Analysing Information Quality Requirements in Business Processes Revisited: Formal Framework

In this document, we describe the disjunctive Datalog formalization of the predicates and axioms that underlie our formal framework. First, we introduce the general predicates in Tables 1-4, then we describe the different axioms that are used for modeling and reasoning about Information Quality (IQ) requirements in Business Processes (BPs).

General Predicates

`Type Predicates`: unary predicate that are used for identifying actors, roles, agents, goals, and information entities respectively.
`Actors Relations`: binary predicates that are used for identifying specialization and instantiation relations among actors respectively. The arguments of specialization predicate are two roles (r_1, r_2), and isa(r_1, r_2) is true, if r_1 is specialization of r_2. While the first argument of instantiation predicate is an agent (x) and the last is a role (r), and plays(x, r) is true, if (x) plays (r).

`Actors Properties`: binary predicates that are used to represent the different relation between actors, goals, and information.

`own`: the first argument of own is an actor (a), and the second is information (i), where own(a, g) is true, if a is the legal owner of i.

`aims`: the first argument of aims is an actor (a), and the second is a goal (g), where aims(a, g) is true, if g is a top level goal of a.

`objective`: the first argument of objective is an actor (a), and the second is a goal (g), where objective(a, g) is true, if g is an objective (directly or indirectly) of a.

`producer`: the first argument of producer is an actor (a), and the second is information (i), while the last is time (currency (age) of information). Producer(a, i, 0) is true, if a is the producer of i.

`reader`: the first argument of reader is purpose of use (*pou*), and the second is an actor (a), while the last is information (i). Reads(*pou*, a, i) is true, if a needs to read i for purpose of use *pou*.

`modifier`: the first argument of modifier is an actor (a), and the second is information (i), where modifier (a, i) is true if a needs to modify i.

`sender`: the first argument of sender is the required time of send, and the second and third are actors (a) (sender and the receiver respectively), while the last is information (i) to be send. Sender(t,a, b, i) is true, if a needs to send i to a within time t.

`has`: the first argument of has is an actor (a), and the second is information (i), while the last is *read-time* the currency (age) of information that an actor has. has(a, i, t) is true, if a has i with currency t.

Table 1: General Predicates I

Type Predicates	
actor(Actor:a)	role(Role:r)
agent(Agent:x)	goal(Goal:g)
info(Info:i)	
Actors Relations	
isa(Role:r_1, Role:r_2)	plays(Agent:x, Role:r)
Actors Properties	
own(Actor:a, Info:i)	aims(Actor:a, Goal:g)
objective(Actor:a, Goal:g)	producer(Actor:a, Info:i)
reader(POU:purpose, Actor:a, Info:i).	sender(Time:t, Actor:a, Actor:b, Info:i).
modifier(Actor:a, Info:i)	has(Actor:a, Info:I, Time:t)
has_perm(Perm:p, Actor:a, Info:i)	need_perm(Perm:p, Actor:a, Info:i)
can_provide(Actor:a, Info:i)	is_responsible(Actor:a, Goal:g)
capable_achieve(Actor:a, Goal:g)	can_achieve(Actor:a, Goal:g)
achieve(Actor:a, Goal:g)	achieved(Actor:a, Goal:g)

has_perm the first argument of has_perm is the permission type p,r,m,s, the second is an actor a, while the last if information i. has_perm(Perm:p, Actor:a, Info:i) is true, if actor a has permission $p/r/m/s$ over information i.

need_perm the first argument of need_perm is the permission type p,r,m,s, the second is an actor a, while the last if information i. need_perm(Perm:p, Actor:a, Info:i) is true, if actor a needs permission $p/r/m/s$ over information i.

can_provide: the first argument of can_provide is an actor (a), and the second is information (i), where can_provide(a, i) is true if a is able to provide i.

is_responsible: the first argument of is_responsible is an actor (a), and the second is a goal (g), where is_responsible(a, g) is true, if a took the responsibility of g achievement.

can_achieve: the first argument of can_achieve is an actor (a), and the second is a goal (g), where can_achieve(a, g) is true, if a has the capability (directly or indirectly) to achieve g.

capable_achieve: the first argument of capable_achieve is an actor (a), and the second is a goal (g), where capable_achieve(a, g) is true, if a has the self-capability to achieve g.

Table 2: General Predicates II

Actors' Goals/ Information Dependency
provide(Type: tp, T, Time:t, Actor:a, Actor:b, Info:i)
prvChain(Type: tp, T, Time:t, Actor:a, Actor:b, Info:i)
delegate(Actor:a, Actor:b, Goal:g)
deleChain(Actor:a, Actor:b, Goal:g)
delegate_perm(Actor:a, Actor:b, perm:p, perm:r, perm:m, perm:s,Info:i)
dele_perm_Chain(Actor:a, Actor:b, perm:p, perm:r, perm:m, perm:s,Info:i)
Trust Analysis
trust(Actor:a, Actor:b, Operation: achieve, Goal:g)
trustChain(Actor:a, Actor:b, Operation: achieve, Goal:g)
trust_perm(Actor:a, Actor:b, TypeP:trust/distrust, TypeR: trust/distrust, TypeM: trust/distrust, TypeS: trust/distrust, Info:i)
trust_perm_chain(Actor:a, Actor:b, TypeP:trust/distrust, TypeR: trust/distrust, TypeM: trust/distrust, TypeS: trust/distrust, Info:i)
trustPerm(Type_perm: tp, Actor:a, Actor:b, type_activity:ta, Info:i)

achieved: the first argument of achieved is an actor (a), and the second is a goal (g), where achieved(a, g) is true, if g is achieved (directly or indirectly) from the perspective of actor a.

achieve: the first argument of achieve is an actor (a), and the second is a goal (g), where achieve(a, g) is true, if g is achieved directly by actor a.

Actors' Goals/ Information Dependency: ternary predicates that are used to represent the different relation between actors, goals, and information.

provide/ prvChain: the first argument of provide/ prvChain is provision *type*,the second argument is *time*, while the third and forth arguments are actors (a, b), and the last is information i, where provide(*type*, *time*, a, b, i)/ prvChain(*type*, *time*, a, b, i) is true if a provides (directly or indirectly) b with information i through *type* provision,and within *time*.

delegate/ deleChain: the first two arguments of delegate/ deleChain are actors (a, b), and the third is a goal g, where delegate(a, b, g)/ deleChain(a, b, g) is true if a delegates a goal g (directly or indirectly) to b.

Table 3: General Predicates III

Goal Analysis	
`andDecomposition(Goal:g, Goal:`g_1`)`	`orDecomposition(Goal:g, Goal:`g_1`)`
`not_leaf(Goal:g)`	
Goals' Properties	
`produces(Goal:g, Info:i)`	`read(Goal:g, Info:i)`
`modify(Goal:g, Info:i)`	`send(Time:t, Goal:g, Actor:a, Info:i)`
`dependent(Goal:g)`	`produce_dependent(Goal:g, Info:i).`
`read_dependent(Goal:g, Info:i).`	`modify_dependent(Goal:g, Info:i).`
`send_dependent(Goal:g, Info:i).`	`prevented(Goal:g)`
`produce_prevented(Goal:g, Info:i).`	`read_prevented(Goal:g, Info:i).`
`modify_prevented(Goal:g, Info:i).`	`send_prevented(Goal:g, Info:i).`

`trust`/ `trustChain`: the first argument of trust is a type (trust/ distrust), the second and third arguments are actors (a, b), while the forth is achieve (operation) and the last is the goal (g). We say trust(Type:t, Actor:a, Actor:b, Operation:o, Trustum:tm)/ trustChain (Type:t, Actor:a, Actor:b, Operation:o, Trustum:tm) is true if a tust/trustChain b for the achievement of goal g.

`delegate_perm`/ `dele_perm_Chain`: the first two arguments of delegate_perm/ dele_perm_Chain are actors (a, b), and arguments from number three until six are permissions for produce, reads, modify, and send, while the last is information i. We say delegate_perm(a, b, p,r,m,s,i)/ dele_perm_Chain(a, b, p,r,m,s,i) is true if a delegates b permission $p/r/m/s$ over information i.

`trust_perm`/ `trust_perm_chain`: the first two arguments are actors, and arguments from number three until six are trust/distrust over produce, read, modify, and send permissions respectively, while the last argument is information i. trust_perm(a, b, p,r,m,s,i)/ trust_perm_Chain(a, b, p/x,r/x,m/x,s/x,i) is true if a trust/distrust b permission concerning $p/r/m/s$ over information i.

`Goal Analysis`: binary predicates that are used for AND/Or goal refinement.

`andDecomposition`: the two arguments of andDecomposition are goals (g, $g1$), and andDecomposition(g, $g1$) is true, if $g1$ is and refinement of g.

`orDecomposition`: the two arguments of orDecomposition are goals (g, $g1$),

Table 4: General Predicates IV

IQ Analysis	
`fits_send(T, Goal:g, Actor:a, Info:i)`	`unauthorized_modify(Info:i)`
`fits_read(POU, Goal:g, Info:i)`	`fits_reader(Actor:a, Info:i)`
`accessible_read(Actor:a, Info:i)`	`accurate_read(Actor:a, Info:i)`
`inaccurate(Actor:a, Info:i)`	`valid_read(Actor:a, Info:i)`
`invalid_read(Actor:a, Info:i)`	`consistent_read(Actor:a, Info:i)`
`inconsistent_reader(Actor:a, Info:i)`	`interdependent_readers(Actor:a, Actor:b, Info:i)`
`read_time(Time: t, Actor:a, Info:i)`	

and orDecomposition(g, $g1$) is true, if $g1$ is or refinement of g.

`not_leaf`: the only argument of not_leaf is a goal (g), where not_leaf(g) is true, if g is not a leaf goal (g is not and/or decomposed from another goal).

`Goals' Properties`: unary and binary predicates that are to describe the different properties of goals.

`produces`: the first argument of produces is a goal (g), and the second is information (i), where produces(g, i) is true, if g produces i.

`read`: the first argument of read is purpose of use (*pou*), and the second is goal (g), while the last is information (i). read(*pou*, g, i) is true, if g needs to read i for purpose of use *pou*.

`modify`: the first argument of modify is a goal (g), and the second is information (i), where modify(g, i) is true, if g modify i.

`send`: the first argument of send is time (t), the second is a goal (g), and the third is an actor a, while the last is information (i), where send(t, g, a, i) is true, if g sends i to actor a within time t.

`dependent`: the only argument of dependent is a goal (g), where dependent(g) is true, if g is information dependent. produce_dependent / read_dependent/ modify_dependent / send_dependent the first argument is a goal (g), and the second is information i, and it is true, if the goals g produce/reads/modify.send information i.

`prevented`: the only argument of prevented is a goal (g), where prevented(g) is true, if g was prevented by any reason for being achieved. produce_prevented/ read_prevented/ modify_prevented/ send_prevented the first argument is a goal (g), and the second is information i, and it is true, if the goals g was prevented due to produce/ reads/ modify/ send information i related issues.

`IQ Analysis` predicates that can be used to analyze IQ related dimensions.

`fits_send`: the first argument of fits_send is time (t), the second is a goal (g), and the third is an actor a, while the last is information (i), where fits_send(t, g, a, i) is true, if g sends i to actor a within time t.

`unauthorized_modify`: has only one argument information (i), where unauthorized_modify(Info:i)is true, if i is modify in unauthorized way.

`fits_read`: the first argument of fits_read is purpose of use (*pou*), and the second is goal (g), while the last is information (i). fits_read(*pou*, g, i) is true if i fit for purpose of use *pou* of g.

`fits_reader`: the first argument of fits_read is purpose of use (*pou*), and the second is an actor (a), while the last is information (i). fits_read(*pou*, a, i) is true if i fit for purpose of use *pou* of a (reader).

`accessible_read(Actor:a, Info:i)` the first argument of accessible_read is an actor (a), while the last is information (i). accessible_read(Actor:a, Info:i) is true if a is allowed to read i.

`accurate_read` / `inaccurate` the first argument of accurate_read/ inaccurate_read is an actor (a), while the last is information (i). accurate_read(Actor:a, Info:i)/ inaccurate_read(Actor:a, Info:i) is true if i is accurate/ inaccurate from the perspective of a (reader).

`valid_read`/ `invalid_read` the first argument of valid_read/ invalid_read is an actor (a), while the last is information (i). valid_read(Actor:a, Info:i)/ invalid_read(Actor:a, Info:i) is true if i is valid/ invalid from the perspective of a (reader).

`consistent_read`/ `inconsistent_read` the first argument of consistent_read/ inconsistent_read is an actor (a), while the last is information (i). consistent_read(Actor:a, Info:i) is true if i is consistent/ inconsistent from the perspective of a (reader).

`interdependent_readers` the first and second arguments of interdependent_readers are actors (a, b), while the last is information (i). `interdependent_readers(Actor:a, Actor:b, Info:i)` is true if a and b reads i for the same purpose of use.

`read_time`: the first argument of read_time is time t, the second is an actor A, while the last if information i. read_time(Time: t, Actor:a, Info:i) is true if actor a reads information i in time t.

Actors Objectives, Entitlements and Capabilities Axioms

Table 5, lists the actors' objectives, entitlements and capabilities axioms. For example, O1 states that if an actor aims for a goal, it became an objective for such actor. O2 states that if a goal is delegated to an actor, it became its objective. O3-4 state that if an actor aims for a goal, and this goal is and/or decomposed, all the sub-goals became an objectives of the actor. E1 states that

an actor became responsible of a goal achievement, if the goal is an objective of the actor, the actor has the capabilities to achieve it, and the goal is leaf goal.

C1 states that an actor is capable of achieving a goal, if the actor plays a role that has such capability. While C2 states that a role is capable of achieving a goal, if the role is specialized of a role that has such capability. C3 states that an actor can achieve a goal, if the actor is capable of achieving it. C4 states that an actor can achieve a goal, if it delegates the goal to an actor who has the capability of achieving it. C5 states that an actor can achieve a goal, if the goal is or decomposed, and the actor has the capability of achieving at least one of the sub-goals. C6 states that an actor can achieve a goal, if the goal is and decomposed (two sub goals), and the actor has the capability of achieving all its sub-goals.

C7-10 state that an actor is producer/ reader/ sender/ modifier of information, if the actor is achieve and/ or is responsible a goal that produces/reads/sends/modifies such information. C11 states that an actor has information, if it is its producer. C12 states that an actor has information, if such information has been provided to it. C13 is used to define whether an actor has information regardless its currency. C14 states that an actor can provide information, if it has such information.

C15-18 state that an actor has produce/read/modify/send permission, if it is the owner of such information. C19-22 state that an actor has produce/read/modify/send permission, if such permission has been delegated to it from an actor has such permission. C23-26 state that an actor needs produce/read/modify/send permission, if it is a producer/reader/modifier/sender of such information.

O1	`objective(A, G) :- aims(A, G).`
O2	`objective(A, G) :- deleChain(B, A, G), objective(B, G).`
O3	`objective(A, G1) :- andDecomposition(G, G1), objective(A, G).`
O4	`objective(A, G1) :- orDecomposition(G, G1), objective(A, G).`
E1	`is_responsible(A, G) :- objective(A, G), can_achieve(A, G), not not_leaf(G).`
C1	`capable_achieve(A, G):-play(A, R), capable_achieve(R, G).`
C2	`capable_achieve(R1, G):-is_a(R1, R2), capable_achieve(R2, G).`
C3	`can_achieve(A, G) :- capable_achieve(R, G).`
C4	`can_achieve(A, G) :- deleChain(A, B, G), capable_achieve(B, G).`

C5	can_achieve(A, G) :- orDecomposition(G, G1), can_achieve(A, G1).
C6	can_achieve(A, G) :- andDecomposition(G, G1), andDecomposition(G, G2), can_achieve(A, G1), can_achieve(A, G2), G1 != G2.
C7	producer(A,I,T) :- achieve(A,G), produces(Ty,G,I,T).
C8	reader(Ty,POU,Bt,A, I):- is_responsible(A, G), read(Ty,POU,Bt,G, I) .
C9	sender(T,A,B,I):- is_responsible(A,G), send(T,G,B,I).
C10	modifier(A,I):- is_responsible(A,G), modify(G,I) .
C11	hasT(A, I, 0) :- producer(A, I, T).
C12	hasT(A,I,T3) :- prvChain(Ty,T1,B,A,I),has(B,I,T2), #int(T1), #int(T2), #int(T3), T3=T1+T2.
C13	has(A, I):- hasT(A, I, T).
C14	can_provide(A, I) :- has(A, I).
C15	has_perm(p, A, I):- own(A, I).
C16	has_perm(r, A, I):- own(A, I).
C17	has_perm(m, A, I):- own(A, I).
C18	has_perm(s, A, I):- own(A, I).
C19	has_perm(P, B, I) :- delegate_perm_chain(A, B, P,_, _, _, I), has_perm(P, A, I).
C20	has_perm(R, B, I) :- delegate_perm_chain(A, B, _, R, _, _, I), has_perm(R, A, I).
C21	has_perm(M, B, I) :- delegate_perm_chain(A, B, _,_, M, _, I), has_perm(M, A, I).
C22	has_perm(S, B, I) :- delegate_perm_chain(A, B, _, _, _, S, I), has_perm(S, A, I).
C23	need_perm(p, A, I):-producer(A, I, T) .
C24	need_perm(r, A, I):-reader(Ty, POU, BT, A, I) .
C25	need_perm(m, A, I):-modifier(A, I) .
C26	need_perm(s, A, I):-sender(T, A, B, I) .

Table 5: Actors' Objectives, Entitlements and Capabilities Axioms

Goal & Information Axioms

Axioms concerning the different relations among goals/information are listed in Table 6. For example, G1-2 state that a goal is not_leaf, if it is and/or decomposed of another goal, and G3-10 states that a goal is dependent, if it produces, reads, modifies, and/or sends information. G11-14 state that a goal is prevented, if it has been prevented due to produce, read, modify, send issues respectively. While G15-65 describe when IQ related issues might prevent a goal.

G1	`not_leaf(G) :- andDecomposition(G, G1).`
G2	`not_leaf(G) :- orDecomposition(G, G1).`
G3	`dependent(G):- produce_dependent(G, I).`
G4	`dependent(G):- read_dependent(G, I).`
G5	`dependent(G):- modify_dependent(G, I).`
G6	`dependent(G):- send_dependent(G, I).`
G7	`read_dependent(G, I):- read(r,POU, G, I).`
G8	`send_dependent(G, I):- send(T, G, B, I).`
G9	`modify_dependent(G, I):- modify(G, I).`
G10	`produce_dependent(G, I):- produces(Ty,G, I, T).`
G11	`prevented(G):- modify_prevented(G, I).`
G12	`prevented(G):- produce_prevented(G, I).`
G13	`prevented(G):- send_prevented(G, I).`
G14	`prevented(G):- read_prevented(G, I).`
G15	`modify_prevented(G, I):- modify(G, I), is_responsible(A, G), has_perm(m, A, I).`
G16	`produce_prevented(G, I):- produces(Ty,G, I, T), is_responsible(A, G), allowed_produce(A, I).`
G17	`allowed_produce(A, I):- producer(A, I, T), has_perm(p, A, I) .`
G18	`accurate_produce(A,I):- is_responsible(A, G), produce(chk_blv, G, I, T), trusted_produce(A,I).`
G19	`trusted_produce(A, I):- producer(A, I, T), own(A,I).`
G20	`trusted_produce(A, I):- producer(A, I, T), has_perm(p, A, I), trustedPerm(B,A,p,I), own(B,I).`
G21	`send_prevented(G, I):- send(T, G, B, I), not fits_send(T, G, B, I).`
G22	`fits_send(T, G, B, I):- is_responsible(A, G), send(T, G, B, I), fits_sender(T, A, B, I).`

G23 `Fits_sender(T, A, B, I):- accurate_send(T, A, B, I), complete_send(T, A, B, I), valid_send(T, A, B, I), allowed_send(A, I).`

G24 `allowed_send(A, I):- has_perm(s, A, I).`

G25 `accurate_send(T, A, B, I):- sender(T, A, B, I), hasT(B, I, _), not unauthorized_modify(I).`

G26 `unauthorized_modify(I):- modifier(A, I), own(B, I), not trustedPerm(B, A, m, I), not own(A, I).`

G27 `complete_send(T, A, B, I):- sender(T, A, B, I), prvChain(iprovision, Tr, A, B, I).`

G28 `valid_send(T, A, B, I):- sender(T, A, B, I), prvChain(_, Tr, A, B, I), #int(T), #int(Tr), Tr <= T.`

G29 `fits_send(T, G, B, I):- is_responsible(A, G), send(T, G, B, I), prvChain(ip, Tr, A, B, I), not unauthorized_modify(I), #int(T), #int(Tr), Tr <= T.`

G30 `unauthorized_modify(I):- modifier(A, I), own(B, I), not trustPerm(trust, A, B, modify, I).`

G31 `read_prevented(G, I):- read(r, POU, Bt, G, I), not fits_read(r,POU, Bt, G, I).`

G32 `fits_read(r,POU, Bt,G, I):- is_responsible(A, G), read(r,POU, Bt, G, I), fits_reader(A, I).`

G33 `fits_reader(A, I):- accessible_read(A, I), accurate_read(A, I), complete_read(A, I), valid_read(A, I), consistent_read(A, I).`

G34 `accessible_read(A, I):- reader(Ty,POU,Bt,A, I), has_perm(r, A, I).`

G35 `accurate_read(A, I) :- reader(r, POU, Bt, A, I), has(A, I, T), not inaccurate(A, I).`

G36 `inaccurate(A, I):- reader(r, POU, Bt, A, I), has(A, I,_), producer(B, I, 0), not own(A,I), prvChain(p, T, B, A, I), #int(T).`

G37 `inaccurate(A, I):- read(Ty, POU, no_chk_blv, G, I), reader(_, _, _, A, I).`

G38 `inaccurate(A, I):- reader(Ty, POU, BT, A, I), hasT(A, I, T), unauthorized_modify(I).`

G39 `inaccurate(A, I):- reader(Ty, PoU, BT, A, I), producer(B, I, T), not trust_produce(trust,A,B,I).`

G40 `trust_produce(trust, A, A, I):- producer(A, I, T), own(A, I).`

G41 `complete_read(A, I):- reader(Ty, PoU, BT, A, I), complete_value(A, I), complete_pou(A, I).`

G42 `complete_value(A, I):- producer(A, I, _), reader(_,_,_, A, I).`

G43 `complete_value(A, I):- reader(_,_,_, A, I), hasT(A, I,_), producer(B, I,_), prvChain(iprovision, T, B, A, I), A != B.`

G44 `complete_pou(A, I):- reader(Ty, PoU, BT, A, I), hasT(A, I, _), not composed(I).`

G45 `complete_pou(A, I):- reader(Ty, PoU, BT, A, I), hasT(A, I, _), composedOfOne(I), partOf(I, I1), hasT(A, I1,_).`

G46 `complete_pou(A, I):- reader(Ty, PoU, BT, A, I), hasT(A, I, _), composedOfTwo(I), partOf(I, I1), partOf(I, I2),hasT(A, I1,_), hasT(A, I2,_), I1 != I2.`

G47 `complete_pou(A, I):- reader(Ty, PoU, BT, A, I), hasT(A, I, _), composedOfThree(I), partOf(I, I1), partOf(I, I2), partOf(I, I3), hasT(A, I1,_), hasT(A, I2,_), hasT(A, I3, _), I1 != I2, I1 != I3, I2 != I3.`

G48 `composedOfOne(I):- numOfParts(I, 1).`

G49 `composedOfTwo(I):- numOfParts(I, 2).`

G50 `composedOfThree(I):- numOfParts(I, 3).`

G51 `composed(I) :- composedOfOne(I).`

G52 `composed(I) :- composedOfTwo(I).`

G53 `composed(I) :- composedOfThree(I).`

G54 `numOfParts(I, X):- partOf(I, I1), #countZ: partOf(I, Z) = X.`

G55 `valid_read(A, I):- reader(r, POU, Bt, A, I), read_time(T, A, I), info(I, V), #int(T), #int(V), V >= T.`

G56 `valid_read(A, I):- producer(A,I, _), reader(r, POU, Bt, A, I).`

G57	`invalid_read(A, I):- reader(r, POU, Bt, A, I), not valid_read(A, I).`
G58	`consistent_read(A, I):- only_reader(A, I).`
G59	`only_reader(A, I):- reader(_,_,_, A, I), numOfReaders(X, I), X = 1.`
G60	`numOfReaders(X, I):- reader(_,_,_, A, I), #countZ: reader(_,_,_, Z ,I) = X.`
G61	`consistent_read(A, I) :- reader(_,_,_,A, I), not only_reader(A, I), not inconsistent_reader(A, I).`
G62	`inconsistent_reader(A, I):- interdependent_readers(A,B,I), read_time(X,A,I), read_time(Y,B,I), #int(X), #int(Y), X != Y, A!=B.`
G63	`read_time(T, A, I):- reader(_,_,_,A,I), has(A,I,T).`
G64	`interdependent_readers(A, B, I):- reader(Ty,POU, _,A, I), reader(Ty,POU, _,B, I), A!=B.`
G65	`interdependent_readers(B, A, I):- interdependent_readers(A, B, I).`

Table 6: General Predicates

Actors' Goals/ Information Dependency

Table 7 lists the actors' goals/information/permissions dependency axioms concerning information provision (S1, S2), goals delegation (S3, S4), and permissions delegation (S5, S6). Moreover, it lists trust axioms concerning permissions (S7, S8) and goals(S9, S10). Finally, axioms s11-14 describe how different kinds of permissions concerning produce/ read/ modify/ send are instantiated from permission delegation.

S1	`prvChain(Ty,T,A,B,I):- provide(Ty,T,A,B,I).`
S2	`prvChain(Ty,Z,A,C,I):- prvChain(Ty,X,A,B,I), prvChain(Ty,Y,B,C,I), #int(X), #int(Y), #int(Z), Z=X+Y.`
S3	`deleChain(A,B,G) :- delegate(A,B,G).`
S4	`deleChain(A,C,G) :- delegate(A,B,G), deleChain(B,C,G).`
S5	`dele_perm_chain(A, B, P, R, M, S, I) :- delegate_perm(A, B, P, R, M, S, I).`

S6	`dele_perm_chain(A, B, P, R, M, S, I) :- dele_perm_chain(A, C, P, R, M, S, I), dele_perm_chain(C, B, P, R, M, S, I).`
S7	`trust_perm_chain(A, B, TyP, TyR, TyM, TyS, I):- trust_perm(A, B, TyP, TyR, TyM, TyS, I).`
S8	`trust_perm_chain(A, B, TyP, TyR, TyM, TyS, I) :- trust_perm_chain(A, C, TyP, TyR, TyM, TyS, I), trust_perm_chain(C, B, TyP, TyR, TyM, TyS, I).`
S9	`trustPerm(TyP, A, B, produce, I):- trust_perm_chain(A, B, TyP, TyR, TyM, TyS, I).`
S10	`trustPerm(TyR, A, B, read, I):- trust_perm_chain(A, B, TyP, TyR, TyM, TyS, I).`
S11	`trustPerm(TyM, A, B, modify, I):- trust_perm_chain(A, B, Typ, TyR, TyM, TyS, I).`
S12	`trustPerm(TyS, A, B, send, I):- trust_perm_chain(A, B, TyP, TyR, TyM, TyS, I).`

Table 7: Social Relations Axioms

Goals Achievement Axioms

Table 8 lists axioms used to identify whether a goal is achieved or not from the perspective of the actor, who aims for it. A1 states that a goal is achieved for an actor, if the goal is not information dependent and the actor took the responsibility of achieving it by itself. A2 states that a goal is achieved for an actor, if the goal is information dependent but not prevented, and the actor took the responsibility of achieving it by itself. A3 states that a goal is achieved, if the goal is achieved from the perspective of the actor who aims for it. A4 states that a goal is achieved for an actor, if the goal is delegated to an actor and a trust relation holds between the delegator and delegate, and the goal is achieved from the perspective of the delegatee. A5-6 state that a goal is achieved for an actor, if one (or decomposition) or all (and decomposition) of its sub goals is/are achieved from the perspective of the actor.

A1	`achieve(A,G) :- is_responsible(A,G), not dependent(G).`
A2	`achieve(A,G) :- is_responsible(A,G), dependent(G), not prevented(G).`
A3	`achieved(A,G) :- achieve(A,G).`

A4	`achieved(A,G) :- deleChain(A,B,G), trustChain(A,B,achieve,G), achieve(B,G).`
A5	`achieved(A,G) :- andDecomposition(G,G1), andDecomposition(G,G2),achieved(A,G1), achieved(A,G2), G1 != G2 .`
A6	`achieved(A,G) :- orDecomposition(G,G1), achieved(A,G1).`

Table 8: Goals Achievement Axioms

WFA-net Axioms

Table 9 lists axioms used for the analysis of WFA-nets. W1-3 state that `start(P)`, `end(P)`, and any `between(P)` positions are positions of the WFA-net. W4-5 state that a goal `start(G)` from the goal model became an activity `end(T)` if it has an `in_goal_arc(_, G)`/ `out_goal_arc(G, _)` in the WFA-net. W7 states that any start arc from the start position to an activity, is considered as incoming arc to such activity. W8-19 axioms are used to define whether an activity of the WFA-net is enabled. W20 states that a position P is reached if, there is an outgoing arc from an activity to such position, and such activity firs. W21 states that an activity fires, if it is enabled, the goal it represent is achieved, such activity is not prevented in the WFA-net. W22 states that an activity produces information, if the goal such activity represents produces such information, and the activty fires. W23 states that an activity reads information, if the goal such activity represents reads such activity information. W24 states that an activity of WFA-net is prevented, if such activity reads information that is not produced in the WFA-net yet. W25 states that an activity of WFA-net successfully reached its end, if the end position has been reached.

Positions Predicates	
W1	`position(P):- start(P).`
W2	`position(P):- between(P).`
W3	`position(P):- end(P).`
WFA-net axioms	
W4	`activity(G):- in_goal_arc(_, G).`
W5	`activity(G):- out_goal_arc(G, _).`
W6	`in_goal_arc(P, G):- start_arc(P, G) .`
W7	`enabled(G):- start_arc(start, G) .`
W8	`enabled(G):- reached(P), in_goal_arc(P,G), oneInArc(G) .`

W9	`enabled(G):- reached(P1), in_goal_arc(P1,G), reached(P2), in_goal_arc(P2,G), twoInArc(G), P1 !=P2.`
W10	`numOfInArcs(X, G):- in_goal_arc(P, G), #countZ: in_goal_arc(Z, G) = X.`
W11	`oneInArc(G):- in_goal_arc(P,G), numOfInArcs(X,G), X=1.`
W12	`twoInArc(G):- in_goal_arc(P,G), numOfInArcs(X,G), X=2.`
W13	`arc_in(G):- oneInArc(G).`
W14	`arc_in(G):- twoInArc(G).`
W15	`numOfOutArcs(G, X):- out_goal_arc(G, P), #countZ: out_goal_arc(G, Z) = X.`
W16	`oneOutArc(G):- out_goal_arc(G, P), numOfOutArcs(G, X), X = 1.`
W17	`twoOutArc(G):- out_goal_arc(G, P), numOfOutArcs(G, X), X = 2.`
W18	`arc_out(G):- oneOutArc(G).`
W19	`arc_out(G):- twoOutArc(G).`
W20	`reached(P):- fired(G), out_goal_arc(G, P) .`
W21	`fired(G):- enabled(G), achieved(_,G), not wf_prevented(G), not not_leaf(G) .`
W22	`wf_produced(I):- produce(Type, G, I, T), fired(G).`
W23	`wf_reads(G, I):- read(r, POU, BType, G, I).`
W24	`wf_prevented(G):- wf_reads(G, I), not wf_produced(I).`
W25	`success_WFA_net:- reached(end).`

Table 9: WFA-net Axioms

Model Analysis and Verification

The properties of the design are shown in Table 10, which can be used to verify the correctness of the mapping, control-flow, information-flow and IQ requirements of the WFA-net model. In what follows, we discuss each of them:

Pro1-6 are used to verify the mapping properties of the WFA-net, where **Pro2-6** are derived from the semantics of the WFA-nets, and they are specialized for verifying whether every activity and every position are on a path between the Start and End positions.

Pro1 states that only leaf goals are allowed to be mapped as activities of WFA-net.

Pro2 states that any activity of a WFA-net that has an outgoing arc, should have at least one incoming arc.

Pro3 states that any activity of a WFA-net that has an incoming arc, should have at least one outgoing arc.

Pro4 states that the Start position in a WFA-net should be connected with at least one activity.

Pro5 states that any position (not `P(S)` or `P(E)` positions) in a WFA-net should be connected with at least two activities through one incoming and one outgoing arcs.

Mapping properties	
Pro1 :-	`activity(G), not_leaf(G).`
Pro2 :-	`incoming_arc(G), not outgoing_arc(G).`
Pro3 :-	`outgoing_arc(G), not incoming_arc(G).`
Pro4 :-	`start(P), not starting_arc(P).`
Pro5 :-	`between(P), not connected(P).`
Pro6 :-	`end(P), not ending_arc(P).`
Information flow property	
Pro7 :-	`wf_reads(G, I), not wf_produced(I).`
Information Quality properties	
Pro8 :-	`is_responsible(A, G), activity(G), produce(Type, G, I, T), not has_perm(produce, A, I).`
Pro9 :-	`is_responsible(A, G), activity(G), produce(Type, G, I, T), not accurate_produce(A, I).`
Pro10 :-	`is_responsible(A, G), activity(G), read(T, P, BT, G, I), not has_perm(read, A, I).`
Pro11 :-	`is_responsible(A, G), activity(G), read(T, P, BT, G, I), not accurate_read(A, I).`
Pro12 :-	`is_responsible(A, G), activity(G), read(T, P, BT, G, I), not valid_read(A, I).`
Pro13 :-	`is_responsible(A, G), activity(G), read(T, P, BT, G, I), not complete_read(A, I).`
Pro14 :-	`is_responsible(A, G), activity(G), read(T, P, BT, G, I), not consistent_read(A, I).`

Pro15 :-	`is_responsible(A, G), activity(G), modify(G, I), not has_perm(modify, A, I).`
Pro16 :-	`is_responsible(A, G), activity(G), send(T, G, B, I), not has_perm(send, A, I).`
Pro17 :-	`is_responsible(A, G), activity(G), send(T, G, B, I), has(B, I), not accurate_send(T, A, B, I).`
Pro18 :-	`is_responsible(A, G), activity(G), send(T, G, B, I), not complete_send(T, A, B, I).`
Pro19 :-	`is_responsible(A, G), activity(G), send(T, G, B, I), not valid_send(T, A, B, I).`
Control flow properties	
Pro20 :-	`wf_prevented(G).`
Pro21 :-	`not reached(end).`

Table 10: Properties of the design

Pro6 states that the `End` position in a WFA-net should be connected with at least one activity.

Pro7 states that any activity of WFA-net should have all information it requires (e.g., read, modify, send), where this property is used to verify information availability (information-flow) for activities of a WFA-net.

Pro8-19 are used to verify IQ related properties of the activities of a WFA-net. For instance, **Pro8** states that a WFA-net should not include any activity that produces information, and the actor who is responsible for achieving such activity does not have a produce permission concerning such information.

Pro9 states that a WFA-net should not include any activity that produces inaccurate information from the perspective of the actor who responsible of achieving such activity, where produced information is accurate, if its believability and the trustworthiness of production have been verified.

Pro10 states that a WFA-net should not include any activity that reads information, and the actor who is responsible for achieving such activity does not have a read permission concerning such information.

Pro11 states that a WFA-net should not include any activity that reads information, and such information is inaccurate from the perspective of the actor (reader) who is responsible the activity achievement.

Pro12 states that a WFA-net should not include any activity that reads information, and such information is invalid from the perspective of the actor who is responsible the activity achievement. Information is valid for read if its currency (age) is smaller than its volatility, otherwise it is invalid.

Pro13 states that a WFA-net should not include any activity that reads

information, and such information is incomplete from the perspective of the actor who is responsible for the activity achievement. Information is complete for read, if it is value complete (information has been preserved against lost and corruption during its transfer), and purpose of use complete (information has all its sub-parts for performing a task at hand).

Pro14 states that a WFA-net should not include any activity that reads information, and such information is inconsistent from the perspective of the actor who is responsible for the activity achievement. Information is consistent for read, if it has only one reader taking into consideration its purpose of use, or it has multiple readers for the same purpose of use, and all of them have the same read-time.

Pro15 states that a WFA-net should not include any activity that modifies information, and the actor who is responsible for achieving such activity does not have a modify permission.

Pro16 states that a WFA-net should not include any activity that sends information, and the actor who is responsible for its achievement does not have a send permission concerning such information.

Pro17 states that a WFA-net should not include any activity that sends information, and such information is inaccurate at its destination from the perspective of the actor (sender) who is responsible for the activity achievement. Information is accurate at its destination, if it has not been inappropriately modified during its transfer (trustworthiness of the provision).

Pro18 states that a WFA-net should not include any activity that sends information, and such information is incomplete at its destination from the perspective of the actor who is responsible for the activity achievement.

Pro19 states that a WFA-net should not include any activity that sends information, and such information is invalid at its destination from the perspective of the actor who is responsible for the activity achievement. Information is valid at its destination, if its transfer (provision) time is less than its send time.

Pro20 states that a WFA-net should not include any activity that has been prevented from being fired. Activities might be prevented from being fired due to several reasons. For example, the responsible actor does not have the capability to achieve the activity (goal), the responsible actor is not trusted for achieving the activity. Moreover, an activity might be prevented because of IQ related properties, e.g., activity is not able to produce, read, modify and/or send information, because the responsible actor does not have the required permission. Furthermore, an activity might be prevented because of reading inaccurate, incomplete, etc. information.

Pro21 states that the End position in a WFA-net should be reached, i.e., there should be at least one activity when fired the WFA-net reaches its End position.