Computing and Informatics, Vol. 36, 2017, 353–385, doi: 10.4149/cai_2017_2_353

# DATA-DEPENDENCY FORMALISM FOR DEVELOPING PEER-TO-PEER APPLICATIONS

Ayoub Ait Lahcen

*LGS, ENSA, Ibn Tofail University, Kenitra, Morocco*
*&*
*LRIT, Faculty of Sciences, Mohammed V University in Rabat, Morocco*
*e-mail:* ayoub.aitlahcen@univ-ibntofail.ac.ma

Didier Parigot

*Zenith Team, Inria, Sophia Antipolis, France*
*e-mail:* didier.parigot@inria.fr

Salma Mouline

*LRIT, Faculty of Sciences, Mohammed V University in Rabat, Morocco*
*e-mail:* 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 an 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 construct an abstract representation (i.e. Data-Dependency Graph) for P2P applications, and then perform data-flow analyzes on it. This approach embodies a formalism called DDF (Data-Dependency Formalism) to capture the behavior of P2P applications and construct their Data-Dependency Graphs. Various properties can be inferred and computed at the proposed level of data abstraction, including some properties that model

checking cannot compute if the system presents a recursive behavior. As examples, we present two algorithms: one to resolve the deadlock problem and another for dominance analysis.

**Keywords:** Data-dependency formalism, data-dependency graph, application development, peer-to-peer (P2P), data-flow analysis

## 1 INTRODUCTION

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 (individuals, groups, communities, etc.) to perform their tasks. P2P architecture is the concept of an entity acting at the same time as a server and as a client in P2P networks [1, 2]. This is completely different from Client/Server networks, within which the participating entities cannot 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 an approach that combines Component-based Software Engineering (CBSE) [3] with well-understood methods and techniques from the field of DFA [4] (commonly used in compiler construction) in order to construct an abstract representation (i.e. DDG) for P2P applications, and then perform data-flow analyses on it. This approach embodies a formalism called DDF (Data-Dependency Formalism) to capture the behavior of P2P applications and construct their DDGs.

DDF formalism 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) [5] 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 capture complex recursive behavior (which is very frequent in P2P applications, cf. Section 2.1) that many other approaches cannot describe. Thus, our rule-based formalism is able to naturally capture this kind of behavior. In fact, it is a well-known result from the formal language theory that Finite-State Automata (FSA) cannot capture such behavior [4]. This implies that FSA-based approaches used to model software applications cannot describe and analyze it. In particular, in the context of CBSE, a large body of component behavior modeling approaches can be reduced to FSA. The well-known component models SOFA [6] and Fractal [7] clearly raise this issue. For instance, in [6] the authors say: *"our approach cannot treat behavior that cannot be modeled by a regular language (e.g. recursion)."*

This paper is organized as follows. In Section 2 and 3, we present in more detail our motivations. In Section 4, the DDF formalism is presented and illustrated through the case-study Gossip protocol. In Section 5, we present how Data-Flow Analysis techniques can be used to analyze the dependency graph with an effective application to deadlock and dominance detection. Section 5 presents related work. Finally, Section 6 concludes and presents future work.

## 2 MOTIVATIONS

### 2.1 Specificity of P2P Applications

Important properties of P2P applications are scalability and self-organization, necessary because of their very large user base and the specificity of connections between different peers (e.g. low-bandwidth connections) [8]. 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[1] which is a problem of great importance in P2P environments [9] (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). Reputation computation is based on a sequence of queries for getting the trust information about a peer $A$ and their corresponding responses. Such computation should be performed recursively since a response returned from another peer $B$ is the result of a query about the truthfulness of $B$. In addition, during this trust computation, we must receive all information in the correct order since the cut-off might rely on that order [10]. 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,

[1] We note that reputation computation presents a particular case of information dissemination and can be performed using Gossip protocol.

the set of sequences describing these recursive call-backs is a Dyck-Language[2]. It is a well-known result from the formal language theory that a Dyck-Language is not a regular language [11]. Thus, no Finite-State Automaton (FSA) exists that accepts the Dyck-Language. A formal proof can be found in diverse books on language theory, e.g. the textbook of Aho et al. [4].

The kind of recursive call-backs presented above, which has a properly nested structure, can be well defined in terms of Pushdown Automata or context-free languages [10] (discussed in Section 5). However, it is frequently the case that P2P protocols present more complex recursive call-backs which give rise to context-sensitive structures (interactive structures that adjust their behavior when the context changes). Consider, for example, the case where four neighboring peers 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 [4].

Referring to the research work on Attribute Grammars (AGs) [12] 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.

### 2.2 Exploring Data-Centric Approach for the CBSE

Component-based approach became increasingly important in software engineering. This emerges from the need to use component-based approach concepts to implement services and raise the level of abstraction by easing packaging, reusing, extending, customizing and composing services [13]. Thus, services can be encapsulated and their interfaces can be exposed into cohesive components to assist in the creation of new applications. Hence, component-based approach yields promising benefits such as service composition, reusability and adaptation. However, the data manipulated by services to produce actionable results and which drive component interactions are considered as an afterthought. Whereas, the data are incorporated as an important part of the development of systems in several research areas such as desktop grid [14], business intelligence and P2P systems. Recently, in the emerging cloud computing area, where everything is a service, data management has been receiving significant attention and has led to much excitement [15]; this can only increase.

Our motivation in this context is to investigate the applicability of the data management for software component systems by allowing run-time data to be specified, viewed and analyzed, especially in P2P environments. While many of the current component approaches emphasize the structural and functional aspects of component composition, we insist on modeling flow and dependencies of run-time data, because the interactions between components are due to exchanged data. Thus, it is

[2] 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 [11].

our belief that data must be considered to be an integral part of design and behavior specifications of component-based systems.

### 2.3 Towards Data-Flow Analysis of Component-Based P2P Applications

#### 2.3.1 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 [16]. 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 the advantage of its power. One reason for this is that some systems cannot be described as finite state models, in particular, in the context of P2P applications (cf. Section 2.1). Another reason is that model checking is not suited for data-intensive applications (which, in many cases, are developed using the P2P paradigm, cf. e.g. [17]). The recent book on model checking of Baier and Katoen [16] clearly shows why the verification of data-intensive applications is extremely hard. Even if there is only a small amount of data, the state space that must be analyzed may be very large. The authors even consider that this is one of the first weaknesses:

"*The weaknesses of model checking:*

- *It is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.*
- *Its applicability is subject to decidability issues; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.*
- *. . .* "

#### 2.3.2 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 [4]. The execution of a system can be viewed as a series of transformations of the system state, which consists of the values of all 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 $OUTPUTS[s]$, respectively.

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 finding a solution to a set of constraints (called Data-Flow Equations) on the $INPUTS[s]$ and $OUTPUTS[s]$, for all system statements $s$. There are two sets of constraints:

- Semantic constraints: they define the relationship between $INPUTS[s]$ and $OUTPUTS[s]$ of each statement $s$. This relationship is usually presented as a transfer method $f$ that takes the $INPUTS[s]$ before the statement and produces $OUTPUTS[s]$ after the statement. That is, $OUTPUTS[s] = f_s(INPUTS[s])$.
- Control-flow constraints: If a system consists of statements $s_1, s_2, \ldots, s_n$, in that order, therefore, the control-flow value out of $s_i$ is the same as the one into $s_{i+1}$. That is, $INPUTS[s_{i+1}] = OUTPUTS[s_i]$, for all $i = 1, 2, \ldots, n-1$.

For example, to verify a property such as liveness of data that determines whether a datum is used in the future along some path in the flow graph, we shall set up the constraints for liveness of data (i.e. define the data-flow equations specifying that a datum $d$ is live at a system point $p$ if some path from $p$ to its end contains a use of $d$). These equations can be solved using an iterative algorithm form as a fixed-point solution. The convergence of the algorithm is assured by the theory of iterative data-flow analysis [18], which demonstrates that a unique fixed point exists for these equations. Liveness information can be very useful. For instance, if the result of a datum assignment in a software system is not used along any subsequent execution path, then the assignment is considered as dead code that we can eliminate. In Section 4.3, we provide another example (detection of dominance) that illustrates in more details the principles of data-flow analysis.

A broad range of other 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. [19]). 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, are 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 combine CBSE with the 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 this abstract representation.

### 2.4 Illustrative Example: Gossip Protocol

In order to motivate and 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 [20, 21]. 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 Jelasity [22],

**Algorithm 1** The gossip algorithm skeleton *(from Jelasity [22])*

**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**

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 [22]. In our example, we will rely on this template shown in Algorithm 2.4.

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 *node interface* as follows:

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

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:

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

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 responds 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.

## 3 DATA-DEPENDENCY FORMALISM

Our formalism is inspired by the Attributed Grammars (AGs) and was introduced in [23, 24]. AGs were introduced by Knuth [25] and, since then, they have been widely studied [12, 26, 27]. 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: $X_0 \rightarrow X_1, \ldots, X_n$ ($X_0$ represents a node in a tree and $X_1, \ldots, X_n$ 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. [28, 29]). 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 [29] 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

Data-Dependency Formalism (DDF) which allows us to construct Data-Dependency Graph (DDG) to perform data-flow analyzes on it.

The DDF formalism is essentially dedicated to applications that can be divided into autonomous components communicating with 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 (*interface*). The semantic descriptions consist of interaction rules (*behavior*).

### 3.1 DDF Specification

#### 3.1.1 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.

An internal service represents a particular action of a component. To describe, for example, time sequence (one component's behavior occurs after some time), an internal service $timer(timeout : Int)$ can be used to represent a timer. This internal service *timer* has an argument *timeout*, which can be set as an integer. Once $timer.timeout$ is set, the component's behavior can only occur when $timer.timeout = 0$.

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

**Definition 1** (Service)**.** A service is a 3-tuple $\delta = \langle type, name, arg \rangle$, where:

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

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

**Definition 2** (Interface)**.** An interface is a 3-tuple $I = \langle S_{in}, S_{out}, S_{int} \rangle$, where: $S_{in}$, $S_{out}$, $S_{int}$ are sets of, respectively, input, output and internal services.

#### 3.1.2 Component

A component encapsulates data (attributes) with methods to operate on the component's data. Methods implement the services provided through the component interface. Each service is implemented by one method. A component contains the

declaration of attributes values of which 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.

The component prohibits concurrent access to its methods. Only one method can be run within the component at any one time. Consequently, the programmer does not need to code this synchronization explicitly; it is built into the component. This technique is widely used in operating systems [30] to simplify reasoning about the implementation of concurrent distributed applications.

During run time, a component might need inputs. When it receives an input, the component will respond to this by executing its methods and/or changing its state (attributes). Otherwise, without inputs, a component may produce an output and/or change its states. This output may have an eventual response as an input.

Formally, a component is defined as follows:

**Definition 3** (Component)**.** A component is a 4-tuple $C = \langle A, I, Imp, m \rangle$, 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 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 3.1.3) 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 $c_i : C$.

### 3.1.3 Behavior with Data Dependency

As in the grammar-based modeling methods which are well suited to describing the control logic for the processing of data streams [31], the aim of our specification is to describe in a structured way what the control logic does while striving not to describe how the control logic is computed or implemented. By *"what"* we mean describing the sub-behaviors (called rules) of the control logic and by *"how"* we mean describing the lower-level implementation details (usually presented as states, transitions, encodings and other details of a FSA controller).

This choice to separate, as far as possible, what is computed from how it is computed has been especially made in the grammar-based approaches for the following reasons. Firstly, when the complexity of the control logic increases, describing the states and transitions of a FSA controller implementing the control logic becomes

problematic. FSA controller of even a few states can have a large number of transitions and if some modifications should be made in the control logic, the FSA can change considerably. Secondly, the lower-level specifying how things are computed can be synthesized from the high-level control specification.

Typically, the synthesis begins with the construction of an abstract representation of the design (Data-Dependency Graph in our case) and then a translation (or transformation) is performed to obtain an initial FSA representation. In our case, and as in Attribute Grammars, we look to have a data/attribute evaluator (which consists of a set of DFA algorithms) rather than a FSA controller. The advantage of a data evaluator comes from the fact that not only one but multiple implementations of the control logic can be synthesized by analyzing the order of data evaluation (incremental, partial, total, parallel, etc.).

Thus, our method is based on describing the sub-behaviors of the control logic as a set of rules. The total behavior of a design is described by composing together the rules using compositional operators. Each rule links one input event to some output events (see Definition 6). When an input event is received, a rule 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.

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

**Definition 4** (Input Event)**.** An input event $v$ of a component $C = \langle A, I, Imp, m\rangle$ is an element of $(S_{in} \cup S_{int})$.

**Definition 5** (Output Event)**.** An output event $v$ of a component $C = \langle A, I, Imp, m\rangle$ 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 1 gives some examples (with abbreviations) of such events.

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

Table 1. Asynchronous events

To take into account the synchronous events, we introduce a synchronization (a rendez-vous) symbol $\uparrow$. Thus, when a service is called, the caller waits until the service response returns. We describe this kind of event in Table 2.

| Input Event $\rightarrow$ Output Events | Informal Meaning |
|---|---|
| $\ldots \rightarrow s_1(arg_{s_1}) \uparrow$ | emission of a service $s_1(arg_{s_1})$, and waiting for its result. |

Table 2. Synchronous event

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 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 6** (Rule)**.** A rule describes the behavior of a component $C$ when it receives an input event $v$. A rule is defined by a 4-tuple $r = \langle L, Guards, R, E \rangle$, 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;
- $R = \{v_1, \ldots, v_n \mid \forall i \in 1..n, v_i \text{ is an output event}\} \cup \{\emptyset\}$. $R$ represents the right side of the rule;
- $E$ is a semantic equation which has the following form:

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

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 a 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 7** (Input data $ID_r$ of a rule $r$)**.** Let a rule $r = \langle L, Guards, R, E\rangle$ describe the behavior of a component $C = \langle A, I, Imp, m\rangle$ when it receives an input event $v$, 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 8** (Computed data $CD_r$ of a rule $r$)**.** Let a rule $r = \langle L, Guards, R, E\rangle$, computed data $CD$ of $r$ are the set of data resulting from the equation $E$:

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

**Definition 9** (Output data $OD_r$ of a rule $r$)**.** Let a rule $r = \langle L, Guards, R, E\rangle$, 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 follows:

**Definition 10** (Constraints of a semantic equation)**.** The constraints to be satisfied by a semantic equation $E : (b_0, \ldots, b_q) = F(a_0, \ldots, a_p)$ of a rule $r$ are:

- Constraint (1): $OD_r$ elements can only be elements of the union of $ID_r$ and $CD_r$:
$$OD_r \subseteq ID_r \cup CD_r. \tag{5}$$
- Constraint (2): $F$ only accepts $ID_r$ elements as inputs:
$$\forall i \in 0..p, a_i \in ID_r. \tag{6}$$

In the right-hand 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,

$$r : s[Guards] \rightarrow \begin{array}{l} s_1 \\ \mid s_2 \end{array}$$

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,

$$\begin{array}{rll} [r_1 : & s[Guards] & \rightarrow s_1 \\ r_2 : & s_1\$ & \rightarrow s] \end{array}$$

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 guard conditions 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 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 expression:

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

### 3.1.4 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 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 12** (System)**.** A system is defined by a 2-tuple $Sys = \langle Inst, Con \rangle$ where:

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

Now, we define the system behavior from the behavior of each underlying component instance. To achieve this, we associate the source and the destination instances to the events of the rules. For example, let a rule $r : v \rightarrow v_1, v_2$ describe the behavior of a component $C$ when receiving the input event $v$, where $v \in S_{in}$ and $S_{in} \in I_C$, and let connections $(c, c_i)$ and $(c, c_j)$, where $c$, $c_i$ and $c_j$ are instances of, respectively, $C$, $C_i$ and $C_j$ components. The rule $r$ will be transformed to $(v, c_i) \rightarrow (v_1, c_j), (v_2, c_i)$ if the source instance causing the input event $v$ is $c_i$ and the destination instances of the output events $v_1$ and $v_2$ are $C_j$ and $C_i$, respectively. This transformation is performed by a function, which specifies in each rule the source component instance causing its input event and the destination component instances of its output events.

**Definition 13** (Rule Transformation)**.** Let a rule $r = \langle L, \mathit{Guards}, R, E\rangle$ describe the behavior of a component $C_i$ when it receives an input event $v \in L$, and let a connection $(c_i, c_j) \in \mathit{Con}$, where $c_i$ and $c_j$ are instances of, respectively, $C_i$ and $C_j$ components. The transformation of $r$ when $c_i$ is connected to $c_j$ is $r^{c_i} = \sigma(r)_{/c_i \rightarrow c_j}$, where:

$$\begin{aligned}\sigma(r)_{/c_i \rightarrow c_j} &= \sigma(r : v \rightarrow v_1 \dots v_n)_{/c_i \rightarrow c_j} \\ &= r : \sigma_r(v)_{/c_i \rightarrow c_j} \rightarrow \sigma_r(v_1)_{/c_i \rightarrow c_j} \dots \sigma_r(v_n)_{/c_i \rightarrow c_j}\end{aligned}$$

where the transformation function $\sigma$ is defined as follows:

$$\sigma \quad : \quad V \quad \overset{r,/c_i \rightarrow c_j}{\longrightarrow} \quad V \times \mathit{Inst} \text{ or } V,$$

$$\sigma_r(v)_{/c_i \rightarrow c_j} = \begin{cases} (v, c_j) & \text{if } v \in r.L \wedge v \in S_{in}(c_i) \cap S_{out}(c_j), \\ (v, c_j) & \text{if } v \in r.R \wedge v \in S_{out}(c_i) \cap S_{in}(c_j), \\ v. & \end{cases} \tag{8}$$

Therefore, the system behavior is defined as follows:

**Definition 14** (System Behavior)**.** A system behavior $B(\mathit{Sys})$ is a set of rules combined by sequence, alternative and recursion operations, where:

$$B(\mathit{Sys}) = \bigcup_{(c_i, c_j) \in T} \{B(c_i)_{/c_i \rightarrow c_j} \cup B(c_j)_{/c_j \rightarrow c_i}\}, \tag{9}$$

$$B(c_x)_{/c_x \rightarrow c_y} = \{\sigma(r)_{/c_x \rightarrow c_y} | r \in B(c_x) \wedge r.L \in S_{in}(c_x) \cap S_{out}(c_y)\}. \tag{10}$$

### 3.2 Case Study

In this section, a detailed DDF specification of the Gossip protocol [20, 21] is described.

In a Gossip protocol, each node in the network periodically exchanges information with a subset of other nodes. In fact, every node maintains a local membership

table providing a partial view on the complete set of memberships and periodically refreshes the table using a gossiping procedure. The table (view) is a list of $c$ node descriptors, where $c$ is the size of the list and is the same for all nodes. A node descriptor contains a node network *address* and an *age* that represents the freshness of the given node descriptor. The list changes by means of usual list operations (e.g., permute) that are defined on it. Therefore, the tables reflect the dynamics of the system by continuously changing random subset of the nodes (in the presence of failure and joining and leaving of nodes).

The protocol consists of two activities (serving and consuming) in each node: an active client gets activated in each $T$ time units exactly once and then initiates communications with other nodes, and a passive server awaits for and answers these requests. The behavior of the passive server just mirrors the same steps of the active client. In terms of DDF, each activity corresponds to a pair of rules given in Table 3.

This table describes the behavior of a *Gossip System* constituted of two nodes ($node_x$ and $node_y$) and the associated methods (implementations) extracted from [21]. This system is formally defined with DDF as follows:

According to Definition 2, the interface of the component *Node* is expressed as $I_{Node} = \langle S_{in}, S_{out}, S_{int} \rangle$, where:

$$\begin{aligned} S_{in} &= \{gossip(buffer : Buffer), answer(buffer : Buffer)\}; \\ S_{out} &= \{gossip(buffer : Buffer), answer(buffer : Buffer)\}; \\ S_{int} &= \{timeout(T : TimeUnit)\}. \end{aligned}$$

According to Definition 3, the component *Node* is expressed as $Node = \langle A, I, Imp, m \rangle$, where:

- $A = \{view : List(IP : Address, age : Int), c : Int, push : Bool, pull : Bool, T : Time, H : Int, S : Int\}$;
- $I = I_{Node}$;
- $Imp = \{F_{r_1}(), F_{r_2}(), F_{r_3}(), F_{r_4}()\}$;
- $m : \{S_{in}, S_{out}\} \rightarrow Imp$.

$A$ attributes are described in Table 4.

According to Definition 11, the behavior of component *Node* is expressed as $B_{Node} = \{r_1, r_2, r_3, r_4\}$, where:

$$\begin{array}{lllll} r_1: & timeout(T) & \rightarrow gossip(buffer_{in}) & F_{r_1} \\ r_2: & answer(buffer_{out}) & \rightarrow & F_{r_2} \\ r_3: & gossip(buffer_{out}), [pull] & \rightarrow answer(buffer_{in}) & F_{r_3} \\ r_4: & gossip(buffer_{out}), [\neg pull] & \rightarrow & F_{r_4} \end{array}$$

According to Definition 12, the *Gossip System* is expressed as $GossipSys = \langle Inst, Con \rangle$, where:

| **Gossip System Behavior** | **Hidden Implementations** |
|---|---|
| **Server Activity** ($node_x$) | |
| $r_1^x : timeout(T) \rightarrow (gossip](buffer_x), node_y)$ | $(buffer_x) = F_{r_1}()\{$<br>$p = view.selectPeer()$<br>**if** $(push)$<br>$buffer_x = (myAddress, 0)$<br>$view.permute()$<br>$view.moveOldestH()$<br>$buffer_x.append(view.head(c/2-1))$<br>$gossip(buffer_x)$<br>**else**<br>$buffer_x = null$<br>$gossip(buffer_x)$<br>$view.increaseAge()$<br>$\}$ |
| $r_2^x : (answer(buffer_y), node_y) \rightarrow$ | $F_{r_2}(buffer_y)\{$<br>**if** $(pull)$<br>$view.select(c, H, S, buffer_y)$<br>$\}$ |
| **Customer Activity** ($node_y$ ) | |
| $r_3^y : (gossip(buffer_x), node_x), [pull] \rightarrow (answer(buffer_y), Node_x)$ | $(buffer_y) = F_{r_3}(buffer_x)\{$<br>$buffer_y = (MyAddress, 0)$<br>$view.permute()$<br>$view.moveOldestH()$<br>$buffer_y.append(view.head(c/2-1))$<br>$answer(buffer_y)$<br>$\}$ |
| $r_4^y : (gossip(buffer_x), node_x), [\neg pull] \rightarrow$ | $F_{r_4}(buffer_x)\{$<br>$view.select(c, H, S, buffer_x)$<br>$view.increaseAge()$<br>$\}$ |

Table 3. Behavior of a *Gossip System* constituted of two nodes ($node_x$ and $node_y$).

- $Inst = \{node_x : Node, node_y : Node\}$;
- $Con = \{(node_x, node_y)\}$.

According to Definition 14, the behavior of *Gossip System* is expressed as $B_{GossipSys} = \{B_{node_x}, B_{node_y}\}$, where:

$$B(GossipSys) = B(node_x)_{/node_x \rightarrow node_y} \cup B(node_y)_{/node_y \rightarrow node_x},$$
$$B(node_x)_{/node_x \rightarrow node_y} = \{r_1^x, r_2^x\},$$
$$B(node_y)_{/node_y \rightarrow node_x} = \{r_3^y, r_4^y\}.$$

| Attributes | Explanation |
|---|---|
| $view(IP, age)$ | is a list with IP address and age that represents the freshness of the given node descriptor. |
| $c$ | is the size of the view. |
| $push$ | equals 0 if the view is empty. |
| $pull$ | equals 1 if a response of an already called output service is expected. |
| $S$ | is the number of items that were sent to a peer previously. If S is high, then the received items will have a higher probability to be included in the new view. |
| $H$ | defines how aggressive the protocol should be when it comes to removing links that potentially point to faulty nodes (dead links). The larger H, the sooner older items will be removed from views. |

Table 4. Component attributes

## 4 SYSTEM ANALYSIS

As described in Section 2.3.2, 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.

In this section, we show how the DDG of a system is constructed, and then we present, as an application of DFA, an algorithm to resolve the deadlock detection problem.

### 4.1 Data-Dependency Graph

The Data-Dependency Graph is extracted from the semantic equations of the system by connecting together the Rule-Dependency Graphs corresponding to each rule used in this system. The Rule-Dependency Graph of a rule $r$ describes internal and external dependency relations of input and output data, which are manipulated by the different services of $r$.

The internal relations are induced from the semantic equation of a given rule, which define the computation of the output data in terms of the input data. Thus, Definition 15 defines the internal dependency relation as follows:

**Definition 15** (Internal Dependency Relation)**.** The internal dependency relation $G_{int}(r)$ in $ID_r \times OD_r$ of a rule $r$ is defined as follows:

$$a_p \; G_{int}(r) \; a_q \text{ if and only if } (\ldots, a_q, \ldots) = F(\ldots, a_p, \ldots). \tag{11}$$

Thus, $a_q$ **depends on** $a_p$, if $a_p$ is an argument in the semantic equation for $a_q$.

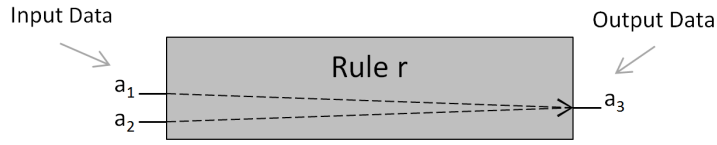


Figure 1. Example of an internal dependency relation

Figure 1 shows an example of an internal dependency relation where output data $a_3$ depends on input data $a_1$ and $a_2$.

The external relations of a rule $r$ are related to the source and destinations of events present in $r$. Therefore, we present the definition of external dependency relation as follows:

**Definition 16** (External Dependency Relation)**.** Let a rule $r^e$ describe the behavior of a component instance $e$ when it receives an input event, and let $(v(a_1^e, \ldots, a_q^e), e')$ be an event in $r^e$. The external relations induced from the event $(v(a_1^e, \ldots, a_q^e), e')$ depend on the position of this event in $r^e$:

$$\text{if } (v(a_1^e, \ldots, a_q^e), e') \in r^e.L \text{ then } \forall k \in 1..q, \quad a_k^{e'} \; G_{ext}(r) a_k^e, \tag{12}$$

$$\text{if } (v(a_1^e, \ldots, a_q^e), e') \in r^e.R \text{ then } \forall k \in 1..q, \quad a_k^e \; G_{ext}(r) a_k^{e'}. \tag{13}$$

Thus, $a_k^e$ **depends on** $a_k^{e'}$, if $a_k^e$ is an argument in the input event received from $e'$. And $a_k^{e'}$ **depends on** $a_k^e$, if $a_k^{e'}$ is an argument in the output event emitted to $e'$.

When no confusion arises between the notions of relation and graph, we shall represent them both by the same notation. Accordingly, we denote the Internal Dependency Graph of a rule $G_{int}(r)$ and the External Dependency Graph $G_{ext}(r)$. The union of these two graphs represents the Rule-Dependency Graphs of $r$, which we denoted by $G(r)$.

The Data-Dependency Graph, the graph of the whole system, is obtained from the union of the Rule-Dependency Graphs and it is defined as follows:

**Definition 17** (Data-Dependency Graph)**.** Let $Sys = \langle Inst, Con \rangle$ be a system, the Data-Dependency Graph of the system $Sys$ is:

$$G(Sys) = \bigcup_{e \in Inst} \left( \bigcup_{r \in B(e)} \left( G_{int}(r) \cup G_{ext}(r) \right) \right).$$

### 4.2 Detection of Deadlocks in a Component Composition

In a component composition, services are often forced to wait for resources from other services to finish execution. If the resources are not available, then the system may enter an infinite wait state. Under the assumption that this issue is not

caused by infinite loops, infinite wait is always caused by deadlocks or starvations. A deadlock is a situation in which two or more actions (services) are mutually waiting for each other to finish, while a starvation is a situation in which an action is perpetually denied access to resources needed to make progress.

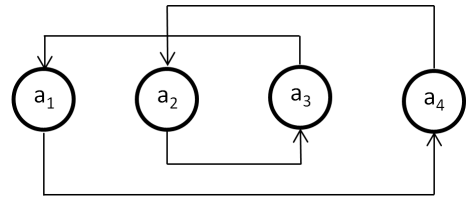


Figure 2. Example of data which depend on themselves

A system deadlock can be viewed as a circular dependency between data exchanged through services. Therefore, the basis of our deadlock analysis is detecting possible circular dependencies in the Data-Dependency Graph of the system.

Once the DDG is defined, we can induce if the system is deadlocked or not by searching for circularity in the graph. In other words, we shall search for a datum which depends on itself. An example of such as situation is given in Figure 2.

Formally, a deadlocked system is defined as follows:

**Definition 18** (Deadlocked system)**.** Let $Sys = \langle Inst, Con \rangle$ be a system and $G(Sys) = \cup_{e \in Inst}(\cup_{r \in B(e)}(G(r)))$ be the Data-Dependency Graph of $Sys$. Then $Sys$ is said to be deadlocked if and only if there exists a rule $r \in B(e)$, $e \in Inst$ such that $G(r)$ contains a cycle.

### 4.2.1 Deadlock Test

Now, we present an algorithm (Algorithm 2) which determines whether or not a system is deadlocked. The first stage of our deadlock test algorithm is to construct the Rule-Dependency Graph $G(r)$ of each rule $r$ in the behavior of each component in the system. This construction is achieved by connecting together the internal and external dependency graph of $r$ obtained as described above.

After that, $G(r)$ is added to the Data-Dependency Graph $G(Sys)$ which is initially empty. Once all rule graphs are added to $G(Sys)$, we compute transitive closure of $G(Sys)$, which we denoted by $G(Sys)^+$, in order to add induced dependencies. Those induced dependencies allow us to determine whether or not a node (a datum) of the graph is circular. If this is true, then we deduce that the system has a deadlock and a message with the rule that contains the circular data is printed.

**Algorithm 2** Deadlock test

**Require:** $Sys = \langle Inst, Con\rangle; G(Sys) := \emptyset$;
{ - - - - - - - - - - Construction of the system graph - - - - - - - - - -}
**for all** $e \in Inst$ **do**
**for all** $r \in B(e)$ such that $r : (v_0, e_0) \rightarrow (v_1, e_1), \ldots, (v_n, e_n)$ **do**
$G(r) := G_{int}(r) \cup G_{ext}(r)$;
$G(Sys) := G(Sys) \cup G(r)$;
**end for**
**end for**
{ - - - - - - - - - - - - - - - Search for deadlocks - - - - - - - - - - - - - - -}
$G(Sys) := G(Sys)^+$;
**for all** $e \in E$ **do**
**for all** $r \in B(e)$ such that $r : (v_0, e_0) \rightarrow (v_1, e_1), \ldots, (v_n, e_n)$ **do**
**if** $G(Sys)_{/r}$ contains a cycle **then**
**print** Deadlock detected in rule $r$;
**end if**
**end for**
**end for**

### 4.3 Dominance Analysis

Dominance analysis is a concept from graph theory and has many applications not only in the real world, but also in computer science. In compilers, dominance analysis is mostly used in code optimization and it is performed over flow graphs representing the execution of programs. One important task in this context is the optimization of loops since the execution of programs tends to spend most of their time in their inner loops. In parallel computing, dominance analysis is used to compute control dependences that identify those conditions affecting statement execution. Such information is critical for detection of parallelism [32]. In peer-to-peer applications, dominance analysis can be used to construct hierarchical overlay networks for more efficient index searching. It can also be used for optimizing routing among a set of nodes by reducing the searching space for a route to the dominating nodes in the set. Dominating nodes are a small set of nodes which are close to all other. Another field where dominance analysis is applied is memory usage analysis. In this field, the dominator tree (defined hereafter) is used to easily find memory leaks and identify high memory usage.

In a Data Dependency Graph, we say that node $d_i$ dominates node $d_j$, written *di dom dj*, if every path from the entry node of the graph to $d_j$ goes through $d_i$.

To make this dominance notion concrete, consider the Data Dependency Graph of Figure 3. Nodes $d_0$, $d_1$, $d_5$, and $d_8$ all lie on every path from $d_0$ to $d_8$, so $Dom(d_8)$ is $\{d_0, d_1, d_5, d_8\}$. The full sets of dominators for the graph are as follows:

$$
\begin{aligned}
Dom(d_0) &= \{d_0\},\\
Dom(d_1) &= \{d_0, d_1\},\\
Dom(d_2) &= \{d_0, d_1, d_2\},\\
Dom(d_3) &= \{d_0, d_1, d_3\},\\
Dom(d_4) &= \{d_0, d_1, d_3, d_4\},\\
Dom(d_5) &= \{d_0, d_1, d_5\},\\
Dom(d_6) &= \{d_0, d_1, d_5, d_6\},\\
Dom(d_7) &= \{d_0, d_1, d_5, d_7\},\\
Dom(d_8) &= \{d_0, d_1, d_5, d_8\}.
\end{aligned}
$$

A useful way of presenting dominance information is a dominator tree, in which each node $d$ dominates only its descendants [4, 33]. For example, Figure 4 shows the dominator tree for the DDG of Figure 3. Notice that $d_6$, $d_7$, and $d_8$ are all children of $d_5$, even though $d_7$ is not an immediate successor of $d_5$ in the DDG. In fact, each node $d_i$ in the tree has a unique immediate dominator $d_j$ that is the last dominator of $d_i$ in the DDG.

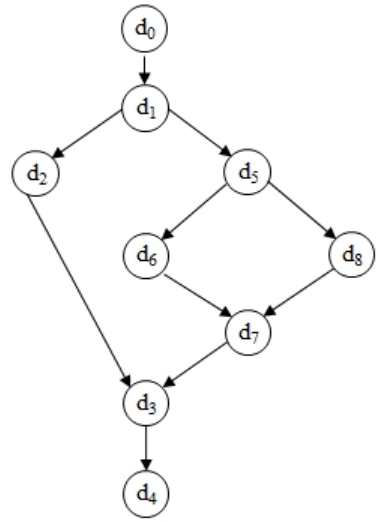


Figure 3. A data dependency graph

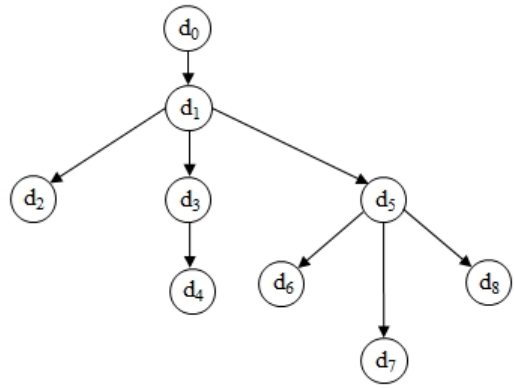


Figure 4. Dominator tree for the DDG of Figure 3

To compute dominance information in a DDG, we can formulate the problem as a set of data-flow equations and solve them with an iterative algorithm. This algorithm is based on the one proposed by Allen and Cocke [34] who relied on the principles of data-flow analysis to guarantee termination and correctness.

**Algorithm 3** Iterative Dominator Algorithm

**Require:** $G(Sys) := (N, E)$;

**for all** $n \in N$ **do**
  $Dom(n) = N$;
**end for**

$changed := True$;

**while** *changed* **do**
  $changed := False$;
  **for all** $n \in N$ **do**
    $temp = \{n\} \cup \left(\bigcap_{m \in preds(n)} Dom(m)\right)$;
    **if** $temp \neq Dom(n)$ **then**
      $Dom(n) = temp$;
      $changed := True$;
    **end if**
  **end for**
**end while**

Given a $DDG = (N, E)$, where $N$ is a set of nodes and $E$ is a set of directed edges, the following data-flow equations defines the Dom sets:

$$Dom(n) = \{n\} \cup \left( \bigcap_{m \in preds(n)} Dom(m) \right) \tag{14}$$

with the initial conditions: $Dom(n_0) = n_0$, and $\forall n \neq n_0$, $Dom(n) = N$. *preds* is a relation defined over $E$ that maps each node to its predecessors in the graph. Algorithm 3 shows an iterative solution for these dominance equations. It initializes the Dom set for each node, then repeatedly computes those sets until they stop changing.

## 5 RELATED WORK

The power of software system analysis approaches depends on the modeling technique for the behavior of software systems. This behavior is usually modeled by Finite-State Automata (e.g. [6]). However, it may also be modeled by process algebras (e.g. [35]), context-free languages (e.g. [36]), pushdown processes (e.g. [37]), etc.

In the context of the component-based system, the finite state approaches usually use regular languages to describe component behavior. However, these finite state approaches can only handle bounded recursion (i.e., up to a certain depth) and limited abstraction of the data-flow. To address this more explicitly, we discuss hereafter some of such approaches.

There is a large body of component models using various formal and semi-formal specifications in the context of component-based systems. These specifications concentrate on different aspects of component modeling. Due to this diversity, we refer to Rausch et al. [38] that provides an interesting study of state-of-the-art in component-based systems. Among the component models discussed in [38], KobrA [39] is a UML-based method for describing components and component-based systems. It uses different diagrams representing three projections: structural, functional and behavioral. KobrA is not a formal language, but rather a set of principles for using mainstream modeling language. It provides a certain degree of flexibility because anything that conforms to its principles can in practice be accommodated within the method. Rich Services [40] provides an architectural framework that reconciles the notion of service with hierarchy (systems-of-systems). It uses message sequence charts in order to specify component behaviors. This allows the approach to model bounded recursion. rCOS [41] is an extended theory of UTP (Unifying Theories of Programming) [42] for object-oriented and component-based programming. UTP combines the reasoning power of predicate calculus with the structuring power of relational algebra. In rCOS approach, each component interface has a contract. A contract only specifies the functional behavior in terms of predicates (pre and post conditions) and a protocol defining the acceptable traces of method calls. The behavior is specified by a state diagram and should

be accepted by an FSA. The protocol is specified by a sequence diagram. The reasons for having these two diagrams are different. In fact, the sequence diagram allows generating CSP processes to deal with concurrency, when the state diagram has an operational semantics which is easier to use for verification with model checking. SOFA [6] is a hierarchical component model. It is dedicated to the development of distributed application with dynamic update of components. It uses *behavior protocols* for the specification of interaction behavior of components. This allows to verify the system architecture independently from the implementation, and the relation of the component model and implementation. In order to fully automate behavior verification, a tool chain is used. It consists of *behavior protocols* to Promela translator and the Spin model checker. However, *behavior protocols* cannot treat behavior that cannot be specified by a regular language. Like SOFA, Fractal also uses *behavior protocols* to specify component behavior. Therefore, they have the same limitation on the description of component behavior.

Since the finite state models are not providing an adequate abstraction of a system that contains recursive call-backs, context-free model checking has been proposed. Among the first works in this direction, we could mention [36], which presents an algorithm that decides whether a property written in the alternation-free modal mu-calculus is satisfied for context-free processes, i.e., for processes that are given in terms of a context-free grammar or equivalently. Burkart et al. [37] propose pushdown processes as a generalization of context-free processes to better support parallel composition. Pushdown processes are processes that can be (finitely) represented by means of classical Pushdown Automata. After introducing these two approaches, several models for infinite-state systems have been proposed especially to decrease checking complexity [43, 44, 45]. But in the end, these models are still closely related to context-free processes and pushdown processes, and usually have the same expressiveness. In contrast to our approach, they cannot handle recursive call-backs which give rise to context-sensitive structures, e.g., interactive structures that adjust its behavior when the context changes (cf. Section 2.1).

Process algebras such as CSP (Communicating Sequential Processes) or CCS (Calculus of Communicating Systems) can be used as an alternative approach for verifying protocol conformance. These algebraic approaches are more powerful than FSA and context-free grammars. According to Milner [46], algebra appears to be a natural tool for expressing how systems are built. However, in order to automate analysis, some constraints on the specification language can be required. For instance, in [35], the authors are restricted to use their CSP notation in a way that processes will always be finite.

Compared to other works where component approach is dedicated to manipulate protocols, Reussner [47] presents the model of counter-constrained finite state machines. It is an extension of finite state machines, specifically created to model protocols containing dependencies between services due to their access to shared resources. However, Reussner does not consider composition operators and does

not provide an underlying discipline. Puntigam [48] shows that it is possible to develop component interfaces specifying non-regular protocols for the communication between components and the rest of a system. The concepts proposed in this paper need support from a programming language. However, no practically usable programming language supports these concepts.

Different data-flow based approaches have been proposed in the domain of system modeling. Garousi et al. [49] provide a control flow analysis for UML 2.0 sequence diagrams to define the control-flow. The authors propose an extended activity diagram meta-model. Yang et al. [50] present DFA-based algorithms to analyze BPEL programs and detect their data-flow anomalies. These algorithms operate on a control-flow graph derived from Activity Object Tree (AOT). The AOT is based on Eclipse Modeling Framework to express the relationships among activities. Zhou and Lee [51] propose a causality interface for deadlock analysis in a concurrent model of computation called *Dataflow*. It shows that deadlock is decidable for synchronous *Dataflow* models with a finite number of actors. Cain et al. [52] present an approach where a meta-model of an object oriented program's runtime is constructed to manage DFA. This meta-model contains classes that represent the relationship between the program elements (e.g., classes, objects and methods) in order to create an abstract representation for DFA. Like these different approaches, we also use DFA-based algorithms to analyze the constructed systems. However, our approach is dedicated to component-based P2P applications. It provides a formalism to capture their specific behavior (i.e. recursive call-backs) and constructs an abstract representation (i.e. Data-Dependency Graph) from which we can obtain multiple implementations of the control logic by analyzing the order of data evaluation.

The principle of the transformation of an abstract representation is also present in other formal systems. Many of them, such as $\lambda$-calculus [53], catamorphisms [54], hylomorphisms [55] and other from category theory, have been studied in previous works of Parigot (e.g. [56]) and a large comparison of these different formal systems can be found in [57]. These works show that those formal systems share a similar global structure. They abstract programs in some mathematical domain. Then, the transformed program is obtained by a backward translation from the mathematical domain. For instance, the HYLO system [55] transforms a program into hylomorphisms and then performs partial data evaluation. After that, these new hylomorphisms could be translated back into a program. However, these formal systems share a surprising constraint: the abstraction always relies on objects where recursive structures or schemes are strongly preserved and cannot be easily modified. For example, with $\lambda$-calculus, the recursive calls are altogether defined in the structure of the $\lambda$-terms. With hylomorphisms, these recursion schemes are exactly pointed out by functors (a special type of mapping in category theory) which are used as transformation parameters. Thus, transformations cannot freely restructure the abstract representation. Taking in mind these previous studies, DDF has then been defined with the following distinctive characteristics: i) allowing parts of the control logic (even if it is recursive) to be described conceptually separated

from other parts by using the concept of rules; ii) the user describes what is to be done rather than the details of how it is to be done; iii) from a single specification, multiple implementations can be synthesized by analyzing the order of data evaluation.

Other works relevant in the context of our approach can be found in database and network protocol communities. They are applied, for example, to cloud computing in order to raise the level of abstraction for programmers and improve program correctness in a data-centric, declarative style [58]. Another interesting approach is P2 [59]. It can be viewed as a synthesis of ideas from these two communities applied to overlay networks [60]. P2 is a system that uses a declarative logic language to express and implement overlay networks. It directly parses and executes such specifications into a data-flow program. The approach proposed by Lin et al. [61] seems to be close to our work in the sense that it also passes through the construction of a dependency graph to perform some optimizations. The difference between those works and our approach is that they are not based on components. This usually drives them to specify into their models (e.g., relational algebras and rule-based specification) the whole application, including business code.

## 6 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 [12] 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 [12], and optimized variants were presented in our previous works, e.g. [28, 29]. In a future work, we plan to extend our formalism by program transformation mechanisms in order to optimize resource allocations (e.g. optimize CPU and memory usage by analyzing the lifetime of data, while taking into account their functional dependencies and redundancies) in large-scale data-centric applications, in particular, in the emerging Cloud Computing area, where data management has been receiving significant attention. Another perspective field where this future work (i.e. optimization of resource allocations) might be useful is the Green Computing. In fact, environmental protection and energy-aware resource management have become popular and important research topics at present [62]. In this direction, the Green Computing is emerging as an indispensable part in sustaining the practice of protecting the environment on both individual and collective levels.

## REFERENCES


[1] Schollmeier, R.: A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. Proceedings of the First International Conference on Peer-to-Peer Computing (P2P '01), IEEE Computer Society, Washington, DC, USA, 2001.

[2] Gupta, A.—Awasthi, L.: Peer-to-Peer Networks and Computation: Current Trends and Future Perspectives. Computing and Informatics, Vol. 30, 2012, No. 3, pp. 559–594.

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

[4] Aho, A. V.—Sethi, R.—Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] Mernik, M.—Heering, J.—Sloane, A. M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, 2005, No. 4, pp. 316–344, doi: 10.1145/1118890.1118892.

[6] Bures, T.—Decky, M.—Hnetynka, P.—Kofron, J.—Parízek, P.—Plasil, F.—Poch, T.—Sery, O.—Tuma, P.: CoCoME in SOFA. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (Eds.): The Common Component Modeling Example. Springer Berlin, Heidelberg, LNCS, Vol. 5153, 2008, pp. 388–417.

[7] Bulej, L.—Bures, T.—Coupaye, T.—Decky, M.—Jezek, P.—Parízek, P.—Plasil, F.—Poch, T.—Rivierre, N.—Sery, O.—Tuma, P.: CoCoME in Fractal. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (Eds.): The Common Component Modeling Example. Springer Berlin, Heidelberg, LNCS, Vol. 5153, 2008, pp. 357–387.

[8] Ripeanu, M.—Foster, I.—Iamnitchi, A.: Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. IEEE Internet Computing Journal, Vol. 6, 2002, No. 1.

[9] Aberer, K.—Despotovic, Z.: Managing Trust in a Peer-2-Peer Information System. Proceedings of the Tenth International Conference on Information and Knowledge Management, ACM, New York, NY, USA, 2001, pp. 310–317, doi: 10.1145/502585.502638.

[10] Gschwind, T.—Assmann, U.—Nierstrasz, O.: Software Composition. 4th International Workshop (SC 2005), Edinburgh, UK. Springer, LNCS, Vol. 3628, 2005.

[11] Stanley, R.: Enumerative Combinatorics. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2001.

[12] Deransart, P.—Jourdan, M.—Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. Springer-Verlag, Inc., New York, NY, USA, 1988.

[13] Yang, J.—Papazoglou, M. P.: Service Components for Managing the Life-Cycle of Service Compositions. Information Systems, Vol. 29, 2004, No. 2, pp. 97–125.

[14] Posypkin, M.—Semenov, A.—Zaikin, O.: Using BOINC Desktop Grid to Solve Large Scale SAT Problems. Computer Science, Vol. 13, 2012, No. 1, pp. 25–34, doi: 10.7494/csci.2012.13.1.25.

[15] Abadi, D. J.: Data Management in the Cloud: Limitations and Opportunities. IEEE Data(base) Engineering Bulletin, Vol. 32, 2009, No. 1, pp. 3–12.

[16] Baier, C.—Katoen, J.-P.: Principles of Model Checking. The MIT Press, May 2008.

[17] Lee, J.—Lee, H.—Kang, S.—Kim, S. M.—Song, J.: CISS: An Efficient Object Clustering Framework for DHT-Based Peer-to-Peer Applications. Computer Networks, Vol. 51, 2007, No. 4, pp. 1072–1094.

[18] Kam, J. B.—Ullman, J. D.: Global Data Flow Analysis and Iterative Algorithms. Journal of the ACM (JACM), Vol. 23, 1976, pp. 158–171, doi: 10.1145/321921.321938.

[19] Govindarajan, R.—Yu, S.—Lakshmanan, V. S.: Attempting Guards in Parallel: A Data Flow Approach to Execute Generalized Guarded Commands. International Journal of Parallel Programming, Vol. 21, 1992, pp. 225–268, doi: 10.1007/bf01421675.

[20] Voulgaris, S.—Gavidia, D.—van Steen, M.: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Journal of Network and Systems Management, Vol. 13, 2005, pp. 197–217, doi: 10.1007/s10922-005-4441-x.

[21] Jelasity, M.—Voulgaris, S.—Guerraoui, R.—Kermarrec, A.-M.—van Steen, M.: Gossip-Based Peer Sampling. ACM Transactions on Computer Systems (TOCS), Vol. 25, 2007, No. 3, Art. No. 8.

[22] Jelasity, M.: Gossip. In: Di Marzo Serugendo, G., Gleizes, M.-P., Karageorgos, A. (Eds.): Self-Organising Software: From Natural to Artificial Adaptation. Chapter 1. Springer Berlin Heidelberg, Natural Computing Series, 2011, pp. 139–162, doi: 10.1007/978-3-642-17348-6_7.

[23] Ait Lahcen, A.—Parigot, D.—Mouline, S.: Defining and Analyzing P2P Applications with a Data-Dependency Formalism. 13th International Conference on Parallel and Distributed Computing, Applications and Technologies, 2012, pp. 317–322.

[24] Ait Lahcen, A.—Parigot, D.—Mouline, S.: Toward Data-Centric View on Service-Based Component Systems: Formalism, Analysis and Execution. Work in Progress Session of the 20th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, Garching, Germany, 2012.

[25] Knuth, D. E.: Semantics of Context-Free Languages. Mathematical Systems Theory, Vol. 2, 1968, No. 2, pp. 127–145, Correction: Mathematical Systems Theory, Vol. 5, 1971, No. 1, pp. 95–96.

[26] Knuth, D. E.: The Genesis of Attribute Grammars. WAGA Proceedings of the International Conference on Attribute Grammars and Their Applications, Springer-Verlag New York, Inc., New York, NY, USA, 1990, pp. 1–12, doi: 10.1007/3-540-53101-7_1.

[27] Aho, A. V.—Lam, M. S.—Sethi, R.—Ullman, J. D.: Compilers: Principles, Techniques, and Tools. 2nd ed. Prentice Hall, 2006.

[28] Jourdan, M.—Parigot, D.: Techniques for Improving Grammar Flow Analysis. Proceedings of the Third European Symposium on Programming, Springer-Verlag New York, Inc., New York, NY, USA, 1990, pp. 240–255, doi: 10.1007/3-540-52592-0_67.

[29] Parigot, D.—Roussel, G.—Jourdan, M.—Duris, E.: Dynamic Attribute Grammars. International Symposium on Programming Languages, Implementations and Logic Programming (PLILP '96). Springer, Lecture Notes in Computer Science, Vol. 1140, 1996, pp. 122–136, doi: 10.1007/3-540-61756-6_81.

[30] Silberschatz, A.—Galvin, P. B.—Gagne, G.: Operating System Concepts. 8th ed. Wiley Publishing, 2008.

[31] Börger, E.: Architecture Design and Validation Methods. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000, doi: 10.1007/978-3-642-57199-2.

[32] Srinivasan, H.—Wolfe, M.: Analyzing Programs with Explicit Parallelism. Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Springer, London, UK, 1992, pp. 405–419, doi: 10.1007/bfb0038679.

[33] Cooper, K.—Torczon, L.: Engineering a Compiler. Elsevier Science, 2011.

[34] Allen, F. E.—Cocke, J.: Graph Theoretic Constructs for Program Control Flow Analysis. IBM T. J. Watson Research Center, NY, Technical Report, 1972.

[35] Allen, R.—Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 6, 1997, pp. 213–249, doi: 10.1145/258077.258078.

[36] Burkart, O.—Steffen, B.: Model Checking for Context-Free Processes. In: Cleaveland, W. (Ed.): CONCUR '92. Springer Berlin, LNCS, Vol. 630, 1992, pp. 123–137.

[37] Burkart, O.—Steffen, B.: Pushdown Processes: Parallel Composition and Model Checking. In: Jonsson, B., Parrow, J. (Eds.): CONCUR '94. Springer Berlin, Heidelberg, LNCS, Vol. 836, 1994, pp. 98–113.

[38] Rausch, A.—Reussner, R.—Mirandola, R.—Plasil, F.: The Common Component Modeling Example: Comparing Software Component Models. 1st ed. Springer, 2008.

[39] Atkinson, C.—Bostan, P.—Brenner, D.—Falcone, G.—Gutheil, M.—Hummel, O.—Juhasz, M.—Stoll, D.: Modeling Components and Component-Based Systems in Kobra. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (Eds.): The Common Component Modeling Example. Springer Berlin, Heidelberg, LNCS, Vol. 5153, 2008, pp. 54–84.

[40] Demchak, B.—Ermagan, V.—Farcas, E.—Huang, T.-J.—Kruger, I.—Menarini, M.: A Rich Services Approach to CoCoME. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (Eds.): The Common Component Modeling Example. Springer Berlin, Heidelberg, LNCS, Vol. 5153, 2008, pp. 85–115.

[41] Chen, Z.—Hannousse, A.—Van, D.—Hung, Knoll, I.—Li, X.—Liu, Z.—Liu, Y.—Nan, Q.—, Okika, J.—Ravn, A.—Stolz, V.—Yang, L.—Zhan, N.: Modelling with Relational Calculus of Object and Component Systems – rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (Eds.): The Common Component Modeling Example. Springer Berlin, Heidelberg, LNCS, Vol. 5153, 2008, pp. 116–145.

[42] Hoare, C.—Jifeng, H.: Unifying Theories of Programming. Prentice Hall Series in Computer Science, Prentice Hall, 1998.

[43] Alur, R.—Benedikt, M.—Etessami, K.—Godefroid, P.—Reps, T.—Yannakakis, M.: Analysis of Recursive State Machines. ACM Transactions on

Programming Languages and Systems (TOPLAS), Vol. 27, 2005, pp. 786–818, doi: 10.1145/1075382.1075387.

[44] Benedikt, M.—Godefroid, P.—Reps, T. W.: Model Checking of Unrestricted Hierarchical State Machines. Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP '01), Springer-Verlag, London, UK, 2001, pp. 652–666, doi: 10.1007/3-540-48224-5_54.

[45] Esparza, J.—Hansel, D.—Rossmanith, P.—Schwoon, S.: Efficient Algorithms for Model Checking Pushdown Systems. In: Emerson, E., Sistla, A. (Eds.): Computer Aided Verification. Springer Berlin, LNCS, Vol. 1855, 2000, pp. 232–247.

[46] Milner, R.: A Calculus of Communicating Systems. Springer-Verlag, LNCS, Vol. 92, 1980, doi: 10.1007/3-540-10235-3.

[47] Reussner, R.: Counter-Constrained Finite State Machines: A New Model for Component Protocols with Resource-Dependencies. In: Grosky, W., Plasil, F. (Eds.): SOFSEM 2002: Theory and Practice of Informatics. Springer Berlin, Heidelberg, LNCS, Vol. 2540, 2002, pp. 20–40, doi: 10.1007/3-540-36137-5_2.

[48] Puntigam, F.: State Information in Statically Checked Interfaces. In Eighth International Workshop on Component-Oriented Programming, Darmstadt, Germany, July 2003.

[49] Garousi, V.—Briand, L.—Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams. In: Hartman, A., Kreische, D. (Eds.): Model Driven Architecture – Foundations and Applications. Springer Berlin, Heidelberg, LNCS, Vol. 3748, 2005, pp. 160–174.

[50] Yang, X.—Huang, J.—Gong, Y.: Static Data Flow Analysis and Anomalies Detection for BPEL. International Conference on Test and Measurement (ICTM '09), Vol. 2, 2009, pp. 18–21.

[51] Zhou, Y.—Lee, E. A.: A Causality Interface for Deadlock Analysis in Dataflow. Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT '06), ACM, New York, NY, USA, 2006, pp. 44–52.

[52] Cain, A.—Chen, T. Y.—Grant, D. D.—Kuo, F.-C.—Schneider, J.-G.: An Object Oriented Approach Towards Dynamic Data Flow Analysis (Short Paper). Proceedings of the 2008 The Eighth International Conference on Quality Software (QSIC '08), IEEE Computer Society, Washington, DC, USA, 2008, pp. 163–168, doi: 10.1109/QSIC.2008.18.

[53] Sheard, T.: A Type-Directed, On-Line, Partial Evaluator for a Polymorphic Language. Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), ACM, New York, NY, USA, 1997, pp. 22–35, doi: 10.1145/258993.258999.

[54] Launchbury, J.—Sheard, T.: Warm Fusion: Deriving Build-Cata's from Recursive Definitions. Proceedings of Programming Languages and Computer Architecture (FPCA '95), ACM Press, 1995, pp. 314–323.

[55] Onoue, Y.—Hu, Z.—Takeichi, M.—Iwasaki, H.: A Calculational Fusion System Hylo. Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi, Chapman & Hall, Ltd., London, UK, 1997, pp. 76–106.

[56] Correnson, L.—Duris, E.—Parigot, D.—Roussel, G.: Equational Semantics. Proceedings of the 6th International Symposium on Static Analysis (SAS'99), Springer-Verlag, London, UK, 1999, pp. 264–283, doi: 10.1007/3-540-48294-6_17.

[57] Duris, É.: Contribution aux Relations Entre les Grammaires Attribuées et la Programmation Fonctionnelle. Ph.D. Thesis, Université de Marne la Vallée, October 1998 (in French).

[58] Alvaro, P.—Condie, T.—Conway, N.—Elmeleegy, K.—Hellerstein, J. M.—Sears, R.: Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. Proceedings of the 5th European Conference on Computer Systems (EuroSys'10), ACM, New York, NY, USA, 2010, pp. 223–236, doi: 10.1145/1755913.1755937.

[59] Loo, B. T.—Condie, T.—Hellerstein, J. M.—Maniatis, P.—Roscoe, T.—Stoica, I.: Implementing Declarative Overlays. SOSP, ACM, 2005, pp. 75–90.

[60] Andersen, D.—Balakrishnan, H.—Kaashoek, F.—Morris, R.: Resilient Overlay Networks. ACM SIGOPS Operating Systems Review, Vol. 35, 2001, No. 5, pp. 131–145, doi: 10.1145/502034.502048.

[61] Lin, S.—Taïani, F.—Bertier, M.—Blair, G.—Kermarrec, A.-M.: Transparent Componentisation: High-Level (Re)Configurable Programming for Evolving Distributed Systems. Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11), ACM, New York, NY, USA, 2011, pp. 203–208, doi: 10.1145/1982185.1982233.

[62] Hu, J.—Deng, J.—Wu, J.: A Green Private Cloud Architecture with Global Collaboration. Telecommunication Systems, Vol. 52, 2013, No. 2, pp. 1269–1279.

**Ayoub Ait Lahcen** is Assistant Professor of computer engineering at the ENSA Kenitra, a Moroccan engineering school, and a researcher at both LGS laboratory (ENSA, Ibn Tofail University) and LRIT laboratory (Mohammed V University, Morocco). Prior to that, he got a Swiss Government Excellence Scholarship to work during an academic year as a postdoctoral researcher, with the Software Engineering Group of the University of Fribourg, Switzerland. He received his Ph.D. degree in computer science from both Nice Sophia Antipolis University, France (prepared at Inria Sophia Antipolis, in the Zenith team) and Mohammed V University (prepared at LRIT laboratory). He was awarded a Moroccan Research Excellence Scholarship for Ph.D. candidates and a Merit Scholarship for his Master in computer science and telecommunications.

**Didier Parigot** is senior research scientist at Inria. He brings more than 30 years of research works in generative software engineering, generative programming, middleware and data driven programming. Since 2000, he leads the SON software at Inria Sophia Antipolis. He holds his Ph.D. in computer science from Paris XI University and H.D.R. (equivalence with the Professor status) from Nice University.

**Salma Mouline** is Professor at Faculty of Sciences, Mohammed V University of Rabat, Morocco. She obtained her Ph.D. in computer sciences from Université Pierre Mendès France, Grenoble, France in 1994. Her research interests include software engineering, information systems, verification of complex systems and formal verification.