# Recovering Runtime Structures of Software Systems from Static Source Code

Thomas Forster, Thorsten Keuler, Jens Knodel
Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern, Germany
{thomas.forster, thorsten.keuler, jens.knodel}@iese.fraunhofer.de

**Abstract:** *While software building blocks and their interdependencies can be recovered from the source code using static fact extraction, behavior and communication paths at runtime are typically gathered from instrumented executions of the system. However, more often than not it is not possible to retrieve data from the running system – either due to a high effort for instrumentation, missing (hardware) infrastructure, or because of advanced communication mechanisms hidden by middleware, frameworks or platforms.*
*In this paper, we present an approach to semi-automatically reconstruct runtime components and connectors using source code analysis, pattern matching, and expert knowledge. We present two applications where we could recover runtime communication paths and component interactions despite the absence of runtime traces.*

## 1 Introduction

Common practice is to reconstruct module views based on source code parsing and component and connector (in the following abbreviated with C&C) views based on dynamic analysis. Recovering the latter is a non-trivial task as reverse engineering runtime information is challenging for several reasons:
Understanding runtime traces and abstracting the programming entities and their interactions towards runtime components and connectors is a challenging task, too. Here, the same problems arise as in abstracting source code towards modules.
Code instrumentation may not always be possible especially embedded systems where software systems are often running on limited hardware resources.
Code instrumentation bears the risk that important components and connectors are not identified due to code coverage problems similar to the challenges found in software testing.
In this paper we show that it is possible to recover large parts of the runtime components and higher level connectors without system execution. In particular, the contributions of this paper are:

- A light weight approach for recovering runtime components and their communications paths from source code.
- Tool support for graphical pattern matching on module dependency graphs to identify components, connectors, and ports.
- Two case studies (industrial and academic).

For further reading please refer to the full paper published at the CSMR'13 [3]

## 2 Approach

We call our approach ReCoV – an acronym for Reconstruction of Component-Connector (C&C) Views. Recovery of C&C views without executing the software system requires understanding of how architectural components interact at runtime using higher level communication paths, the connectors. In short, understanding the architectural style and its manifestations in source code is the basis of the ReCoV approach. Our key assumption is that whenever a communication port is detected (e.g., an access to a connector middleware), we identified parts of a runtime component's interface. Applications of our approach in industry so far have reinforced this assumption. The ReCoV approach is structured as follows:

**Identification of relevant architectural styles for communication:** Initially we inspect the available documentation to learn about predominant styles defining the abstract communication principles in the system. Ideally, we can get answers to questions like: Is the system distributed across different nodes? Is it running in different processes? Is the communication paradigm time triggered, event triggered synchronous, or asynchronous? How is data exchanged? How are relevant architectural styles realized in code, i.e. how are components and connectors implemented? What are the component and connector types?

**Parsing of Source code:** We parse the source code automatically and relevant facts are extracted and represented in form of a detailed source code model capturing object oriented and procedural language constructs like packages, classes, interfaces, types methods and procedures, attributes as well as the relations between these elements like calls, variable accesses, import and inheritance relations and so on.

**Identification of Communication Ports in Source Code, Graphical Pattern Modeling and Matching:** The goal of this step is to bridge the abstraction gap between implementation and the

C&C models. We define graphical patterns representing the generic module-dependency-substructures (comparable to a regular expression) formalizing the manifestations of the architectural style at source level with a special focus on the connector endpoints (i.e. the ports). Then we match this pattern against the extracted source code model automatically. Figure 1 shows an example of a pattern describing a port as we model it in our tool. The corresponding structural elements in the figure have been marked with numbers to support the explanation: We want to match all classes (#1) that extend (#1.1) the *ApplicationComponent* base class (#2). Moreover this class (#1) has to implement (#1.2) an interface (#3) that itself extends the interface *IRemoteInterface* (#4). (The fact that these elements are interfaces is configured in the editor's property section for that element). To be sure that we match servers that are really instantiated we also check whether there is an instantiation call to #1 from another class (#5 and #5.1) and a subsequent registration call (#5.2) to the *AdaptationManager* (#6) where #1 is registered as service.

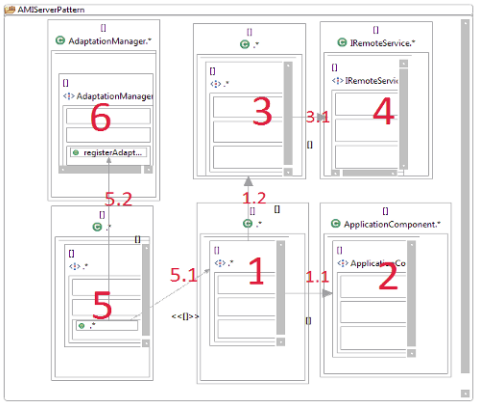


**Figure 1: Server-Side Port Pattern**

Using structural patterns like the ones in Figure 1 we search the extracted source code model and return concrete matches as illustrated in Figure 2.

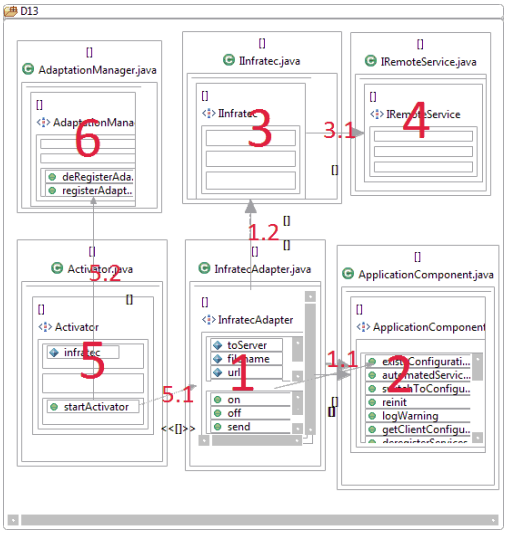


**Figure 2: A matched Server-Side Port pattern**

**Recovery of Components and actual connections:** Based on the extracted ports we can navigate to the code context in which the port was detected and iteratively create component abstractions and their attached ports. This process may be supported by experts and existing documentation. At this point we reconstruct the actual connections between the ports of the components. This can be done either semi-automatically, by navigating from our C&C ports to the source code, or fully automatically, by analyzing the extracted model with context knowledge about the communication patterns manifestation in code.

**Result Preparation:** Eventually, the recovered information can be presented in a graphical C&C view and in a supporting spreadsheet to communicate, analyze and consolidate the reconstruction results and findings.

## 3 Applications of ReCoV

In the following we briefly describe the application of ReCoV on a Qt [2] based system developed by John Deere and a system based on OSGi [1] .

As described in [3], the John Deere software is part of a display system which is used in agricultural vehicles such as tractors or combines. The code size was approximately 300KLOC. One of the Qt peculiarities is its communication paradigm of "signals and slots" which are means to set up point to point and broadcast connectors. Using the graphical pattern matching and the context knowledge of the Qt signal and slots connectors we automatically recovered 734 possible runtime connections, from which about 15% proofed to be relevant. At first sight, this doesn't seem significant, however, using the tooling and its comprehensible result preparation allowed an efficient filtering of the false positives in a subsequent manual step.

The OSGi based system (120KLOC) uses Remote Procedure Calls to communicate over a network. Here typed interfaces are the communication endpoints, hence our pattern matching and subsequent dependency analysis brought better results and we could fully automatically recover 24 client server connections from which 23 (96%) proofed to be relevant.

The full paper illustrating the approach and the case studies in more detail was published at the CSMR'13 [3] .

### References


[1] http://www.osgi.org/Main/HomePage, OSGi home, Retrieved 02/03/2013

[2] http://qt.nokia.com, Qt home, Retrieved 02/03/2013

[3] Becker, M, Forster T., Keuler T.& Knodel J. (2013) Recovering Component Dependencies Hidden by Frameworks – Experiences from Analyzing OSGi and Qt. 17th European Conference on Software Maintenance and Reengineering (CSMR'13)