View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Universidade do Minho: RepositoriUM

Extracting and verifying coordination models from source code

Nuno F. Rodrigues[1] and Luís S. Barbosa[1]

CCTC, Universidade do Minho 4710-057 Braga, Portugal
{nfr, lsb}@di.uminho.pt

Summary. Current software development relies increasingly on non-trivial coordination logic for combining autonomous services often running on different platforms. As a rule, however, in typical non-trivial software systems, 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 which cannot be overestimated along any program understanding or refactoring process.

Open access to source code, as granted in OSS certification, provides an opportunity for the development of methods and technologies to extract, from source code, the relevant coordination information. This paper is a step in this direction, combining a number of program analysis techniques to automatically recover coordination information from legacy code. Such information is then expressed as a model in Orc, a general purpose orchestration language.

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

Such techniques find in the whole process of Open-Source Software (OSS) certification a promising application area, given that

- OSS applications often emerge by composition of multi-source, heterogeneous and previously unrelated pieces of code, which makes architectural recovery processes both useful and challenging;
- full access to source code enables the effective application of approaches and tools entirely based in code analysis.

Several approaches have been proposed for reverse architectural analysis. For example, *class diagram* generators which extract class diagrams from object oriented source code, *module diagram* generators that construct box-line diagrams from system's modules, packages or namespaces, *uses diagram* generators which reflect the import dependencies of the system and *call diagram* generators which expose the direct calls between system parts. However, none of these techniques/tools make it possible to answer a critical question about the dynamics of a system: *how do system's components interact, internally and with external services, and orchestrate their behaviours to achieve common, complex 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? and what is the sequence of actions for such a deadlock to take place?

Actually, recovering a *coordination model* is a complex process. This complexity arises from the need to deal with multiple activities and multiple participants which in turn are influenced by multiple constrains, such as exceptions, interrupts and failures. On the other hand, the ever growing number of systems relying on non-trivial coordination logic for combining autonomous services, entails the need for methods and tools to support such a process.

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 adapts typical program analysis algorithms, namely slicing [22], to recover coordination information from legacy code. This is based on a notion of *coordination dependence graph* proposed here as a specialisation of standard program dependence graphs [5] used in classical program analysis. The recovered coordination patterns are automatically expressed in Orc, a general purpose orchestration language developed by J. Misra and W. Cook [15]. Orc scripts can be animated to simulate such patterns and study alternative coordination policies.

In an OSS development context, the method discussed in the paper can be used both

- to identify, extract and record the coordination layer of running OSS applications;
- to validate the coordination strategies implemented in a OSS program (i.e., a program to which access to code is granted) against a previous specification of its requirements.

The latter use is illustrated through a small, academic example in section 5.

Finally, it should be stressed that 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. In fact it can be implemented to target any language with basic communications and multi-threading capabilities. In the sequel this is illustrated with the $C^\sharp$ language in order to keep the presentation self-contained. However, a support tool [21] was developed, as a "proof-of-concept" for this technique, which analyses any piece of CIL (Common Intermediate Language) source code.

The paper is organised as follows. Section 2 provides the necessary background with a brief review of both slicing techniques and the Orc coordination language. Section 3.1 introduces a new graph based program representation suitable for representing concurrent object oriented programs and its tuning to what we call a *coordination dependence graph*. Then, in section 4, the generation of an Orc script from the system's graph representations is discussed in details. Section 5 shows the "method-in-action" through a small example in which a previous specification is checked against the coordination model recovered from the implementation. Finally, section 6 concludes and identifies a number of topics for future work.

2 Background

2.1 Program slicing and graph-based representations

Introduced by Weiser [25, 23, 24] in the late 1970s, program slicing is a family of techniques for isolating parts of a program which depend on or are depended upon a specific computational entity referred to as the *slicing criterion*. In Weiser's view, program slicing is an abstraction exercise that every programmer has gone through, aware of it or not, every time he undertakes source code analysis. Weiser's original definition has been since then re-worked and expanded several times, leading to the emergence of different methods for defining and computing program slices [3, 22].

Weiser's approach corresponds to what would now be classified as an *executable*, *backward*, *static* slicing method. A dual concept is that of *forward slicing* introduced by Horwitz et al. [7]. In forward slicing one is interested on what depends on or is affected by the entity selected as the slicing criterion. Note that combining the two methods also gives interesting results. In particular

the union of a backward to a forward slice for the same criterion n provides a sort of a selective window over the code highlighting the *region* relevant for entity n.

Another duality pops up between *static* and *dynamic* slicing. In the first case only static program information is used, while the second one also considers input values [9, 10] leading frequently, due to the extra information used, to smaller and easier to analyse slices, although with a restricted validity.

In this paper we are interested in a static, backward, inter-procedural slicing algorithm which can isolate the sub-program that potentially affects a particular statement.

Most slicing techniques are based on different Graph-based representations of the programs being sliced. In particular, the so-called System Dependence Graph (SDG)[7] stands as the base representation for most inter-procedural slicing techniques. The coordination analysis algorithm proposed in this paper is also based on SDG's.

2.2 Program coordination in Orc

Purpose and syntax.

Many traditional concurrency problems, like business workflows, resource sharing and composite web services built from service level agreements, can be regarded as orchestrations of third party resources. ORC [15] is a simple, yet powerful task orchestration language. Unlike other concurrency models, ORC regards the coordination of different activities and participants in a centralised manner. Thus, in ORC, external services never take the initiative of beginning communication, there is always a centralised entity that controls the calling of foreign operations.

The language builds upon few simple basic constructs that provide succinct and understandable coordination specifications of systems. In summary, the language provides a platform for simple specification of third-party resources invocations with a specific goal to accomplish, while managing concurrency, failure, time-outs, priorities and other constrains.

A number of formal semantics [8, 1] have been purposed for ORC which provide a solid theoretical background upon which one can base equivalence and program transformation.

In ORC, third party services are abstracted as sites which can receive calls from an ORC specification (orchestration). Included in this definition of sites are user interaction activities and third party data manipulation. An orchestration consists of a set of auxiliary definitions and a main goal expression upon which the specification evaluation begins.

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

Fig. 1. ORC syntax

The syntax of the language is presented in figure 1, where definitions for ORC *Expressions*, *Actual* parameters $\overline{p}$, *Formal* parameters $\overline{q}$ and *Definitions* are given.

An ORC expression can be composed by 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.

There are a few fundamental sites in ORC which are essential for effective programming of real world orchestrations. These fundamental site along with its informal semantics are presented in Table 1.

ORC also provides means for creating dynamic orchestrations, i.e., orchestrations that are able to create local sites at runtime. This feature is provided by special sites, called *Factory Sites*, which return a local site when invoked. Table 2 describes some useful factory sites, taken from [4], that will be required ahead for capturing some coordination schemas.

$let(x, y, ...)$	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

Site	**Operations**	**Description**
Buffer	*put*, *get*	The *Buffer* factory 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.
Lock	*acquire*, *release*	The *Lock* factory site returns a *lock* local site which provides two operations, *acquire* and *release*. When an expression invokes the *acquire* operation on a *lock*, that expression becomes its owner and subsequent calls to *acquire* from other expressions will block. Once the *lock* owner expression releases it, ownership of the lock will be given to one of the *acquire* operation waiting expression, if any.

Table 2. Factory sites

Informal semantics.

A site in Orc is an independent entity with the capacity of publishing values to the calling expressions. The evaluation of a site call holds indefinitely (possibly forever if 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 execution of expressions $f > x > g$ is executed by evaluating expression f, binding the value published by f to x and then evaluating expression g which may contain references to x. In case x isn't used by g, the sequential expression may be written as $f >> g$. Parallel composition of expressions is carried out by the concurrent execution of the intervening expressions. Finally, asymmetric parallel composition f **where** $x :\in g$ is evaluated by evaluating f and g in parallel and evaluation of f holds whenever it depends on variable x and g has not published any value. 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 [8, 1]. Table 3 presents a number of typical Orc definitions upon which the Orc generation process of section 4 builds.

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

Table 3. Some Orc definitions

The *XOR* definition takes as arguments a predicate expression p, and two orchestrations f and g. If p evaluates to *true* then orchestration f is executed, otherwise g is executed. Regard that, in spite of the parallel operator the definition only executes one of the expressions f and g and that one of them is always executed.

The $IfSignal$ definition 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, it always publishes a value, thus permitting the calling orchestration to proceed.

The *Loop* expression receives a predicate p and an orchestration f. This definition executes f continuously until predicate p evaluates to $false$. If predicate p evaluates to $false$ the definition doesn't block and returns a signal in order for the calling orchestration to proceed.

Definition *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, the signal is forwarded to the calling orchestration, but leaving the other parameter orchestration executing until it finishes.

3 Two graph representations for program analysis

3.1 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 the 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 [12, 13, 26]. 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. In the sequel a brief overview of the structure of a MSDG is made; the reader is, however, referred to [20] for a formal specification of a MSDG, as well as for a detailed description of the techniques used in its construction.

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 methods' 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 a MSta 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 example, 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 [13].

There are, of course, several types of dependencies represented as edges in a MSDG. A 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 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,

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

Typical dependencies between statement nodes are 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:

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

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

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

$$\langle v, \underline{\mathsf{mc}}, vis, v' \rangle \in \mathsf{Edge} \Leftrightarrow v \in \mathsf{MSta}(\mathsf{mcall}, -) \wedge v \in \mathsf{MSig}$$

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 in the corresponding actual parameters. Summing up, these adds the following assertions to the MSDG invariant:

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

Class inheritance and the fact the 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 [20] for further details).

3.2 The coordination dependence graph

The second stage in the discovery process is the construction of a coordination dependence graph (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` usually used to call web services:

```
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 statement matches the regular expression of a rule, it is labelled with the information in the rule's second component.

On completion of this labelling stage, the method proceeds to abstract 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 n such that $n \in \mathsf{MSta}$ with $\mathsf{SType}\, n = \mathsf{mcall}$) for which there is a control flow path (i.e., a chain of $\underline{\mathsf{cf}}$ dependency 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) 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 [7]. 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 edges are introduced to represent by direct control flow dependencies which were, previously to the removal operation, chains of such dependencies in the original MSDG. This ensures that future traversals of this graph are performed with the correct control order of statements.

4 Generating Orc scripts

The final phase in the method proposed in this paper is the generation of an Orc specification from the CDG previously built. This abstracts the entire behaviour of the system in a rigorous specification.

We believe that a coordination specification that is closer in structure to the original system is more understandable and, moreover, easier to confront with the original system. Therefore, in order to keep the original system's procedure calls nesting structure, an Orc definition is generated for each procedure in the CDG. The calling structure involving these procedures and recorded in the graph is also kept. Actually, it is this structure preservation goal that justifies the first exception in the pruning process mentioned in the previous section.

Note, however, the process does not generate an Orc definition for every procedure in the system, since during the construction of the CDG most procedures (more specifically, the ones not contributing to the coordination layer) were dropped. Also notice, it is quite simple to transform the nested Orc specification into a flat one, whenever this simplifies reasoning about the coordination specification.

The Orc generation process for a procedure is based on the program captured by the procedure sub-graph of the entire system CDG. The construction of the program represented by a CDG is quite straightforward and basically amounts to collecting the statements of the visited vertices by following the control flow edges.

To keep the presentation as succinct as possible, one formalises the Orc generation process over the subset[1] of $C^\sharp$ presented in figure 2. The representation of CDG instances in this language is a straightforward process, since of the constructs defined by the language are common to most popular language and the ones less so, like `LOCALCALL` and `ASYNCCALL`, are easily extracted from the vertices labelling information of the CDG.

The language is quite self explanatory. We consider that a local procedure call is as a synchronous call to a resource in the same machine not involving any communication primitive. Every asynchronous procedure call must be performed as if being made to an external resource, in which case it must specify the resource site uniquely (internal asynchronous procedure calls may be performed using the `ASYNCCALL` construct with `localhost` as resource site). The < > brackets used in the language definition stand for optional expressions.

As it happens in the complete $C^\sharp$ language, this subset also provides two possibilities for performing asynchronous calls. One 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. All the remaining

[1] Actually, we address all the relevant control flow, concurrency, and communication primitives of the language.

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

Fig. 2. Modified $C^\sharp$ language subset

details concerning the syntax and the semantics of the language are borrowed from the complete $C^\sharp$ language.

Although the input of the Orc generation process is the CDG of the program, for practical reasons one has opted to formalise the functions that specify this process over the subset of the $C^\sharp$ language.

The Orc generation is composed of two distinct 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 introduces a *Lock* site for each variable in a `LOCK` statement, while keeping track of all visited variables for avoiding site duplication. In the following definitions one uses product projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ as well as their dual co-product embeddings $\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$.

$$
\begin{array}{ll}
\psi \ (\texttt{LOCK } (x) \ \{st\}, \ V) & \equiv \begin{cases} (\iota_2(\mathit{Lock} > x\mathit{Lock} > \mathit{Signal}), \{x\} \cup V) & \text{if } x \notin V, \\ (\iota_1(), V) & \text{otherwise} \end{cases} \\
\psi \ (\texttt{ASYNCCALL } s \ f(\overline{x}) \ \{st\}, \ V) & \equiv (\psi_1 \ st, \ \psi_2 \ st \cup V) \\
\psi \ (\texttt{IF } p \texttt{ THEN } \{st\}, \ V) & \equiv (\psi_1 \ st, \ \psi_2 \ st \cup V) \\
\psi \ (\texttt{IF } p \texttt{ THEN } \{st_1\} \texttt{ ELSE } \{st_2\}, \ V) & \equiv \begin{cases} (\psi_1 \ st_1, \ \psi_2 \ st_1 \cup V) & \text{if } \psi_1 \ st_1 \neq \iota_1() \wedge \psi_1 \ st_2 = \iota_1(), \\ (\psi_1 \ st_2, \ \psi_2 \ st_2 \cup V) & \text{if } \psi_1 \ st_1 = \iota_1() \wedge \psi_1 \ st_2 \neq \iota_1(), \\ (\iota_2(\psi_1' \ st_1 \gg \psi_1' \ st_2), & \\ \ \psi_1 \ st_1 \cup \psi_1 \ st_2 \cup V) & \text{if } \psi_1 \ st_1 \neq \iota_1() \wedge \psi_1 \ st_2 \neq \iota_1(), \\ (\iota_1(), V) & \text{otherwise} \end{cases} \\
\psi \ (\texttt{WHILE } p \texttt{ DO } \{st\}, \ V) & \equiv (\psi_1 \ st, \ \psi_2 \ st \cup V) \\
\psi \ (st1 \ \texttt{;} \ st_2, \ V) & \equiv ((\iota_2(\psi_1' \ st_1 \gg \psi_1' \ st_2), \ \psi_1 \ st_1 \cup \psi_1 \ st_2 \cup V) \\
\psi \ (st, \ V) & \equiv (\iota_1(), \ V)
\end{array}
$$

where

$$
\begin{aligned}
\psi_1 &= \pi_1 \ . \ \psi \\
\psi_2 &= \pi_2 \ . \ \psi \\
\rho_2(\iota_2 \ x) &= x \\
\psi_1' &= \rho_2 \ . \ \pi_2 \ . \ \psi
\end{aligned}
$$

The second phase of this process is performed by function φ 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.

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

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 cases. Here, one must specify in ORC the behaviour of performing a request to a site without blocking for an answer and leaving the rest of the specification to carry on executing. This behaviour can be captured in ORC by using the previously presented *Discr* pattern and the fundamental site *Signal*. The *Discr* pattern 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 block upon a $false$ value over an if site. Thus one uses 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 pattern captures the intended behaviour. Similarly to the previous case 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:

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

5 An example

To illustrate the method introduced in this paper, consider the development of a client application to be part of a meteorological network. Instances of this application are to be installed in a number of geographically separated stations. Each station has at its disposal a set of sensors which provide some meteorological data relative to current weather conditions. The objective of the application to be developed is, among other functionalities, to communicate the data read from its sensors to a central server whose purpose is to predict the weather forecast for the next 5 days.

Since the production of weather forecasts is a demanding computational operation, the central server will be most of the time devoted to internal activity and only sporadically will interact with the client stations. Therefore, such communication is required to be asynchronous, to free the station application for other tasks while not interacting with the server. Another requisite of the application to be developed is that since client stations are aware of current weather conditions, they must compare the generated forecast with the verified weather conditions and, if great discrepancies are found, ask the central server to check and correct its forecast.

Although this coordination scenario is not unfeasible to be implemented directly, it has still enough details to justify the previous development of a specification of the communication protocol. Such specification, written in ORC is presented in Fig. 3.

$$
\begin{aligned}
Station()) &\triangleq Server.CalculateForecast() > fid > \\
&\quad GetResult(fid) \\
GetResult(fid) &\triangleq GetWeatherConditions() > x > \\
&\quad Server.GetForecast(x) > fc > \\
&\quad XOR(let(fc == null) \\
&\qquad , \\
&\qquad RTimer(1000) \gg \\
&\qquad GetResult(fid) \\
&\qquad , \\
&\qquad VerifyResult(fc) \\
VerifyResult(res)) &\triangleq XOR(\neg\ ConfirmForecast(res) \\
&\qquad , \\
&\qquad Server.VerifyForecast(res) > vfcid > \\
&\qquad GetVerification(vfcid)) \\
&\qquad , \\
&\qquad let(res)) \\
GetVerification(vfid)) &\triangleq Server.GetVerifiedForecast(vfid) > vf > \\
&\quad XOR(vf == null \\
&\qquad , \\
&\qquad RTimer(1000) \gg \\
&\qquad GetVerification(vfid) \\
&\qquad , \\
&\qquad let(vf))
\end{aligned}
$$

Fig. 3. The ORC specification

Note that in this specification *Server* is used as the central weather forecast sever. Operation *GetWeatherConditions* is intentionally undefined (its purpose is to gather local meteorological data). Finally, *ConfirmForecast* denotes another undefined internal operation intended to deal with the verification of the generated weather forecast with respect to the current weather conditions.

The next step in the development of the station application is to implement the above specification in a programming language. Suppose this task is given to a programmers team which produce the following $C\sharp$ 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 = GetResult(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 opId = cs.CalculateForecast(op);
17         return opId;
18     }
19
20      private Result GetResult(WeatherServer cs, int opId) {
21         Result res = null;
22
23         while(res == null) {
24            Console.WriteLine("Querying server for forecast.");
25            res = cs.GetForecast(opId);
26            Thread.Sleep(1000);
27         }
28         // Check if the result still needs further calculation
29         if(!ConfirmForecast(res)) {
30              Console.WriteLine("Querying server to confirm forecast.");
31              Operation op2 = ...confirm forecast parameter construction...
32              int op2Id = cs.VerifyForecast(op2);
33              res = GetVerification(cs, op2Id);
34         }
35         return res;
36      }
37
38      private Result GetVerification(WeatherServer cs, int opId) {
39         Console.WriteLine("Querying server for verification result.");
40         Result res = cs.GetVerifiedForecast(opId);
41         if(res == null) {
42            Thread.Sleep(2000);
43            return GetVerification(cs, opId);
44         } else {
45            return res;
46         }
47      }
48  }
```

This is the point where our method may come to scene: a new Orc specification can be extracted from the source code and compared with the original one. Fig. 4 shows the generated MSDG. The corresponding CDG, obtained through application of rules

$$\begin{array}{c}(\text{“}\texttt{CalculateForecast(*);}\text{”}, (WebService, Sync, Consumer))\\ (\text{“}\texttt{GetForecast(*);}\text{”}, (WebService, Sync, Consumer))\\ (\text{“}\texttt{VerifyForecast(*);}\text{”}, (WebService, Sync, Consumer))\\ (\text{“}\texttt{GetVerifiedForecast(*);}\text{”}, (WebService, Sync, Consumer))\end{array}$$

is represented by the same graph once all dashed vertices have been removed.

From this CDG a new Orc specification is derived resorting to the Orc generation strategy presented in section 4. The result is shown in Fig. 5.

Apart some minor differences concerning a few internal names, it is easy to conclude that both specifications (Fig. 3 and 5) represent the same behaviour in what respects to the invocation of the foreign services ($CalculateForecast$, $GetForecast$, $VerifyForecast$, and $GetVerifiedForecast$). This conclusion, which is quite trivial for this example, may, in practice require a bit of Orc rewriting to eventually transform both designs into a canonical form, therefore showing (or refuting) their (observational) equivalence.

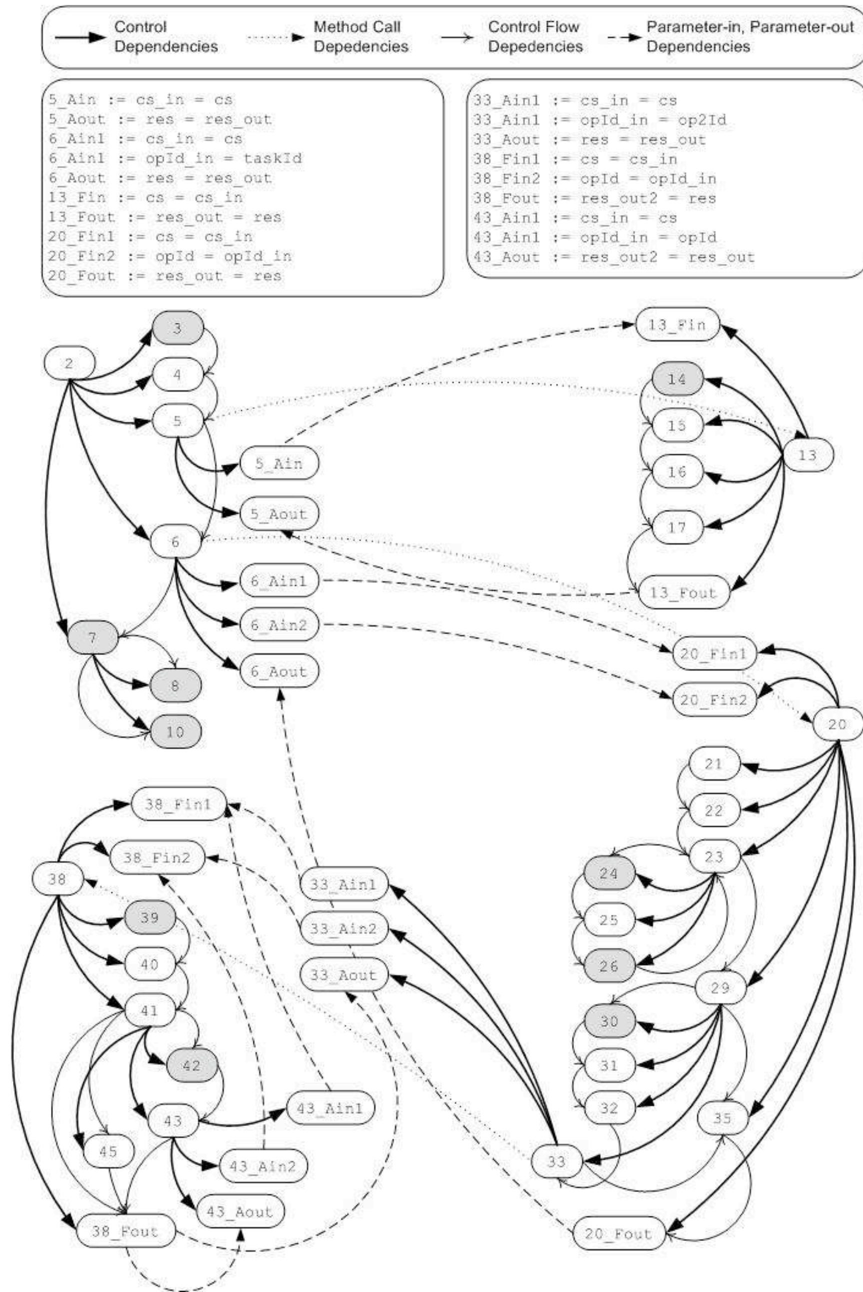

Fig. 4. Example code MSDG

6 Conclusion

This paper introduced a method that combines a number of program analysis techniques (namely, *dependence graphs*, *program slicing*, and *graph pattern analysis*) to extract the coordination layer of a system from its source code. The whole process is parametric on the type of coordination it abstracts. This feature enables the process, when instantiated with the suitable rules, to extract, for instance, the web service coordination structure of a system or its distributed object calling model, or even its multithread coordination layer. Further, it is possible to analyse more than one of these

$$
\begin{aligned}
GetWeatherForecast() &\triangleq new\ WeatherServer() > cs > \\
&\quad RequestServerTask(cs) > taskId > \\
&\quad GetResult(cs, taskId) \\
\text{RequestServerTask(cs)} &\triangleq GetWeatherConditions() > op > \\
&\quad cs.GetForecast(op) > opId \\
&\quad let(opId) \\
\text{GetResult(cs, opId)} &\triangleq Null() > res > \\
&\quad Loop(let(res == null), \\
&\qquad cs.GetForecast(opId) > res > \\
&\qquad RTimer(1000)) \gg \\
&\quad IfSignal(let(\neg\ ConfirmForecast(res)) \\
&\qquad\quad , \\
&\qquad\quad cs.VerifyForecast(op2) > op2id > \\
&\qquad\quad GetVerification(cs, op2id) > res > \\
&\qquad\quad Signal) \gg \\
&\quad let(res) \\
\text{GetVerification(cs, opId)} &\triangleq cs.GetVerifiedForecast(opId) > res > \\
&\quad XOR(let(res == null) \\
&\qquad , \\
&\qquad RTimer(2000) \gg \\
&\qquad GetVerification(cs, opId) \\
&\qquad , \\
&\qquad res)
\end{aligned}
$$

Fig. 5. Orc script extracted from the example code

types of coordination layers, given that the appropriate parametrisation of the communication primitives is taken into consideration during to the labelling phase.

Technically our contribution has been to extend previous work (namely [11, 16]) on program graph representation and devise a strategy for the identification and extraction of coordination information from applications. Related work include [19, 18, 17]. We believe this research is relevant for the analysis and (formal) certification of OSS, a topic that has been attracting some interest recently.

In this paper (as well as in its accompanying tool), Orc is used as the specification language for the abstracted coordination layers. Note that the Orc generation phase in this method is quite straightforward, resorting to a small set of Orc behavioural patterns. This can sometimes lead to big and repetitive specifications that demand further simplification to easier understanding of some of the aspects of the specified coordination. The whole method can, however, be adapted to other specification languages like CSP [6] or CCS [14]. Moreover, given the stateless behaviour of Orc site calls, the possibility of resorting to exogenous coordination models like Reo [2] arises as an interesting topic of future work.

Although Orc provides a powerful calculational framework, it would be interesting to tune the Orc generation process in order to look for more and more complex coordination patterns from the outset. Such search, should be based not only on the coordination scripts extracted from the Coordination Dependence Graph (CDG), but also on the entire CDG itself, as well as on the original Managed System Dependence Graph (MSDG) which captures other information that may be relevant to the discovery of more complex patterns. Such is exactly the topic of our current research.

References

1. 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.

2. F. Arbab. Reo: a channel–based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.
3. D. Binkley and M. Harman. A survey of empirical results on program slicing.
4. 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.
5. 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.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.
7. 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.
8. D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.
9. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
10. B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.
11. J. Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, 2003.
12. 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.
13. 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.
14. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.
15. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006.
16. 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.
17. 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.
18. 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.
19. 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.
20. N. F. Rodrigues. *Generic software slicing applied to architectural analysis of legacy systems*. PhD thesis, Dep. Informática, Universidade do Minho, 2008. (forthcoming PhD thesis).
21. N. F. Rodrigues and L. S. Barbosa. Coordinspector: a tool for extracting coordination data from legacy code. In *SCAM '08: Proc. of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2008. to appear.
22. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
23. M. Weiser. *Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Methods*. PhD thesis, University of Michigan, An Arbor, 1979.
24. M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
25. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
26. 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.