COORDINSPECTOR: a tool for extracting coordination data from legacy code

Nuno F. Rodrigues
DI-CCTC, Universidade do Minho
4710-057 Braga, Portugal
Email: nfr@di.uminho.pt

Luis S. Barbosa
DI-CCTC, Universidade do Minho
4710-057 Braga, Portugal
Email: lsb@di.uminho.pt

***Abstract*—More and more current software systems rely on non trivial coordination logic for combining autonomous services typically running on different platforms and often owned by different organizations. Often, however, coordination data is deeply entangled in the code and, therefore, difficult to isolate and analyse separately.**

COORDINSPECTOR is a software tool which combines slicing and program analysis techniques to isolate all coordination elements from the source code of an existing application. Such a reverse engineering process provides a clear view of the actually invoked services as well as of the orchestration patterns which bind them together.

The tool analyses Common Intermediate Language (CIL) code, the native language of Microsoft .Net Framework. Therefore, the scope of application of COORDINSPECTOR is quite large: potentially any piece of code developed in any of the programming languages which compiles to the .Net Framework. The tool generates graphical representations of the coordination layer together and identifies the underlying business process orchestrations, rendering them as Orc specifications.

I. MOTIVATION

The ubiquity of software and its exponential growth, both in size and complexity, is producing an equally growing amount of legacy code that has to be maintained, improved, replaced, adapted and accessed for quality every day. Paradoxically, in a situation in which the only quality certificate of the running software artifact still is life-cycle endurance, customers and software producers are little prepared to modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence - i.e. the level of understanding of - on their code. Moreover, software quality, requiring systems to comply to strict and specific quality standards, and conformance of design specifications with the actual implementations is impossible to be assessed without rigorous models of running systems.

Popular expressions, as, for example, *program understanding*, *reverse engineering* and *model extraction*, were coined in such a context. They refer to a broad range of techniques to extract specific knowledge from legacy code, represent it in malleable representations, proceed to their analysis, classification and reconstruction.

The research reported in this paper is supported by FCT, under contract POSC/EIA/56646/2004, in the context of the IVY project.

The tool described here is actually a tool for *program understanding*, but designed to capture the underlying coordination patterns [1]. Therefore, it addresses the macro level of *software architecture* [2] identifying patterns of *interactions* between distinct architectural elements.

Several approaches have been proposed for reverse architectural analysis. Among them *Class Diagram* generators which extract class diagrams from object oriented source code, *Module Diagram* generators that construct box-line diagrams from system's modules, packages or namespaces, *Uses Diagram* generators which reflect the import dependencies of the system and *Call Diagram* generators which expose the direct calls between system parts.

However, none of these techniques/tools makes it possible to answer questions which are about the *dynamics* of a software system. For example, *how does it interact with its own components and external services and coordinate them to achieve its goals? How does the system behave when trying to access an ilexternal resource unavailable? Can the system enter in a deadlock situation? and what is the sequence of actions for such a deadlock to take place?*, among others.

The reason why such questions cannot be answered from most of the models built from code extraction, is that behavioural analysis requires a level of abstraction which, for a number of reasons, is not common in such models. Actually, disentangling *coordination data* and recovering a system model able to capture its behaviour with respect to the interactions with different components, is a complex process, as it deals with multiple activities and multiple participants which in turn are influenced by multiple constrains, such as exceptional situations, interrupts and failures.

II. COORDINSPECTOR: THE TOOL

COORDINSPECTOR[1], a snapshot of which is presented in Fig. , is a prototype of a software analysis, which aims at providing answers to such questions. By extracting the coordination model of a system from its source code described in Microsoft *Common Intermediate Language* (CIL). Note that CIL is the language interpreted by the .Net Framework for which every Microsoft .Net compliant language compiles to. Therefore, COORDINSPECTOR is able to analyze heterogenous

[1]The tool is available from http://www.di.uminho.pt/∼nfr

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.10

IEEE computer society

systems implemented in multiple languages within the set of .Net Framework languages, which by now counts more than 40 languages[2], and this number has only but potential to increase. Such a decision of targeting CIL code was not an arbitrary one. Indeed we intended the tool to be able to cope with as many programming languages as possible, because most real world software systems are developed in more than one language. Moreover, given the potential of the tool to assist legacy systems evolution, the "language agnostic" feature became an important invariant.

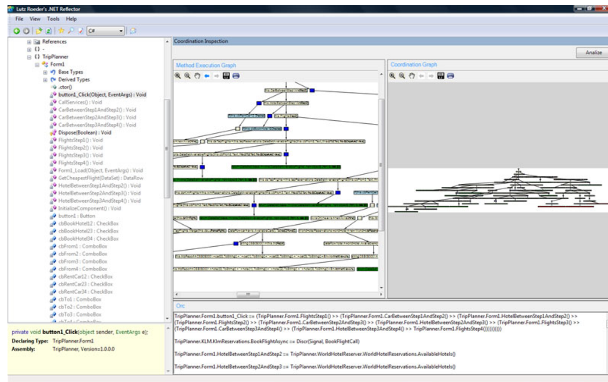

Fig. 1. COORDINSPECTOR

By analyzing a system's CIL code, COORDINSPECTOR performs the following steps to abstract the system's coordination model:

- Constructs the Control Flow Graph (CFG) for each property, constructor, method, anonymous delegate or lambda expression in the CIL
- Based on each CFG, COORDINSPECTOR builds the Method Dependence Graph (MDG)[3] for each of the previous program entities.
- By inspecting the class where each of the previous entities are defined, the tool constructs the Class Dependence Graph, and follows a similar approach in the construction of the Interface Dependence Graph and Namespace Dependence Graph.
- Given the Managed System Dependence Graph MSDG (constructed upon the previous graphs), the tool is constructs the system Coordination Dependence Graph (CDG).
- COORDINSPECTOR is also able to visually represent both the MSDG and CDG of the system as well to navigate over these graph structures. This graphical presentation of the graphs provides the user with specific vertex information, like labeling and the CIL code captured, by applying a double click on a particular vertex of the graph.
- Finally the tool generates the Orc coordination of the system by analyzing the previously calculated CDG.

III. COORDINSPECTOR: THE IMPLEMENTATION

The implementation of COORDINSPECTOR combines a number of techniques from program analysis and graphical representation of code, with a particular emphasis on slicing. This builds on top of the first author previous work on program slicing [4], the techniques of which were found applicable to the sort of 'coordination data extraction' problem addressed in this tool. The whole approach is described in detail in an accompanying paper submitted to this Conference.

In order to take advantage of existing CIL analysis tools, COORDINSPECTOR is developed as a plug-in for the CIL decompiler .Net Reflector[4], from where it just retains the parse for CIL code, which delivers an object tree representation of the CIL concrete syntax tree. Such tree is then processed to build a dependence graph which extends the original definition in [3] with support for objects and new representations for concurrent constructs and specific managed code (i.e., code that executes under the management of the Common Language Runtime virtual machine) details.

The computation of this graph is parametric on the coordination features one wants to trace back in the code. At the moment of writing the tool is equipped with sets of rules to deal with web services communications, distinguish between synchronous and asynchronous calls as well as between invocation and provisioning of functionality using web services. Other sets of rules can, however, be easily added.

It is well known that the kind of algorithms in which tools like COORDINSPECTOR are based lead to highly time-consuming implementations, given input size and computational complexity involved. Therefore, we have adopted a distributed strategy based on multithreading sub-graph calculation, which reduced the MSDG calculation time to roughly on third of the original time.

Finally, it should be noted that code generation by COORDINSPECTOR was not implemented as a syntax-directed operation, but, instead, by an extension of the same graph traversal operations that were defined for the labeling process in the construction of the dependence graph.

IV. CONCLUSIONS AND FUTURE WORK

REFERENCES

[1] W. M. P. V. D. Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.

[4] N. F. Rodrigues and L. S. Barbosa. Higher-order lazy functional slicing. *Journal of Universal Computer Science*, 13(6):854–873, jun 2007.

[2]Source: http://en.wikipedia.org/wiki/CLI_Languages

[3]See the accompanying paper "Discovering orchestration patterns in legacy code", also submitted to SCAM'08

[4]http://www.aisto.com/roeder/ dotnet