

Smart Contracts for Multiagent Plan Execution in Untrusted Cyber-physical Systems

Anshu Shukla, Swarup Kumar Mohalik, Ramamurthy Badrinath
Ericsson Research, Bangalore, India

Email: {anshu.shukla, swarup.kumar.mohalik, ramamurthy.badrinath}@ericsson.com

Abstract

Intelligent Cyber-physical systems can be modeled as multi-agent systems with planning capability to impart adaptivity for changing contexts. In such multi-agent systems, the protocol for plan execution must result in the proper completion and ordering of actions in spite of their distributed execution. However, in untrusted scenarios, there is a possibility of agents not respecting the protocol either due to faults or due to malicious reasons thereby resulting in plan failure. In order to prevent such situations, we propose to implement the execution of agents through smart contracts. This points to a generic architecture seamlessly integrating intelligent planning-based CPS and smart-contracts.

Keywords

Artificial Intelligence; Planning; Multi-agent; Blockchain; Smart Contract; Cyber-physical systems;

1. Introduction

Cyber-physical systems (CPS) comprise sensing and actuation capabilities controlled typically through software agents. A view of intelligence in these systems attempts to equip the systems with self-adaptivity so that the systems can adapt to dynamic changes in the environment and system capabilities automatically. Such self-adaptivity can be imparted by integrating artificial intelligence (AI) planning subsystems in the software agents. Frameworks along this philosophy have been studied in the autonomous systems literature [1]

Extending the above approach to distributed CPS, one naturally has multi-agent planning based systems where both plan synthesis and execution need coordination between the agents. The plans are maintained in data structures that can be global and is orchestrated by a central agent, or in local data structures and are executed through agreed protocols between the agents. These are the central themes of multi-agent planning literature [2]. In this paper, we assume that the plans have already been synthesized (global or local) and concern ourselves only with the execution phase.

A hidden assumption in multi-agent planning is that once the plan is synthesized and the protocol has been agreed upon, the agents execute strictly according to the protocol. However, in an untrusted environment, this assumption may not be tenable. Either due to malicious reasons or selfishness, or due to faults, the agents may not be able to adhere to the protocols, which may lead to one or more number of anomalies such as out-of-order execution, untimely initiation

and improper completion of actions etc. In critical CPS, the cost of resulting failures being very high, it is necessary to devise a mechanism where agents cannot deviate from the established and agreed-upon protocol.

Towards this, we propose a multi-agent plan execution framework based upon Smart contracts on blockchains. The design of smart contracts for the plans along with the properties transparency, authenticity and immutability enforced by blockchains ensure the desired execution of plans in spite of the untrusted environment. The structure of the smart contracts is dependent upon the multi-agent plan execution architecture (centralized or decentralized execution).

To the best of our knowledge, this is the most direct application of smart contracts and blockchains in AI planning. So far, blockchains have been used mainly to store data, AI models and algorithms to ensure integrity in untrusted environments. Controlling the plan execution via validated smart contract transactions seems to be a completely novel contribution.

The paper is organized as follows. In the following section, we give some basic introduction to planning, multi-agent planning, blockchains and smart contracts. Section 3 describes the system model formally and outlines the issues of plan execution in untrusted environments. We give the solution based on smart contracts in Section 4. The review of literature in the intersection of smart contracts AI is given in Section 5. Section 6 concludes the paper with a summary and number of possible extensions.

2. Background

Cyber-physical systems(CPS) consist of a number of physical *devices* that can be controlled by one or more *agents*. Typically, the devices expose a set of actions through an API that is utilized by the agents. In autonomous CPS, the agents are intelligent i.e. they adapt to changes in the environment and/or requirements and control the devices differently to achieve changed goals. In AI literature, planning - plan synthesis and execution - is one of the approaches of endowing agents with intelligence.

2.1. Planning

AI Planning is an umbrella of techniques which derives a sequence of actions (called a Plan) that can lead from an

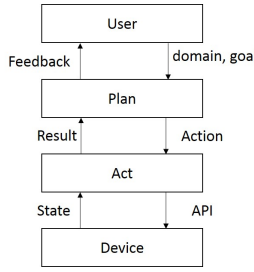


Figure 1: Architecture of planning based intelligent agent

initial state to a specified goal state. Each *state* is described by a set of predicates about the objects of the world and their relationships. *Actions* describe the change they cause to a state of world. They are specified through $\langle precondition, effect \rangle$ pairs that are boolean combination of predicates. An action is enabled in a state only when its precondition is satisfied in that state. The execution of the action leads to a new state where its effect holds.

A *planning domain* specifies the predicates (and functions) necessary to capture the states of the domain under consideration. It also specifies the actions through their precondition and effect pairs. On the other hand, a *planning problem* captures the planning requirements through a specific state of the domain as initial state and a specific goal state. For more details on planning see [3].

AI planning involves plan synthesis and plan execution. The typical architecture of an intelligent agent is given by the basic sense-plan-act loop or its variations, such as MAPE - K [1]. Fig. 1 gives the schema of a bare-boned planning based intelligent agent. Note that the *plan* component may just acquire a plan, say from a user or plan database, instead of having to actually compute it.

Planning algorithms derive *plans*, which are a partially ordered set of nodes labeled with actions. A plan is said to be *properly* executed when for each action a in the partial order, (1) a is executed only when the actions on which it is dependent - denoted $dep(a)$ - have already completed execution, (2) the pre-conditions of a are satisfied and (3) when a completes execution, the effects of a hold true. Otherwise one could conclude that either the action did not execute properly or the modeling of the effects of the action was wrong. Essentially, the actions in the partial order plan should execute in a topological order with additional constraints imposed by preconditions and effects.

2.2. Multi-agent Planning

Multi-agent planning is a natural extension of the basic planning problem where there are more than one agent. The extension gives rise to a number of variations such as: whether the agents are cooperative or competitive, whether planning is centralized or not, whether communication is reliable or not [2] etc.

The basic problem in multi-agent agent planning is the distribution of plans over the agents. These plans must be

executed in a coordinated manner so as to achieve the system goals. Therefore, the agents must follow certain protocols that are part of the plan generation.

In the present paper, we address two execution architectures for multi-agent planning. The first one is a centralized architecture where a *scheduler* agent has the full plan and orchestrates the execution by dispatching them to the agents in a timely manner (Fig. 3a). The second one is where each agent acquires its plan and the execution is done in a coordinated manner through communication between the agents (Fig. 3b).

2.3. Blockchain & Smart Contract

A *blockchain* is a distributed, decentralized ledger that maintains a common set of updated and secure records of transactions across many computers based on peer-to-peer networks and cryptographic algorithms. Each peer in the network validates the transaction against the rules set in the specific blockchain network [4], [5], [6]. A certain number of valid transactions are grouped to form a *block*. Each block has a reference to the preceding block in the blockchain allowing the participants to confirm the integrity of the blocks all the way back to the original genesis block. A copy of blockchain is stored on every machine participating in the blockchain network and is synchronized among the peers continuously. Thus, it is a permanent ledger that all peers on the network use for verification, coordinating actions and to reach an agreement in an auditable manner.

A smart contract [7] is essentially a collection of code and data representing some business logic running on a blockchain. The smart contract resides at specific address on the blockchain. Solidity [8] is one of the commonly used languages for coding smart contracts. The basic example for smart contract can be just a data update operation on blockchain; e.g., updating the account balance after validating that account has enough money before debiting it. As a more complex example, in the smart logistics domain one may dynamically adjust the shipping cost based on the time of the request. The ledger will have agreed terms by both parties coded as smart contracts. The appropriate funds incorporating the dynamic delivery charges are transferred automatically based on delivery time of item.

3. System Model

In this section, we describe the formal model of the multi-agent planning based CPS and define the problems to be addressed.

The CPS under consideration consists of a set of devices \mathcal{D} where each device d is associated with a number of actions Act_d exposed through a REST-based API. We denote the set of all the actions from the devices as Act . The REST-based API abstracts the underlying communication and device libraries and is not relevant to the scope of the present paper. We denote by $execute(a)$ a call to the API that executes an action a on the corresponding device.

Each device d has a number of sensors collecting data about the state of the device. We model the state of the devices via a set of predicates whose values can be decided by processing the sensor values. The set of all predicates is denoted as \mathcal{P} and the predicates associated with a device d is denoted as \mathcal{P}_d . Similar to the actions, the values of the predicates are assumed to be accessible through a REST-based API. We denote by $getVal(p)$ the REST-based call to get the value of the predicate p .

One or more devices have an associated *agent* which is implemented in software. The set of all agents is denoted as AG . The spatial distribution function $loc : Act \rightarrow AG$ maps each action of a device to the associated agent. At the basic level, each *agent* can call the $getVal(\cdot)$ API to access the value of a predicate and call $execute(\cdot)$ to execute an action in a device. Moreover, each *agent* has the mechanisms to communicate with each other or with a separate central agent to exchange information. For the purpose of the paper, the information to be exchanged is essentially about the completion of an action by the device.

To assist automatic synthesis and control plan execution, the actions are annotated with two sets of predicates: *pre-condition*, which should be true for the action to be enabled and *effect*, which must hold as a result of action execution. Given an action a , we denote the pre-condition and effect of the action as $precond(a)$ and $effect(a)$ respectively.

An initial state *init* is specified as a conjunction of state predicates from \mathcal{P} and a *goal* state is specified as a boolean expression over \mathcal{P} as well. A plan for the problem (*init*, *goal*) is derived as a partial order over Act .

A proper execution of the plan starts at the *init* state, executes the actions in the plan in topological order and finally results in a state satisfying *goal*. The agents check the precondition and effect constraints by accessing the predicate values through $getVal(\cdot)$. The actual execution of the actions is done through the $execute(\cdot)$ calls. In addition, for each action a to be executed by the agent $loc(a)$, the agent needs to know that the actions in $dep(a)$ have already completed execution. Since these actions in $dep(a)$ might be with other agents, the action completion events must be notified through the communication mechanism between agents. This leads to two natural architectures for distributed plan execution.

3.1. Distributed Plan Execution in Centralized Mode

In the centralized mode of plan execution, there is an external *Scheduler* agent who schedules the execution of actions taking into account $dep(\cdot)$ of actions. The device agents are only responsible for the verification of precondition and effects and dispatch of actions through the action APIs. The general schema is depicted in Fig. 3a.

The Scheduler implements the topological order of the plan by maintaining a datastructure *completedActionsList* to record the completion of execution of actions as notified by the agents. When the dependencies of an action a is detected to be fulfilled by inspecting the *completedActionsList*,

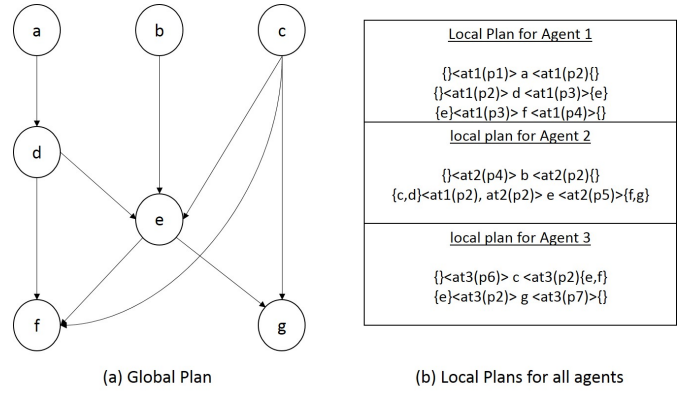


Figure 2: Global and local plans

Scheduler sends a request for dispatch to the agent $loc(a)$. The device agent accesses the state of the devices through the call $getVal(precond(a))$ and if it is true, issues the $execute(a)$ call. When the action is completed successfully (result of the $execute()$ call), the agent sends the completion message to the Scheduler which updates the *completedActionsList*. The interactions continue till all actions in the plan are completed ($completedActionList == actions\ in\ the\ plan$).

3.2. Distributed Plan Execution in Decentralized Mode

In the decentralized mode for distributed plan execution, there is no single scheduler. The agents communicate and coordinate with each other to execute a plan (Refer Fig. 3b). Each agent maintains a *completedActionsList* locally. It also maintains a local plan which is a set of local actions $\langle In(a), precond(a), a, effect(a), Out(a) \rangle$, where a is the label of the action, $In(a)$ is the set of actions on which a is dependent and $Out(a)$ is the set of actions that are dependent upon a . Given a global partial order plan, it is easy to derive the local plans for each agent.

Fig. 2 shows an example with three agents where Agent1 operates the actions $\{a, d, f\}$, Agent2 operates $\{b, e\}$ and Agent3 operates $\{c, g\}$. The preconditions and effects of the actions are specified through a bunch of predicates named $atX()$. Fig. 2a shows the global partial order plan. Corresponding local plans for the agents are shown in Fig. 2b. For example, for the action labeled e , the in-dependency is $IN(e) = \{c, d\}$ and the out-dependency is $Out(e) = \{f, g\}$.

The local plan execution proceeds as follows: when an action a is completed, the information is communicated to the agents in $Out(a)$ i.e. the agents whose actions have a in their dependency. Then, each agent updates its *completedActionsList* and decides whether it can schedule some action for execution by checking against $In(a)$. Only after that, the agent checks the precondition through $getVal(precond(a))$, calls the $execute(a)$ function and then again checks the effect through $getVal(effect(a))$ in that order.

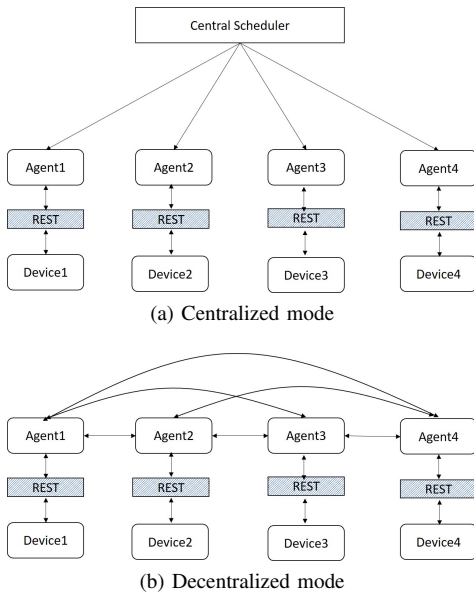


Figure 3: Architectures for distributed plan execution.

3.3. Issues with multi-agent plan execution in an untrusted environment

The distributed plan execution mechanisms described above (both centralized and de-centralized) implicitly assume that the agents are trusted and collaborative. Therefore, plan execution can be implemented using simple datastructures and communication mechanisms. However, in highly distributed CPS's with mobile entities entering and exiting the system dynamically, such assumptions may be misplaced. In the following, we discuss the points of failure in distributed plan execution in untrusted environments where agents may be faulty or malicious.

- 1) An agent may not execute the action it is supposed to execute. This will result in blocking the dependent actions in the same or other agents, or not resulting in the desired effects, consequently leading to plan failure.
- 2) An agent may execute a local action out of order even though the dependencies are not yet fulfilled. This can happen when such early execution is beneficial to the agent but leads to failure in achieving the system goal (a la "tragedy of commons").
- 3) An untrusted agent may call the *execute()* API even though the preconditions do not hold. This may be because of malicious or greedy reasons on part of the agent since the agent might have predicted that such preemptive execution might lead to system failure (e.g. baking a cake without pre-heating the oven).
- 4) Agents may notify the completion of actions even though they might not have issued the API corresponding to the action or even though the effects were not satisfied. This may be because an agent may profit by "shirking" its responsibility.

In the rest of the paper, we describe how one can implement

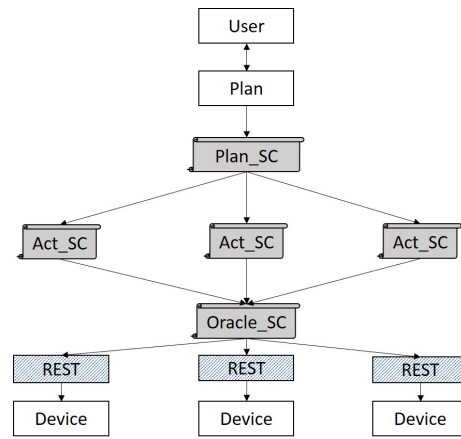


Figure 4: Smart contract based Distributed plan execution in centralized mode.

the agents in a distributed plan execution system with smart contracts such that the above-mentioned issues are prevented.

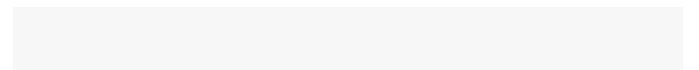
4. Distributed Plan Execution with Smart Contracts

The main idea is to coordinate the steps in distributed plan execution through smart contracts. Then, the semantics of smart contracts and the underlying blockchain ensures that the (in)actions of the agents are securely recorded so that agents violating the smart contract (hence the intended plan execution) can be determined unequivocally. The associated penalty of contract violation then forces the agents to follow the contract resulting in proper plan execution.

4.1. Smart contracts for plan execution in centralized mode

The architecture of the Smart Contract enhancement is shown in Fig. 4. There is a smart contract (*Plan_SC*) which is the Scheduler agent that orchestrates a given plan, a bunch of smart contracts (*Act_SC*) each responsible for getting the predicate values and getting the actions executed. Since these actions need to access the datasources external to the underlying blockchain, the operations are executed as transactions on an Oracle smart contract (*Oracle_SC*) (explained in the following). In addition, for ease of information access, we use a Register smart contract (*Register_SC*) that essentially stores a mapping from the actions to the corresponding Act SC.

We give the schema of the smart contracts in Listing 1. Complete code is given in the Appendix¹.



1. The code is attached only for ready reference. To respect page limits, we will just provide a link to the code in a publicly available archive in the final version.

```

contract Register_SC {
    event logDeployedContract(address
        tenant,string s);
    mapping(uint256 => Act_SC)
        action_scAddrMap;
    function setAct(uint[] memory
        action_names) public { ... }
    function getAct(uint256 action_name)
        public returns(Act_SC act_scAddr) {
        ...}
}

contract Plan_SC{
    event loglist(uint256[] x,string s);
    event logexist(uint256[] x,uint s,bool
        b);

    function Plan_SC(){//DAG Creation
    :
    }
    function dispatchNext(Register_SC
        register_addr, uint256 Action_id)
        returns(bool _success){ ... }

import "github.com/oraclize/ethereum-api/
    oraclizeAPI.sol";
contract Act_SC is usingOraclize{
    struct Action{
        uint256 Action_id;
        string Action_URI;
        bytes32 [] preCond,effects;
    }
    Action private action_1,....,action_6;
    mapping(uint256 => Action)
        actionid_ActionMap;

    function Act_SC(uint[] memory
        action_names){...}
    function getPreCondition(uint256
        Action_id) public returns (bytes32
        [] preCond){...}
    function getEffect(uint256 Action_id)
        public returns(bytes32[] effect) {
        ...}
    function checkPreCondition(bytes32[]
        preCond,bytes32[] preCond_orac)
        returns(bool cond){...}
    function checkEffect(bytes32[] eff,
        bytes32[] eff_orac) public returns (
        bool cond){...}
    function execute(uint256 Action_id)
        external returns(bool success){...}
}

```

Listing 1: Smart Contracts for Plan Execution in Centralized Mode

4.1.1. Outline of the smart contracts. The Plan smart contract acquires a plan as a part of its constructor. In the listing, we have hardcoded the plan, but it can be obtained from the user or application or from some service. The plan is represented as a list of lists, the head of each sub-list denoting an action a , and the rest denoting $In(a)$. Whenever there is a completion notification for an action, this is deleted from all the lists. Thus, when the sublist is a singleton, it is known that its in-dependencies have been already completed and the head of the list is taken up for execution by calling *dispatchNext*.

There are multiple instances of the Act smart contract each corresponding to an agent. For each action that the Plan smart contract wants to dispatch, it must know the corresponding Act smart contract address. This information is maintained in a mapping (actions \rightarrow Act SC instance address) in the Register smart contract. The *setAct* function is used to create the mapping and *getAct* function is used to extract the address of the Act smart contract instance.

Taking advantage of the shared blockchain, the different instances of Act smart contract instance share a common *completedList* of actions. When the action is executed by an Act instance and the *effect* has been verified, it updates the list. The Plan smart contract accesses the *completedList* to see if the execution was successful and whether it can update the plan datastructure (list of lists).

Note the import statement in the Act smart contract: *import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";* and the definition of Act smart contract as derived from Oraclize: *contract Act_SC is usingOraclize*. This prepares Act SC to interface with Oraclize with the *__callback(bytes32 myid, bytes32[] result)* function. In the main function *execute()*, the Act smart contract checks the precondition after getting the predicate values from the oracle, calls the device execution through the Oracle (this is skipped in the code since we are just faking the execution, but in real application too, this is no different from the Oracle calls), and checks the effect after which it updates the *completedList*. Any failure in any of the above steps results in aborting the transaction, which can ultimately be cascaded to the user or application.

4.1.2. Oracle SC from Oraclize. The Oraclize [9] solution guarantees that the data fetched from the original data-source is genuine and untampered. This is achieved by creating a document called authenticity proof for the data returned from the sensors. These proofs can be build using different technologies like Trusted Execution Environments e.g. Intel SGx [10] and auditable virtual machines. The communication between smart contract and Oraclize is asynchronous in nature and happens as follows: Every transaction broadcast by an agent participating in the blockchain will have special instruction which manifest to Oraclize. The 3rd party service running the oraclize will be constantly monitoring the blockchain for such requests of sensor Data. As per the request, Oraclize will collect the result, sign and execute the *_callback* function in smart contract. It also follows an "If This Then That" logical model. For

example, it could continuously check for a condition and notify when the condition has been met. The oracles can be both Inbound, to get data inside the blockchain or Outbound, to notify an actuator outside of the blockchain about an event.

4.1.3. Deployment and Running. First the oracle smart contract is deployed. Using its address, the Act SC instances are deployed. User then deploys the Register SC that, in turn, calls the Act SC functions to populate the mapping (actions → Act SC instance address). Lastly, the Plan SC is deployed. After deployment, user or application can upload a plan to the Plan SC through the constructor. We plan to provide a `plan.setPlan(plan)` function to upload plans during operation as well which can help in replanning. The `plan.startExec()` starts off the execution of the plan by repeated calls to the `dispatchNext()` function.

The function `plan.dispatchNext()` selects actions whose dependencies have been fulfilled. These actions are then passed to `Act.execute()`. Preconditions of these actions are checked by obtaining the predicates from the Oracle SC. After the action is invoked through the corresponding API, effect predicates are checked. Any exception in any of the steps leads to call of `plan.abort()`. These steps are illustrated in a pictorial manner in Fig. 5.

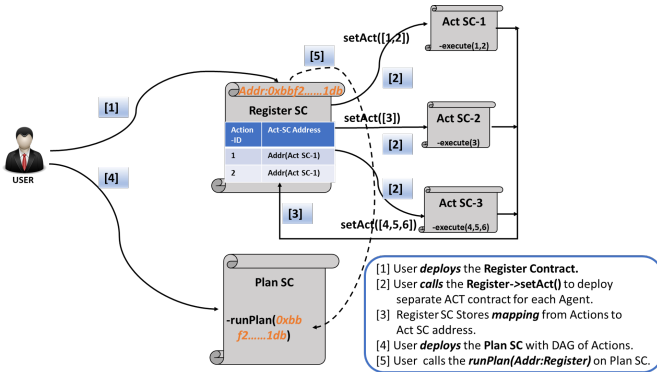


Figure 5: Deployment and Execution of smart contracts for plan execution

4.2. Smart Contracts based plan execution in decentralized mode

The main ideas for the smart contracts for decentralized mode operation is similar to the centralized case, except that the Scheduler `Plan_SC` and `Act_SC` are now combined with action dependencies maintained and updated in the single `Plan_Act_SC`. The architecture for the distribute plan execution with smart contracts is given in Fig. 6. As before, we have a Register smart contract which provides a function to map actions to the corresponding agents. This mapping is used to call the update function of the `Plan_Act_SC` instances. We also use the same Oracle smart contract to access the predicates and execute the actions.

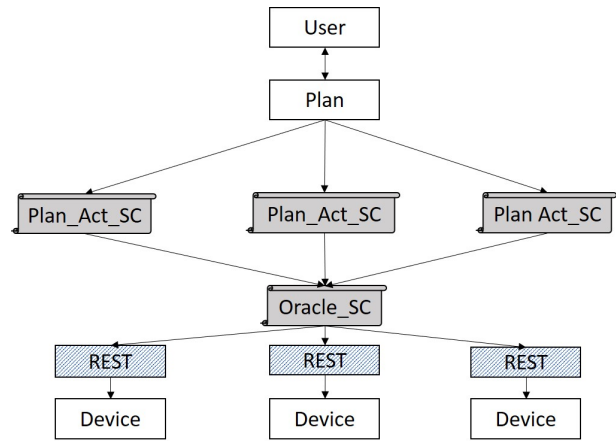


Figure 6: Distributed plan execution with Smart Contracts.

In the centralized case the Act smart contracts were interacting with the Plan smart contract and there was no communication among the Act instances. On the other hand, in the distributed case, the `Plan_Act` smart contract interact peer-to-peer. Each `Plan_Act` smart contract maintains a `completedList` which is used to find if any action can be enabled at all. When an action is enabled for execution, as before, precondition and effect are verified through the Oracle. When an action completes, the update function of the Act instances in `Out(a)` are called so that their `completedLists` are updated.

5. Related Work

There is very little published research work in the intersection of AI and blockchain. But there is immense interest and a large number of magazine style articles out there that talk about the potential. [11] talks about AI used as a cognitive technology for user interaction and how that enables a blockchain based solution. The online article [12] argues at a high level several benefits of blockchains to AI and vice versa. In a number of papers from [13] AI is suggested to analyze the effectiveness of past contracts and suggest new terms and conditions.

Blockchains are used to market and incentivize data, AI models and algorithms. Decisions from AI algorithms are proposed to be kept in the blockchain for provenance since blockchain ensures integrity of the data. Blockchain is used in federated learning to ensure accountability and prevent data poisoning [14]. However, it seems the exploration of the intersection of AI and blockchain is still very much in its early days.

[15] is a recent survey of the state of the art in the application of Blockchains to multi-agent systems. However in most cases, existing works do not address the issues in plan execution. For example, in [16] the focus is on multi-agent coordination and control in the context of UAVs using blockchains.

Most work is focused on multi-agents and distributed vs centralized planning or the issues in coordinating planning activity in a distributed setup. There is very little on execution time coordination of plans among multiple agents. For example, [2], [17], [18], [19] survey several multi-agent planning methods including distributed plans. [20] talks about how individual agents in a multi-agent setup may deal with failure situations. However, there is very little on how to enforce coordinated execution, especially in untrusted environments. In this paper, we show probably the most direct use of smart contracts in an AI use case to date.

6. Conclusion

Intelligent cyber-physical systems can be implemented by integrating planning capability in agent software, which transforms the system to a multi-agent planning paradigm. However, the traditional methods for planning and plan execution in multi-agent system can fail when the agents are untrusted. In this paper, we propose a smart-contract based extension of the agents which ensure proper plan execution even in untrusted environments.

We have described two specific architectures - one for a centralized and the other for a decentralized mode of execution of distribution plans. The architectures depend only upon the underlying devices and their agents and are independent of any applications built on top of the CPS. Moreover, the proposed architectures suggest that smart contracts can be automatically generated from the derived plans, as a result of which the entire system of smart, intelligent agents can be fully automated.

Another point to be noted is the clean separation of the physical layer operations (sensing and actuation) from the contract logic through the oracle smart contracts. Due to this characteristic, it is envisaged that functions such as plan synthesis and replanning can be easily integrated in the system by accessing these external services through the Oracle.

References

- [1] "An architectural blueprint for autonomic computing," White Paper, IBM, June 2005.
- [2] M. de Weerd and B. Clement, "Introduction to planning in multiagent systems," *Multiagent Grid Syst.*, vol. 5, no. 4, pp. 345–355, Dec. 2009. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1735317.1735318>
- [3] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.
- [4] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus. url:<https://www.stellar.org/papers/stellar-consensusprotocol.pdf>," 2016.
- [5] S. Popov, "The tangle. url:<https://iota.org/iota-whitepaper.pdf>," *IOTA Whitepaper. pdf*, 2017.
- [6] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, vol. 310, 2016.
- [7] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.
- [8] "The solidity contract-oriented programming language." [Online]. Available: <https://github.com/ethereum/solidity>
- [9] Oraclize-Dev-Community, "Documentation of oraclize. url:<https://docs.oraclize.it>," *Documentation*, 2018.
- [10] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.
- [11] P. Treleaven and B. Batrinca, "Algorithmic regulation: Automating financial compliance monitoring and regulation using ai and blockchain," *Journal of Financial Transformation*, vol. 45, pp. 14–21, 2006.
- [12] F. Corea, "The convergence of ai and blockchain: whats the deal?" https://medium.com/@Francesco_AI/the-convergence-of-ai-and-blockchain-whats-the-deal-60c618e3acc, 2017, accessed: 2018-09-01.
- [13] Blockchain, "Combining blockchain and ai to make smart contracts smarter. url:<https://legalconsortium.org/uncategorized/combining-blockchain-and-ai-to-make-smart-contracts-smarter/>," *Blockchain Consortium*, 2017.
- [14] J. Weng, J. Weng, M. Li, Y. Zhang, and W. Luo, "Deepchain: Auditable and privacy-preserving deep learning with blockchain-based incentive," *IACR Cryptology ePrint Archive*, vol. 2018, p. 679, 2018. [Online]. Available: <https://eprint.iacr.org/2018/679>
- [15] D. Calvaresi, A. Dobovitskaya, J. P. Calbimonte, K. Taveter, and M. I. Schumacher, "Multi-agent systems and blockchain: Results from a systematic literature review," in *16th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2018)*, 2018.
- [16] A. Kapitonov, S. Lonshakov, A. Krupenkin, and I. Berman, "Blockchain-based protocol of autonomous business activity for multi-agent systems consisting of uavs," in *Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS)*, October 2017, pp. 84–89.
- [17] A. Bonisoli, "Distributed and multi-agent planning, challenges and open issues," in *13th Doctoral Workshop in Artificial Intelligence in International Conference of the Italian Association for Artificial Intelligence (AI*IA 2013)*, December 2013.
- [18] Y. Dimopoulos and P. Moraitis, "Multi-agent coordination and cooperation through classical planning," in *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, December 2006.
- [19] R. Nissim, R. I. Brafman, and C. Domshlak, "A general, fully distributed multi-agent planning algorithm," in *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, May 2010.
- [20] C. Guzman, P. Castejon, E. Onaindia, and J. Frank, "Robust plan execution in multi-agent environments," in *IEEE 26th International Conference on Tools with Artificial Intelligence*, 2014, pp. 384–391.

7. Appendix

7.1. Register Smart Contract

```
pragma solidity 0.4.0;
contract Register_SC {
    event logDeployedContract(address
        tenant, string s);
    mapping(uint256 => Act_SC)
        action_scAddrMap;

    function setAct(uint[] memory
        action_names) public {
        Act_SC addr=new Act_SC(action_names)
            ; // returns address of new
            contract
        for(uint256 x = 0; x < action_names.
            length; x++) {
            action_scAddrMap[action_names[x]
                ]=addr;
        }
        logDeployedContract(addr, "
            new_contract_address");
    }

    function getAct(uint256 action_name)
        public returns (Act_SC act_scAddr) {
        return action_scAddrMap[action_name
            ];
    }
}
```

Listing 2: Register Action API

7.2. Plan Smart Contract

```
pragma solidity ^0.4.0;
import "Register.sol";
contract Plan_SC{
    event loglist(uint256[] x, string s);
    event logexist(uint256[] x, uint s, bool
        b);

    using ListInteger for *;
    uint action_count;
    uint256[][6] dag;
    uint256 CompletedListLength;
    function Plan_SC(){//DAG Creation
        action_count=6;
        dag[0]=[1]; dag[1]=[2];
        dag[2]=[3,1]; dag[3]=[4,2];
        dag[4]=[5,2,3]; dag[5]=[6,5];
    }
    function runPlan(Register_SC
        register addr){
        while(CompletedListLength!=
            action_count)
```

```
for(uint256 x = 0; x < dag.length; x
    ++) {
        uint current;bool _success;
        if(dag[x].getSize()==1){
            current =dag[x][0];
            _success=dispatchNext(
                register_addr,current);
        }
        if(_success){
            for(uint256 y = 0; y < dag.
                length; y++) {
                if(exist(dag[y],current))
                    dag[y].removeByValue(current);
            }
        }
    }
    function dispatchNext(Register_SC
        register_addr, uint256 Action_id)
        returns (bool _success){
        Act_SC act_addr=register_addr.getAct
            (Action_id);
        bool success=act_addr.execute(
            Action_id);
        uint256[] memory res=act_addr.
            getCompletedList();
        CompletedListLength=res.length;
        loglist(res, "completed_list_updated"
            );
        return success;
    }
}
```

Listing 3: Plan API

7.3. Act Smart Contract

```
pragma solidity ^0.4.0;
import "github.com/oraclize/ethereum-api/
    oraclizeAPI.sol";
contract Act_SC is usingOraclize{
    event display(string x);
    event display(string x, uint[] s);
    event abort(string s, uint256 actionID);
    event completed(string s, uint256
        actionID);
    event displayActionName(uint256 x);
    struct Action{
        uint256 Action_id;
        string Action_URI;
        bytes32 [] precondition;
        bytes32[] effects;
    }
    Action private action_1, ..., action_6;
    mapping(uint256 => Action)
        actionid_ActionMap;
    uint256[] public completedList;
```



```

function Act_SC(uint[] memory
  action_names){
  for(uint256 x = 0; x < action_names.
    length; x++) {
    if(action_names[x]==1){
    action_1.Action_id=action_names[x];
    action_1.Action_URI="http://a1.com";
    action_1.precond.push("precond1_0");
    action_1.effects.push("effects1_0");
    actionid_ActionMap[1]=action_1;
    }//Similarly initialise other
      actions
    }
}

function getCompletedList() external
  returns(uint256[] list) {
  return completedList;
}

function getPreCondition(uint256
  Action_id) public returns (bytes32
  [] preCond){
  return actionid_ActionMap[Action_id
  ].precond;
}

function getEffect(uint256 Action_id)
  public returns (bytes32[] effect) {
  return actionid_ActionMap[
  Action_id].effects;
}

function checkPreCondition(bytes32[]
  preCond,bytes32[] preCond_orac)
  returns (bool cond) {
  return true;
}

function checkEffect(bytes32[] eff,
  bytes32[] eff_orac) public returns (
  bool cond) {
  return true;
}

//oraclize start from here...
bytes32 [] orac_res;// for storing
  preCond and effects
function __callback(bytes32 myid,
  bytes32 [] result) {
  if (msg.sender != oraclize_cbAddress
  ()) throw;
}

```

```

    orac_res = result;
  }
}

function getPredicatesValues(uint256
  Action_id,string s) returns (bytes32
  [] effect){
  string action_url=actionid_ActionMap
  [Action_id].Action_URI;
  oraclize_query("URL", action_url);//
  will call callback
  return orac_res;
}

function execute(uint256 Action_id)
  external returns (bool success){
  bytes32[] memory preCond=
  getPreCondition(Action_id);
  bytes32[] memory preCond_orac=
  getPredicatesValues(Action_id,"
  preCond");
  if(checkPreCondition(preCond,
  preCond_orac)){
  display("callAPI(action_api_map[
  actionID])");
  }
  else{
  abort("Transaction_Rollback",
  Action_id);
  return false;
  }
  bytes32[] memory eff=getEffect (
  Action_id);
  bytes32[] memory eff_orac=
  getPredicatesValues(Action_id,"
  eff");
  if(checkEffect(eff,eff_orac)){
  completedList.push(Action_id);
  completed("Action_Completed",
  Action_id);
  return true;
  }
  else
  abort("Transaction_Rollback",
  Action_id);
  return false;
}
}

```

Listing 4: Act API