Verifying temporal properties of stigmergic collective systems using CADP

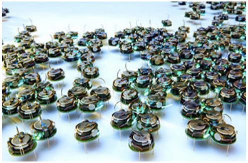
Luca Di Stefano Frédéric Lang

CONVECS, Inria/LIG, Grenoble, France
http://convecs.inria.fr

REoCAS@ISoLA - 28th October 2021

Stigmergic collective systems

- Agents have a **limited view** of the system
- Stigmergy: indirect interaction through a **shared medium**
- Collective behaviour **emerges** from stigmergy, feedback
- Can we obtain guarantees about such emergent features?



The LAbS language

LAbS is a language to describe stigmergic collective systems[1]

The shared medium (*virtual stigmergy*) is a distributed **key-value store**

Each agent has its own replica (*local stigmergy*)

Each key (*stigmergic variable*) in the local stigmergy is mapped to either a *value* with a *timestamp*, or the *undefined* value $\perp$

[1] De Nicola, Di Stefano, Inverso, *Multi-agent systems with virtual stigmergy*, Sci. Comput. Program. 187, 2020.

Stigmergic assignments and messages

An agent may assign the result of expression *e* to variable *x*

- A timestamp *t* is taken from a global clock
- In the local stigmergy, *x* is set to the result of *e* (with *t*)
- After the assignment, the agent *propagates* the new value and timestamp (`put`-message)

Whenever an agent evaluates an expression:

- It uses local values to evaluate stigmergy variables
- For every such variable *x*, it asks for *confirmation* about the value of *x* (`qry`-message)

All messages are sent **asynchronously**

Stigmergic assignments and messages

An agent may assign the result of expression *e* to variable *x*

- A timestamp *t* is taken from a global clock
- In the local stigmergy, *x* is set to the result of *e* (with *t*)
- After the assignment, the agent *propagates* the new value and timestamp (`put`-message)

Whenever an agent evaluates an expression:

- It uses local values to evaluate stigmergy variables
- For every such variable *x*, it asks for *confirmation* about the value of *x* (`qry`-message)

All messages are sent **asynchronously**

Stigmergic assignments and messages

An agent may assign the result of expression *e* to variable *x*

- A timestamp *t* is taken from a global clock
- In the local stigmergy, *x* is set to the result of *e* (with *t*)
- After the assignment, the agent *propagates* the new value and timestamp (`put`-message)

Whenever an agent evaluates an expression:

- It uses local values to evaluate stigmergy variables
- For every such variable *x*, it asks for *confirmation* about the value of *x* (`qry`-message)

All messages are sent **asynchronously**

Receiving and reacting to messages

When an agent receives a `put`-message about variable *x*:

- If the received timestamp is *lower* than the local one, nothing happens;
- Otherwise, the receiver *updates* its value and timestamp for *x* with those in the message, and asynchronously sends a `put`-message about *x* as well.

When an agent receives a `qry`-message about *x*:

- If the received timestamp is *lower* than the local one, the receiver will asynchronously send a `put`-message about *x*;
- Otherwise, same as for `put`-messages: update *x* with received value and send a `put`-message

Receiving and reacting to messages

When an agent receives a `put`-message about variable *x*:

- If the received timestamp is *lower* than the local one, nothing happens;
- Otherwise, the receiver *updates* its value and timestamp for *x* with those in the message, and asynchronously sends a `put`-message about *x* as well.

When an agent receives a `qry`-message about *x*:

- If the received timestamp is *lower* than the local one, the receiver will asynchronously send a `put`-message about *x*;
- Otherwise, same as for `put`-messages: update *x* with received value and send a `put`-message

LAbS: interaction constraints

For each variable x, the user may define a *link predicate* $\theta_x(a, b)$

Agent a may send a message about x to b iff. $\theta_x(a, b)$ holds

Examples

- Broadcast: $\theta_x(a, b)$ is always satisfied
- Ranged broadcast: $\theta_x(a, b)$ iff. a, b are close enough
- Group-based communication: $\theta_x(a, b)$ iff. a is in the same group as b

LAbS: interaction constraints

For each variable *x*, the user may define a *link predicate* $\theta_x(a, b)$

Agent *a* may send a message about *x* to *b* iff. $\theta_x(a, b)$ holds

Examples

- Broadcast: $\theta_x(a, b)$ is always satisfied
- Ranged broadcast: $\theta_x(a, b)$ iff. a, b are close enough
- Group-based communication: $\theta_x(a, b)$ iff. *a* is in the same group as *b*

LAbS leader election

Stigmergic variable *leader* storing the leader's id

Each agent follows this behavior:

$$B \triangleq leader > id \rightarrow leader \leftsquigarrow id; B$$

Eventually, agents elect the agent with lowest id

LAbS needs formal verification

Plenty of nondeterminism:

- Agents' individual behaviour may contain nondet. choices (like $P + Q$ in CCS)
- User may define a set of potential initial states
- Agents may be fully interleaved
- Asynchronous messaging
- Dynamic communication partners

Even simple systems feature vast state spaces

Quantified predicates

Each agent in LAbS has a *type*

A quantified predicate ψ describes a state by quantified variables that range over agents of a given type

Example: Consensus among voters (or lack thereof)

"All Voter agents have the same leader"

$$\forall x \in \mathrm{Voter} \bullet \forall y \in \mathrm{Voter} \bullet x.leader = y.leader$$

"There is at least one agent with different leader from everyone else"

$$\exists x \in \mathrm{Voter} \bullet \forall y \in \mathrm{Voter} \bullet x = y \vee x.leader \neq y.leader$$

ATLAS (A Temporal Logic for Agents with State)

Quantified predicates

Each agent in LAbS has a *type*

A quantified predicate ψ describes a state by quantified variables that range over agents of a given type

Example: Consensus among voters (or lack thereof)

"All Voter agents have the same leader"

$$\forall x \in \mathrm{Voter} \bullet \forall y \in \mathrm{Voter} \bullet x.leader = y.leader$$

"There is at least one agent with different leader from everyone else"

$$\exists x \in \mathrm{Voter} \bullet \forall y \in \mathrm{Voter} \bullet x = y \vee x.leader \neq y.leader$$

ATLAS (A Temporal Logic for Agents with State)

Temporal modalities

If ψ is a quantified predicate, we obtain an ATLAS temporal property by attaching a temporal modality to it

$\texttt{always}\,\psi$	All reachable states satisfy ψ
$\texttt{fairly}\,\psi$	All **fair** executions reach a state where ψ holds (We ignore unfair loops: if there is a way, the execution will eventually break out of any loop)
$\texttt{fairly}_\infty\,\psi$	All fair executions contain ∞ states where ψ holds (Equivalent to "All reachable states satisfy $\texttt{fairly}\,\psi$")

CADP

https://cadp.inria.fr

Toolbox to design and analyse formally-specified concurrent systems. We mainly use the *Evaluator* on-the-fly model checker with these languages:

- LNT Specification language influenced by process algebras and general-purpose programming languages (both functional and imperative), with LTS-based semantics;[2]
- MCL Value-passing temporal logic based on the alternation-free modal μ-calculus.[3]

---

[2] Garavel, Lang, and Serwe, *From LOTOS to LNT*, ModelEd, TestEd, TrustEd, 2017.

[3] Mateescu and Thivolle, *A Model Checking Language for Concurrent Value-Passing Systems*, FM, 2008.

MCL in a nutshell

MCL is based on **action patterns** α: `{ GATE offer(s) }` matching transition labels in the LTS

- Each `offer` is either an expression `!expr`, or a pattern `?var:Type`. These allow to *bind* values to variables and reference them later

MCL has PDL-style **modalities** $[\rho]\phi$, $\langle\rho\rangle\phi$, etc. where ρ is a *regular* formula composed of action patterns

- Examples of ρ: α; $\rho.\rho$ (sequence); α^* (Kleene star)

Fixed point operators (`nu`, `mu`) allow to characterize infinite (or finite but arbitrarily large) sub-LTSs, and can be *parameterised* in one or more variables

MCL in a nutshell

MCL is based on **action patterns** α: `{ GATE offer(s) }` matching transition labels in the LTS

- Each `offer` is either an expression `!expr`, or a pattern `?var:Type`. These allow to *bind* values to variables and reference them later

MCL has PDL-style **modalities** $[\rho]\phi$, $\langle\rho\rangle\phi$, etc. where ρ is a *regular* formula composed of action patterns

- Examples of ρ: α; $\rho.\rho$ (sequence); α^* (Kleene star)

Fixed point operators (`nu`, `mu`) allow to characterize infinite (or finite but arbitrarily large) sub-LTSs, and can be *parameterised* in one or more variables

MCL in a nutshell

MCL is based on **action patterns** α: `{ GATE offer(s) }` matching transition labels in the LTS

- Each `offer` is either an expression `!expr`, or a pattern `?var:Type`. These allow to *bind* values to variables and reference them later

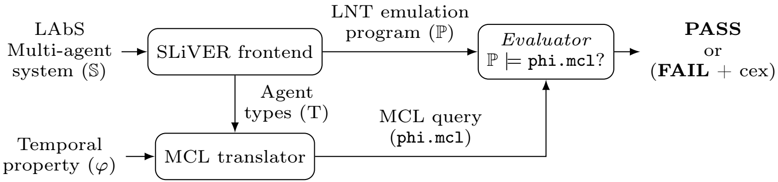
MCL has PDL-style **modalities** $[\rho]\phi$, $\langle\rho\rangle\phi$, etc. where ρ is a *regular* formula composed of action patterns

- Examples of ρ: α; $\rho.\rho$ (sequence); α^* (Kleene star)

Fixed point operators (`nu`, `mu`) allow to characterize infinite (or finite but arbitrarily large) sub-LTSs, and can be *parameterised* in one or more variables

Verification workflow

Implemented as part of the SLiVER verification tool[4]



[4] https://github.com/labs-lang/sliver

Translating predicates into MCL

Simple quantifier elimination + encoding into an MCL *macro*

Example

$$\forall x \in \mathrm{T} \bullet x.var > 0$$

Assuming that there are n agents $a_1, \ldots, a_n$ of type T:

```
macro Predicate(a1_var, a2_var, ..., an_var) =
  (a1_var > 0) and (a2_var > 0) and
  ... and (an_var > 0)
end_macro
```

ψ may involve n agents and m variables for each agent $\Longrightarrow$ `Predicate` has nm parameters

Translating predicates into MCL

Simple quantifier elimination + encoding into an MCL *macro*

Example

$$\forall x \in \mathrm{T} \bullet x.var > 0$$

Assuming that there are n agents $a_1, \ldots, a_n$ of type T:

```
macro Predicate(a1_var, a2_var, ..., an_var) =
  (a1_var > 0) and (a2_var > 0) and
  ... and (an_var > 0)
end_macro
```

ψ may involve n agents and m variables for each agent $\Longrightarrow$ `Predicate` has nm parameters

Translating predicates into MCL

Simple quantifier elimination + encoding into an MCL *macro*

Example

$$\forall x \in \mathrm{T} \bullet x.var > 0$$

Assuming that there are n agents $a_1, \ldots, a_n$ of type T:

```
macro Predicate(a1_var, a2_var, ..., an_var) =
  (a1_var > 0) and (a2_var > 0) and
  ... and (an_var > 0)
end_macro
```

ψ may involve n agents and m variables for each agent $\Longrightarrow$ `Predicate` has nm parameters

Encoding `always` ψ in MCL

```
(* Capture initial value of all relevant variables *)
[{assign !1 !"v_1" ?x̄_11:Int} . … .
 {assign !n !"v_m" ?x̄_nm:Int}]
nu Inv (x_11:Int=x̄_11, …, x_nm:Int=x̄_nm) . (
  Predicate(x_11,...,x_nm) and
  [not {assign …} or {assign to other variables}]
  Inv (x_11,...,x_nm) and
  [{assign !1 !"v_1" ?w:Int}]
  Inv (w,x_12,...,x_nm)
  and … and
  [{assign !n !"v_m" ?w:Int}]
  Inv (x_11,x_12,...,w) )
```

where {`assign` *i x v*} denotes that agent *i* sets variable *x* to *v* (either by assignment or after receiving a message)

Encoding $\texttt{fairly}_\infty\ \psi$ in MCL

First, we encode fair reachability of `Predicate` as a macro:

```
macro Reach (v11,...,vnm) =
  mu R(x11:Int=v11, …, xnm:Int=vnm) . (
  Predicate(x11,...,xnm) or
  <not {assign …} or {assign to other variables}>
  R(x11,...,xnm) or
  <{assign !1 !"v1" ?w:Int}> R(w,...,xnm)
  or … or
  <{assign !n !"vm" ?w:Int}> R(x11,...,w) )
end_macro
```

Then, we check that `Reach` is an invariant (as in the previous slide, but with `Reach` instead of `Predicate`)

Encoding `fairly` ψ is almost the same, but we “quit” as soon as `Predicate` holds

Encoding `fairly`$_{\infty}$ ψ in MCL

First, we encode fair reachability of `Predicate` as a macro:

```
macro Reach (v11,...,vnm) =
  mu R(x11:Int=v11, …, xnm:Int=vnm) . (
  Predicate(x11,...,xnm) or
  <not {assign …} or {assign to other variables}>
  R(x11,...,xnm) or
  <{assign !1 !"v1" ?w:Int}> R(w,...,xnm)
  or … or
  <{assign !n !"vm" ?w:Int}> R(x11,...,w) )
end_macro
```

Then, we check that `Reach` is an invariant (as in the previous slide, but with `Reach` instead of `Predicate`)

Encoding `fairly` ψ is almost the same, but we “quit” as soon as `Predicate` holds

Experiments

System	kStates	kTransitions	Property	Time (s)	Memory (MiB)
`formation-rr`	8786	160984	`safety`	1931	1683
			`distance`	2147	2015
`flock-rr`	58032	121581	`consensus`	3772	13792
`flock`	60122	223508	`consensus`	4375	13778
`leader5`	42	1631	`consensus0`	11	42
`leader6`	422	27874	`consensus0`	240	219
`leader7`	4439	497568	`consensus0`	3963	3164
`twophase2`	19	1125	`infcommits`	17	53
`twophase3`	291	22689	`infcommits`	849	142

- `-rr` = round-robin agent scheduling
- All properties are $\texttt{fairly}_\infty$ except `safety` (`always`)

Improvements: with a previous encoding,[5] `formation-rr` and `flock-rr` would go out of memory at 32GiB

Adding 1 agent = **10x** increase in LTS size (`leader`, `twophase`)

[5] Di Stefano, Lang, and Serwe, "Combining SLiVER with CADP to Analyze Multi-agent Systems," in COORDINATION, 2020.

Parallel emulation programs

Previous experiments refer to **sequential** programs, i.e., concurrent agents are emulated by a **scheduler** which repeatedly:

- Selects an agent and makes it perform the next action of its behaviour
- Passes a stigmergic message (if any) from its sender to all potential receivers, or

A **parallel** emulation program has an LNT process for each agent, plus some helper processes, composed using LNT's parallel composition operator `par`

Can be verified with **compositional** techniques

Parallel emulation programs

Previous experiments refer to **sequential** programs, i.e., concurrent agents are emulated by a **scheduler** which repeatedly:

- Selects an agent and makes it perform the next action of its behaviour
- Passes a stigmergic message (if any) from its sender to all potential receivers, or

A **parallel** emulation program has an LNT process for each agent, plus some helper processes, composed using LNT's parallel composition operator `par`

Can be verified with **compositional** techniques

Parallel emulation programs

Preliminary experiments

- Encode `flock-rr` as a parallel LNT program
- Generate LTS compositionally (divbranching reduction)
- Verify a $\texttt{fairly}_\infty$ property (same as sequential experiment)

Process	States	Transitions	Time (s)	Memory (kiB)
Timestamps	13	234	3	34360
Scheduler	6	12	2	34212
$Agent_1$	25637	1989572	537	3102904
$Agent_2$	25637	1989572	537	3102908
$Agent_3$	25637	1989572	538	3100408
Main	28800	74906	73	70828
Property check	–	–	2	43428
Time, max memory			1692	3102908
(Sequential)			3772	14122756

Conclusions

- Stigmergic systems involve **large state spaces**
- We can **model-check** some temporal properties about them by relying on CADP
- Fully mechanised approach: knowledge of CADP not required
- **Compositional verification** appears to be promising

Future work

- Extend ATLAS
  - Modalities (e.g., ψ Until ψ?)
  - Quantifiers (e.g., counting agents?)
- Further investigate compositional approach
- Experiment with distributed LTS generation

Conclusions

- Stigmergic systems involve **large state spaces**
- We can **model-check** some temporal properties about them by relying on CADP
- Fully mechanised approach: knowledge of CADP not required
- **Compositional verification** appears to be promising

Future work

- Extend ATLAS
  - Modalities (e.g., ψ Until ψ?)
  - Quantifiers (e.g., counting agents?)
- Further investigate compositional approach
- Experiment with distributed LTS generation

Backup slides

Local stigmergy: reading and writing

Agent i may perform **stigmergic assignments** $x \leftsquigarrow e$

Semantics:

- Get a value v by evaluating expression e
- Get a timestamp t from a global clock
- Update local stigmergy L_i so that $L_i(x) = \langle v, t \rangle$
- Add x to a set Zp_i of pending `put`-messages

For every stigmergic variable y referenced in e:

- Retrieve $L_i(y) = \langle w, \_ \rangle$ and use w to evaluate e
- Add y to a set Zq_i of pending `qry`-messages

Local stigmergy: reading and writing

Agent i may perform **stigmergic assignments** $x \rightsquigarrow e$

Semantics:

- Get a value v by evaluating expression e
- Get a timestamp t from a global clock
- Update local stigmergy L_i so that $L_i(x) = \langle v, t \rangle$
- Add x to a set Zp_i of pending `put`-messages

For every stigmergic variable y referenced in e:

- Retrieve $L_i(y) = \langle w, \_ \rangle$ and use w to evaluate e
- Add y to a set Zq_i of pending `qry`-messages

Stigmergic messages

If $x \in Zp_i$ or Zq_i, and $L_i(x) = \langle v, t \rangle$, agent i may send a message

$\langle \texttt{put}, x, v, t \rangle$ "At time t, someone set x to v"
$\langle \texttt{qry}, x, v, t \rangle$ "My value for x is $\langle v, t \rangle$, is it up-to-date?"

When an agent i receives a message $\langle (\texttt{put} \text{ or } \texttt{qry}), x, v, t \rangle$:

- Retrieve $L_i(x) = \langle v', t' \rangle$
- If $L_i(x) = \perp$, or if $t > t'$, then update L_i so that $L_i(x) = \langle v, t \rangle$, and add x to Zp_i
- If it is a $\texttt{qry}$-message and $t < t'$, add x to Zp_i

Stigmergic messages

If $x \in Zp_i$ or Zq_i, and $L_i(x) = \langle v, t \rangle$, agent i may send a message

$\langle \mathtt{put}, x, v, t \rangle$ "At time t, someone set x to v"
$\langle \mathtt{qry}, x, v, t \rangle$ "My value for x is $\langle v, t \rangle$, is it up-to-date?"

When an agent i receives a message $\langle (\mathtt{put} \text{ or } \mathtt{qry}), x, v, t \rangle$:

- Retrieve $L_i(x) = \langle v', t' \rangle$
- If $L_i(x) = \perp$, or if $t > t'$, then update L_i so that $L_i(x) = \langle v, t \rangle$, and add x to Zp_i
- If it is a $\mathtt{qry}$-message and $t < t'$, add x to Zp_i

Communication constraints

Agents send messages in an **attribute-based** multicast fashion:

- Each variable x has a predicate $\theta_x(i, j)$ over the state of two agents
- i = message sender, j = (potential) receiver
- Agents i, j may exchange a message $\langle \_, x, \_, \_ \rangle$ iff. $\theta_x(i, j)$ holds

LNT translation: "symbolic" timestamps

Previously, we represented timestamps as `Nats` taken from a global clock. But:

- The global clock must reset at some point (e.g., after reaching 255) = likely weird behavior after the reset
- Big state space

Observation:

We never need the *actual* value of a timestamp, just its *relation* with others (> < =)

Thus, we track only that relation, by using a 3-valued matrix for each stigmergy variable

LNT translation: “symbolic” timestamps

Previously, we represented timestamps as `Nats` taken from a global clock. But:

- The global clock must reset at some point (e.g., after reaching 255) = likely weird behavior after the reset
- Big state space

Observation:

We never need the *actual* value of a timestamp, just its *relation* with others (> < =)

Thus, we track only that relation, by using a 3-valued matrix for each stigmergy variable

LNT translation: “symbolic” timestamps

Previously, we represented timestamps as `Nats` taken from a global clock. But:

- The global clock must reset at some point (e.g., after reaching `255`) = likely weird behavior after the reset
- Big state space

Observation:

We never need the *actual* value of a timestamp, just its *relation* with others (> < =)

Thus, we track only that relation, by using a 3-valued matrix for each stigmergy variable

Representing timestamps symbolically

For each stigmergic variable x we define a $n \times n$ matrix M_x ($n = \#$ of agents) with this invariant:

$$M_x[i,j] = \begin{cases} 1 & \text{if } i\text{'s timestamp for } x \text{ is \textbf{greater} than } j\text{'s} \\ -1 & \text{if } i\text{'s timestamp for } x \text{ is \textbf{lower} than } j\text{'s} \\ 0 & \text{otherwise (same timestamp)} \end{cases}$$

- We can maintain M_x in our LNT program without tracking the underlying timestamps
- Furthermore, we can store all of M_x in a 1D array of length $n(n-1)/2$

Representing timestamps symbolically

For each stigmergic variable x we define a $n \times n$ matrix M_x ($n = \#$ of agents) with this invariant:

$$M_x[i,j] = \begin{cases} 1 & \text{if } i\text{'s timestamp for } x \text{ is } \textbf{greater} \text{ than } j\text{'s} \\ -1 & \text{if } i\text{'s timestamp for } x \text{ is } \textbf{lower} \text{ than } j\text{'s} \\ 0 & \text{otherwise (same timestamp)} \end{cases}$$

- We can maintain M_x in our LNT program without tracking the underlying timestamps
- Furthermore, we can store all of M_x in a 1D array of length $n(n-1)/2$

ATLAS syntax

$e ::= \kappa \mid x.var \mid e \circ e \mid \lvert e \rvert$	(value expression)
$p ::= e \bowtie e \mid x = x \mid \neg p \mid p \wedge p$	(predicate)
$\psi ::= p \mid \exists x \in \mathrm{T} \bullet \psi \mid \forall x \in \mathrm{T} \bullet \psi$	(quantified predicate)
$\phi ::= \mathtt{always}\, \psi \mid \mathtt{fairly}\, \psi \mid \mathtt{fairly}_\infty\, \psi$	(temporal property)

where $\circ \in \{+, -, \times, \ldots\}$ and $\bowtie \in \{=, <, >, \ldots\}$

From state-based to action-based logics

- ATLAS predicates on the **state** of agents
- MCL is **action-based** (predicates on transition labels)

Whenever agent *id* sets a variable *x* to a value *v*, we emit a transition with label

$$\{\texttt{assign} \quad \texttt{!}id \quad \texttt{!"}x\texttt{"} \quad \texttt{!}v\}$$

We will use these transitions to track the state of the system via appropriate MCL queries

From state-based to action-based logics

- ATLAS predicates on the **state** of agents
- MCL is **action-based** (predicates on transition labels)

Whenever agent *id* sets a variable *x* to a value *v*, we emit a transition with label

$$\{\texttt{assign} \quad \texttt{!}\mathit{id} \quad \texttt{!"}\mathit{x}\texttt{"} \quad \texttt{!}\mathit{v}\}$$

We will use these transitions to track the state of the system via appropriate MCL queries

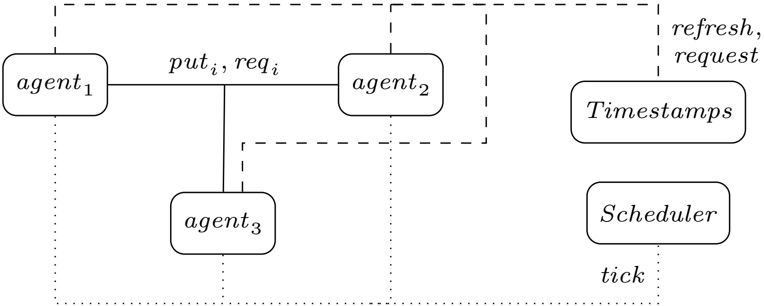
Encoding `fairly` ψ in MCL

Like $\texttt{fairly}_\infty\ \psi$: however, we "break out" of the invariance check as soon as `Predicate` is satisfied

```
[{assign !1 !"v_1" ?x̄_11:Int} . … .
 {assign !n !"v_m" ?x̄_nm:Int}]
 nu F (x_11:Int=x̄_11, …, x_nm:Int=x̄_nm) . (
 Predicate(x_11,...,x_nm) or (
     Reach(x_11,...,x_nm) and
     [not {assign …} or {assign to other variables}]
       F(x_11,...,x_nm) and
     [{assign !1 !"v_1" ?w:Int}] F(w,x_12,...,x_nm)
     and … and
     [{assign !n !"v_m" ?w:Int}] F(x_11,x_12,...,w)))
```

Parallel emulation programs

Structure



put_i, req_i	Exchange a stigmergy message
request	Ask whether my timestamp for a variable is greater/equal/less than the one of another agent
refresh	Set new timestamp for a given variable
tick	Tells an agent to perform an action