A Simulation for Supply Chains Contract Execution

Long Tran[1], Tran Cao Son[1], Dylan Flynn[2], and Marcello Balduccini[2]

[1] New Mexico State University
[2] Saint Joseph's University

Abstract. Supply chains exhibit complex dynamics and intricate dependencies among their components, whose understanding is crucial for addressing the challenges highlighted by recent global disruptions. This paper presents a novel multi-agent system designed to simulate supply chains, linking reasoning about dynamic domains and multi-agent systems to reasoning about the high-level primitives of the NIST CPS Framework. Our approach synthesizes existing research on supply chain formalization and integrates these insights with multi-agent techniques, employing a declarative approach to model interactions and dependencies. The simulation framework models a set of autonomous agents within a partially observable environment, and whose interactions are dictated by contracts. The system dynamically reconciles agents' actions, assessing their feasibility and consequences. Based on the state of the domain, the simulation framework also draws conclusions about the high-level notions of requirements and concerns of the NIST CPS Framework, which provide a uniform and domain-agnostic vocabulary for the understanding of such complex systems as supply chains.

Keywords: Answer Set Programming, Dynamic Domains, Multi-Agent Systems, MAS Simulation, Contract Formalization, Supply Chains

1 Introduction

Recent global events have underscored the paramount importance of supply chains' resilience, significantly testing their robustness and responsiveness. Optimized historically for cost, efficiency, and speed, supply chains are inherently brittle when facing sudden shifts in demand or supply interruptions. The interdependencies within these complex networks pose significant challenges not only in terms of logistics and operational management but also to simply understand them and how they are impacted by disruptions.

Building on our previous work on formalizing supply chains and the underlying contracts [7], this paper introduces a simulation of the dynamics of supply that relies on a view of the supply chain as a multi-agent system. Central to our approach is the use of formal contracts, which define the obligations and interactions between agents within the simulation environment.

Our simulation integrates an additional knowledge layer based on the National Institute of Standards and Technology's Cyber-Physical Systems (NIST CPS) Framework [6]. The CPS Framework makes it possible to link the low-level view of contracts to the high-level aspects of stakeholders' requirements and concerns about the supply chain as a whole. By integrating a multi-agent simulation with the structured approach of the NIST CPS Framework, we aim to capture the nuanced interplay of contractual obligations, agent behaviors, and their ramifications on the supply chain, providing insights into the potential points of failure and enabling strategies to improve resilience.

The agents and the simulation environment are modeled using declarative techniques through Answer Set Programming (ASP) [9]. This approach allows for a clear specification of the logic governing agent behaviors and their contractual interactions. This paper presents not only the theoretical underpinnings of our simulation model but also a practical use case that illustrates its potential. Through these simulations, stakeholders can better understand the critical dependencies within their supply chains, evaluate the robustness of contractual arrangements, and explore strategies to enhance overall resilience.

The remainder of this paper is organized as follows. Section 2 provides background on the formalization of dynamic domains, of supply chain contracts, and of our approach to reasoning about supply chains through the lens of the CPS Framework. In Section 3, we describe the architecture of our multi-agent simulation framework. Section 4 presents a case study built on real-world supply chain information. We conclude the paper with a summary of our contributions.

2 Background

In this section, we review action language $\mathcal{B}$, the CPS ontology and reasoner, and the language for contract specification.

Action Language $\mathcal{B}$. We use the action language $\mathcal{B}$ to model the effects of actions on the state of the domain. $\mathcal{B}$ was chosen since it allows for the specification of state constraints, which play an important role in our application. An action domain in the action language $\mathcal{B}$ [8] is defined over two disjoint sets, a set of actions $\mathbf{A}$ and a set of fluents $\mathbf{F}$. A *fluent literal* is either a fluent $f \in \mathbf{F}$ or its negation $\neg f$. A *fluent formula* is a propositional formula constructed from fluent literals. An action domain is a set of laws of the following form:

$$\textit{Executability condition}: \quad \textbf{executable } a \textbf{ if } \varphi \tag{1}$$

$$\textit{Dynamic law}: \quad a \textbf{ causes } \psi \textbf{ if } \varphi \tag{2}$$

$$\textit{Static Causal Law}: \quad \psi \textbf{ if } \varphi \tag{3}$$

where ψ and φ are sets of fluent literals, representing their conjunctions,[3] and a is an action. Intuitively, an executability condition of the form (1) states that

[3] In general, ψ and φ can be fluent formulas. For the purpose of this paper, it suffices that conjunctions are considered.

a can only be executed if φ holds. (2), referred to as a *dynamic causal law*, states that ψ is caused to be true after the execution of a in any state of the world where φ is true. (3) represents a *static causal law*, i.e., a relationship between fluents. It conveys that whenever the fluent formula φ holds then so is ψ. Intuitively, an action domain specifies a transition system between states, which are interpretations of the set of fluents and satisfy the static causal laws. This transition system can be described by a transition function Φ that maps pairs of actions and states into sets of states. Given a state s and an action a, $\Phi(a, s)$ is the set of possible states that can be reached after the execution of a in s. Due to the lack of space, we omit the precise description of Φ and refer the readers to [8].

A *trajectory* over an action domain D is an alternate sequence of states and actions $\alpha = s_0 a_0 s_1 \ldots a_{n-1} s_n$, where s_i's are states and a_i's are actions and $s_{i+1} \in \Phi_D(a_i, s_i)$ for every $i = 0, \ldots, n-1$. We say that n is the length of α and s_0 is the starting state of α. Furthermore, α satisfies a fluent formula φ over the set of fluents $\mathbf{F}$, denoted by $\alpha \models \varphi$, if s_n satisfies φ.

CPS Ontology and Reasoner. An important challenge with supply chains is that stakeholders of varying backgrounds may use different terminology when discussing a supply chain and likely have different, possibly even conflicting, goals. As a "common foundation", in this paper we adopt the view of a supply chain as a large, complex CPS and leverage the NIST CPS Framework as a lens through which we can look at a supply chain. By design, the scope of the CPS Framework is very broad so that it may be adopted by a broad range of applications.

The CPS Framework provides the taxonomy and methodology for designing, building, and assuring CPS that meet the expectations and concerns of system stakeholders, including engineers, users, and the community that benefits from the system's functions. The *concerns* of the Framework are represented in a forest, where branching corresponds to the decomposition of concerns. We refer to each tree as a *concern tree* of the CPS Framework. The concerns at the roots of this forest are called *aspects*. Associated with each concern is a set of *requirements* that address the concern in question. For a concern to be satisfied, all linked requirements must be satisfied. The dependencies among concerns and between requirements and concerns can be formally represented by means of an ontology. Leveraging the ontology, tasks related to reasoning about the satisfaction of concerns can be reduced to: (a) identifying which requirements are satisfied in the current state of the system and which ones are not, and (b) propagating this information up the concern forest, ultimately determining the satisfaction of the aspects. For details on this approach, we refer the interested reader to [15]. For the purpose of this paper, it is sufficient to mention the existence of algorithms for determining whether a requirement or concern γ is satisfied given the ontology $\mathcal{O}$ and a state s (written $\mathcal{O} \cup s \models \gamma$).

Contracts Between Agents. $\mathcal{L}_c$, proposed in [7], is a language for representing and reasoning about contracts. In this language, each contract has two parts. The public part is shared between parties of the contract and includes the clauses that they agree. The private part is internal to each party and details the concerns of the party which are related to the contract.

Given two agents A and B. Assume that D_A and D_B are the action domain of A and B, respectively, i.e., D_x ($x \in \{A, B\}$) encodes the set of fluents that agent x is aware of and the actions which x can execute. We assume that A and B use the same language in encoding the fluents, i.e., a property shared between A and B will have the same representation in D_A and D_B. The public part of a contract between A and B over D_A and D_B consists of clauses of the form:

$$ref\_id: \quad agent \ \ \textbf{responsible\_for} \ \ goal \ \ \textbf{when} \ \ time\_expression \tag{4}$$

where

- *ref_id* is an identifier of the clause;
- $agent \in \{A, B\}$;
- *goal* is a fluent formula constructed over fluents appearing in $D_A \cup D_B$; and
- *time_expression* is a formula representing a time constraint of one the following forms:

$$always \mid eventually \mid per\_unit[n \ldots m] \mid by\_unit\, n \tag{5}$$

  where *unit* can be any time unit such as day, week, etc., n and m are integers, $n \leq m$, and $[n \ldots m]$ denotes the range $[n, n+1, \ldots, m]$.

Given the public part of a contract C between A and B, the private part of C for either agent A or B is represented by statements of the form

$$ref\_id: \quad \rho \tag{6}$$

where *ref_id* is a reference identifier and ρ is a requirement. It is assumed that the ontology $\mathcal{O}$ associates each requirement with one or more concerns (of the agent) from the concern forest defined in the CPS Framework, or any customized concern forest specific to the agent. Formally, a contract $\mathcal{C}$ between two agents A and B constructed over two actions domains D_A and D_B is a triple (C, P_A, P_B) where

- C is a set of clauses of the form (4); and
- P_A (resp. P_B) is a set of statements of the form (6) for A (resp. B).

Here, (C, P_A) or (C, P_B) is the contract under A or B's perspective, respectively, and it will be used by A or B to evaluate the progress of the contract. Observe that A (resp., B) does not necessarily know about P_B (resp., P_A).

Given a contract $\mathcal{C} = (C, P_A, P_B)$ between two agents A and B. The semantics of $\mathcal{L}_c$ is defined over pairs of trajectories over D_A and D_B of the form (H_A, H_B). Without the loss of generality, we will assume that H_A and H_B have the same length.

Given D_A and D_B, a *joint state s over D_A and D_B* (or, *joint state*, for short) is an interpretation over the set of fluents in $D_A \cup D_B$ that is closed with respect to the set of static causal laws in $D_A \cup D_B$. For a joint state s over $D_A \cup D_B$, by s_A (or s_B) we denote the restriction of s over the fluents in D_A (or D_B), respectively. Obviously, s_A (s_B) is a state in D_A (D_B). The truth value of a formula φ over the language of $D_A \cup D_B$ in a joint state s is defined as usual.

In the following, we say that $H_A = s_0^A a_0^A \dots s_{n-1}^A a_{n-1}^A s_n^A$ over D_A and $H_B = s_0^B a_0^B \dots s_{n-1}^B a_{n-1}^B s_n^B$ over D_B, are *compatible* if $s_i^A \cup s_i^B$ is a joint state for every $i = 0, \dots, n$.

The satisfaction of a clause is defined next. Given two compatible trajectories $H_A = s_0^A a_0^A \dots s_{n-1}^A a_{n-1}^A s_n^A$ in D_A and $H_B = s_0^B a_0^B \dots s_{n-1}^B a_{n-1}^B s_n^B$ in D_B, a clause

$$\mathit{ref\_id} : x \ \textbf{responsible\_for} \ \varphi \ \textbf{when} \ \ \mathit{time\_exp}$$

is *satisfied* by (H_A, H_B), denoted by $(H_A, H_B) \models \mathit{ref\_id}$, if

- φ is true in every $s_i = s_i^A \cup s_i^B$ for $i = 0, \dots, n$ when *time_exp* is *always*; or
- φ is true in $s_i = s_i^A \cup s_i^B$ for some $i = 0, \dots, n$ when *time_exp* is *eventual*; or
- φ is true in $s_i = s_i^A \cup s_i^B$ for $i = u, \dots, l$ when *time_exp* is *per_unit* $[u \dots l]$; or
- φ is true in $s_k = s_k^A \cup s_k^B$ when *time_exp* is *by_unit* k.

We say that *ref_id* is *violated* by (H_A, H_B) if $(H_A, H_B) \not\models \mathit{ref\_id}$. Building on the above definition, the satisfaction of a contract is defined as follows.

Definition 1. *Given two compatible trajectories $H_A = s_0^A a_0^A \dots s_{n-1}^A a_{n-1}^A s_n^A$ in D_A and $H_B = s_0^B a_0^B \dots s_{n-1}^B a_{n-1}^B s_n^B$ in D_B and a contract (C, P_A, P_B) between A and B, we say that C is satisfied by (H_A, H_B) if every clause in C is satisfied by (H_A, H_B).*

Definition 1 allows for the reasoning about the satisfaction of the public part of a contract. The satisfaction of the private part of a contract with respect an ontology $\mathcal{O}$ is defined next.

Definition 2. *Given two compatible trajectories $H_A = s_0^A a_0^A \dots s_{n-1}^A a_{n-1}^A s_n^A$ in D_A and $H_B = s_0^B a_0^B \dots s_{n-1}^B a_{n-1}^B s_n^B$ in D_B and a contract (C, P_A, P_B) between A and B. Let $X \in \{A, B\}$ and $s(P_X) = \{reg \mid \mathit{ref\_id} : reg \in P_X\}$. We say that a concern c of agent X is satisfied by (H_A, H_B) if $\mathcal{O} \cup s(P_X) \models c$.*

3 A Distributed Multi-Agent Simulator

The overall architecture of our system is given in Figure 1. Two main components of the system are the environment simulator and the CPS server (`cps`). These components are answer set programs and will be described in detail in the following subsections. The environment simulator, henceforth referred as simply the simulator, is responsible for maintaining the global state of the world and executing the actions sent from the agents, which result in changes in the global state of the world, and informing the agents about the changes that are local to the agents. Each agent is responsible for the fulfillment of its parts within his contracts, and thus, needs to take actions that are required for the satisfaction of their contracts.

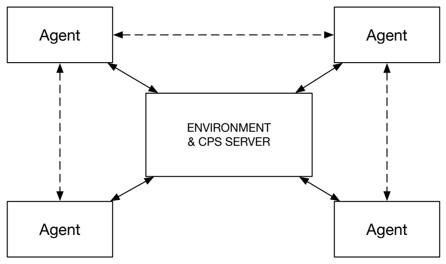

Fig. 1. System Architecture

Both the environment simulator and the agent program utilize the code for reasoning about actions and change and planning similar to the code described in [12,17]. This code defines the predicate `hold`/2 where `hold`(f, t) indicates that f is true at time step t. The initial state is encoded with a collection of atoms of the form `hold`$(f, 0)$. Rules for checking the executability condition of action or calculating the effects of an action at a time step t are defined as usual. Atoms encoding action occurrences for the simulator are of the form `occurs`(id, a, t) which says that agent id executes action a at step t. The code for the simulator is available on GitHub at `github.com/ltran1612/Research_CPS_SupplyChain`.

Communications. We used publish/subscribe architecture to facilitate the communications between agents and the environment. We observe that there is no direct communication between agents. However, this can be easily added. This is an architecture where each entity sends and receives messages by topics [16]. Communications are implemented using the MQTT protocol, a lightweight publish/subscribe messaging protocol [19]. The components in this architecture are [16]: one or more entities that communicate with each other; and a publish/subscribe broker, i.e. a middle-man server handling the communications between the entities.

The implementation of this architecture in our system included an environment, agents (each with a unique ID), and a publish/subscribe broker (Figure 2). The publish/subscribe broker is a Mosquitto MQTT broker. We used Python to implement the agents and the environment with the "paho-mqtt" package for MQTT communications. Communication between an agent and the environment relies on two topics: "env/*AgentID*" for the agent to send information to the environment, and "for/*AgentID*" for the environment to send information to the agent.

Environment Simulator. The environment simulator consists of two main parts: the controller and the reasoning engine. The controller is responsible for all the communication between the environment and the agents (e.g., receiving actions that agents execute, informing agents about the changes in the local state of agents). It also maintains the trajectories of the state of the environment. The reasoning engine is a logic program that takes a state (of the environment) and a set of actions and determines whether the actions can be executed, i.e., they are a part of compatible trajectories. In this case, the reasoning engine computes the state of the world. The environment simulator works with the global domain, denoted by D_{env}, which is the union of domains of all agents. The main portion of the code of the reasoning engine is for computing the effects of a set of actions on a state of the environment and is similar to the code for computing a plan in [12,17]. The key difference is the code only refer to two type steps, 0 and 1,

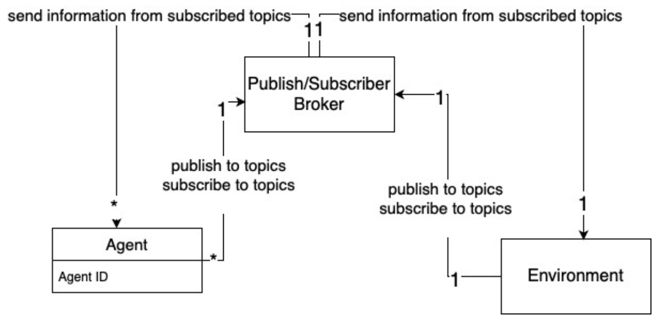

Fig. 2. The communication architecture of the system.

representing the current and the next time step. This portion of code is omitted here for brevity. Some of the rules are detailed in Section 4.

Because agents can execute their actions in parallel, we need to check for conflicts that may arise due to incompatible actions executed in parallel by different agents. We say that a set of actions X in D_{env} is *incompatible* in a state s if one of the following conditions holds:

– X contains two actions a and b whose preconditions are φ_a and φ_b, respectively, as specified by the laws (1), and $s \not\models \varphi_a \wedge \varphi_b$; or
– X contains two actions a and b and there exists some fluent literal ℓ and two dynamic laws of the form (2) such that a **causes** ℓ **if** ψ_a and b **causes** $\neg\ell$ **if** ψ_b belong to D_{env} and $s \models \psi_a \wedge \psi_b$.

The conditions are captured by: (some rules are omitted to save space)

```
incompatible(A1,A,A2,B):- action(A1,A,_), action(A2,B,_), A1!=A2,
    precondition(A1, A, S1), member(L1, S1),
    precondition(A2, B, S2), member(L2, S2),
    conflict(L1, L2).
:- occurs(A1, A, T), occurs(A2, B, T), A1!=A2,
   incompatible(A1,A,A2,B).
```

where `conflict(L,L')` is true when L and L' are contradictory[4], `incompatible(A1,A,A2,B)` means that $A1$'s and $A2$'s actions are incompatible, and the final constraint prevents incompatible actions from occurring.

Agent Program. Each agent program consists of the following components: the action domain, the set of rules for reasoning about concerns of the agent, the set of rules for reasoning about the agent's actions as well as planning, and

[4] The predicate `conflict(L,L')` can be defined for domains with static causal laws. In our case study, we need to deal with numeric fluents and thus will need some rules preventing a fluent be assigned two different values at the same time by different actions.

a controller. The controller is responsible for the communication of the agent with the environment. It registers the agent with the environment. In the first iteration, the controller sends the domain of the agent to the environment. It will then compute a plan to achieve the agent's objectives. In each iteration, it sends an action of the agent to the environment and waits for the information about the local state from the environment. It will then decide whether it should generate a new plan or continue with the execution of the old plan. The controller's pseudocode is given in Algorithm 1. The set of rules for reasoning about the agent's actions and planning is similar to the code for planning as described in [12,17] and is omitted for brevity.

Algorithm 1 Overall Behavior of the Agent Program

```
Require: agent ID, action domain, set of contracts
  registers ID with environment and wait for acknowledgement
  sends the action domain to the environment and wait for acknowledgement
  $step = 0$
  generate a plan $p$ for the set of contracts
  while true do
      send $p[step]$ (the $step$-th action of $p$) to environment
      wait for response ($local_s$ – the local state) from the environment
      if $local_s \neq \bot$ then          ▷ all actions were executed successfully
          update the current state with $local_s$
          $step = step + 1$
      else                                 ▷ some action cannot be executed
          generate a new plan $p$ for the agent
          $step = 0$                       ▷ resetting
      end if
  end while
```

Besides disallowing the execution of incompatible actions because some actions, the simulator can also disrupt the execution of a supply chain by randomly disallowing the execution of some action. This forces the agents, whose actions are not executed, to replan with the new local state.

4 Case Study: Supply Chain Simulation

As a case study, we leveraged a dataset developed by the Observatory of Economic Complexity (OEC, `https://oec.world/en`), an organization that collects and analyzes international trade data. The dataset contains data about international imports and exports, from which we extracted part of an automotive supply chain.

The scenario involves 8 agents with 7 contracts containing a total of 21 clauses, 8 supply chain requirements, and 5 concerns. The agents and their contracts are depicted in Figure 3. For example, the '*Car Producer*', denoted by `x`,

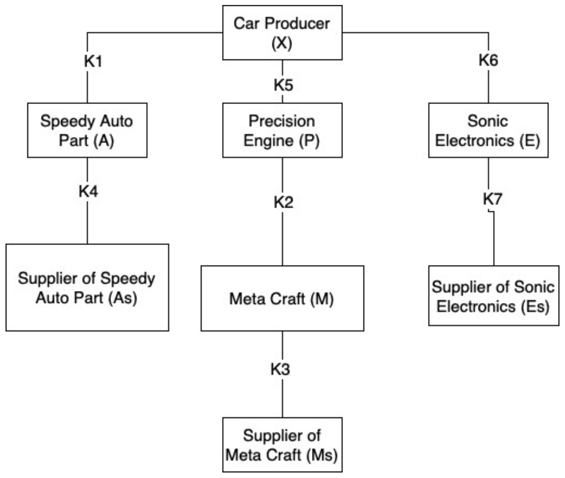

Fig. 3. The agents and their contracts (K1-K7) in OEC.

has contract K1 with the 'Speed Auto Part' (`A`), who in turn has contract K4 with the 'Supplier of Speed Auto Part' (`As`), etc. The contracts between `A` and `x` and `A` and `As` specify three clauses that `A` is responsible for:

```
C1: A responsible_for produced(vehicle_parts, 9K) when by_week 4
C2: A responsible_for delivered(vehicle_parts, 9K) when by_week 4
C12: A responsible_for payment(tool_parts, 30K) when by_week 4
```

`C1` and `C2` belong to K1 and `C12` belongs to K4. `C1`, for example, says that `A` is responsible for producing and delivering 9,000 vehicle parts to `x` by week 4.

The private parts of `C1` and `C2` (for agent `A`) maps these clauses to various requirements, which in turn address concerns of the CPS ontology:

```
C1: material-safe-for-production
C1: product-sufficiently-durable
C1: on-time-production
C2: on-time-delivery
addressedBy("cpsf:Material_Safe_For_Production", "C1").
addressedBy("cpsf:Product_Sufficiently_Durable", "C1").
addressedBy("cpsf:On_Time_Production", "C1").
addressedBy("cpsf:On_Time_Delivery", "C2").
```

`C1` addresses there requirements: safe material for production, sufficient durable product, and on time delivery. The requirements and concerns in this scenario are depicted in Figure 4.

In this scenario, each agent can perform 4 types of actions: produce, deliver, pay, and receive. Furthermore, each agent keeps track of:

1. The total amount of payments and items sent/delivered/received.

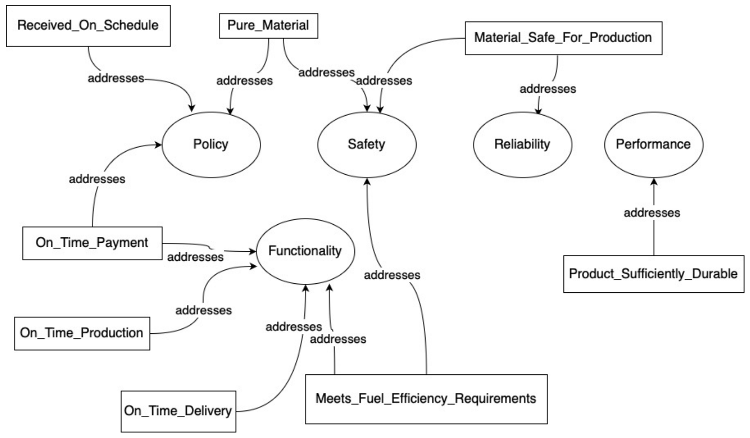

Fig. 4. Requirements (squares) and related concerns (ovals) for the OEC supply chain.

2. The individual payments and amounts of items sent/delivered/received.
3. The available amounts of specific items.
4. The available amount of funds.

The corresponding domains are encoded using a template. For example, action *pay* from `A` and related fluents are specified by rules including:

```
produce_item(speedy_auto_parts, vehicle_parts).
deliver_item(speedy_auto_parts, car_producer, vehicle_parts).
recv_item(speedy_auto_parts,
          supplier_of_speedy_auto_parts, tool_parts, 3..5).
action(Agent, pay, (ToAgent, Reason, Amount)) :- number(Amount),
       paying_reason(Agent, ToAgent, Reason).
causes(Agent, pay, Value, available_funds, decrease, ()):-
       action(Agent, pay, Value).
executable(Agent, pay, Value, enough_funds) :- action(Agent, pay,
↪  Value).
satisfy_condition(Agent, enough_funds, greater, (pay, Value,
↪  available_funds)), action(Agent, pay, Value).
```

The first three lines specify the 'fluents' that are related to `A` such as `produce_item(speedy_auto_parts, vehicle_parts)`, etc.. The next four lines define the action `pay`. When the action is executed, an amount (`Amount`) is paid to an agent (`ToAgent`) and for a reason (`Reason`). This action can only be executed when the agent has sufficient fund. `executable(...)` atoms represent this executability; an action `pay` with value `Value` (Value is `(ToAgent, Reason, Amount)`) is only executable when the conditional fluent `enough_funds` linked with the action is true. A `satisfy_condition` atom then describes that `enough_funds` linked to action `pay` with value `Value` is true/satisfied when the conditional operation `greater` outputs `true` with the input including the action's name, value, and the fluent to check the amount of funds.

A sample run of the simulator with the OEC agents progresses as follows. At first, the agents register with the environment and send their information (action domains and initial state) to the environment. At each step i, the agents send to the environment the actions they plan to execute. The environment computes the next state, sending back to the agents their local states, and waits for another set of actions. For example, at step 0, `A` has 0 `vehicle_part` and plans to take the action that produces 9,000 parts. The global state of the environment records at step 1 that `A` now has 9,000 `vehicle_part`, which is reflected in the local state of `A` as well. At the end of each iteration, the system determines which clauses are satisfied. For example, at step 0 no clauses are satisfied; at step 1, clause $C1$ is satisfied; at step 2, clauses $C1$ and $C2$ are satisfied. In turn, the system uses this information to infer which requirements and concerns are satisfied.

5 Related Work

Due to space considerations, only the most relevant research in this vast area is discussed here. The ultimate goal of a supply chain is to transform raw materials into products that will be distributed to end consumers or customers. Entities within the network need to collaborate with each other and create wealth for themselves as well as others. For this reason, supply chains have been a critical part of the economy [1,13].

The focus on cost optimization and efficiency in supply chains has also traditionally not focused on intangibles such as reputation of relationships within a supply chain [18]. Alternative models of supply chains have been proposed. For example, it is proposed that the concept of actor-network theory be used to formalize supply chains in [10]; this concept advocates the view that supply chains are composed by agents whose actions change the state of affairs.

The use of multi-agent systems in supply chain management has been explored by various researchers (see, e.g., [5,14,11]). However, the integration of formalized contracts and of the NIST CPS Framework within these systems presents a unique angle of research. Our work extends these concepts by incorporating advanced contract formalization techniques, as also discussed in [7].

Further, the use of declarative programming techniques such as Answer Set Programming (ASP) for creating our framework is inspired by the advancements in logic programming that facilitate complex decision-making processes, as explored for example by [12]. The integration of the NIST CPS Framework aligns with previous efforts [15] to apply structured, standardized approaches to the evaluation of cyber-physical interactions in industrial settings.

Last but not least, the proposed system is also related to platforms that support the simulation or programming of multi-agent systems such as NetLogo[5], Jason[6], and DALI [3,2]. It is worth pointing out that we focus on the development

[5] http://ccl.northwestern.edu/netlogo/

[6] `jason-lang.github.io/`

of a simulation platform, with supply chain as a target application. To the best of our knowledge, NetLogo supports the simulation of reactive agents while and Jason focuses on BDI[7] agents. For this types of agents, it is usually required that the agent (or the agent program) knows what should be executed given a state. In addition, planning and diagnosis are not a focus of these approaches. In particular, the agent often does not have the ability to diagnose at all, or this ability needs to be integrated within the agent program. Our agents use traditional planning/diagnosis modules developed in ASP. For this reason, these simulation environments are not suitable for our purpose.

DALI [3,2] is a superset of Prolog and requires Sicstus. While Prolog is known for its support of declarative specifications, it has difficulty in applications where cyclic dependencies are present. Reasoning about actions and changes, especially with static causal laws, requires this feature. Furthermore, Prolog is not an easy language to master for non-experts. On the other hand, our approach is focused on languages that are higher level, have simpler semantics, and are closer to natural language in order to make the languages more immediately usable by users who are not familiar with logic programming. This is the case for both the formalization of contracts and the representation of the dynamic domains. For example, when it comes to the formalization of contracts, DALI does not provide an explicit, dedicated formalization of responsibilities, deadlines, and of the links to the concern tree. Similarly, for the representation of dynamic domains, we build on action languages, which were specifically developed as a higher-level alternative to direct representation via logic programming.

6 Conclusions

In this paper, we presented a novel multi-agent simulation framework designed to address the complexities and challenges inherent in modern supply chains. We demonstrate that standard ASP programs such as planning or diagnosing modules can be integrated into a system for monitoring contract executions. By integrating the NIST CPS Framework with advanced contract formalization techniques, we have developed a robust system capable of simulating diverse supply chain scenarios and assessing the impact of various types of disruptions. Our evaluation on a realistic case study demonstrates that the framework not only enhances the understanding of supply chain dynamics but also provides actionable insights into improving resilience and reliability.

It is understandable that a system utilizing ASP technology will inherit all of its problems (e.g., grounding, scalability). However, the prototype works fine with the use cases and appears acceptable to the owner of the use cases. Identifying the limit of the current system in terms of the limit of the system (e.g., the complexity of the domains or contracts) is an interesting issue and is our immediate future work.

[7] The **B**elief-**D**esire-**I**ntention architecture includes reasoning about beliefs (updating beliefs given an observed event), goals (deciding on what to achieve given the desires), and plans (how to achieve the goal) [4].

Acknowledgement. Portions of this publication and research effort are made possible through the help and support of NIST via cooperative agreement 70NANB21H167.

References

1. K Alicke and B Iyer. Next generation supply chain: Supply chain 2020. McKinsey & Company, 2013.
2. Stefania Costantini, Giovanni De Gasperis, Valentina Pitoni, and Agnese Salutari. DALI: A multi agent system framework for the web, cognitive robotic and complex event processing. In *CILC, 26-28 September 2017, Naples, Italy*, 2017.
3. Stefania Costantini and Arianna Tocchio. A logic programming language for multi-agent systems. In *LNAI, Springer Berlin Heidelberg*, pages 1–13, 2002.
4. Lavindra de Silva, Felipe Meneguzzi, and Brian Logan. BDI agent architectures: A survey. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4914–4921. ijcai.org, 2020.
5. Roberto Dominguez and Salvatore Cannella. Insights on Multi-Agent Systems Applications for Supply Chain Management. *Sustainability*, 12(5), 2020.
6. Edward, Christopher Greer, David A. Wollman, and Martin J. Burns. Framework for cyber-physical systems: volume 1, overview. 2017.
7. Dylan Flynn, Chasity Nadeau, Jeannine Shantz, Marcello Balduccini, Tran Cao Son, and Edward Griffor. Formalizing and Reasoning about Supply Chain Contracts between Agents. In *25th PADL*, volume 13880, 2023.
8. M. Gelfond and V. Lifschitz. Action Languages. *Electronic Transactions on Artificial Intelligence*, 3(6), 1998.
9. Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
10. Kim Sundtoft Hald and Martin Spring. Actor–network theory: A novel approach to supply chain management theory development. *Journal of Supply Chain Management*, 59(2):87–105, 2023.
11. Keonsoo Lee, Wonil Kim, and Minkoo Kim. *Supply Chain Management using Multi-agent System*, pages 215–225. Springer Verlag, Berlin, 2004.
12. Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138:39–54, 2002.
13. LX Lu and JM Swaminathan. *Supply Chain Management*, pages 709–713. Elsevier, 2015.
14. Thierry Moyaux, Brahim Chaib-draa, and Sophie D'Amours. *Supply Chain Management and Multiagent Systems: An Overview*, pages 1–27. Springer Verlag, Berlin, 2006.
15. Thanh Hai Nguyen, Matthew Bundas, Tran Cao Son, Marcello Balduccini, Kathleen Campbell Garwood, and Edward R. Griffor. Specifying and reasoning about CPS through the lens of the NIST CPS framework. *TPLP*, 2022.
16. B. Reselman. The pros and cons of the pub-sub architecture pattern. Red Hat, 2021.
17. Tran Cao Son, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub. Answer set planning: A survey. *TPLP*, page 1?73, 2022.
18. SM Wagner, LS Coley, and E Lindemann. Effects of suppliers' reputation on the future of buyer–supplier relationships: The mediating roles of outcome fairness and trust. *Journal of Supply Chain Management*, 47(2):29–48, 2011.
19. M. Yuan. Getting to know mqtt. IBM, 2021.