The Role of Coordination Analysis in Software Integration Projects*

Nuno F. Rodrigues[1,2], Nuno Oliveira[2], and Luís S. Barbosa[2]

[1] DIGARC - Polytechnic Institute of Cávado and Ave,
4750-810 Barcelos, Portugal
[2] DI-CCTC - Universidade do Minho,
4710-057 Braga, Portugal

Abstract. What sort of component coordination strategies emerge in a software integration process? How can such strategies be discovered and further analysed? How close are they to the coordination component of the envisaged architectural model which was supposed to guide the integration process? This paper introduces a framework in which such questions can be discussed and illustrates its use by describing part of a real case-study. The approach is based on a methodology which enables semi-automatic discovery of coordination patterns from source code, combining generalized slicing techniques and graph manipulation.

1 Introduction

Integrating running software applications, usually referred in the literature as the *Enterprise Application Integration* (EAI) problem [5,3], is one of most challenging tasks in enterprise systems development and management. According to Forrester Research, more than 30% of all investments made in information technologies are spent in the linkage of software systems in order to accomplish global coherent enterprise software solutions. Actually, tuning to new markets, fusion or acquisition of companies, evolution of legacy software, are just but examples of typical scenarios which entail the need for integration.

EAI aims at the smooth composition of services, data and functionality from different software systems, to achieve a single, integrated and coherent enterprise solution. Conceptually, however, a main issue behind most EAI projects concerns the definition and implementation of a specific *coordination model* between the systems being integrated. Such a model is supposed to capture system's behaviour with respect to its network of interactions. Its role is fundamental to help the software architect to answer questions like, which sub-systems are connected, how do they communicate and under which discipline, what are the dependencies between such connections, how do component's (w.r.t to integrating software systems) local constraints scale up to integrated systems, among many others.

* This research was partially supported by FCT in the context of the MONDRIAN project, under contract PTDC/EIA-CCO/108302/2008.

R. Meersman, T. Dillon, and P. Herrero (Eds.): OTM 2011 Workshops, LNCS 7046, pp. 83–92, 2011.

If some sort of coordination strategy is a necessary component of any EAI project, what gets implemented on completion of the integration project often deviates from the envisaged strategy in a significant way. Reconstructing the implemented strategy and rendering it into a suitable model becomes, therefore, an important issue in EAI. On the one hand, such a reconstructed model plays a role in validating the (coordination component) of the integration process. On the other, it provides feedback to the software architect, eventually suggesting alternative strategies.

Such is the problem addressed in this paper. In a series of recent papers the authors have developed both a methodology [8,10] and a tool, CoordInspector [9] to extract, from source code, the structure of interactions among the different, reasonably independent loci of computation from which a system is composed of. The target of this methodology is what will be referred to in the sequel as the *coordination layer*, i.e. the architectural layer which captures system's behaviour with respect to its network of interactions[1]. The extraction methodology combines suitable slicing techniques over a family of *dependence graphs* built directly from source code, in the tradition of *program dependence graphs* (see, for example, [1,4]), which abstract all the relevant code information for interaction control. Such graphs are called *coordination dependence graphs* (CDG) in [10], where further details on their construction may be found. The tool processes CIL code, for which every .Net compliant language compiles to, thus making it potentially able to analyse systems developed in (combinations of) more than 40 programming languages.

The paper introduces a case study on verification of design time integration strategies. Section 2 sums up the methodology, and examples of its application are discussed in 3. Conclusions and pointers for future work are enumerated in section 4.

2 The Method

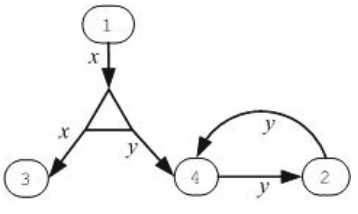
2.1 Coordination Patterns

Throughout this paper we adopt a *coordination-driven view* of software architecture in general and architectural analysis, in particular. On the other hand, *patterns*, in the context of this research, are placed at a low-level: they aim to be suitable representations of equivalence classes of (sub-graphs of) CDG extracted from code. Qualifier 'low-level' means that our focus is not on the description of traditional architectural styles, or even typical architectural elements, such as components, software buses or connectors, but the specification of architectural abstractions over dependence graphs extracted from code.

As an example consider the pattern depicted in Fig. 1 used to identify, in the client side of a service interaction, the so-called *asynchronous query pattern*

[1] The qualifier is borrowed from research on *coordination* models and languages [2], which emerged in the nineties to exploit the full potential of parallel systems, concurrency and cooperation of heterogeneous, loosely-coupled components.

with client multithreading. It is described in the graphical notation associated to CoordL, a domain-specific language for coordination patterns introduced by the authors in [6]. The corresponding textual version is shown in the right in a window of CoordInspector. Even if it is not the aim of this paper to provide a full description of CoordL, its graphical notation is almost self-explicative. For the moment, and to get a flavour of what coordination patterns are, note in this example how it encodes the following protocol: a client orders the execution of an operation in one thread, x, and then launches a second thread, y, to retrieve the result. Instances of this pattern are common whenever time consuming services are to be invoked and calling threads can not suspend until a response is returned.



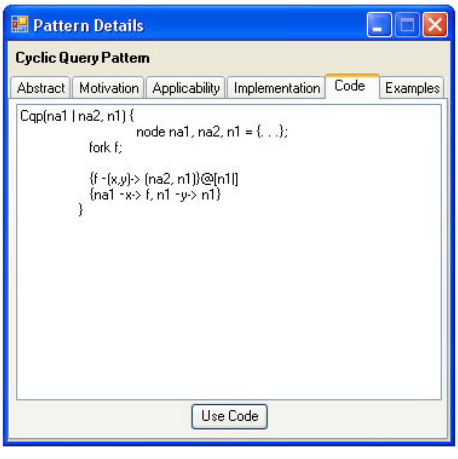


Fig. 1. Asynchronous query pattern with client multithreading

2.2 Discovering Coordination Patterns

Communication primitives, understood in the broad sense of any possible mechanism a component resorts to interact with another one, are the building blocks of the coordination layer of a software system. Direct foreign calls to well referenced components such as web-services calls, RMI or .Net Remoting calls to distributed objects are typical examples but, by no means, the only ones. The specific combinations of such primitives is what allows systems to control and interact, in complex ways, with other systems, processes, databases and services in order to achieve common goals. Thus, it is reasonable to expect that any coordination discovery strategy should start by identifying such primitive communication statements in the source code, together with the specific program context in which they are embedded. Therefore, our approach is *parametric* on the communication primitives as well as on the mode they are invoked (synchronous or asynchronous).

The reverse engineering process starts by the extraction of a comprehensive dependence graph from source code, the *Managed System Dependence Graph* (MSDG), which captures program statements in the vertices while the edges represent data, control and inter-thread dependencies. Then MSDG vertices containing primitive communication calls in their statements are singled out. We call this operation the *labelling* phase which is parametric on both the communication primitives and the calling mode. The result of this phase is another graph structure, the CDG, retaining only coordination relevant data with respect to the set of rules specifying the communication primitives to look for. The CDG is computed from the MSDG in a two stage process. First, nodes matching rules encoding the use of specific interaction or control primitives are suitably labelled. Then, by backward slicing, the MSDG is pruned of all sub-graphs found irrelevant for the reconstruction of the program coordination layer. Once the CDG has been generated, it is used to search for specific coordination patterns and trace them back to source code.

2.3 The Strategy

Recall the questions proposed in the introduction to this paper: *What sort of coordination strategies emerge in a software integration process? How close they are to the coordination component of the envisaged architectural model which was supposed to guide the integration process?*. This paper focus in *post-integration analysis*; thus on a verification of the coordination models developed before the integration process against the final, integrated system. Often, however, such models are only informally recorded, and relevant information scattered among documents describing the integration architecture.

The envisaged strategy has 4 stages. First the basic coordination solutions which were designed for the integration project have to be identified, by analysing the relevant documentation and, often, by interviewing the development team. In each case, a correspondent *coordination pattern* is specified in the diagrammatical notation of CoordInspector. At a third stage, such patterns are looked for in the source code of the integrated system, with the support of CoordInspector.

Often, however, patterns are discovered only in an incremental way. The strategy is to start the search with the pattern as described by the development team and, if it not directly found, split it into small patterns until a match is found. Then work on the reverse direction, re-building the shape of the patterns which is actually implemented. In the limit, the graph pattern is reduced to a unstructured collection of nodes (corresponding, e.g. to web-service calls) which the architect has to aggregate in patterns, in an iterative way. Actually, often what has been recorded, or what developers report, as the envisaged coordination policy is far from what is effectively implemented. Our approach helps in identifying and documenting such cases and also, once they have been retrieved, to discuss possible alternatives, eventually leading to improvements in the implementation. Such critical analysis is the fourth stage of our strategy.

Technically, coordination patterns are defined in a specific notation [6]. For a brief explanation consider Fig. 2 . A pattern is a graph, where upwards triangles

represent the spawning of a new thread, downwards triangles denote a thread join, vertices contain predicates (composed by regular expressions) and edges represent control flow dependencies, both to be verified against a CDG. Edges bare two different types of labels, one capturing the number of edges to be matched on the CDG control flow edges (as in Fig. 2, this label can take the value +, standing for one or more edges), and a second label containing a variable to be bound to the thread id being matched on the CDG. This last label type avoids having to impose an order on the outgoing (incoming) thread edges of an upwards (downwards) triangle, which facilitates the graphical layout of the patterns.

3 A Case-Study in Architectural Analysis

This section illustrates the use of CoordInspector and coordination patterns in a particular case-study in architectural analysis. To respect space limits the real case-study is merely glimpsed here and a small detail taken for illustrative purposes.

The case-study, however, was a big one, dealing with the need to re-engineering a set of independent, but enduring software systems to achieve an effective degree of integration. It involved a Portuguese professional training company, with facilities in six different locations (see [8] for a complete description).

Before the integration project, the company relied in four main software systems, to be referred in the sequel as the *base components*. They comprised an *Enterprise Resource Planner* (ERP), a *Customer Relationship Management* (CRM), a *Training Server* (TS), and a *Document Management System* (DMS). The decision to integrate all these systems was pushed by the necessity of introducing a Web Portal, for on-line selling of both training courses and networking devices. Thus, the final system included these four existing components, plus the Web Portal developed during the integration project.

All those components operated in complete isolation from each other. Thus, every exchange of information between them was performed manually. The absence of integration led to numerous information synchronisation problems which had to be dealt with manually, at a daily basis. A sudden growth in the company business level, made it no longer feasible to maintain all the information synchronised manually. Actually, several incoherencies in critical company data inevitably started to emerge. When the integration project was launched an administrative decision forced the choice of a point-to-point integration architecture.

Although the case-study encompassed the whole integration process we will concentrate our attention here on a specific problem related to consistent data update. The problem resided in the Web Portal application, the component which was laid responsible for the user's data update across all systems.

The re-engineering process started with an attempt to recover from the relevant components source code all the coordination protocols governing user's data update.

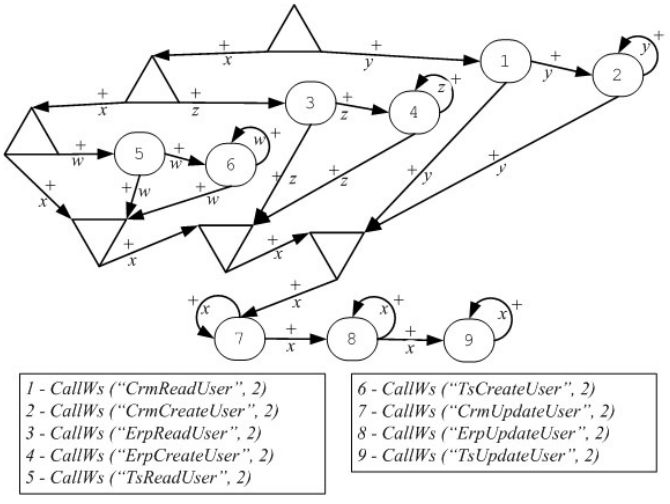


Fig. 2. User update operation

Fig. 2 depicts a common coordination pattern for updating user's data across applications, found recurrently in the system. This patterns however is not able to prevent the creation of already registered users (which in the generic description of the pattern would be translated to recurrent invocations of nodes 2, 4 and 6). To prevent this, the pattern can be changed by inserting extra vertices for checking if a user already exists in the relevant data base component. However, the calls to the remote creation operation (nodes 2, 4 and 6) are always carried after a read operation (nodes 1, 3 and 5), which forces the first remote call to the former to be aware of an eventual previous registration of the user. Thus, only the subsequent remote creation calls (executed through loop edges $2 \rightarrow 2$, $4 \rightarrow 4$ and $6 \rightarrow 6$) suffer from the problem of inserting duplicate users. Therefore, a small change is enough: insert user existence check nodes (nodes 3, 6 and 9 in Fig. 3) after each remote creation call.

The remote update of a user is only performed at the very end of this coordination pattern, in nodes 7, 8 and 9 (Fig. 2) or 10, 11 and 12 (Fig. 3). Moreover, all updates occur in a single thread. This opens the possibility of a previous call introducing delays in subsequent calls, resulting in significant delays for the overall remote operation. This single thread sequence of remote calls also demands for a rigorous exception and error handling, given that each call may influence subsequent ones and consequently the entire operation. In this case-study, once the pattern was discovered, manual inspection of error handling routines was required, because these mechanisms were, then, not incorporated in the CoordL pattern language.

Once identified, this pattern was improved by replacing the sequence of update calls by their parallel execution, as represented in Fig. 3. This potentially minimised the effect of delays introduced by individual calls. A possibility remains, however, for a remote call to continually fail. In such a case, this pattern

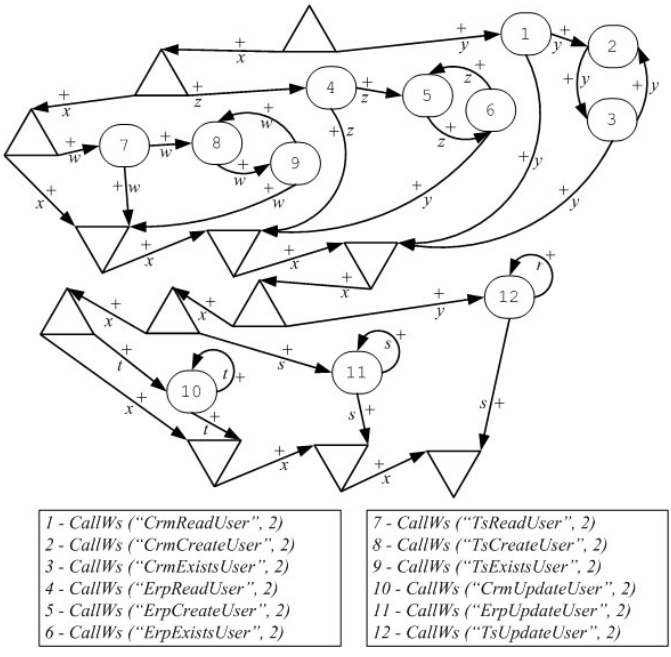


Fig. 3. Corrected user update operation

may not only fail, but, what is worse, deadlock, which could ultimately lead to a complete halt of the entire ECS system. Note that the discovery of such deadlock situations was made easily by using the discovery algorithm to look for loop patterns.

A solution to avoid deadlocks consists of introducing a counter for each identified loop and include a guard (or extend one, if a guard was already there) in the loop to inspect the number of cycles performed. In case one of these loops has overcome the maximum number of cycles allowed, the guard not only guarantees that the program control leaves the loop, but also that the operation not carried out is written to an *error log* table. The deadlock removal strategy introduces a mechanism for error recovery and enables the introduction of different amounts of tries for each remote call. Furthermore, the *error log* table can be used, for instance, to run periodically a *batch* job responsible for the re-invocation of failed operations.

What is important to underline at this stage is the method which lead to this new protocol. First the problem was clearly identified by recovering, from source code and at different components, the pattern corresponding to the protocol in use. Then CoordInspector was used again to identify the associated loops, source of possible deadlocks. Finally the knowledge gathered along the analysis process was used to design a new solution, encode it as a new coordination pattern and its integration back into the repository.

Another example of how effective this approach is is provided by coordination protocol for the online sale of a set of training courses. The pattern actually extracted with CoordInspector is depicted in Fig. 4. Without entering in detail, it is easy to recognise a purely sequential flow of activities.

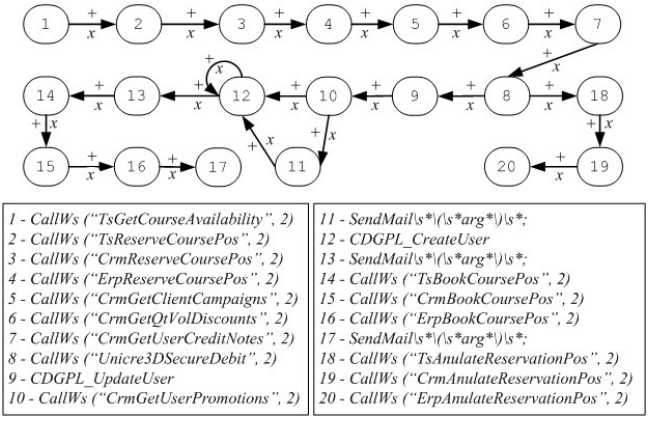


Fig. 4. Training courses sale operation

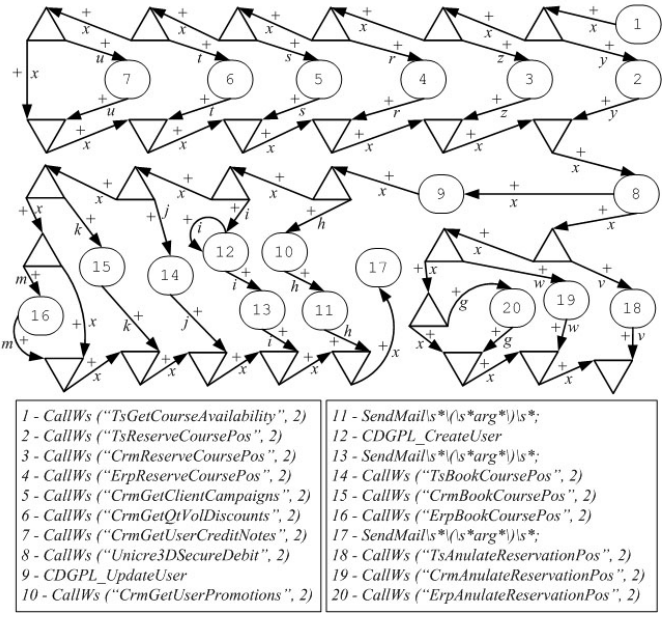


Fig. 5. Improved training courses sale operation

Note that, even though the user update and create operations are multi-threaded, the actual sale operation is entirely performed on a single thread. This, of course, degrades performance, while it would be possible to execute in parallel several activities that do not depend on each other.

Therefore, the system's integrator has proposed the modification of this pattern as depicted in Fig. 5. The main modification concerns the introduction of concurrency between every independent activity. However, some activities still have to be performed in sequence. For example, vertex 9 is executed after vertex 8 because updating (in the former vertex) depends on the completion of the payment operation contained in the latter.

4 Conclusions and Future Work

The need for methods and tools to identify, extract and record the coordination layer of software systems is becoming more and more relevant as an increasing number of software systems rely on non trivial coordination logic for combining autonomous services, typically running on different platforms and owned by different organisations. Actually, if coordination policies can be extracted from source code and made explicit, it becomes easier to understand the system's emergent behaviour (which, by definition, is the behaviour which cannot be inferred directly from the individual components) as well as to verify the adequacy of the software architecture (and of the code itself) with respect to expected interaction patterns and constraints.

This seems particularly relevant in the context of software integration projects, as discussed in the paper. We proposed a tool-supported methodology for recovering, from source code, the coordination strategies *as-implemented* in a software integration project. The whole process is driven by the views of such strategies *as-stated* in the project documentation, often in an informal and vague way. The case study illustrates further how this sort of analysis can be useful in validating design decisions, taken during the integration stage, eventually improving the final system or, at least, providing a precise documentation of its (coordination) architecture.

In the case study reported here, we were able to detect and correct several coordination problems even before these have showed any evidences of themselves as runtime errors or data inconsistencies between the integrated components. Moreover, a number of subtle, yet important, coordination mistakes (e.g., the wrong order of execution of two statements) were detected, that would be much more difficult to discover by manual code inspection.

To the best of our knowledge, this line of enquiry in reverse architectural analysis, focused on coordination issues, is largely unexplored. There is, however, a lot of work on automatic gathering and registering of architectural information understood in a more classical way as the *system's gross structure*. Among others, the Alborz system [11] and the DiscoTect platform [12] should be mentioned. Alborz presents the architecture as a graph of components and keeps a relationship between this graph and the actual source code. The same

applies to BAUHAUS [7], a system which maps architectures to graphs whose nodes may represent types, routines, files or components and the edges model relationships between them, from which different architectural views can be generated. A detailed comparison with our own, largely complementary work is still lacking. Future developments of this work include a number of improvements to COORDINSPECTOR, namely the inclusion of the possibility of directly modifying the implementation code by editing the sub-graphs identified in the analysis process and a performance study concerning different integration scenarios.

References

1. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
2. Gelernter, D., Carrier, N.: Coordination languages and their significance. Communication of the ACM 2(35), 97–107 (1992)
3. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
4. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI 1988: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation, pp. 35–46. ACM Press (1988)
5. Linthicum, D.S.: Enterprise application integration. Addison-Wesley Longman Ltd., Essex (2000)
6. Oliveira, N., Rodrigues, N., Henriques, P.R., Barbosa, L.S.: A pattern language for architectural analysis. In: 14th Brazilian Symposium in Programming Languages, SBLP 2010, Slavador, Brasil, vol. 2, pp. 167–180. SBC — Brazilian Computer Society (September 2010) ISSN: 2175-5922
7. Raza, A., Vogel, G., Plödereder, E.: Bauhaus - a Tool Suite for Program Analysis and Reverse Engineering. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 71–82. Springer, Heidelberg (2006)
8. Rodrigues, N.F.: Slicing Techniques Applied to Architectural Analysis of Legacy Software. PhD thesis, Escola de Engenharia, Braga, Portugal (2008)
9. Rodrigues, N.F., Barbosa, L.S.: Coordinspector: a tool for extracting coordination data from legacy code. In: SCAM 2008: Proc. of the Eighth IEEE Inter. Working Conference on Source Code Analysis and Manipulation, pp. 265–266. IEEE Computer Society (2008)
10. Rodrigues, N.F., Barbosa, L.S.: Slicing for architectural analysis. Science of Computer Programming (March 2010)
11. Sartipi, K., Dezhkam, N., Safyallah, H.: An orchestrated multi-view software architecture reconstruction environment. In: 13th Working Conference on Reverse Engineering (WCRE 2006), Benevento, Italy, October 23-27, pp. 61–70 (2006)
12. Schmerl, B.R., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: Discovering architectures from running systems. IEEE Trans. Software Eng. 32(7), 454–466 (2006)