AMiRALE Formal Model

Vincent Autefage

▶ **To cite this version:**

Vincent Autefage. AMiRALE Formal Model: A Service Discovery and Collaboration System Formalism based on Dynamic Graph Relabeling. [Research Report] LaBRI - Laboratoire Bordelais de Recherche en Informatique; Université de Bordeaux. 2015. hal-01114961v2

HAL Id: hal-01114961
https://hal.science/hal-01114961v2

Submitted on 9 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AMiRALE Formal Model

A Service Discovery and Collaboration System Formalism based on Dynamic Graph Relabeling

Version 2 - July 2015

Vincent Autefage
autefage@labri.fr

Abstract

AMiRALE (*Asynchronous Missions Relay for Autonomous and Lively Entities*) is a service discovery and collaboration mechanism dedicated to autonomous swarms of highly mobile and heterogeneous nodes. AMiRALE is only based on asynchronous communications and local computations. This report details the internal operations of AMiRALE based on dynamic graph relabeling.

Contents

université de BORDEAUX LaBRI DGA REGION AQUITAINE

1 Introduction

A certain set of events (e.g., temperature threshold exceeded, suspicious movement, etc) requires a reaction of the swarm (e.g., sending a signal to a control center, triggering an alarm, etc.). A *mission* is the set of information that describes such an action to perform by the swarm. A *sensor* is a node that is able to detect an event (through its sensors) and to create the relative mission. A *solver* is a node that is able to apply the specific action required by the mission. Finally, a *forwarder* is just a node which forwards a mission through the swarm.

2 Overall Description

2.1 Node types

For a specific event e, we call $Sens_e$ (sensor) a node which can capture this event. This node generates a new mission called $m_e^{n:k}$ where n is the identifier of the creator node and k is a mission sequence number relative to this node.

This mission is described by a message (a *view*[1] called $v_e^{n:k}$) that travels the network through intermediate nodes called $Forw_e$ (forwarders), until it reaches a final node which can solve the mission $m_e^{n:k}$. Such a final node is called $Solv_e$ (solver).

2.2 Mission state

One of the fields of the mission description $m_e^{n:k}$ is an internal state which evolves while the view travels through the network. This state can take five different values which are strictly ordered:

1. *start*: the mission has been created but no node is currently trying to solve it;
2. *will*: the mission has been caught by a solver, and it is preparing to solve it (e.g., moving to the location where the mission takes place);
3. *do*: the mission is currently being solved;
4. *abort*: the mission has been dropped because it has apparently already been solved (e.g., a target object that the solver is supposed to remove has disappeared);
5. *end*: the mission is solved.

The *start* state is only set at mission creation time by the node that generates it. The other states can only be set by solvers. A forwarder is a *read-only node*; i.e., it cannot modify the state field. As explained before, states are strictly ordered: $start < will < do < abort < end$. A finite state machine of the evolution of the state field is shown in Figure 1.

2.3 Mission life

Each mission description contains a state, a creator identifier and creation date[2], the date of modification and the identifier of the last node that updated it.

At regular time intervals, the nodes of the swarm broadcast to their neighborhood the list of mission views they are aware of in order to update each other. Note that nodes communicate only asynchronously and one way.

When a $Sens_e$ node catches an event e, it is possible that another mission initiated because of e already exists. However, the decision to generate a new mission is application dependent, therefore a user function called f_{ignore}^e will decide if the event e has to be ignored or not.

When a node receives a newer view of one of the missions it is aware of, it updates this mission in its local memory and broadcasts the new mission view.

[1] A *view* is a reduced form of a mission.

[2] We assume here that each node has a unique identifier and its own clock.

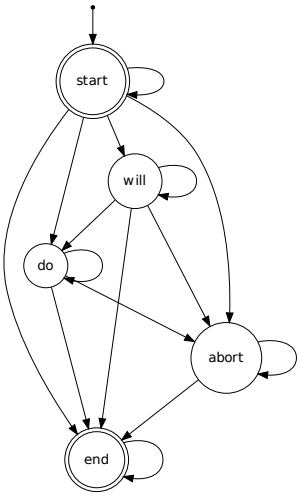


Figure 1: Finite state machine of the state mission field

When a solver gets a mission in *start* state that it is able to solve, it turns the state field of the mission to *will* and prepares itself to solve it (e.g., by moving to a specific location if required). When it begins solving the mission, it turns its state field to *do*. If the solver detects that the mission has apparently already been solved (e.g., a target object that the solver is supposed to remove has disappeared), it turns the state field to *abort*. Finally, when it has succeeded in solving the mission, it turns the state field to *end*. A solver can be in *will* or *do* mode for only one mission at a time.

Furthermore, a solver $Solv_e$ can stay in the *will* (resp. *do*) state only for a limited time Ψ^e_{will} (resp. Ψ^e_{do}). If one of these thresholds is exceeded, the solver leaves the mission if it is informed by another solver that this last one is now taking care of the same mission.

Additionally, if a solver is informed that the mission it is dealing with (*will* or *do* state) has evolved to a greater state, it leaves the mission and updates the relative description.

For a specific mission relative to an event e, if a node is not a $Sens_e$ nor a $Solv_e$, it is considered as a $Forw_e$. By default, a forwarder is able to deal with a mission of type e even it has not been aware of this kind of event before. Indeed, views contain specific information (i.e., thresholds, etc.) of the relative type. In other words, it is possible to add several new mission types during the swarm operation.

3 Time synchronization consideration

Each node has its own internal clock and the time can thus drift differently from one node to the other. It is thus necessary to take this problem into account in our discovery system. Consequently, we have developed three versions of AMiRALE model, one for each synchronization technique:

- *relative ordering* where events are ordered without time reference;
- *relative timing* where nodes take into account the time drift relative to the others;
- *global timing* where all the clocks are synchronized (by using GPS information for instance).

4 Meta-Model

4.1 Overall description

To simplify the understanding of the different versions of AMiRALE, we define a meta-model which describes the operations and communications used to solve the mission at the level of each entity.

As explained before, a node can have three different roles:

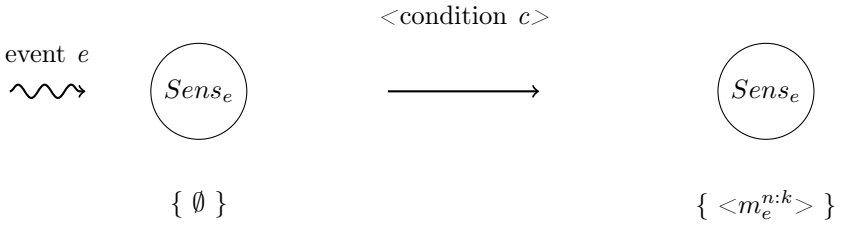
- $Sens_e$ which means that the node can sense an event e and creates the relative mission $m_e^{n:k}$;
- $Solv_e$ which means that the node can solve a mission $m_e^{n:k}$;
- $Forw_e$ which means that the node can forward a mission $m_e^{n:k}$ (default role if the node does not know the type e).

We define an additional generic role called Any_e that stands for any of the above roles and that we use to simplify the description of the formal model.

Rules are used to describe the interactions of a node with a mission of type e. There are 5 different rule types. For each of these rules, the 2 circles represent the role of the node before and after applying the rule. The set under the circles represents the current value of the mission $m_e^{n:k}$. The rule is applied only if the condition c is satisfied.

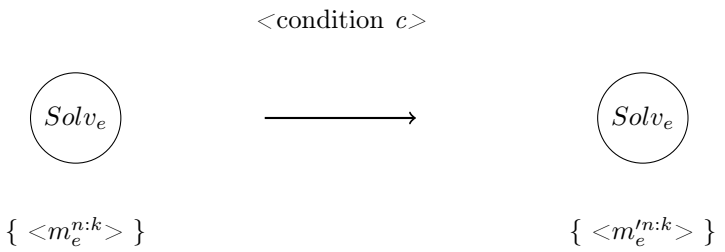
4.2 Meta-model rule types

4.2.1 Sensor meta-rule



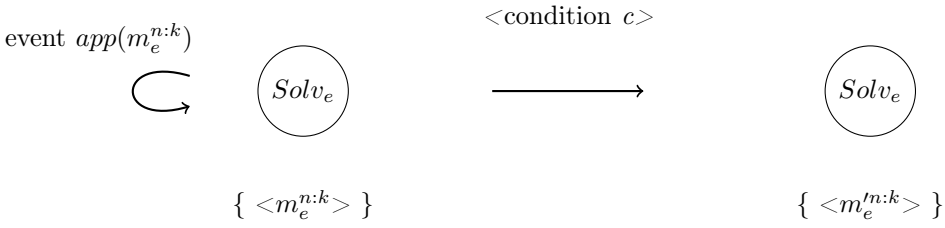
This rule is the only one that causes a new mission to be created. A $Sens_e$ node named n creates the mission $m_e^{n:k}$ if condition c is true. This condition will be used in the instantiation to check if a similar mission initiated because of the same event e is already being solved or under resolution.

4.2.2 Solver local computation meta-rule



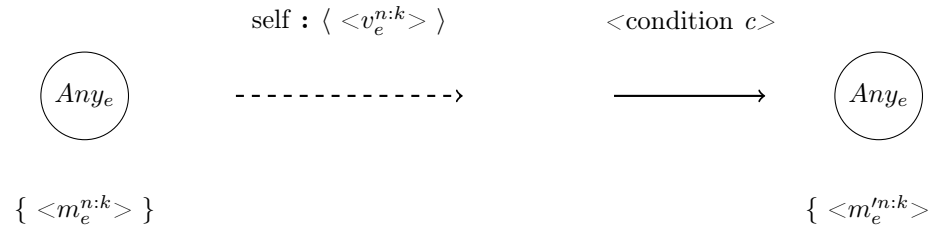
This rule enables a solver node to autonomously modify the state of a mission. After applying this rule, the mission $m_e^{n:k}$ is updated to its new version $m_e'^{n:k}$.

4.2.3 Solver local applicative event meta-rule



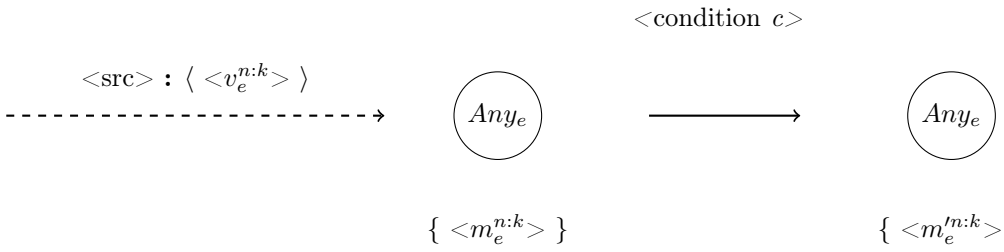
This rule enables a solver node to react to an applicative event $app(m_e^{n:k})$ (e.g., action success, action aborted, etc.) and to modify the related mission description.

4.2.4 Sending view meta-rule



This rule enables a node to broadcast a mission view. This broadcast operation is described as $self :: v_e^{n:k}$ where $self$ is the identifier of the current node and $v_e^{n:k}$ the view of the current mission $m_e^{n:k}$.

4.2.5 Receiving view meta-rule



This rule enables a node to modify a mission after being informed of a new version thereof. The received message is described as $src :: v_e^{n:k}$ where $v_e^{n:k}$ is the view of the mission $m_e^{n:k}$ received from the sender called src.

4.3 Mission description

Each mission $m_e^{n:k}$ is a 7-tuple $\{e, k, n, t, s, n', t'\}$ where e is the type of the mission (i.e., event type), k is a mission sequence number relative to its initiator node and n is the identifier of the node that has created the mission (the initiator node). The 3-tuple $\{e, k, n\}$ is the identifier of the mission. t is the date of its creation, s is its current state (i.e., *start*, *will*, etc.), n' is the identifier of the last node that updated the current state, and t' is the date of the last mission update. Dates are stored as integers representing the elapsed time since a common starting point (e.g., the well known 1970-01-01 00:00:00 UTC). The data of the mission is not represented in the formal model.

4.4 Filters

The model uses several user functions which are application dependent. We call those user functions *filters*. The complete list of filters is detailed as follows:

- $f_{ignore}^e(m_e^{n:k})$ is used to decide if a new mission should be created or not when a new event is captured by a sensor node (i.e., a similar mission was perhaps already initiated because of the same event e).
- $f_{select}^e(self : m_e^{n:k}, src : v_e^{n:k})$ enables the local node $self$ to decide if it has to leave the mission $m_e^{n:k}$ because src, which has sent the view $v_e^{n:k}$, is also taking care of the mission and is more advanced in the process.
- $f_{blind}^e(m_e^{n:k})$ is used to decide if a view should be broadcasted or not. This can help to reduce potential network traffic and collisions.
- $f_{pass}^e(m_e^{n:k})$ is used by a $Solv_e$ to decide if the mission $m_e^{n:k}$ should not be selected. This function enables to implement a mission selection scheduler and to prevent several nodes to select certain missions (e.g., battery is too weak to solve this mission, etc.).
- $f_{check}^e(x : v_e^{n:k})$ is used by nodes to ignore a view sent by the node x. This function enables to implement safety verifications or security policy (e.g., several nodes are not allowed to share or to modify certain mission types, signature required in views, etc.).

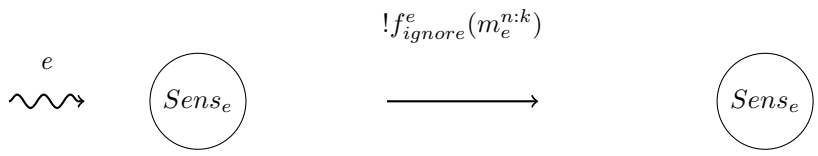
5 AMiRALE rules

We detail here all the rules of the 3 versions of AMiRALE.

5.1 AMiRALE common rules

We describe here the common explicit rules shared between the 3 versions of AMiRALE. The underline fields of the mission set under the second circle are those which have been modified by the current rule.

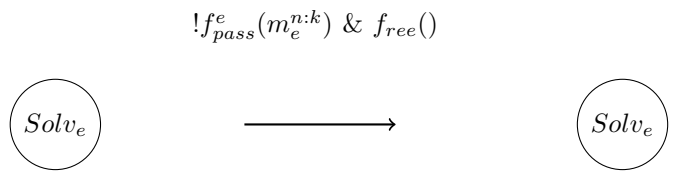
5.1.1 Sensor rule



(1) { ∅ } { e, k, self, now, start, self, now }

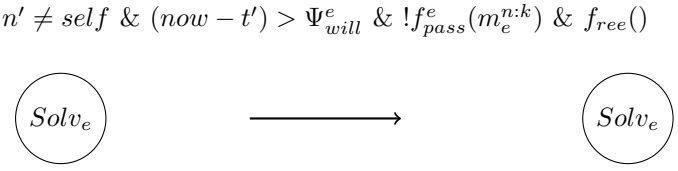
In this rule, a sensor node creates a mission relative to the event e if the filter f^e_{ignore} does not bypass the mission creation. Creation date in the mission is set to the current date (i.e., the *now* flag).

5.1.2 Solver local computation rules



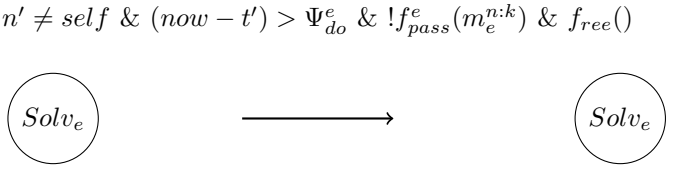
(2) { e, k n, t, start, n, t } { e, k, n, t, will, self, now }

This rule is used by a solver to take a mission in the *start* state. A solver can be in the *will* or *do* state for only one mission simultaneously (i.e., the $free$ function). Also, the solver can take the mission only if the filter f^e_{pass} does not disallow the node to take it.



(3) { e, k, n, t, will, n', t' } { e, k, n, t, will, self, now }

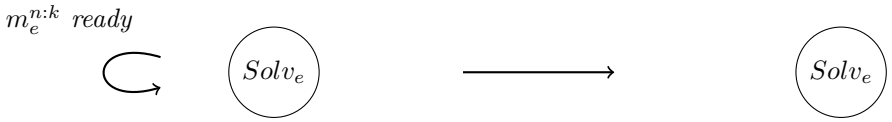
This rule is used by a solver to take a mission that has been already taken (in *will* state) but where the other solver has overstayed the allowable threshold Ψ^e_{will}. Other conditions for taking are the same as for the rule 3. A solver is not allowed to take back a mission that it is currently dealing with.



(4) { e, k, n, t, do, n', t' } { e, k, n, t, do, self, now }

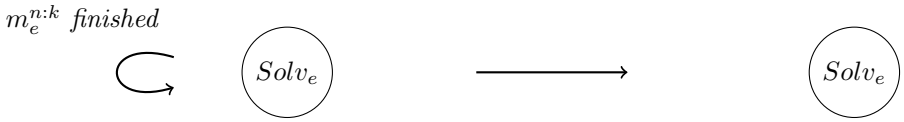
This rule is used by a solver to take a mission that has been already taken (in *do* state) but where the other solver has overstayed the allowable threshold Ψ^e_{do}. Other conditions for taking are the same as for the rule 3.

5.1.3 Solver local applicative event rules



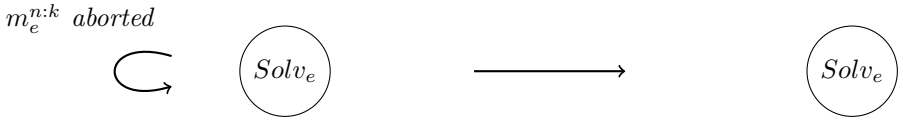
(5) { e, k, n, t, will, self, t' } { e, k, n, t, do, self, now }

This rule is used by a solver to update the *will* state to *do* when it received the applicative event indicating that the solver can immediately start the solving process.



(6) { e, k, n, t, do, self, t' } { e, k, n, t, end, self, now }

This rule is used by a solver to update the *do* state to *end* when it received the applicative event indicating that the mission is now solved.



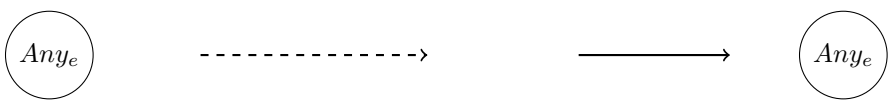
(7) { e, k, n, t, do, self, t' } { e, k, n, t, abort, self, now }

This rule is used by a solver to update the *do* state to *abort* when it received the applicative event indicating that the mission is not solvable anymore (e.g., already solved).

5.2 Global timing model

In the global timing model, views are equal to local missions; i.e., $m_e^{n:k} = v_e^{n:k}$. Indeed, nodes are supposed to be timely synchronized in this version. Thus the model can operate on absolute time information without jeopardizing the integrity of dates. The following rules round off the common rules focusing one interactions between a view and its relative mission.

5.2.1 Sending view rule of the global timing model



(8) { e, k, n, t, s, n', t' } { e, k, n, t, s, n', t' }

This rule is used to share a view with other nodes if the filter f_{blind} does not forbid the mission diffusion. This rule does not modify the content of the mission. As explained here before, in the global timing model $m_e^{n:k} = v_e^{n:k}$.

5.2.2 Solver view reception rules of the global timing model

$$n' \neq self \ \&\ f^e_{check}(n' : v^{n:k}_e) \ \&\ (now - t') < \Psi^e_{will} \ \&\ \lceil\, f^e_{select}(self : m^{n:k}_e, n' : v^{n:k}_e) = n' \parallel (now - t'') > \Psi^e_{will} \,\rfloor$$

(9) { e, k, n, t, will, self, t'' } { e, k, n, t, will, $\underline{n'}$, $\underline{t'}$ }

This rule is used to make a local election between two solvers treating the same mission, in the *will* state, at the same time. This rule is only applied if the communication is direct between the two solvers. In other words, the local election is not proceeded if the view is sent by a forwarder, a sensor node or even another solver which is not dealing with the current mission. The filter f^e_{select} is used to perform the local election (i.e, no need to use symmetric nor synchronized communication). In the case that only one solver received the view from the other, the local election mechanism can lead to two different situations:

- The solver which received the view keeps the mission. Thus, the two solvers continue to treat the mission.
- The solver which received the view leaves the mission. Thus, the second solver continue to treat the mission.

In both cases, the mission is treated by at least one solver. Therefore, we assume that the local election cannot lead a mission to be orphaned. This last assumption is true only if the filter f^e_{select} is consistent.

$$n' \neq self \ \&\ f^e_{check}(n' : v^{n:k}_e) \ \&\ (now - t') < \Psi^e_{do} \ \&\ t'' > t'$$

(10) { e, k, n, t, do, self, t'' } { e, k, n, t, do, $\underline{n'}$, $\underline{t'}$ }

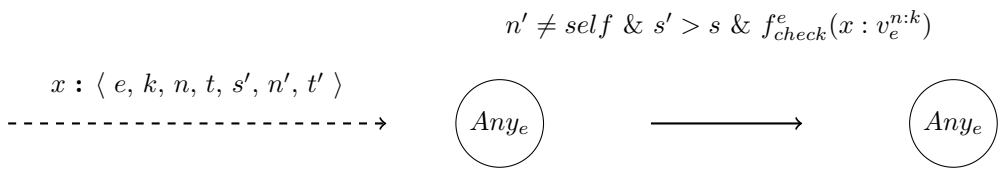
This rule is similar to the rule 9 but referring to the *do* state. Here, the difference of set times is used instead of the filter f^e_{select}. Indeed, the solver which has set the *do* state first will keep the mission if it has not exceeded the threshold Ψ^e_{do}.

5.2.3 General view reception rules of the global timing model

$$f^e_{check}(x : v^{n:k}_e)$$

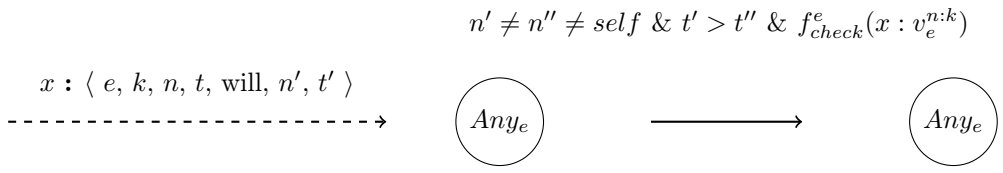
(11) { ∅ } { $\underline{e}$, $\underline{k}$, $\underline{n}$, $\underline{t}$, $\underline{s}$, $\underline{n'}$, $\underline{t'}$ }

This rule is applied to integrate an unknown mission from a view. This rule is applied only if the filter f^e_{check} does not bypass the mission integration.



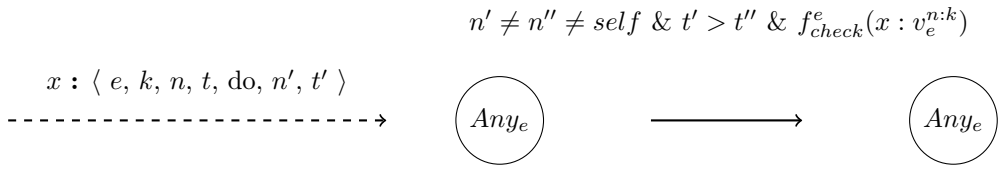
(12) { e, k, n, t, s, n'', t'' } { e, k, n, t, $\underline{s'}$, $\underline{n'}$, $\underline{t'}$ }

This rule is used to update a mission when the node receives a view indicating that the mission has evolved (i.e., state advanced).



(13) { e, k, n, t, will, n'', t'' } { e, k, n, t, will, $\underline{n'}$, $\underline{t'}$ }

This rule is used to update a mission in the *will* state when the node receives a view indicating that the solver dealing with the mission has changed. This operation is based on absolute dates. The rule is applied only if the current node is not dealing with the current mission itself in order to not collide with the rule 10.



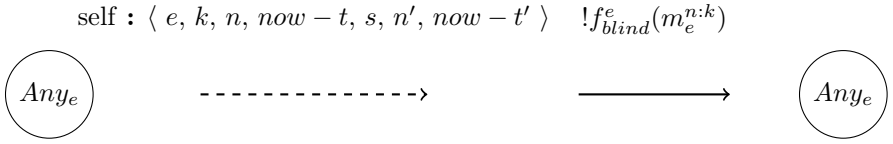
(14) { e, k, n, t, do, n'', t'' } { e, k, n, t, do, $\underline{n'}$, $\underline{t'}$ }

This rule is used to update a mission in the *do* state when the node receives a view indicating that the solver dealing with the mission has changed. This operation is also based on absolute dates. The rule is applied only if the current node is not dealing with the current mission itself in order to not collide with the rule 11.

5.3 Relative timing model

In the relative timing model, views are different from local missions; i.e., $m^{n:k}_e \neq v^{n:k}_e$. Indeed, nodes are not timely synchronized in this version. Thus the model cannot operate on absolute time information without jeopardizing the integrity of dates. Consequently, this model uses relative deviations (Δ) instead of absolute dates. In other words, views contain *delta deviation times*. Apart from this difference, rules of the relative timing model are equivalent to those of the global timing model. The following rules round off the common rules focusing one interactions between a view and its relative mission.

5.3.1 Sending view rule of the relative timing model



(8) { e, k, n, t, s, n', t' } { e, k, n, t, s, n', t' }

5.3.2 Solver view reception rule of the relative timing model

$$n' \neq self \ \& \ f^e_{check}(n' : v^{n:k}_e) \ \& \ \Delta' < \Psi^e_{will} \ \& \ \lfloor \ f^e_{select}(self : m^{n:k}_e, n' : v^{n:k}_e) = n' \parallel (now - t'') > \Psi^e_{will} \ \rfloor$$

(9) { e, k, n, t, will, self, t'' } { e, k, n, t, will, $\underline{n'}$, $\underline{now - \Delta'}$ }

$$n' \neq self \ \& \ f^e_{check}(n' : v^{n:k}_e) \ \& \ \Delta' < \Psi^e_{do} \ \& \ \Delta' > (now - t'')$$

(10) { e, k, n, t, do, self, t'' } { e, k, n, t, do, $\underline{n'}$, $\underline{now - \Delta'}$ }

5.3.3 General view reception rule of the relative timing model

$$f^e_{check}(x : v^{n:k}_e)$$

(11) { ∅ } { $\underline{e}$, $\underline{k}$, $\underline{n}$, $\underline{now - \Delta}$, $\underline{s}$, $\underline{n'}$, $\underline{now - \Delta'}$ }

$$n' \neq self \ \& \ s' > s \ \& \ f^e_{check}(x : v^{n:k}_e)$$

(12) { e, k, n, t, s, n'', t'' } { e, k, n, t, $\underline{s'}$, $\underline{n'}$, $\underline{now - \Delta'}$ }

$$n' \neq n'' \neq self \ \& \ \Delta' < (now - t'') \ \& \ f^e_{check}(x : v^{n:k}_e)$$

(13) { e, k, n, t, will, n'', t'' } { e, k, n, t, will, $\underline{n'}$, $\underline{now - \Delta'}$ }

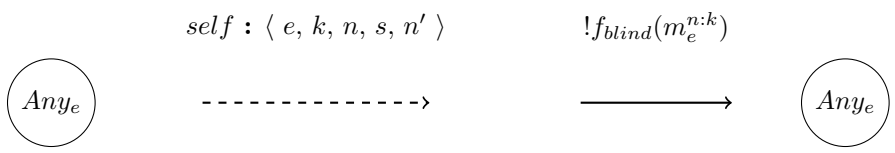
$$n' \neq n'' \neq self \ \& \ \Delta' < (now - t'') \ \& \ f^e_{check}(x : v^{n:k}_e)$$

(14) { e, k, n, t, do, n'', t'' } { e, k, n, t, do, $\underline{n'}$, $\underline{now - \Delta'}$ }

5.4 Relative ordering model

In the relative ordering model, views are a subset of local missions; i.e., $v_e^{n:k} \in m_e^{n:k}$. Indeed, nodes are not timely synchronized in this version. Thus the model cannot operate on absolute time information without jeopardizing the integrity of dates. Consequently, this model does not use any time information in views. The following rules round off the common rules focusing one interactions between a view and its relative mission.

5.4.1 Sending view rule of the relative ordering model



(8) $\{\ e, k, n, t, s, n', t'\ \}$ $\{\ e, k, n, t, s, n', t'\ \}$

5.4.2 Solver view reception rule of the relative ordering model

$$n' \neq self\ \&\ f^e_{check}(n' : v_e^{n:k})\ \&\ \lceil\ f^e_{select}(self : m_e^{n:k}, n' : v_e^{n:k}) = n'\ \|\ (now - t') > \Psi^e_{will}\ \rfloor$$

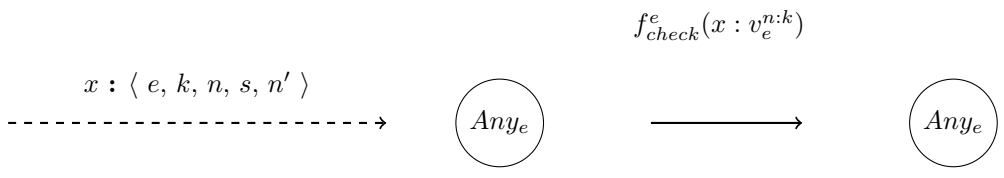
(9) $\{\ e, k, n, t, \text{will}, \text{self}, t'\ \}$ $\{\ e, k, n, t, \text{will}, \underline{n'}, \underline{\text{now}}\ \}$

This rule enables to perform the local election between two solvers in the will state for the same mission. Here, the time information is relative. This means that the respect of the threshold (i.e., Ψ^e_{will}) is computed from the date of mission integration. It means that, durations are evaluated from the date when the current node has last updated the local mission.

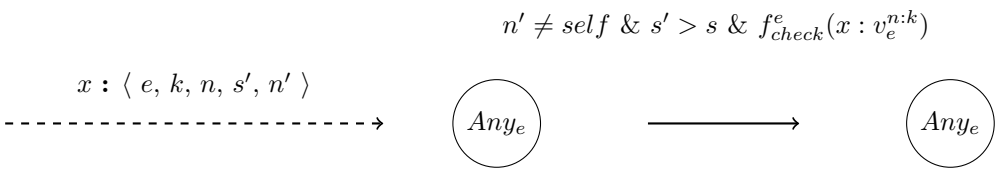
$$n' \neq self\ \&\ f^e_{check}(n' : v_e^{n:k})\ \&\ (now - t') > \Psi^e_{do}$$

(10) $\{\ e, k, n, t, \text{do}, \text{self}, t'\ \}$ $\{\ e, k, n, t, \text{do}, \underline{n'}, \underline{\text{now}}\ \}$

5.4.3 General view reception rule of the relative ordering model

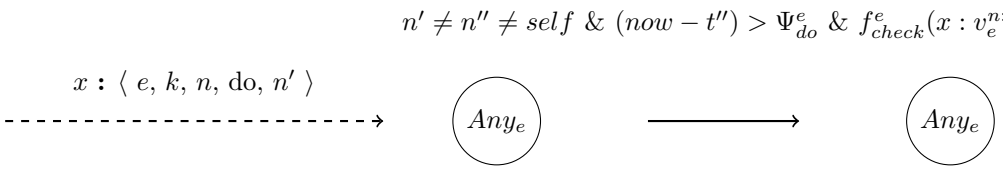


(11) $\{\ \emptyset\ \}$ $\{\ \underline{e}, \underline{k}, \underline{n}, \underline{\text{now}}, \underline{s}, \underline{n'}, \underline{\text{now}}\ \}$



(12) { e, k, n, t, s, n'', t'' } { e, k, n, t, $\underline{s'}$, $\underline{n'}$, $\underline{\text{now}}$ }

(13) { e, k, n, t, will, n'', t'' } { e, k, n, t, will, $\underline{n'}$, $\underline{\text{now}}$ }

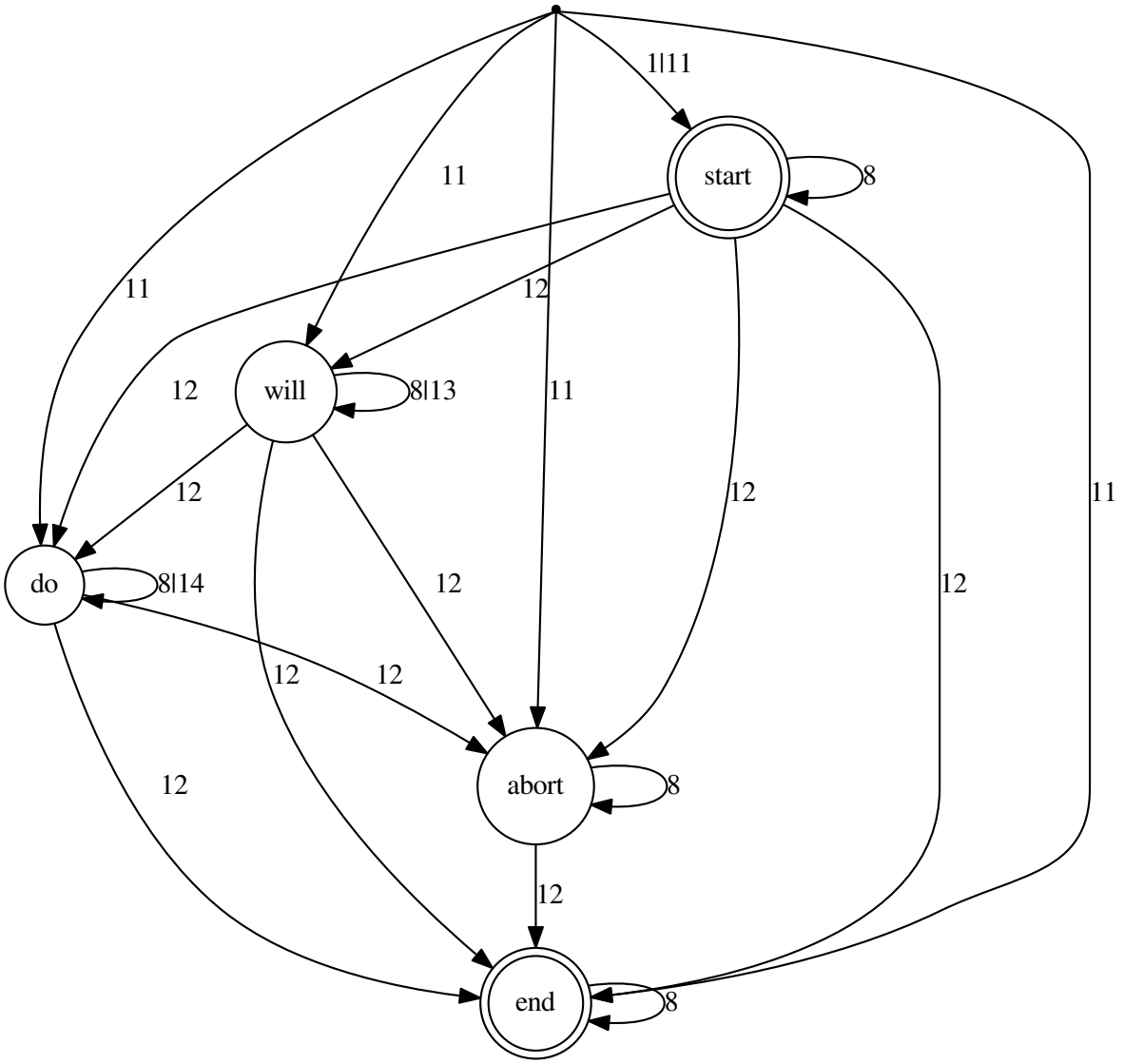


(14) { e, k, n, t, do, n'', t'' } { e, k, n, t, do, $\underline{n'}$, $\underline{\text{now}}$ }

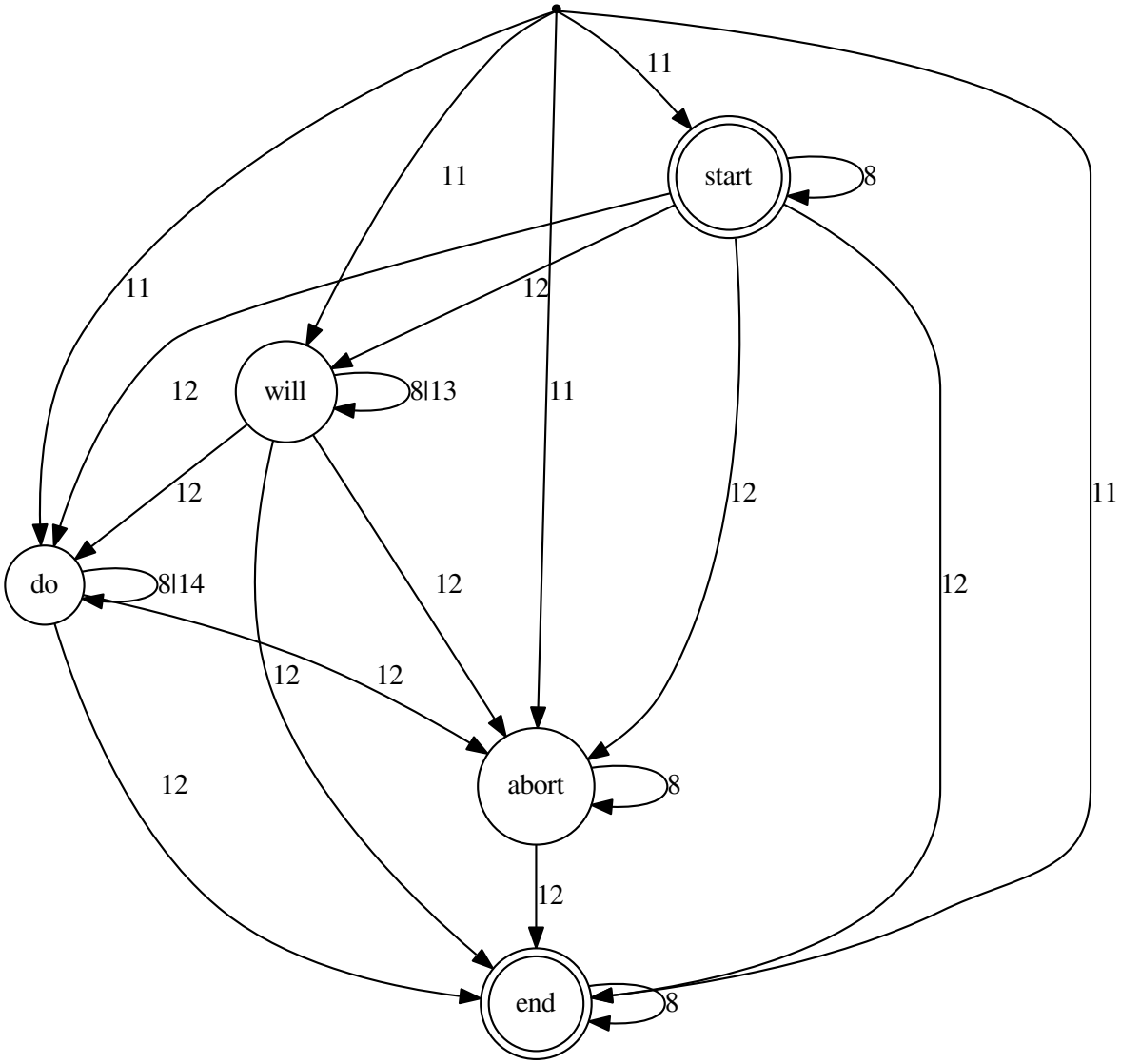
6 FSM diagrams of the state field for the various node types

In this section, we provide *finite state machines* representing the life of the state field of a mission in function of the node type. The finite state machines indicate also the rule applied in order to update the state filed.

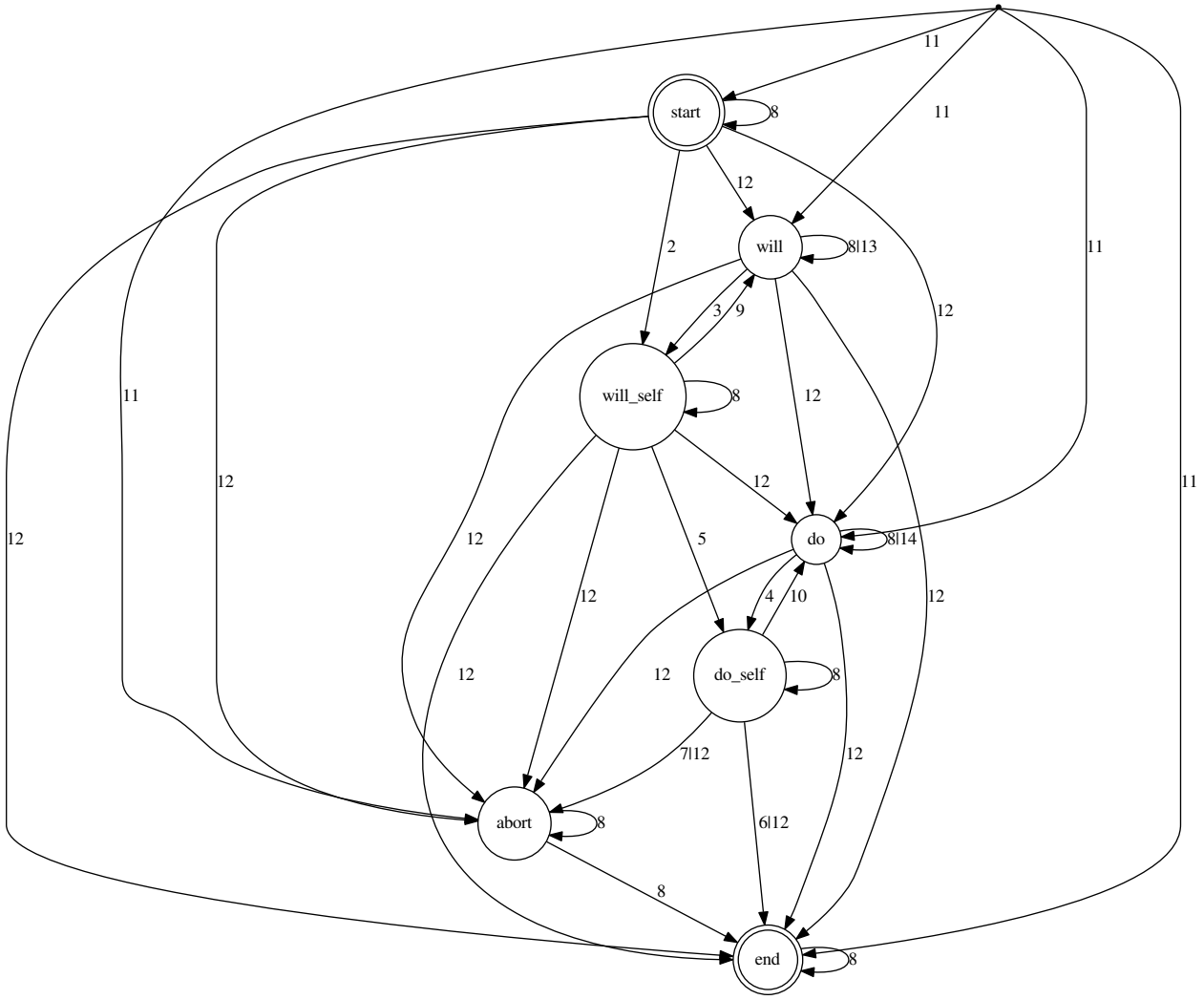
6.1 Sensor node



6.2 Forwarder node



6.3 Solver node



7 Acknowledgment

This work is co-funded by the *Direction Générale de l'Armement*[3] and the *Région Aquitaine*[4].

[3] http://www.defense.gouv.fr/dga
[4] http://aquitaine.fr