Domain-Specific Language for Coordination Patterns

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

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

Abstract. The composition of software systems in a new one requires a good architectural design phase to speed up remote communications between these systems. But at the implementation phase, the code to coordinate such components ends up mixed in the main business code. This leads to maintenance problems, raising the need of, on the one hand, separate the coordination code from the business code, and on the other hand provide 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 write 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.

1 Introduction

Software Architecture [12] is a discipline within the Software Development [11] concerned with the design of a system. It embodies the definition of a structure and the organization of components which will be part of the system. The architecture design concerns also with the way these components interact with each other and which are the constraints in their interactions. In their turn, software components [8] may be seen as objects in the object-oriented paradigm, however, besides data and behavior, they may embody whatever we think 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 a system to another, giving birth to component-based software engineering methodology [9].
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 modules of a system. Such interaction definition between the components should be *exogeneous*, that is, the coordination of components is made from the outside of a component, not needing to change its internals to make possible the communication with new components [3].

However, this *rule* of separating computational code from the coordination code is not always adopted by those who develop software. The code is all weaved in a single layer where there is no space for separation of concerns. This behavior would arise problems in the future of the system, namely in maintenance phases. These problems are mainly concerned with the comprehension of the code and architectural decisions by hampering their analysis.

Reverse engineering [20] of legacy systems for coordination layer recovery would play an important role on maintenance phases, diminishing the difficulties on analyzing the architectural decisions. But extracting code dedicated to the coordination of the system components from the whole intricate code is not an easy task. There is not a standard (and unique) way of programming the interactions between the components. However, and fortunately, there are a lot of code patterns which the majority of the developers use to write the coordination code. Once the code of a system can be represented as a graph of dependences between the statements and procedures, the so-called *System Dependence Graph* (`SDG`) [10], we are 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 [18] 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`) [22] named `CoordL`, to write coordination patterns. Our main objective is to translate `CoordL` specifications into a suitable graph representation. Such representation feed a graph-based search algorithm applied on a dependence graph, in order to discover the coordination code weaved in the system code. In more detail, the paper follows this structure: 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 applications of the language and the patterns and finally, in Section 7 we conclude the paper by expressing what we have done.

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 multi-component software system. The main idea of this code separation into concern-oriented layers is to recover architectural decisions and ease the comprehension of the system and its architecture.

The recovery of the system architecture for software comprehension is not a novelty. Tools like Alborz [19] or Bauhaus [16] 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 architecture 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 visualizing 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. The tools mentioned before do not support this feature and do not take advantage of code patterns to do the job.

Although we may reference Architectural Patterns [4] or Design Patterns [5] 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 and a re-engineering to recover architectural decisions, and are focused on low-level compositions of code, the architectural/design patterns play in a higher level, being used to define the architecture of a system in earlier phases of the software development process [11].

Architecture Description Language (`ADL`), are languages to formally describe the architectures and the interactions between the components of the system. 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 of these languages are `ACME` [7], `ArchJava` [1], `Wright` [2], and `Rapide` [13]. The great majority of these languages has tool support for analyzing the described architecture. Such analysis made at high level, allows one to reasoning 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 language with the same exact purpose as `CoordL`.

3 CoordL - Design and Syntax

The design of a `DSL` is always a task embodying a lot of steps. As a first step it is needed to collect all the information about the domain in which the language will actuate. Then this information shall be organized using, for instance, ontologies [21]. Once the main concepts of the domain are identified it is needed to choose those that are really needed to be encapsulated in the syntax of the language; this leads to the last step which concerns with the choice of a suitable syntax for the language.

Figure 1 presents an ontology to organize the domain knowledge of the area where we want to solve problems. 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.

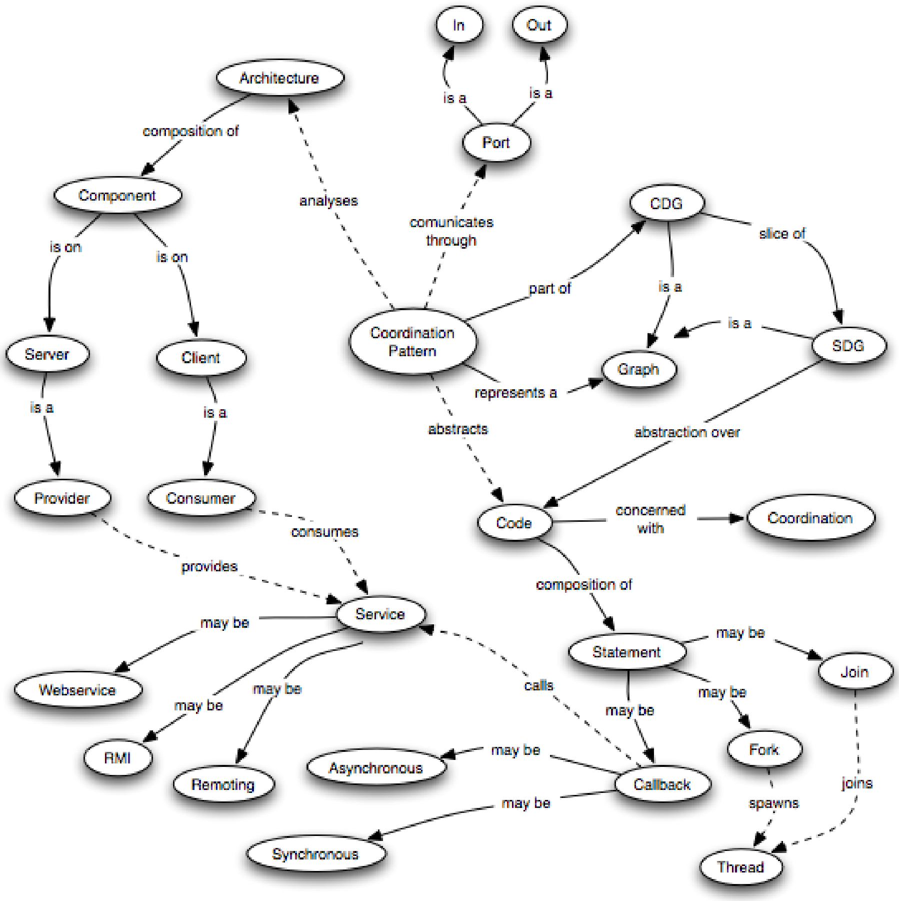

Fig. 1. Ontology Describing the Coordination Pattern Domain Knowledge

Notice also that in this ontology we used operational relations (marked as dashed arrows) besides the normal compositional ones. This would provide a deeper comprehension of how the concepts interact between them 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) [18] abstracting code which is seen as a composition of statements concerned with coordination aspects, and are used to analyze architectures. As novelty the web of knowledge shows that coordination patterns communicate with each other through ports.

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, then some kind of graph representation need to be embodied in the language. An obvious reference for representing graphs in a textual way is the DOT language [6], so, CoordL borrows some aspects from that language. The notion of communication ports (in and out) came from the ACME language [7], although the notion of *ports* is very different in these two contexts. To know which ports exist for a pattern, it was adopted the notion of *arguments* from any *general-purpose programming language* (GPL). The description of what are these ports led to the introduction of declarations and initializations in the language. Declarations describe the types of statements represented as nodes in the graph, while the initialization 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 shown (partially) in Listing 1.1.

Listing 1.1. Partial grammar for CoordL

```
lang → pattern+
pattern → ID ‘(’ ports ‘|’ ports ‘)’ ‘{’ decls graph ‘}’
ports → lstID
decls → (decl ‘;’)+
decl → ‘node’ lstID ‘=’ nodeRules | ‘fork’ lstID | ‘join’ lstID |
 ‘ttrigger’ lstID | ID instances
instances → instance (‘,’ instance)*
instance → ID ‘(’ ports ‘|’ ports ‘)’
...
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 ‘−’ ‘>’
...
```

Figure 2 presents two examples of patterns written with CoordL. The pattern a) is known as the *Asynchronous Sequential Pattern* which is a pattern used when the system has to invoke a series of services but the order of the answer is not important. The pattern b) is known as the *Joined Asynchronous Sequential Pattern*, which 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, pattern in Figure 2.a). It has a unique identifier (pattern_-1) and declares in and out ports, identifiers 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 its right. Then, a space is reserved for node declarations and initializations. There are 5 types of nodes in CoordL: 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 or join.

Nodes of type *node* require an initialization where it is described a list of rules addressing the corresponding coordination code fragment, the type of interaction or 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 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
11   {f2 -(x,w)-> (p3, p2)}
12   {f1 -(x,y)-> (f2, p1)}
13   {p0 -x-> f1}
14 }
```

(a)

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

(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 initialized with unique identifiers referring to all the in and out ports of the pattern typing it.

In `CoordL`, a node is seen as a *pseudo-pattern* (with in and out ports, which may be the node itself). Any operation over these *pseudo-patterns* define a new *pseudo-pattern*. The main operations are the aggregation and the connection. Aggregation[3] is the combination of two or more *pseudo-patterns* by putting them *side-by-side*, this is, not connecting them. The syntax for the aggregation operation is at line 6 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 11, 12 and 13, of `pattern_1` and 7, 8, 9 and 10 of `pattern_2`.

These two operations are used to build the graph of the pattern, which comes after all the node declarations. There are two ways of defining the graph: (i) the implicit composition, where there are only used connection operations and (ii) the explicit composition,

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

where aggregation and connection operations are used simultaneously. The graph of the `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. But, as sometimes it is needed to reuse a node as a in or an out port of a connection, it is needed to re-open them to be used in the sequent connections. This is done with the '@' (alive) operator.

We acknowledge that with all the operators and the associated syntax, the code of the pattern is not easily readable. This way, we defined a visual notation with a suitable "translation" into 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.

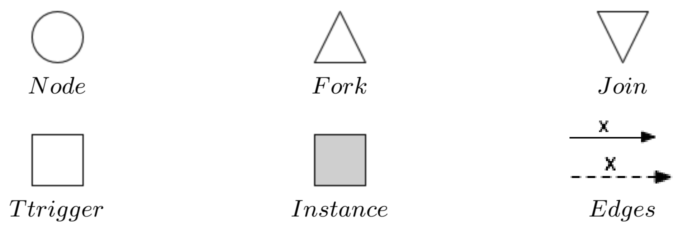

Fig. 3. Components of the Visual Notation for `CoordL`

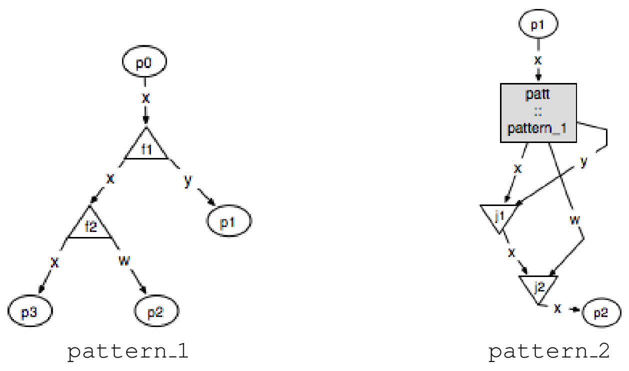

Fig. 4. Visual Representation of Two Coordination Patterns

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 analyzed. The following paragraphs explain, using informal textual description, the 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 the pattern, or in the graph of the pattern, when it is needed to keep ports opened for further use.

Aggregation: $pp_1 + pp_2$
This construct sets two *pseudo-patterns* side by side but do not connect them. This is used to re-inforce the existence of the *pseudo-patterns* in the graph, before connect their ports.

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 two 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 two 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 two nodes in the graph of the pattern, where the end node is a ttrigget. 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 is replied to express what happens when the threads synchronize (tt.sync), or when the threads synchronization 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 that list do not depend on any node, node property or even on other connections that are used and defined in the list. This independence dues to the fact that there is no order between the connections inside a list of connections. Sequent 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 order dependence. 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 are 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 [15] to produce a parser for `CoordL`. Taking advantage of the `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 specified 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.

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, for 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 reckon that the separation of concerns on the modules and the dependence between them may be seen as a problem to maintain 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 brings also 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 would ease the maintenance of each feature.

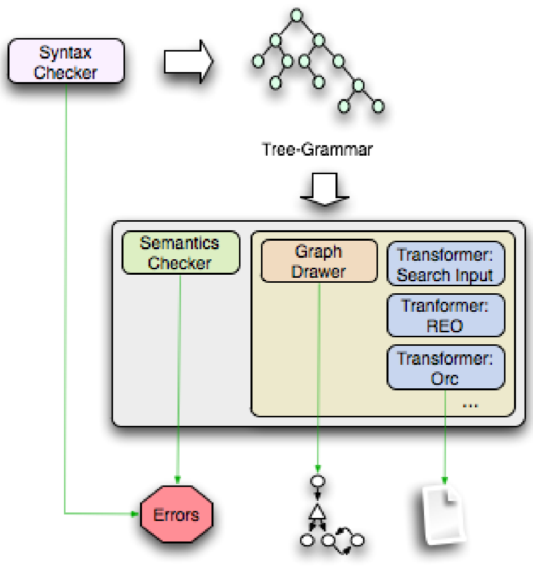

Fig. 5. `CoordL` Compiler Architecture

The transformation modules have, as main objective, to provide perspectives about the coordination patterns, namely, their transformation into Orc [14] or REO [3] specifications. 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 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.

6 Applications and Further Work

The list of possible applications of `CoordL` is not very long. Its precise objective of discovering coordination patterns in a dependence graph reduces its applicability into other areas. Nevertheless, the area of architectural analysis and comprehension allows a profound application of this language.

`CoordInspector` [17] is a tool to extract the coordination layer of a system 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 that tool. The `CIL` code is then transformed into an `SDG` which is sliced to produce a `CDG`. Then it is used *ad-hoc* graph notations and rules to perform a blind search for non-formalized patterns in the `CDG`. Here is where `CoordL` has its relevance. Due to its systematization and robust formal semantics, the process of discovering patterns in the code would 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. 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.

CoordInspector is used for integration of complex information systems, resorting to the discovering of coordination patterns. The use of CoordL in this task is crucial for a faster and systematized search for such parts of code.

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. This repository is not yet finished, but the 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. 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.

As further work we may think about how a *calculus* over the language would allow the development of a model checker for analyzing the properties of these patterns.

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 discovering 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. Then we showed 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 arise some discussions about the flexibility at maintenance phase. We are aware of such problems, nevertheless we argue that the separation of concerns by modules allow a better use of the compiler when integrated in other tools, and the problems of maintenance are not that numerous because 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, 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. Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference*

on Software Engineering, pages 187–197, New York, NY, USA, 2002. ACM.
2. Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997.
3. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.
4. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
6. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.
7. David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
8. A. Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, 1986.
9. George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
10. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, volume 23, pages 35–46, New York, NY, USA, July 1988. ACM.
11. Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
12. Lih-ren Jen and Yuh-jye Lee. IEEE Recommended Practice for Architectural Description of Software-intensive Systems. *IEEE Architecture*, pages 1471–2000, 2000.
13. David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, 21(4):336–355, 1995.
14. Misra, Jayadev, Cook, and William. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling (SoSyM)*, 6(1):83–110, March 2007.
15. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.
16. Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies - Ada-Europe 2006*, pages 71–82. LNCS (4006), June 2006.
17. Nuno Rodrigues. *Slicing Techniques Applied to Architectural Analysis of Legacy Software*. PhD thesis, Engineering School, University of Minho, October 2008.
18. Nuno F. Rodrigues and Luis S. Barbosa. Slicing for architectural analysis. *Science of Computer Programming*, March 2010.
19. Kamran Sartipi, Lingdong Ye, and Hossein Safyallah. 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*, pages 256–259, Washington, DC, USA, 2006. IEEE Computer Society.
20. Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, September 2006.
21. Robert Tairas, Marjan Mernik, and Jeff Gray. Using ontologies in the domain analysis of domain-specific languages. *Models in Software Engineering*, pages 332–342, 2009.
22. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.