Predictable dynamic deployment of components in embedded systems

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

Abstract—Dynamic reconfiguration – the ability to hot swap a component, or to introduce a new component into the system – is essential to supporting evolutionary change in long-live and highly available systems. A major issue related to this process is to ensure applications consistency and performance after reconfiguration. Due to systems dynamicity it is hard to keep the model of a system updated which makes impossible to test the new component in real conditions. Consequently, component testing and verification has to be performed within reconfiguration. We are particularly interested in preserving performance attributes which is important in embedded systems with limited resources available. Additionally, performance constraints must be met even while the system is reconfiguring. Our main objective is to develop a dynamic deployment mechanism for embedded systems which is resource efficient and ensures predictability of system behavior by introducing performance attributes verification of deployed components.

Keywords-component-based software engineering; dynamic deployment; dynamic software updating ; verification; predictability

I. INTRODUCTION

Component-based software engineering (CBSE) is an emerging approach in software engineering which tends to build software systems out of pre-existing, and reusable components. Every component can be developed and tested independently and separate from systems they compose. One of advanced features of component-based systems is the possibility to introduce new components and update or replace old components in order to improve the system characteristics. This characteristic of CBSE has proved to be very beneficial as today’s software evolves over time and improvements in software are developed in a rapid succession. In consumer electronics (CE) for example, the value and the economic lifetime of a device and the software on it can be increased by upgrading and extension of the software.

There are particular cases where applications require a very high level of availability in the sense that they cannot be stopped (e.g. in order to restart the system for updates to take effect). Therefore operations like deployment of new versions, patches or extended features need to be done while the application is running. This process is usually referred as dynamic deployment of components. Examples are most mission critical areas such as telecommunication, banking systems, avionics, and industry control. Interruption of these systems implies high cost (think of shutting down one industry pipeline) and even life-threatening (think of shutting down one life-support system). Consequently, maintaining and upgrading of a system while preserving continuous availability of services is an issue of great importance.

A critical issue that dynamic reconfiguration has to deal with is preserving of system’s integrity and ensuring that the newly updated component meets system requirements, both functional and extra-functional. In dynamic systems, it is often the case that the system model does not exist due to system’s dynamicity. Consequently it is not possible to test the new component in real conditions, this is especially the case in large dynamic systems such as telecommunication switching networks. There is a need to perform component verification during its deployment to a system in order to be able to predict its behaviour after reconfiguration. We are particularly interested in verification of non-functional properties (e.g. CPU share, memory, energy, bus bandwidth) which are critical for embedded systems. There are solutions that successfully deal with dynamic reconfiguration of component-based systems, some of them dealing with functional verification of components, but there is no work on verification of extra-functional properties.

The limited nature of available resources in embedded systems, especially memory size and computation resources, also implies a demand on a verification process to be lightweight and not to encumber a system. Our goal is to develop a dynamic deployment mechanism for embedded systems which is resource efficient and ensures predictability of system behavior by introducing performance attributes verification of deployed components. This paper gives a background and an overview of current work concerning this topic, as well as initial research questions.

The rest of the paper is organized as follows; Section 2 gives a short introduction to embedded systems domain and motivation for this work. Sections 3 and Section 4 give the

1 From perspective of a component we use term dynamic deployment. When talking from system’s point of view, we use term dynamic reconfiguration (refers to changing, updating or otherwise modifying system during execution).
overview of dynamic reconfiguration in component based frameworks and deployment mechanisms in general. Section 5 introduces some important questions concerning verification of components during reconfiguration, and Section 6 covers related work describing deployment mechanisms in two relevant approaches – ROBOCOP and OSGi. Finally Section 7 summarizes future work plans concerning research goal and questions as well as expected results.

II. EMBEDDED SYSTEMS DOMAIN

Embedded systems span all aspects of modern life and most, if not all, of the system examples mentioned above consist of embedded systems. They range from portable devices such as cell-phones (consumer electronics), to vehicle control systems and large stationary installations like traffic lights, factory controllers, or the systems controlling nuclear power plants. Complexity of embedded systems varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

Embedded devices are characterized by their long life cycle, and as such require and can always benefit from seamless upgrading of their software. Except for upgrading, dynamic software reconfiguration allows the system to adapt its functionality in response to frequent changes in the device’s context. Finally, devices can be customized to the needs of a consumer in the period that it is owned and used by the consumer.

As they are dedicated to specific tasks, embedded systems are always optimized. Some have real-time performance constraints that must be met, for reasons such as safety and usability; others may have low performance requirements, allowing the system hardware to be simplified to reduce costs. They run with limited computer hardware resources, in most cases with little memory. All these characteristics imply many constraints imposed at embedded systems, which makes system updating or upgrading, a very delicate operation.

Due to these constraints, except for functional verification, in case of upgrading an embedded system detail performance verification has to be done, in order to be sure the system will keep its performance after new component has been deployed. If we speak about dynamic deployment, then this task is even harder, as the target system has to keep running and cannot be used for testing. There are cases where system replica exists, but also there are cases such as telecommunication domain where it is extremely hard to simulate the system or to make its replica. In such a case it is necessary to have mechanisms to specify system requirements constraints, as well as characteristics of each component, and to provide ways for testing new component in order to confirm its compliance with the target system.

III. DYNAMIC DEPLOYMENT IN COMPONENT-BASED FRAMEWORKS

Component based software engineering and component based frameworks provide an infrastructure which is suitable for dynamic deployment processes in a way that only some components in a system can updated, changed or deployed instead of the whole application. Today’s component frameworks can support upgrading and extension at different stages of software life-cycle (design-time, compile-time, run-time, etc.). Several general-purpose component technologies support dynamic deployment of components, such as OSGi framework [20], Robocop [19, 23, 16] or SOFA [21] (first two frameworks are explained later on in more detail). Run-time upgrading is also supported by some industrial component models like COM [2], EJB [24, 26] and .NET [15].

However, the requirements on a specific component framework highly depend on the problem domain in which the framework is used. For this reason, the feature of dynamic deployment often has problems with system dependability; a new component can break the system’s integrity, which can lead to the system failure or degradation of its performance. The mechanisms that enable dynamic reconfiguration require certain system resources which are not critical for large systems, but are not under the control of the system. As such, these mechanisms are not appropriate for embedded systems.

Another, even more important issue is the verification. Most of the existing solutions focus at the upgrading mechanism, and only a few mention the verification of deployed components.

IV. DYNAMIC DEPLOYMENT MECHANISM

In a context of run-time reconfiguration, a distinction is usually made between functional and structural changes. Functional changes include the addition of new/improved code to a running system in order to modify/upgrade its functionality, whereas structural changes refer to reconfigurations that change the relationship between different components of the system, or replicate portions of the application for execution on a different machine. The focus of this research is on functional changes.

Most component-based approaches consider contractually defined components with well defined interfaces for checking consistency and interoperability. When updating a component, there are several possible cases that affect the deployment process:

- Changes do not affect external interface of the component i.e. some methods were being re-implemented and improved.
- Changes to the component are affecting its external interface in a way that the update only needs to preserve the old interface being possible to add new methods to it in order for new client to use them.
- Some approaches such as GNU/EDMA [18] consider cases where external interface of the component is being changed. One of the solutions is to insert new, adapter component between the current clients and the new updated component. By doing so, all method invocations pass through the adapter which performs matching transformation of parameters.

It is important to note that the observation applies to both provided and required interfaces. This can also be discussed
from perspective of component input and output. Its range can stay the same or it can be narrowed or expanded. In general case these observations can only be applied to functional properties of a component, non-functional properties and impacts of their changing need to be observed individually (different properties require different approaches).

A mechanism of dynamic deployment usually consists of the following four steps (the mechanism needed for uploading new functionality can be largely shared with that for upgrading):

1. Initiation of a change
2. Identification of affected components (identification of change type and scope)
3. Accomplishment of upgrading
4. Consistency analysis/check

Various existing approaches have different focus on a particular step and thus aiming at different goals. However, most of them, are lacking the last phase, either completely (e.g. GNU/EDMA), or it is predicted in the model but a little work is done on that account (e.g. Robocop). Also most of the solutions are missing the verification phase which verifies and tests the component (its functional and extra-functional properties) prior to deployment. In order to be predictable and safe in terms of not breaking system’s integrity and performance, desired reconfiguration mechanism should consist of the following phases:

1. Initiation of a change
2. Verification phase
3. Identification of affected components (identification of change type and scope)
4. Accomplishment of upgrading
5. Consistency analysis/check

V. VERIFICATION OF COMPONENTS

Verification phase is the main focus of our research. Particularly we are interested in the verification of extra-functional properties (EFPs), as these are critical in embedded systems domain. This section will give some of the questions that should be taken into consideration when designing the verification mechanism as a part of dynamic deployment. These are (note that answers to these questions are different for diverse EFPs):

- How to acquire system’s and component’s requirements and performance attributes?
- Is this data defined during system modelling/component development or is calculated and obtained at run-time?
- If the data is obtained at run-time, how is it obtained, by inspection or introspection?
- How to keep specified requirements during run-time? How is this data structured?
- What aspects have to be tested in order to be sure that the new component is suitable for target system?
- Since the target system has to keep running (i.e. it is occupied), how to run the tests?
- What entities are in charge of performing the verification?

These questions can be grouped to several important areas as follows.

A. Describing and dynamically obtaining functional and performance requirements

Requirements can be defined during development and described in documentation attached to each component or to the whole system. But in case of dynamically evolving system this is not a suitable solution. As the system changes its specification may be changed. Also the requirements can change according to the context and environment of the system (e.g. in mobile phones user can set his preferences).

There are several possible approaches in dynamic context:

- Requirements can be obtained during run-time by inspection or introspection.
- Collecting statistical data about the system.
- There is a formal model of the system which is evolving in parallel with the system and is used for analyses.

B. Non-functional characteristics and their verification

It is important to identify extra-functional characteristics that are of interest in embedded systems. As the nature of these properties varies, so does the approach for their verification. For example, for execution time it is necessary to observe the whole system or at least the task that the specific component is involved in. If the new component has better execution time it may happen that it computes too fast.

The internal state of the system and the component may also influence the verification result.

C. Verification mechanism

This refers to entities that perform the verification mechanism (for example special modules in the system that are in charge of deployment mechanism and verification). Also the mechanism should not influence the system in terms of resource consumption and system response time.

D. Disparity between design-time and run-time model

In most cases run-time model of the system is different from design-time model. During run-time the system is task oriented, i.e. its computation is distributed to tasks. During design time, the focus is on components that build the system, i.e. its architecture. Usually, several components are involved in computing one task. This means that when verifying a component, the task that this component is involved in should be taken into consideration.

VI. RELATED WORK

The problem of dynamic reconfiguration is not new. It is almost as old as software engineering itself, but it especially
became apparent with development and expansion of distributed systems. Kramer andMagee founded the area of dynamic reconfiguration with their work from 1990. [14] which was highly influential. Their outcome was to safely change a system at runtime by keeping the unaffected part operational, while the affected part is passive. The foundation for their approach is a concept of quiescence. They describe a system in which a programmer can specify the desired changes in a declarative way. The concept of quiescent nodes is used for safe removal of a component from the running system. They did not attempt to transfer state between different versions of a component, so their system needs extension in order to truly implement dynamic updating.

Later on a lot of work in the area was done and it has been an active research field especially in last few years. But very a few of them focus on embedded systems domain.

In the beginning the problem was mainly tackled from programmer’s point of view, with proposals that aim at easing reuse of existing components in the development of new applications. Most proposals in this category were based on code modification [3, 1, 12, 13]. There were proposals on adapting running applications, based on wrappers [21] or meta-level architectures [17].

Many approaches focus on updating of procedural systems (e.g. [7, 9, 10]). Using indirection, a method call is diverted to a new/updated version of the procedure. Wrappers are used to handle changes to interface and in some cases state is transferred using mapping functions. Dynamic updates can be initiated by the application itself or by an external event. However, applying procedure-based approaches to component-oriented applications is extremely difficult, since this breaks component-oriented abstractions and type safety. Therefore, these systems will not be considered any further in this research.

A simple method for implementing dynamic deployment in an object-oriented system is by using the Strategy Design Pattern [8]. This solution allows for changing an algorithm or strategy at run-time, but it requires that the application is designed with future changes in mind, and that all possible strategies must be known at compile-time. Therefore, it is not suitable for unanticipated updating which is in the focus of this work.

A very flexible approach can be found in meta-level architectures (e.g. [5, 22]). Through reification of object oriented concepts (class, method-call etc.), a meta-model is build on top of the application. This way, changes to the application are possible by changing the metamodel. A great advantage of this approach is that this provides a clean separation between the application code and the code responsible for the reconfiguration (meta-code). In some cases it is possible to build an adaptation framework [6], however, it requires the large overhead which is not acceptable in an embedded system context.

The rest of this section describes two significant approaches that support dynamic deployment in component-based systems, ROBOCOP and OSGi.

A. ROBOCOP

Robocop defines an open middleware layer for high volume consumer electronics. It aims to support definition, modelling and trading of software components, their use in CE applications, and run-time upgrades and reconfiguration of such applications. Most important requirements for CE domain that Robocop deals with are robust and reliable operation, run-time upgrading and extension, low resource footprint and support for component trading.

Robocop defines components as units of trade. It is a set of different models that are related to each other. These models address different aspects of the component, for example interface definition (Robocop IDL), behavioural models, resource consumption models, etc. There is no limit in number or types of models that a component consists of. A special type of model is the executable component. It is a binary representation of the component that implements its functionality and can be executed. A component can have multiple executable components (implementing the same IDL) that are targeted for different platforms or operating systems.

Robocop architecture defines three frameworks that support different concerns of component's life cycle:

The Development framework defines how different stakeholders (e.g. component vendors, system integrators) relate to each other, to the component model and entities like component repositories, and target devices.

The Run-time framework defines a partial architecture for Robocop devices. It consists of the Robocop component model and the Robocop Runtime Environment (RRE). To achieve a minimal resource footprint, standard RRE supports only registration of components and services, and location and instantiation of services. Robocop component model defines a standardized part of service interface that, together with RRE, provides mechanisms for run-time binding of components and reconfiguration of Robocop systems.

The Download framework enables run-time upgrades and extension of applications built with Robocop technology. This is achieved by providing mechanisms for locating new components, testing if they are suitable for use in a given system, and transfer of new components from repositories to target devices.

Within Robocop project, there is a little work done considering tailoring of components and their automated registration within RRE. Considering locating of components and target loading there is a solution proposed which consists of the following phases:

- Locating of entities that participate in the download process
- Decision about feasibility of the download
- Transfer of Robocop component to the device (target loading)
- Confirmation of download and registration of the downloaded component at RRE

In this process there are several roles existing, initiator, locator, decider, repository and target device. These roles can
run at different devices, except the target role which runs at the
device to which a component is transferred.

Initiator is responsible for verifying that all entities required
in the download process are present, and also to coordinate the
process of download. Initiator is triggered either by the
component upgrading process or as result of a change in user preferences settings.

Locator deals with locating last three entities in the
download process, repository which contains the component to be
downloaded, target device where component will be
deployed and the decider which makes a decision whether the
download will take place or not.

Decider role is here to compare component existing in the
repository to the requirements in the target device, and then to
decide whether the component is suitable for downloading.
Decision procedure performed by decider may be sophisticated
and can depend on the domain, for example in case of real-time
systems a schedulability test can be performed. However, until
now there is little work been done on deciding techniques in
Robocop.

Considering resource consumption during dynamic
reconfiguration process, Robocop implements resource
management through the Resource management framework.
The aim of this framework is to prevent resource overloads on
embedded devices that support dynamic updates or upgrades. It
introduces a notion of resource-aware consumers, which are
application entities that have information about resources
needed for its functioning. Special types of such entities are the
quality-aware consumers, which consume different amount of
resources depending on the level of quality they provide in a
given moment. The consumers can register their resource needs
to the framework, which can then guarantee them requested
resources or deny their request. The framework can also
optimize system quality depending on the available resources.
A solution of memory consumption of Robocop applications is
given in [11].

B. OSGi

The OSGi stands for Open Service Gateway initiative, an
industry alliance founded in 1999, with a mission to specify
and develop an open service platform for delivery and
management of multiple applications and services to all types
of network devices in home, vehicles, telecommunication
network and many other environments. The OSGi emphasis is
on a lightweight framework that can be executed in low-
memory devices such as all kinds of embedded systems. To
ensure portability on different hardware, OSGi relies on Java.
OSGi supports dynamic evolution of a system. Components
can be downloaded, updated, and removed dynamically,
without stopping the system. Moreover, the OSGi allows for
remote administration of the system via network.

OSGi is based on two main concepts that can both be
interpreted as components, bundles and services. Application
is developed as a set of bundles that contain services. Each
service (defined as an implementation and as a set of
interfaces) implements a part of the overall functionality. A
service can be considered as a unit of composition; and a

system as a set of cooperating services that are connected. On
the other hand, bundle is a unit of deployment that groups a set
of services that can be dynamically deployed as a unit.

From external point of view, a bundle can be presented as
shown in Figure 1 (figure is taken from [4]). We can
distinguish three kinds of interaction points:

Interaction with traditional technology. A bundle may
require and provide one or more Java packages. This part of the
interface is declared statically (coloured white on the figure).

Interaction with other bundle components. This part of
interface is intended to manage dynamic connections between
services, so that service interfaces (either required or provided)
can be attached or detached dynamically (coloured grey on the
figure).

Interaction with the run-time environment. This is intended
for a bundle to listen to events published by the run-time
framework such as the insertion of a new component in a
system or the publication of a new service. This way, a bundle
can adapt to the evolving architecture (coloured black in the
figure).

From internal point of view, a bundle is represented as a
JAR archive containing service components, Java packages,
and other files such as configuration files and images. The
entities contained in the bundle are not visible from outside as
long as the bundle does not export them.

Once a bundle is installed, it can be started, but before
bundle services can be executed, the bundle must be resolved.
OSGi bundle life-cycle consists of several states as depicted on
Figure 2.

- **INSTALLED** – The bundle has been successfully
  installed.
- **RESOLVED** – All Java classes that the bundle needs
  are available. This state indicates that the bundle is
  either ready to be started or has stopped.
- **STARTING** – The bundle is being started, the bundle
  activator method will be called, and this method has
  not yet returned. When the bundle has a lazy activation
  policy, the bundle will remain in this state until the
  bundle is activated.
• ACTIVE – The bundle has been successfully activated and is running; its activator method has been called and returned.
• STOPPING – The bundle is being stopped. The method has been called but the stop method has not yet returned.
• UNINSTALLED – The bundle has been uninstalled. It cannot move into another state.

Figure 2. OSGi bundle life-cycle

In OSGi, a system is an evolving set of bundle components. Components can connect to each other dynamically based on their own decisions, however this also implies that components cannot assume that the interfaces they use will be available at all times. This means that components may have some knowledge about how to connect and disconnect. Dynamic connection mechanism goes as follows:

• When a bundle component publishes a service interface, it can attach to it a set of properties describing its characteristics.
• When a component requires an interface, it will select one using a query expression based on these properties.
• The final connection is never given explicitly as the result of the query depends on the actual state of the system (depending on the components existing in the system and services they provide at the moment).
• Once a connection is established, there is no guarantee that the service will remain available. Each bundle component must listen to events generated by the OSGi run-time environment and must take appropriate action as the system evolves.

The OSGi framework includes a service gateway, a central entity to enable, consolidate and manage service requests and service delivery between some local area network and wide area network such as the Internet. The framework also provides a hosting environment with the following services:

• Managing life-cycle of bundles
• Resolving interdependencies between bundles and making classes and resources available from a bundle
• Maintaining a registry of services
• Notifying listeners on events when some bundle's state change, when a service is registered or unregistered, or when some error occurs

In the process of dynamic reconfiguration in OSGi, an explicit verification process is missing. However, there are several mechanisms that can ensure compatibility such as the compatibility of version numbers of requested component interface(s) or already mentioned properties attached to each provided service. On the other hand, comprehensive verification of performance characteristics which is very significant in embedded systems is not supported.

VII. FUTURE RESEARCH WORK

A. Research goal
The goal of our research is to develop a theory and a technology for dynamic deployment of components in embedded systems domain that meets the following requirements:

• A resource efficient mechanism for dynamic component deployment in embedded systems. The mechanism can be taken from an existing component model or service-oriented architecture, and adapted for usage in embedded systems domain.
• Predictable deployment. This includes a verification of whether a component to be deployed satisfies system requirements and whether a system can provide conditions defined by component’s requirements. The focus will be on verification of a set of selected EFPs.

B. Research questions
Questions that will be addressed in future research are:

• What properties of a component and a system need to be tested and how, in order to be sure that the new component is suitable for a target system?
• How to specify and how to obtain system’s and component's requirements and performance attributes?
• How can one run the verification if a target system is running?
• Which entities take part in dynamic deployment mechanism and particularly which entities are in charge of performing the verification?
• How to ensure low resource consumption of deployment mechanism and preserve system response time?

C. Expected results
As result of our future research we expect to have:

• An identified set of important extra-functional properties in embedded systems domain, with established verification approach for each property.
• A verification framework that verifies component compliance with the running system in terms of extra-
functional properties thus ensuring predictable

dynamic deployment of components.

- An extension of some existing dynamic deployment
  mechanisms with the verification framework.
- A prototype tool and demonstration on an industrial
case study.

REFERENCES

  for Gluing Components. In Wolgang Weck, Jan Bosch and Clemens
  Szyperski, editors, Proc. Of Third International Workshop on
  Component Oriented Programming (WCOP 98), pages 101-108. Turku

[2] D. Box. Essential COM. Object Technology Series. Addison-Wesley,
  1997.

  Information and Software Technology4(5), 249–305.


[5] J. Dowling and V. Cahill. Dynamic software evolution and the k-
  component model. In Workshop on Software Evolution, OOPSLA,

  reflection to support dynamic adaptation of system software: A case
  study driven evaluation. In OOPSLA Workshop on Object-Oriented
  Reflection and Software Engineering, pages 169–188, Denver,

[7] O. Frieder and M. E. Segal. On dynamically updating a computer
  program: From concept to prototype. The Journal of Systems Software,

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns:
  Elements of Reusable Object-Oriented Software. Addison Wesley,
  Massachusetts, 1994.

  Computer Science and Engineering, Indian Institute of Technology,

  Computer and Information Science, University of Pennsylvania, June

  of run-time resource consumption in component-based software
  systems, In Proceedings: 6th ICSE Workshop on Component Based

  Proceedings of the 12th European Conference on Object-Oriented

  of Components Through Binary Component Adaptation. Technical
  Report. UMI Order Number: TRCS97-15., University of California at
  Santa Barbara.

  16, no. 11. pp. 1293-1306, Nov. 1990, doi:10.1109/32.60317


[16] H. Maaskant, A Robust Component Model for Consumer Electronic
  Products, Philips Research Book Series Volume 3.

  Ubilab Technical Report, 97.6.1, Union Bank of Switzerland, Zürich,
  Switzerland, 1997.

[18] D. Martinez Oliveira, X. Fernandez Hermita, Run-time component
  extension and update, Technical report, University of Vigo. September,
  2002.

  Framework for Consumer Electronics Middleware, C. Adkinson et al.
  (eds.), Component-Based Software Development for Embedded

[20] The OSGi Alliance. OSGi Service Platform Specification, Release 4,

  Component Trading and Dynamic Updating, eds, pp-43, International

[22] B. Redmond and V. Cahill. Iguana/j: Towards a dynamic and efficient
  reflective architecture for java. In Workshop on Reflection and Meta-
  Level Architectures at 14th European Conference on Object-Oriented

[23] Robocoup Consortium. Robocoup: Robust open component based software

