Discovering coordination patterns

Nuno F. Rodrigues

DI-CCTC, Universidade do Minho
Braga, Portugal

Abstract

A large and growing amount of software systems rely on non-trivial coordination logic for making use of third party services or components. Therefore, it is of outmost importance to understand and capture rigorously this continuously growing layer of coordination as this will make easier not only the verification of such systems with respect to their original specifications, but also maintenance, further development, testing, deployment and integration. This paper introduces a method based on several program analysis techniques (namely, *dependence graphs*, *program slicing*, and *graph pattern analysis*) to extract coordination logic from legacy systems source code. This process is driven by a series of pre-defined coordination patterns and captured by a special purpose graph structure from which coordination specifications can be generated in a number of different formalisms.

Keywords: Program analysis, coordination, orchestration discovery.

1 Introduction

The increasing relevance and exponential growth of software systems, both in size and quantity, is leading to 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 on - i.e. the level of understanding of - 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. This is particularly critical in the emerging *service-oriented* paradigm where non-trivial coordination problems lie at the very heart of applications.

Such is the scenario for the emergence of expressions like *program understanding*, *reverse engineering* and *model extraction*, referring to a broad range of techniques to

[1] Email: nfr@di.uminho.pt

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

extract from legacy code specific and rigorous knowledge, represent it in malleable representations, proceed to their analysis, classification and reconstruction.

The extraction of the entire system's *software architecture* can be considered the ultimate goal in the software reconstruction process. By this we understand, following [2], the set of specific scoped models that expose particular aspects of *parts* (possibly components, modules, processes) of the system and the *interactions* between them.

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 make it possible to answer a critical question about the dynamics of a system: *how does it interact with its own components and external services and coordinate them to achieve its goals*? From a *Call Diagram*, for example, one may identify which parts of a system (and, sometimes, even what external systems) are called during the execution of a particular procedure. No answers are provided, however, to questions like: Will the system try to communicate indefinitely if an external resource is unavailable? If a particular process is down, will it cause the entire system to halt? Can the system enter in a deadlock situation?

It is not surprising that these questions cannot be answered from most of the models built from code extraction, because behavioural analysis is placed at a much higher abstraction level than most of such architectural models. Actually, recovering a *coordination model*, able to capture system's behaviour with respect to its interactions with different components, is a complex process. This complexity arises from dealing with multiple activities and multiple participants which in turn are influenced by multiple constraints, such as exceptional situations, interrupts and failures. On the other hand, the need for methods and tools to identify, extract and record the coordination layer of running applications 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.

This paper is a step towards addressing such a problem. Its main contribution is a technique which adopts typical program analysis algorithms, namely slicing, to recover coordination information from legacy code. This is based on a notion of *coordination dependence graph*, abbreviate to CDG in the sequel, proposed here as a specialisation of standard program dependence graphs [3] used in classical program analysis. The discovery of coordination patterns in the source code of an application is achieved by a process of (sub-)graph identification in the corresponding CDG. The overall strategy is illustrated in Fig. 1.

The process starts by the extraction of a comprehensive dependence graph, denoted in the sequel by the acronym MSDG (after *Managed System Dependence Graph*), from source code. This is the fundamental structure underlying our approach, and extends, in several respects, previous work on such sort of program representations. This is briefly explained in section 2; a complete formalisation ap-

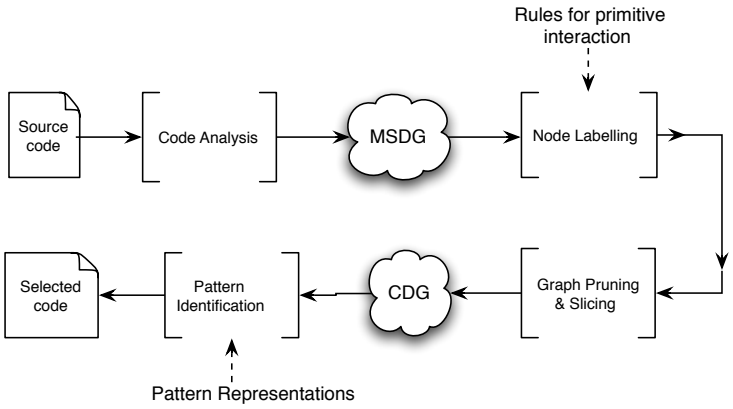


Fig. 1. The overall strategy

pears in [13]. The CDG mentioned above is, then, computed from this structure in a two stage process, presented in section 3. First nodes matching rules encoding the use of specific interaction or control primitives are suitably labelled. Then, by backwards slicing, the MSDG is pruned of all sub-graphs found irrelevant for the reconstruction of the program coordination layer. Note the first stage is parameteric in the set of rules and, therefore, in the type of interaction mechanisms used in the program under analysis. Once the CDG has been generated, the discovery of coordination patterns proceeds by the identification of which patterns in the graph encode them. Such patterns, which constitute another parameter in the method, and the associated discovery algorithm are discussed in section 4. From each coordination pattern discovered in the CDG, the corresponding chunk of source code is identified and returned.

The technique discussed in this paper is *generic* in the sense that it does not depend upon the programming language or platform in which the original system was developed. Actually, it can be implemented to target any language with basic communications and multi-threading capabilities. In section 5 the method is illustrated with a (toy) example in $C^\sharp$, in order to keep the presentation self-contained. However, a prototype tool, a preliminary version of which is available from the author's web-page, is being developed, as a 'proof-of-concept', which analyses *Common Intermediate Language* (CIL) source code, the language interpreted by the .Net Framework for which every Microsoft .Net compliant language compiles to.

2 The Managed System Dependence Graph

The fundamental information structure underlying the coordination discovery method proposed in this paper is a comprehensive dependence graph — the MSDG — recording the elementary entities and relationships that may be inferred from code by suitable program analysis techniques.

A MSDG is an extension of a *system dependence graph* to cope with object-oriented features, as considered in [6,7,16]. Our own contribution was the introduction of new representations for a number of program constructs not addressed before, namely, partial classes and methods, delegates, events and lambda expres-

sions. For a formal specification of a MSDG, as well as for a detailed description of the techniques used in its construction, the reader is referred to [13]. In this section, however, we provide a brief overview of the structure of a MSDG, as detailed as necessary to the presentation of the pattern discovery algorithm presented in section 4.

Before proceeding with the calculation of the MSDG, the program under analysis needs to be pre-processed first. This pre-processing phase, amounts to calculating the *used* and *defined* variables of a given statement as well as the control dependencies between statements. Used and defined variables of a statement can be easily calculated with suitable expression analysis. Control dependencies can also be trivially calculated for well structured languages, like the ones being addressed here, so we assume that such analysis are being performed in this stage. Further more, we also assume that all object reference aliases are being handled in this pre-processing phase. Although objected reference aliases resolution is not a trivial operation to perform, we rely on the several research works [14,15] addressing this issue, and assume that all object aliases have been properly resolved.

A MSDG is defined over three types of nodes representing program entities: *spatial nodes* (subdivided into classes Cls, interfaces Intf and name spaces Nsp), *method nodes* (carrying information on method's signature MSig, statements MSta and parameters MPar) and *structural* nodes which represent implicit control structures (for example, recursive references in a class or a fork of execution threads). Formally,

$$\begin{aligned} \mathsf{Node} &= \mathsf{SNode} + \mathsf{MNode} + \mathsf{TNode} \\ \mathsf{SNode} &= \mathsf{Cls} + \mathsf{Intf} + \mathsf{Nsp} \\ \mathsf{MNode} &= \mathsf{MSig} + \mathsf{MSta} + \mathsf{MPar} \\ \mathsf{TNode} &= \{\triangle, \triangledown, \circ\} \end{aligned}$$

where + denotes set disjoint union. Nodes of type SNode contain just an identifier for the associated program entity. Other nodes, however, exhibit further structure. For example, a MSta node includes the statement code (or a pointer to it) and a label to discriminate among the possible types of statements in a method, i.e.,

$$\begin{aligned} \mathsf{MSta} &= \mathsf{SType} \times \mathsf{SCode} \\ \mathsf{Stype} &= \{\mathsf{mcall}, \mathsf{cond}, \mathsf{wloop}, \mathsf{assgn}, \cdots\} \end{aligned}$$

where, for instance, mcall stands for any statement containing a call to a method and cond for a conditional expression. Similarly, a MSig node, which in the graph acts as the method entry point node, records information on both the method identifier and its signature, i.e., $\mathsf{MSig} = \mathsf{Id} \times \mathsf{Sig}$. Method parameters are handled through special nodes, of type MPar, representing input (respectively, output) actual and formal parameters in a method call or declaration. Formally,

$$\mathsf{MPar} = \mathsf{PaIn} + \mathsf{PaOut} + \mathsf{PfIn} + \mathsf{PfOut}$$

Finally, the structural nodes TNode were introduced to cope with concurrency (case of $\triangle$ and $\triangledown$) and to represent recursively defined classes (case of $\circ$). A brief

explanation is in order. A $\triangle$ node captures the effect of a spawning thread: it links an incoming control flow edge, from the vertex that fired the fork, and two outgoing edges, one for the new execution flow and another for the initial one. Dually, a thread join is represented by a $\triangledown$ node with two incoming edges and an outgoing one to the singular resumed thread. A $\circ$ node represents a recursively defined class, what seems a better alternative than expanding the object tree to a certain, but fix, depth, used, for example, in [7].

There are, of course, several types of program dependencies represented as edges in a MSDG. Formally, an edge is a tuple of type

$$\mathsf{Edge} \;=\; \mathsf{Node} \times \mathsf{DepType} \times (\mathsf{Inf} + \mathbf{1}) \times \mathsf{Node}$$

where DepType is the relationship type and the third component represents, optionally, additional information associated to it. Let us briefly review the main types of dependency relationships. Data dependencies, of type $\underline{\mathsf{dd}}$, connect statement nodes with common variables. Formally,

$$\langle v, \underline{\mathsf{dd}}, x, v' \rangle \in \mathsf{Edge} \Leftrightarrow \; \mathsf{definedIn}(x, v) \wedge \mathsf{usedIn}(x, v')$$

where x is a program variable and notation $\mathsf{definedIn}(x, v)$ (respectively, $\mathsf{usedIn}(x, v)$) stands for x is defined (respectively, used) in node v. Typical dependencies between statement nodes are of types control flow, $\underline{\mathsf{cf}}$, and control, $\underline{\mathsf{ct}}$, the latter connecting guarded statements (e.g. loops or conditionals) or method calls to their possible continuations and method signature nodes (which represent the entry-points on a method invocation) to each of the statement nodes within the method which is not under the control of another statement. Formally, these conditions add the following assertions to the invariant of type Edge [2] :

$$\begin{aligned}
&\langle v, \underline{\mathsf{ct}}, g, v' \rangle \in \mathsf{Edge} \Leftarrow \; v \in \{\mathsf{MSta}(t, -) |\; t \in \{\mathsf{mcall}, \mathsf{cond}, \mathsf{wloop}\}\} \wedge v' \in \mathsf{MSta} \\
&\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} \Leftarrow \; v \in \mathsf{MSig} \wedge v' \in \mathsf{MSta}
\end{aligned}$$

where g is either undefined or the result of the evaluation of the statement guard.

A method call, on the other hand, is represented by a $\underline{\mathsf{mc}}$ dependence from the calling statement wrt the method signature node. Formally,

$$\langle v, \underline{\mathsf{mc}}, vis, v' \rangle \in \mathsf{Edge} \Leftrightarrow \; v \in \mathsf{MSta} \wedge \mathsf{SType}\, v = \mathsf{mcall} \wedge v' \in \mathsf{MSig}$$

where vis stand for a visibility modifier in set $\{\mathsf{private}, \mathsf{public}, \mathsf{protected}, \mathsf{internal}\}$. Specific dependencies are also established between nodes representing formal and actual parameters. Moreover, all of the former are connected to the corresponding method signature node, whereas actual parameter nodes are connected to the method call node via control edges. Finally, any data dependence between formal parameters nodes is mirrored to the corresponding actual parameters. Summing

[2] All conditions constraining types Edge and Edge are formally recorded in two data type invariants associated to these types in the specification of the MSDG given in [13]; such invariants are only partially stated in this paper.

up, these adds the following assertions to the MSDG invariant:

$$\begin{aligned}
\langle v, \underline{\mathsf{pi}}, -, v' \rangle \in \mathsf{Edge} &\Leftrightarrow\ v \in \mathsf{PaIn} \wedge v' \in \mathsf{PfIn} \\
\langle v, \underline{\mathsf{po}}, -, v' \rangle \in \mathsf{Edge} &\Leftrightarrow\ v \in \mathsf{PaOut} \wedge v' \in \mathsf{PfOut} \\
\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} &\Leftarrow\ v \in \mathsf{MSig} \wedge v' \in (\mathsf{PaIn} \cup \mathsf{PaOut}) \\
\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} &\Leftarrow\ v \in \mathsf{MSta} \wedge \mathsf{SType}\, v = \mathsf{mcall} \wedge v' \in (\mathsf{PfIn} \cup \mathsf{PfOut}) \\
\langle v, \underline{\mathsf{dd}}, -, v' \rangle \in \mathsf{Edge} &\Leftarrow\ v \in \mathsf{PaIn} \wedge v' \in \mathsf{PaOut} \wedge \exists_{\langle u, \underline{\mathsf{dd}}, -, u' \rangle}\, .\, (u \in \mathsf{PfIn} \wedge u' \in \mathsf{PfOut})
\end{aligned}$$

Class inheritance and the fact that a class owns a particular method is recorded as follows

$$\begin{aligned}
\langle v, \underline{\mathsf{ci}}, -, v' \rangle \in \mathsf{Edge} &\Leftrightarrow\ v, v' \in \mathsf{Cls} \wedge v \neq v \\
\langle v, \underline{\mathsf{cl}}, vis, v' \rangle \in \mathsf{Edge} &\Leftrightarrow\ v \in \mathsf{Cls} \wedge v' \in \mathsf{MSig}
\end{aligned}$$

and, similarly, for interface and namespace nodes.

Other program entities and properties typically found in modern programming languages are also captured in a MSDG. They include, namely, *properties* (a special program construct in $C^\sharp$ and other .Net-based languages, intended to encapsulate access to class variables. But also *partial classes* and *partial methods*, the latter entailing the need for a $\underline{\mathsf{mc}}$ dependence edge between the declaration of the partial method and its implementation, as well as *delegates*, *events* and λ-*expressions*. A delegate is a sort of a function whose values are objects, thus possibly defining class member types. From the subscribed side, i.e., the class with the delegate definition that invoke the subscribed method, a method node is added to represent the delegate type, as well as parameter nodes for its arguments and results. Every call to the delegate inside the subscribed class is represented by a method call edge to the MSig node introduced by the delegate type. This acts like a proxy dispatching its calls to objects and methods which subscribed the delegate. In what concerns to graph representation, the difference between delegates and *events* is that the latter can be subscribed by more than one method, whilst delegate subscriptions override each other. Therefore, their representation in a MSDG is similar to that of delegates, but for the possibility of co-existence of more than one $\underline{\mathsf{mc}}$ edge between the subscribed and the actual method to be called in the subscriber. A similar approach is taken for the representation of λ-expressions, which in $C^\sharp$ are stateful and behave as anonymous methods (see [13] for further details).

3 The Coordination Dependency Graph

The second stage in the discovery process introduced in this paper is the construction of a CDG, which basically prunes the MSDG of all information not directly relevant for the reconstruction of the application coordination layer. This stage is guided by a specification of a set of rules specifying the interaction primitives used in the source code, which are actually the building blocks of any coordination

scheme. Such rules are specified as

$$\begin{aligned} \mathsf{CRule} &= \mathsf{RExp} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole}) \\ \mathsf{CType} &= \{\underline{\mathsf{webservice}}, \underline{\mathsf{rmi}}, \underline{\mathsf{remoting}}, \cdots\} \\ \mathsf{CDisc} &= \{\mathsf{sync}, \mathsf{async}\} \\ \mathsf{CRole} &= \{\mathsf{provider}, \mathsf{consumer}\} \end{aligned}$$

where RExp is a regular expressions, CType is the type of communication primitive types (extensible to other classes of communication primitives), CDisc is the calling mode (either synchronous or asynchronous) and, finally, CRole characterises the code fragment role wrt the direction of communication. In the $C^\sharp$, for example, the identification of invocations to web services can be captured by the following rule, which identifies the primitive synchronous framework method `SoapHttpClientProtocol.Invoke` typically used to invoke a web service:

```
R = ("SoapHttpClientProtocol.Invoke(*);",
     (webservice, sync, consumer))
```

Given a set of rules, the CDG calculation, starts by testing all the MSDG vertices against the regular expressions in the rules. If a node of type MSta or MSig matches one of such regular expressions, it becomes labelled with the information in the rule's second component. The types of the resulting nodes are, therefore,

$$\begin{aligned} \mathsf{CMSta} &= \mathsf{MSta} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole}) \\ \mathsf{CMSig} &= \mathsf{MSig} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole}) \end{aligned}$$

Note that, because of this labelling process, the type of a CDG node is

$$\mathsf{CNode} = \mathsf{Node} + \mathsf{CMSta} + \mathsf{CMSig}$$

On completion of this labelling stage, the method proceeds by abstracting away the parts of the graph which do not take part in the coordination layer. This is a major abstraction process accomplished by removing all non-labelled nodes, but for the ones verifying the following conditions:

(i) method call nodes (i.e., nodes v such that $v \in \mathsf{MSta}$ with $\mathsf{SType}\, v = \mathsf{mcall}$) for which there is a control flow path (i.e., a chain of $\underline{\mathsf{cf}}$ dependence edges) to a labelled node.

(ii) vertices in the union of the backward slice of the program with respect to each one of the labelled nodes.

Note that the first condition ensures that the relevant procedure call nesting structure is kept. This information will be useful to nest, in a similar way, the generated code on completion of the discovery process. The second condition keeps all the statements in the program that may potentially affect a previously labelled node. This includes, namely, MSta nodes whose statements contain predicates (e,g., loops or conditionals) which may affect the parameters for execution of the communication primitives and, therefore, play a role in the coordination layer.

This stage requires a slicing procedure over the MSDG, for which we adopt a backward slicing algorithm similar to the one presented in [4]. It consists of two phases. The first phase marks the visited nodes by traversing the MSDG backwards, starting on the node matching the slicing criterion, and following ct, mc, pi, and dd labelled edges. The second phase consists of traversing the whole graph backwards, starting on every node marked on phase 1 and following ct, po, and dd labelled edges. By the end of phase 2, the program represented by the set of all marked nodes constitutes the slice with respect to the initial slicing criterion.

Except for cf labelled edges, every other edge from the original MSDG with a removed node as source or target, is also removed from the final graph. The same is done for any cf labelled edge containing a pruned node as a source or a sink. On the other hand, new ct edges are introduced to represent what were chains of such dependencies in the original MSDG, i.e. before the removal operation. This ensures that future traversals of this graph are performed with the correct control order of statements.

4 Coordination Patterns Discovery

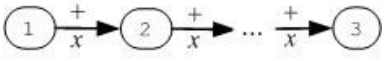
4.1 Describing Coordination Patterns

In contrast with the MSDG, which is usually a large and complex structure, the CDG extracted from a typical system is much smaller, since all code alien to the coordination layer has already been removed. Nevertheless, the correct identification of the structure of such a layer is, usually, far from trivial. In our approach this process is driven by a series of predefined coordination patterns encoded as subgraphs instances of which are to be discovered over the CDG. Formally, coordination patterns are defined as pairs formed by a *matching condition* (of type PCondition) and a graph over nodes of type NodeId as follows

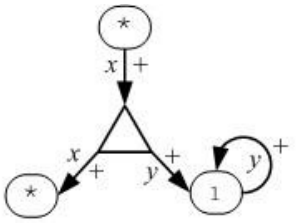
$$\begin{aligned}
\mathsf{Pattern} &= \mathsf{PCondition} \times (\mathsf{NodeId} \times \mathsf{ThreadId} \times \mathsf{NodeId} \times \mathsf{PathPattern})^* \\
\mathsf{PCondition} &= \mathsf{NodeId} \rightharpoonup \mathbf{2}^{\mathsf{GNode}} \\
\mathsf{NodeId} &= \mathbb{N} \cup \{\triangle, \triangledown\} \\
\mathsf{PathPattern} &= \mathbb{N} \cup \{*\}
\end{aligned}$$

A matching condition is a mapping (i.e., a partial function) which associates to each pattern node (of type NodeId) a predicate over CDG nodes (of type GNode). In practice, a common definition such a predicate resorts to a regular expression intended to be tested for matching with the program information collected on CDG nodes. Symbol $*$ is used to abbreviate the everywhere true predicate. Examples of pattern conditions are shown later in this section.

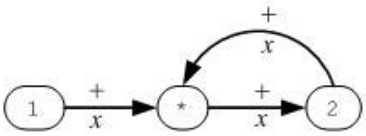
The second component of a pattern is a sequence of edges labelled by a thread identifier (ThreadId), which is used to specify the intervening threads in a pattern, and a qualifier (of type *PathPattern*) which specifies the number of edges in the CDG that may mediate between the node matching the source and the target node in the pattern. In particular, symbol + is used to stand for one or more edges. Note this qualifier is always greater than 0. We also assume that all nodes in the



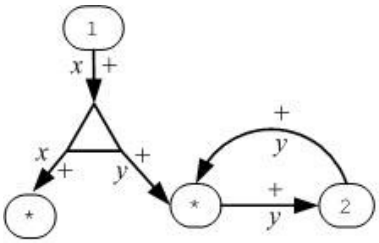
(a) Synchronous Sequential Pattern



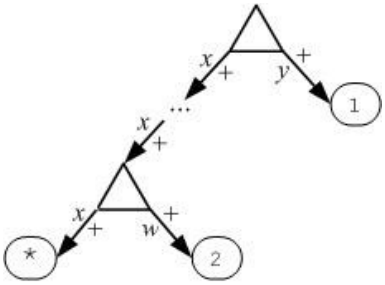
(b) Cyclic Query Pattern



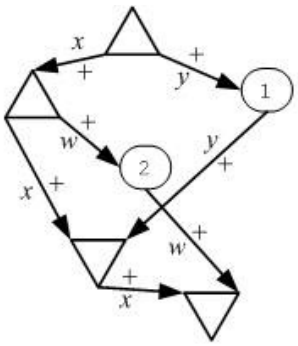
(c) Asynchronous Query Pattern



(d) Asynchronous Query Pattern with Client Multithreading



(e) Asynchronous Sequential Query Pattern



(f) Joined Asynchronous Sequential Pattern

Fig. 2. *CDGPL Patterns*

sequence of edges of a pattern which do not belong to the domain of the respective condition, are implicitly labelled by the everywhere true predicate.

Based on the data specifications above, we have defined a small language to express coordination patterns. Such notation, referred to as the *Coordination Dependence Graph Pattern Language* (CDGPL) was specifically designed to describe CDG graph patterns and to facilitate the automatic discovery of such patterns — see [13] for a complete specification. The discovery process, in particular, is guided by what we call a *search pattern*, i.e. an expression defined simply as a pattern (of type Pattern) or either as a conjunctive (&&) or disjunctive (‖) aggregation of patterns.

For illustration purposes, however, we resort in this paper to a graphical notation to present a number of most typically found coordination patterns, depicted in Fig. 2. In each pattern, notation vc_x denotes the node condition for node x.

4.1.1 Synchronous Sequential Pattern

This is one of the most simple patterns in which external services in a sequence are invoked one after the other. This simple, yet often used, pattern is usually employed when there are dependencies between a number of service calls, i.e., when a service call depends on the response received from a previous one.

In our notation this pattern is specified as in Fig. 2(a), where each node corresponds to a service call of the series of services to be invoked in sequence. If the original source code implements coordination through access to web-services, the condition for each these vertices can be defined by the following predicate template:

$$pc(x) = \; x == (\mathsf{MSta}(t, s), cp, cm, cd) \Rightarrow \\ match(s, \text{“}\texttt{ServiceCall(*)}\text{”}) \wedge cp == \underline{\mathsf{webservice}} \wedge cm == \mathsf{sync} \wedge cd == \mathsf{consumer}$$

where “`ServiceCall`” is to be replaced by the name of the invoked web service method.

4.1.2 Cyclic Query Pattern

This pattern is characterized by a point in which a new thread is spawned becoming responsible for a on-going invocation of an external service. It is often used by systems that have to monitor the state of some foreign resource or that must be constantly updating an internal resource which depends upon an external service.

In practice, the pattern appears in several variations. For instance, it may include a time delay between each cyclic service call or use different strategies to implement the service invocation cycle, e.g. resorting to be recursive function definition or iterative control statements. The pattern presented in Fig. 2(b) captures its most generic version. It basically states that a new thread y must be spawned and that under the execution of these new thread a service must be called repeatedly. Again, vertex 1 must be instantiated with a predicate, similar to the previous one, limiting the service being called.

4.1.3 Asynchronous Query Pattern

. The Asynchronous Query Pattern is usually employed if there is a need to invoke time consuming services, and calling threads can not suspend until a response is returned. To overcome this situation, the server component provides two methods, one for the request of an operation on the server and another for the querying of an answer (if available) from the previously posted request. Both this server methods return very quickly, since they are not involved in the execution of any complex operation but rather in the control of the execution of complex operations and results retrieval. From the client side this pattern is specified by the definition in Fig. 2(c), encoding the invocation of a service to request the execution of some operation execution (node 1) and a cyclic invocation of another service (node 2) to retrieve the result. Once more, in practice, both vertices 1 and 2 may be further characterized by predicates that clearly identify the operation request and result request services.

4.1.4 Asynchronous Query Pattern (with client multithreading)

This often used pattern is actually a variation of the previous one, where the client orders the execution of an operation in one thread and then launches a second thread to retrieve the result. Note that this pattern, presented in Fig. 2(d) is also quite similar to the cyclic pattern, but for an extra node, marked with $*$ to represent the program statement that controls the need to perform more invocations to retrieve the result for an operation.

4.1.5 Asynchronous Sequential Pattern

This is similar to the *Asynchronous Sequential Pattern* except that it invokes each service in a new thread specifically created for the effect. This pattern is often used when a system has to invoke a series of services, the order of invocation as well as the responses returned are irrelevant. Note that, under this premises, this pattern is substantially faster than the *Asynchronous Sequential Pattern* in the invocation of the series of services. This pattern is specified in Fig. 2(e) where each of the service calling nodes (1 and 2) are invoked in different threads (y and w respectively).

4.1.6 Joined Asynchronous Sequential Pattern

This is similar to the previous pattern in the sense that, in both of them, services are invoked asynchronously. The difference is that in this pattern one is interested in controlling the point where each of the called services have finished execution and, possibly, returned a value. The specification of this pattern is presented in Fig. 2(f) where each thread that was spawned to invoke a service, joins later in a point where the execution may proceed with the guarantee that all service calls have finished executing.

4.2 The discovery algorithm

The algorithm presented in this section retrieves every sub-graph of a CDG that conforms to a given *graph pattern*. The notation used in self-explanatory. However, let us point out the use of dot . as a field selector in a record as well as the adoption of the Haskell syntax for lists (including functional *map* and operators : for appending and ++ for concatenation). An assignment is denoted by the $\leftarrow$ operator; note that it can be prefixed by an expression declaring the type of the variable being bound.

The algorithm resorts to the data types in Fig. 3, also expressed in the Haskell syntax for data type declarations. Note, in particular, how both the CDG and the graph representing the pattern to be discovered are made available to the algorithm through embedding in *Graph* and *GraphPattern*: in both cases a node is selected as 'root', i.e. as a starting point for searching.

The overall strategy used by the discovery algorithms 1 and 2 consists of traversing the graph pattern and incrementally constructing a list of candidate graphs with nodes of type *Attribution*. This type is used by the algorithm because it maintains a mapping between the graph pattern nodes and CDG nodes. If a pattern is found, during the traversal of the graph pattern, for which a candidate graph cannot be extend to conform with, then the graph in question is removed from the candidate graphs list. On the other hand, if the candidate graph can be extend with one of the

$$
\begin{array}{lcrcl}
Graph & = & G\ \{\ root & :: & GNode, \quad G :: CDG\ \} \\
GraphPattern & = & GP\ \{\ root & :: & NodeId, \quad G :: VertexPattern\ \} \\
VertexPattern & = & VP\ \{\ id & :: & Int, \\
 & & cdts & :: & [GNode] \\
 & & visited & :: & \mathbb{B}\ \} \\
Attribution & = & AT\ \{\ vp & :: & VertexPattern, \\
 & & v & :: & GNode\ \} \\
Extension & = & E\ \{\ g & :: & Graph, \\
 & & att & :: & [Attribution]\ \}
\end{array}
$$

Fig. 3. Data types for the Graph Pattern Discovery algorithm

several CDG candidate nodes, it originates a series of new candidate graphs (one for each CDG candidate node) and the original (incomplete) candidate is removed from the candidate list.

Most auxiliary functions used in the algorithm are self-explained by its identifier names, with the possible exception of function GetSuccCombinations which calculates a list of lists of *Attributions*, i.e., a list for each possible set of possible attributions for a given node pattern. By using the graph pattern discovery algorithm we are now able to identify coordination patterns in legacy code. Moreover, if each pattern is associated to a pattern 'implementation' in one of the several coordination languages available in the literature, one will be able to reconstruct a specification of the system whose code has been analysed.

5 Example

As an example of the presented coordination pattern discovery method, consider the $C^\sharp$ code fragment in appendix A. The class `WeatherServer` used in the code is the web-service proxy class, automatically generated by the tool *Microsoft.VSDesigner*.

The example code is intended to be run on a client that calls a server to predict the weather for the next couple of days based on the current weather conditions. Because weather prediction is a complex and time consuming task it is unfeasible for the client execution thread to be hold until a response from the server is return. Thus, the client submits the prediction operation to the server, the server returns immediately yielding an operation identifier for the client request and then the responsibility to request for an answer is passed to the client which has to perform multiple queries to the server until a weather prediction answer is returned. Once the client receives the answer with the prediction from the server it inspects the result and if it seems wrong (method `CheckPrediction`) it submits a new request to the server in order to reevaluate the prediction.

By the description of the client behaviour, it becomes more or less clear that this client probably implements one or more instances of the previously presented *Asynchronous Query Pattern*. More difficult, is to identify exactly which statements in the code are responsible for the implementation of the pattern. Note that in real world systems, this difficulty is even greater since the code would most certainly be 'intermediated' by other statements (with completely different purposes other that coordinating other components, like updating the user interface, or freeing resources) as well as spread among different parts of the system, thus making it quite difficult to identify any coordination pattern.

Algorithm 1 Pattern Discovery - Part I

```
1:  function DiscoverPattern(Graph cdg, GraphPattern cdgp)
2:      cdgp ← FillCandidateVertices(cdg, cdgp)
3:      cdgp ← FillCandidateEdges(cdg, cdgp)
4:      Graph bg ← emptyGraph()
5:      [Extension] gel ← [(bg, map (λx → (cdgp.root, x)) cdgp.root.cdts]
6:      repeat
7:          𝔹 b ← False
8:          for all Extension ge in gel do
9:              for all Attribution datt in ge.att do
10:                 datt.vp.visited ← True
11:                 c1 ← HasSuccessors(cdgp, datt.v)
12:                 c2 ← ! HasSuccessors(ge.g, datt.vp)
13:                 if c1 ∧ c2 then
14:                     [Extension] dgel ← ExtendBaseGraph(ge.g, datt)
15:                     [Extension] r ← ge : r
16:                     [Extension] a ← dgel : a
17:                     b ← b ∨ Length(dgel) > 0
18:                 end if
19:             end for
20:         end for
21:         gel ← Remove(gel, r)                    ▷ Remove all r elements from gel
22:         gel ← gel ++ r                          ▷ Add all a elements to gel
23:         r ← []
24:         a ← []
25:         nv ← NotVisited(cdgp)                   ▷ Get first not visited Vertex Pattern
26:         if b ∧ nv ≠ null then
27:             b ← True
28:             vpa ← map (λx → (nv, x)) nv.cdts
29:             map (λx → (x.g, vpa)) gel
30:         end if
31:     until b == True
32:     return gel
33: end function
```

Due to space limitations we omit some code details, which are clearly identified by underlined comments. Two of this omissions are concerned with the construction of the parameters being passed to the server operations (lines 15 and 31), which amount to the gathering of the current weather conditions. The second omission (line 52) regards the code to setup the web service proxy class, which contains the code to control all the Simple Object Access Protocol (SOAP) communications as well as all object marshalling operations.

The process of discovering instances of the *Asynchronous Query Pattern* starts by the construction of the MSDG for the code under analysis. Fig. B.1, in appendix B, presents the MSDG for the example code. To maintain the readability of the graph, one has opted to include only control, method call, control flow, formal-in and out dependencies.

Algorithm 2 Pattern Discovery - Part II

```
34: function ExtendBaseGraph(Graph bg, Attribution att)
35:     tcs ← GetSuccCombinations(cdgp, vp)
36:     for all tc in tcs do
37:         ng ← bg
38:         gel ← (ge, [])
39:         for all cv in tc do
40:             if b ∧ nv ≠ null then
41:                 AddEdge(ng, att, cv)
42:                 ge.DiscoveredAttributions.Add(cv)
43:             else
44:                 gel.Remove(ge)
45:                 break
46:             end if
47:         end for
48:     end for
49:     return gel
50: end function
```

The following phase consist in the calculation of the CDG, based on the constructed MSDG. In this example one is interested in identifying synchronous calls to web services. Such identification can be performed using the rule (“`SoapHttpClientProtocol.Invoke(*);`”, $(WebService, Sync, Consumer)$), which identifies web-services calls made by the *Microsoft.VSDesigner* tool.

The process of calculating the CDG, as explained in section 3, leads to the elimination of the code lines 3, 7, 8, 9, 10, 14, 24, 26, 30, 39 and 42 or in graphical terms to the dashed vertices in Fig. B.1. Note that the removed statements are exactly the ones not directly involved in the invocation of web-services, which in this small and highly coordination devoted code corresponds almost entirely to IO statements. Nevertheless in a real world system, containing logic to control many other aspects besides coordination of foreign resources, the percentage of program statements being sliced, with respect to the entire system, would certainly be much higher.

The following phase is to define in CDGPL an expression that characterises the coordination pattern one is looking for. For this example, one will use the *Asynchronous Query Pattern* CDGPL definition presented in section 4.1, with the following *pc* pattern condition. This pattern condition makes use of a regular expression matching function named *match*.

$$pc(1) = \lambda(\mathsf{MSta}(t,s), cp, cm, cd) \rightarrow (match(s, \text{“}GetForecast(*)\text{”}) \vee match(s, \text{“}ConfirmForecast(*)\text{”})) \wedge cp == \underline{\text{webservice}} \wedge cm == \text{sync} \wedge cd == \text{consumer}$$

$$pc(2) = \lambda(\mathsf{MSta}(t,s), cp, cm, cd) \rightarrow match(s, \text{“}GetOperationResult(*)\text{”}) \wedge cp == \underline{\text{webservice}} \wedge cm == \text{sync} \wedge cd == \text{consumer}$$

Using the graph pattern discovery algorithm presented in section 4.2 one clearly identifies two instances of the *Asynchronous Query Pattern* used in the code and highlighted by the two mappings f_1, f_2 between the vertex pattern identifiers and the example code line statements. In order to facilitate the identification of the statements involved in the coordination pattern instances, the example code in

appendix A highlights these statement in red and italic font.

$$f_1(1) = 16 \qquad f_2(1) = 32$$
$$f_1(2) = 23 \qquad f_2(2) = 38$$
$$f_1(3) = 25 \qquad f_2(3) = 40$$

Note that, although being discovered as instances of the same coordination pattern, the implementations are quite different from each other. In the first case the querying for an answer is performed by a `while` cycle, whereas in the second case this is performed by a recursive call to method `GetForecastConfirmationResult`.

6 Conclusions and Future Work

This paper introduced a method that combines a number of program analysis techniques (namely, *dependence graphs*, *program slicing*, and *graph pattern analysis*) to extract coordination logic from legacy systems source code. The process is driven by a series of pre-defined coordination patterns and captured by a special purpose graph structure from which coordination specifications can be generated in a number of different formalisms. The use of dependence graphs to represent different sorts of program entities and the ways they depend on each other has already a long history in the program analysis community — see, e.g. [10] for an early reference. Our contribution has been to extend previous work (namely [5,8]) to collect all the information that may be necessary to extract the (often deeply hidden) coordination layer of an application. Note that most of the work and tools developed for reverse engineering have limited scope, typically intended to obtain module, class diagrams and method call dependencies from legacy code.

One of the most relevant parts if this approach is its parametrisation by rules identifying the communication primitives one is interested in, thus making it adaptable to diverse kinds of coordination analysis and programming frameworks. Given the language heterogeneity that pervades most software, such a "language agnosticism" of the technique stands as another very important feature.

A prototype tool [3] to implement the techniques presented here is currently under development. The tool, named CoordInspector, targets the *Common Intermediate Language* (CIL), thus making it able to cope with any Microsoft .Net Language. Although targeting a completely different language, the development o CoordInspector shares a number of intuitions discussed in [12,11,9].

Although the most direct application of this approach and tool is to assists on the coordination analysis of legacy systems, they can also be used to assess the correctness of systems implementations with respect to its design specifications or even with respect to the growing software quality regulations. Even more, with the provision of rules for COM or RMI communication discovery, it can be used to assist the conversion of distributed object systems towards web-service oriented systems (or vice versa).

An interesting topic for future work is the classification of coordination patterns,

[3] A preliminary version of which is available at the authors web page.

as in [1], in terms of their graph pattern representation expressed in CDGPL. This information would allow the creation of a coordination patterns repository which could be used not only for reverse, but also for forward, systems engineering.

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] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[4] 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.

[5] J. Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, 2003.

[6] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[7] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 358, Washington, DC, USA, 1998. IEEE Computer Society.

[8] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 180–190, New York, NY, USA, 2000. ACM.

[9] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, 2006.

[10] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering posium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[11] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5):27, 2007.

[12] V. P. Ranganath and J. Hatcliff. Slicing concurrent java programs using indus and kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9(5):489–504, 2007.

[13] N. F. Rodrigues. *Generic software slicing applied to architectural analysis of legacy systems*. PhD thesis, Dep. Informática, Universidade do Minho, 2008. (forthcoming PhD thesis).

[14] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *International Conference on Software Engineering*, pages 433–443, 1997.

[15] J. Woo, J.-L. Gaudiot, and A. L. Wendelborn. Alias analysis in java with reference-set representation for high-performance computing. *Int. J. Parallel Program.*, 32(1):39–76, 2004.

[16] J. Zhao. Applying program dependence analysis to java software. In *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, December 1998.

A Example Code

```
1  class Example {
2     private void GetWeatherForecast() {
3        Console.WriteLine("Calculating forecast.");
4        WeatherServer cs = new WeatherServer();
5        int taskId = RequestServerTask(cs);
6        Result res = GetTaskResult(cs, taskId);
7        if(res != null)
8           Console.WriteLine("Forecast: " + res.ToString());
```

```
9          else
10            Console.WriteLine("Operation failed");
11      }
12
13     private int RequestServerTask(WeatherServer cs) {
14         Console.WriteLine("Requesting forecast.");
15         Operation op = ...current weather conditions gathering code...
16         int operationId = cs.GetForecast(op);
17         return operationId;
18     }
19
20      private Result GetTaskResult(WeatherServer cs, int opId) {
21         Result res = null;
22         int i = 0;
23         while(res == null && i++ < 10) {
24            Console.WriteLine("Querying server for forecast.");
25            res = cs.GetOperationResult(opId);
26            Thread.Sleep(1000);
27         }
28         // Check if the result still needs further calculation
29         if(!CheckPrediction(res)) {
30              Console.WriteLine("Querying server to confirm forecast.");
31              Operation op2 = ...confirm forecast parameter construction...
32              int op2Id = cs.ConfirmForecast(op2);
33              res = GetForecastConfirmationResult(cs, op2Id);
34         }
35         return res;
36      }
37
38      private Result GetForecastConfirmationResult(WeatherServer cs, int opId) {
39         Console.WriteLine("Querying server for simplification result.");
40         Result res = cs.GetOperationResult(opId);
41         if(res == null) {
42            Thread.Sleep(2000);
43            return GetForecastConfirmationResult(cs, opId);
44         } else {
45            return res;
46         }
47      }
48  }
49
50  class WeatherServer : System.Web.Services.Protocols.SoapHttpClientProtocol {
51
52      ...proxy class setup code...
53
54      public int GetForecast(Operation op) {
55         object[] results =
56            this.Invoke("PerformComplexOperation",
57                        new object[] { op });
58               return ((int)(results[0]));
59      }
60
61      public int ConfirmForecast(Operation op) {
62         object[] results =
63            this.Invoke("ConfirmForecast",
64                        new object[] { op });
65               return ((int)(results[0]));
66      }
67
68      public Result GetOperationResult(int opId) {
69         object[] results =
70            this.Invoke("GetOperationResult",
71                        new object[] { opId });
72               return ((Result)(results[0]));
73      }
74  }
```

B Example Code MSDG

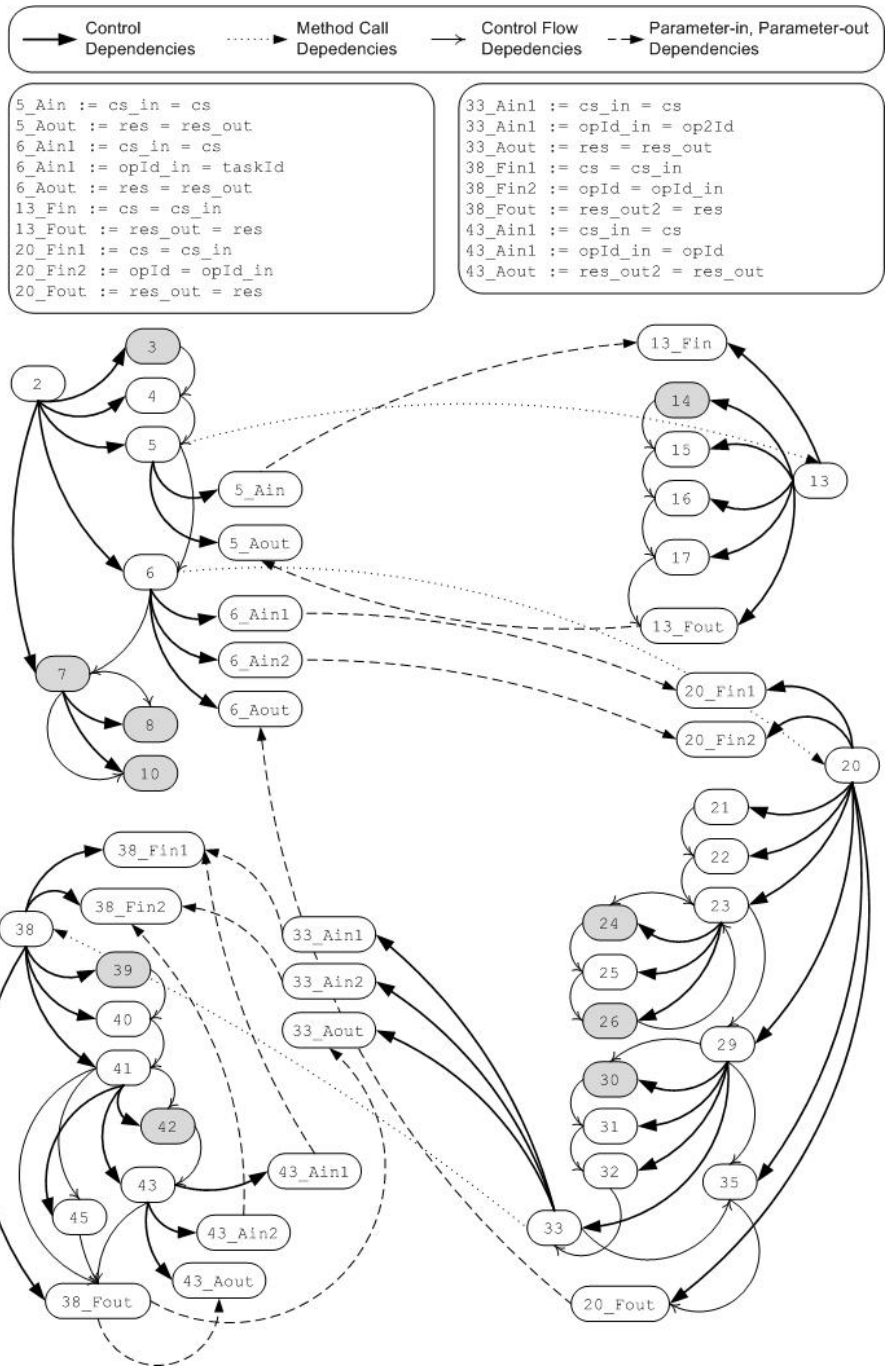


Fig. B.1. Example code MSDG