On The Discovery of Business Processes Orchestration Patterns

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

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

Abstract—COORDINSPECTOR is a Software Tool aiming at extracting the coordination layer of a software system. Such a reverse engineering process provides a clear view of the actually invoked services as well as the logic behind such invocations. The analysis process is based on program slicing techniques and the generation of, System Dependence Graphs and Coordination Dependence Graphs. The tool analyzes Common Intermediate Language (CIL), the native language of the Microsoft .Net Framework, thus making suitable for processing systems developed in any .Net Framework compilable language. COORDINSPECTOR generates graphical representations of the coordination layer together with business process orchestrations specified in WS-BPEL 2.0.

I. Introduction

Our dependency on software systems' allied to their exponential growth, both in size and complexity, is pushing the adoption of service oriented architectures. One of the main reasons for such a scenario to take place is that it has become impossible to manually coordinate the growing amount of software systems one as to deal with in order to accomplish a single task.

Take for instance the example of purchasing an airplane ticket and booking a hotel from a travel agency. In a pre service oriented environment, the travel agency attendant would have to consult every airline company with flights between the requested locations, and then she would have to perform a similar iterative task to book an hotel. Even more, she would have to pay attention to many particular details like special fares and promotions involving both hotels and flights or deal with flights with intermediate connections from a unique or several airline companies. To cope with the orchestration of these kind of different systems, companies start developing computational services that can be invoked by third party applications, enabling the previously described task to be automated by an application invoking a series of external services.

Software development industry soon understood the potential of software as a service, and began to implement service oriented solutions everywhere, sometimes even taking wrong approaches by transforming everything into a service. Despite the flavours and approaches taken to implement service oriented systems, the reality is that the world is becoming crowded with computational services ready to fulfill almost every software system needs on demand, and the amount of available services is growing every day. Again, developers understood this higher availability of services and started to implement software systems from alloys of external services, as it would be the case of a possible implementation of the above mentioned travel agency attendant task.

Although it may sound trivial at the beginning, to implement and maintain a service oriented system, there are some practical details which can make it a quite complex task, specially if one is demanding for a rigorous and flexible solution. So the idea that a real service oriented system is just a series of instructions invoking services which perform all the complex work, is usually one that will lead to useless systems.

Problems arise when such systems have to deal with multiple service-based activities and multiple participants at the same time, which in turn are influenced by multiple and different constrains which may also be enforced by other services. Even more, professional service oriented systems often have to work in multithreaded environments, because users have to be informed while the system is performing some time consuming task, or because the latency introduced by relying on external services instead of local components requires the developer to perform asynchronous calls to external services and proceed execution only, and if, the service returns an answer.

So, correct, responsive and high-available service-oriented systems have to be highly multithreaded and filled of specific details able to orchestrate the myriad of external services the system may depend upon. In order to develop such kind of systems, developers must be supported by new programming analysis tools to help in solving the specific problems that may arise. In particular it would be of outmost interest to have analysis tools capable of extracting the external services coordination schema of a system and to represent it in suitable visual ways to the developer. Such a model, exposing services calls and the programming logic that directly (or indirectly) influences (or is influenced by) such calls, would much facilitate the evolution of legacy systems to the service oriented paradigm, the development of new service oriented systems and also the understandability and maintenance of such a kind of solutions.

Even more, this tool should be able to capture multithreaded information and to confront it with the services calling model.

Such a tool would then be able to assist the developer in answering questions like: What services are actually being invoked in the implementation of a particular functionality? How are these services being combined to achieve the desired functionality? If one of these services fails, how does the system behaves? What is the logic, in terms of internal and external services invocations, behind the system provision of services?

COORDINSPECTOR is a software analysis tool developed to address such problems. In particular to extract a coordination model from source code, it resorts to a number of techniques to analyze and transform a system's implementation in order to expose its service coordination logic. In particular our method is based on dependence graphs, program slicing [8], graph and programming languages transformation techniques.

The tool analyzes Common Intermediate Language (CIL), the language interpreted by the .Net Framework for which every .Net language compiles to. By targeting the CIL, COORDINSPECTOR is able to analyze heterogenous systems implemented with multiple languages within the set of .Net Framework languages, which by now counts more than 40 languages. We believe that such is a very important feature for an analysis tool of this kind, since it is known that most mid to large size systems are implemented with several programming languages.

The remainder of the paper is organized as follows. Section II and IV introduces program representations structures as well as the specifications of algorithms for coordination analysis in COORDINSPECTOR. Further implementation details are discussed in section V. Finally we present an example of usage of the tool in action followed by some conclusions and topics for future work.

a) Contributions.: We present a generic reverse engineering technique for abstracting the service orchestration layer of running systems. This technique is based on two novel program representations: the MSDG and the CDG, which are described in section II and III, respectively. As a proof-of-concept of the ideas introduced, we present what we believe to be the first software tool aiming at the disentanglement and identification of service orchestration models from source code.

II. THE MSDG

Our approach to abstract a system's service architecture is based on an extended version of the System Dependence Graph (SDG) [2], which we call the Managed System Dependence Graph (MSDG). The MSDG includes many features from other object oriented SDG's extensions (namely [5], [6], [9]) and introduces new representations for concurrent constructs and specific managed code[1] details.

We assume a basic abstraction of CIL stack based control instructions to equivalent higher-order control expressions like IF THEN ELSE and WHILE[2] clauses.

[1]Code that executes under the management of the Common Language Runtime virtual machine

[2]There are a number of CIL analysis tools that provide this functionality.

A MSDG is actually the combination of 4 other graphs, namely the *Method Dependence Graph*, the *Class Dependence Graph* the *Interface Dependence Graph* and the *Namespace Dependence Graph*. In each of these graphs, vertices represent program statements or specific programmatic entities like methods, classes, interfaces or namespaces. We proceed by giving a brief description of each of these graphs that constitute the MSDG.

A. Method Dependence Graph

Most program graph representations, like SDG's and MDG's, are calculated from a graph representation of the flow of control in the program, often referred to as Control Flow Graph (CFG) of the program.

The Method Dependence Graph (MDG) is a multigraph resorting to different kinds of edges and vertices to represent a single method in a program. The vertices of a MDG are composed by the vertices representing the method statements and a special vertex called the method entry vertex which contains the signature of the method. The edges of a MDG represent control, data, parameter-in and parameter-out dependencies.

There are several kinds of data dependencies [7] that can be represented in a MDG. However for the purpose of our work one is only interested in data flow dependencies which we shall call just data dependencies.

The second kind of dependencies that a MDG captures are control dependencies, which represent the dependencies between control predicate statements (like IF THEN ELSE or WHILE statements) and assignment or method call statements. Edges representing control dependencies can be labeled with boolean values, indicating the flow of control upon the result value of a boolean expression eventually present in the source vertex. A MDG also uses control edges to connect the method entry vertex to each of the vertices representing method statements.

Although MDG's target the representation of single methods, there are extensions to cope with method calls. Actually such extensions are closely related to some details of single method MDG's. Thus, in a MDG, method calls and their parameters are recorded through the introduction of temporary auxiliary variables that mediate value passing between calling and called procedures. The definition of such temporary variables is captured in new sorts of vertices, called *actual-in* (vertices 2 and 3 in figure 1) and *actual-out* (vertex 4) vertices (for the calling methods), and *formal-in* (vertices 6 and 7) and *formal-out* (vertex 9) vertices (for the called methods).

Because one is targeting CIL, a stack based object oriented language, one has to capture the possibility that a method, may not only modify its parameter variables, but also some class or instance variables. To cover such situations, a MDG introduces formal vertices for each of these class or instance variables that are modified within the method, and, from the calling function side, it introduces the corresponding actual vertices. In what respects to edges, a method call is represented by *method call* edges between vertices containing method calls

and the method entry vertex of the called method, *parameter-in* edges between *actual-in* and *formal-in* vertices, *parameter-out* edges between *formal-out* and *actual-out* vertices. All formal vertices are connected to the method entry vertex and all actual vertices are connected to the calling vertex via control edges.
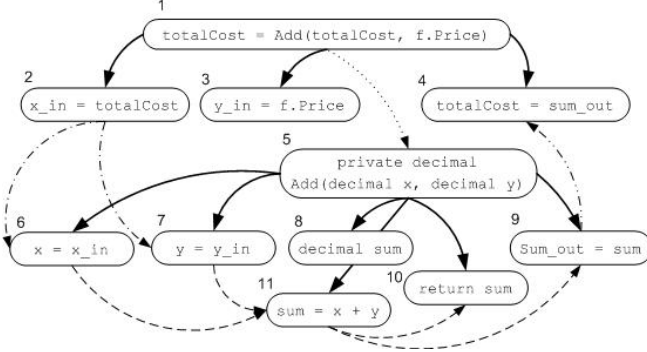

Fig. 1.   Method Dependence Graph

In order to cope with concurrency, whenever there is a spawning thread or process (a *fork* in operating systems terminology) we introduce a new triangular vertex in the graph, with one incoming control flow edge, from the vertex that fired the fork, and two outgoing edges, one for the newly created control flow (*asynchronous flow edge*) and another representing the continuation of the initial control flow.

$$
\begin{array}{lll}
z & \in & \textit{Values} \\
x & \in & \textit{Variables} \\
s & \in & \textit{Sites} \\
e & \in & \textit{Expressions} \\
st & \in & \textit{Statements} \quad ::= \quad z \\
& & \qquad\qquad\quad\; \mid \quad x \\
& & \qquad\qquad\quad\; \mid \quad x \; = \; e \\
& & \qquad\qquad\quad\; \mid \quad st_1 \; ; \; st_2 \\
& & \qquad\qquad\quad\; \mid \quad \texttt{LOCK} \; \{st\} \\
& & \qquad\qquad\quad\; \mid \quad \texttt{LOCALCALL} \; f\,(\overline{x}) \\
& & \qquad\qquad\quad\; \mid \quad \texttt{SYNCCALL} \; s \; f\,(\overline{x}) \\
& & \qquad\qquad\quad\; \mid \quad \texttt{ASYNCCALL} \; s \; f\,(\overline{x}) \; \prec \{st\} \succ \\
& & \qquad\qquad\quad\; \mid \quad \texttt{IF} \; p \; \texttt{THEN} \; \{st_1\} \; \prec \texttt{ELSE} \; \{st_2\} \succ \\
& & \qquad\qquad\quad\; \mid \quad \texttt{WHILE} \; p \; \texttt{DO} \; \{st\} \\
f & \in & \textit{Procedures} \quad ::= \quad \prec \texttt{CM, CT} \succ \; f\,(\overline{x})\,\{st\} \\
c & \in & \textit{Classes} \quad\;\; ::= \quad c\,\{x_1 = e_1 \; ... \; x_n = e_n \; f_1 \; f_2 \; ... \; f_n\}
\end{array}
$$

Fig. 2.   Statement Language

$$
\begin{array}{l}
\Phi\,(c\,\{x_1 = e_1 \; ... \; x_n = e_n \; f_1 \; ... \; f_n\}) \equiv \\
\quad \Psi\,([x_1 = e_1, ..., x_n = e_n] + \pi_1\,a + \pi_1\,b) \\
\quad \Upsilon\,(\pi_2\,b) \\
\quad \texttt{<flow>} \; \pi_2 a \; \texttt{</flow>} \\
\quad (\pi_3\,b) \\
\quad \text{Where} \\
\quad a = foldr \; g \; ([], []) \; (map \; \Phi_h \; [f_1, ..., f_n]) \\
\quad g\,(u, v)\,(t, k) = (u : t, v : k) \qquad b = map \; \Phi_b \; [f_1, ..., f_n]
\end{array}
$$

Fig. 3.   BPEL Generation

B. Class and Interface Dependence Graphs

The Class Dependence Graph (ClDG) serves to represent a class and aggregate all its members i.e., variables and methods. A ClDG contains a *class entry vertex*, which contains

$$
\begin{array}{l}
\Phi_h \; (\texttt{CM} \; f\,(\overline{x})\,\{st\}) \equiv \\
\quad (\Psi(\overline{x}), \; \texttt{<receive partnerLink=\#\#opaque>} \\
\qquad \texttt{operation=} f \; \texttt{variable} = (f+ \; \texttt{Request} \;) \; \texttt{>} \\
\qquad \texttt{<sources><source name=} f \; \texttt{/></sources>)} \\
\Phi_h \; \_ \equiv \bot
\end{array}
$$

Fig. 4.   Function Header BPEL Generation

the name of the class. There are *class membership edges* between the class entry vertex and the vertices representing the class variables and the method entry vertex of each of the class methods. Class membership edges are labeled with the visibility modifier (private, public, static, etc) of the target vertex [4].

Inheritance between classes is represented by *class inheritance edges* between the class entry vertices of the *involved classes*.

Abstract classes and Interfaces are represented like normal classes, except for methods which do not provide an implementation. The latter are solely represented by a method entry vertex.

C. Objects and Polymorphism

As in [6], we represent references to objects individually i.e., each reference to an object in a statement is represented by a tree depicting all the object variables. A difference in our representation of objects from the approach taken in [6] concerns the representation of recursive defined classes. Instead of using a k-limiting solution (only expanding the object tree to a level k) we use a special vertex called *fixed point vertex* defining recursive references in classes.

For dynamically typed references to objects, we build the object trees for every possible object type the reference may hold. Each of these trees root vertices are then connected to the corresponding object reference vertex.

D. Namespace Dependence Graph

The Namespace Dependence Graph (NDG) serves to represent the namespace division of classes and interfaces in a system. For these we follow a similar approach taken to represent Package Dependence Graphs in [4] and [9]. A namespace is represent in NDG by a *namespace entry vertex*, which contains *namespace membership edges* targeting every class or interface (entry vertex) declared under the defining namespace.

III. THE CDG

The building blocks of any service oriented system are the primitive communication calls that such systems use to invoke foreign services. It is based on these communication primitives, together with specific internal logic, that systems are able to construct elaborated orchestrations of foreign resources to deliver new functionalities to users in easy-to-use applications or services.

But what are such communication primitives? In theory they can be any mechanism by which an application may access a foreign resource. In practice they are instances of different technologies, namely web services, CORBA, RMI and .Net

$\Phi_b\ v\ l\ \mathbf{z} \equiv$
$\quad (v,\ l,\ \texttt{<literal>}\ z\ \texttt{</literal>}\ )$

$\Phi_b\ v\ l\ \mathbf{x} \equiv$
$\quad ((\Psi\ x) : v,\ l,\ \bot)$

$\Phi_b\ v\ l\ (\mathbf{x\ =\ e}) \equiv$
$\quad ((\Psi\ (x\ =e)) : v,\ l,\ \bot)$

$\Phi_b\ v\ l\ (st_1\ \texttt{;}\ st_2) \equiv$
$\quad ((\pi_1\ a) + (\pi_2\ b) + v,\ (\pi_2\ a) + (\pi_1\ b) + l,\ (\pi_3\ a) + (\pi_3\ b))$
$\quad$ Where
$\qquad a = \Phi_b\ v\ l\ st_1$
$\qquad b = \Phi_b\ v\ l\ st_2$

$\Phi_b\ v\ l\ (\texttt{LOCK}\ \{st\}) \equiv$
$\quad ((\pi_1\ a) + v,\ (\pi_2\ a) + l,$
$\qquad \texttt{<scope isolated=yes>}\ (\pi_3\ a)\ \texttt{</scope>}\ )$
$\quad$ Where
$\qquad a = \Phi_b\ v\ l\ st$

$\Phi_b\ v\ l\ (\texttt{LOCALCALL}\ f\ (\overline{x})) \equiv$
$\quad (v,\ l,\ \texttt{})$

$\Phi_b\ v\ l\ (\texttt{SYNCCALL}\ s\ f\ (\overline{x})) \equiv$
$\quad (\Psi(\overline{x}) + v,\ l,\ \texttt{})$

$\Phi_b\ v\ l\ (\texttt{ASYNCCALL}\ s\ f\ (\overline{x})) \equiv$
$\quad (\Psi(\overline{x}) + v,\ l,\ \texttt{<flow>}$
$\qquad \texttt{</flow>}$

$\Phi_b\ v\ l\ (\texttt{ASYNCCALL}\ s\ f\ (\overline{x})\ \{st\}) \equiv$
$\quad (\Psi(\overline{x}) + (\pi_1\ a) + v,\ (linkId : l) + (\pi_2\ a),$
$\quad \texttt{<flow>}$
$\quad \texttt{}$
$\qquad \texttt{<sources><source linkName =}\ linkId\ \texttt{/></sources>}$
$\quad \texttt{</flow>}$
$\quad \texttt{<scope name=}\ fCompleted\texttt{>}$
$\qquad \texttt{<targets><target linkName =}\ linkId\ \texttt{/></targets>}$
$\qquad \pi_3\ a$
$\quad \texttt{</scope>}\ )$
$\quad$ Where
$\quad linkId = f + getUToken()$
$\quad a = \Phi_b\ v\ l\ st$

$\Phi_b\ v\ l\ (\texttt{IF}\ p\ \texttt{THEN}\ \{st_1\}\ \prec \texttt{ELSE}\ \{st_2\}\ \succ) \equiv$
$\quad ((\pi_1\ a) + (\pi_1\ b) + v,\ (\pi_2\ a) + (\pi_2\ b) + l,$
$\qquad \texttt{<if><condition>}\ \beta(p)\ \texttt{</condition>}$
$\qquad \pi_3\ b$
$\qquad \prec \texttt{<else>}\pi_3\ b\texttt{</else>}\ \succ$
$\qquad \texttt{</if>}$
$\quad$ Where
$\qquad a = \Phi_b\ v\ l\ st_1$
$\qquad b = \Phi_b\ v\ l\ st_2$

$\Phi_b\ v\ l\ (\texttt{WHILE}\ p\ \texttt{DO}\ \{st\}) \equiv$
$\quad ((\pi_1\ a) + v,\ (\pi_2\ a) + l,$
$\qquad \texttt{<while><condition>}\ \beta(p)\ \texttt{</condition>}$
$\qquad \pi_3\ a$
$\qquad \texttt{</while>}$
$\quad$ Where
$\qquad a = \Phi_b\ v\ l\ st_1$

Fig. 5. Function Body BPEL Generation

Remoting, among others. The specific communication needs of a system will determine which of the remote procedure calls technologies to use. For instance if one is interested in maintaining the communication state between systems, usually a distributed object approach like CORBA or RMI is preferred. Otherwise, in cases where one is targeting a wide number of service consumers platforms, a web service approach is usually a better choice.

Since the coordination analysis of a system is based on tracing the use of communication primitives, and because the latter may change from system to system, our approach is parametric on the kind of communication primitives one is interested in. This parametrization is accomplished by a set of *rules* where a *rule* is a tuple composed of a regular expression and a set of attributes. The set of attributes serves to characterize what the regular expression identifies, in terms of communication primitive type (p.e. Web Service Call, COM, CORBA), calling mode (asynchronous or synchronous) and communication direction (communication provider or consumer).

Given a MSDG and a set of rules, the calculation of the CDG, with respect to the MSDG, starts with a vertex labeling process that processes the entire graph and checks for conformance of each statement in the vertices to the regular expressions in the set of rules. If a vertex respects a particular regular expression of a rule than it inherits the attributes of the rule. By the end of this labeling process, one obtains a graph whose vertices hold attributes which characterize the vertex in terms of communication primitive, if any, it represents (web service call, web service provider, COM call, etc) and the calling mode used (synchronous or asynchronous).

Given the MSDG with all the communication primitives and their calling modes identified, the second step is to abstract away the parts of the graph which do not take part in the coordination layer. The abstraction is accomplished by removing all the vertices that were not labeled except the ones in the following conditions:

1) procedure call vertices for which there is a control flow path to an annotated vertex,
2) vertices in the union of the backward slice of the program with respect to each annotated vertex.

The first exception above maintains the relevant procedure call nesting. This information will be useful to nest, in a similar way, the generated BPEL orchestration script, thus leading to more understandable coordination specifications.

The second exception covers all the statements in the program that may potentially affect the previously identified communication primitive vertices. In this set of vertices one finds predicate vertices[3] which are controlling and defining the parameters for execution of the communication primitives and, therefore, play a role in the coordination specification.

The slicing operation mentioned in the second exception is of outmost importance in terms of the overall abstraction process discussed in this paper. Even more, it is this program analysis technique that justifies the complex structure of the MSDG presented in section II and allows for the implementation of the slicing algorithm which follows.

We adopt a backward slicing algorithm very similar to the one presented in [2]. It consists of two phases. The first

[3]I.e., vertices containing predicate statements, like WHILE, IF THEN ELSE, FOR.

phase resorts to marking the visited vertices by traversing the MSDG backwards, starting on the vertex capturing the slicing criterion, and following control, method call, parameter in and data dependence edges. The second phase consists of traversing the graph backwards, starting on every vertex marked on phase 1 and following control, parameter out and data dependence edges. By the end of phase 2, the program represented by the set of all marked vertices constitute the slice with respect to the initial slicing criterion.

Except for control flow edges, every other edge from the original MSDG which contains a removed vertex as a source or target, is also removed from the final graph. Control flow edges containing a pruned vertex as a source or a sink are also dropped. On the other hand, new edges are introduced for representing direct control flow relations marking what were transitive control flow relations before the vertices removal. This ensures that future traversals of this graph, namely the one required for BPEL generation, are performed with the correct control order of statements.

Finally one ends up with a graph containing only the coordination relevant entities of the system. We call this structure, the *Coordination Dependence Graph*.

IV. ORCHESTRATION PATTERNS DISCOVERY

In contrast with the MSDG, which is usually a large and complex structure not suitable for direct human understanding, the CDG of a typical system is much smaller, since all logic details not concerned with coordination have bee removed. Even though, there are systems which contain great amounts of inter systems communications and originate large and complex CDGs. For these cases, one can take a step further and derive algorithms to automatize the search for orchestration patterns in CDG instances.

In this section we introduce an algorithm for representing in WS-BPEL the information captured by the CDG of a system.

It should be stressed that this algorithm is generic ("language agnostic"). To make things concrete, however, and the exposition easier to follow we present the BPEL generation algorithm over the simple statement language presented in Fig. 2. Note that this is not the language in which systems to be analyzed should be expressed, but rather the language used to represent CDG instances that facilitate the presentation of our orchestration discovery algorithm. In any case this could be eliminated by specifying the orchestration discovery algorithms directly over the CDG structure.

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 not so common, like LOCALCALL and ASYNCCALL, are easily extracted from the vertices labeling 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 $\prec \succ$ brackets used in the language definition stand for optional expressions and the functions prefix symbols CM and CT serve to identify a procedure as a communication primitive exposure (p.e. a method implementing a web service exposure logic) and a constructor respectively.

Like most programming languages with multi-threading capabilities, this language 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. In the later case the callback expression may reference a special variable result which holds the value returned by the call.

The LOCK statement behaves as expected i.e., it gives access of a specific statement execution in a single thread or process.

The generation of the abstract BPEL orchestration is accomplished by functions Φ, Φ_h, Φ_b presented in Fig. 3, Fig. 4 and Fig. 5 respectively. In order to being able to present here the BPEL generation algorithm, some details have to be dropped and a less verbose functional specification form was used to present the algorithm.

The specification of the BPEL generation borrows some Haskell [3] constructs, namely the list representation syntax (by the use of square brackets), the map function which applies a given functions to every element of a given list, the : function which appends an element to the head of a list and the $foldr$ function which applies a given binary function between all the elements of a list. Further more, we denote the first, second and third tuple projections by functions π_1, π_2 and π_3 respectively.

To avoid declaring every string concatenation used to generate the BPEL XML code, one has chosen to represent constant strings values in courier font. This way, whenever there is a functional expression followed or preceded by a sting constant in courier font, it should be interpreted as the concatenation of the value represented by the functional expression with the string constant. We also denote the empty string by $\perp$ and string concatenation operation by $+$.

Function Φ receives as input a *class* and returns the BPEL orchestration capturing all the services coordination contained inside the diverse class entities. This function depends upon four other auxiliary functions, namely Ψ which is responsible for converting a list of *Statement Language* variables to their equivalent BPEL forms, Υ which generates the BPEL links declarations that are used in the orchestration definition, Φ_h (presented in Fig. 4) that derives BPEL code specifying the provided services, and function Φ_b (presented in Fig. 5) responsible for calculating the BPEL logic defined inside each function body.

Note that the generated BPEL is in an abstract form as a consequence of using some ##opaque attribute values. Function Φ_h receives a list of *Statement Language* functions and for each function with attribute CM it computes a pair

containing a list of the variables found (which are converted to BPEL by function Φ using function Ψ) and a receive BPEL activity specifying the provision of a service that was preformed by some specific logic in the original system.

Function Φ_b receives a *Statement Language* function body and returns a tuple containing a list of variables to be initialized, a list of links to be initialized and the given functions body business logic translated to BPEL.

V. IMPLEMENTING COORDINSPECTOR

COORDINSPECTOR[4] is a software analysis tool developed as a proof-of-concept of the ideas presented in this paper.

The tool, a snapshot of which is presented in Fig. targets CIL code, the native language of the Microsoft .Net Framework, to which every .Net compilable language ultimately gets translated to before being executed by the framework. This decision to target CIL code was not an arbitrary one. Indeed we intended the tool to be able to cope with as many programming languages as possible, because most real world software systems are developed in more than one language. Moreover, given the potential of the tool to assist legacy systems evolution, the "language agnostic" feature became an important invariant. Thus, by choosing CIL, the tool is presently able to analyse more than 40 programming languages[5], and this number has only but potential to increase.
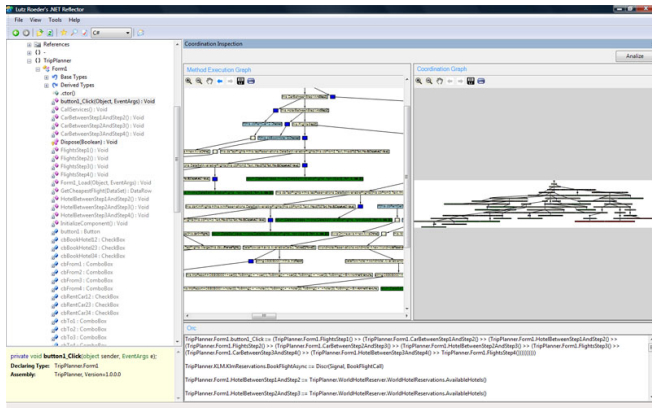

Fig. 6.   COORDINSPECTOR

In order to take advantage of existing CIL analysis tools, COORDINSPECTOR is developed as a plug-in for the CIL decompiler .Net Reflector[6]. The only, and important component COORDINSPECTOR takes from .Net Reflector is the parse for CIL code which delivers an object tree representation of the CIL concrete syntax tree.

Such 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 sub-graphs i.e., the MDG, ClDG, IDG and the NDG, as detailed in section II. Each

---

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

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

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

```
import System.Web.Services.Protocols.SoapHttpClientProtocol;
import System.Web.Services.Protocols;
public Class TimesheetSubmission {

    [WebMethod]
    public void SubmitTimesheet(TimeSheet t,
                                Consultant c, Client clt) {
        Decimal total = Invoke("GetTimesheetWithCost",
                            new object[] { c });
        if(total > 2000)
            this.InvokeAsync("AnalyzeSheet",
                            new object[] { t, c},
                            this.OnAnlyzeResponse, null);
        else {
            Invoke("CommunicateClientExpense",
                    new object[] { expense, total });
            Invoke("NotiffyApprovedExpense",
                    new object[] { expense, total });
        }
    }

    private void OnAnalyseResponse(object arg) {
        InvokeCompletedEventArgs invokeArgs =
            ((InvokeCompletedEventArgs)(arg));
        if (invokeArgs.Approved) {
            Invoke("CommunicateClientExpense",
                    new object[] { invokeArgs.Expense,
                                   invokeArgs.Total });
            Invoke("NotiffyApprovedExpense",
                    new object[] { invokeArgs.Expense,
                                   invokeArgs.Total });
        } else {
            Invoke("ResubmitSheet",
                    new object[] { invokeArgs.TimeSheet });
        }
    }
}
```

Fig. 7. $C^\sharp$ Example Program

component traverses the concrete syntax tree, using the object oriented proxy pattern, and collects the relevant information for the construction of a particular graph.

When applied to real world systems, and if executed sequentially, the MSDG calculation process can be a time consuming task because of the size and computational complexity involved. In order to cope with this situation one has improved the MSDG calculation performance by multithreading the tasks which build each MSDG sub-graph. This improvement reduced the MSDG calculation time to roughly on third of the original time.

The CDG calculation implemented by COORDINSPECTOR follows the approach presented in the previous section, thus starting by labeling the vertices based on rules identifying communication primitives. At the moment of writing, COORDINSPECTOR is only instantiated with rules identifying web services communications, distinguishing between synchronous and asynchronous calls as well as between invocation and provisioning of functionality using web services. Other sets of rules can, however, be easily added.

The graph pruning and slicing operations were once again implemented by following the specifications presented in the previous section and implemented by a series of graph traversal algorithms and transformation functions.

COORDINSPECTOR is also able to depict and navigate through both the calculated MSDG and CDG graphs, by resorting to the Microsoft Research GLEE graph library. The graphs provide different colors for the vertices, based on

```
<process>
 <variables>
  <variable name="SubmitTimeSheetRequest" />
  <variable name="GetTimesheetWithCostResponse" />
  <variable name="AnalyzeSheetResponse" />
 </variables>
 <flow>
  <receive partnerLink="##opaque"
           operation="SubmitTimesheet"
           variable="GetTimesheetWithCostResponse">
   <sources><source linkName="SubmitTimesheet" />
   </sources>
  </receive>
 </flow>
 <scope name="SubmitTimesheet">
  <targets><target linkName="SubmitTimesheet" />
  </targets>
  <sequence>
   <invoke partnerLink="##opaque"
           operation="GetTimesheetWithCost"
           input="SubmitTimeSheetRequest"
           output="GetTimesheetWithCostResponse" />
   <if>
    <condition>
     getVariableProperty(GetTotalCostResponse,
                         total) > 2000
    </condition>
    <flow>
     <invoke partnerLink="##opaque"
        operation="AnalyzeSheet"
        input="SubmitTimeSheetRequest"
        output="AnalyzeSheetResponse" >
        <sources>
         <source linkName="OnAnalyseResponse" />
        </sources>
     </invoke>
    </flow>
    <scope name="OnAnalyseResponse">
     <sequence>
      <targets>
       <target linkName="OnAnalyseResponse" />
      </targets>
      <if>
       <condition>
        getVariableProperty(AnalyzeSheetResponse,
                            Approved)
       </condition>
       <invoke partnerLink="##opaque"
           operation="CommunicateClientExpense"
           input="GetTimesheetWithCostResponse" />
       <invoke partnerLink="##opaque"
           operation="NotiffyApprovedExpense"
           input="GetTimesheetWithCostResponse" />
       <else>
        <invoke partnerLink="##opaque"
            operation="ResubmitSheet"
            input="GetTimesheetWithCostResponse" />
       </else>
      </if>
     <sequence>
    </scope>
    <else>
     <invoke partnerLink="##opaque"
         operation="CommunicateClientExpense"
         input="GetTotalCostResponse" />
     <invoke partnerLink="##opaque"
         operation="NotiffyApprovedExpense"
         input="GetTotalCostResponse" />
    </else>
   </if></sequence></scope></process>
```

Fig. 8.   Abstract BPEL of the Example Business Process

the labels the vertices hold, which facilitates direct manual reasonings over the graphs.

The graphical presentation of the graphs is also able to supply the user with specific vertex information, like labeling and the CIL code captured, by applying a double click on a particular vertex of the graphs.

Code generation in COORDINSPECTOR though based on function φ defined above, was not implemented as a syntax oriented operation. Instead, this functionality is implemented by using and extending the same graph traversal operations that were defined for the labeling process of the MSDG.

VI. EXPERIENCE

For a brief example of the technique presented, consider the following $C^\sharp$ code implementing a company's time sheet submission business process. The program provides a method (SubmitTimesheet) bound to a web service that is responsible for receiving consultants time sheets.

Once a time sheet arrives, the total cost of the time sheet is calculated by the foreign web service GetTimesheetWithCost according to the time sheet's consultant fees. If the total cost retrieved by GetTimesheetWithCost is above 2000 then the business proceeds by asynchronously invoking the AnalyzeSheet web service with callback function OnAnalyseResponse. On completion of function AnalyzeSheet, the business process proceeds by evaluating function OnAnalyseResponse which based on the time sheet cost approval, communicates the response to client and consultant in case of a positive approval, or requests the resubmission of the time sheet to the consultant in case of a negative response.

If the total cost of the time sheet is bellow or equal to 2000, the business process communicates the cost both to the consultant and client through invocation of the web services NotiffyApprovedExpense and CommunicateClientExpense.

Because of space limitations this example may seem like a trivial case upon which to apply the process presented, nevertheless we would like to point out out that if this same business process was not isolated (like presented) but mixed with other program statements for controlling user interface or other resources at diverse levels in the system, the same (or an equivalent) BPEL orchestration would have been lifted.

By applying the overall process presented to the program of Fig. 7, one would obtain the BPEL orchestration depicted in Fig. 8.

VII. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm and tool for abstracting the coordination layer of software legacy systems. Such layer is often spread among various parts of a system and, even more problematic, it is usually mixed with code devoted to implement this internal computations.

Although not being a complete abstraction process, in the sense that the generated WS-BPEL orchestration are abstract and need manual adaptation to become executable, the work presented here is a relevant step towards automatic business process discovery (if it will ever be possible to become completely automatic).

On of its main features is its parametrization by rules identifying the communication primitives one is interested in,

thus making it adaptable to diverse kinds of coordination analysis anf programming frameworks. Given the language heterogeneity that most real world systems present, the language agnosticism of the technique stands as another very important feature.

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

Overall, we regard this work as part of the broad area of software architecture analysis, where the ultimate goal is the discovery of the business process orchestration logic laying beneath a software system implementation. Techniques, like the one presented here, to assist the correct discovery of business processes (or even to perform it automatically), contribute to the evolution of such systems towards the (web) service oriented world.

As a proof-of-concept of the ideas presented in this paper we also introduced a tool, called COORDINSPECTOR, which applies the algorithm to perform the discovery process on systems targeting the Microsoft .Net framework.

An interesting topic for future work is the classification of orchestration patterns, as in [1], and their use in guiding the discovery and extraction process.

Another interesting improvement would be to allow changes to be made in the BPEL generated orchestrations and, based on such changes, regenerate equivalent transformations to be applied to the original source code. Translation of system's business process exception logic to their equivalent BPEL elements would also be interesting.

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] 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.

[3] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[4] G. Kovcs, F. Magyar, and T. Gyimthy. Static slicing of java programs.

[5] 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.

[6] 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.

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

[8] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[9] 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.