Slicing for Architectural Analysis

Nuno F. Rodrigues, Luis S. Barbosa

DI-CCTC, Universidade do Minho
Braga, Portugal
{nfr,lsb}@di.uminho.pt

Abstract

Current software development often relies on non trivial coordination logic for combining autonomous services, eventually running on different platforms. As a rule, however, such a coordination layer is strongly weaved within the application at source code level. Therefore, its precise identification becomes a major methodological (and technical) problem and a challenge to any program understanding or refactoring process.

The approach introduced in this paper resorts to slicing techniques to extract coordination data from source code. Such data is captured in a specific dependency graph structure from which a coordination model can be recovered either in the form of an Orc specification or as a collection of code fragments corresponding to the identification of typical coordination patterns in the system. Tool support is also discussed.

Key words: Program analysis, architectural recovery, coordination.

1. Introduction

1.1. Context and motivation

By the end of the last century *program understanding* and *reverse engineering* had emerged as key concerns in software engineering, attracting an ever-increasing attention both in industry and academia. Actually, the increasing relevance and exponential growth of software systems, both in size and quantity, lead to an equally growing amount of legacy code that has to be maintained, improved, replaced, adapted and assessed for quality regularly.

The high dependence of modern societies on such legacy systems and the incredibly fast rate of evolution which characterises software industry, make companies and managers willing to spend resources to increase confidence

on — *i.e.*, the level of understanding of — their running code. Actually, the technological and economical relevance of *legacy* software, as well as the complexity of its re-engineering and the (often exponential) costs involved, justify the increased attention the problem has been receiving recently.

The approach introduced in this paper is part of this research effort on techniques to extract, from source code, specific knowledge, to be suitably represented and visualised, and to provide a basis for systems analysis and reconstruction. More specifically,

- On the technical side, it targets program understanding at a *macro*, architectural level. I.e., the identification and analysis of the underlying *coordination model*, as an abstraction of the behavioural interplay between the various services, components, and the (more or less explicit) independent *loci* of computation from which a system is composed of.

- On the methodological side, it resorts to *slicing* — a decomposition technique to extract from a program, information relevant to a given computation, originally proposed by M. Weiser, 30 years ago [32].

Several approaches have been proposed for reverse architectural analysis. For example, in the context of model-driven engineering [29], generators for UML diagrams became rather popular. A *Class Diagram* generator, for instance, extracts classes from object oriented source code, whereas a *Module Diagram* generator builds box-line diagrams from system's modules, packages or namespaces. On the other hand, *Uses Diagram* generators reconstruct the import dependencies of the system, and *Call Diagram* generators expose the direct calls between system parts. All of them are relevant at the architectural level, understood, according to norm ANSI/IEEE Std 1471-2000, as *the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*

However, none of these techniques/tools makes 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 are called during the execution of a particular procedure. However, no answers are provided 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 come to a deadlock situation, and what is the sequence of actions leading to it?

This sort of questions belongs to what can be called the *coordination* layer, which captures system's behaviour with respect to its network of interactions. The qualifier is borrowed from research on *coordination* models and languages [17, 4], which emerged a decade ago to exploit the full potential of parallelism, concurrency and cooperation of heterogeneous, loosely-coupled components. At present, 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.

We claim that, if coordination policies can be extracted from 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 (as implemented) with respect to its expected interaction patterns and constraints. The approach proposed here, as detailed in the sequel, is a step in that direction.

1.2. Overview of contribution

The paper's main contribution is a slicing-based technique to recover coordination information from legacy code. Actually, what can be achieved by slicing, *i.e.* the isolation of a particular sub-computation of interest inside an entire program, goes far beyond its initial purpose of error detection. Program slicing techniques became relevant to a large number of areas, such as, reverse engineering [7, 30], program understanding [10, 12], debugging [2], software integration [5, 15], software maintenance [6, 8], testing and test planning [13], among others.

The approach introduced in this paper is based on first building an extended system dependence graph, to provide a structural and easy-to-manipulate representation of program data, and then resorting to slicing techniques over such a graph to extract the relevant coordination policies.

Two alternatives are considered to address the problem of discovering and extracting coordination data from code, once a specific dependence graph structure — to be referred to as the *coordination dependence graph*, CDG, in the sequel — has been built.

- The first one, proceeds by systematically translating the data recorded in the CDG into a specific software orchestration language. The outcome is, therefore, a high-level specification of the recovered coordination policies. We resort to Orc, a recent general purpose orchestration language proposed by J. Misra and W. Cook [23] for this task. Orc scripts can be animated to simulate such specifications and study alternative coordination policies.

- An alternative approach inspects the entire CDG for the identification of graph patterns which are known to encode particular coordination schemes. For each instance of one of these patterns, discovered in the graph, the corresponding fragment of the source code is identified and returned.

The second alternative scales better to systems with complex coordination policies, where a set of coordination patterns identified in (typically huge) CDGs, provides more insight than a long, flat Orc specification.

The construction of the CDG, proposed here as a specialisation of standard program dependence graphs [11] used in classical program analysis, is fundamental to both approaches. In the first one, Orc specifications are directly generated from it. In the second, the discovery of coordination patterns in the source code 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 appears in [27]. 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. Such rules are the first parameter of the method.Then, by backwards 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 can be "translated" into a Orc specification, as explained in section 4. Alternatively, the discovery of coordination patterns proceeds by the identification of corresponding graph patterns. Such patterns, which constitute a second parameter in the

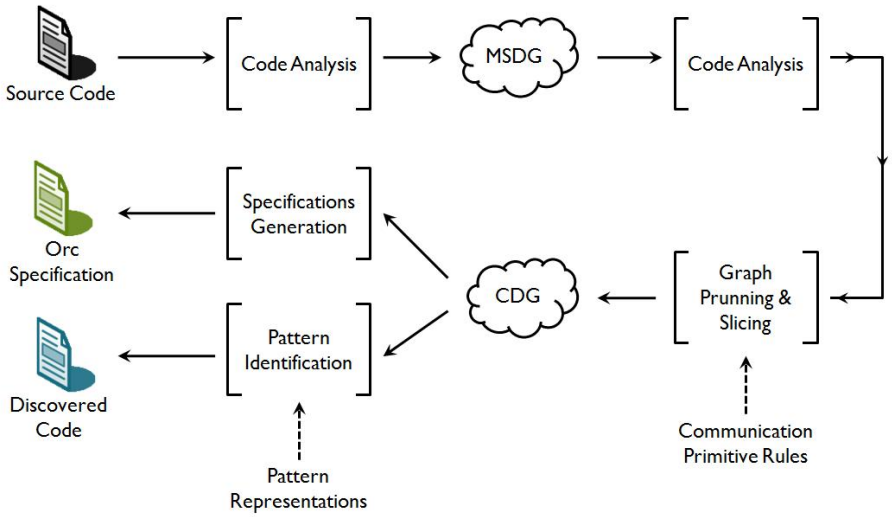


Figure 1: The overall strategy

method, and the associated discovery algorithm are discussed in section 5. From each coordination pattern discovered in the CDG, the corresponding chunk of source code is identified and returned.

Both approaches are *generic* in the sense that they do not depend upon the programming language or platform in which the original system was developed. Moreover, the construction of the CDG is parametric on a set of rules which captures the sort of interaction mechanisms used in the program under analysis.

CoordInspector, a support tool developed as a "proof-of-concept" for the proposed method, is described in section 6. Able to analyse *Common Intermediate Language* (CIL) source code, the intermediate language for the .Net Framework to which every Microsoft .Net compliant language compiles, CoordInspector supports systems developed in more than 40 different programming languages and combinations thereof. In section 7 the method is illustrated with a (toy) example in $C^{\natural}$.

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 classical *system dependence graph* to cope with object-oriented features, as considered in [20, 21, 34]. 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 expressions. 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 [27]. In this section, however, we provide a brief overview of the structure of a MSDG, as detailed as necessary for the presentation of the pattern discovery algorithm in section 5.

To prepare the grounds for computing a MSDG, the system under analysis needs to be pre-processed, calculating the *used* and *defined* variables in each 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, so we assume that such analysis is performed in this stage. Furthermore, we also assume that all object reference aliases are handled in this pre-processing phase. Although the resolution of objected reference aliases is not trivial , we rely on the several research works [31, 33] 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 the 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,

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

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.,

$$\mathsf{MSta} = \mathsf{SType} \times \mathsf{SCode} \qquad \mathsf{SType} = \{\mathsf{mcall}, \mathsf{cond}, \mathsf{wloop}, \mathsf{assgn}, \cdots\}$$

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 (respectively, formal) parameters in a method call or declaration. Formally,

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

Finally, 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 fires a 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 node labelled with $\circ$ represents a recursively defined class. The introduction of such a node is required as our representation of a class is done by unfolding all of its class variables. Since such variables may introduce direct and indirect recursion, a way is needed to stop unfolding their definitions. This mechanism provides an alternative to expanding the object tree to a certain, but fix, depth, used, for example, in [21].

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 $\mathsf{DepType}$ is the relationship type and the third component represents, optionally, additional information associated with. Let us briefly review the main types of dependency relationships. Notation $\mathbf{1}$ stands for the (isomorphism class of) the singleton set whose (unique) value is, by convention in this paper, denoted by $-$. 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 for method invocation) to each of the statement nodes, within the method, that are not under the control of another statement. Formally, these conditions add the following assertions to the invariant of type Edge[1]:

$$\langle v, \underline{\mathsf{ct}}, g, v' \rangle \in \mathsf{Edge} \Leftarrow\ v \in \{\mathsf{MSta}(t, -) \mid t \in \{\mathsf{mcall}, \mathsf{cond}, \mathsf{wloop}\}\} \wedge v' \in \mathsf{MSta}$$
$$\langle v, \underline{\mathsf{ct}}, -, v' \rangle \in \mathsf{Edge} \Leftarrow\ v \in \mathsf{MSig} \wedge v' \in \mathsf{MSta}$$

where g is either undefined or the result of the evaluation of the statement guard. Note that control flow edges (cf) correspond to the most basic type of dependencies identified in most graph-based program analysis. Actually, they interfere in the computation of almost all other dependency edges. This justifies their inclusion in the formalisation of the graph structure, even if no explicit use of them is made in the specific coordination analysis discussed in this paper.

A method call, on the other hand, is represented by a 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 at the level of the corresponding actual parameters. Summing up, these adds the following assertions to the MSDG invariant:

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

[1]All conditions constraining types Node and Edge are formally recorded in two data type invariants associated to these types in the specification of the MSDG given in [27]; such invariants are only partially stated in this paper.

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

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

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, for example, *properties* (a special program construct in some .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 invokes the subscribed method, a method node is added to represent the delegate type, as well as the necessary 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 subscribe the delegate. 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 and anonymous methods (see [27] for further details).

3. The coordination dependency graph

The second stage in the discovery process is the construction of a CDG, by pruning 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 values of the following type:

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

where RExp is a regular expression, CType is the type of communication primitive involved, CDisc represents the calling mode (either synchronous or asynchronous) and, finally, CRole characterises the code fragment role wrt the communication flow. The role of the RExp component is to direct the search of specific communication primitives inside program statements, which constitute the atoms in the regular expression. Which communication primitives to look for depends, obviously, on the code under analysis: both the method and the tool described in section 6 are parametric on such primitives, the 'building blocks' of interaction. As an example consider the following rule intended to identify, in $C\sharp$ code, a call to a web service:

$$R \;=\; (rex, (\underline{\mathsf{webservice}}, \underline{\mathsf{sync}}, \underline{\mathsf{consumer}}))$$
$$rex \;=\; \text{“}System.Web.Services.Protocols.SoapHttpClientProtocol.Invoke\backslash(\backslash w\backslash);\text{”}$$

Given a set of rules, CDG construction 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 the regular expressions, it becomes labelled with the information in the rule's second component. The types of the resulting nodes are, therefore,

$$\mathsf{CMSta} \;=\; \mathsf{MSta} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole})$$
$$\mathsf{CMSig} \;=\; \mathsf{MSig} \times (\mathsf{CType} \times \mathsf{CDisc} \times \mathsf{CRole})$$

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:

1. 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.

2. 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) that may affect the parameters for execution of the communication primitives and, therefore, play a role in the coordination layer.

The slicing stage involved in the construction of a CDG uses a backward slicing algorithm similar to the one presented in [16]. 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 slicing criterion is, as expected, the set of vertices marked by the matching with the regular expressions in the rules considered. 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 pruning stage. Actually, along the process some vertices are pruned from the graph, in particular vertices which were intermediaries, with respect to ct dependencies. Later such dependencies have to be rebuilt. This procedure may seem weird, but it ensures that future traversals of the graph are performed with the correct control order of statements.

A comment is now in order with respect to the effective reduction of size on detaching a CDG from a MSDG. Actually this can not be accurately predicted as it depends on the amount of coordination logic present on the specific code under analysis, *i.e.*, the volume of interaction measured by the number of statements with calls to communication primitives and their cohesion in the entire system. The higher the latter the less significative the

graph reduction will be. However, in most cases, only a small portion of code is dedicated to coordination, which makes the generated CDG considerably smaller than the original MSDG.

4. Generation of coordination scripts

Once a CDG is built, the recovery of the system's coordination model can be achieved in two different ways, as explained in section 1. This section discusses one of those methods: the direct generation of *coordination script* in Orc, therefore abstracting the whole relevant behaviour into a single specification.

The Orc coordination language is briefly described in the sequel, whereas subsection 4.2 introduces the script generation process.

4.1. Specifying coordination in Orc

Purpose and syntax.. Orc [23] is a simple, executable, yet powerful language designed for task orchestration. In brief, it introduces a platform for specification of protocols involving the access to external resources and services to accomplish specific goals, while managing concurrency, failure, time-outs, priorities and other constrains. The language builds upon a few basic combinators for expressing such coordination logic of applications. A number of formal semantics have been proposed for Orc [14, 18, 3], providing a solid theoretical background upon which one can base equivalence and program transformation.

External services and components are abstracted as *sites* which can receive calls from a coordination script written in Orc. Such a script, called an *orchestration*, consists of a set of auxiliary definitions and a main goal expression. The syntax is presented in figure 2.Notation $\overline{p}$ refers to a list of p elements separated by commas.

$$
\begin{array}{rcll}
e, f, g, h & \in & \textit{Expression} & ::= \quad M(\overline{p}) \parallel E(\overline{p}) \parallel f > x > g \parallel f|g \parallel \\
 & & & \quad f \textbf{ where } x :\in g \parallel x \\
p & \in & \textit{Actual} & ::= \quad x \parallel M \parallel c \parallel f \\
q & \in & \textit{Formal} & ::= \quad x \parallel M \\
 & & \textit{Definition} & ::= \quad E(\overline{q}) \triangleq f
\end{array}
$$

Figure 2: Orc syntax

An Orc expression can consist of a site call $M(\overline{p})$, an expression call $E(\overline{p})$, a sequential execution of expressions $f > x > g$, a parallel execution of expressions $f|g$, an asymmetric parallel composition of expressions f **where** $x :\in g$, or a variable x. The language includes a number of pre-defined sites which are essential for effective programming of real world orchestrations. Their informal semantics is presented in Table 1.

$let(x, y, ...)$	Returns argument values as a tuple.
$if(b)$	Returns a signal if b is true, and it does not respond if b is false.
$Signal$	Returns a signal. It is same as $if(true)$
$RTimer(t)$	Returns a signal after exactly t time units

Table 1: Fundamental Sites

Orc scripts can also represent dynamic orchestrations, i.e., orchestrations that are able to create local sites at runtime. This feature is provided by the so-called *factory sites*, which return a local site when invoked [9]. Table 2 describes some of those sites which will be used in the sequel to express the coordination logic recovered from legacy code. The *Buffer* site returns a n-buffer local site with two operations, *put* and *get*. The *put* operation stores its argument value in the buffer and sends a signal after the storage. The *get* operation removes an item from the buffer and returns it. In case the buffer is empty the *get* operation suspends until a value is putted in the buffer. The *Lock* factory site, on the other hand, returns a *lock* local site providing two operations, *acquire* and *release*. When an orchestration calls *acquire* on a *lock* it becomes its owner and subsequent calls to *acquire* from other expressions will block. Once the *lock* is released ownership is given to a waiting orchestration if any.

Site	**Ports**
Buffer	*put* and *get*
Lock	*acquire* and *release*

Table 2: Factory sites

Informal semantics.. A site in Orc is an autonomous entity with the capacity of publishing values to the calling expressions. The evaluation of a site call holds indefinitely (possibly forever if the the site never publishes a value) until the called site publishes a value. An expression call simply passes the control from the current expression being evaluated to the called expression with the associated parameters. A sequential composition of Orc expressions $f > x > g$ is executed by evaluating expression f, binding the value published by f to x and then evaluating expression g, which eventually contains references to x. In case x is not used by g, sequential composition abbreviated to $f >> g$. Parallel composition amounts to concurrent execution of its parameters. Finally, asymmetric parallel composition f **where** $x :\in g$ forces the evaluation of both f and g in parallel; the evaluation of f, however, holds whenever it depends on x and this variable is not instantiated by a values published by g. Once g publishes a value, its evaluation is halted and the value produced is stored in x, enabling expression f evaluation to continue. For a formal semantics of the language see [14, 18, 3].

Table 3 presents a number of elementary coordination scripts upon which the Orc generation process to be described in next subsection builds up.

$$
\begin{array}{lcl}
XOR(p,f,g) & \triangleq & if(p) \gg f \mid if(\neg p) \gg g \\
IfSignal(p,f) & \triangleq & if(p) \gg f \mid if(\neg p) \\
Loop(p,f) & \triangleq & p > b > IfSignal(b, f \gg Loop(g,f)) \\
Discr(f,g) & \triangleq & Buffer > B > ((f \mid g) > x > B.put(x) \mid B.get)
\end{array}
$$

Table 3: Orc Definitions

A brief explanation is in order. The *XOR* orchestration takes as arguments a predicate expression p, and two orchestrations f and g. Orchestrations f or g are executed depending on to which value p evaluates. Note that, in spite of the parallel combinator, the definition only executes one of the expressions, one of them being always, effectively executed.

Orchestration $IfSignal$ receives a predicate and an orchestration and executes the orchestration if the predicates evaluates to $true$. Again, notice that whether p evaluates to $true$ or $false$ the definition never blocks, but always publishes a value, thus permitting the calling orchestration to proceed.

Loop also receives a predicate p and an orchestration f. It executes f continuously until p evaluates to $false$. Even in this case the orchestration does not block, returning instead a signal to allow the calling orchestration

to proceed.

Finally, orchestration $Discr$ makes use of the factory site $Buffer$ in order to capture the signal of the first of its two parameter orchestrations to respond. Once one of the orchestrations returns a signal, this is forwarded to the calling orchestration, while leaving the other parameter executing until it terminates.

4.2. Orc *generation process*

A main concern in generating Orc coordination scripts is to assure their structure, in particular, the nested structure of calls, is close enough to that of the original system, therefore making easy further comparisons. Therefore an Orc definition is generated for each procedure in the CDG, even though some of these procedures have no other objective than calling other services, and may therefore be replaced by direct calls. The calling structure involving these calls recorded in the graph is also kept in the generated script. Actually, it is this structure preservation goal that justifies the first exception in the pruning process mentioned in the previous section.

Note, however, that the process does not generate an Orc definition for every procedure in the original system, since during the construction of the CDG all procedures not contributing to the coordination layer, were dropped. Also notice, that it is simple to transform the nested script into a flat one, whenever this simplifies formal analysis. Such a transformation resorts to an algebra of Orc orchestrations well documented in the literature [9].

The generation of an Orc script for a procedure is based on the program captured by the procedure sub-graph of the entire system CDG. The construction of the overall coordination script represented by a CDG basically amounts to collecting the statements of the vertices visited by following the control flow edges. The result is expressed in the language summarised in figure 3 (see [27] for a detailed presentation of both the language and the translation process).

This language, actually a subset of $C^{\natural}$, is self explanatory. Notice the use of $\prec \succ$ brackets for optional expressions.We consider that a local procedure call behaves like a synchronous call to a local resource, therefore not involving any form of communication. Every asynchronous procedure call, on the other hand, is be performed as if directed to an external resource. Thus, it must specify the resource site uniquely (internal asynchronous procedure calls may be performed using the `ASYNCCALL` construct with `localhost` as resource site). Two possibilities are provided for dealing with asynchronous

$$
\begin{array}{rcll}
z & \in & \textit{Values} & \\
x, x_1, x_n & \in & \textit{Variables} & \\
s & \in & \textit{Sites} & \\
e, e_1, 2_n & \in & \textit{Expressions} & \\
st, st_1, st_2 & \in & \textit{Statements} & ::= \quad z \\
 & & & \mid \quad x \\
 & & & \mid \quad x \texttt{ = } e \\
 & & & \mid \quad st_1 \texttt{ ; } st_2 \\
 & & & \mid \quad \texttt{LOCK (}x\texttt{) \{}st\texttt{\}} \\
 & & & \mid \quad \texttt{LOCALCALL } f(\overline{x}) \\
 & & & \mid \quad \texttt{SYNCCALL } s\ f(\overline{x}) \\
 & & & \mid \quad \texttt{ASYNCCALL } s\ f(\overline{x}) \prec \{st\} \succ \\
 & & & \mid \quad \texttt{IF } p \texttt{ THEN } \{st_1\} \prec \texttt{ELSE } \{st_2\} \succ \\
 & & & \mid \quad \texttt{WHILE } p \texttt{ DO } \{st\} \\
f_1, f_n & \in & \textit{Procedures} & ::= \quad f(\overline{x})\{st\} \\
c_1, c_n & \in & \textit{Classes} & ::= \quad c\ \{x_1 = e_1\ ...\ x_n = e_n\ f_1\ ...\ f_n\} \\
ns_1, ns_n & \in & \textit{Namespaces} & ::= \quad ns\ \{c_1\ ...\ c_n\}
\end{array}
$$

Figure 3: The CDG representation language

calls. One of them simply launches the procedure call in a separate thread and continues execution of the rest of the program. The other executes an expression when and if the asynchronous call returns. The `LOCK` statement behaves as expected i.e., it gives a variable access to a specific statement block execution in a single thread or process.

The generation of an Orc script proceeds in two phases. The first one is performed by function ψ which identifies all the variables in the language for which an access control may be required, and sets up an environment for controlling the access to such variables. Basically, function ψ, typed as $\psi : C \times \mathbb{P}\ V \to (1 + Orc) \times \mathbb{P}\ V$ introduces a *Lock* site (see tables 4.1) for each variable in a `LOCK` statement, while keeping track of all visited variables for avoiding site duplication. In its definition projections of product datatypes are represented by $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. dually, the notation for embeddings in sum datatypes is $\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$. Formally, $\pi_1(x, y) = x$ and $\iota_1(x) = (1, x)$, where 1 is a label which identifies the origin of x in the sum.

$$
\begin{array}{lll}
\psi\ (\texttt{LOCK (}x\texttt{) \{}st\texttt{\}},\ V) & \equiv \begin{cases} (\iota_2(Lock > xLock > Signal), \{x\} \cup V) & \text{if } x \notin V, \\ (\iota_1(), V) & \text{otherwise} \end{cases} \\
\psi\ (\texttt{ASYNCCALL } s\ f(\overline{x})\ \{st\},\ V) & \equiv (\psi_1\ st,\ \psi_2\ st \cup V) \\
\psi\ (\texttt{IF } p \texttt{ THEN } \{st\},\ V) & \equiv (\psi_1\ st,\ \psi_2\ st \cup V)
\end{array}
$$

$$
\begin{array}{ll}
\psi\ (\texttt{IF}\ p\ \texttt{THEN}\ \{st_1\} & \\
\qquad \texttt{ELSE}\ \{st_2\},\ V) & \equiv \begin{cases}
(\psi_1\ st_1,\ \psi_2\ st_1 \cup V) & \text{if } \psi_1\ st_1 \neq \iota_1() \wedge \\
& \quad \psi_1\ st_2 = \iota_1(), \\
(\psi_1\ st_2,\ \psi_2\ st_2 \cup V) & \text{if } \psi_1\ st_1 = \iota_1() \wedge \\
& \quad \psi_1\ st_2 \neq \iota_1(), \\
(\iota_2(\psi_1'\ st_1 \gg \psi_1'\ st_2), & \text{if } \psi_1\ st_1 \neq \iota_1() \wedge \\
\quad \psi_1\ st_1 \cup \psi_1\ st_2 \cup V) & \quad \psi_1\ st_2 \neq \iota_1(), \\
(\iota_1(), V) & \text{otherwise}
\end{cases} \\
\psi\ (\texttt{WHILE}\ p\ \texttt{DO}\ \{st\},\ V) & \equiv (\psi_1\ st,\ \psi_2\ st \cup V) \\
\psi\ (st1\ \texttt{;}\ st_2,\ V) & \equiv ((\iota_2(\psi_1'\ st_1 \gg \psi_1'\ st_2),\ \psi_2\ st_1 \cup \psi_2\ st_2 \cup V) \\
\psi\ (st,\ V) & \equiv (\iota_1(),\ V)
\end{array}
$$

where $\psi_1 = \pi_1\ .\ \psi$, $\psi_2 = \pi_2\ .\ \psi$, $\rho_2(\iota_2\ x) = x$ and $\psi_1' = \rho_2\ .\ \pi_2\ .\ \psi$. Note that the type of the result of $\pi_2\ .\ \psi$ is a sum, which makes the whole expression $\rho_2\ .\ \pi_2\ .\ \psi$ well-typed; function ρ_2 simply removes the sum label. The second phase in the method is performed by function $\varphi : C \to Orc$ which, for every procedure body, generates the corresponding Orc definition. Note that function φ assumes the existence of a previously created environment of sites, more specifically an environment with a *Lock* controlling the access to each critical variable.

$$
\begin{array}{lcl}
\varphi\ \texttt{z} & \equiv & let(z) \\
\varphi\ \texttt{x} & \equiv & x \\
\varphi\ \texttt{x = } e & \equiv & let(e) > x > Signal \\
\varphi\ \texttt{x = } e\ \texttt{;}\ st_2 & \equiv & let(e) > x > \varphi(st) \\
\varphi\ \texttt{LOCK (}x\texttt{) \{}st\texttt{\}} & \equiv & xLock.acquire \gg \varphi(st) \gg xLock.release \\
\varphi\ \texttt{LOCALCALL}\ f\texttt{(}\overline{x}\texttt{)} & \equiv & F(\overline{x}) \\
\varphi\ \texttt{SYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)} & \equiv & s.F(\overline{x}) \\
\varphi\ \texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{)} & \equiv & Discr(s.F(\overline{x}), Signal) \\
\varphi\ \texttt{ASYNCCALL}\ s\ f\texttt{(}\overline{x}\texttt{) \{}st\texttt{\}} & \equiv & Discr(s.F(\overline{x}) > result > \varphi(st), Signal) \\
\varphi\ \texttt{IF}\ p\ \texttt{THEN \{}st\texttt{\}} & \equiv & IfSignal(let(p), \varphi(st)) \\
\varphi\ \texttt{IF}\ p\ \texttt{THEN \{}st_1\texttt{\} ELSE \{}st_2\texttt{\}} & \equiv & XOR(let(p), \varphi(st_1), \varphi(st_2)) \\
\varphi\ \texttt{WHILE}\ p\ \texttt{DO \{}st\texttt{\}} & \equiv & Loop(let(p), \varphi(st)) \\
\varphi\ st1\ \texttt{;}\ st_2 & \equiv & \varphi(st_1) \gg \varphi(st_2)
\end{array}
$$

Notice the difference between variables x and z in the definition of function φ: the former is a meta-variable (storing itself a variable from the CDG representation language) whereas the second is a value.

Function φ converts a value or a variable from the language to the correspondent variable or constant in Orc. A synchronous procedure invocation is translated to a site call in Orc.

The asynchronous procedure call case is not as straightforward as the previous ones. The envisaged behaviour involves a non-blocking request to a site to return an answer. This behaviour can be captured in Orc by the *Discr* orchestration pattern, described in the previous sub-section, and the fundamental site *Signal*. Recall that *Discr* executes both arguments in parallel and waits for a signal from any of the sites. Since *Signal* publishes a signal immediately, the behaviour of the *Discr* with a *Signal* argument is to return immediately leaving the other argument to execute in parallel.

Given the blocking behaviour of the fundamental site if when faced with a $false$ value, one cannot perform a direct translation of the `IF THEN` statement to the if Orc fundamental site. Such a direct translation would make the entire specification to block upon a $false$ value over an if site. We resort instead to the $IfSignal$ pattern that never blocks and executes the second expression if the predicate evaluates to *true*.

The behaviour specification of the `IF THEN ELSE` statement is easier to capture because one of the branches of the statement is always executed. Therefore, a direct translation to the XOR orchestration pattern is enough. Similarly, the `WHILE DO` statement is captured by the *Loop* coordination pattern which does not block upon evaluation of false predicates.

Given functions ψ and φ, specifying the two main phases of the Orc generation process, the overall generation algorithm is obtained as follows:

$$\beta\ (f(\overline{x})\ \{st\}) = \begin{cases} F(\overline{x}) \triangleq \psi_1'(st, \varnothing) \gg \varphi(st) & \text{if } \psi_1(st, \varnothing) \neq \iota_1() \\ F(\overline{x}) \triangleq \varphi(st) & \text{otherwise} \end{cases}$$

5. Discovering coordination patterns

5.1. Describing Coordination Patterns

In contrast to 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, recovering a specification of such a layer is, usually, far from trivial. The Orc script extracted by the process just described is often too long and unstructured,

as its generation follows strictly the CDG strucutre. An alternative method is described in this section. It is driven by a series of predefined coordination patterns encoded as sub-graph instances whose presence is systematically investigated in the CDG under analysis. Formally, these coordination patterns are defined as pairs grouping together a *matching condition* (of type PCondition) and a graph pattern. Formally,

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

The first component is a *matching condition* defined as a mapping which associates to each pattern node (of type NodeId) a predicate over CDG nodes (of type GNode). In practice, a common definition of such a predicate resorts to a regular expression intended to be tested over the statement collected on CDG nodes. Symbol *tt* 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. Notice that PathPattern is either a positive natural number or the annotation +, standing, as usual, for one or more. it is also assumed that all nodes in the 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 [27] for a complete specification. The discovery process, in particular, is guided by 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. 4.

The coordination patterns were selected based on empirical data gathered

from a number of case studies in which the method, and accompanying tool, has been applied. In each pattern, edges are labeled with the amount of actual edges to be found in the CDG and a variable to be bound to the corresponding thread id. Vertices contain regular expressions to be matched with the statements inside each CGD vertex.

5.1.1. Synchronous Sequential Pattern

This is one of the most simple patterns in which external services in a row are invoked one after the other. The pattern is usually found in presence of a chain of service dependencies, *i.e.*, when a service call depends on the response to a previous one.

This pattern is specified as in Fig. 4(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 of these vertices can be defined by the following predicate template:

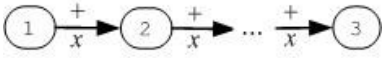
$$\begin{aligned} pc(x) = \; & x == (\mathsf{MSta}(t, s), cp, cm, cd) \Rightarrow \\ & match(s, \text{“}\texttt{ServiceCall(*)}\text{”}) \wedge cp == \underline{\mathsf{webservice}} \\ & \wedge cm == \underline{\mathsf{sync}} \wedge cd == \underline{\mathsf{consumer}} \end{aligned}$$

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

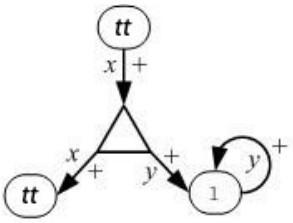
5.1.2. Cyclic Query Pattern

This pattern is characterized by a point in which a new thread is spawned, becoming responsible for invoking an external web service cyclically (which explains loop in vertex 1). This is often used in applications which have to monitor the state of a foreign resource or must be constantly updating an internal resource which depends upon an external service.

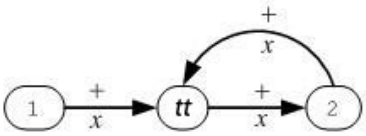
In practice, this pattern appears in several variations. For instance, it may include a time delay between each cyclic service call or use different strategies to implement the service invocation cycle, e.g. recursion or iteration. The version presented in Fig. 4(b) captures its most generic version. It basically states that a new thread y must be spawned and that, under the execution of such a new thread, a service must be called repeatedly. Again, vertex 1 is instantiated with a predicate, as in the previous patterns, constraining the kind of services that can be invoked.



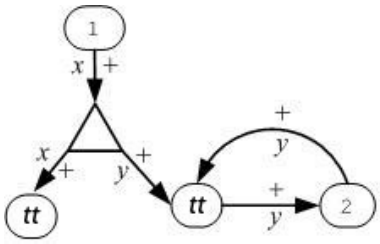
(a) Synchronous Sequential Pattern



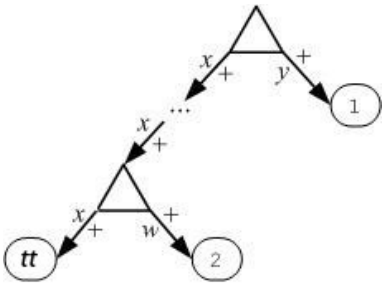
(b) Cyclic Query Pattern



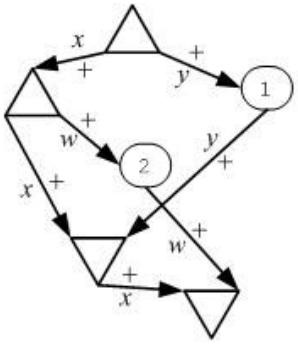
(c) Asynchronous Query Pattern



(d) Asynchronous Query Pattern with Client Multithreading



(e) Asynchronous Sequential Query Pattern



(f) Joined Asynchronous Sequential Pattern

Figure 4: *CDGPL Patterns*

Note the pattern does not define when the loop ends, but simply states that the right thread is responsible for a recurring invocation of an external service. It does not specify either if the cyclic behaviour, represented by a loop in the pattern, is due to recursion or iteration. Actually, the way such a behaviour is implemented is not relevant: it is enough to record its effects.

5.1.3. Asynchronous Query Pattern

The Asynchronous Query Pattern is usually employed when time consuming services and to be called, and calling threads can not suspend until a response is returned. Typically, in such cases, the server component provides two methods, one for requesting of an operation on the server and another for retrieving a reply (if available) from a previous request. Both these 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 in Fig. 4(c), which encodes the invocation of a service to request the execution of some operation (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 characterised by predicates that specify the precise operation one is looking for.

5.1.4. 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. 4(d), is also quite similar to the cyclic pattern, but for an extra node, marked with $*$, to represent the program statement controling the need to perform more invocations to retrieve the result of an operation.

5.1.5. Asynchronous Sequential Pattern

This is similar to the *Synchronous 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 being irrelevant. Note that, under this premises, this pattern is substantially faster than the *Synchronous Sequential Pattern* in the invocation of the series of services. Thepattern is specified in Fig. 4(e) where each of the service calling nodes (1 and 2) is invoked in a different thread (y and w respectively).

```
conds :: PCondition
conds 1 = \x -> x == fork
conds 2 = \x -> x == fork
conds 3 = \x -> cond1 x
conds 4 = \x -> cond2 x
conds 5 = \x -> x == join
conds 6 = \x -> x == join

edges :: [(NodeId, ThreadId, NodeId, PathPattern)]
edges = [(1, "x", 2, "+"),(1, "y", 3, "+"),(2, "w", 4, "+"),
         (2, "x", 5, "+"),(3, "y", 5, "+"),(4, "w", 6, "+"),
         (5, "x", 6, "+")]
```

Figure 5: Textual representation of the *Joined Asynchronous Sequential Pattern*

5.1.6. Joined Asynchronous Sequential Pattern

In this pattern services are also invoked asynchronously. But now there is an explicit interest in controlling the point where each of the services called finishes execution and, possibly, returns a value. The specification is given in Fig. 4(f) where each thread that was spawned to invoke a service, joins later at a point where the execution may proceed with the guarantee that the execution of all service calls has already finished.

Fig. 5 represents this same pattern in the textual version of the pattern language. This is expected to give a flavour of the concrete syntax, with a strong Haskell flavour, used in the tool described in section 6. Notice that annotation $\triangle$ is represented by `fork`, $\triangledown$ by `join`, whereas generic conditions from vertices 1 and 2, with type 2^{GNode}, are presented by `cond1` and `cond2` respectively.

5.2. The Pattern Discovery Algorithm

The algorithm presented in this sub-section discovers and retrieves any sub-graph of a CDG conforming to a given *graph pattern*. The notation used is 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. 6, also expressed in the HASKELL syntax for data type declarations. Note that 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.

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

Figure 6: Data types for the graph pattern discovery algorithm

The overall strategy used by the pattern discovery algorithm[2] consists of traversing the graph pattern, starting from its root, and incrementally building a list of candidate graphs with nodes of type Attribution. This type is used by the algorithm in order to maintain a mapping between the graph pattern nodes and the CDG matching nodes. If the graph pattern vertex is found for which a candidate graph cannot be extended 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 extended with one of the 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.

The purpose of most auxiliary functions in the algorithm is easy to grasp, 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. Note that, whenever there are vertices in the pattern graph which are not reachable from the root vertex, one must re-iterate the discovery process (line 26) based on the first not visited vertex.

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 speci-

[2]The complete algorithm implementation in $C^\sharp$ is available at `http://alfa.di.uminho.pt/~nfr/PhDThesis/SubGraphIsomorphismAlgorithm.zip`

Algorithm 1 Pattern Discovery

```
1:  function DiscoverPattern(Graph cdg, GraphPattern cdgp)
2:      cdgp ← FillCandidateVertices(cdg, cdgp)
3:      cdgp ← FillCandidateEdges(cdg, cdgp)
4:      Graph baseGraph                          ▷ Initial empty graph to base the discovery process
5:      [Attribution] rootAtts ← getAttributions(cdgp)   ▷ Attributions list for the root vertex pattern
6:      [Extension] gel ← [(baseGraph, rootAtts)]                    ▷ Initial graph and attributions
7:      repeat
8:          𝔹 b ← False
9:          for all Extension ge in gel do
10:             for all Attribution datt in ge.att do
11:                 datt.vp.visited ← True
12:                 c1 ← HasSuccessors(cdgp, datt.vp)   ▷ Check if cdgp has successors for vertex datt.vp
13:                 c2 ← HasSuccessors(ge.g, datt.vp)   ▷ Check if the discovered graph ge.g has any already successors for the vertex pattern datt.vp
14:                 if c1∧ !c2 then ▷ If the graph pattern has successors for attribution datt and the discovered graph doesn't have any, then the discovered graph must be extended
15:                     [Extension] dgel ← ExtendBaseGraph(ge.g, datt)
16:                     [Extension] r ← ge : r
17:                     [Extension] a ← dgel + + a
18:                     b ← b ∨ Length(dgel) > 0
19:                 end if
20:             end for
21:         end for
22:         gel ← Remove(gel, r)          ▷ Remove the initial (incomplete) discovered graphs from gel
23:         gel ← gel + + a                          ▷ Add all the recently extended graphs to gel
24:         r ← []
25:         a ← []
26:         nv ← NotVisited(cdgp)                          ▷ Get first not visited Vertex Pattern
27:         if b ∧ nv ≠ null then
28:             b ← True
29:             vpa ← map (λx → (nv, x)) nv.cdts      ▷ Initialize attributions for the non visited vertex
30:             map (λx → (x.g, vpa)) gel      ▷ Extend the discovered graphs with the vpa attributions
31:         end if
32:     until b == True
33:     return gel
34: end function
35:
36: function ExtendBaseGraph(Graph bg, Attribution att)
37:     tcs ← GetSuccCombinations(cdgp, vp)
38:     for all tc in tcs do
39:         ng ← bg
40:         gel ← (ge, [])
41:         for all cv in tc do
42:             if b ∧ nv ≠ null then
43:                 AddEdge(ng, att, cv)
44:                 ge.DiscoveredAttributions.Add(cv)
45:             else
46:                 gel.Remove(ge)
47:                 break
48:             end if
49:         end for
50:     end for
51:     return gel
52: end function
```

fication of the system whose code has been analysed.

6. CoordInspector

In practice recovering a model of the coordination layer of an application is not a trivial task. Actually, this has to cope, simultaneously, with the size of the source code to be analysed, the heterogeneity of languages and technologies employed and the specific level of coordination (inter-thread coordination, component coordination, services coordination, etc) relevant to each particular case. Moreover, the result of the recovery process should be expressed, as much as possible, in terms of well-known coordination patterns, and clear specifications of the coordination policies, in order to make easier their analysis, re-engineering and reuse. This entails the need for suitable tool support.

Such is the purpose of CoordInspector, a prototype tool developed as "proof-of-concept" for the techniques discussed above. The tool, a snapshot of which is presented in appendix A, is available from `http://alfa.di.uminho.pt/~nfr/Tools/CoordInspector.zip`. A preliminary version is reported in [28].

6.1. Architecture and implementation

A basic choice in CoordInspector design was to make it as generic as possible. Therefore, it targets Common Intermediate Language (CIL)[22] code, the native language of Microsoft .Net Framework, to which every .Net compilable language ultimately gets translated to before being executed in the framework. The decision to target CIL code was not arbitrary: the design aim was to cope with as many programming languages as possible, as most systems resort to several different programming languages. Moreover, given the potential of the tool to assist legacy systems evolution, this sort of "language agnosticism" becomes even more important. Thus, by choosing CIL, the tool is presently able to analyse more than 40 programming languages, and this number has only but potential to increase.

In order to take advantage of existing CIL analysis tools, CoordInspector is developed as a plug-in to the CIL decompiler .Net Reflector[3]. In particular, the implementation takes advantage of the parser for CIL code,

[3] `http://www.aisto.com/roeder/dotnet`

which delivers an object tree representation of the CIL abstract syntax tree, and the code representation plug-ins, which transform CIL code into higher-level languages, like $C\sharp$ and C++.

Such a tree is then processed to build the corresponding MSDG instance. Given the intrinsic modularity of this process, it is executed by different components that are responsible for the calculation of each of the MSDG constituents, i.e., the nodes representing statements and all sorts of dependencies between them, as detailed in section 2. Each component traverses the concrete syntax tree and collects the relevant information for the construction of a particular graph. By multithreading the independent tasks which build each of the MSDG set of dependencies, the overall performance of the tool improved significantly.

The CDG calculation implemented by CoordInspector closely follows the approach presented in section 3, starting with labelling the vertices based on rules identifying communication primitives and, then, pruning the vertices according to the strategy presented in the same section. Specific instances of the rule set to identify Web-Services communications, distinguish between synchronous and asynchronous calls as well as between invocation and provisioning of functionality using Web-Services, are available in the accompanying library.

The graph pruning and slicing operations were implemented according to the specifications presented in section 3, based on a number of graph traversal and transformation algorithms.

The tool is currently able to generate Orc specifications, corresponding to the discovered coordination logic. For this, it closely follows the algorithm presented in section 4. The tool is also able to re-construct the analysed code, i.e., the code represented by the calculated CDG instance, which focuses the specific aspects determined by the set of rules used. For this feature, CoordInspector uses the code representation plug-ins, available for .Net Rotor. This includes $C\sharp$, Visual Basic, MC++, Chrome, Delphi and, of course, CIL itself.

CoordInspector is also able to depict and navigate through both the calculated MSDG and CDG graphs. For this, the tool resorts to the *Microsoft Research Graph Layout Execution Engine* (MSR GLEE) graph library. The generated graphs provide different colours for the vertices, based on the labels the vertices hold, which facilitates direct manipulation of such structures. Double clicking on a particular vertex displays all the associated information, for example, the corresponding labelling and the CIL code it abstracts.

The architecture of CoordInspector is depicted in Figure 7 as a typical box component diagram, representing the main components into which the implementation is divided.

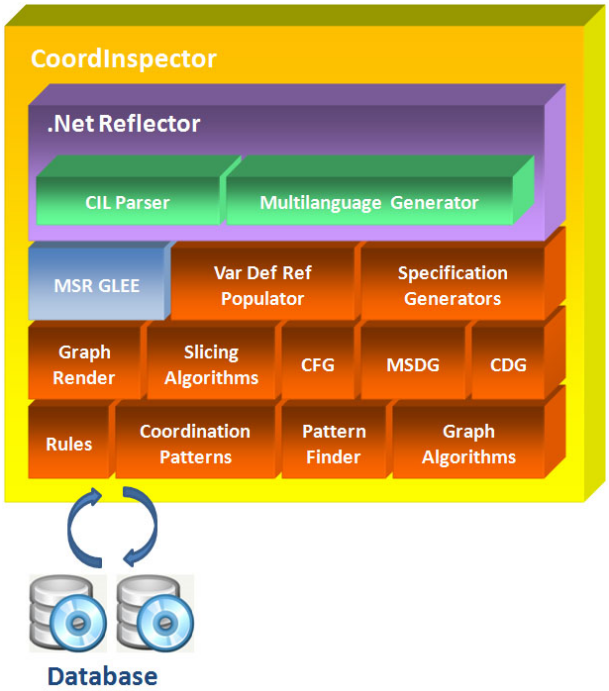


Figure 7: CoordInspector architecture

Reading the diagram from top to bottom, the first component block represents .Net Reflector, exposing the *CIL parser* and the *Multilanguage Generator* sub-components. The *MSR GLEE* component is used for the graphic layout of all graphs calculated during the analysis process. This component is completely isolated from the others and uses a graph representation that is different from the ones used internally by CoordInspector for instance, to capture MSDG's and CDG's. Therefore, whenever a component has to display graphically a CDG or MSDG, it resorts to the *Graph Render* component, which is responsible for translating the CoordInspector internal graph representations to the representation used by the *MSR GLEE* component.

Apart from .Net Reflector and *MSR GLEE*, all the remaining components were developed specifically for CoordInspector. The *CFG* com-

ponent interprets the abstract syntax tree retrieved by the *.Net Rotor CIL Parser* and extracts the control flow graph by translating the base language control flow statement semantics into a graph representation. The *Var Def Ref Populator* component, navigates back and forth along the CFG (resorting to the *Graph Algorithms* component) in order to calculate, for each CFG vertex, the set of variables defined and used in each programming construct contained in the vertex. This information is vital for the *MSDG* component which is responsible for the calculation of the MSDG, following closely the process explained in section 2.

The calculation of the CDG is, of course, the responsibility of the component *CDG* which makes use of the rules stored in the rule management component *Rules*, together with the *slicing* component, in order to prune the MSDG according to the strategy defined in section 3. As expected, the *Rules* component is responsible for the CRUD (*i.e.*, create, read, update and delete) operations for rules, using a XML database for this matter.

The *Specification Generators* component contains a set of sub-components for generating specifications in different coordination formalisms. Each of them contains an abstract representation of a particular specification language, and often resorts to the *Graph Algorithms* component for traversing and consuming the CDG. For now, the *Specification Generators* component is populated by a single sub-component, responsible for the generation of Orc specifications.

Besides consuming the CDG in order to generate coordination specifications, the tool is also able to discover previously defined coordination patterns. For this matter, CoordInspector uses the *Pattern Finder* component, which implements the coordination pattern discovery algorithm presented in section 5. The coordination patterns used in this task are managed by the *Coordination Patterns* component, which implements the corresponding CRUD operations and uses an XML database for permanent storage of patterns.

6.2. Using CoordInspector

After populating the tool with the system under consideration, analysis is launched by calling option *Tools* → *Coordination Analysis*. The display is similar to the one shown in Appendix A.

For performing analysis, one has to select the programming entity upon which the process will begin, by choosing a node in the programming entities tree (area 7 in in Appendix A.). Once selected, its details appear in area 8,

and the user may click the button in area 4 to start the MSDG calculation. During the MSDG calculation, area 6 will provide information about progress and details on the calculation process. Once CoordInspector finishes calculating and rendering the MSDG, the graph is displayed in area 5, which can be inspected by the graphical operations provided in area 3. The user may perform this same operation over other program entities displayed in the tree, which allow him to inspect the different MSDG's of the application under consideration.

Once the MSDG has been calculated, the user may proceed to the CDG calculation, by clicking on a button similar to the one presented in area 4, but this time in the tab *CDG* displayed in area 2. Again, area 6 will provide information and report progress of the calculation process. At any time during the analysis process, the user may change the rules upon which the CDG is calculated, by using the rules management interface provided in *Rules* tab.

Finally, the user may generate an Orc specification based on the calculated CDG, first by accessing the *Orc* tab in area 2, and then by clicking on the *Generate Orc* button. The Orc specification is provided in the central area of the *Orc* tab. The discovery of coordination patterns follows a similar interface.

7. An example

This section illustrates the application of the method proposed in the paper, in the pattern identification variant discussed in section 5. For this consider the $C^\natural$ code fragment in appendix C.

This code is supposed to 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 held until a response from the server is returned. 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 prediction from the server it inspects the result and, if not satisfied (method `CheckPrediction`), submits a new request to reevaluate the prediction. Note

that the class `WeatherServer` is the web-service proxy class, automatically generated by the tool *Microsoft. VSDesigner*.

By the description of the client behaviour, the analyst may suspect this client probably implements one or more instances of the *Asynchronous Query Pattern* mentioned above. Another issue is to identify exactly which statements in the source code are responsible for such an implementation. This is often non trivial because often the same coordination pattern admits several different implementations, eventually with minor deviations. Moreover, in real cases, such implementations are found interleaved with other code and spread among different components of the system.

Due to space limitations we omit some code details, which are clearly identified by underlined comments. Two such omissions are concerned with the construction of the parameters being passed to the server operations (lines 15 and 31), which amounts to gathering of the current weather conditions. The third omission (line 52) concerns the code to setup the web service proxy class, which contains the code to control all 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. The result is shown in Appendix B. To maintain the readability of the graph, only control, method call, control flow, and formal-in and out dependencies are shown.

The following phase computes the CDG out of the MSDG. In this example we are interested in identifying synchronous calls to web services. Such identification is performed by the rule (“`Invoke(*);`”, (webservice, sync, Consumer)), which identifies web-services calls made by the *Microsoft. VSDesigner* tool.

The computation of the CDG, as explained in section 3, leads to the elimination of code lines 3, 7, 8, 9, 10, 14, 24, 26, 30, 39 and 42 or, in graphical terms, to the elimination of the dashed vertices in the figure in Appendix B. Note that the removed statements are exactly the ones not directly involved in the invocation of web-services, which, in this toy example, almost entirely corresponds to IO statements. Nevertheless, in a real system, the percentage of program statements sliced out, with respect to the entire system, is certainly much higher.

The following phase is to define in CDGPL an expression that characterises the coordination pattern one is looking for. For this example, we resort to the definition presented in section 5.1, with the following *pc* pattern

condition:

$$
\begin{aligned}
pc(1) = \lambda(\mathsf{MSta}(t,s), cp, cm, cd) \rightarrow\ & (match(s, \text{“}GetForecast(*)\text{”}) \vee \\
& match(s, \text{“}ConfirmForecast(*)\text{”})) \wedge \\
& cp == \underline{\mathsf{webservice}} \wedge cm == \underline{\mathsf{sync}} \wedge cd == \mathsf{consumer} \\
pc(2) = \lambda(\mathsf{MSta}(t,s), cp, cm, cd) \rightarrow\ & match(s, \text{“}GetOperationResult(*)\text{”}) \wedge \\
& cp == \underline{\mathsf{webservice}} \wedge cm == \underline{\mathsf{sync}} \wedge \\
& cd == \underline{\mathsf{consumer}}
\end{aligned}
$$

The graph pattern discovery algorithm clearly identifies two instances of the *Asynchronous Query Pattern* in the code. They are highlighted by the two mappings f_1, f_2 between the vertex pattern identifiers and the example code line statements.

$$
\begin{array}{ll}
f_1(1) = 16 & f_2(1) = 32 \\
f_1(2) = 23 & f_2(2) = 38 \\
f_1(3) = 25 & f_2(3) = 40
\end{array}
$$

Note that, although they have been discovered as instances of the same coordination pattern, their implementations are quite different, resorting, in the first case, to a `while` loop, and, in the second, to a recursive call of method `GetForecastConfirmationResult`.

8. Conclusions and future work

This paper introduced a method that combines a number of program analysis techniques to extract system's coordination layer from legacy 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. [25] for an early reference. Our contribution was to extend previous work (namely [19, 24]) 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. A distinguished characteristic of this approach is its parametrisation by rules identifying specific communication primitives, thus making it adaptable to diverse kinds of coordination analysis and programming frameworks.

It should be mentioned that slicing of concurrent code is still a topic of active research. Reference [26] discusses challenges in finding the right slicing technique for reactive, concurrent systems, especially for liveness analysis. In designing CoordInspector we adapted object slicing techniques to handle concurrency through multithreading (which explains the use of *fork* and *join* vertices, as well as interference edges, in the coordination patterns). This proved enough for the method purposes, although failing, for example, in computing minimal slices.

Even though a full complexity analysis falls beyond the objectives of this paper, experience gathered in conducting a number of case studies allows a few comments on the scalability of the whole approach. In particular, the performance of the method's first stage, dealing with the construction of different graph structures, can be largely improved by multithreading the computation of the different kinds of dependencies involved. Plain sequential computation of these graphs, as used in the current version of CoordInspector, takes about 5 minutes to analyse a system with 20 KLOC in a PC with a core 2 duo 2,20GHz cpu and 2 GB of memory[4]. The scalability of this stage is linear with the size of the program under analysis. The pattern discovery algorithm performs reasonably well, always retrieving answers in less than 1 minute. The reason for such good response times is due to the vertex conditions (specified by regular expressions) and edge conditions (given by the number and threads associated to the edges), which in real world examples are often quite restrictive, leading to substantially reduced amounts of candidate vertices for each pattern vertex. Of course should the vertex conditions be relaxed (a situation rarely found in real systems analysis) the algorithm performance will be dramatically affected.

Although the most direct application of this approach is to assist the coordination analysis of legacy systems, it can also be used to assess the correctness of systems implementations with respect to their design specifications or even with respect to the independent software quality regulations. Furthermore, with the provision of rules capturing COM or RMI communication primitives, it can be used to assist the conversion of distributed object

[4]Time measured excludes visual rendering of the graphs in CoordInspector.

systems towards web-service oriented systems (or vice versa).

Systematic comparison of CoordInspector with tools pursuing similar goals is now in order. The *Fujaba Tool Suite RE*, in particular its *Pattern-Specification* and *InferenceEngine* plug-ins, shares a similar objective and has a powerful inference mechanism. It it tunned, however, to the discovery of UML sub-diagrams, in particular sub-class diagrams. Our approach, on the other hand, is clearly oriented towards the discovery of patterns in dependency graphs directly built from code inspection. Therefore, they have to keep track of details such as thread identifiers and complex regular expressions in the nodes.

Another interesting topic for future work is the classification of coordination patterns, as in [1], in terms of their graph representation expressed in CDGPL. Such a taxonomy could be taken as a basis for a repository of coordination patterns, relevant 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] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[3] M. AlTurki and J. Meseguer. Real-time rewriting semantics of orc. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming*, pages 131–142, New York, NY, USA, 2007. ACM.

[4] F. Arbab. What do you mean, coordination. In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI*, 1998.

[5] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[6] G. Canfora, A. Cimitile, A. D. Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *ICSM '94: Proceedings of the Interna-*

tional Conference on Software Maintenance, pages 424–433, Washington, DC, USA, 1994. IEEE Computer Society.

[7] G. Canfora, A. Cimitile, and M. Munro. Re^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, 6(2):53–72, 1994.

[8] A. Cimitile, A. D. Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance*, 8(3):145–178, 1996.

[9] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in orc. In P. Ciancarini and H. Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006.

[10] A. de Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 9, Washington, DC, USA, 1996. IEEE Computer Society.

[11] 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.

[12] M. Harman, R. Hierons, S. Danicic, J. Howroyd, and C. Fox. Pre/post conditioned slicing. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 138, Washington, DC, USA, 2001. IEEE Computer Society.

[13] R. Hierons and M. Harman. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9:233–262, 1999.

[14] T. Hoare, G. Menzel, and J. Misra. A tree semantics of an orchestration language, August 2004.

[15] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proc of the ACM SIGPLAN Conf. on*

Programming Usage, Design and Implementation, pages 35–46. ACM Press, 1988.

[17] D. L. M. Jean-Marc Andreoli, Chris Hankin. *Coordination Programming: Mechanisms, Models, and Semantics*. Imperial College Press, 1996.

[18] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.

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

[20] 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.

[21] 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.

[22] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development. Addison-Wesley Professional, 1 edition, November 2003.

[23] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Jour. of Software and Systems Modeling*, 2006.

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

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

[26] 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), 2007.

[27] N. F. Rodrigues. *Slicing techniques applied to architectural analysis of legacy software*. PhD thesis, DI, Universidade do Minho, 2009.

[28] N. F. Rodrigues and L. S. Barbosa. Coordinspector a tool for extracting coordination data from legacy code. In *SCAM '08: Proc. Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2008.

[29] D. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[30] D. Simpson, S. Valentine, R. Mitchell, L. Liu, and R. Ellis. Recoup—maintaining fortran. *SIGPLAN Fortran Forum*, 12(3):26–32, 1993.

[31] 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.

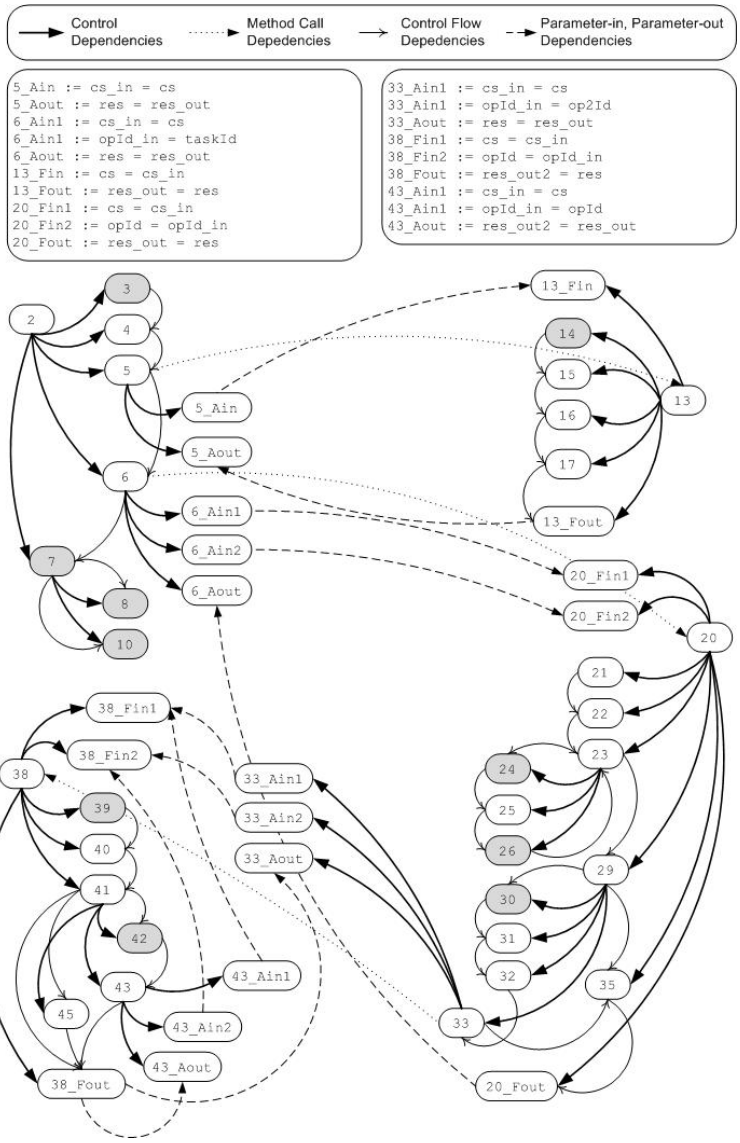
[32] M. Weiser. *Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Methods*. PhD thesis, University of Michigan, An Arbor, 1979.

[33] 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.

[34] J. Zhao. Applying program dependence analysis to java software. In *Proc. of Workshop on Software Engineering and Database Systems*, pages 162–169, 1998.

A. CoordInspector snapshot

B. MSDG (example section 7)



C. Code fragment (example section 7)

```
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   }
```