UNDER PEER REVIEW

Information Flow in Concurrent Logic Programming

Original research paper

Abstract

This paper presents a new formalization of information flow detection in concurrent logic programming and applies it to the problem of deadlock detection. This work is based on a recent study of the detection of information flow in Datalog programs. Firstly, we define the concept of information flow in concurrent logic programming. Then, we propose a set of definitions of flow based on observation and transition systems while solving goals. Finally, we formalize a mechanism for deadlock detection in concurrent logic programs.

Keywords: Information flow; Concurrent logic programming; Transition systems; Observation; Deadlock detection;

1 Introduction

Keeping information secret was one of the preoccupation of nearly every human being. Benjamin Franklin noted in "Poor Richards Almanac" published in 1734, that "Three persons can keep secret, if two of them are dead" [9]. In computer science as well as in current life, a program is called secure if it is able to keep "*information*" confidential. In order to illustrate the crucial idea of Franklin, consider the following commands sequence in imperative programming $tmp := x; y := tmp;$. It is clear that, after the execution of these two commands, the initial value of the x variable is revealed by the value of the variable y or tmp. Thus, the x secret, shared with y and tmp, is known at the end of the execution of the sequence. However, after the execution of the following sequence $tmp = x; tmp = 0;$ the initial value of the variable x is not revealed since the value of the tmp variable was crashed or "*dead*".

This principle led to several studies of information flow in imperative programming [2, 3, 5, 6]. Most of the work done in this field addressed the framework of program optimization and aimed to improve the quality of programs and to detect eventual errors. Bergeretti *et al.* [4] address information flow relations for while-programs and use it to detect errors in the static analysis of a program. Smith

UNDER PEER REVIEW

et al. [17] develop a type system to ensure secure information flow in a multi-threaded language. Sabelfeld *et al.* [14, 15] propose an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations. In the domain of logic programming, Debray [8] elaborates an algorithm to analyze sequential logic programs and extends it to handle parallel executions.

The work presented in this paper is based on a recent study done by Yaacoub *et al.* [1, 19], who proposed three definitions of information flows in logic programs. These definitions correspond to what can be observed by the user when a query $\leftarrow G(x, y)$ is run on a logic program P. The first definition considers that the user only sees whether his query succeeds or fails; thus, information flows exists from x to y in G when there exists constants a and b such that $P \cup \{\leftarrow G(a, y)\}$ succeeds whereas $P \cup \{\leftarrow G(b, y)\}$ fails. The second definition is based on the sets of substitution answers computed by the interpreter with respect to his queries. There is a flow of information from x to y in G if there are constants a and b such that the set of substitution answers of $P \cup \{\leftarrow G(a, y)\}$ and $P \cup \{\leftarrow G(b, y)\}$ are different. Lastly, the third definition considers the existence of information flow if the SLD-refutation trees of the queries $P \cup \{\leftarrow G(a, y)\}$ and $P \cup \{\leftarrow G(b, y)\}$ can be distinguished (non bisimilar).

Since most of the programs in the real world are interactive in the sense that they send and receive permanently data by interacting with their environment (web server, GUI applications ...), it is tempting to address the question of information flow detection in such programs. In fact, the information flow detection conditions proposed in imperative and first order logic programming are unable to capture the same notion of information flow in concurrent programming. This is due to the fact that data dependency is based on variable matching and not on assignation as in imperative programming nor on substitutions as in first order logic programming. Due to the prevalence of interactive programs, it is important to see what could be an information flow in concurrent logic programming.

In this paper, we propose several definitions of information flow in concurrent logic programming. These definitions correspond to what can be "*observed*" by the user when a query $\leftarrow G(x, y)$ is run on a concurrent logic program P. An observation is made on the set of succeeded goals, failed goals, blocking states and infinite states. We propose definitions based on the set of transitions in concurrent logic programming to finally elaborate a final set of definitions based on the number of transitions while resolving a goal.

In section 2 of this paper, we present some basic notions about logic and concurrent logic programming. Then, several definitions of information flow in concurrent logic programming are proposed relatively to a logic program P and a goal $\leftarrow G(x, y)$ of arity 2, (which stipulates the existence of a flow from the x variable to the y variable in the goal $\leftarrow G(x, y)$). The implications between these definitions are then studied. Finally, we show the utility of these definitions by studying the deadlock detection problem in concurrent logic programming.

2 Framework

2.1 Syntax and Semantics

The language L considered here is essentially that of first order predicate logic [11]. It has countable sets of variables and predicate symbols, these sets being mutually disjoint. A term is a variable or a constant. A term is ground if no variable occurs in it. The Herbrand universe of L, denoted U_L, is the set of all ground terms that can be formed with the constants in L. An atom is

UNDER PEER REVIEW

of the form $p(t_1, \cdots, t_n)$, where p is an n-ary predicate symbol and the t_i are terms, $1 \leq i \leq n$. An atom is ground if all t_i are ground. A clause is an expression of the form $A \leftarrow B_1, \cdots, B_n$ where $A, B_1, \cdots, B_n$ are atoms. A is called the `head` of the clause and $B_1, \cdots, B_n$ is called its `body`. A goal is an expression of the form $\leftarrow B_1, \cdots, B_n$. A clause r of the form $A \leftarrow$ (i.e., whose body is empty) is called a `fact`, and if A is a ground atom, then r is called a `ground fact`. The empty goal is denoted $\square$. A predicate definition is assumed to consist of a finite set (possibly ordered) of clauses defining the same predicate. A logic program consists of a finite set of predicate definitions. With each logic program P, we associate the language $L(P)$ that consists of the predicates, functions, and constants occurring in P. If no constant occurs in P, we add some constant to $L(P)$ to have a nonempty domain. A substitution is an idempotent mapping from a finite set of variables to terms. The identity substitution will be denoted ϵ. A substitution σ_1 is said to be more general than a substitution σ_2 if there is a substitution θ such that $\sigma_2 = \theta\sigma_1$. Two terms t_1 and t_2 are said to be unifiable if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$, in this case σ is said to be a unifier for the terms. If two terms t_1 and t_2 have a unifier, then they have a most general unifier $mgu(t_1, t_2)$ that is unique up to variable renaming.

The operational behavior of logic programs can be described by means of SLD-derivations. An SLD-derivation for a goal $G = \leftarrow A_1, \cdots, A_n$ with respect to a program P is a sequence of goals $G_0, \cdots, G_i, G_{i+1}, \cdots$, such that $G_0 = G$, and if $G_i = B_1, \cdots, B_m$, then $G_{i+1} = \theta B_1, \cdots, \theta B_{i-1}, \theta B_1^{'}, \cdots, \theta B_k^{'}, \theta B_{i+1}, \cdots, \theta B_m$ such that $1 \leq i \leq m$, and $B \leftarrow B_1^{'}, \cdots, B_k^{'}$ is a variant of a clause in P that has no variable in common with any of the goals $G_0, \cdots, G_i$, and $\theta = mgu(B_i, B)$. The goal G_{i+1} is said to be obtained from G_i by means of resolution step, and B_i is said to be the resolved atom. Let $G_0, \cdots, G_n$ be an SLD-derivation for a goal G with respect to a program P, and let θ_i be the unifier obtained when resolving the goal G_{i-1} to obtain $G_i, 1 \leq i \leq n$. If this derivation is finite and maximal, i.e., one in which it is not possible to resolve the goal G_n with any of the clauses in P, then it corresponds to a terminating computation for G: in this case, if G_n is the empty goal then we say that $P \cup \{G\}$ succeeds and the computation is said to succeed with answer substitution θ, where θ is the substitution obtained by restricting the substitution $\theta_n \cdots \theta_1$ to the variables occurring in G. if G_n is not the empty goal, then the computation is said to fail. We say that $P \cup \{G\}$ fails if all computations from G in P fail. If the derivation is infinite, the computation does not terminate.

2.2 Transition systems for First order Logic Programming

Definition 2.1. (Transition Systems for logic programs) We associate for each logic program P a transition system composed of:

- A set of states: A state is a couple of the form $< G; \theta >$, where G is the goal (possibly a failure goal), and θ is a substitution. The initial state is denoted by $< G; \epsilon >$ where G is the initial goal and ϵ denotes the absence of subtitutions.
- A set of transitions: A transition is a function from a state to a set of states. For the states S and S', and the transition t, we denote the possibility to pass from state S to state S' using the transition t by $S \xrightarrow{t} S'$

There is two types of transitions:

1. Reduce : $< A_1, \cdots, A_{i-1}, A_i, A_{i+1}, \cdots, A_n; \theta > \xrightarrow{Reduce} < (A_1, \cdots, A_{i-1}, B_1, \cdots, B_k, A_{i+1}, \cdots, A_n)\theta'; \theta \circ \theta' >$ if $mgu(A_i, A) = \theta'$ for some renamed clauses $A \leftarrow B_1, \cdots, B_k$ of P.
2. Fail : $< A_1, \cdots, A_{i-1}, A_i, A_{i+1}, \cdots, A_n; \theta > \xrightarrow{Fail} < fail; \theta >$ for some renamed clauses $A \leftarrow B_1, \cdots, B_k$ of P and $mgu(A_i, A) = fail$.

Example 2.1. *Let P be the following program :*
$ListenToMusic(X) \leftarrow ListenToMusic(laura).$

UNDER PEER REVIEW

$ListenToMusic(laura).$
and let $G(X)$ *be the following goal :* $\leftarrow ListenToMusic(X).$
Then the transition system of $G(X)$ *is :*
$< G(X); \epsilon > \xrightarrow{Reduce} < G(laura); \{X \mapsto laura\} > \xrightarrow{Reduce} < \Box; \{X \mapsto laura\} >$

Definition 2.2. (Activated state, terminal state, success state, and failure state)

- a transition t is `activated` on a state S if $t(S)$ is not empty.
- A state on which no transition is activated is call `terminal state`. $< true; \theta >$, $< fail; \theta >$ are called respectively success and failure states.

Definition 2.3. The calculus of a a program *P* on a goal *G* is a (finite or infinite) set of states $\mathcal{C} = \{S_1, S_2, \cdots\}$ that satisfies:

- Initiation: $S_1 =< G; \epsilon >$, where ϵ is the empty substitution.
- Consecution : for each $k, S_{k+1} \in t(S_k)$ for some transition.
- Termination $\mathcal{C}$ is finite of length k iff S_k is terminal.

2.3 Concurrent Logic Programming

Advanced research in recent years and the increasing availability of computers led to a new style of programming called concurrent programming that allows multiple calculations to occur simultaneously and in cooperation with each other. Most people distinguish between two categories of concurrent programming: distributed programming which refers to calculations that do not share a common memory, and parallel programming which refers to calculations that share a common memory. We are interested in the concurrent logic programming, which is characterized by a very different calculation process from that of the classical first order logic programming. Several studies addressed languages of concurrent logic programming such as FCP(|) [16], FCP(:,?) [20], Parlog [7], $\cdots$

2.3.1 Syntax

In concurrent logic programming [10], each atom $p(T_1, \cdots, T_n)$ is seen as a process and the goal (set of atoms) as a network of processes that communicate by instantiating shared logical variables. The possible behaviors of a process are specified by the "Guarded Horn Clauses" [18] that have the following form: $Head \leftarrow Guard|Body$
The $Head$ and the $Guard$ specify the conditions under which the transition `Reduce` could use the clause. The $Body$ specifies the state of the process after taking the transition. A process may stop, change its state or became multiple processes as shown in the following table:

Operation	Syntax
Stop	$A \leftarrow G\|true$
Change its state	$A \leftarrow G\|B$
Became k concurrent processes	$A \leftarrow G\|B_1, \cdots, B_k$

2.3.2 Semantics

Among the differences between the semantics of classical first-order logic and concurrent logic, we mention the following [12]:

UNDER PEER REVIEW

Non-determinism There are two types of non-determinism:

1. `Don't Know non-determinism`: If someone asks a class of students to write all their names on a piece of paper, it means that he knows that he want all the names but he does not have to know the order in which they are written. Having said that all the alternatives are as good and lead to the same result.
2. `Don't Care Non-determinism`: If someone asks a class of students the following question: "One of you must clear the board before the course starts" so this person wants only that one of the students to do it but he does not care about the identity of the student. That said we do not know what is the right choice to take among the offered alternatives. All possible choices should be exploited by exploring the different alternatives using search tree.

Don't Know non-determinism is used in logic programming and particularly in Prolog, while Don't Care non-determinism is used by the "Committed choice logic language" as Parlog or FCP(|).
Committed choice is related to the idea of "guard" and the "commit" operator (that is the equivalent of the "cut" in Prolog - once selected, there is no backtracking). In concurrent systems, a process cannot "undo" a choice, even if it turns out to be wrong. The reason is that the choices and actions carried out subsequently, have already influenced the environment.

In concurrent logic programming, it is question about matching and not about unification between atoms as in the first order logic programming. In fact, the matching of an atom A of a goal with the head of the clause $A' \leftarrow G|B$ succeeds if A is an instance of A'. In this case, the matching returns the most general unifier θ such that $A = A'\theta$. The matching fails if A and A' are not unifiable. Otherwise, the matching is suspended:

$$\texttt{match}(A, A') = \begin{cases} \theta & \text{if } \theta \text{ is a mgu such that } A = A'\theta \\ fail & \text{if } mgu(A, A') = fail \\ suspend & \text{else} \end{cases}$$

Example 2.2. *: Examples of matching a goal with the head of the clause.*

Goal	***Head of the clause***	***Result***
$\leftarrow P(a)$	$P(X)$	$\{X \mapsto a\}$
$\leftarrow P(X)$	$P(a)$	*Suspend*
$\leftarrow P(a)$	$P(b)$	*Fail*

Transition systems The single difference between transition systems of first order logic programming and that of concurrent logic programming is that, in the latter, transitions employ matching and guards verification instead of unification [13]. The process of matching and guards verification is captured by the "`try`" function.
The `mgu` function unifies the goal with the head of the clause and returns a substitution or a failure. The `try` function pairs the goal with the head of the clause. In the event that it is successful, we say that it satisfies the guard. The function can also return suspension if the matching or checking guards are suspended.
`try` is defined as follows :
$\texttt{try}(T_1 = T_2, X = X) = \texttt{mgu}(T_1, T_2)$.

$$\texttt{try}(A, (A' \leftarrow G|B)) = \begin{cases} \theta & \text{if } (match(A, A') = \theta \wedge \text{ verify } G\theta \text{ succeed}) \\ fail & \text{if } (mgu(A, A') = \theta \wedge \text{ verify } G\theta \text{ fails}) \vee (mgu(A, A') = fail) \\ suspend & \text{else} \end{cases}$$

Thus, transitions `fail` and `reduce` use the `try` function instead of `mgu`:

- `Reduce`: : $< A_1, \cdots, A_{i-1}, A_i, A_{i+1}, \cdots, A_n; \theta > \xrightarrow{Reduce} < (A_1, \cdots, A_{i-1}, B_1, \cdots, B_k, A_{i+1}, \cdots, A_n)\theta'; \theta \circ \theta' >$ if $try(A_i, C) = \theta'$ for some renamed clauses $A \leftarrow G|B_1, \cdots, B_k$ of P.

UNDER PEER REVIEW

- `Fail` : $< A_1, \cdots, A_{i-1}, A_i, A_{i+1}, \cdots, A_n; \theta > \xrightarrow{Fail} < fail; \theta >$ for some renamed clauses (other than C) of P and $try(A_i, C) = fail$.

Note that the result of the suspension is not used in both transitions. Its effect is to prevent an atom in the goal to be reduced to a failure clause.
The notation ($\xrightarrow{*}$) is used to denote the transitive closure of $\rightarrow$, and $\not\rightarrow$ to indicate the absence of any other transition.

Observations Having a program P, we define by :

- $O_{ss}(P) = \{< G; \epsilon > \xrightarrow{*} < \Box; \theta >\}$, the set of goals G that succeed.
- $O_{ff}(P) = \{< G; \epsilon > \xrightarrow{*} < G'; \theta > \not\rightarrow \neq \Box\}$, the set of goals G that fail.
- $O_{dd}(P) = \{< G; \epsilon > \xrightarrow{*} < G_n; \theta >\}$, $G_n \neq \Box$, the set of goals G that lead to a blocking state.
- $O_{ii}(P) = \{< G; \epsilon > \rightarrow < G_1; \theta_1 > \rightarrow \cdots < G_n; \theta_n > \rightarrow \cdots\}$, the set of goals G that lead to an infinite state.

Example 2.3. *:*
Let P be the following program:
$C_1 : p(b) \leftarrow q.$
$C_2 : p(a).$
$C_3 : q(X) \leftarrow q(b).$
$C_4 : r(X) \leftarrow r(X).$
Let G_1, G_2, G_3, and G_4 be the following goals respectively: $\leftarrow p(a)$, $\leftarrow p(b)$, $\leftarrow q(X)$, and $\leftarrow r(X)$.

Thus,
$O_{ss}(P) = \{< G_1; \epsilon > \rightarrow < \Box; \theta >\}.$
$O_{ff}(P) = \{< G_2; \epsilon > \rightarrow < q; \epsilon > \not\rightarrow\}.$
$O_{dd}(P) = \{< G_3; \epsilon > \rightarrow < q(b); \theta >\}.$
$O_{ii}(P) = \{< G_4; \epsilon > \rightarrow < G_4; \theta > \rightarrow < G_4; \theta > \rightarrow \cdots\}.$

3 Formulation of information flow

As we mentioned earlier, we proposed three definitions of information flow in logic programs. We complete these definitions by introducing new ones adapted to concurrent logic programming. Recall that there is an information flow from x to y in program P and a goal $G(x, y)$ when there exists constants a and b such that the outputs of $P \cup \{\leftarrow G(a, y)\}$ and $P \cup \{\leftarrow G(b, y)\}$ computations are distinguishable by the user without seeing what concerns a and b.

3.1 Observation-based information flow

Having a program P, and a goal $G(x, y)$, there is a flow of information from x to y in $G(x, y)$ (denoted by $x \rightarrow_G^P y$) iff $\exists a, b \in U_L$ such that:

Definition 3.1. $G(a, y) \in O_{ff}(P)$ and $G(b, y) \notin O_{ff}(P)$.

Example 3.1. *Let P_1 be the following program:*
$C_1 : eat(bob, y) \leftarrow$
and $G_1(x, y)$ the following goal $\leftarrow eat(x, y)$
Since $P_1 \cup \{G_1(tim, y)\} \in O_{ff}(P_1)$
and $P_1 \cup \{G_1(bob, y)\} \notin O_{ff}(P_1)$,
then $x \rightarrow_{G_1}^{P_1} y$.
In other words, if we hide tim and bob from the goals and since the first goal fails and the second one does not fail, we can distinguish by looking at the facts that the first constant is not bob while the second one is bob, consequently the flow occurs.

UNDER PEER REVIEW

Definition 3.2. $G(a, y) \in O_{ss}(P)$ and $G(b, y) \notin O_{ss}(P)$.

Example 3.2. *Let P_2 be the following program:*
$C_2 : love(bob, y) \leftarrow$
and $G_2(x, y)$ the following goal $\leftarrow love(x, y)$
Since $P_2 \cup \{G_2(bob, y)\} \in O_{ss}(P_2)$
and $P_2 \cup \{G_2(tim, y)\} \notin O_{ss}(P_2)$,
then $x \rightarrow^{P_2}_{G_2} y$.
If we hide bob and tim from the goals and since the first goal succeeds and the second one fails, we can distinguish by looking at the facts that the first constant is bob while the second one is not bob, consequently the flow occurs.

Definition 3.3. $G(a, y) \in O_{dd}(P)$ and $G(b, y)(P) \notin O_{dd}$.

Example 3.3. *Let P_3 be the following program:*
$C_3 : p(a, a) \leftarrow$
and $G_3(x, y)$ the following goal $\leftarrow p(x, y)$
Since $P_3 \cup \{G_3(a, y)\} \in O_{dd}(P_3)$
and $P_3 \cup \{G_3(b, y)\} \notin O_{dd}(P_3)$,
then $x \rightarrow^{P_3}_{G_3} y$.
Informally, if we hide constants a and b from the goals and since the first goal is suspended and the second one fails, we can distinguish by looking at the facts that the first constant is a while the second one is a, consequently the flow occurs.

Definition 3.4. $G(a, y) \in O_{ii}(P)$ and $G(b, y) \notin O_{ii}(P)$.

Example 3.4. *Let P_4 be the following program:*
$C_4 : p(a, y) \leftarrow p(a, y)$
and $G_4(x, y)$ the following goal $\leftarrow p(x, y)$
Since $P_4 \cup \{G_4(a, y)\} \in O_{ii}(P_4)$
and $P_4 \cup \{G_4(b, y)\} \notin O_{ii}(P_4)$,
then $x \rightarrow^{P_4}_{G_4} y$.
Obviously the first goal loops indefinitely while the second one fails, thus, we can distinguish by looking at the facts that the first constant is a while the second one is not a, consequently the flow occurs.

Definition 3.5. $G(a, y) \in O_{ff}(P)$ and $G(a, y) \notin O_{ss}(P)$ and
$G(b, y) \in O_{ss}(P)$ and $G(b, y) \notin O_{ff}(P)$.

Example 3.5. *Let P_5 be the following program:*
$C_5 : kill(b, y) \leftarrow$
and $G_5(x, y)$ the following goal $\leftarrow kill(x, y)$
Since $P_5 \cup \{G_5(a, y)\} \in O_{ff}(P_5)$ and $P_5 \cup \{G_5(a, y)\} \notin O_{ss}(P_5)$
and $P_5 \cup \{G_5(b, y)\} \in O_{ss}(P_5)$ and $P_5 \cup \{G_5(b, y)\} \notin O_{ff}(P_5)$,
then $x \rightarrow^{P_5}_{G_5} y$.
In fact, the first goal $G_5(a, y)$ fails and thus does not succeed too, while the second goal $G_5(b, y)$ succeeds and thus does not fail. Thus, according to the definition, we can distinguish by looking at the facts that the first constant is not b while the second one is b, consequently the flow occurs.

Definition 3.6. $G(a, y) \in O_{ff}(P)$ and $G(a, y) \notin O_{ss}(P)$ and
$G(b, y) \in O_{ss}(P)$.

UNDER PEER REVIEW

Example 3.6. *Let P_6 be the following program:*
$C_1 : kill(b, y) \leftarrow$
$C_2 : kill(b, y) \leftarrow q$
and $G_6(x, y)$ the following goal $\leftarrow kill(x, y)$
Since $P_6 \cup \{G_6(a, y)\} \in O_{ff}(P_6)$ and $P_6 \cup \{G_6(a, y)\} \notin O_{ss}(P_6)$,
and $P_6 \cup \{G_6(b, y)\} \in O_{ss}(P_6)$,
then $x \rightarrow_{G_6}^{P_6} y$.
Obviously the first goal fails and does not succeeds while the second one succeeds and fails, thus it succeeds. Consequently, we can distinguish by looking at the facts that the first constant is not b while the second one is b, therefore the flow occurs.

3.2 Information flow based on transition system

Having a program P, and a goal $G(x, y)$, there is a flow of information from x to y in $G(x, y)$ related to a transition system (noted $x \rightarrow_G^P y$) iff $\exists a, b \in U_L$ such that:

$$\exists a < G(a, y); \epsilon > \xrightarrow{*} < G_n; \theta_n >$$
$$\forall b < G(b, y); \epsilon > \xrightarrow{*} < G_m; \theta_m >$$

Definition 3.7. $G_n, G_m \in O_{ss}(P)$ and $\theta_n \neq \theta_m$.

Example 3.7. *Let P_7 be the following program:*
$C_1 : eat(bob, y) \leftarrow y = apple$
$C_2 : eat(tim, y) \leftarrow y = banana$
and $G_7(x, y)$ the following goal $\leftarrow eat(x, y)$
Since $< G_7(bob, y); \epsilon > \rightarrow < y = apple; \epsilon > \rightarrow < \Box; y = apple >$
and $< G_7(tim, y); \epsilon > \rightarrow < y = banana; \epsilon > \rightarrow < \Box; y = banana >$,
then $x \rightarrow_{G_7}^{P_7} y$.
In other words, if we hide bob and tim from the goals and since both succeed with different substitution answers, respectively $\{y = apple\}$ and $\{y = banana\}$, we can distinguish by looking at the facts that the first constant is bob while the second one is tim, consequently the flow occurs.

Definition 3.8. $G_n, G_m \in O_{ff}(P)$ and $\theta_n \neq \theta_m$.

Example 3.8. *Let P_8 be the following program:*
$C_1 : eat(bob, y) \leftarrow y = apple, q$
$C_2 : eat(tim, y) \leftarrow y = banana, q$
and $G_8(x, y)$ the following goal $\leftarrow eat(x, y)$
Since $< G_8(bob, y); \epsilon > \rightarrow < y = apple, q; \epsilon > \rightarrow < q; y = apple > \not\rightarrow$
and $< G_8(tim, y); \epsilon > \rightarrow < y = banana, q; \epsilon > \rightarrow < q; y = banana > \not\rightarrow$,
then $x \rightarrow_{G_8}^{P_8} y$.
Here both goals fail with different substitution answers, respectively $\{y = apple\}$ and $\{y = banana\}$, we can distinguish by looking at the facts that the first constant is bob while the second one is tim, consequently the flow occurs.

Definition 3.9. $G_n, G_m \in O_{dd}(P)$ and $\theta_n \neq \theta_m$.

Example 3.9. *Let P_9 be the following program:*
$C_1 : eat(bob, y) \leftarrow y = apple, eat(x, y)$
$C_2 : eat(tim, y) \leftarrow y = banana, eat(x, y)$
and $G_9(x, y)$ the following goal $\leftarrow eat(x, y)$
Since $< G_9(bob, y); \epsilon > \rightarrow < y = apple, eat(x, y); \epsilon > \rightarrow < eat(x, apple); y = apple > \not\rightarrow$
and $< G_9(tim, y); \epsilon > \rightarrow < y = banana, eat(x, y); \epsilon > \rightarrow < eat(x, banana); y = banana > \not\rightarrow$,

UNDER PEER REVIEW

then $x \rightarrow^{P_9}_{G_9} y$.

In this example, both goals are suspended due to the matching process. The outputs of both goals in terms of substitution answers are respectively $\{y = apple\}$ *and* $\{y = banana\}$*. Thus, the user can distinguish by looking at the facts that the first constant is* bob *while the second one is* tim*, and consequently the flow occurs.*

Definition 3.10. $G_n, G_m \in O_{ii}(P)$ and $\theta_n \neq \theta_m$.

Example 3.10. *Let* P_{10} *be the following program:*
$C_1 : eat(bob, y) \leftarrow y = apple, eat(bob, z)$
$C_2 : eat(tim, y) \leftarrow y = banana, eat(tim, z)$
and $G_{10}(x, y)$ *the following goal* $\leftarrow eat(x, y)$
Since $< G_{10}(bob, y); \epsilon > \rightarrow < y = apple, eat(bob, z); \epsilon > \rightarrow < eat(x, apple); y = apple > \rightarrow$
$< G_{10}(bob, z); y = apple > \rightarrow \cdots$
and $< G_{10}(tim, y); \epsilon > \rightarrow < y = banana, eat(tim, z); \epsilon > \rightarrow < eat(x, banana); y = banana > \rightarrow$
$< G_{10}(tim, z); y = banana > \rightarrow \cdots$,
then $x \rightarrow^{P_{10}}_{G_{10}} y$.

Here, both goals loop indefinitely. However, the outputs of both goals in terms of substitution answers are respectively $\{y = apple\}$ *and* $\{y = banana\}$*. Thus, as in the previous example, the user can distinguish by looking at the facts that the first constant is* bob *while the second one is* tim*, and consequently the flow occurs.*

Definition 3.11. $G_n \in O_{ss}(P)$, and $G_m \notin O_{ss}(P)$.

Example 3.11. *Let* P_{11} *be the following program:*
$C_1 : eat(bob, y) \leftarrow$
$C_2 : eat(tim, y) \leftarrow q$
and $G_{11}(x, y)$ *the following goal* $\leftarrow eat(x, y)$
Since $< G_{11}(bob, y); \epsilon > \rightarrow < \Box; \epsilon >$
and $< G_{11}(tim, y); \epsilon > \rightarrow < q; \epsilon >$,
then $x \rightarrow^{P_{11}}_{G_{11}} y$.

In this example, the first goal succeeds while the second one does not. The user can distinguish by looking at the facts that the first constant is bob *while the second one is* tim*, and consequently the flow occurs.*

Definition 3.12. $G_n \in O_{dd}(P)$, and $G_m \notin O_{dd}(P)$.

Example 3.12. *Let* P_{12} *be the following program:*
$C_1 : eat(bob, apple) \leftarrow$
$C_2 : eat(tim, y) \leftarrow q$
and $G_{12}(x, y)$ *the following goal* $\leftarrow eat(x, y)$
Since $< G_{12}(bob, y); \epsilon >$
and $< G_{12}(tim, y); \epsilon > \rightarrow < q; \epsilon > \not\rightarrow$,
then $x \rightarrow^{P_{12}}_{G_{12}} y$.

In this example, the first goal succeeds while the second one got suspended due to the matching process. The user can distinguish by looking at the facts that the first constant is bob *while the second one is* tim*, and consequently the flow occurs.*

Definition 3.13. $G_n \in O_{ii}(P)$, and $G_m \notin O_{ii}(P)$.

Example 3.13. *Let* P_{13} *be the following program:*
$C_1 : eat(bob, y) \leftarrow eat(bob, z)$
$C_2 : eat(tim, y) \leftarrow q$

UNDER PEER REVIEW

and $G_{13}(x, y)$ *the following goal* $\leftarrow eat(x, y)$
Since $< G_{12}(bob, y); \epsilon > \rightarrow < G_{13}(bob, z); y \rightarrow z > \rightarrow ...$
and $< G_{13}(tim, y); \epsilon > \rightarrow < q; \epsilon > / \rightarrow$,
then $x \rightarrow^{P_{13}}_{G_{13}} y$
Here, the first goal loops indefinitely while the second one got suspended due to the matching process. The user can distinguish by looking at the facts that the first constant is bob *while the second one is* tim*, and thus the flow occurs.*

Definition 3.14. $G_n \in O_{ff}(P)$, and $G_m \notin O_{ff}(P)$.

Example 3.14. *Let* P_{14} *be the following program:*
$C_1 : eat(bob, apple) \leftarrow$
$C_2 : eat(tim, y) \leftarrow q$
and $G_{14}(x, y)$ *the following goal* $\leftarrow eat(x, y)$
Since $< G_{14}(bob, y); \epsilon >$
and $< G_{12}(tim, y); \epsilon > \rightarrow < q; \epsilon > \not\rightarrow$,
then $x \rightarrow^{P_{14}}_{G_{14}} y$.
Here, the first goal fails while the second one got suspended due to the matching process. The user can thus distinguish by looking at the facts that the first constant is bob *while the second one is* tim*. The flow thus occurs.*

3.3 Information flow based on number of transitions

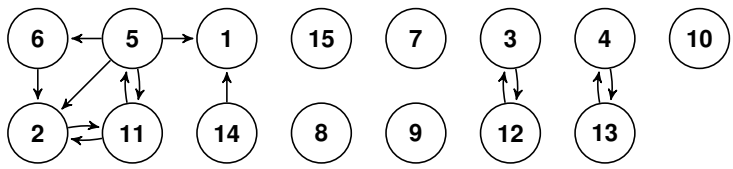
Definition 3.15. Having a program P, and a goal $G(x, y)$, there is a flow of information from x to y in $G(x, y)$ related to the number of transitions (noted $x \rightarrow^{P}_{G} y$) iff $\exists a, b \in U_L$ such that:

$$\exists a < G(a, y); \epsilon > \xrightarrow{*} < G_n; \theta_n >$$
$$\forall b < G(b, y); \epsilon > \xrightarrow{*} < G_m; \theta_m > \text{ such that } n \neq m$$

Example 3.15. *Let* P_{15} *be the following program:*
$C_1 : eat(bob, y) \leftarrow eat(bob, apple)$
$C_2 : eat(bob, apple) \leftarrow$
$C_3 : eat(tim, y) \leftarrow$
and $G_{15}(x, y)$ *the following goal* $\leftarrow eat(x, y)$
Since $< G_{15}(bob, y); \epsilon > \rightarrow < G_{15}(bob, apple); y \rightarrow apple > \rightarrow < \square; y \rightarrow apple >$
and $< G_{15}(tim, y); \epsilon > \rightarrow < \square; \epsilon >$,
then $x \rightarrow^{P_{15}}_{G_{15}} y$.
In this example, both goals succeed, however, the length of the computation varies for each goal. The user can distinguish by looking at the facts and the program that the first constant is bob *while the second one is* tim*. Consequently, the flow occurs.*

3.4 Relationships among the Definitions

The following graph shows the relationships among the different proposed definitions. An edge from node i to node j represents the property *if* $x \rightarrow^{P}_{G} y$ *according to definition* $3.i$*, then* $x \rightarrow^{P}_{G} y$ *according to definition* $3.j$.



UNDER PEER REVIEW

To motivate the existence and/or the absence of edges in the graph, consider the following:
The existence of a flow with respect to definition 3.1 does not entail the existence of a flow with respect to definition 3.5. To show this, let us consider the following example:

Example 3.16. *Let P_{16} be the following program:*
$C_1 : q(a) \leftarrow$
$C_3 : p(b, y) \leftarrow q(y)$
and $G_{16}(x, y)$ the following goal $\leftarrow p(x, y)$
Recall that for a program P, and a goal $G(x, y)$, there is a flow of information from x to y in $G(x, y)$ iff $\exists a, b \in U_L$ such that according to definition 3.1 $G(a, y) \in O_{ff}(P)$ and $G(b, y) \notin O_{ff}(P)$ and according to definition 3.5 $G(a, y) \in O_{ff}(P)$ and $G(a, y) \notin O_{ss}(P)$ and $G(b, y) \in O_{ss}(P)$ and $G(b, y) \notin O_{ff}(P)$.
Obviously there is a flow according to definition 3.1 (the goal $G_{16}(b, y) \notin O_{ff}(P)$ while $G_{16}(a, y) \in O_{ff}(P)$) but there is no flow according to definition 3.5 since one of the conditions is violated (i.e.; $G_{16}(b, y) \notin O_{ss}(P)$).

Let P a concurrent logic program and $G(x, y)$ be a two-variable goal. If $x \rightarrow_G^P y$ according to definition 3.5, then $x \rightarrow_G^P y$ according to definition 3.1. The proof is as follows:
Suppose that $x \rightarrow_G^P y$ according to definition 3.5, then $G(a, y) \in O_{ff}(P)$ and $G(a, y) \notin O_{ss}(P)$ and $G(b, y) \in O_{ss}(P)$ and $G(b, y) \notin O_{ff}(P)$ and in particulary $G(a, y) \in O_{ff}(P)$ and $G(b, y) \notin O_{ff}(P)$. Consequently, $x \rightarrow_G^P y$ according to definition 3.1.

4 Application to Deadlock Detection

In order to show the usefulness of the proposed definitions, we illustrate in this section a method to identify the presence of deadlock in concurrent logic programs. For this, we start by defining the notions of process, resource and deadlock.

4.1 Processes, Resources, and Deadlocks

The execution of a process requires a set of resources (disk, files, etc.) assigned by the operating system. The use of a resource involves the following steps:

- Requesting for the resource: If is is impossible to fulfill the request, the requesting process must wait. The request shall be put into a table containing the processes waiting for resources.
- Acquiring the resource: The process can use the resource. Note that a process cannot use a resource without first requesting it.
- Releasing the resource: The process releases the allocated resource and becomes again available.

When a process requires exclusive access to a resource already allocated to another process, the operating system decides to put it on hold until the required resource becomes available.
Problems can arise when the processes get exclusive access to resources. For example, a process P_1 holds a resource R_1 and is waiting for another resource R_2 that is used by another process P_2; and if the process P_2 holds resource R_2 and waits for resource R_1: a deadlock situation. In fact, P_1 waits for P_2 and P_2 waits for P_1. Both processes will wait indefinitely.
In general, an set of processes is deadlocked if each process waits for the releasing of a resource that is allocated to another process in the set. As all processes are waiting, none will run and thus freeing resources requested by others is impossible.

UNDER PEER REVIEW

4.2 Information-Flow-based Deadlock Detection

Let P be a concurrent logic program, $P_r = \{Pr_1, \cdots, Pr_n\}$ a set of process , and $R = \{R_1, \cdots, R_m\}$ a set of resources. As in common, we require that all studied processes must request at least one resource. This led us to the following characterization of deadlock detection:

Definition 4.1. Let P be a concurrent logic program and let $P_r = \{Pr_1, \cdots, Pr_n\}$, and $R = \{R_1, \cdots, R_m\}$, we say that there is a deadlock in P iff:
$\forall i \in \{1, \cdots, n\}$,

- there is a flow of information from Pr to R restricted to R_i relatively to the goal $acquire(Pr, R_i)$ according to the flow definition based on definition 3.1 or 3.2 (success/failure),
  and
- there is no flow of information from Pr to R restricted to R_i relatively to the goal $request(Pr, R_i)$ according to the flow definition based on definition 3.1 or 3.2 (success/failure).

Formally, $\forall i \in \{1, \cdots, n\}, Pr \rightarrow^P_{acquire(Pr,R_i)} R|_{R_i} \wedge Pr \not\rightarrow^P_{request(Pr,R_i)} R|_{R_i}$

Example 4.1. *Let P be the following concurrent logic program, $Pr = \{P_1, P_2, P_3\}$ and $R = \{R_1, R_2\}$.*
$C_1 : start \leftarrow P_1, P_2, P_3$
$C_2 : P_1 \leftarrow assert(request(Pr_1, R_1)), assert(acquire(Pr_1, R_1)), assert(request(Pr_1, R_2)),$
$\cdots, assert(acquire(Pr_1, R_2)), assert(release(Pr_1, R_1)), assert(release(Pr_1, R_2))$
$C_3 : P_2 \leftarrow assert(request(Pr_2, R_2)), assert(acquire(Pr_2, R_2)), assert(request(Pr_2, R_1)),$
$\cdots, assert(acquire(Pr_2, R_1)), assert(release(Pr_2, R_2)), assert(release(Pr_2, R_1))$
$C_4 : P_3 \leftarrow assert(request(Pr_3, R_2)), assert(acquire(Pr_3, R_2)), \cdots, assert(release(Pr_3, R_2))$

When running the program P by calling the goal $\leftarrow start$, and in the case where process P_1 acquires the resource R_1 while then second process P_2 acquires the second resource R_2, the following facts are asserted:

From P_1	***From*** P_2	***From*** P_3
$request(Pr_1, R_1)$	$request(Pr_2, R_2)$	$request(Pr_3, R_2)$
$acquire(Pr_1, R_1)$	$acquire(Pr_2, R_2)$	
$request(Pr_1, R_2)$	$request(Pr_2, R_1)$	

And since, $Pr \rightarrow^P_{acquire(Pr,R_1)} R|_{R_1} \wedge Pr \not\rightarrow^P_{request(Pr,R_1)} R|_{R_1}$ and $Pr \rightarrow^P_{acquire(Pr,R_2)} R|_{R_2} \wedge Pr \not\rightarrow^P_{request(Pr,R_2)} R|_{R_2}$, we can prove the existence of the deadlock.

5 Conclusion

In this paper, we proposed several definitions of information flow in concurrent logic programming.

a In section (2), we presented the syntax and semantics of first order logic programming. Since most of our definitions of flow are based on transition systems, we introduced this concept for logic programming and then we adapted it to concurrent logic programming. Furthermore, we introduced the notion of observation on which we relied to develop other definitions of information flow.

b In section (3), Fifteen definitions based on transition systems and observation in concurrent logic programming were given. The implications among these definitions were then studied.

c Finally, in section (4), we represented deadlock in a concurrent logic program in terms of information flow. We used one of our definitions (based on success/failure) to accomplish this task.

UNDER PEER REVIEW

However, it seems interesting to represent deadlocks using the other definitions we proposed, and therefore to study the equivalence among the associated detection mechanisms.
Moreover, our study focused on the use of FCP(|) as concurrent logic language. A future study using other languages such as Parlog or FCP(:) seems to be a good way to complete this work.

References

[1] BALBIANI, P., AND YAACOUB, A. Deciding the bisimilarity relation between Datalog goals (regular paper). In *European Conference on Logics in Artificial Intelligence (JELIA), Toulouse, 26/09/2012-28/09/2012* (http://www.springerlink.com, septembre 2012), L. Farias del Cerro, A. Herzig, and J. Mengin, Eds., Springer, pp. 67–79.

[2] BELL, D., AND LAPADULA, L. Secure computer systems: Mathematical foundations and model. *The MITRE Corporation Bedford MA Technical Report M74244 1*, M74-244 (1973), 42.

[3] BELL, D. E., AND LAPADULA, L. J. Secure computer system: Unified exposition and multics interpretation. *Proc 10 1*, MTR-2997 (1976), 118?121.

[4] BERGERETTI, J.-F., AND CARRÉ, B. A. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 7*, 1 (1985), 37–61.

[5] BIBA, K. J. Integrity considerations for secure computer systems. *Proceedings of the 4th annual symposium on Computer architecture 5*, 7 (1977), 135–140.

[6] BISKUP, J. *Security in Computing Systems: Challenges, Approaches and Solutions*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[7] CLARK, K., AND GREGORY, S. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems (TOPLAS) 8*, 1 (1986), 1–49.

[8] DEBRAY, S. Efficient dataflow analysis of logic programs. *Journal of the ACM (JACM) 39*, 4 (1992), 949–984.

[9] FRANKLIN, B. *Poor Richard's Almanack*. Barnes & Noble Books, 2004.

[10] GALLAGHER, J., CODISH, M., AND SHAPIRO, E. Specialisation of prolog and fcp programs using abstract interpretation. *New Generation Computing 6*, 2-3 (1988), 159–186.

[11] LLOYD, J. W. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[12] MAHER, M. J. Logic semantics for a class of committed-choice programs. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25-29, 1987 (2 Volumes)* (1987), pp. 858–876.

[13] PNUELI, A. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, J. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 224 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1986, pp. 510–584.

[14] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003), 5–19.

UNDER PEER REVIEW

[15] SABELFELD, A., AND SANDS, D. A per model of secure information flow in sequential programs. *Higher-order and symbolic computation 14*, 1 (2001), 59–91.

[16] SHAPIRO, E. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR) 21*, 3 (1989), 413–510.

[17] SMITH, G., AND VOLPANO, D. Secure information flow in a multi-threaded imperative language. In *Proceedings of the* 25th *ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1998), ACM, pp. 355–364.

[18] UEDA, K. Guarded horn clauses: A parallel logic programming language with the concept of a guard. In *Programming of Future Generation Computers* (1987), North, pp. 441–456.

[19] YAACOUB, A. Towards an information flow in logic programming. *International Journal of Computer Science Issues (IJCSI) 9*, 2 (2012).

[20] YARDENI, E., KLIGER, S., AND SHAPIRO, E. The languages fcp (:) and fcp (:,?). *New Generation Computing 7*, 2-3 (1990), 89–107.