Towards Reconstructing Architectural Models of Software Tools by Runtime Analysis

Ian Peake, Jan Olaf Blech, Lasith Fernando

RMIT University, Melbourne, Australia
{ian.peake, janolaf.blech,lasith.fernando}@rmit.edu.au

Abstract. We present a method and initial results on reverse engineering the architecture of monolithic software systems. Our approach is based on analysis of system binaries resulting in a series of models, which are successively refined into a component structure. Our approach comprises the following steps: 1) instrumentation of existing binaries for dynamically generating execution traces at runtime and connected analysis, 2) static inspection of binaries, 3) interpretation using domain knowledge, and 4) identifying component boundaries using software clustering. We motivate a generic method which covers a large class of software systems, and evaluate our method on concrete software tools for industrial automation systems development, focusing on Intel x86 and Microsoft Windows-compatible applications.

1 Introduction

We present an architectural reverse engineering approach. Instead of solely analysing binaries statically, we perform analysis at runtime thereby taking into account runtime dependencies between entities. This detailed dependency information denotes an abstract system model. Clustering is used to identify candidates for a component-based software architecture view suitable for human understanding. Additional information from binary inspection and domain specific knowledge is used to select an architectural system model. Furthermore, in the experimental part of this paper, we apply our method to tools for distributed industrial automation programmable logic control (PLC) specification and configuration. Such tools typically support specifications compliant with the IEC 61131–3 or IEC 61499 standards (e.g. CoDeSys [6] and 4DIAC [8], respectively). Restriction to a particular domain gives us information for calibrating our analysis. In this paper our novel contributions are as follows: 1) A suggested architecture reverse engineering method based on runtime instrumentation, automated clustering, hand-inspection of binaries, and domain knowledge. 2) Tailoring this method for Intel x86, in particular Microsoft Windows. 3) A case-study applying our method to PLC specification and configuration tools.

Related Work Published work on architecture reconstruction and related reverse engineering tasks focussing on derivation of component candidates and inter-dependencies is covered in existing surveys and overview papers [5, 15, 3]. Two main directions are 1) based on analysis of source code and 2) based on the analyses or execution of system binaries. In [3] a taxonomy of reverse engineering techniques by classifying according to the artefacts used, and whether analysis is static (based on syntactic analysis of source or executable) or dynamic (based on running, observing and/or animating the system itself) is presented. We focus on runtime analysis for architecture derivation (also called dynamic analysis) [7]. DiscoTect [16, 17] is a framework that observes running systems to reconstruct their architectures. A key feature of DiscoTect is its flexibility to cope with a range of high architectural styles and a range of possible realizations in implementations. DiscoTect uses a language: DiscoSTEP to define mappings interpreting low level system events as more abstract architectural operations, which are formally defined as coloured Petri Nets. The authors note that such mappings must be provided by experts with correct domain knowledge. Reconstructing software architecture from execution traces requires the analysis of the execution traces and the identification of potential components. Combining potential *component candidates* into disjunct sets denoting suggestions for aggregation of components is known as clustering and is an important step for gaining suggestions on the original and potential future architectures. The field of clustering for software components has been studied by several authors including [10] featuring a proposition, [12] featuring the analysis of source code for component detection, [9] studying clustering in the context of software evolution. In this work we are using the Pin tool [4] for binary instrumentation and tracing hints about architecture. Other well known tools comprise the more heavy weight [13] tool which does not have native Windows support, but offers a wider range of instrumentation possibilities potentially resulting in slower code.

2 Our Approach

Our method for collecting runtime based architecture information has these steps:1) We instrument an existing tool such that dynamically loaded libraries and control flow events are tracked and collated as execution traces. These traces contain information (e.g. available methods) for dynamically loaded libraries, as well as the order of method calls. Instrumentation is done on a binary level. 2) The instrumented tool is run and execution traces are generated. A user interacts with the tool (e.g. editing, simulating, compiling). All dynamically loaded libraries and method calls (traced by memory address) and time of invocation are traced. The generated execution traces are further processed and abstracted. This involves the resolution of traced memory addresses to *primitives* such as methods, objects or executables. Calls between methods denote a graph. Here, each primitive corresponds to a node and the number of distinct caller/callee combinations in the execution trace is annoted as a weight on a directed edge. We cluster primitives into candidate components using the LIMBO algorithm (see below). This gives first candidates for a component architecture. Final clustering is based on interactions between methods, existing dll structure, analysis of names and knowledge about reference architectures. 3) The generated data is interpreted by using information from binaries and the domain, to derive information about the underlying architecture of the tool. Manual binary inspection and domain knowledge are used to complete reconstruction.Several tasks are carried out for the runtime-based analysis part of our method:

Usage Scenarios for Runtime Based Evaluation We evaluate our tools with the help of usage scenarios. These are sequences of user interactions with tools. The component interactions are then extracted from the generated execution trace in order to gain hints on architectural details. The idea is to invoke the distinct components of a tool by user interaction. For example a user may trigger a compilation at a certain time and the execution trace may show the loading of distinct libraries and the invocation of the desired methods. We can also compare interaction sequences in order to see if different tools have a similar way of interacting e.g. with a compilation component.

Evaluating Execution Traces, Clustering and Component Candidate Identification We aim to generate a graphical view showing a few high-level components and their interactions. In Windows and similar environments components are most often associated with distinct executables and libraries (or possibly packages), and their inter-dependencies (associated with dynamic linking, import or transfer of control between their respective methods). However a high-level view at such a level is often inappropriate because the view has either too many or too few executables. We therefore tried to identify high level component candidates by clustering groups of other programming language-specific notions such as classes/object or method. We will use the term *primitives* to refer to low-level component categories selected for clustering.

Software clustering is a long-standing and commonly used method for imposing abstract, high level structure on an over-detailed view of primitives and their relationships. For software, a set of low-level components is typically clustered on the basis of properties such as which other components they call, authorship, or location in source directories. As shown, clustering may be thought of as partitioning a collection of objects based on the similarity of their properties. Typically clustering is based on static analysis, here, we are using clustering based on the dynamic call structure between primitives observed at runtime.

Figures 1 (a) and (b) depict the clustering for a usage scenario in the open source tool PLCEdit [14]. Both take the dynamic call structure between primitives into account and are generated from the same execution trace file. Primitives of each cluster are listed in nodes (boxes). The number of calls between primitives are provided as labels on the edges. The main call direction is given first. Calls in the opposite direction are in parentheses. The figures exemplify that based on the same execution trace files there are different possible ways to depict abstract system structure. Arguably, good component structures are selected based on domain specific knowledge.

We use the LIMBO clustering algorithm [2]. LIMBO is based on a generic method called Agglomerative Information Bottleneck (AIB). It has been used for the analysis of large systems across scientific disciplines. LIMBO and the underlying AIB method are generic in the sense that they operate fundamentally on a set of objects O, a set of attributes A and relation $R \subseteq O \times A$ with non-negative real number weighting $w : O \times A \rightarrow \mathbb{R}^+ \cup \bot$. In our approach we represent primitives as follows. Each primitive is modelled both by an object in O and an attribute in A. The weighting w reflects the number of different ways an object o in O calls a different object o' in A. R and w are constructed from the execution traces in an application-dependent way. LIMBO uses a generic information-theoretic approach as its basis for clustering. First, weights are modified via a suitable weighting transformation such as TF.IDF, which transforms weights according their significance (the more rarely held an attribute A is overall by all objects, and the more frequently by some given object O, the more sig-

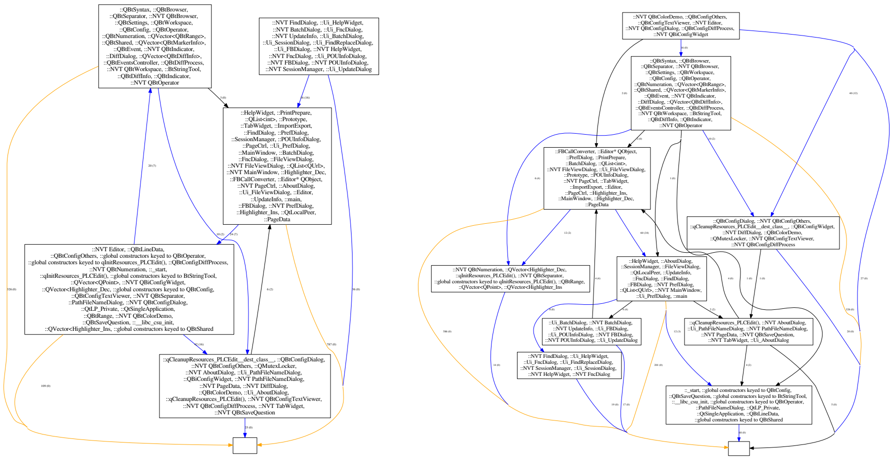
::QBtSyntax, ::QBtBrowser, ::QBtSeparator, ::NVT QBtBrowser, ::QBtSettings, ::QBtWorkspace, ::QBtConfig, ::QBtOperator, ::QBtNumeration, ::QVector<QBtRange>, ::QBtShared, ::QVector<QBtMarkerInfo>, ::QBtEvent, ::NVT QBtIndicator, ::DiffDialog, ::QVector<QBtDiffInfo>, ::QBtEventsController, ::QBtDiffProcess, ::NVT QBtWorkspace, ::BtStringTool, ::QBtDiffInfo, ::QBtIndicator, ::NVT QBtOperator
::NVT FindDialog, ::Ui_HelpWidget, ::NVT BatchDialog, ::Ui_FncDialog, ::NVT UpdateInfo, ::Ui_BatchDialog, ::Ui_SessionDialog, ::Ui_FindReplaceDialog, ::Ui_FBDialog, ::NVT HelpWidget, ::NVT FncDialog, ::Ui_POUInfoDialog, ::NVT FBDialog, ::NVT POUInfoDialog, ::NVT SessionManager, ::Ui_UpdateDialog
3 (0)
16 (16)
::HelpWidget, ::PrintPrepare, ::QList<int>, ::Prototype, ::TabWidget, ::ImportExport, ::FindDialog, ::PrefDialog, ::SessionManager, ::POUInfoDialog, ::PageCtrl, ::Ui_PrefDialog, ::MainWindow, ::BatchDialog, ::FncDialog, ::FileViewDialog, ::NVT FileViewDialog, ::QList<QUrl>, ::NVT MainWindow, ::Highlighter_Dec, ::FBCallConverter, ::Editor* QObject, ::NVT PageCtrl, ::AboutDialog, ::Ui_FileViewDialog, ::Editor, ::UpdateInfo, ::main, ::FBDialog, ::NVT PrefDialog, ::Highlighter_Ins, ::QtLocalPeer, ::PageData
20 (7)
10 (2)
24 (7)
326 (0)
6 (2)
36 (0)
::NVT Editor, ::QBtLineData, ::QBtConfigOthers, ::global constructors keyed to QBtOperator, ::global constructors keyed to qInitResources_PLCEdit(), ::QBtConfigDiffProcess, ::NVT QBtNumeration, ::_start, ::qInitResources_PLCEdit(), ::global constructors keyed to BtStringTool, ::QVector<QPoint>, ::NVT QBiConfigWidget, ::QVector<Highlighter_Dec, ::global constructors keyed to QBtConfig, ::QBtConfigTextViewer, ::NVT QBtSeparator, ::PathFileNameDialog, ::NVT QBtConfigDialog, ::QtLP_Private, ::QtSingleApplication, ::QBtRange, ::NVT QBtColorDemo, ::QBtSaveQuestion, ::__libc_csu_init, ::QVector<Highlighter_Ins, ::global constructors keyed to QBtShared
42 (16)
787 (0)
::qCleanupResources_PLCEdit__dest_class__, ::QBtConfigDialog, ::NVT QBtConfigOthers, ::QMutexLocker, ::NVT AboutDialog, ::Ui_PathFileNameDialog, ::QBiConfigWidget, ::NVT PathFileNameDialog, ::NVT PageData, ::NVT DiffDialog, ::QBtColorDemo, ::Ui_AboutDialog, ::qCleanupResources_PLCEdit(), ::NVT QBtConfigTextViewer, ::NVT QBtConfigDiffProcess, ::NVT TabWidget, ::NVT QBtSaveQuestion
109 (0)
25 (0)
::NVT QBtColorDemo, ::QBtConfigOthers, ::QBtConfigTextViewer, ::NVT Editor, ::NVT QBtConfigDialog, ::QBtConfigDiffProcess, ::NVT QBiConfigWidget
16 (0)
2 (0)
40 (12)
::QBtSyntax, ::QBtBrowser, ::QBtSeparator, ::NVT QBtBrowser, ::QBtSettings, ::QBtWorkspace, ::QBtConfig, ::QBtOperator, ::QBtNumeration, ::QVector<QBtRange>, ::QBtShared, ::QVector<QBtMarkerInfo>, ::QBtEvent, ::NVT QBtIndicator, ::DiffDialog, ::QVector<QBtDiffInfo>, ::QBtEventsController, ::QBtDiffProcess, ::NVT QBtWorkspace, ::BtStringTool, ::QBtDiffInfo, ::QBtIndicator, ::NVT QBtOperator
2 (0)
10 (2)
::FBCallConverter, ::Editor* QObject, ::PrefDialog, ::PrintPrepare, ::BatchDialog, ::QList<int>, ::NVT FileViewDialog, ::Ui_FileViewDialog, ::Prototype, ::POUInfoDialog, ::NVT PageCtrl, ::TabWidget, ::ImportExport, ::Editor, ::PageCtrl, ::Highlighter_Ins, ::MainWindow, ::Highlighter_Dec, ::PageData
6 (4)
1 (0)
::QBtConfigDialog, ::NVT QBtConfigOthers, ::qCleanupResources_PLCEdit__dest_class__, ::QBiConfigWidget, ::NVT DiffDialog, ::QBtColorDemo, ::QMutexLocker, ::NVT QBtConfigTextViewer, ::NVT QBtConfigDiffProcess
12 (2)
60 (34)
::NVT QBtNumeration, ::QVector<Highlighter_Dec, ::qInitResources_PLCEdit(), ::NVT QBtSeparator, ::global constructors keyed to qInitResources_PLCEdit(), ::QBtRange, ::QVector<QPoint>, ::QVector<Highlighter_Ins
4 (4)
::HelpWidget, ::AboutDialog, ::SessionManager, ::FileViewDialog, ::QtLocalPeer, ::UpdateInfo, ::FncDialog, ::FindDialog, ::FBDialog, ::NVT PrefDialog, ::QList<QUrl>, ::NVT MainWindow, ::Ui_PrefDialog, ::main
4 (0)
1 (0)
1 (0)
27 (0)
8 (8)
4 (4)
2 (2)
326 (0)
::Ui_BatchDialog, ::NVT BatchDialog, ::NVT UpdateInfo, ::Ui_FBDialog, ::Ui_POUInfoDialog, ::NVT FBDialog, ::NVT POUInfoDialog, ::Ui_UpdateDialog
::qCleanupResources_PLCEdit(), ::NVT AboutDialog, ::Ui_PathFileNameDialog, ::NVT PathFileNameDialog, ::NVT PageData, ::NVT QBtSaveQuestion, ::NVT TabWidget, ::Ui_AboutDialog
586 (0)
12 (3)
20 (0)
::NVT FindDialog, ::Ui_HelpWidget, ::Ui_FncDialog, ::Ui_FindReplaceDialog, ::NVT SessionManager, ::Ui_SessionDialog, ::NVT HelpWidget, ::NVT FncDialog
14 (0)
201 (0)
4 (2)
::_start, ::global constructors keyed to QBtConfig, ::QBtSaveQuestion, ::global constructors keyed to BtStringTool, ::__libc_csu_init, ::global constructors keyed to QBtOperator, ::PathFileNameDialog, ::QtLP_Private, ::QtSingleApplication, ::QBtLineData, ::global constructors keyed to QBtShared
19 (0)
17 (0)
5 (0)
58 (0)

Fig. 1: Example control flow graphs for 5 and 10 components

nificant, thus heavily weighted, A is for O.) Next, the new weights are converted to probabilities such that the sum of all weights per object is 1. Finally, LIMBO attempts to compress its representation of R by iteratively merging the *closest* pair of objects and generating a new relation R' which approximates R under merging. The closest pair is the one for which merging minimises information loss in R'. LIMBO's genericity enables it to support both "structural" and "non-structural" attributes. Structural attributes reflect program dependence structure as described above. In our work so far clustering is purely on a structural basis. Non-structural attributes refer to the general case and cover properties such as a time stamp or authorship. There is ambiguity about whether it is best to generate structural attributes by interpreting the primitive call graph as directed or undirected—That is, whether two primitives which call each other have the same value in both directions (sum of the number of ways they can call each other) or possibly-distinct values. In our work the call graph is interpreted as undirected.

Additional Static and Domain Specific Information Binaries like .dll files can encapsulate multiple components and provide hints on development history. Names and size of components can indicate usage. Binaries can contain method names and plain text that hint on component functionality. A major source of knowledge in our reverse engineering method is the PLC development domain. For example we know what types of components to expect. We started with the following expected components: *Source and target code storage* manage the modeling, storage and exchange of source and target specification models and code by using a file system or a database. *Compilers, Analyzers and Simulators* parse specification models and perform operations on them, like generating target code, interacting with a GUI component in order to visualize behaviour or properties. *Editors* manage the editing of models by the user. *License Management* and other miscellaneous functionality can be realized as a separate component e.g. that may interact with a third party license server. The *GUI* provides a user interface. It does not have to be realized as a separate component inside the tool, since existing GUI frameworks can be used.

3 Analysis and Evaluation

Instrumentation of binaries is done by using the Intel Pin tool [4]. We instrument the binaries of our analyzed tools to extract: (i) A list of the loaded binaries and the names of the methods (called routines) inside these binaries, if available, including their memory addresses. (ii) A list of control flow operations that occurred during the execution of the tool, and in particular the source and destination addresses.

Case Study Tools We have used our method for analysing the architecture of a mix of proprietary and open source tools. Tools are designed for performing at least some of the following operations for the development of PLC software: 1) Editing PLC specification models, 2) Saving and loading of PLC specification models, 3) Analysing and compiling PLC specification models, 4) Simulating PLC specification models. We initially expected that this functionality is provided by distinct software components as described in the previous section. Open source systems considered included PLCEdit [14], Beremiz [1] and MATIEC [11]. Of these, PLCEdit and MATIEC (also a Beremiz component) were immediately suitable, consisting of plain binary executables. Beremiz is written in Python and thus the Pin-based method is not immediately suitable.

Example Runs An Example usage scenario (Section 2) consists of the steps: Start tool; Create new project file; Add ladder diagram; Invoke editor; Add coil (lamp) and contact (switch) to ladder diagram, add connections; Save project; Compile and check project; Close project; Start simulation of saved project; Close tool.

Evaluation and Improvements The method was applied to different PLC development tools. Execution of the usage scenarios was done manually, while the processing of the execution traces was done automatically to generate models – one single model for each usage scenario and number of desired components – comprising component candidates and their interactions. Clustering based on dynamically linked libraries and executables did not always provide the right granularity, since several major components are typically encapsulated in a main executable. Determining the begin and end of entities like methods or classes in the binaries as a basis for clustering was sometimes possible. In some cases e.g. due to the use of different programming languages, additional information on the location of entities for the basis of clustering was provided by the tool developers and used by us. For example for some applications while symbol table information is not available in the executable, a ".map" file provides similar information for debugging purposes. There are a number of possible reasons why a clustering may not reflect a system's true architecture, for example there may be insufficient data in the run time call graph, or architectural anti-patterns may be present. It may be desirable to associate each component with a meaningful name or feature. As discussed in clustering literature, this depends on understanding what abstractions (e.g. aspects) are semantically common to all objects of a component, or the principle abstraction of the component, which can be difficult. Currently all attributes are structural, derived from the Pin call graph, however the graph is created by exercising tools using just a few use case scenarios. There is scope to assign so-called non structural attributes, that is, properties other than call relationships on which clustering could be based, possibly based on manual assessment and with input from domain experts. These could pertain to specific features or aspects such as GUI or safety. For example if several objects are clearly GUI components, an additional GUI attribute could be assigned to those objects and taken into account during clustering. There are existing aspect mining approaches in the literature which may be applicable or adaptable to this purpose.

The LIMBO algorithm was implemented by us in few hundred lines of Python. This is supported by additional scripts which process output from our pin plugin.

References

1. Beremiz IDE. Version 1.1 RC3. Downloaded from `beremiz.org` (July 2013)
2. Periklis Andritsos and Vassilios Tzerpos. Information-Theoretic Software Clustering. In IEEE Trans. on Software Eng., 31(2): 150-165 (2005)
3. Gerardo Canfora, Massimiliano Di Penta, Luigi Cerulo: Achievements and challenges in software reverse engineering. Commun. ACM 54(4): 142-151 (2011)
4. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Programming Language Design and Implementation (PLDI), Chicago, IL (June 2005)
5. Elliot J. Chikofsky, James H. Cross II: Reverse Engineering and Design Recovery:A Taxonomy. IEEE Software 7(1): 13-17 (1990)
6. CoDeSys — industrial IEC 61131-3 PLC programming: `www.codesys.com`
7. Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, Rainer Koschke: A Systematic Survey of Program Comprehension through Dynamic Analysis. IEEE Trans. Software Eng. 35(5): 684-702 (2009)
8. 4DIAC IDE. Version 1.3: `fordiac.org` (Accessed July 2013)
9. Rainer Koschke. Atomic architectural component recovery for program understanding and evolution. Software Maintenance (2002).
10. Chung-Horng Lung. Software Architecture Recovery and Restructuring through Clustering Techniques. 3rd International Software Architecture Workshop (ISAW): 101-104 (1998)
11. MATIEC compiler. Source from `bitbucket.org/mjsousa/matiec` (July 2013)
12. Brian S. Mitchell, Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. Software Maintenance (2001)
13. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN conference on Programming language design and implementation (2007)
14. PLCEdit Editor. Version 2.1.1. Downloaded from `www.plcedit.org` (July 2013)
15. Damien Pollet, Stéphane Ducasse, Loïc Poyet, Ilham Alloui, Sorana Cîmpan, Hérve Verjus: Towards A Process-Oriented Software Architecture Reconstruction Taxonomy. Conference on Software Maintenance and Reengineering: 137-148 (2007)
16. Hong Yan, David Garlan, Bradley R. Schmerl, Jonathan Aldrich, Rick Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. International Conference on Software Engineering: 470-479 (2004)
17. Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman and Hong Yan. Discovering Architectures from Running Systems. IEEE Trans. Software Eng. 32(7): 454-466 (2006)