DOI: 10.2298/CSIS101228017O

Domain-Specific Language for Coordination Patterns

Nuno Oliveira[1], Nuno Rodrigues[1,2], and Pedro Rangel Henriques[1]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{nunooliveira,prh}@di.uminho.pt
[2] IPCA – Polytechnic Institute of Cavado and Ave
Campus do IPCA, Barcelos, Portugal
nfr@ipca.pt

Abstract. The integration and composition of software systems requires a good architectural design phase to speed up communications between (remote) components. However, during implementation phase, the code to coordinate such components often ends up mixed in the main business code. This leads to maintenance problems, raising the need for, on the one hand, separating the coordination code from the business code, and on the other hand, providing mechanisms for analysis and comprehension of the architectural decisions once made.
In this context our aim is at developing a domain-specific language, `CoordL`, to describe typical coordination patterns. From our point of view, coordination patterns are abstractions, in a graph form, over the composition of coordination statements from the system code. These patterns would allow us to identify, by means of pattern-based graph search strategies, the code responsible for the coordination of the several components in a system. The recovering and separation of the architectural decisions for a better comprehension of the software is the main purpose of this pattern language.

Keywords: coordination patterns, software architectures, domain-specific languages, CoordInspector.

1. Introduction

Software Architecture [17] is a discipline within Software Development [16] concerned with the design of a system. It embodies the definition of the structure and the organisation of components which will be part of the software system. The architecture design also concerns the way these components interact with each other as well as the constraints in their interactions. In their turn, software components [12] may be seen as objects in the object-oriented paradigm, however, besides data and behaviour, they may embody whatever one prefers as a software abstraction. Although they may have their own functionality (sometimes a component is a remote system), most of the times they are developed to be composed with other components within a software system and to be

reused from one system to another, giving birth to component-based software engineering methodology [14].

The definition of the interaction between the components of a system may be seen from two perspectives: (i) integration and (ii) coordination. The differences between these two perspectives is slightly none. The former is related with the integration of some functionalities of a system into a second one, which needs to borrow such a computation; the latter is concerned with the low level definition of the communication and its constraints between the components of a system. Such interaction definition between the components can be either *endogeneous* or *exogeneous*. In the latter, the coordination of components is made from the outside of a component, not needing to change its internals to make possible the communication with other components [4], the former is the dual methodology.

This *rule* of separating computational code from coordination code is not always adopted by software developers. The code is often weaved in a single layer where there is no space for separation of these kind of concerns. This behaviour could raise problems in the future of the software system, namely in maintenance phase. These problems are mainly concerned with the comprehension of both the code and the architectural decisions, which hampers their analysis.

Reverse engineering [28] of legacy systems for coordination layer recovery would play an important role on maintenance phases, diminishing the difficulties on analysing the architectural decisions. But, extracting code dedicated to the coordination of the system components from the entire intricate code is not an easy task. There is not a standard (nor unique) way of programming the interactions between the components. However, and fortunately, there is a great number of code patterns, which the majority of the developers use to write coordination operations. Once the code of a system can be represented as a graph of dependencies between the statements and procedures, the so-called *System Dependence Graph* (`SDG`) [15], one is also able to represent code patterns as graphs, allowing the search for these patterns in the `SDG`.

In this context, we define the notion of *coordination patterns* as follows:

> Given a *dependence graph* $\mathcal{G}$, as in [26], a coordination pattern is an equivalence class, a *shape* or a sub-graph of $\mathcal{G}$, corresponding to a trace of coordination policies left in the system code.

In this paper we show how we developed a *Domain-Specific Language* (`DSL`) [8, 21] named `CoordL`, to write coordination patterns. The main objective of this language is to translate `CoordL` specifications into a suitable graph representation. Such representation would feed a graph-based search algorithm, to be applied to a dependence graph, in order to find the coordination code weaved in the system code. In Section 2 we address related work; in Section 3 we present and describe the syntax of `CoordL`; in Section 4 we address its semantics; in Section 5 we show how we used the `AnTLR` system to define the syntax and the semantics of `CoordL`; in Section 6 we expatiate upon actual and future ap-

plications of the language and the patterns. Finally, Section 7 presents some conclusions about the presented research.

2. Related Work

`CoordL` is a `DSL` to write coordination patterns with the purpose of extracting and separating the coordination layer from the source code of a component-based software system. The main domain of this work is the reverse engineering (of legacy software systems architecture) and the idea of the code separation between concern-oriented layers aims to recover architectural decisions and ease the comprehension of the entire system and its architecture which is one of the most important parts within the maintenance phase of software engineering.

The recovery of the system architecture for software comprehension is not a novelty. Tools like Alborz [27] or Bauhaus [24] recover the blueprints of an object-oriented system. Bauhaus recovers architectures as a graph where the nodes may be types, routines, files or components of the system, and the edges model the relations between these nodes. Such architectural details are presented in different views for an easy understanding of the global architecture. Alborz presents the architecture as a graph of components and keeps a relation between this graph and the source code of the software system. Our main aim is not at visualising the blueprints of a system, but to provide mechanisms for understanding the rationale behind the architectural decisions once made. This embodies the recovering of the coordination code. Although visualization is really important for architectural analysis, the tools mentioned before, and even more recent and focused tools like [19] do not support the mentioned feature and do not take advantage of code patterns to do the job.

Although we may reference Architectural Patterns [6] or Design Patterns [9] as related work, because of the common methodology of patterns and the borrowed notions and description topics, there is a huge difference between their application. While coordination patterns are used to lead a reverse engineering to recover architectural decisions, and are focused on low-level compositions of code, the architectural/design patterns work at a higher level, being used to define the architecture of a system in earlier phases of the software development process [16]. Patterns and patterns finding are a very interesting matter on data mining discipline [5], and are being applied on several areas, with focus on social networks [18, 13]. Although the work reported in this paper is far from the area of data mining and social networks, the purpose of `CoordL` is to extract useful information based on patterns that are a result of previous knowledge on how coordination of components is made. The recovering algorithms that we may use may be based on those used to mine data on social networks since they also rely on graph-based search. The applications of the recovering strategies on data mining is specially concerned with optimization and adaptability to new contexts. By recovering the architectural decisions, also optimizations and adaptability (on a different perspective than that of data mining) may be also

a valuable application of the subject in this paper. In fact there is, somehow, a parallel between data mining and our work that we may follow, but the essence of both kind of works is very different. The same does not happen with process mining [7], which, in fact, is very close to our work. The search for business process workflows [1] makes heavy use of process mining techniques. Although process mining is typically based on event logs, a dependence graph may also uncover traces of a workflow. A workflow between components or modules is what, at the end, we want to obtain for the coordination code separation.

Architecture Description Language's (ADL), are languages used to formally describe a system components' and the interactions between them. Although CoordL is not to be considered an ADL, we must acknowledge that there are some similarities in the concepts embodied in these languages and those encapsulated in our. Some well-known examples of these languages are ACME [11], ArchJava [2], Wright [3], and Rapide [20]. The great majority of these languages has tool support for analysing the described architecture. Such analysis, made at high level, allows one to reason about the correctness of the system, and may provide important information about future improvements that can, or can not, be done according to the actual state of the architecture.

According to our knowledge, there is no other language with the same specific purpose as CoordL.

3. CoordL - Design and Syntax

The design of a DSL is always a task embodying some well defined steps. As a first step, one needs to collect all the information about the domain in which the language will actuate. Afterwards, this information must be properly organised using, for instance, ontologies [29]. Once the main concepts of the domain are identified, one needs to choose those that are really needed to be encapsulated in the syntax of the language; this leads to the last step which concerns the choice of a suitable syntax for the language.

Figure 1 presents an ontology to organise the domain knowledge of the area where we want to actuate. The main concept of this domain is the coordination pattern. The majority of the concepts incorporated in this domain description are wider than what we show, however, to keep the description limited to the domain, we narrowed the possible relations between each concept, as well as the examples they may have.

Note that in this ontology we use operational relations (marked as dashed arrows) besides the normal compositional ones. This provides a deeper comprehension of how the concepts interact between each other in the domain.

The core of the knowledge base represented in Figure 1, describes that a coordination pattern is a part of a *coordination dependence graph* (CDG) [26], abstracting code which is seen as a composition of statements concerned with coordination aspects, and are used to analyse architectures. As a novelty, the web of knowledge shows that coordination patterns communicate with each other through ports.

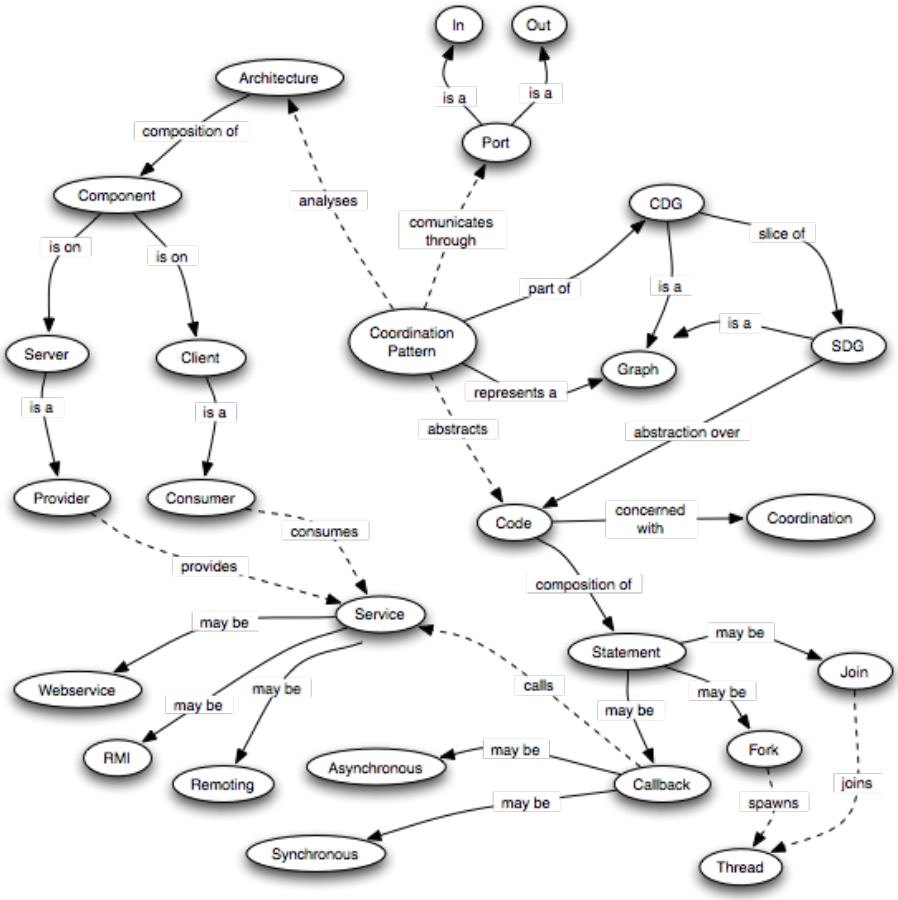

Fig. 1. Ontology Describing the Coordination Pattern Domain Knowledge

From this description, and knowing that the main objective of `CoordL` is to define a graph over the composition of statements in the source code of a system, one needs some kind of graph representation to be embodied in the language. An obvious reference for representing graphs in a textual form is the DOT language [10], so, `CoordL` borrows some aspects from that language. The notion of communication ports (in and out) came from the `ACME` language [11], although the notion of *ports* is very different in these two contexts. To know which ports exist in a pattern, the notion of *arguments* – taken from any *general-purpose programming language* (`GPL`) – was adopted . The description of what are these ports led to the introduction of declarations and initialisations in the language. Declarations describe the types of statements represented as nodes in the graph, while initialisations describes the service call which is performed by the node.

From this textual description we defined a syntax by means of a context free grammar (partially) shown in Listing 1.1.

Listing 1.1. Partial grammar for `CoordL`

```
lang → pattern+
pattern → ID '(' ports '|' ports ')' '{' decls root graph '}'
ports → lstID
decls → (decl ';')+
decl → 'node' lstID '=' nodeRules | 'fork' lstID | 'join' lstID |
  'ttrigger' lstID | ID instances
instances → instance (',' instance)*
instance → ID '(' ports '|' ports ')'
...
root → 'root' ID ';'
graph → aggregation | connections
aggregation → patt_ref ('+' patt_ref)*
patt_ref → cnode | '(' aggregationn ')' connection
...
cnode → node | ID '.' propTT
...
connection → '{' operations '}' '@' '[' ports_alive '|' ports_alive ']'
...
operation → cnode link cnode | fork | join | ttrigger
...
fork → node sp_link '{' cnode ',' cnode '}'
...
link → '−' ID '−' '>' | '−' '(' ID ',' ID ')' '−' '>'
...
```

Figure 2 presents two examples of patterns written with `CoordL`. Pattern a), known as the *Asynchronous Sequential Pattern*, is a pattern often used when the system has to invoke a series of services but the order of the answer is not important. Pattern b), known as the *Joined Asynchronous Sequential Pattern*, is a transformation of the first pattern to impose order in the responses.

Both of these patterns address different aspects of the syntax, but the main structure of the patterns is the same. Moreover, they address the composition and reuse of patterns.

Regard, for instance, the pattern in Figure 2.a). It has a unique identifier (`pattern_1`) and declares *in* and *out* ports, identified by p_0 and p_1, p_2 and p_3 respectively. The *in* ports go on the left side of the '|' (bar) symbol, and the *out* ports on the right. Then, a space is reserved for node declarations and

initialisations. There are 5 types of nodes in CoordL, namely *node*, *fork*, *join*, *ttrigger* and *pattern instance*. In Figure 2.a) we use the node and fork types, and in Figure 2.b) we use node, join and pattern instance types. The ttrigger type is similar to fork and join.

Nodes of type *node* require an initialisation, describing a list of rules addressing the corresponding coordination code fragment, the type of interaction and the calling discipline. These rules are composed using the $\&\&$ (and) and/or $||$ (or) logical operators, and the list must, at least, embody one of the following: (i) Statement (st), presents the code fragment of the statement responsible by the coordination request. This statement may be described by a regular expression or may be a complete sentence; (ii) Call Type (ct), defines the type of service requested. The options are not limited, but some of the most used are web services, RMI or .Net Remoting; (iii) Call Method (cm), defines the method in which the request is made. It can be either synchronous or asynchronous, and (iv) Call Role (cr), describes the role of the component that is requesting the service. It can be either consumer or producer.

```
1  pattern_1 (p0 | p1, p2, p3){
2    node p0, p3 { st == "*" }
3    node p1, p2 {
4      st == "calling(*)" &&
5      ct == webservice &&
6      cm == sync &&
7      cr == consumer
8    };
9    fork f1, f2;
10   root p0;
11
12   {f2 -(x,w)-> (p3, p2)}
13   {f1 -(x,y)-> (f2, p1)}
14   {p0 -x-> f1}
15 }
```

(a)

```
1  pattern_2 (p1 | p2){
2    node p1, p2, pa = {st == "*"};
3    pattern_1 patt(i1 | o1, o2, o3);
4    join j1, j2;
5    root p1;
6
7    (p1 + patt + p2)
8    {p1 -x-> patt(i1),
9    (patt(o1), patt(o3)) -(x,y)-> j1}
10   {( j1, patt(o2)) -(x,w)-> j2}
11   {j2 -x-> p2}
12 }
```

(b)

Fig. 2. Definition of Two Coordination Patterns with CoordL

Pattern instance nodes have the type of an existent pattern. In Figure 2.b), line 3, it is declared an instance of pattern pattern_1. Each instance of a pattern must be initialised with unique identifiers referring to all the in and out ports of the pattern typing it.

In CoordL, we define patterns by giving them a name, defining the ports and their body. However, nodes and other anonymous structures used (within the body) to define a pattern are also seen as patterns (or *pseudo-patterns* for disambiguation purposes). The main operations over patterns (including *pseudo-patterns*) are the aggregation and the connection. Aggregation[3] is the

[3] Aggregation may be used alone in a pattern body definition, but will never define a usable pattern.

combination of two or more patterns by putting them *side-by-side*, this is, not making any connection between their ports. The syntax for the aggregation operation is presented at line 7 of Figure 2.b). Connection is the combination of two nodes by means of an edge with the identification of, at least, a running thread. Examples may be seen in lines 12, 13 and 14, of `pattern_1` and 8, 9, 10 and 11 of `pattern_2`.

These two operations are used to build the pattern graph, which comes after all node declarations and identification of the root node[4]. There are two ways of defining the graph: (i) the implicit composition, where there are only connection operations and (ii) the explicit composition, where aggregation and connection operations are used simultaneously. The graph of `pattern_1` uses implicit composition, while `pattern_2` uses explicit composition.

The connection operation uses one or more `out` nodes and one or more `in` nodes (depending on the type of `in` and `out` nodes). When the connection uses these nodes, their implicit `in` or `out` ports are closed, meaning that no newer connection can use these nodes as `in` or `out` ports again. Sometimes one needs to reuse a node as an `in` or an `out` port of a connection. This leads to the re-opening of a port to be used in the sequent connections. In order to facilitate this, we introduce the '@' (alive) operator.

We acknowledge that with all the operators and the associated syntax, the pattern code is not easily readable. This way, we define a visual notation with a suitable "translation" from the textual notation of `CoordL`. In Figure 3 we present the components of the visual notation, corresponding to the textual elements that define the graph.

In Figure 4 we present how the patterns in Figure 2 look like in this notation.

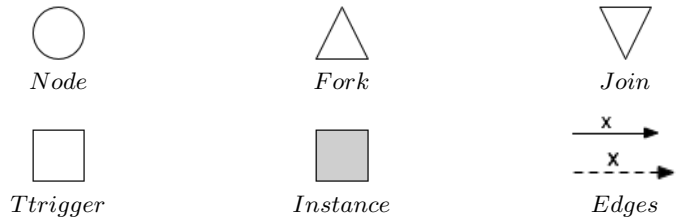

Fig. 3. Components of the Visual Notation for `CoordL`

[4] The root node identifies the start node of the pattern and is only useful for graph-based search of these patterns in a `CDG`. It must be one of the `in` ports of the pattern, chosen nondeterministically, by the pattern definer.

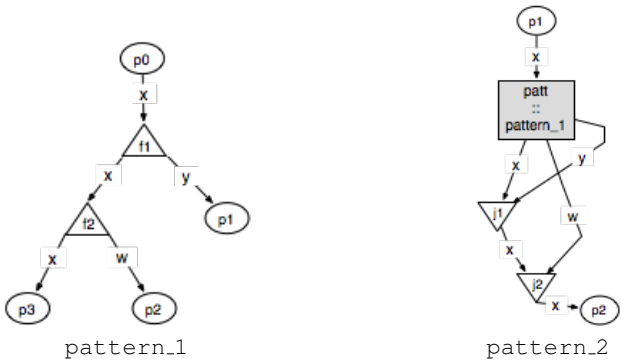

Fig. 4. Visual Representation of Two Coordination Patterns

4. CoordL - The Semantics

The constructs presented in Section 3 have a precise meaning in `CoordL`. In some cases it is possible to draw a mapping between the meaning of a construct and the dependence graph, which is extracted from the source code of the system being analysed. The following paragraphs provide an informal semantics of each construct in the language.

Bar: $|$

This construct separates a list of identifiers into two. The identifiers on the left side list are called `in` ports and those on the list at the right side are called `out` ports. It may appear in the *signature* of a pattern, or in the graph of a pattern, whenever it is needed to keep ports opened for further use.

Aggregation: $pp_1 + pp_2$

This construct sets two patterns side by side, but it doesn't connect them. This is used to reinforce the existence of the patterns in the graph, before connecting their ports. The aggregation operation (meaning collecting patterns, in our standpoint) is not required to build a graph, however, for completeness of the language and for a calculus of `CoordL` language (as envisaged as future work), this would be an important operation.

Connection: $n_1 -x-> n_2$

This construct creates a link between two nodes in the graph of the pattern. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there is a path of one or more edges going from n_1 to n_2 through one or more edges in a thread identified by x.

Fork Connection: $f -(x, y)-> (n_1, n_2)$

This construct creates a link between three nodes in the graph of the pattern, where the start node is a fork. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there are two parallel paths (p_1 and p_2) going from f to n_1 through one or more edges in a thread identified by x, and from f to n_2 in a freshly spawned thread identified by y, respectively. A necessary pre-condition

is that in the dependence graph, there is some path p_0 from any node to f in a thread identified by x.

Join Connection: (n_1, n_2) $-(x, y)->$ j

This construct creates a link between three nodes in the graph of the pattern, where the end node is a join. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there are two parallel paths (p_1 and p_2) going from n_1 to j through one or more edges in a thread identified by x, and from n_2 to j in a thread identified by y, respectively. A necessary pre-condition is that in the dependence graph, there are two paths (p_0 and p'_0) from a fork node to n_1 in a thread identified by x and from the same fork node to n_2 in a thread identified by y, respectively.

Thread Trigger Connection: (n_1, n_2) $-(x, y)->$ $tt.sync$, (n_1, n_2) $-(x, y)->$ $tt.fail$

This construct creates a link between three o nodes in the graph of the pattern, where the end node is a ttrigger. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there are two parallel paths (p_1 and p_2) going from n_1 to tt through one or more edges in a thread identified by x, and from n_2 to tt in a thread identified by y, respectively. This meaning aims at expressing what happens when the threads synchronise (tt.sync), or when the threads synchronisation fails (tt.fail). A necessary pre-condition is that in the dependence graph there are two paths (p_0 and p'_0) from a fork node to n_1 in a thread identified by x and from the same fork node to n_2 in a thread identified by y, respectively.

List of Connections: $\{$ $connection$, ... $\}$

This construct creates a list of independent connections. That is, a connection inside this list does not depend on any node, node property or even on other connections that are used and defined in the list. This independence resorts to the fact that there is no order between the connections inside a list of connections. Subsequent lists of connections may, but are not obliged to, depend on previous lists.

Along with this construct comes the notion of *fresh nodes*. A fresh node is a control node (like a fork, join or ttrigger) that is firstly used in a connection, and cannot be reused in the same list because of the dependence order. For instance, a fork node must be used as an out port in a connection before being used as a `in` node.

Alive: @

This construct instructs that a list of identifiers is kept alive as `in` and `out` ports. Ports need to be reopened because once a connection uses a node, the implicit port of such node is *killed*. The '@' construct is followed by a list of identifiers divided into two by the bar construct.

5. CoordL - Compiling & Transforming

We used `AnTLR` system [23] to produce an attribute-grammar-based parser for `CoordL`. Taking advantage of `AnTLR` features we adopted a *separation of concerns* method to generate the full-featured compiler. Figure 5 shows the architecture of the compiler system. The main piece of the compiler system is

the syntax module where we specify both the concrete and abstract syntax for `CoordL`, using the context free grammar presented in Listing 1.1. Based on the abstract syntax, `AnTLR` produces an intermediate structure of that grammar known as a tree-grammar.

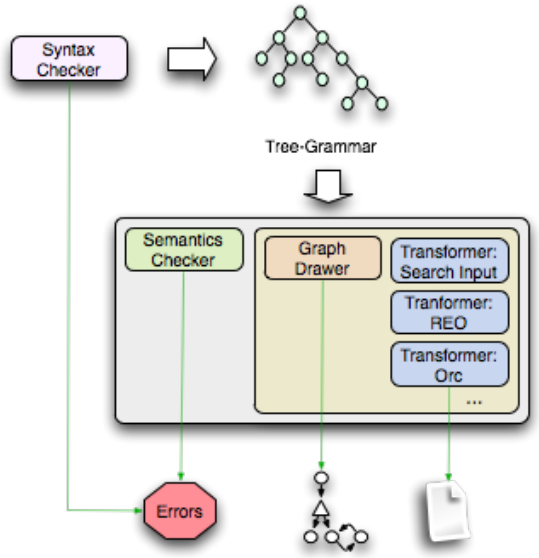

Fig. 5. `CoordL` Compiler Architecture

From the tree-grammar (using attribute grammars methodology) we were able to define new modules that do not care about the concrete syntax. These modules embody the semantics checker, the graph drawer and the unimaginable number of possible transformations applied to that tree-grammar.

The following hierarchical dependence on these modules is observed: the semantics module depends on the syntax module; the graph drawer and the transformation modules depend on the semantics module, so, by transitivity, they also depend on the syntax module. This holds the requirement that some modules may only be used if the syntax and the semantics of the `CoordL` sentence are correct.

We recognise that the separation of concerns in the modules and the dependence between them may be seen as a problem in maintaining the compiler. For instance, if something in the abstract syntax of the language changes, these changes must be performed in every dependent module. Nevertheless, this method also brings positive aspects: (i) the number of code lines in each file decreases, easing the comprehension of the module for maintenance; (ii) since each module defines an operation over the coordination patterns' code,

the compiler may be integrated in a software system providing independent features to manipulate the patterns and (iii) the separation of concerns into modules eases the maintenance of each feature.

The transformation modules have, as main objective, to provide perspectives about the coordination patterns, namely, their transformation into Orc [22] or REO [4] specifications. The transformation of the patterns in Orc is more or less simple since it may be used an algorithm similar to that presented in [26] adapted to `CoordL` (since it is originally adapted to the `CDG`). Concerning REO, adequacy of transforming these patterns into REO circuits cares for deep research, since the paradigms are, somehow, different.

An important module to be considered is the transformation of the pattern code into a suitable input to search for these patterns in the dependence graph of a system's code. As for the syntax and the semantic modules, their main output is the syntactic and semantic errors, respectively. The graph drawer module outputs the visual representation of the coordination patterns.

The transformation of the patterns in their visual notation is the most direct and easy transformation from the mentioned ones. Technically, it was defined a new module using the tree-grammar for `CoordL` that is created by `AnTLR`. The main idea of the transformation is to define visual representation of all (complete) patterns sent as input of that module. This way, the module receives a string with `CoordL` patterns and outputs a list of visual representations. Then, on the interesting parts of the `CoordL` abstract grammar, we introduce blocks of code that define the visual representation of each single pattern. To be more precise, the graph of a pattern is constructed, in several productions of the grammar, using C# objects that are synthesized as attributes of the grammar. Later, these objects are encapsulated in a *Graph* object and set into a slot of the output list. Each of these *Graph* objects will be processed (by the GLEE/MSAGL library mechanisms) in order to produce the visual representation of the pattern. An example of a visual representation can be viewed on Figure 6. Although with different objectives, the main process for all the other modules is similar to this one. In fact, this one is imposed by the attribute grammar methodology, and is very efficient and intuitive and with good support on `AnTLR`.

6. Applications and Further Work

Being a DSL, the range of possible applications of `CoordL` is very narrow. Its precise objective of matching coordination traces in a dependence graph reduces its applicability to other areas. Nevertheless, the area of architectural analysis and comprehension allows a deep application of this language.

`CoordInspector` [25] is a tool to extract the coordination layer of a system and to represent it in suitable visual ways. In a fast overview, `CoordInspector` processes *common intermediate language* (`CIL`), meaning that systems written in more than 40 `.NET` compliant languages can be processed by the tool. The tool works by transforming `CIL` code into an `SDG` which is sliced to produce a `CDG`. The tool then uses *ad-hoc* graph notations and rules to perform a blind

search for non-formalised patterns in the CDG. Here is where CoordL has its relevance. Due to its systematisation and robust formal semantics, the process of matching patterns in the code can be more reliable than using the *ad-hoc* rules. The integration of CoordL in CoordInspector led to the development of an editor to deal with the language. Figure 6 presents an overview of the editor integrated in CoordInspector. The editor makes heavy use of the CoordL compiler system, namely the syntax and semantics modules in order to check whether there are or there are not errors in the patterns' specification, and also performs transformations of the patterns into their visual representation.

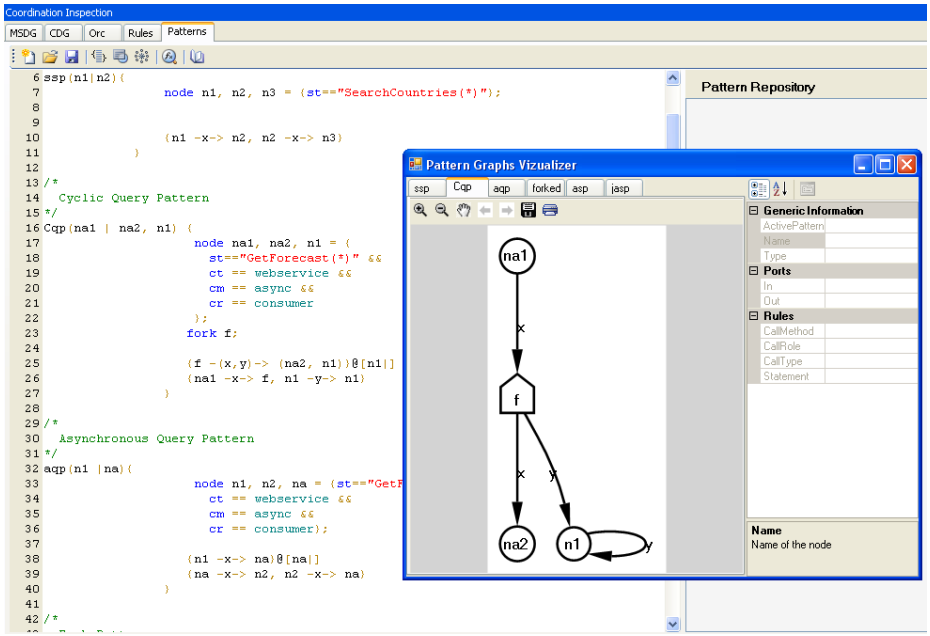

Fig. 6. CoordInspector with CoordL editor

CoordInspector is used for integration of complex information systems, resorting to the recovering of coordination patterns. The use of CoordL in this task is crucial for a faster and systematised search for such parts of code. Although the graph-based search is not finished yet, there is a contract to follow, as close as possible, the algorithm defined in [26], that is the same used to perform the searching for those *ad-hoc* patterns mentioned before.

In order to avoid the repetition of writing recurrent patterns, we decided to create a repository of coordination patterns. The repository may be accessed by means of web services from the editor in CoordInspector. The repository main objective is to give developers and analysts the possibility of expressing

recurrent coordination problems in a `CoordL` pattern and documenting them with valuable information[5]. The existence of the repository of coordination patterns and the fact of being possible the definition of a *calculus* over the language, allows the creation of relations between the patterns, defining an order of patterns.

In what concerns to further work, we believe that the development of a *calculus* over the language would allow the development of a model checker for analysing the properties of these patterns. Also, the application of these patterns to pursuit the work of *van der Aalst* [1] in workflow mining, but applied on a low-level dependence-graph-based search is an interesting perspective for later work. The completion of the graph-based search algorithm and the extraction of code to a newly separated layer is the most important and urgent work to do, in order to finish our approach and start to work in other perspectives.

7. Conclusion

In this paper we introduced a domain-specific language named `CoordL`. This language is used to describe coordination patterns for posterior use in finding and extracting recurrent coordination code compositions in the tangled source code of a software system.

We explained how the language was designed resorting to (i) the application domain description, by means of an ontology, and (ii) existing programming language and associated knowledge. We proceed showing how we took advantage of `AnTLR` to define a full-featured and concern-separated compiler for the language. The adoption of this systematic development of modules for the compiler and the dependencies between them may raise some discussions about the flexibility at maintenance phase. We are aware of such problems, nevertheless we argue that the separation of concerns by modules allows for a better use of the compiler when integrated in other tools, and the problems of maintenance are not that numerous, since the comprehension of the modules is easier due to having a small number of lines of code, and the issue solved in these lines is known *a priori*.

Finally, we argue for the applicability of `CoordL` along with `CoordInspector`[6], a tool to aid in architectural analysis and systems reengineering, and the creation of a pattern repository for (i) cataloguing of valuable information about these coordination patterns and (ii) allowing their adoption reuse by developers and analysts.

References

1. van der Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., Weijters, A.: Workflow mining: A survey of issues and approaches. Data & Knowl-

[5] `http://gamaepl.di.uminho.pt/coordinspector/patternlist.aspx`
[6] `http://gamaepl.di.uminho.pt/coordinspector/`

edge Engineering 47(2), 237–267 (Nov 2003), http://dx.doi.org/10.1016/S0169-023X(03)00066-1
2. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering. pp. 187–197. ACM, New York, NY, USA (2002), http://dx.doi.org/10.1145/581339.581365
3. Allen, R.: A Formal Approach to Software Architecture. Ph.D. thesis, Carnegie Mellon, School of Computer Science (January 1997)
4. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14(3), 329–366 (June 2004), http://dx.doi.org/10.1017/S0960129504004153
5. Bishop, C.: Pattern Recognition and Machine Learning. Springer, Berlin, 1st edn. (2006)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, Chichester, UK (1996)
7. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. 7(3), 215–249 (Jul 1998), http://dx.doi.org/10.1145/287000.287001
8. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices 35, 26–36 (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.8207
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
10. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exper. 30(11), 1203–1233 (2000)
11. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
12. Goguen, A.J.: Reusing and interconnecting software components. Computer 19(2), 16–28 (1986), http://dx.doi.org/10.1109/MC.1986.1663146
13. Goldberg, M., Hayvanovych, M., Hoonlor, A., Kelley, S., Magdon-Ismail, M., Mertsalov, K., Szymanski, B., Wallace, W.: Discovery, analysis and monitoring of hidden social networks and their evolution. In: IEEE Conference on Technologies for Homeland Security (2008). pp. 1–6 (Oct 2008), http://dx.doi.org/10.1109/THS.2008.4637294
14. Heineman, G.T., Councill, W.T. (eds.): Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
15. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. vol. 23, pp. 35–46. ACM, New York, NY, USA (July 1988), http://dx.doi.org/10.1145/53990.53994
16. Jacobson, I., Booch, G., Rumbaugh, J.: The unified software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999), http://portal.acm.org/citation.cfm?id=309683
17. Jen, L.r., Lee, Y.j.: IEEE Recommended Practice for Architectural Description of Software-intensive Systems. IEEE Architecture pp. 1471–2000 (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.9904

18. Lauw, H., Lim, E.P., Pang, H., Tan, T.T.: Social network discovery by mining Spatio-Temporal events. Computational & Mathematical Organization Theory 11(2), 97–118 (Jul 2005), http://dx.doi.org/10.1007/s10588-005-3939-9
19. Lee, L., Kruchten, P.: A Tool to Visualize Architectural Design Decisions. In: Becker, S., Plasil, F., Reussner, R. (eds.) Quality of Software Architectures. Models and Architectures, Lecture Notes in Computer Science, vol. 5281, chap. 3, pp. 43–54. Springer Berlin / Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-87879-7_3
20. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Softw. Eng. 21(4), 336–355 (1995), http://dx.doi.org/10.1109/32.385971
21. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (December 2005), http://dx.doi.org/10.1145/1118890.1118892
22. Misra, Jayadev, Cook, William: Computation orchestration: A basis for wide-area computing. Software and Systems Modeling (SoSyM) 6(1), 83–110 (March 2007), http://dx.doi.org/10.1007/s10270-006-0012-1
23. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Raleigh (2007), http://www.amazon.de/Complete-ANTLR-Reference-Guide-Domain-specific/dp/0978739256
24. Raza, A., Vogel, G., Plödereder, E.: Bauhaus - a tool suite for program analysis and reverse engineering. In: Reliable Software Technologies - Ada-Europe 2006, pp. 71–82. LNCS (4006) (June 2006), http://dx.doi.org/10.1007/11767077_6
25. Rodrigues, N.: Slicing Techniques Applied to Architectural Analysis of Legacy Software. Ph.D. thesis, Engineering School, University of Minho (October 2008)
26. Rodrigues, N.F., Barbosa, L.S.: Slicing for architectural analysis. Science of Computer Programming (March 2010), http://dx.doi.org/10.1016/j.scico.2010.02.002
27. Sartipi, K., Ye, L., Safyallah, H.: Alborz: An interactive toolkit to extract static and dynamic views of a software system. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension. pp. 256–259. IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ICPC.2006.8
28. Storey, M.A.: Theories, tools and research methods in program comprehension: past, present and future. Software Quality Journal 14(3), 187–208 (September 2006), http://dx.doi.org/10.1007/s11219-006-9216-4
29. Tairas, R., Mernik, M., Gray, J.: Using ontologies in the domain analysis of domain-specific languages. Models in Software Engineering pp. 332–342 (2009), http://dx.doi.org/10.1007/978-3-642-01648-6_35

Nuno Oliveira received, from University of Minho, a degree in Computer Science (2007) and a M.Sc. in Informatics (2009), for his thesis “Improving Program Comprehension Tools for Domain Specific Languages”. He is a member of the Language Processing group at CCTC (Computer Science and Technology Center) , University of Minho. He participated in several projects with focus on Visual Languages and Program Comprehension. Currently, he is starting his PhD studies on Architectural Reconfiguration of Interacting Services, under a research grant funded by FCT.

Nuno F. Rodrigues got a degree in “Mathematics and Computer Science”, at University of Minho, and finished a Ph.D. thesis in “Software Architectures” also at University of Minho. Currently he is an Assistant Professor at the Polytechnic Institute of Cavado and Ave, where he is also the director of the Digital Games Development Degree and head of the Digital Games Research Centre.

Pedro Rangel Henriques got a degree in “Electrotechnical/Electronics Engineering”, at FEUP (Porto University), and finished a Ph.D. thesis in “Formal Languages and Attribute Grammars” at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the “Language Processing group” at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the “XML & XSL: da teoria a prática” book, published by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Received: December 28, 2010; Accepted: May 10, 2011.