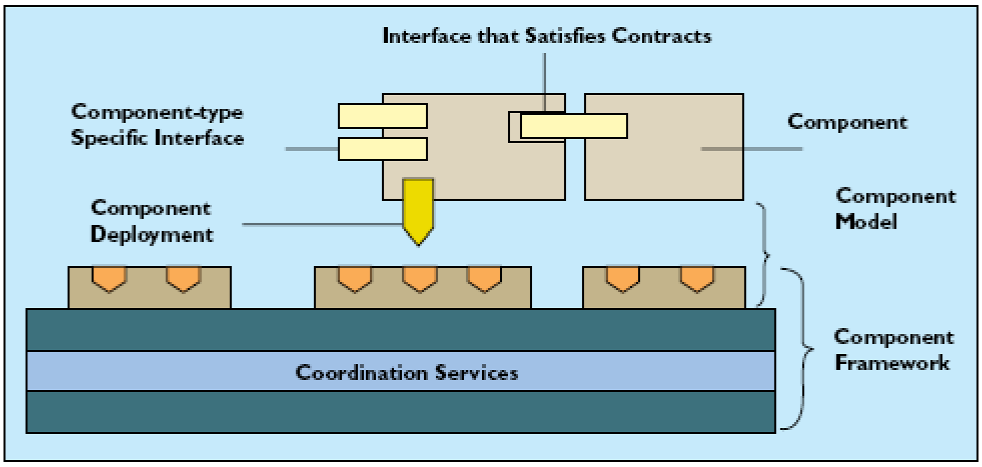

Figure 3.1: Relation between basic concepts of component-based technology [11].

3.1.1 Components

Components are the basic building blocks in the Component-based development. There are many definitions of a component. One of the most accepted is given by Szyperski [9]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.*

Most important characteristic of components are [8]:

- reusability, the ability of a component to be used in different systems,
- usability by third parties,
- seamless replacement of a component with newer or different components.

To achieve this it is necessary to separate the implementation of a component from its interface.

3.1.2 Interfaces

Interfaces can be viewed as access points through which the components can exchange information and cooperate with other components or the component framework.

Interfaces can generally be divided into two groups: provided and required interfaces. Provided interfaces define services that a component can provide to other components or the framework. Required interfaces define services that the component requires from other components or the framework.

As already stated, it is essential that the component's interfaces are separated from the implementation of the component's services. They are used just to list and describe those services. By having the services listed and described, it is much easier to combine components into complex systems. The separation of interface and implementation enables a component to be replaced with new component that provides the same interface.

In most modern component models (e.g. COM, JavaBeans, .NET) interface's description gives just syntactical information. In many cases this information is not sufficient and the need of contractual definition of interfaces arises.

3.1.3 Contracts

Interface semantics can be viewed as contracts between the provider and the user of the interface [12]. Through a contract, user of the interface obliges to constraints (preconditions) that the provider sets, and in respect to that, provider of the interface guarantees some functional and/or non-functional properties.

Hierarchically, we can divide contract definition in four levels [9].

- **Level 1: Syntactic interface.** A list of operations including types of their inputs and outputs. Using the knowledge about types of inputs and outputs, type safety can be established. Type safety ensures that no run-time error will occur from usage of operation with wrong type of object.
- **Level 2: Constraints on values of parameters and of persistent state variables.** This constraints can be viewed as pre- and post- conditions for an operation. An example would be a range for the value of variable.
- **Level 3: Synchronization between different services and method calls.** This level describes the ordering between different interaction at the component interface. It also enables the interaction between the component and its environment to be non-atomic. Automata, temporal logic, process algebras or sequence diagrams can be used for the description.
- **Level 4: Extra-functional properties.** Level 4 describes properties like latency, worst-case execution time, memory usage, reliability, robustness and availability. These properties are of great importance in real-time, embedded and safety-critical domains. By knowing the extra-functional properties of the components that will be used in the system, some properties of the system can be derived before it is actually built.

3.1.4 Component model

Component model imposes a set of conventions that the components using that model must adhere to. With that conventions, component model ensures that the components can be deployed to the component framework and interact with each other.

Types of rules that component models define:

- types of components that can be used,
- how the components interact with each other,
- how the components bind resources.

In a way, component models define the architecture of systems. That limits the flexibility of the system, but also speeds up the process of the development because new architecture does not have to be created.

3.1.5 Component framework

Component framework is a run-time infrastructure that upholds the component model [9]. It manages resources for components and supports component interaction.

Component framework can be viewed as an operating system for the components [9]. From that viewpoint, components are to framework what processes are to the operating system. The difference is that the component frameworks are more compact than operating systems. They are specialized to support a limited range of component types and interactions between those types. By limiting the diversity, component composition becomes simpler, more robust and more predictable.

Another difference between component frameworks and operating systems is that the implementation of the framework does not have to be completely separated from the components. It is possible that a part of the framework is implemented by components themselves.

3.2 Component-based development process

In order to fully utilize the advantages of component based architecture, standard development process needs to be adapted to CBD. Component based development process is divided into two processes: development of components and development of systems [10]. In most cases this two processes can be completely separated from each other. As shown in Figure 3.2, in component development process components are developed and stored in a common *component repository.* In system development process, components are fetched from the component repository and used to build systems. Separation of this two processes allows parallel development which results in shorter time-to-market. It also enables organization of a global component market.

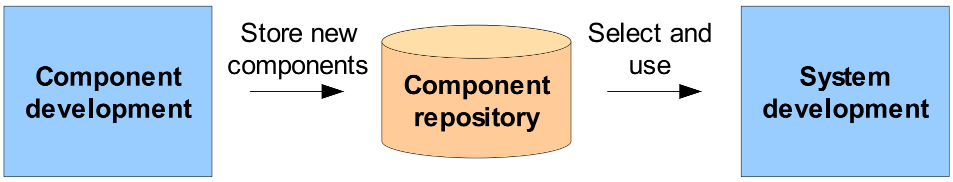

Figure 3.2: Component-based development process.

3.2.1 Component development process

Development process for components does not differ much from the standard development processes used in software development. Main difference is that components are developed to be fully reusable and available for composition by third party. There is no (or very little) knowledge about the systems the component will be used in. Because of that component's functionality must be carefully devised, well defined and thoroughly tested.

3.2.2 System development process

Due to the orientation to reuse in CBD, standard system development must be adapted to system development using components. This changes will be discussed on the adapted V-model shown in Figure 3.3.

First difference is in the way requirements and system design are defined. In these two stages the component model used and components that are available must be taken into account. In some cases requirements will have to be conformed to fit available components because adaptation or development of new components would be to expensive or time consuming. In other cases components that support features not originally foreseen may be found, allowing enhancements in the system.

Instead of unit implementation, component-based development of systems is based on finding and reusing components. Here, three stages can be identified.

- **Selecting a component**. This stage consists of finding and evaluating existing components in search of a component that satisfies the requirements and fits system design. In case that an appropriate component cannot be found it must be created.
- **Adaptation of the component**. Components will often implement a generalized functionality to extend the possibility of reuse. Because of that, most components will have to be adapted to fit the system design.
- **Testing**. When reusing components (regardless if they were developed by third party or not) their specification, adaptation and composition with other components has to be thoroughly tested to be sure that they work properly, and that the system specifications are satisfied.

Operation and maintenance stage of system life cycle is based on updating, replacing or adding new components. This approach makes the maintenance much easier and the system much more suitable for changes than a monolithic one because there is no need to rebuild the whole system. When replacing or adding a new component, the same select-adapt-test process that was used while building the system is used.

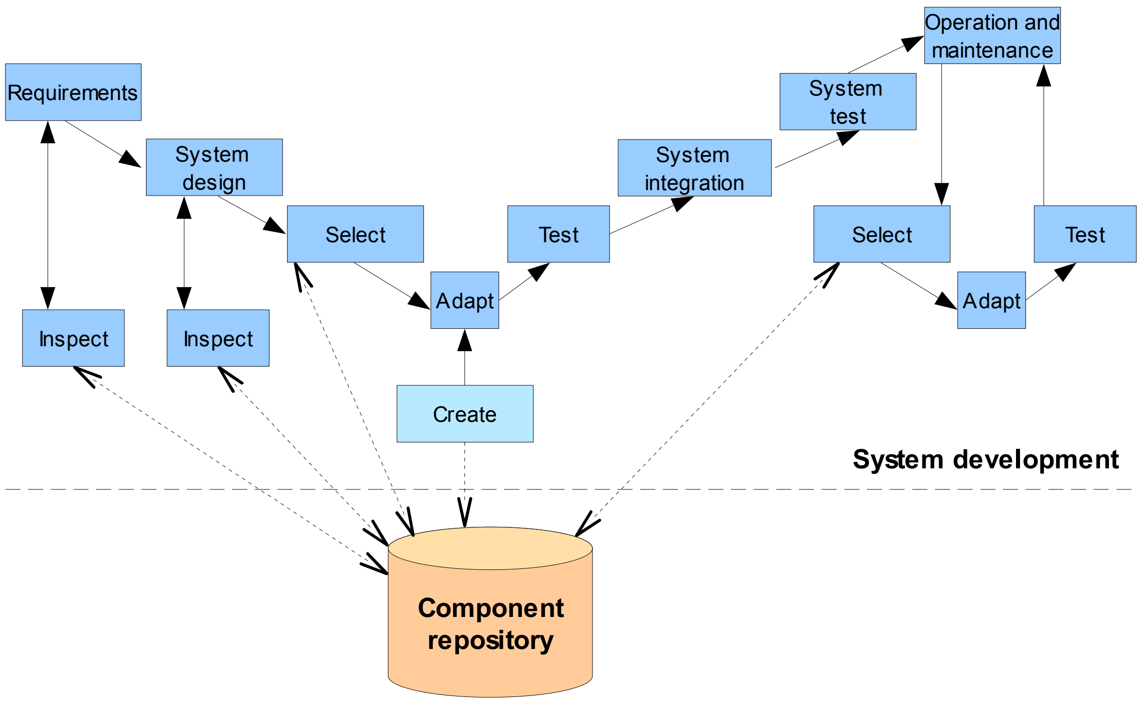

Figure 3.3: V development model for component-based system development [10].

3.3 Component-based development for embedded systems

Embedded systems use computers to monitor and control their environment. Most of them need to work in real-time and often have a safety-critical role. Due to their need to blend into the environment they usually have very limited memory and processing power at their disposal.

With the growing complexity of such systems the need for development model that would enable reuse of once developed parts and provide a way for more efficient development that results in more reliable systems arises. These need could be met through component based development.

3.3.1 Main issues

System constraints

Due to the limited memory and processing power available, component models for embedded systems must have high run-time efficiency. Most general purpose component models (e.g. JavaBeans, EJB, .NET, COM...) are built to maximize the efficiency of the development process, counting on the powerful hardware to deal with the heavy overhead of the model and the framework.

Meeting the required quality attributes

Quality attributes are often as important as the functionality of the embedded systems. Component-based embedded systems in the real-time and safety-critical domain must provide robustness, reliability and predictability. Unpredictable behavior in such systems can result in immense damage or even loss of life. With the ability to determine some of the properties of systems by knowing the properties of the components that make up the system, CBD could meet the quality demands set by the embedded domain. Unfortunately, these methods are still in the research phase and most of the component models in use do not enable definition of component's semantic and extra-functional properties that are essential in the embedded domains.

3.4 The SaveComp Component Model

SaveComp Component Model (SaveCCM) is a component modeling language for embedded systems designed with vehicle applications and safety concerns in focus [14].

Systems are built from interconnected element with well-defined interfaces consisting of input and output ports. There are three types of elements: *components*, *switches* and *assemblies*. Element hierarchy is achieved by *composite components* and *assemblies*. The model is built on XML syntax, and a modified subset of UML2 diagrams is used for the graphical notation of the elements (shown in Figure 3.4).

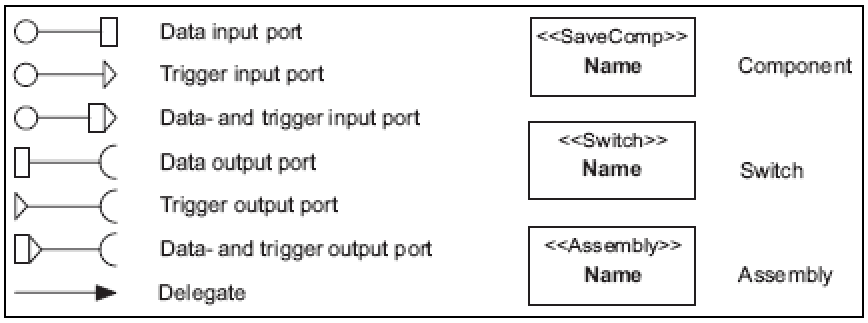

Figure 3.4: Graphical notation of SaveCCM [14].

To enable the analysis of the system during development, semantic information is provided for the components. The semantics is defined by a transformation into timed automata with tasks which allows modeling of timing and real-time task scheduling.

3.4.1 Components

Components are basic building blocks of SaveCCM. Their interfaces consist of *ports* used for communication with other elements in the system and a series of quality attributes, e.g. worst case execution time for different target platforms, reliability estimates, safety models, etc. The attributes are associated with values and can define a measure of confidence. These attributes can be used to analyze the developed system and predict its characteristics.

The functionality of the component is typically provided by a single function implemented in C. The component execution process is described in section 3.4.6.

3.4.2 Switches

Switches provide the means to change the component interconnection structure, either statically for pre- run-time static configuration, or dynamically, run-time switching. The switch defines a number of mappings between its input and output ports. Each mapping has a logical expression over the values on the input ports which is used to determine if that mapping is active or not. If the values of the ports that are used in the logical expressions are fixed, the switch is optimized into ordinary connection in the final system.

Switches do not use trigger ports to activate their execution. All data or trigger signals that arrive to input ports of the switches are immediately relayed to its output ports accordingly to the active port mapping.

3.4.3 Composite components

Composite components are special type of SaveCCM components whose behavior is defined by internal structure. An example of a composite component is given in Figure 3.5. The gray area represents the separation between the external interface of the composite component and its internal structure. Dashed lines describe the transfer of data between external and internal ports. Unlike the data, trigger signals are not transferred. When the component becomes active (described in section 3.4.6) all trigger ports to the internal composition will become active. Trigger signals sent from the internal components are discarded.

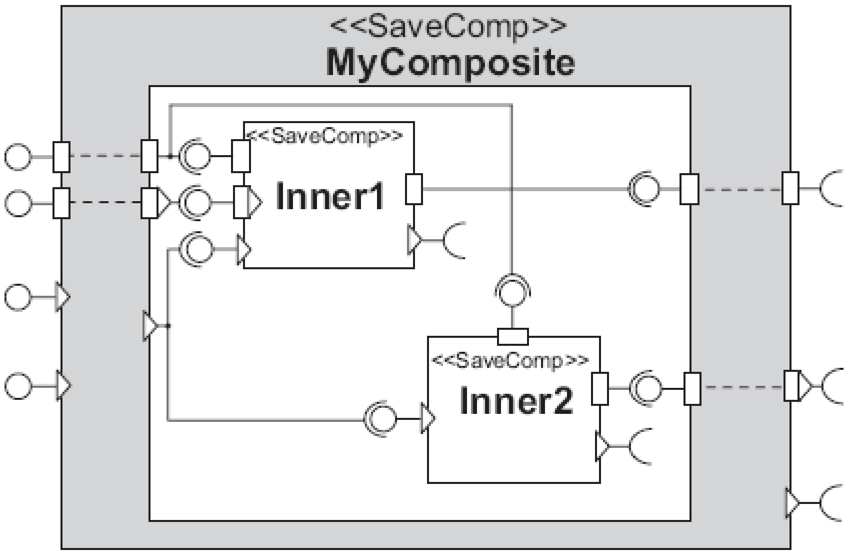

Figure 3.5: An example of a composite component viewed in the expanded mode [14].

3.4.4 Assemblies

Assemblies are elements that encapsulate sub-systems. Internal elements and connections that assembly consists of are hidden from the rest of the system and can be accessed only through ports of the assembly. Like switches, data and trigger signals are directly relayed from the assembly's ports to the ports of its sub-elements, and vice versa.

3.4.5 Ports

As mentioned, to define element's functional interface, input and output ports are used. Three different types of both input and output ports exist.

- **Data ports**. Data ports enable the exchange of data between elements. All data ports are typed and can contain an initial value. Data input ports also buffer the last value set to them. Data output port can only be connected to a data input port.
- **Trigger ports**. Trigger ports are used to build control flow of the system. Their function is to trigger components into execution. Trigger output port can be connected only to a trigger input port. When a trigger output port

sends the trigger signal to a trigger input port, the input port becomes *activated*, and remains in that state until the component is executed.

- **Combined (data and trigger) ports**. Combined ports have the function of both data and trigger ports. Combined output port can be connected to either data input port, trigger input port or combined input port.

With these port types, the data flow and control flow can be separated. The isolation of the control flow makes the system support both periodic and event-driven activities, and makes the temporal behavior of the system more suitable for analysis. On the other hand, the isolation of data flow enables the elements to exchange data without handling over control, which simplifies the creation of feedback loops or connection of parts of the system which run at different clock frequencies.

3.4.6 Component execution

A component starts its execution when it is *triggered*. For a component to become triggered, all its trigger input ports must be activated. Component execution is performed in three-phase *read-execute-write* semantics.

When the component is triggered it goes in the execution state and starts the *read* phase. In the read phase values of all the data input ports are stored internally in the component. This ensures the consistency of the computation. After the input values are stored, *execute* phase begins. In this phase component performs its computation. After the computation is finished output values are written to the component's data output ports in the *write* phase. In the end, all component's trigger output ports are activated and all its trigger input ports reset, and the component goes back to the idle state.

4 UComp – Combining CBD and UPnP

The drive for combining UPnP technology and CBD comes from the needs of both of them. UPnP defines an easy way for devices to discover and control each other on a managed or unmanaged networks. But there is no simple way for this devices to cooperate in such a way that they produce useful, non-trivial, functionality. To achieve that, control point would have to implement very complex algorithms, and even then the resulting system would almost never fully comply to the user requirements. Component based approach could help in building complex UPnP systems that are fully adapted to the user requirements. On the other hand, UPnP introduces a way for building distributed embedded systems using standard protocols and networks that can connect different platforms and types of devices, which is of great interest to CBD.

The *UComp* (UPnP Comp) component model is based on the SaveComp component model described in section 3.4. Some changes were made to conform the SaveCCM to the use of UPnP and to make the model more simple to use.

Domains in which the proposed architecture is useful is limited by the drawbacks of UPnP. Long response time from the embedded devices caused by the complex protocol and the possibility of “disappearance” of the devices (due to the possible loss of UDP discovery messages) makes it unsuitable for any hard real-time or safety-critical systems. With that in mind, the architecture is designed to allow as simple development as it can. Main goals of this component model are to enable development of non-critical embedded systems using UPnP devices in as simple as possible way and to explore the possibilities of using UPnP in component based development.

4.1 UComp architecture

The system 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 UPnP control point. The functionality of the system (Java application) is defined by the components it uses and connections between those components. Use of such centralized architecture has the downside of generating more network traffic and longer response time. For two devices to communicate, data needs to be sent from one device to the central application, which then forwards it to the other device, instead of just one device sending the data directly to the other. But this architecture also has its benefits.

- Data sent by one device can be processed by the application before it is forwarded to other components, 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.
- Modification of systems is much easier. System's behavior can be modified by simple changes in the interconnection of components (or by changing components themselves) in the central application. No changes in the behavior of the devices is needed. If devices were to communicate directly to one another, a way for changing their configuration at run-time would have to be devised. Such a functionality would mean that standard UPnP devices could not be used. Also, it would take up a portion of device's resources.

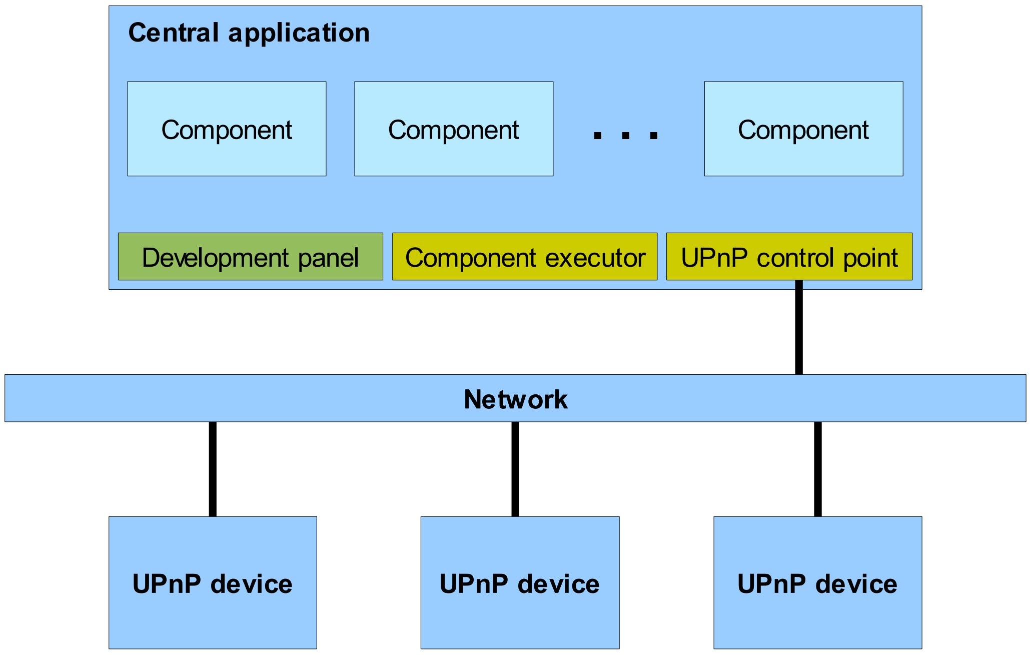

Figure 4.1: UComp architecture.

4.1.1 Framework and containers

A framework provides components with design- and run-time resources:

- *development panel* on which the components can draw themselves (this element is not necessary at run-time, but can be used to provide the overview of the system),
- *component executor* – a Java object which runs the thread for component execution and manages the execution queue,
- *UPnP control point*, which handles the communication with UPnP devices on the network.

Containers manage collections of components. They provide methods for providing visual (development) information, component initialization, drawing components and adding, removing and finding components. Containers also provide frameworks to components they contain. More detail of the *Framework* and *Container* Java interfaces are given in Figure 4.3.

4.1.2 Components

UComp defines three basic types of components: *UPnP components*, *software components* and *composite components*.

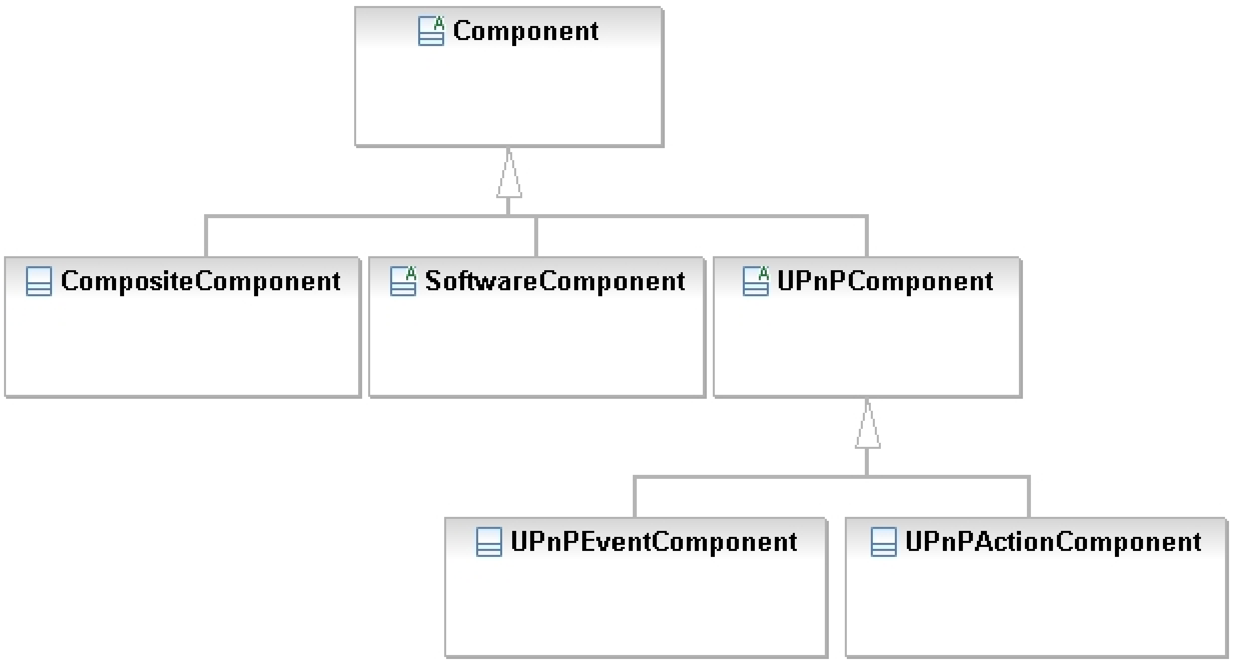

Figure 4.2: Class diagram for the component classes.

To make the browsing of components simpler they are arranged in groups. Every component defines a list of groups it belongs to. The list is viewed hierarchically: every group in the list treated like a sub-group of the group that precedes it in the list. Groups are defined by their name. They also include a description. An example of a group list would be (showing only the names of the groups): “Software component”, “Math”, “Simple”. This list defines a group “Simple” that is sub-group of the group “Math”, which is sub-group of the group “Software component”.

4.1.3 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* and *UPnP event components*. Action arguments and evented state variables are used for the interfaces. Although it would be more natural to view UPnP devices as components and their services as interfaces of these components, this view gives a component model that is more simple and easier to use.

Input and output ports of UPnP components are generated according to arguments of device's actions (for action components) or 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 the subscription to the events is accepted in case of event components.

UPnP action components

UPnP action components represent actions of UPnP devices. Every action component is bound to a specific device by its 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.

Input ports of action components are generated according to the arguments of the UPnP action it is bound to. Mapping between UPnP data types and port data types is given in Table 4.1. 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 arguments. The names of the ports are equal to the names of the arguments. Every UPnP action component also has an input port named “trigger” that accepts *Object* (any) data type. This port is used for additional triggering of the component enabling more complex triggering patterns and triggering of components whose actions don't have any input arguments.

When an 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, using the UPnP control point, a control message is sent to the device to invoke the action. In the end, output arguments of the action are parsed from the result message and their values used to set the values of the output ports.

UPnP event components

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

Ports of UPnP event components are generated using the state variable tables of UPnP services. For each evented state variable of the service event component is bound to, an output port is created with the same name that the state variable has. Same data type mapping as for action components (given in Table 4.1) is used. Event components have no input ports.

UPnP event components are active components, they don't need to be triggered to start execution. Instead, execution is started when the UPnP control point receives event notification from the service. New values of state variables are used to set the values of output ports of the component.

4.1.4 Software components

Software components are components whose functionality is fully implemented in Java code. They are not associated with any UPnP device. Role of software components is to process the data received from or sent to UPnP components, direct the execution of components (e.g. generation of periodical triggers), data flow control (using switches), definition of constants, etc.. Their function can vary from very simple (e.g. 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 for connecting the components and thus enabling fully visual development.

Distribution of software components

Software components are distributed as Java classes. This makes the distribution 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. Since software components have to be placed into Java package *hr.fer.rasip.upnp.ucomp.softwarecomponents* (or a sub-package of that package), any directory path that conforms to that package can be used. Software components must not be packed into Jar files because that makes them undetectable for the development tool.

4.1.5 Composite components

In current implementation, composite components serve just as containers for other components. They also provide a framework for the contained components.

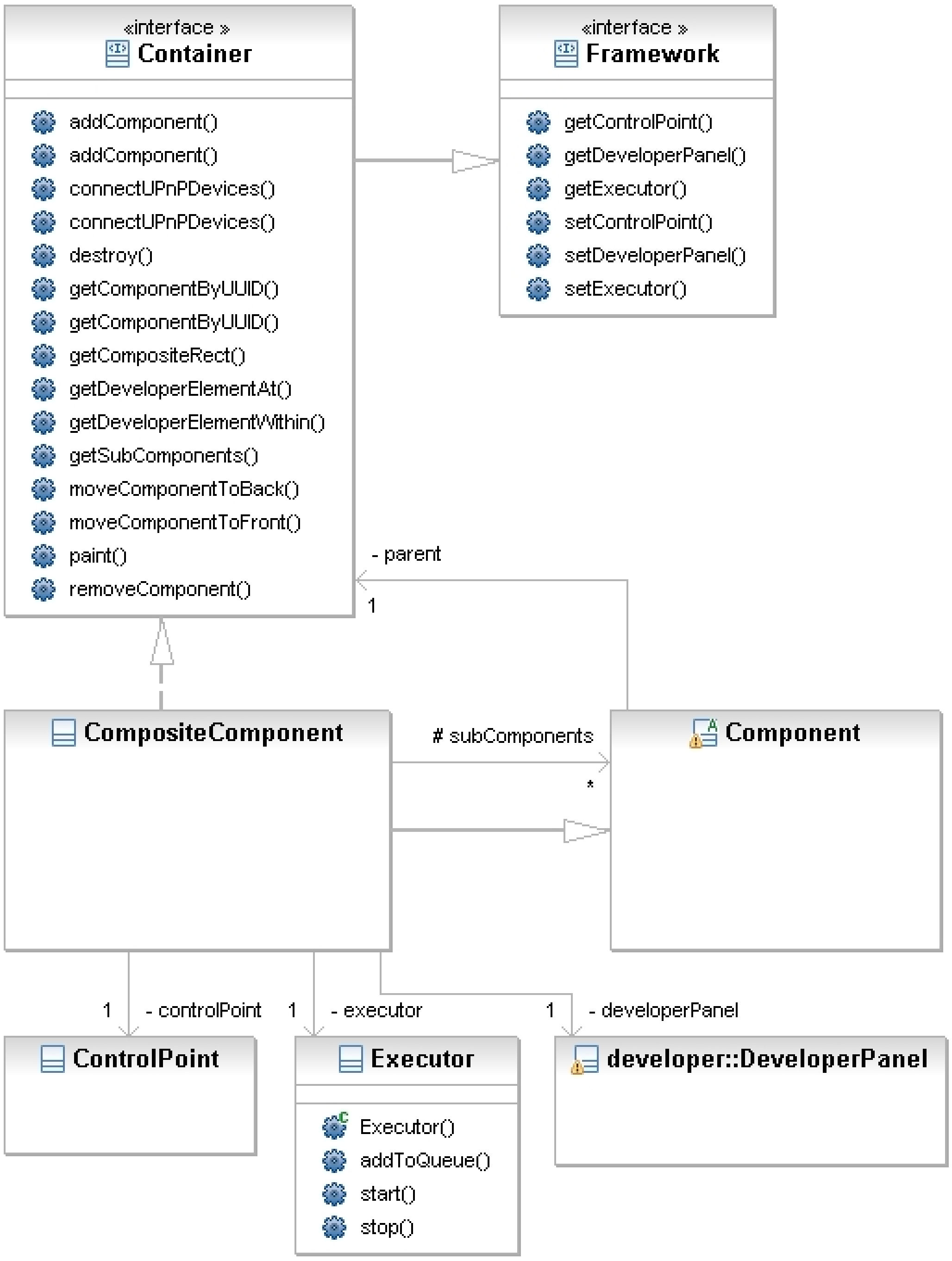

Figure 4.3: Class diagram describing relationships between framework, container, composite component and components.

4.1.6 Ports

Components exchange data and control each other through ports. Output ports send data and triggering signals, while input ports receive them. Ports are defined by their name and data type. All input and output port names of a component must be unique (although an input port can have the same name as an output port). Input ports also define their trigger type.

One output port can be connected to multiple input ports, but an input port can be connected to only one output port. This limitation is set to make the analysis of the model simpler. If there is a need for a input port to be connected to multiple output ports, it should by done by using switching software components. The actual connection of the ports is done by ports themselves: output ports keep a list of input ports that they are connected to, and input ports keep a reference to the connected 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. Both input and output ports buffer the last data that was set to them. Port's data can also be reset, making the port signal that there is no data available. Data is also transferred from an output port to an input port when a connection between the two is made. This provides better system behavior when the system is modified at run-time.

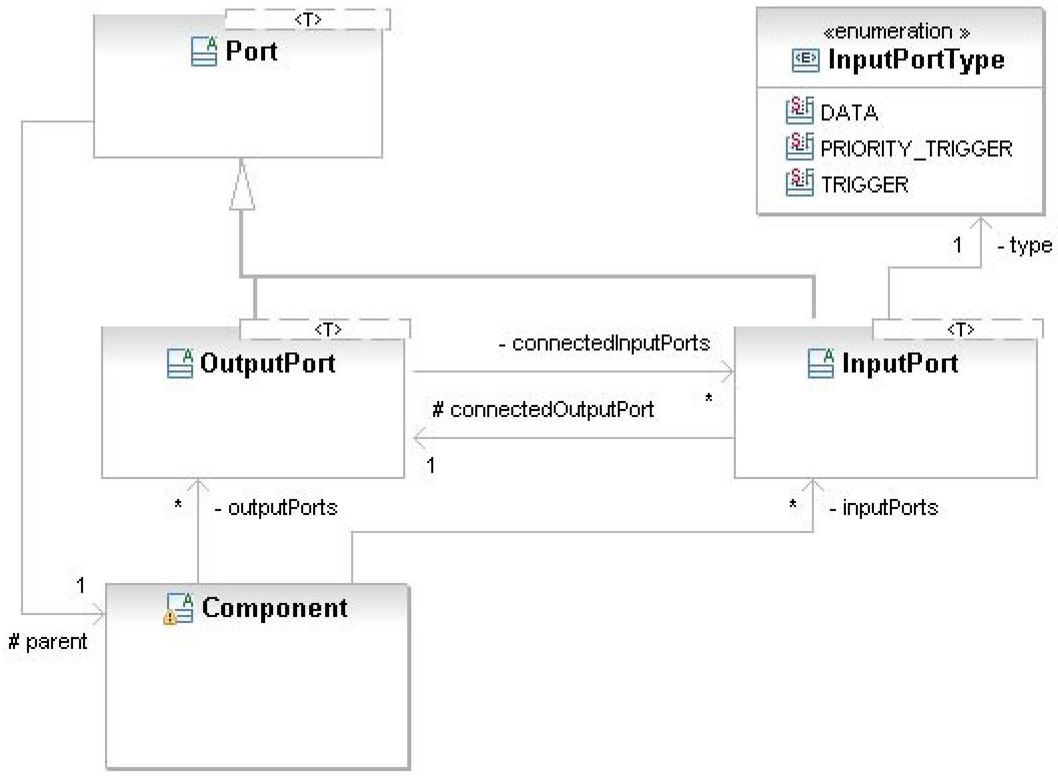

Figure 4.4: Class diagram describing relationships between components and ports.

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*, *String* and *Object*. These types are chosen to cover data types defined for UPnP arguments and state variables. An overview of port data types, mapping to UPnP data types and types of data accepted by the input ports is given in Table 4.1.

Table 4.1: Port data types, associated UPnP data types and types of data accepted by input ports of that type.

Port data type	UPnP data types	Accepted types for input ports
Boolean	boolean	**Boolean**
		Integer (*0* is evaluated to *false*, all other values to *true*)
Integer	ui1, ui2, ui4, i1, i2, i4, int	**Integer**
		Double (the value is rounded by *math.Round()* method)
		Boolean (*true* is evaluated to *1*, *false* to *0*)
Double	r4, r8, number, float	**Double**
		Integer (same value is used in format of a Double)
		Boolean (*true* is evaluated to *1.0*, *false* to *0.0*)
String	All other types.	**String**
		Boolean (toString() method is used)
		Integer (toString() method is used)
		Double (toString() method is used)
Object		**Object** (accepts any data type)

Although ports with Object data type accept any other data type, they are not used for exchanging data. Instead, they should only be used for triggering components. By setting the value of Object input port, the port is triggered and the data ignored. Output Object ports can only be connected to input object ports because they do not provide any data.

Trigger types

To be executed, a component needs to be *triggered*. Trigger signals are sent from an output port to all input ports it is connected to along with the new data when output port changes its data. All output ports send trigger signals. When an input port receives a trigger signal, it becomes active. There are three types of trigger 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*, its trigger state is ignored when checking if component is triggered. It is only used to receive data.

With combination of these three trigger types, complex triggering patterns or feedback-loops can be achieved.

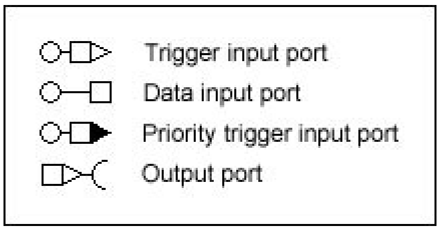

Figure 4.5: Graphical representation of ports.

4.1.7 Component execution

By looking at the way a component is executed, two types of components can be distinguished: *passive* and *active* components. Passive components execute only when they are triggered by other components, while active components may start their execution by an internal event.

Execution of passive components is done by a separate class, *Executor*. This class is provided to the components by their framework. Executor manages a queue for the components that need to be executed and runs a thread that does the actual execution of components. When a component is triggered, it adds itself to the queue of the Executor object. 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. Seeing that the Executor executes all components in the same thread, developer of new software components should start a new thread in the *execute* method if the method is expected to have a long processing time. After the component's *execute* method has finished, all component's input triggers are reset.

An example of an active component is UPnP event component. Its execution starts when a UPnP event notification is received by the component. Although this event is in fact external, it is viewed as internal by the component model because it wasn't generated by any interaction with other components, but in the component itself. Another example of active component would be a component that generates periodical triggers.

4.1.8 Development of new software components

New software components can be developed by extending the *SoftwareComponent* class or any of its subclasses. Extending a subclass has the benefit of putting new component in the same group as other components that have extended that class and inheriting common features (e.g. abstract class

SimpleMathComponent already defines Double input ports *a* and *b* and an Double output port *out*).

Constructor

For a software component to be valid, it has to implement a public constructor that takes *java.util.Properties* as an (only) argument. First line of the constructor must call the super-class constructor passing component name (String) and the received Properties object as arguments. That constructor should also add input ports of the component to the *inputPorts* Vector and output ports of the component to the *outputPorts* Vector. A sample of the constructor is given in Code 4.1.

```
public Substring(Properties properties) {
    super("Substring", properties);

    inPort = new StringInputPort("in", this);
    inPort.setType(InputPortType.PRIORITY_TRIGGER);
    inPort.setDescription("The input string.");
    inputPorts.add(inPort);

    startPort = new IntegerInputPort("start", this);
    startPort.setType(InputPortType.PRIORITY_TRIGGER);
    startPort.setDescription("Index of the starting character of the
substring in the <b>in</b> string.");
    inputPorts.add(startPort);

    endPort = new IntegerInputPort("end", this);
    endPort.setType(InputPortType.PRIORITY_TRIGGER);
    endPort.setDescription("Index of the last character of the substring in
the <b>in</b> string.");
    inputPorts.add(endPort);

    outPort = new StringOutputPort("out", this);
    outPort.setDescription("Sustring of the <b>in</b> string.");
    outputPorts.add(outPort);
}
```

Code 4.1: Sample constructor of an software component.

Component description

Developers should also override the *getDescription* method when developing new software components. The string that the method returns is used for showing tool-tip description to the user. Method *getPortsDescription* can be used to automatically add description of ports to the component description. Code 4.2 Shows an example of the override method.

```
public String getDescription() {
    StringBuilder sb = new StringBuilder("<html>Gets a substring of a
string." +
            "<br />Substring starting at <b>start</b> and ending at
<b>end</b>" +
            "<br />character of the string at the <b>in</b> port" +
            "<br /> is given at the <b>out</b> port.");
    sb.append(getPortsDescription());
    sb.append("</html>");
    return sb.toString();
}
```

Code 4.2: An example of the getDescription method override.

Adding custom properties dialog window

To add a custom properties dialog window to a component, first the dialog has to be created. It should extend *JDialog* class and be defined as the inner class of the component class. To enable showing the dialog, *hasPropertiesDialog* and *showPropertiesDialog* methods have to be overridden. The first method should just return true, and the second one should show the dialog and handle the results of that showing. Developer must also override the *getProperties* method to enable storing the properties set by the user to a file, and also modify constructor so that it reads the same properties from the *Properties* object passed to it. When overriding the *getProperties* method, developer must append new properties to the properties defined by the superclass. A sample of the overridden methods, and a sample of the constructor is given in Code 4.3.

```
public boolean hasPropertiesDialog() {
    return true;
}

public void showPropertiesDialog() {
    ConstantSoftwareComponent.PropertiesDialog dialog = new
PropertiesDialog();
    dialog.setValue(getValueString());
    dialog.setLocationByPlatform(true);
    dialog.setVisible(true);
    if (dialog.isResultOK()) {
        if (!setValueFromString(dialog.getValue(), true)) {
            // TODO do something...
        }
    }
}

public Properties getProperites() {h
    Properties properties = super.getProperites();
    properties.setProperty("value", getValueString());
    return properties;
}

protected ConstantSoftwareComponent(Properties properties) {
    super("Constant", properties);

    String value = properties.getProperty("value");
    if (value != null) {
        setValueFromString(value, false);
    } else {
        setInitialValue();
    }
}
```

Code 4.3: An example of the override of hasPropertiesDialog, showPropertiesDialog, and getProperties methods, and a sample of the constructor setting properties.

When distributing a software component that has a custom properties dialog window, it must be kept in mind that both components and dialogs class files are included.

Development of new port types

Any Java class can be used as port's data type. To create an input or output port using new data type, generic classes *InputPort*<> or *OutputPort*<> have to be extended. In both cases, the class that is to be used as port's data type needs to be used as the type parameter for the generic port class. Because the generic type in Java is erased at compile time, developer must set *data* member of the *Port* class *(this.data)* to a new instance of the class to be used for data in the constructor. Beside that, new output ports just have to call the superclass constructor with name of the port and parent component as arguments. New input ports also need to set the classes for the data types they can accept (by adding them to the *acceptableClasses Vector*) and override *castAndSetData* method. The *castAndSetData* method has to implement code for casting all acceptable data types to the data type of the port and return *true* if the cast was successful. Example of input and output ports is given in Code 4.4.

```
public class DoubleOutputPort extends OutputPort<Double> {

    public DoubleOutputPort(String name, Component parent) {
        super(name, parent);
        this.data = new Double(0);
    }
}

public class DoubleInputPort extends InputPort<Double> {

    public DoubleInputPort(String name, Component parent) {
        super(name, parent);
        acceptableClasses.add(Double.class);
        acceptableClasses.add(Integer.class);
        acceptableClasses.add(Boolean.class);
        this.data = new Double(0.0);
    }

    protected boolean castAndSetData(Object data) {
        if (data instanceof Double) {
            this.data = (Double) data;
            return true;
        } else if (data instanceof Integer) {
            this.data = ((Integer) data).doubleValue();
            return true;
        } else if (data instanceof Boolean) {
            if (((Boolean) data)) {
                this.data = 1.0;
            } else {
                this.data = 0.0;
            }
            return true;
        }
        return false;
    }
}
```

Code 4.4: An example of input and output port class.

4.2 *Visual development tool*

For building UComp systems, a visual development tool named *UComp Developer* has been created within this thesis. It enables browsing available components through a component tree, visual representation of components on a development panel, connecting that components, setting their properties and the properties of their ports, and starting and stopping the execution of the developed system. Run-time modification of the system is supported: while the system is running, components can be added or removed, connections between components altered and the properties of the components or their ports changed.

The tool is built on Java Swing framework. To run, it needs JRE 1.6.

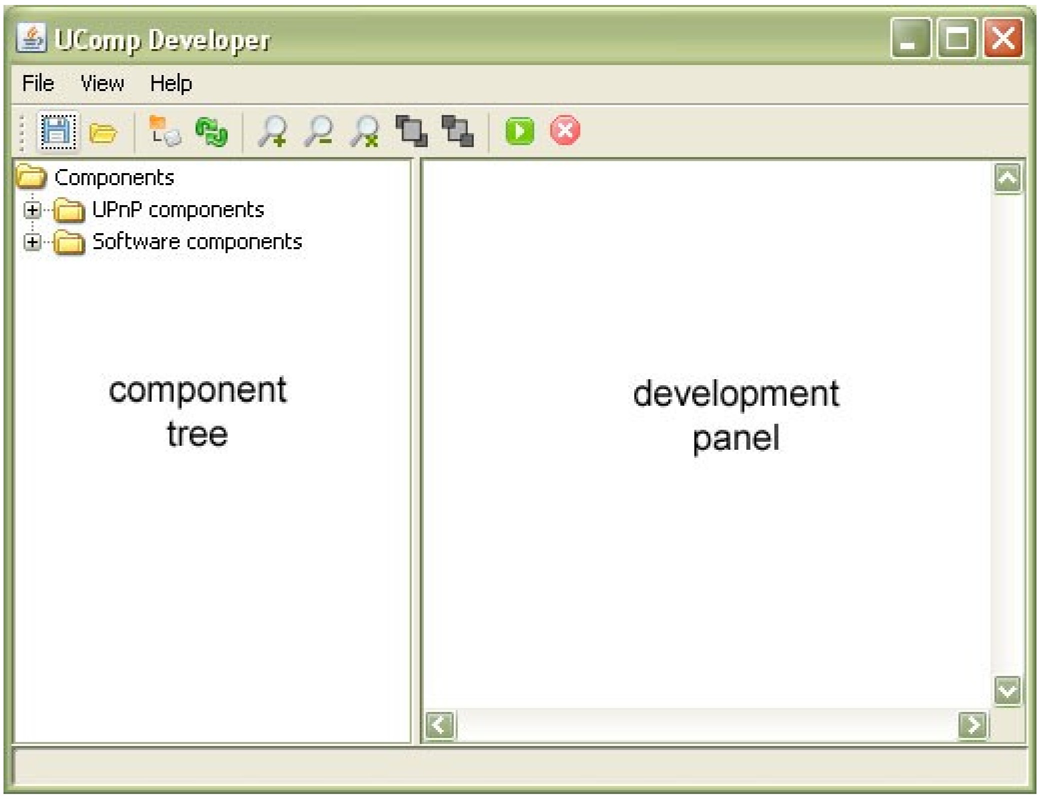

Figure 4.6: UComp Developer application.

4.2.1 Tool-bar

A screenshot of the applications tool-bar is given in Figure 4.7. From left to right, icons represent following actions:

- save system to a file,
- load system from a file,
- show or hide the component tree,
- refresh the list of UPnP components,
- zoom in the development panel,

- zoom out the development panel,
- zoom the development panel to actual size,
- bring selected component(s) to front,
- send selected component(s) to back,
- start the execution of the system,
- stop the execution of the system.

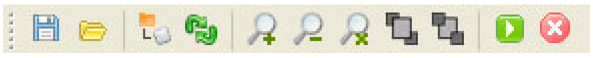
Figure 4.7: Tool-bar.

4.2.2 Component tree

Component tree is used to browse available UPnP and software components. Tree nods are organized by the component groups (described in section 4.1.2). It shows UPnP components currently available on the network and software components that were found while loading the application. The list of UPnP or software components can be refreshed (reloaded in case of software components) by right-clicking on the component tree and choosing either *refresh UPnP components* or *reload software components* menu item from the pop-up menu. By hovering the mouse pointer over a component or a component group a tool-tip with the description of the component or group is shown.

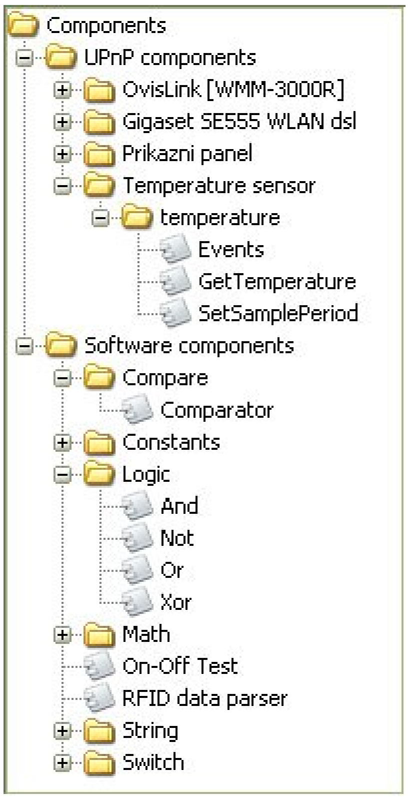

Figure 4.8: Component tree.

4.2.3 Development panel

Development panel enables fully visual development and overview of the system. Components that make up the system can be manipulated, connected or their properties changed using the mouse. It also provides information about components and ports (including their value during run-time) by tool-tips when mouse hovers over them.

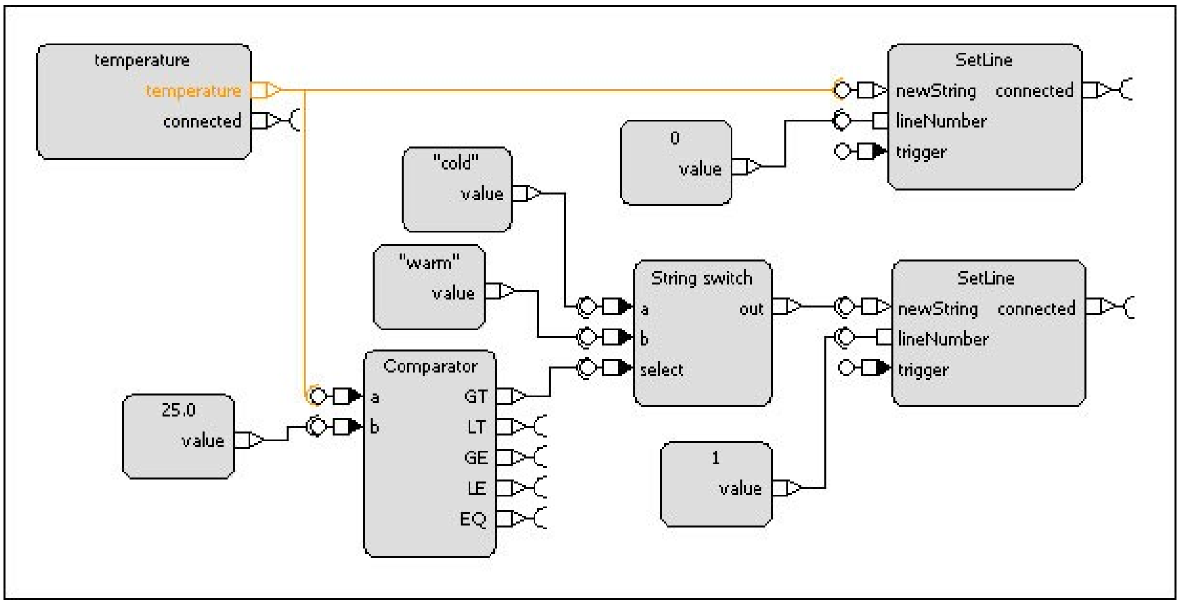

Figure 4.9: Development panel.

Adding components to the panel

Components are added to the panel by pressing down the left mouse button on the desired component in the component tree, dragging the mouse cursor to the development panel and releasing the mouse button. The component then appears in the development panel on the place where the button was released.

Component pop-up menu

By right-clicking on a component, the component pop-up menu appears. From that menu user can choose to bring the component forward or send it back in the panel (in case two components overlap), remove the component from the panel or show the component's properties window if the component has one.

Port connections

Two ports (input and output) can be connected by pressing down the left mouse button over one of them, dragging the mouse cursor to the other and releasing the mouse button. Input and output ports can only be connected if the input port can accept the data type the output port offers. In that case, when one port is selected and mouse is dragged over the other, line between the ports becomes orange and snaps on that ports.

Connections between ports cannot be selected directly. Instead, they are selected by selecting the ports they connect. By selecting an output port all connections that start at that output port are selected. By selecting an input port the connection that ends at that port is selected.

Connections are removed by either right-clicking an output port and clicking on the *remove all connections* menu item (thus removing all connections that start at that port), or by right-clicking on an input port and clicking on the *remove connection* menu item (which results in removing only the connection that ends at that port).

Changing trigger type of an input port

Trigger type of an input port is changed by right-clicking on that port and clicking on the *properties* menu item in the pop-up menu. Then, a dialog window appears showing radio buttons for selecting the trigger type. Select the desired type and click on the O*K* button.

4.3 UComp file format

Once developed, system configuration can be stored to an XML file. Only static information about the system is stored, current values of ports or information about the state of components is not stored.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.fer.hr/rasip/UComp"
elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:UComp="http://www.fer.hr/rasip/UComp">
 <element name="compositeComponent">
  <complexType>
   <sequence>
    <element ref="UComp:componentList"></element>
    <element ref="UComp:connectionList"></element>
   </sequence>
  </complexType>
 </element>
 <element name="componentList">
  <complexType>
   <sequence>
    <element ref="UComp:component"></element>
   </sequence>
  </complexType>
 </element>
 <element name="connectionList">
  <complexType>
   <sequence>
    <element ref="UComp:connection" maxOccurs="unbounded"
minOccurs="0"></element>
   </sequence>
  </complexType>
 </element>
 <element name="component">
  <complexType>
   <sequence>
    <element ref="UComp:developerInfo" maxOccurs="1"
minOccurs="1"></element>
```

```
    <element ref="UComp:inputPortList" maxOccurs="1"
minOccurs="0"></element>
    <element ref="UComp:outputPortList" maxOccurs="1"
minOccurs="0"></element>
    <element ref="UComp:propertyList" maxOccurs="1"
minOccurs="0"></element>
   </sequence>
   <attribute name="class" type="string"></attribute>
   <attribute name="name" type="string"></attribute>
   <attribute name="uuid" type="string"></attribute>
  </complexType>
 </element>
 <element name="developerInfo">
  <complexType>
   <attribute name="x" type="unsignedInt"></attribute>
   <attribute name="y" type="unsignedInt"></attribute>
  </complexType>
 </element>
 <element name="inputPortList">
  <complexType>
   <sequence>
    <element ref="UComp:inputPort" maxOccurs="unbounded"
minOccurs="0"></element>
   </sequence>
  </complexType>
 </element>
 <element name="outputPortList">
  <complexType>
   <sequence>
    <element ref="UComp:outputPort" maxOccurs="unbounded"
minOccurs="0"></element>
   </sequence>
  </complexType>
 </element>
 <element name="propertyList">
  <complexType>
   <sequence>
    <element ref="UComp:property" maxOccurs="unbounded"
minOccurs="0"></element>
   </sequence>
  </complexType>
 </element>
 <element name="inputPort">
  <complexType>
   <attribute name="class" type="string"></attribute>
   <attribute name="name" type="string"></attribute>
   <attribute name="type">
    <simpleType>
     <restriction base="string">
      <enumeration value="DATA"></enumeration>
      <enumeration value="TRIGGER"></enumeration>
      <enumeration value="PRIORITY_TRIGGER"></enumeration>
     </restriction>
    </simpleType>
   </attribute>
  </complexType>
 </element>
 <element name="outputPort">
  <complexType>
   <attribute name="class" type="string"></attribute>
   <attribute name="name" type="string"></attribute>
  </complexType>
 </element>
 <element name="property">
  <complexType>
```

```
   <attribute name="key" type="string"></attribute>
   <attribute name="value" type="string"></attribute>
  </complexType>
 </element>
 <element name="connection">
  <complexType>
   <sequence>
    <element ref="UComp:source" maxOccurs="1" minOccurs="1"></element>
    <element ref="UComp:target" maxOccurs="1" minOccurs="1"></element>
   </sequence>
  </complexType>
 </element>
 <element name="source">
  <complexType>
   <attribute name="componentUUID" type="string"></attribute>
   <attribute name="portName" type="string"></attribute>
  </complexType>
 </element>
 <element name="target">
  <complexType>
   <attribute name="componentUUID" type="string"></attribute>
   <attribute name="portName" type="string"></attribute>
  </complexType>
 </element>
</schema>
```

Code 4.5: XML Schema for UComp file.

4.4 Mapping between UComp and SaveCCM

As already mentioned, UComp component model is based on SaveCCM. There are two main reasons for this. First, SaveCCM has proved itself successful in modeling embedded systems. Second reason is that by keeping UComp model similar to SaveCCM, transformation between this two models is easier. Transformation between models may appear very useful because the two models can be used in separate stage of development. SaveCCM provides a lot of possibilities for testing the system before it is deployed. On the other hand, UComp provides a simple way for implementation of distributed systems.

4.4.1 Main differences between models

When designing UComp the aim was to generate a model and architecture that is very simple to use. Its main focus is to enable implementation of the systems. For that reason some differences between UComp and SaveCCM emerge.

Priority trigger

To simplify the development of systems, UComp introduces priority trigger which is not present in SaveCCM. There is no direct way to map priority triggers to SaveCCM. One of the solution would be to use SaveCCM switch elements to create an *or* operation on all connections that lead to priority trigger port in

UComp, and connect the output of the switch to all trigger ports of the SaveCCM component.

SaveCCM switch

UComp does not define separate switch elements. Instead, switches are defined through software components. In case of SaveCCM to UComp transformation, switches should be transformed into software components (possibly extending *SwitchSoftwareComponent* class). For the transformation in other direction there are two possibilities: (1) treat switch components as all other software components, and (2) try to detect if a software component extends the *SwitchSoftwareComponent* class and transform it to a SaveCCM switch element.

Complex connections

In SaveCCM, connection between components can be defined to have complex behavior. To keep development of systems simple, UComp defines no such connection. In transformation between models, only normal SaveCCM connection should be used.

Components attributes

UComp does not provide an ability to define non-functional (or semantical) attributes of the component in the way SaveCCM does. If a transformation from UComp to SaveCCM is made, this attributes should be determined from the tests on the implemented components.

Assemblies and composite components

No means of component hierarchy is currently implemented in UComp (composite components serve only as containers for components). At the time, it is not possible to transform SaveCCM assemblies and composite components into UComp.

4.5 Testing results

The UComp architecture was tested with several, both simple and complex, system configurations. A screenshot of one of them is given in Figure 4.9. For the purpose of testing UPnP devices running on personal computers and RCM2200 network microcontrolers were used.

4.5.1 Development process

The combination of component-based approach (building systems from preexisting components) and UPnP technology (enabling control, event notification and description of available devices over the standard computer network) resulted in very efficient development of distributed embedded systems. The ability of fully visual development through the *UComp Developer* tool adds to the simplicity of development. One of the possibilities for further improvement of this process by extending the description of UPnP devices is given in section 4.6.1.

4.5.2 False expiration of UPnP devices

Due to use of unreliable UDP protocol in UPnP discovery, a control point may miss a notification of device's presence on the network and mark the device as expired, making the device invisible to the system. Much attention was given to this problem while testing. Rare occurrences of this problem have been detected. On all occasions the connection to the device was reestablished with the next notification message from the device. Nevertheless, this remains one of the biggest problems because it makes the system unreliable. Section 4.6.5 gives a proposal on how to try to solve this problem.

4.5.3 Component execution time

During testing, the execution time of components was measured. The data collected has shown that almost all of the system's execution time relates to UPnP action components. Execution time of software components was constantly under 1 ms. Average execution time of UPnP action components was between 150 and 350 ms. In some rare occasions, the action invocation took several seconds. The cause of such long execution time is the low processing power of the embedded devices and the complexity of the UPnP control protocol. One workaround for this problem is given in section 4.6.4.

4.5.4 Serial vs. parallel execution of UPnP components

The system was tested both with (parallel execution) and without (serial execution) executing UPnP action components in separate threads. When the components were executed (UPnP actions invoked) in the same thread as all other components, the system suffered from delays while waiting for devices to respond to action requests. In that time no other passive components were able to start their execution. On the other hand, when testing the system with starting a new thread for execution of each UPnP action component problems arose when the system contained multiple UPnP action components that were bound to the same device. In that case some devices received multiple action invocations at the same time, and sometimes were not able to process all of them. This problem comes from low processing power of the RCM2200 microcontrolers on which the devices were built and their inability of preemptive multitasking. Although slower, serial execution of UPnP action components was chosen as a better solution at this moment, because loss of information that parallel execution exhibits is unacceptable.

4.6 Possibilities for improvement

In this sections a few possibilities for future improvement of UComp are given. While considering this improvements, it was always kept in mind that the changes made to the UPnP devices or control points would still leave them compatible with standard UPnP architecture.

4.6.1 Adding semantic information to UPnP device description

Due to use of XML in UPnP device description, and the instructions of UPnP forum for the control points to ignore any unknown elements or attributes while reading it, the description could easily be extended to include semantic and extra-functional information about the device and its services and actions. Such information would greatly improve the development process.

4.6.2 Implementing component hierarchy

Development could be made easier by allowing the composite components to be used as parts of the system just like UPnP and software components. Another possibility would be for composite components to implement UPnP device stack, thereby becoming UPnP devices themselves. Then they could be included as UPnP components in other systems.

4.6.3 Storing UPnP device descriptions

Current implementation of *UComp Developer* tool only allows building systems using only UPnP devices available on the network at the moment of development (with exception of the devices that were added to system before they were removed from the network). It would be very useful to add the ability to store device description and make the devices available for development even when they are disconnected from the network.

4.6.4 Changing the control protocol

Use of HTTP and SOAP protocols in UPnP control protocol makes the action invocation very time-consuming, which greatly contributes to reduced responsiveness of the UComp systems. The overhead of these two protocols most often exceeds the useful data by several times. The characteristics of the architecture could be enhanced by using a simpler control protocol that is more suited to the embedded devices. This protocol could be used in parallel with the standard UPnP control protocol. Devices that would implement this protocol could respond to action invocation faster, while standard UPnP devices would be controlled by standard UPnP control protocol.

4.6.5 Testing the devices that are about to expire

A way to compensate for the unreliability of the UDP protocol which is used in UPnP discovery, and thus make the UComp architecture more reliable, would be to check for existences of the devices that are about to expire using TCP protocol. One way this could be done would be to define a “ping” service, with only one action that takes no arguments, which devices could implement. UPnP control point stack could then be modified to try to invoke that action for devices implementing the “ping” service that are about to expire. If the device responds to the invocation (and thereby confirms that it is still connected to the network) the control point would postpone the device expiration. In this way both the device and the control point would still be fully compatible with the UPnP standard.

5 Conclusion

Component model that was created within this thesis demonstrates how the development of distributed embedded systems can be improved by combining the component-based approach with UPnP technology. Implementation of the model manifests some notable problems that arise from fully adhering to UPnP standard. These problem make the architecture unsuitable for application in safety-critical or hard real time domains. Many of them can be resolved by simple upgrades to the UPnP protocol. The model also shows plenty of possibilities for improvement which could lead to a more efficient development process and more robust, reliable and efficient systems.

6 Bibliography

[1] UPnP Forum, UPnP Device Architecture 1.0, 2003

[2] UPnP Forum, http://www.upnp.org

[3] Cybergarage, http://www.cybergarage.org/net/upnp/java/index.html

[4] Intel Software for UPnP technology, http://www.intel.com/software/upnp

[5] Satoshi Konno, CyberLink for Java Programming Guide, 2005

[6] M. Vitulić, Sustav za prikupljanje podataka temeljen na mrežoj arhitekturi UPnP, thesis no. 1566, Faculty of Electrical Engineering and Computing, University of Zagreb, 2007

[7] Universal Plug and Play, Wikipedia, http://en.wikipedia.org/wiki/Upnp

[8] I. Crnković, M. Larsson, Building Reliable Component-Based Software Systems, Artech House, 2002

[9] Component-Based Design and Integration Platforms, http://www.artist-embedded.org/, 2003

[10] I. Crnković, S. Larsson, M. Chaudron, Component-based Development Process and Component Lifecycle, International Conference on Software Engineering Advances, 2006

[11] I. Crnković, B. Hnich, T. Jonsson, T. Kiziltan, Specification, Integration and Deployment of Components, Communications of the ACM (CACM), vol 45, nr 10, Association for Computing (ACM), October, 2002

[12] A. Beugnard, Jean-Marc Jezequel, N. Plouzeau, D. Watkins, Making components contract aware, IEEE Software, 1999

[13] Component-based software engineering, Wikipedia, http://en.wikipedia.org/wiki/Component-based_software_engineering

[14] Håkansson J.: The SaveCCM Language Reference Manual

7 Abstract / Sažetak

This thesis explores the possibility of combining component-based approach to development with UPnP technology to enable an easy and efficient way for building distributed embedded systems.

First, the basics of UPnP technology and component-based development are described. Then a description of a simple component model (created within the thesis) that enables development of distributed embedded systems using UPnP technology is given. A visual development tool created for that component model is also described. In the end, results of testing of systems built with the model, along with some suggestions for future improvement are given.

Komponentni razvoj sustava od programskih i sklopovskih komponenti

Ovaj rad istražuje mogućnost povezivanja razvoja temeljenog na komponentama i UPnP tehnologije kako bi se omogućio jednostavan i efikasan način izgradnje distribuiranih ugrađenih sustava.

Prvo je dan pregled osnova UPnP tehnologije i razvoja temeljenog na komponentama. Tada je opisan jednostavan komponentni model (izrađen u okviru ovoga rada) namjenjen razvoju distribuiranih ugrađenih sustava koristeći UPnP tehnologiju. Dan je i opis grafičkog alata za razvoj sustava prema tom modelu (također izrađenog u okviru ovoga rada). Na kraju su dani rezultati testiranja sustava stvorenih pomoću izrađenog modela zajedno sa prijedlozima za buduća poboljšanja.

8 List of abbreviations

CBD	Component-based Development
CBSE	Component-based Software Engineering
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
GENA	General Event Notification Architecture
HTTP	Hypertext Transfer Protocol
HTTPMU	HTTP over UDP (multicast)
HTTPU	HTTP over UDP
JRE	Java Run-time Environment
SOAP	Simple Object Access Protocol
SSDP	Simple Service Discovery Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URL	Uniform Resource Locator
VPN	Virtual Private Network
XML	Extensible Markup Language