Defining and Analyzing P2P Applications with a Data-Dependency Formalism

Ayoub Ait Lahcen, Salma Mouline, Didier Parigot

Defining and Analyzing P2P Applications with a Data-Dependency Formalism

Ayoub Ait Lahcen[a,b], Didier Parigot[a], Salma Mouline[b]
[a]*Zenith Team, Inria Sophia Antipolis, Sophia Antipolis, France*
[b]*LRIT, Unité associée au CNRST URAC 29, Faculté des Sciences, Rabat, Morocco*
Email: {ayoub.ait_lahcen, didier.parigot}@inria.fr, mouline@fsr.ac.ma

***Abstract*—Developing peer-to-peer (P2P) applications became increasingly important in software development. Nowadays, a large number of organizations from many different sectors and sizes depend more and more on collaboration between actors to perform their tasks. These P2P applications usually have a recursive behavior that many modeling approaches cannot describe and analyze (e.g., finite-state approaches). In this paper, we present a formal approach that combines component-based development with well-understood methods and techniques from the field of Attribute Grammars and Data-Flow Analysis in order to specify the behavior of P2P applications, and then construct an abstract representation (i.e., Data-Dependency Graph) to perform analyzes on it.**

***Keywords*-Data-Dependency Formalism; Peer-to-Peer Applications; Data-Flow Analysis.**

I. Introduction

P2P architecture is the concept of an entity acting at the same time as a server and as a client in P2P networks [1]. This is completely different to Client/Server networks, within which the participating entities can act as a server or as a client but cannot embrace both capabilities. Therefore, the responsibilities of entities are approximately equal and each entity provides services to each other as peers.

In software systems, especially those that support P2P applications, data are required for achievement of the computing activity and driving the interactions between software entities. Nevertheless, software system design is usually based on computational aspects with data as an afterthought. A data-centric approach provides a different way of viewing and designing applications. It lets us focus on the flow and transformation of data through the software system.

In this context, we have defined a Data-Dependency Graph (DDG). It has been chosen as an abstract representation for P2P applications for the following two reasons. Firstly, it represents only one data-flow model (dictated by the dependence between data) on the execution. Further, DDG exposes the right level of detail—enough to perform Data-Flow Analysis (DFA).

In this paper, we present a formal approach that combines Component-based Software Engineering (CBSE) [2] with well-understood methods and techniques from the field of DFA [3] (commonly used in compiler construction) in order to construct an abstract representation (i.e., DDG) for P2P applications, and then perform data-flow analyzes on it.

This approach consists of a formalism called DDF (Data-Dependency Formalism). DDF provides the necessary set of operations to specify and analyze P2P applications. DDF can be considered as a minimal and lightweight formalism for the following two reasons. Firstly, the goal of DDF is to formally construct the dependency graph which exposes the right level of detail to perform data-flow analysis. Secondly, DDF is not intended to express business code or to be a general-purpose programming language. This is performed according to Domain-Specific Language (DSL) [4] principles. We note that DDF is highly inspired by the main characteristics of the Attributed Grammars (AGs) because they are able not only to construct similar dependency graphs, but also to naturally capture complex recursive behavior (which is very frequent in P2P applications cf. Section II-A) that many other approaches cannot describe.

This paper is organized as follows. In Section 2, we present in more detail our motivations. In Section 3, we illustrate our approach through the example Gossip protocol. In Section 4, the DDF formalism is presented. In Section 5, we present how Data-Flow Analysis techniques can be used to analyze the dependency graph. Finally, a conclusion is presented in Section 6.

II. Motivations

A. Specificity of P2P applications

Important properties of P2P applications are scalability and self-organization because of their very large user base and the specificity of connections between different peers (e.g., low-bandwidth connections). To support scalability and self-organization in such networks, a large number of P2P-specific algorithms and protocols have been developed. These algorithms and protocols are often executed recursively. Consider, for instance, reputation computation which is a problem of great importance in P2P environments [5] (a simple example justifying this importance is the case where, while downloading files with a P2P file sharing software, we want to choose only reliable peers). The reputation computation relies on a sequence of queries for getting the trust information about a peer *A* and the corresponding responses. This computation must be performed recursively because a response returned from another peer *B* results in a query about the trustworthiness of *B*. In addition, this trust computation needs the reception of all information in

the right order since the cut-off may rely on that order. Such recursive call-backs can be viewed as a sequence of well-formed parentheses if a query call is replaced by a left parenthesis and the corresponding response by a right parenthesis. Therefore, the set of sequences describing these recursive call-backs is a Dyck-Language[1]. It is a well-known result from the formal language theory that a Dyck-Language is not a regular language [6]. Thus, no Finite-State Automaton (FSA) exists that accepts a Dyck-Language.

The kind of recursive call-backs presented above, which has a properly nested structure, can be well defined in terms of context-free languages or Pushdown Automata [3]. However, it is frequently the case that P2P protocols present more complex recursive call-backs which give rise to context-sensitive structures, e.g., interactive structures that adjust their behavior when the context changes. Consider, for example, the case where four neighboring nodes exchange information according to an interaction that corresponds to two interleaved recursive call-backs. Such kind of interaction ($a^n b^m c^n d^m$) is context-sensitive and cannot be described by context-free languages [3].

Referring to the research work on Attribute Grammars (AGs) [7] which are context-sensitive languages, the recursive behavior of P2P applications can be captured by describing both control and data flow of each interaction. In addition, this behavior can be analyzed using DFA techniques.

B. Towards Data-Flow Analysis of component-based P2P applications

1) Model checking and the specificity of P2P applications: Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [8]. It explores all possible states of the system in an exhaustive manner. Model checking has been successfully applied to a wide range of systems such as embedded systems, hardware design and software engineering. Unfortunately, not all systems can take advantage of its power. One reason for this is that some systems cannot be described as a finite-state model. In particular, in the context of P2P applications. Another reason is that model checking is not suited for data-intensive applications (which, in many cases, are developed using the P2P paradigm). The recent book on model checking [8] clearly shows why the verification of data-intensive applications is extremely hard. Even if there are only a small number of data, the state space that must be analyzed may be very large.

2) Verification by Data-Flow Analysis: Data-flow analysis refers to a body of techniques, which derive information about the flow of data along software system execution paths [3]. The execution of a system can be viewed as a series of transformations of the system state, which consists of the values of all the data in the system. Each execution of an intermediate statement transforms an input state to an output state. We denote these data-flow values before and after a statement s by $INPUTS[s]$ and $OUTPUTS[s]$.

To analyze the behavior of a system, we must consider all the possible paths (i.e., sequences of system states) through a flow graph that the system execution can take. Thus, solving a problem in data-flow analysis is reduced to find a solution to a set of constraints (called Data-Flow Equations) on the $INPUTS[s]$ and $OUTPUTS[s]$, for all system statements.

A broad range of system properties can be computed at this level of data abstraction, including some properties like safety and liveness that model checking cannot compute for infinite state systems (cf. e.g., [9]). In addition, several algorithms have been proposed in literature to compute these properties. Unfortunately, to date, the most dominant application of these algorithms, and more generally, Data-Flow Analysis, is in the context of compiler construction. In particular, for Attribute Grammar formalism, which is used to describe the semantic analysis in most compilers.

Our motivation in this context is to use these well-understood methods and techniques from the field of AGs in order to construct an abstract representation for P2P applications and then perform data-flow analyzes on it.

III. Illustrative example: Gossip protocol

In order illustrate that our approach is useful, especially in the context of P2P applications, we explain our dependency formalism in an example that consists of a Gossip protocol [10]. Gossip protocol, also called epidemic protocol, is well-known in the community of P2P. It is mainly used to ensure a reliable information dissemination in a distributed system in a manner closely similar to the spread of epidemics in a biological community. This kind of dissemination is a common behavior of various P2P applications, and according to [11], a large number of distributed protocols can be reduced to Gossip protocol. There exist different variants of Gossip protocol. However, a template that covers a considerable number of those variants has been presented by Jelasity in [11]. In our example, we will rely on this template shown in Algorithm III.

To model this Gossip protocol, we consider a set of nodes, which get activated in each T time units exactly once and then spread data in a network by exchanging messages. Basically, when a node receives data, it responds to the sender and propagates the data to another node in the network (in practice, the data are propagated to a subset of nodes selected according to a specific algorithm). In terms of service, a $node$ is a component that has two activities: serving and consuming data. There are two input services for the serving activity and two output services for the consuming activity. These services are described in the

[1]The Dyck-Language D is the subset of $\{x, y\}^*$ such that if x is replaced by a left parenthesis and y by a right parenthesis, then we obtain sequence of properly nested parentheses [6].

Algorithm 1 The gossip algorithm skeleton *(from [11])*

loop
  $timeout(T)$
  $node \leftarrow selectNode()$
  send $gossip(state)$ to $node$
end
procedure onPushAnswer(msg)
  send $answer(state)$ to $msg.sender$
  $state \leftarrow update(state, msg.state)$
end
procedure onPullAnswer(msg)
  $state \leftarrow update(state, msg.state)$
end

$node\ interface$ as follows:

$$(\{answer(resp : String), gossip(info : String)\}_{in},$$
$$\{gossip(info : String), answer(resp : String)\}_{out})$$

The $gossip$ service is for the propagation of data and the $answer$ service is for sending a response to the sender. The behavior of input services (serving activity) just mirrors the same steps of the output services (consuming activity). From this description of services, we can construct intuitively a simple dependency graph between services, i.e., output services of a $node_x$ are connected to input services of $node_y$, and so on. This graph represents a part of the control flow but it is not very explicit about the data flow. In fact, we do not know the dependencies between services and between data within a $node$.

To complete this interface with a description of both control and data flow, our formalism specifies the behavior with a set of rules:

$$\begin{array}{ll} r_1 : timeout(T) & \rightarrow (gossip(state_x), node_y) \\ r_2 : (gossip(state_y), node_y), [onPush] & \rightarrow (answer(state_x), node_y) \\ r_3 : (gossip(state_y), node_y), [onPull] & \rightarrow \\ r_4 : (answer(state_y), node_y) & \rightarrow \end{array}$$

where, r_1 indicates that the internal service $timeout$ activates the $node_x$ in each T time and then sends the data $state_x$ to $node_y$ through the service $gossip$. r_2 indicates that the $node_x$ receives the data $state_y$ from $node_y$ and then responses by sending the data $state_x$ through the service $answer$ if the condition $onPush$ is satisfied. $onPush$ is a guard condition (to keep things simple, we will ignore guard conditions in this example). r_3 indicates that the $node_x$ receives the data $state_y$ from $node_y$ through the service $gossip$. r_4 indicates that the $node_x$ receives the data $state_y$ from $node_y$ through the service $answer$.

By introducing these rules, the system can be viewed as a set of components where each component has inputs (left side of the rules) and outputs (right side of the rules). The inputs receive data carried by services, and after computation, these data can be sent through outputs. Therefore, we can extract a Data-Dependency Graph of the whole system by connecting together the partial data dependency graphs corresponding to each component used in this system. Once the DDG is defined, we can perform several data-flow analyzes.

IV. Data-Dependency Formalism

Our formalism is highly inspired by the main characteristics of the Attributed Grammars (AGs). AGs were introduced by Knuth [12] and, since then, they have been widely studied [7]. An attributed grammar is an extension of context-free grammar to precisely describe both control and data flow. In this context, an AG's production describes an elementary control-flow that has the following form: $X_0 \rightarrow X_1, ..., X_n$ (X_0 represents a node in a tree and $X_1, ..., X_n$ are its child nodes), whereas a semantic method f describes the computation of the *synthesized* attributes of X_0 and the *inherited* attributes of $X_{1 \leq i \leq n}$. The *synthesized* attributes are the result of the attribute computation, and may use the values of the *inherited* attributes. *Synthesized* attributes are used to pass computed information up the tree, while *inherited* attributes pass information down and across it. Many techniques and algorithms for data-flow analysis were introduced in AG literature and in our previous works (e.g, [13], [14]). These techniques and algorithms are commonly used in compiler construction for performing optimizations from a program's abstract representation (an attribute-dependency graph induced by the Abstract Syntax Tree of the source code). In [14] we have argued that in the term "Attributed Grammar" the notion of *grammar* does not necessarily imply the existence of an underlying tree, and that the notion of *attribute* does not necessarily mean decoration of a tree. We have presented Dynamic Attributed Grammars as an extension to the AG formalism. They are consistent with the general ideas underlying AGs, hence we retain the benefits of the results that are already available in that domain. In the same direction, we explore to use similar techniques to define a Data-Dependency Formalism (DDF) which allows us to construct a Data-Dependency Graph (DDG).

The DDF formalism is essentially dedicated to applications that can be divided into autonomous components communicating to each other over channels. For this purpose, we separate clearly computational activities and component interactions. Thus, we distinguish two types of descriptions, grouped as syntactic and semantic descriptions. The syntactic descriptions consist of a collection of input, output and internal services described only by their signatures. The semantic descriptions consist of interaction rules that define not only the valid sequences of service invocations, but also data exchange required for achieving of the functional activities and driven the interactions between components. We

call *interface* the syntactic part and *behavior* the semantic part.

A. DDF specification

1) Interface: A service is a functional activity supported by a component. If the component provides a service through its interface, the service is called input service; if the component requires a service through its interface, the service is called output service. If the component provides a service that is invoked only by itself, the service is called internal service. A service call refers to an output service or an internal service.

Formally, a service and an interface are defined as follow:

Definition 4.1 (Service): A service is a 3-tuple $\delta =< T, name, arg >$, where:

- T is the service type;
- $name$ is the service name;
- arg is a set of the service arguments.

A service s is written as $s(a_0, ..., a_n)$, its result is denoted by $s\$$ and its arguments are denoted by arg_s with $arg_s = (a_0, ..., a_n)$.

Definition 4.2 (Interface): An interface is a 3-tuple $I =< S_{in}, S_{out}, S_{int} >$, where:

S_{in}, S_{out}, S_{int} are a set of, respectively, input, output and internal services.

2) Component: A component encapsulates data (attributes) with methods to operate on the component's data. Methods implement the services provided through the component interface. A service is implemented by one method. A component contains the declaration of attributes whose values define the state of its instances, along with the bodies of methods that operate on those attributes. A method defined within a component can access only those attributes that are declared within the component, along with any arguments that are passed to the method.

Formally, a component is defined as follows:

Definition 4.3 (Component): A component is a 4-tuple $C =< A, I, Imp, m >$, where:

- A is a set of typed attributes;
- I is an interface;
- Imp is a set of methods (implementing the services provided through the interface). A method is denoted F and defined in Definition 4.6;
- $m : \{S_{in}, S_{out}\} \rightarrow Imp$ is a function that maps each service $s \in (S_{in} \cup S_{int})$ of I to a component method in Imp.

An attribute may be chosen as a component state. State changes are caused by an input, output or internal service. Thus, for the external environment, the input or output services may describe a visible state change. These states may be used by guarded conditions (defined in Section IV-A3) to control the component behavior.

A component may have multiple instances. An instance c_i of a component $C = (A_C, I_C, Imp_C, m_C)$ is denoted by $c_i : C$.

3) Behavior with data dependency: We define the component behavior as a set of rules, where each rule links one input event to some output events (a rule is defined hereafter in Definition 4.6). When a component receives an input event, it will respond to this by executing computations, changing values of its attributes or sending output events. In a rule, the input event is linked to output events by a transition labeled by optional guard conditions. The guard conditions indicate the circumstances under which a rule can be applied. Hence, a rule describes a one-step behavior.

To keep the rule definition simple, we define first input and output event.

Definition 4.4 (Input Event): An input event v of a component $C =< A, I, Imp, m >$ is an element of $(S_{in} \cup S_{int})$.

Definition 4.5 (Output Event): An output event v of a component $C =< A, I, Imp, m >$ is an element of $(S_{out} \cup S_{int})$.

Based on these events, a rule may specify four kinds of events (asynchronous events): receiving an input service, receiving an internal service, emitting an output service and emitting an internal service. Table I gives some examples (with abbreviations) of such events.

Input Event → Output Events	Informal meaning
$s_1(arg_{s_1})[Guards] \rightarrow ...$	receipt of a service $s_1(arg_{s_1})$, where is an input or internal service.
$... \rightarrow s_2\$$	emission of a response $s_2\$$ of a service s_2, where is an input or internal service.
$... \rightarrow s_3(arg_{s_3})$	emission of a service $s_3(arg_{s_3})$, where is an output or internal service.
$s_4\$[Guards] \rightarrow ...$	receipt of a response $s_4\$$ of a service s_4, where is an output / internal service.

Table I
ASYNCHRONOUS EVENTS.

In a rule r, we distinguish three types of data grouped as input, computed and output data. The input data denote the known data used during the computation achieved by the method implementing the service corresponding to the input event of r (this method is called F and it is defined hereafter in Definition 4.6). The input data consist only of internal component attributes and the arguments or result of the service causing the input event. The computed data consist of the results of F and the output data consist of the arguments or result of the service causing the output event. The output data are presented as the union of the input and computed data.

Guard conditions act on the input data. They ensure that the input data are valid or conforms to the conditions before applying the rule. They can be used, for instance, to ensure that two events are mutually exclusive if they occur at the same time.

Formally, a rule is defined as follows:

Definition 4.6 (Rule): A rule describes the execution of an input event v in a component C. It is defined by a 4-tuple $r =< L, Guards, R, E >$, where:

- $L = \{\ v\ \}$ with v is an input event. L represents the left side of the rule;
- $Guards$ are the guard conditions, indicating the circumstances under which the input event v can be executed. A guard condition consists on a set of Boolean expressions. An input event v is executed if each Boolean expression is true;
- $R = \{v_1, ..., v_n |\ \forall i \in 1..n, v_i$ is an output event$\} \cup \{\emptyset\}$. R represents the right side of the rule;
- E is a semantic equation which has the following form:

$$(b_0, ..., b_q) = F(a_0, ..., a_p) \tag{1}$$

where F is a method that implements the service corresponding to the input event v and defines the computation of the output data (b_i) in terms of the input data (a_i).

Before giving the definition of the constraints on the equation E, we define first three sets of data: Input Data ID_r, Computed Data CD_r and Output Data OD_r.

Definition 4.7 (Input data ID_r of a rule r): Let a rule $r =< L, Guards,\ R, E >$ describes the execution of an input event $v \in L$ in a component $C =< A, I, Imp, m >$, the input data ID of r are:

$$v \in L, ID_r = \begin{cases} arg_s \cup A & \text{if} \quad v = s(arg_s) \\ \\ \{s\$\} \cup A & \text{if} \quad v = s\$ \end{cases} \tag{2}$$

Definition 4.8 (Computed data CD_r of a rule r): Let a rule $r =< L,\ Guards, R, E >$, computed data CD of r are the set of data resulting from the equation E:

$$CD_r = \{\ b_0, ..., b_q\ \} \tag{3}$$

Definition 4.9 (Output data OD_r of a rule r): Let a rule $r =< L, Guards,\ R, E >$, output data OD of r are the data emitted by the output events of r:

$$OD_r = \bigcup_{v_i \in R} \begin{cases} arg_s & \text{if} \quad v_i = s(arg_s) \\ \\ \{s\$\} & \text{if} \quad v_i = s\$ \end{cases} \tag{4}$$

Once these three sets of data are defined, the constraints on the semantic equation E of a rule r can be defined as follow:

Definition 4.10 (Constraints of a semantic equation): The constraints to be satisfied by a semantic equation $E : (b_0, ..., b_q) = F(a_0, ..., a_p)$ of a rule r are:

- Contraint (1): OD_r elements can only be elements of the union of ID_r and CD_r:

$$OD_r \subseteq ID_r \cup CD_r \tag{5}$$

- Contraint (2): F only accepts ID_r elements as inputs:

$$\forall i \in 0..p,\ a_i \in ID_r \tag{6}$$

In right side R of a rule, output events (separated by ",") may be output service emitted to different remote components, and each component is a process that can be executed separately. This parallel relation between output events is nearly implicit. For example, $r : s \rightarrow s_1, s_2$ means services s_1 and s_2 do not have sequential relation.

This relation characterizes the activity of a unique rule. So, in order to characterize the activity of a set of rules, we define three operations for rules:

- *Sequence operation* " ; ": Indicating a sequential order among rules. For example, $r_1; r_2; r_3$ means rule r_1 acts before r_2 and r_2 acts before r_3.
- *Alternative operation* " | ": Indicating an alternative choice concerning the output events of a rule. For example,
$r : s[Guards] \rightarrow \quad s_1$
$\qquad\qquad\qquad\qquad | \ s_2$
means services s_1 and s_2 may have same chance to occur. This alternative can be controlled by the guard conditions.
- *Recursive operation* "[ ]": Indicating that an internal service s will be called recursively. This recursion can be controlled by the guard conditions. Thus, recursion operations can be used to have repetition (loop) indicating that some rules will be executed n times continuously. For example,
$[r_1 : \quad s[Guards] \quad \rightarrow s_1$
$\ r_2 : \quad s_1\$ \qquad\quad \rightarrow s]$
means that the rule r_1 execute the internal service s if guard conditions are satisfied, and then it calls the service s_1. When the service s_1 response arrives, the rule r_2 calls the internal service s, which will be executed again by r_1 if the guards are still satisfied.

Therefore, from the definition of an interface, a rule and rule operations, we have the following definition of a component behavior.

Definition 4.11 (Behavior): The behavior of a component C is a set of rules combined by sequence, alternative and recursion operations with respect to the following regular expressions:

$$B ::= r^+ \mid [B^+] \mid \{B^+\} \tag{7}$$

$$r ::= r \mid (r \backslash r) \tag{8}$$

4) System: The component composition is based on connections among component instances. A connection between two instances occurs when one of them provides its interface and another instance uses it. Hence, input (resp. output) services are connected to signature-matching output (resp. input) services. There is a unique connection between two instances.

Once component instances are connected, the behavior of the entire resulting system is obtained by composition of behaviors of participating instances. Since one rule is a one-step behavior and the component instance behavior is a set of rules connected by sequence, alternative and recursive operations, the system behavior can be again viewed as a set of rules connected by these same operations.

Formally, a system is defined as follows:

Definition 4.12 (System): A system is defined by a 2-tuple $Sys =< Inst, T >$ where:

- $Inst$ is a set of component instances;
- $T = \{(c_1, c_2)|(c_1, c_2) \in Inst \times Inst\}$ is a set of connections between component instances.

V. System analysis

As described in Section II-B2, Data-Flow Analysis refers to a body of techniques, which derive information about the flow of data along software system execution paths in order to infer or compute some system properties. To achieve this, we must first consider all the possible paths through a flow graph that the system execution can take. Therefore, we have defined a Data-Dependency Graph. It presents an abstract representation of the system. This abstraction exposes the right level of detail to perform DFA.

The DDG models the flow of data values from the point where a datum value is created, a definition, to any point in a system where it is used, a use. A node in a DDG represents a low-level operation on data. In most cases, nodes contain both definitions and uses. A directed edge in a DDG connects two nodes (head and tail). The head defines a datum value and the tail uses it. The edges in the DDG represent interesting constraints on the control flow, i.e., a datum value can be used only if it has been defined. This only implies a partial order on the execution. Therefore, no total order among system operations is needed to be given by the system designer who often set it as an automaton to perform analysis. Moreover, it is possible through a data-flow analysis on this graph to infer various data evaluation orders during run time (e.g., total, parallel and incremental). Thanks to the theory of iterative data-flow analysis based on a fixed-point theorem [15].

VI. Conclusion and perspective

This paper presents a formalism called DDF (Data-Dependency Formalism). The goal of DDF is to formally specify the behavior of P2P applications, and then construct an abstract representation (i.e., Data-Dependency Graph) to perform analyzes on it. We note that our approach shares with the theory of Attribute Grammars [7] the same semantics of the Data-Dependency Graph. The theoretical algorithms and techniques of AGs and DFA show that it is possible through analysis on these dependency graphs to infer various evaluation orders of data and compute different properties. The reliability of those algorithms was proven in different works [7], and optimized variants were presented in our previous works, e.g., [13], [14]. A reformulation of some of these AGs analysis/testing algorithms is in progress. In particular, an algorithm that infers the evaluation orders of data to determine formally which services in a system can be executed in a parallel or incremental way. In a future work, we plan to extend our formalism by program transformation mechanisms in order to optimize CPU and memory usage (by analyzing lifetime of data taking into account their functional dependencies and redundancies) in large-scale data-centric applications. Especially, in the emerging Cloud Computing area, where data management has been receiving significant attention.

References

[1] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," *Peer-to-Peer Computing, IEEE International Conference on*, vol. 0, p. 0101, 2001.

[2] C. Szyperski, *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[4] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[5] K. Aberer and Z. Despotovic, "Managing trust in a peer-2-peer information system," in *Proceedings of the tenth international conference on Information and knowledge management*, ser. CIKM '01. New York, NY, USA: ACM, 2001, pp. 310–317.

[6] R. Stanley, *Enumerative combinatorics*, ser. Cambridge studies in advanced mathematics. Cambridge University Press, 2001.

[7] P. Deransart, M. Jourdan, and B. Lorho, *Attribute grammars: definitions, systems and bibliography*. New York, NY, USA: Springer-Verlag New York, Inc., 1988.

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, May 2008.

[9] R. Govindarajan, S. Yu, and V. S. Lakshmanan, "Attempting guards in parallel: A data flow approach to execute generalized guarded commands," *International Journal of Parallel Programming*, vol. 21, pp. 225–268, 1992.

[10] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, August 2007.

[11] M. Jelasity, "Gossip," in *Self-organising Software*, ser. Natural Computing Series, G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos, Eds. Springer Berlin Heidelberg, 2011, pp. 139–162.

[12] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968.

[13] M. Jourdan and D. Parigot, "Techniques for improving grammar flow analysis," in *Proceedings of the third European symposium on programming on ESOP '90*. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 240–255.

[14] D. Parigot, G. Roussel, M. Jourdan, and E. Duris, "Dynamic attribute grammars," in *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, vol. 1140. Springers, 1996, pp. 122–136.

[15] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *J. ACM*, vol. 23, pp. 158–171, 1976.