Mario Žagar
Faculty of Electrical Engineering and Computing
University of Zagreb, Zagreb, Croatia
mario.zagar@fer.hr

Ivica Crnković
Mälardalen Real-Time Research Centre
Mälerdalen University, Västerås, Sweden
ivica.crnkovic@mdh.se

Darko Stipaničev
Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split, Split, Croatia
darko.stipanicev@fesb.hr

Maja Štula
Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split, Split, Croatia
maja.stula@fesb.hr

Juraj Feljan
Mälardalen Real-Time Research Centre
Mälerdalen University, Västerås, Sweden
juraj.feljan@mdh.se

Luka Lednicki
Faculty of Electrical Engineering and Computing
University of Zagreb, Zagreb, Croatia
luka.lednicki@fer.hr

Josip Maras
Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split, Split, Croatia
josip.maras@fesb.hr

Ana Petričić
Faculty of Electrical Engineering and Computing
University of Zagreb, Zagreb, Croatia
ana.petricic@fer.hr

DICES: Distributed Component-based Embedded Software Systems

Abstract

This article gives a short overview of the contribution of DICES project. The goal of the project is to advance theories and technologies used in development of distributed embedded systems. Three examples of the contributions are presented: a) reverse engineering of web-based applications, design extraction and extraction of reusable user-interface controls, b) a fretwork for building systems that use UPnP devices and treat them as components in the same way as software components, and c) PRIDE – development environment for designing, modeling and developing embedded systems, based on ProCom technology.

Keywords: Software components, embedded systems, web-based applications, reverse engineering

1. Introduction

DICES (Distributed Component-based Embedded Software Systems) is a project funded by UKF (Unity Through Knowledge Fund) with contributions from Faculty of Electrical Engineering and Computing (FER) - University of Zagreb, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture (FESB), University of Split, School of Innovation, Design and Engineering - Mälardalen University (MDU) in Sweden, and Ericsson Nikola Tesla. The project was performed during three years, from 2007 to 2011.

The DICES project goal is to advance development of distributed embedded software systems with emphasis on software reusability and predictability of software quality. The aim of the project is increasing the software development efficiency and quality by applying service-oriented and component-based approaches.

The overall presence of distributed embedded systems in the modern society is a fact. Examples of such systems are telecommunication systems, grid systems, control and information systems of vehicular systems (cars, trains), environmental monitoring systems. Embedded systems development is one of the strategic research areas of EU-FP7 programmes. It is also of significant importance in Croatia, since many leading companies in Croatia either produce such systems (e.g. Končar, Ericsson Nikola Tesla) or use such systems (e.g. Pliva, or many small companies).

DICES is addressing efficient reusability of software components and prediction of the important properties for embedded systems: resource utilization, and performance, by applying the service-oriented software engineering and component-based software engineering methods and technologies.

The main contribution of DICES is provided in two directions: a) analysis of legacy web-based systems, extraction of its design and its automatic modeling, and extraction of user interface controls and packaging them as reusable units, and development of component-based technology appropriate for design of embedded systems.

The reverse engineering, architecture recovery and extraction of reusable UI units is applied on “iForestFire - Intelligent Forest Fire System” system developed at FESB Split, which enables thorough validation of the approach and provides input for further development of this system and possible commercialization of the improved product.

Development of component-based technology is done in cooperation with Swedish Strategic Research Centre, PROGRESS that has developed theories and methods for modeling component-based embedded systems. DICES contribution was in development of PRIDE, PROGRESS Integrated Development Environment, a tool modeling component-based embedded systems, simulation and analysis of resource utilization.

In this article we give a short overview of these contributions as follows. In section 2 we describe Componentizing Web Information Systems, including phpModeller, a tool for design extraction and its presentation in UML. Section 3 shows a component model that includes both, software and hardware components based on UPnP devices and enables a unique treatment for software and hardware components. Section 4 describes PRIDE.

2. Analysis and Componenitization of Web-based systems

Web-based systems are systems that use the Internet as its core infrastructure and are accessible from anywhere in the world via the Web. Often these systems extensively communicate with databases and third-party information sources. In some cases they even communicate with embedded devices such as cameras or meteo-stations in order to provide real-time information about the environment. One of these systems is the iForestFire system developed at the University of Split, Croatia. **iForestFire** is an intelligent and integrated video based monitoring system for early detection of forest fire. The main idea is that forest fires are detected in incipient stage using advanced image processing and image analyses methods. The system is based on field units and central processing unit. The field units are embedded devices and include day & night, pan/tilt/zoom controlled IP based video cameras and IP based mini meteorological stations connected by wired or wireless LAN to the central processing unit where all analysis, calculation, presentation, image and data archiving is done. In the scope of the DICES project, iForestFire was chosen as a case study application in order to test the ideas of componentization of Web Information Systems that extensively communicate with embedded devices.

Since one of the goals of the DICES project was the analysis of legacy web-based systems, extraction of its design and its automatic modeling, extraction of User Interface controls and packaging them as reusable units in this section, we will describe two approaches that we have developed in order to achieve those goals: phpModeler – an approach for reverse engineering of legacy web applications, and Firecrow, an approach for extracting reusable Web User Interface controls.

2.1. PhpModeler – Reverse Engineering Legacy Web Applications

In order to componentize web information systems, first a correct assessment of the current state is necessary. Web information systems, especially ones that are communicating with complex embedded devices can be hard to understand because a single behavior can be realized by code executed on a large number of nodes (embedded devices, web servers, clients, etc.). Also, there usually exist many inter-dependencies between certain parts of the system that are hard to track and manage manually, and naturally the complexity of these inter-dependencies grows with the size of the web application. In order to tackle this problem and to improve development and maintenance efficiency, these applications need to be modeled on a high level of abstraction. One of the ways to achieve this goal is to use a process called Reverse Engineering (RE).

Reverse Engineering (RE) is used for information extraction from source artifacts (primarily source code) and their transformation to easily understandable abstract representations (e.g. standard UML diagrams). Even though there exists a certain number of already available tools for reverse engineering Web Information Systems (WARE [6], WebUml [3], ReWeb [10], etc.), none of them have the following functionalities:

- Static analysis of web application source code which as an end result produces UML diagrams that can be used for architecture recovery
- Generates dependency models that show inter-dependence between web application elements
- Visualizes Web application evolution.

In order to tackle this problem we have developed a plugin for the Eclipse IDE (Integrated Development Environment) that facilitates modeling and web application architecture recovery called *phpModeler.* It has three main features: page modeling, dependency modeling and model comparison.

In order to be able to build page UML models, the information important for those models has to be extracted. *phpModeler* gathers this information by parsing PHP, HTML, and JavaScript code responsible for the behavior of the web application. Once this information has been collected further analysis techniques can be used to establish dependencies between parts of the system. For every web page entity (JavaScript library, database table, file, PHP library, web page), the analysis finds all other entities dependent on it. For example, for every database table this means locating all web pages that access them and for function libraries all web pages that use those functions. Naturally, all information gathered in the analysis process can be generated to UML diagrams.

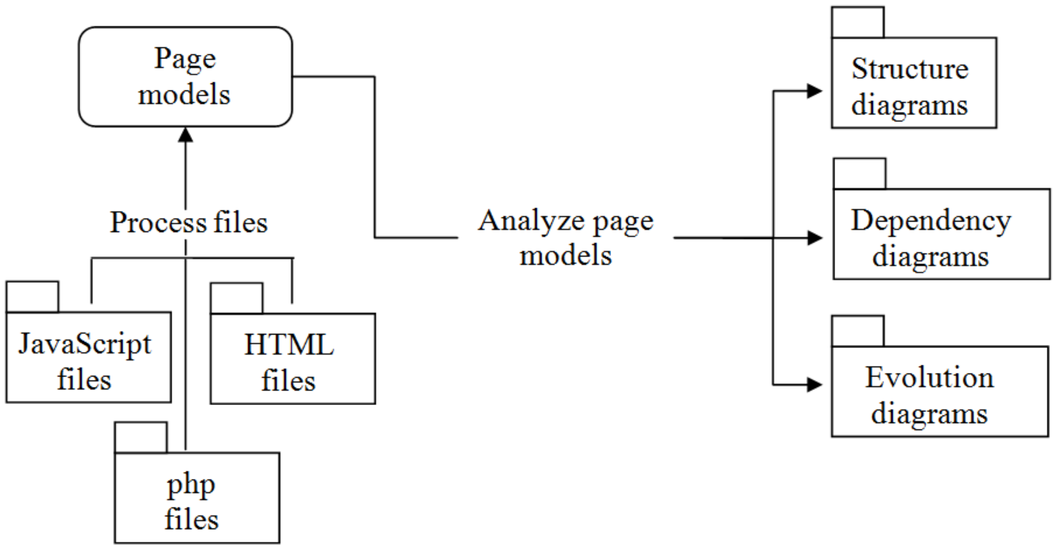


Figure 1: Reverse Engineering Web applications with phpModeler.

Web information systems evolve with the addition of new functionalities. This usually means that the existing code is modified to support the new user requirements. At different steps of the WIS evolution the system is represented with different models, so *phpModeler* provides a way to track the differences between two existing models. This gives a simpler, and more visual representation of the evolution of the target system. The whole process is shown in Figure 1.

We used *phpModeler* in the reverse engineering process of the *iForestFire* system in order to recover the architecture and gain better understanding of the system.

2.2. Firecrow – reusing Web User Interface controls

Important part of the web application code is the code that is used to realize the user-interface of the web application. Web application user-interface (UI) is often composed of distinctive UI elements, the so called UI controls. Similar controls are often used in different parts of web applications (and even in different web applications) and facilitating their reuse could lead to faster development. Unfortunately, preparing code for reuse is a slow process which is often not a priority.

Often, when developers encounter problems that have already been solved in the past, rather than re-inventing the wheel, or spending time componentizing, they reuse the code that is already functioning in another context [4]. Reuse tasks are complex and prone to errors, primarily because it is difficult to establish the minimum amount of code responsible for implementing the desired behavior [8].

In the web application domain, reuse is especially difficult: there is no trivial mapping between source code and the page displayed in the browser, responsible code is often scattered between several files and is found in between code that is not important from the reuse perspective. Next, the developer has to locate and download the necessary resources, source code, and adjust for the changes so that the component can be easily integrated into an already existing system.

The structure of a web page is defined by HTML code, the presentation by CSS (Cascading Style Sheets) code, and the behavior by JavaScript code. In addition, a web page usually contains various resources such as images or fonts. The interplay of these four basic elements produces the end result displayed in the user's web browser. Visually and behaviorally a web page can be viewed as a collection of UI controls, where each control is defined by a combination of HTML, CSS, JavaScript and resources (images, videos, fonts, etc.) that are intermixed with code and resources defining other parts of the web page.

In order to reuse a web UI control, we have to extract all that is necessary for the control to be visually and functionally autonomous. This means extracting all HTML, CSS, JavaScript and resources that are used in the visual presentation and the desired behavior of the control. The process can be separated into three phases: 1) Interaction recording, 2) Resource extraction, and 3) UI control reuse (Figure 2).

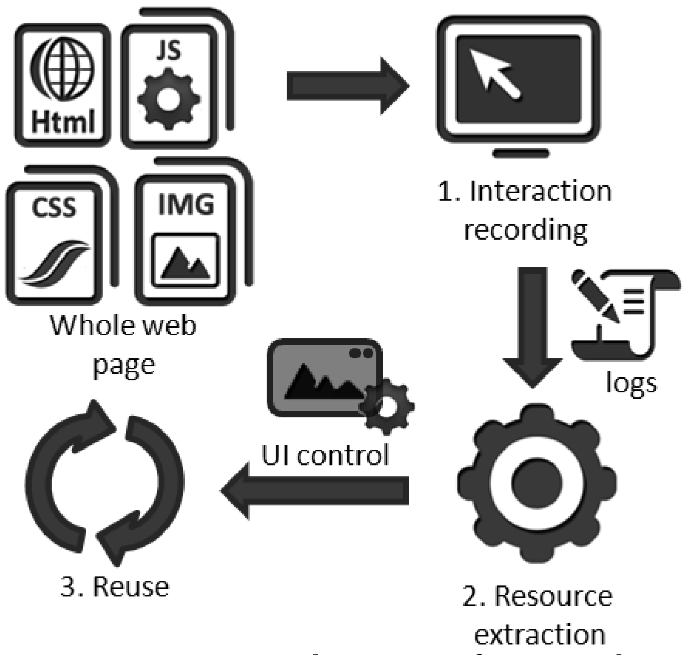


Figure 2: Extracting Web User-Interface controls.

The first step of the *Interaction recording* phase is to select the HTML node that defines the chosen UI control. Next, the user performs a series of interactions that represent the behavior of the control. The purpose of this phase is to gather a log of all resources required for replicating visual and behavioral aspects of the control. This is done by logging all executed code, all CSS styles and all resources used in the lifecycle of the control.

When the user chooses to end the recording, the process enters the *Resource extraction* phase, where code models for all code files (HTML, CSS, and JavaScript) are built. Based on those models and logs gathered during the recording phase, the code necessary for replicating the visuals and the demonstrated behavior is extracted.

After the extraction phase is completed, the user can enter the *Reuse phase* and automatically integrate the extracted control in an already existing web page, either by replacing, or by embedding it inside an existing node. With this, a full cycle is completed: from seeing the potential for reuse, through control extraction, all the way to actual reuse and gaining new functionalities on the target web application.

The whole process is currently supported by the Firecrow tool, which is an extension for the Firebug web debugger. Currently, the tool can be used from the Firefox web browser, but it can be ported to any other web browser that provides communication with a JavaScript debugger, and a DOM explorer (e.g. IE, Chrome, Opera). The *interaction recording* phase is the only part of the process that is browser dependent; building source models, extracting code, downloading resources, merging code and resources are all functionalities that are all encapsulated in a library that can be called from any browser on any operating system. The whole source of the program can be downloaded from [18].

3. Hardware and Software Components

The main purpose of embedded systems is to monitor or control processes in their environment. This interaction of software with the real world requires integration of software components with hardware components such as sensors and actuators. However, coping with these two different parts of embedded systems simultaneously is still a challenge. Current component models for embedded systems mostly focus on software components and rarely try to provide extensive support hardware component [7]. In DICES project we have also addressed this aspect of component-based software development. One of the results of our research is UComp component model and supporting technology [9].

3.1. UComp – component-based development for hardware and software components

The focus of UComp is on distributed systems whose functionality is implemented using various devices connected to a computer network. These devices may either be physical, i.e. realized using hardware, or virtual, i.e. realized using software applications. Main goal of UComp is to utilise the component-based approach and manage the hardware and software components in a uniform way. Further, our goal is to apply the component-based approach during whole life-cycle of a system, including run-time phase. By having the components available at run-time, systems can be extremely flexible because any modifications can be done while the system is running and the embedded devices are deployed. In addition, this would allow easier late deployment of new devices (i.e. components) or replacement of existing ones during run-time.

3.1.1. The UComp component model

UComp distinguishes between two types of components: *Device components* and *software components*. Device components are used to communicate with hardware or virtual network devices and provide their functionalities in a component-based manner. Functionality of software components is fully implemented in program code, and these components are not associated with any devices. They are used for additional computations in order to avoid the need for “glue code”.

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

When an input port receives a signal from the output port it is connected to, it becomes active. A component can be configured to start it's execution when either all or just selected ports are activated. This is done by selecting *activation* types for ports. There are three activation types for input ports:

- Trigger. A component is activated if *all* input ports with activation type set to *trigger* are active.
- Priority trigger. A component is activated if *any* of its input port with activation type set to *priority trigger* is active.
- Data. If port's activation type is set to data, it is only used to receive data, and does not affect the triggering of the component.

The graphical representation of output ports and all input port types can be seen in Figure 3. 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*.

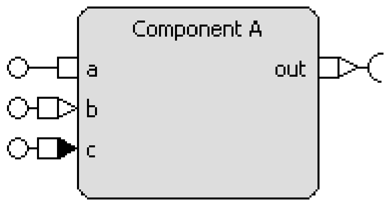


Figure 3: Graphical representation of an UComp component and its input and output ports.

3.1.2. Device Components

Device components are the base for accomplishing the uniform treatment of components of a system, which is not dependant on component realization (hardware or software). They represent hardware (physical) and virtual (realized using software applications) network devices.

Device components, together with their input and output ports, are automatically generated by the UComp framework using device descriptions. Automatic generation of devices and their ports eliminates need for specialized drivers or manual configuration of such components by the developer of the system.

Every device component signals if the actual device is available on the network. This information can be very useful in distributed systems where connection between components is not reliable and some of them can be temporarily unavailable.

Device components can represent actions (synchronous request-response communication) or events (asynchronous publish-subscribe messaging) of network devices. Therefore, we have defined two types of device components: *action components* and *event components.*

Action Components. Action components are designed to wrap around synchronous action invocations or data queries of a device connected to the network.

When an action component is triggered, action invocation is performed, using values of its input ports as arguments. When the invocation finishes, results of the invocation are propagated to output ports of the components.

Event Components. Event components allow receiving asynchronous messages from devices. These messages may signal data changes or other events that device may provide.

Interfaces of event components have only output ports, which obtain their values from event notifications provided by the network device.

3.1.3. Software Components

Functionality of software components is fully implemented by program code, and they are not bound to any hardware elements. They are used to process the data received from, or sent to, device components or manipulate the execution of components. Their function can vary from very simple (for example addition of two numbers) to complex data processing.

3.2. Execution Semantics

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. 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.

4. Realisation of UComp Component Model

The UComp architecture, shown in Figure 4, is realised as Java application that implements the Universal Plug and Play (UpnP) [16] technology to control devices available on the network, process their data, and relay data between them. The application communicates with devices through a single UPnP control point implemented using CyberLink UPnP stack [11]. Centralized architecture allows us to process data received from a device by UComp application before it is forwarded to other devices, making the system much more flexible and eliminating the need to change the code of devices to adapt them to the needs of the developed system. Also, 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.

4.1.1. Development environment

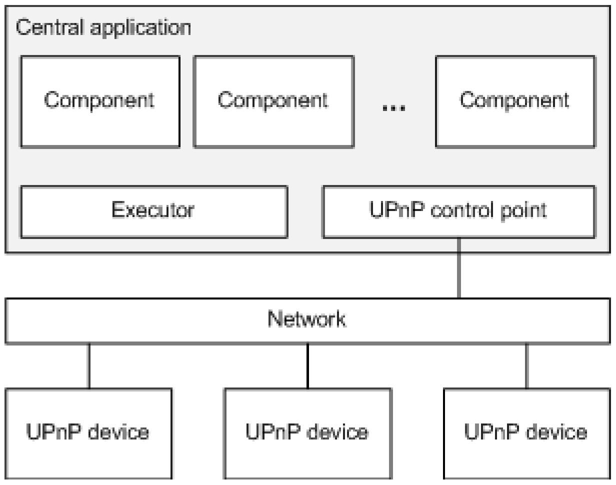


Figure 4: The UComp architecture.

To facilitate the development we have created a tool for visual development of UComp systems - *UComp Developer.* The *UComp Developer* enables browsing available device and software components using a tree structure, 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 are saved or restored from XML files.

A screen-shot of *UComp Developer* is shown in Figure 5.

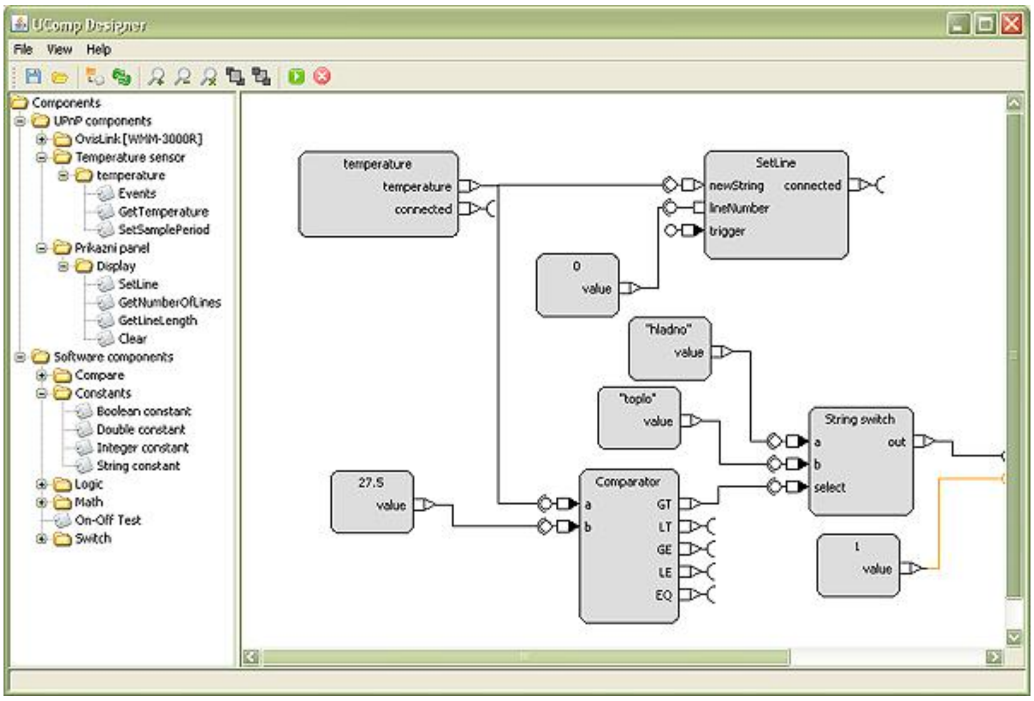


Figure 5: Screenshot of UComp Developer tool.

5. An overview of the PRIDE tool

PROGRESS-IDE (PRIDE) is an integrated development environment used for software development of distributed embedded systems (ES), primarily in the automotive, automation and telecom domains. The grand vision of Progress-IDE is to cover the whole software development cycle of distributed embedded systems, from early system design to deployment and synthesis. The development process of ES requires a strong emphasis on analysis, verification and validation in order to ensure the necessary quality of the delivered product, therefore PRIDE integrates various analysis tools. Compared to the majority of existing IDEs that focus mainly on the programming aspect, PRIDE uses the notion of component as a first-class concept allowing the manipulation and modelling of components, with reusability as one of the key concerns and supporting:

- Component and system design using the ProCom component model [13],
- System analysis (worst-case execution time, model checking of behavioural models and fault propagation and transformation calculus),
- Deployment modelling
- Code synthesis.

In the following list we give some overall reflections that apply to Progress-IDE.

- Progress-IDE covers the whole system development process,
- Progress-IDE is developed as a stand-alone application on top of Eclipse RCP,
- Progress-IDE consists of a number of editors: architectural editors, attribute editor, node description (virtual/physical), allocation editor, code generation, various analysis editors,
- Progress-IDE is user-friendly by providing a user interface respecting the usability features, partial auto-completion features, default values etc.

5.1. The PRIDE approach

In the recent years ES development has changed significantly due to the rapid increase of software in these systems and software becoming as complex as in conventional systems. In non-embedded domains, new approaches such as model-based, component-based, and service-oriented development have been proposed to manage software complexity and there is a trend to apply these approaches also in ES development. ES correctness is strongly correlated to specific extra-functional properties (EFPs) such as timing (e.g. execution and response time) or dependability (e.g. reliability and safety) under constrained resources such as memory, energy, or computation speed. This calls for additional domain-specific technologies that provide support not only for functional development but also for analysis and verification of EFPs.

As a possible solution, we have been developing a new component-based approach [5] built around a two-layer component model - ProCom, which addresses the particularity of ES development from big complex functionalities, to small, close to control loop functionalities. This approach requires specific tool support that enables:

- Efficient system design by using existing components,
- Seamless integration of different tools to provide the analysis and verification required for system correctness, and
- Efficient EFP management of components and systems.

PRIDE uses reusable software components as the central development units, and as a means to support and aggregate various analysis and verification techniques. In difference to similar approaches [1, 2, 17], PRIDE puts emphasis on EFPs during the entire lifecycle - from early specification to deployment and synthesis.

PRIDE has been designed to support four design strategies that are especially important to consider for having an efficient component-based development of ES.

Levels of abstraction
Using components throughout the whole development process implies that the component concept spans a wide range of abstractions, from a vague and incomplete early specification, to very "concrete" with a fixed specification, a corresponding implementation and information about their EFPs. This means that components at different levels of abstraction must be able to co-exist within the same model.

Component granularity
In distributed ES, components span a large variety in size and complexity; the larger components are typically active (i.e. with their own thread of execution) with an asynchronous message passing communication style, whereas the smaller components are responsible for a part of control functionality with a strong synchronization. For an efficient development, a support for handling different types of components must be provided.

Component vs. system development
The common distinction between component development and system development brings issues in ES development, where the coupling between the hardware platform and the software is particularly tight. As a consequence, component development needs some knowledge of where the components are to be deployed. This requires support to handle the coupling between components, system and target platform, while still allowing separate development of components and systems.

Extra-functional properties
The correctness of ES is ascertained based on both the functional and extra-functional aspects. However, many EFPs typically encountered in ES are assessed through different methods during the development lifecycle (from early estimation to precise measurements) and different values may be obtained according to the characteristics of the resources of the platform on which the components are to be deployed. For this reason, an efficient ES development should provide a means for specifying, managing and verifying these properties with respect to the context in which their values are provided.

To comply with the design strategies, the following requirements have been identified as principles that guided the design and development of PRIDE:

- Allowing to move freely between any development stages,
- Displaying the consequences of a change in the system or within a component,
- Supporting the coupling with the hardware platform, and
- Enabling and enforcing the analysis, validation and verification steps.

In addition, a central requirement relates to the notion of component. Components are the main units of development and seen as rich-design artifacts that exist throughout the whole development lifecycle, from early design stage, in which little information about them exists, to deployment and synthesis stages, in which they are fully implemented. PRIDE views a component as a collection of all the development artefacts (requirements, models, EFPs, documentation, tests, source code, etc.), and enables their manipulation in a uniform way.

Driven by the aforementioned principles, several tools have been developed and tightly integrated into PRIDE. Since the PRIDE is built as an Eclipse RCP application, it can be easily extended with addition of new plugins.

5.2. Implementation of PRIDE

The Eclipse platform is chosen as the supporting architecture for PRIDE. In order to reduce the overhead which exists when using Eclipse directly as the main integration environment, it has been decided that the PRIDE will be developed as a stand-alone application built on top of the Eclipse Rich Client Platform (Eclipse RCP). This decision is driven by the will to keep the control over the features present in the environment such as, for example, avoiding the presence of menus not directly related to the particular use of the PRIDE. When using Eclipse RCP, only the features explicitly added to the environment are present. In other words, the layout and function of the environment are fully controlled by the plugin developers.

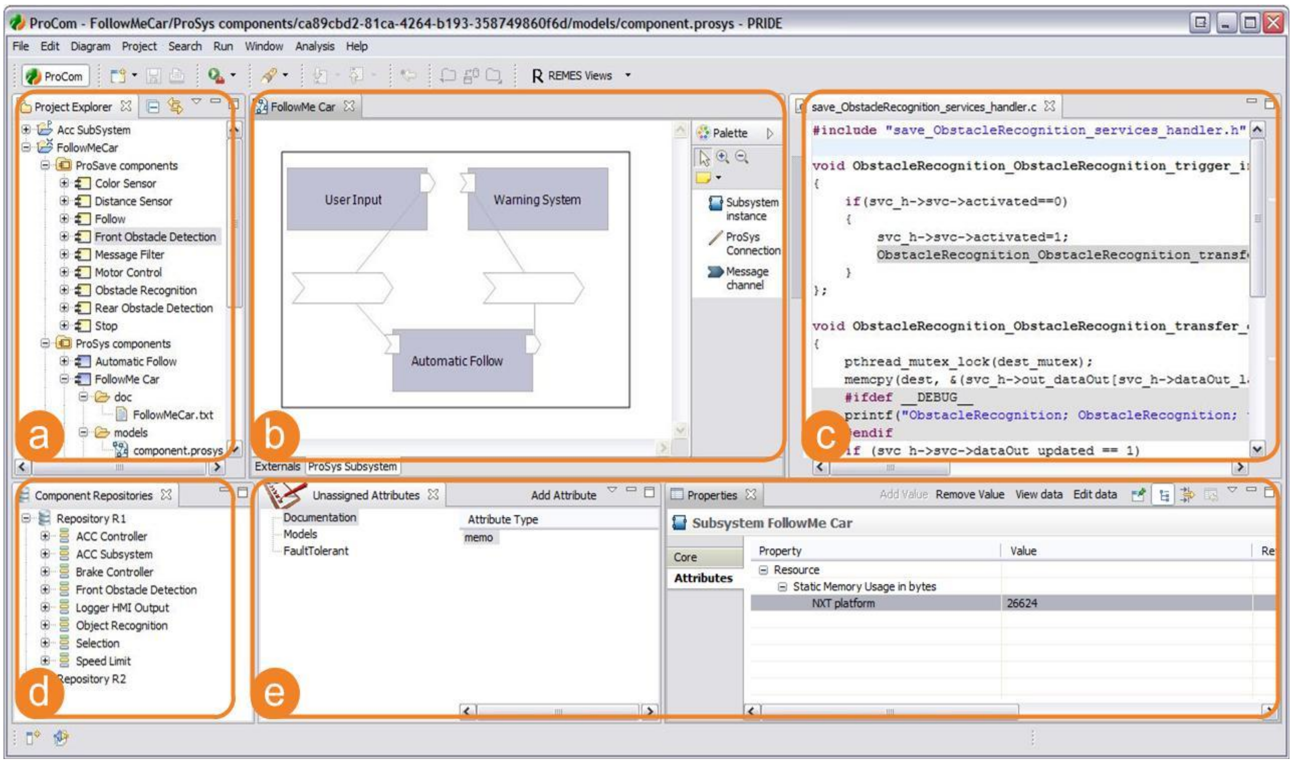
The component architecture design part of the environment was implemented using EMF [14] and GMF [15]. EMF is a modelling framework and code generation facility for building tools based on a structured data model. It offers a graphical editor for describing metamodels. With the aid of this, the ProCom metamodel (model of the ProCom model) was defined. From this graphical description of the metamodel, EMF automatically generates Java classes representing the model, classes used for modifying the model and textual tree editors for the model. GEF then takes this existing application model (i.e. the model generated using EMF) and quickly creates a rich graphical editor. However, this automatically generated editor has rudimentary functionality and needs to be further manually modified and tweaked in order to achieve the desired functionality.

5.3. Software architecture modelling and analysis

As already mentioned, PRIDE uses component-based approach which allows components to be developed independently and reused in different contexts. Reusability is one of the main concepts in PRIDE aiming to significantly shorten development time. The tool makes a distinction between a component type and a component instance. Each reusage of a component creates a component instance of the given component type. By editing the component all of its instances are affected.

In PRIDE components are rich design entities encapsulating a collection of development artefacts; requirements, various models (e.g. architectural, behavioural, resource usage etc.), EFPs, documentation, tests and source code.

Figure 6 shows a screenshot from PRIDE with main parts highlighted. PRIDE's modelling part consists of a component explorer and component editors.



save_ObstacleRecognition_services_handler.c

```
#include "save_ObstacleRecognition_services_handler.h"

void ObstacleRecognition_ObstacleRecognition_trigger_i
{
    if(svc_h->svc->activated==0)
    {
        svc_h->svc->activated=1;
        ObstacleRecognition_ObstacleRecognition_transf
    }
};

void ObstacleRecognition_ObstacleRecognition_transfer_
{
    pthread_mutex_lock(dest_mutex);
    memcpy(dest, &(svc_h->out_dataOut[svc_h->dataOut_l
    #ifdef __DEBUG__
    printf("ObstacleRecognition; ObstacleRecognition;
    #endif
    if (svc_h->svc->dataOut_updated == 1)
```

Property	Value	Re
Resource		
Static Memory Usage in bytes		
NXT platform	26624	

Figure 6: *A screenshot from PRIDE showing a) the component explorer; b) the component editor; c) the code editor; d) the repository browser; and e) the attribute framework.*

Component Explorer

Enables browsing the list of the components available in the current development project. In it a component owns a predefined and extensible information structure corresponding to a rich component concept. It also provides a support for component versioning and importing and exporting from a project to a component repository making them available to other projects thus facilitating the component reusability.

Component Editors

Although the ProCom component model distinguishes between two different types of components (ProSave and ProSys), in the component editors all components are treated in a uniform way. Component editors provide two independent views on a component, *external* and *internal* view. The external view manages the component specification and interface such as the information about the component name, its interface (services and ports) and EFPs. The internal view focus on component's internal structure i.e. its realization and implementation and it depends in the component realization type. For primitive components, the internal view is linked to the component implementation and the source code is displayed. For composite components, the internal view corresponds to an interconnection of subcomponent instances and a graphical form is made available allowing to make modifications in this inner structure (addition/deletion of component instances, connectors, change in the connections, etc.).

The analysis support in PRIDE is based on two main parts, an attribute framework and an analysis framework.

Attribute Framework

The purpose of the attribute framework [12] is to provide a uniform and user-friendly structure to seamlessly manage EFPs in a systematic way. The attribute framework enables attaching extra-functional properties to any architectural element such as a specific port, service or the component as a whole. Attributes are defined by attribute types, and include attribute values with metadata and the specification of the conditions under which the attribute value is valid. One key feature is that the attribute framework allows an attribute to be given additional values during the development without replacing old values. This allows defining of early estimates for EFPs even before the component has been implemented and use it for analysis in early stages of system development. New, user-defined attribute types can also be added to the model.

Analysis framework
The analysis framework provides a common platform for integrating in a consistent way various analysis techniques, ranging from simple constraint checking and attribute derivation (e.g., propagating port type information over connections), to complex external analysis tools. Analysis results can either be presented to the user directly, or stored as component attributes. They are also added to a common analysis result log, allowing the user easy access to earlier analysis results. PRIDE also allows to easily integrate new analysis techniques together with their associated EFPs.

6. Conclusion

In this paper we gave an overview of some of the results of DICES project. The project focus was on distributed embedded systems and the overview shows that a large variety of technologies and approaches are needed for an efficient development and maintenance of embedded systems. The project is focused on several phases in the lifecycle: a) reverse engineering and analysis of the existing legacy code that helps in reusability in development of new systems from existing components; b) modelling and designing embedded systems that include both software and hardware components, and ability to perform different types of analysis important for embedded systems (e.g. resources utilization, performance, timing characteristics). The work done shows that the embedded systems of today are complex system, in the first hand with complex software that is approaching in size and complexity software from other domains.

Acknowledgment
This work was supported by the Swedish Foundation for Strategic Research via the strategic research centre Progress, Croatian Ministry of science, education and sports via the research project Software engineering in ubiquitous computing, and the Unity Through Knowledge Fund via the Dices project.

References

1. Akerholm M., Carlson J., Fredriksson J., Hansson H., Hakansson J., Möller A., Pettersson P., Tivoli M. (2007) The SAVE Approach to Component-Based Development of Vehicular Systems, In: Journal of Systems and Software, Elsevier Science, pp. 655-667
2. Articus Systems, Rubus Software Components, http://www.arcticus-systems.com , Accessed 7 March 2011
3. Bellettini C., Marchetto A., Trentini A. (2004) WebUml: reverse engineering of web applications, In: Symposium on Applied computing, ACM New York, NY, USA, 1662-1669
4. Brandt J., Guo J.P., Lewenstein J., Klemmer S.R. (2008) Opportunistic programming: How rapid ideation and prototyping occur in practice, In: Workshop on End-user software engineering, ACM New York, NY, USA, pp. 1-5
5. Bureš T., Carlson J., Sentilles S., Vulgarakis A. (2008) A Component Model Family for Vehicular Embedded Systems, In: The Third International Conference on Software Engineering Advances, IEEE Computer Society Washington, DC, USA, pp. 437-444
6. Di Lucca G., Fasolino A.r., Tramontana P. (2004) Reverse engineering Web applications: the WARE approach, In: Journal of Software Maintenance and Evolution: Research and Practice -

Special issue: Web site evolution Volume 16 Issue 1-2, John Wiley & Sons, Inc. New York, NY, USA, pp. 71-101

7. Feljan J., Lednicki L., Maras J., Petricic A., Crnković I. (2009) Classification and survey of component models, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-242/2009-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University
8. Holmes R.(2008) Pragmatic Software Reuse, Ph.D. Dissertation, University of Calgary
9. Lednicki, L. and Petricic, A. and Zagar, M. (2009) A Component-Based Technology for Hardware and Software Components, In: Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on, IEEE Computer Society Washington, DC, USA, pp. 450 -453
10. Ricca F., Tonella P. (2001) Understanding and Restructuring Web Sites with ReWeb, IEEE Multimedia, Vol 8, Issue 2, pp. 40-51
11. Satoshi Konno, CyberLink for Java, http://www.cybergarage.org/twiki/bin/view/Main/CyberLinkForJava , Accessed 7 March 2011
12. Sentilles S., Štěpán P., Carlson J., Crnković I. (2009) Integration of Extra-Functional Properties in Component Models, In: Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE '09), Springer-Verlag Berlin, Heidelberg, pp. 173-190
13. Sentilles S., Vulgarakis A., Bureš T., Carlson J., Crnković I. (2008) A Component Model for Control-Intensive Distributed Embedded Systems, In: Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE '08), Springer-Verlag Berlin, Heidelberg, pp. 310-317
14. The Eclipse Foundation, Eclipse Modeling Framework Project, http://www.eclipse.org/emf/ , Accessed 7 March 2011
15. The Eclipse Fundation, Graphical Modeling Framework, http://http://www.eclipse.org/gmf , Accessed 7 March 2011
16. UPnP forum, Universal Plug and Play, http://www.upnp.org , Accessed 7 March 2011
17. van Ommering R., van der Linden F., Kramer J., Magee J. (2000) The Koala component model for consumer electronics software, In: IEEE Computer, , IEEE Computer Society Washington, DC, USA, pp. 78-85
18. Maras J., Štula M., Carlson J, Crnković I. (2011). Firecrow web page, http://www.fesb.hr/~jomaras/?id=Firecrow, Accessed 7 March 2011