Normative Multi-Agent Programs

Mehdi Dastani
Nick Tinnemeier
John-Jules Meyer
Background and Motivation

- MAS consist of individual agents that are:
  - autonomous: pursues its own objectives.
  - heterogeneous: internal state and operations not known to external entities.

- Need for coordination of external behavior of agents to achieve MAS’s overall objectives.
Existing coordination mechanisms:

- Coordination artifacts and languages defined in terms of low-level coordination concepts such as synchronization, shared-space, and channels, e.g., REO and Linda.

- Organizational models, normative systems, and electronic institutions defined in terms of social and organizational concepts, e.g., Moise+ and ISLANDER/AMELI.

- Design and develop a **programming language** to support the implementation of coordination mechanisms in terms of normative concepts.
Simple railway simulation:

- simple railway station
- passengers travelling by train
- rules of conduct (e.g. having a ticket while on the train)

Passengers are agents that can:

- embark the train
- enter the platform
- buy a ticket
General idea of Normative Multi-Agent Organization

Organization is developed as separate entity

Agents:
- specified by 2APL programming language
- perform external actions
- internal architecture unknown to organization

Organization:
- determines effect of external actions (no autonomy)
- normatively assesses effect of agents’ actions (monitoring)
- sanctions agents’ wrongdoings (enforcement)
- prevents ending up in really bad states (regimentation)
Programming a normative multi-agent organization is to specify:

- references to *2APL agent programs*, e.g.,
  
  passenger PassProg 1

- the initial state of organization by *brute facts*, e.g.,
  
  {-at_platform, -in_train, -ticket}

- the *effects of actions*, e.g.,
  
  {-ticket} \(\rightarrow\) buy_ticket {ticket}
  {at_platform, -in_train} \(\rightarrow\) embark {-at_platform, in_train}

- the norms through *counts-as rules*, e.g.,
  
  {at_platform , -ticket} \(\rightarrow\) {viol_{ticket}}
  {in_train , -ticket} \(\rightarrow\) {viol_{⊥}}

- possible sanctions for agent’s through *sanction rules*, e.g.,
  
  {viol_{ticket}} \(\rightarrow\) {fined_{10}}
Agents: passenger PassProg 1

Facts: {-at_platform, -in_train, -ticket}

Effects: {-at_platform} enter {at_platform},
{-ticket} buy_ticket {ticket},
{at_platform, -in_train} embark {-at_platform, in_train}

Counts as rules: {at_platform , -ticket} ⇒ {viol_ticket},
{in_train , -ticket} ⇒ {viol⊥}

Sanction rules: {viol_ticket} ⇒ {fined_{10}}
Normative Programming Language (Semantics)

\[ \langle \{A_1, \ldots, A_n\}, \quad \sigma_b \subseteq P_b, \quad \sigma_n \subseteq P_n \rangle \]

- $P_b$ and $P_n$ are disjoint sets of literals
- Brute facts describe current state of the environment
- Normative facts describe normative assessment of organisation
When an agent performs an external action, the organization:

- determines new brute state based on effect rules
  - by using function $up(\alpha(i), \sigma_b)$, i.e. $\sigma'_b = up(\alpha(i), \sigma_b)$
- normatively judges this state by applying counts-as rules
  - by taking the closure of $\sigma'_b$ under rules $R_c$, i.e. $\sigma'_n = \text{Cl}^{R_c}(\sigma'_b) \setminus \sigma'_b$

Based on this normative judgment, the organization either:

- effectuates action and applies all sanction rules accordingly
  - by taking closure of $\sigma'_n$ under rules $R_s$, i.e. $S = \text{Cl}^{R_s}(\sigma'_n) \setminus \sigma'_n$
  - new brute state becomes $\sigma'_b \cup S$
- blocks the action if it would lead to a state marked by $\text{viol} \perp$
Applicable rules given set of literals $X$ and rules $R$, $\text{Appl}^R(X)$:

$$\text{Appl}^R(X) = \{ \Phi \Rightarrow \Psi \mid X \models \Phi \}$$

Closure of set of literals $X$ under rules $R$, $\text{Cl}^R(X)$:

$$\text{Cl}^R(X) = \text{Cl}^R_{m+1}(X) \text{ iff } \text{Cl}^R_{m+1}(X) = \text{Cl}^R_m(X), \text{ where}$$

- **B:** $\text{Cl}^R_0(X) = X \cup (\bigcup_{l \in \text{Appl}^R(X)} \text{cons}_l)$
- **S:** $\text{Cl}^R_{n+1}(X) = \text{Cl}^R_n(X) \cup (\bigcup_{l \in \text{Appl}^R(\text{Cl}^R_n(X))} \text{cons}_l)$

Effect of action $\alpha(i)$ specified by brute effect rule ($\Phi \alpha(i) \Phi'$):

$$\text{up}(\alpha(i), \sigma_b) = (\sigma_b \cup \Phi') \setminus (\{p \mid -p \in \Phi'\} \cup \{-p \mid p \in \Phi'\})$$
Let action $\alpha(i)$ be specified as: $(\Phi \alpha(i) \Phi')$.

Transition rule for individual agent’s external actions:

$A_i \xrightarrow{\alpha(i)} A'_i :$ agent $i$ can perform external action $\alpha$.

Transition rule for normative multi-agent organization:

$\sigma'_n = Cl^{R_{c}}(\sigma'_b) \setminus \sigma'_b \quad \sigma'_n \not\models viol_{\bot} \quad \sigma'_b = up(\alpha(i), \sigma_b) \quad S = Cl^{R_{s}}(\sigma'_n) \setminus \sigma'_n \quad \sigma'_b \cup S \not\models \bot$

\[
\langle A, \sigma_b, \sigma_n \rangle \longrightarrow \langle A', \sigma'_b \cup S, \sigma'_n \rangle
\]

where $A' = (A \setminus \{A_i\}) \cup \{A'_i\}$ and $viol_{\bot}$ is the designated literal for regimentation.
Conclusion and Future Works

- A Programming language to implement multi-agent system organization.
- Logic to verify properties of normative multi-agent program.
- Adding more social constructs such as roles, relations between roles, and contracts.
- More expressive language for norms and sanctions, e.g., use temporal, deontic operators.
- Facilitating tools for norm aware agents.
Modularity in BDI-based Agent Programming

- Modularity is an essential principle in structured programming.

- Modularization is a mechanism to structure a computer program in separate modules.

- Modularization can be used for information hiding and reusability.

- Modularization in existing BDI-based Agent programming languages is to structure an individual agent’s program in separate modules, each encapsulating cognitive components such as beliefs, goals, events, and plans that together can be used to handle specific situations.
In Jack and Jadex, modules (capabilities) encapsulate cognitive components that implement a specific capability/functionality of the agent.

The interpreter searches the modules in order to determine, e.g., how an event can be processed.
• In GOAL, modules are used as a mechanism to focus the program execution. It is to realize a specific policy or mechanism in order to control nondeterminism in agent execution.

• In 3APL a module is associated with a specific goal indicating which and how planning rules should be applied to achieve that specific goal.

• Belief or goal conditions are assigned to modules. The agent’s interpreter uses the modules when the respective conditions hold.
:main:deliveryAgent
{
  :beliefs{ home(a).
    loc(p1,a). loc(p2,a). loc(p3,a). loc(p4,a). loc(truck,a).
    loc(c1,b). loc(c2,c). order(c1,[p1,p2]). order(c2,[p3,p4]).
  }
  :goals{ delivered_order(c1). delivered_order(c2). ... } 
  :program{ ... }
  :action-spec{ ... }
  :module: deliverOrder{
    :context{ bel(order(C,O), in(O,a)), goal(delivered_order(C)) }
    :beliefs{ 
      ordered(C,P) :- order(C,Y), member(P,Y).
      ... 
    }
    :goals{ } 
    :program{ 
      if bel(ordered(C, P)), ~bel(in(P, truck)) then load(P).
      if bel(loc(truck, X), loaded_order(C), loc(C, Y)) then goto(Y).
      if bel(loc(truck, X), in(P,truck), loaded(C,P)) then unload(P).
      if bel(loc(C, X), empty, home(Y)) then goto(Y).
    }
    :action-spec{ ... }
  }
  :module: stockMgt{
    :context{ bel(ordered(C,P), empty), ~bel(in(P,a)) }
    :goals{ in(P,a) } 
    :program{ ... }
    :action-spec{ ... }
  }
  ...
}

Modularity: Our Vision

- Provides agent programmer **more control** over how and when modules are used.

- In AOSE **roles** are considered as functionalities to handle specific situations. In BDI approach, roles are specified in terms of beliefs, goals, events, and plans.

- Agent may want to construct and maintain **profiles** of users or other agents. A user or other agents can be specified in terms of beliefs, goals, events, and plans.

- Using modules for **information hiding** and **reusability**.

- Providing a set of **generic programming constructs** that can be used by an agent programmer to perform a variety of (role and profile related) operations on modules.
Extending 2APL with Modules

• A 2APL multi-agent program is implemented in terms of a set of module specifications. Each module is specified by beliefs, goals, plans, and practical reasoning rules.

• A subset of these modules is identified as the specification of individual agents constituting multi-agent system. The execution of a 2APL multi-agent program is the instantiation and execution of these modules.

• Several operations/actions can be performed on modules.

• Operations on Modules
  – Create/Release module
  – Execute module
  – Update module
  – Test module
Multi-Agent Program

\[ \langle \text{MAS}_\text{Prog} \rangle \ := \ "\text{Modules :" } \langle \text{module} \rangle^+ \\
"\text{Agents :" } (\langle \text{agentname} \rangle \ ":: \langle \text{moduleIdent} \rangle [\langle \text{int} \rangle])^+ \]

\[ \langle \text{module} \rangle \ := \ \langle \text{moduleIdent} \rangle \ "\cdot \text{2apl}" \ [\langle \text{environments} \rangle] \]
\[ \langle \text{agentname} \rangle \ := \ \langle \text{ident} \rangle \]
\[ \langle \text{moduleIdent} \rangle \ := \ \langle \text{ident} \rangle \]
\[ \langle \text{environments} \rangle \ := \ "\text{@" } \langle \text{ident} \rangle^+ \]

Modules:
   manager.2apl @clientdatabase
   admin.2apl
   userCreator.2apl @userdatabase

Agents:
   richard: manager.2apl
   administrator: admin.2apl
A 2APL Module

\[2APL\_Module\] ::= ("private" | "public") "singleton"?
   ("Include:" \[ident\])
   | "BeliefUpdates:" \[BelUpSpec\]
   | "Beliefs:" \[belief\]
   | "Goals:" \[goals\]
   | "Plans:" \[plans\]
   | "PG-rules:" \[pgrules\]
   | "PC-rules:" \[pcrules\]
   | "PR-rules:" \[prrules\])*
\[baction\] ::= ... | \[createaction\] | \[releaseaction\] | \[return\] | \[moduleaction\]
\[createaction\] ::= "create(" \[ident\]," \[ident\])"
\[releaseaction\] ::= "release(" \[ident\])"
\[return\] ::= "return"
\[moduleaction\] ::= \[ident\]." \[maction\]
\[maction\] ::= "execute(" \[test\])" | "executeasync(" \[test]? ")"
   | "stop" | \[test\] | \[adoptgoal\] | \[dropgoal\] | \[updBB\]
\[updBB\] ::= "updateBB(" \[literals\])"
Creating a Module Instance

- One module instance can create several instances of one and the same module specification.

- The creating module instance assigns a unique name to the module instance.

- A creating module instance becomes the owner of the created module instance. The creating module instance is the only module instance that can operate on the created module instance until the created module is released.
A module instance can execute another one and wait until the execution of the module instance is halted.

- A condition should be given to indicate when the execution of a module instance must halt.

A 2APL module instance can execute another one in parallel.

- The executed module instance can be halted either by means of a condition evaluated on the internals of the executed module instance, or
- Explicitly by means of a stop action performed by its owner.
Update and Test Module

- A module instance can test and update the beliefs and goals of a module instance that it owns.

- In order to control the access to the internals of a module instance, two types of modules instances are introduced:
  - A **private** module instance does not allow its owner to access to its internals. The owner can only execute it.
  - The internals of a **public** module instance are accessible to its owner module.
Singleton Modules

- One and the same module instances can be used by two different module instances. For this purpose **singleton module** is introduced.

- The ownership of a singleton module instance can be changed through **create** and **release** operations.

- The state of the singleton module instance is invariant with respect to these operations, i.e., the state of a singleton module instance is maintained after it is released and owned by another.
Creating/Release Module: Syntax

create(mod-name, mod-ident)
release(m)
mod-ident . operation

- If the module is not a singleton, then its instance will be removed/lost.
- If the module is a singleton, then its instance will be maintained (in the multi-agent system) such that it can be used by another module instance using the create action.
- A singleton module can only have one instance at a time such that it can always be accessed by means of the module name `mod-name`.
- The subsequent creation of a singleton module instance (by another module), which may be assigned a different name, will refer to the same instance of the module as when it was released by its last owner.
m.execute($test$)
return

- The execution of a module instance starts the deliberation process based on the internals of the module instance. The execution of the owning module instance halts until the execution of the owned module instance halts.
- A module instance is notified to stop its execution by a \texttt{stop!} event. The multi-agent system interpreter evaluates the test condition and sends the \texttt{stop!} event.
- The module instance that receives a \texttt{stop!} event starts a cleaning operation and sends a \texttt{return!} event back when it is ready by performing the \texttt{return} action.
- After the reception of this event, the owning module’s deliberation process is continued, after which it may decide to release the owned module instance.
Execute in Parallel Operation: Syntax

\[
m.\text{executeasync}(<test>?)
m.\text{stop}
\]

Identical to \texttt{execute} action, except that the owner instance does not have to wait until the execution of the module instance halts.

- A parallel executing module instance can be halted through the \texttt{test} argument as before, or
- By performing the \texttt{stop} action on the module instance by the owning module instance. This action will send a \texttt{stop!} event to the owned module instance.
Updating and Testing Module: Syntax

\[
m.\langle test \rangle \\
m.\text{updateBB}(\varphi) \\
m.\text{adopta}(\varphi) \\
m.\text{adoptz}(\varphi) \\
m.\text{dropgoal}(\varphi) \\
m.\text{dropsubgoals}(\varphi) \\
m.\text{dropsupergoals}(\varphi)
\]

- A module instance can test whether certain beliefs and goals are entailed by a public owned module instance \( m \) through action \( m.\text{B}(\varphi) \& G(\psi) \).
- The beliefs of a module instance \( m \) can be updated by \( m.\text{updateBB}(\varphi) \) action.
- A goal can be added to the goals of a module instance \( m \) by means of \( m.\text{adopta}(\varphi) \) and \( m.\text{adoptz}(\varphi) \) actions.
- The goals of a module instance \( m \) can be dropped by means of \( m.\text{dropgoal}(\varphi) \), \( m.\text{dropsubgoals}(\varphi) \) and \( m.\text{dropsupergoals}(\varphi) \) actions.
Example

------------- a plan in a module -------------
{
    create(userCreator, u);
    u.updateBB(user(dave, hopkins));
    u.adopta(registered(dave));
    u.execute(B(registered(dave)));
    release(u)
}

------------- userCreator module -------------

public BeliefUpdates:
    { true } AddUser(FirstName) { registered(FirstName) }

PG-Rules:
    registered(FirstName) <- user(FirstName, LastName) |
    {
        @userdatabase(adduser(FirstName, LastName));
        AddUser(FirstName)
    }

PC-Rules:
    event(stop) <- true | return
• Multi-Agent Configuration: $\langle A, \chi \rangle$, where
  – $A$ be a set of module configurations
  – $\chi$ be a set of external shared environments
• Module Configuration: $(A_i, p, r, e, \varphi)$, where
  – $A_i$ is a module instance with the unique name $i$,
  – $p$ is the name of the owner of the module instance,
  – $r$ is an identifier referring to the module specification,
  – $e$ is the execution flag, and
  – $\varphi$ is the execution stop condition
• Initial Configuration: $\langle A, \chi \rangle$, where
  – $A = \{(A_{i_1}, \text{mas}, m, t, \perp), \ldots, (A_{i_N}, \text{mas}, m, t, \perp) \mid (i : m@env_1 \ldots env_k, N) \in \text{MAS_Prog}\}$
  – $\chi = \{env_i \mid env_i \in \text{MAS_Prog}\}$
A non-singleton module instance can be created by another module instance $A_i$ if $A_i$ is in the execution mode (the execution flag equals $t$ and $A_i \not\models \varphi$) and there is no module instance with the same name already created by the same module ($\neg \exists r'', e, \varphi' : (A_{i.n}, i, r'', e, \varphi') \in A$).

$$(A_i, p, r', t, \varphi) \in A \& A_i \not\models \varphi \& A_i \xrightarrow{\text{create}(r, n)!} A'_i \& \neg \text{singleton}(r) \& \neg \exists r'', e, \varphi' : (A_{i.n}, i, r'', e, \varphi') \in A$$

$$\langle A, \chi \rangle \rightarrow \langle A', \chi \rangle$$

where $A' = (A \setminus \{(A_i, p, r', t, \varphi)\}) \cup \{(A'_i, p, r', t, \varphi), (A_{i.n}, i, r, f, \bot)\}$. 


Conclusion

- 2APL is extended with Modules.

- Agents can be added dynamically during the multi-agent system execution.

- Modules can be used to implement roles and profiles.

- `execute` and `executeasync` may not be appropriate for profile execution as it should not have consequences for the environment and other agents. We may introduced `dryrun` and `dryrunasync`.

- The notion of singleton can be generalized to allow a minimum and maximum of instances of a module that can be active at one time.

- New actions `add` and `remove` that accept plans or rules as argument.