SmartUpdater - DSL Design Document

1 Overview

This document provides a comprehensive guide on using the DSL in SmartUpdater to define both smart contracts and modifying policies of logic and state in smart contracts. The DSL is designed to provide an entire and flexible approach to contract development, enabling developers to annotate contract states and specify modification policies with ease.

The DSL in SmartUpdater is composed of two main parts:

- **DSL-written Contract**: This component allows developers to annotate contract states with storage attributes, which SmartUpdater uses to optimize contract partition. Annotations include the approximate storage size of states with dynamic storage strategies and state update probabilities.
- **DSL-written Policy**: The policy component provides database-like interfaces, enabling developers to tailor contract maintenance requirements, significantly streamlining the maintenance process.

By following this document, developers can leverage the DSL to define the smart contract and modifying policies, enabling SmartUpdater to further optimization and maintenance.

2 DSL-written Contract

2.1 Syntax and Grammar

The DSL-written contract follows specific annotation guidelines for state storage attributes,i.e., the approximate storage size of one state. Note that, this storage attribute is focused on the state with dynamic storage strategies (e.g., dynamic array-type and mapping-type). In addition, to facilitate more efficient maintenance, a non-obligatory option for developers is provided by SmartUpdater to specify the state update probability, such that variables with high update probability might be placed separately to avoid frequent migration of unchanged states.

Each state in the contract can be annotated with the following attributes:

- `@mag`: This annotation specifies the approximate storage size for individual states that use dynamic storage. It is intended for cases where a single state requires a specific storage allocation.
- `@bat`: This annotation is used to indicate the batch size for states that share similar storage requirements.
- `@prob`: This optional annotation denotes the update probability of a state, which helps SmartUpdater manage states with high update frequency separately. By isolating frequently updated states, SmartUpdater can avoid unnecessary migrations of unchanged states during maintenance.

2.2 Contract Examples

Below are examples of DSL-written contracts, along with explanations to clarify each annotation.

2.2.1 Example: OnePlanetToken

```
contract OnePlanetToken {
    //@prob 0.8
    string public ETHUSD;
    string public tokenPrice;

    //@mag 500
    mapping(address => uint) balances;
    ...
}
```

In the `OnePlanetToken` contract, two annotations are used to specify storage attributes that guide SmartUpdater in optimizing contract maintenance:

- `@prob`: The `@prob 0.8` annotation indicates a high likelihood that the `ETHUSD` variable will be updated frequently.
- `@mag`: The `@mag 500` annotation specifies the approximate storage size allocated for the `balances` mapping. Here, it suggests that the `balances` mapping requires an estimated storage size of 500.

2.2.2 Example: CreditDAO Contract

```
contract CreditDAO {
    struct Election{
        // @mag 100 @bat 2
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        address maxVotes;
        uint numMaxVotes;
    }
    mapping(uint => Election) public elections;
}
```

In this example, the mapping-type states of `Election` struct in the `CreditDAO` contract are annotated with two attributes:

- `@mag`: Here, `@mag 100` indicates an estimated storage size of 100 for the dynamic mapping-type state variables within the `Election` struct.
- `@bat`: In this case, `@bat 2` specifies that two states (i.e., `candidates` and `userHasVoted`) are grouped together, sharing common storage characteristics.

3 DSL-written Policy

The state modifying policy includes operations for adding, deleting, and updating states, which are common requirements in smart contract maintenance. As handling complex struct-type states in smart contracts requires unique operations, we have introduced the operations using action keywords *CREATE* and *ALTER*, where the former operation is to generate the new struct-type state and the latter is to change its internal elements. For logic updates, our DSL-written policy accommodates the direct submission of the revised code.

The DSL-written policy in SmartUpdater provides the following key features:

- **State Modification**: The core functionality of the DSL-written policy is to enable time-efficient to the state of a smart contract. This includes adding new states, deleting obsolete states, and updating existing states.
- **Database-like Syntax**: The DSL syntax is designed to resemble database-like operations, such as `INSERT`, `DELETE`, and `UPDATE`, making it intuitive for developers familiar with database management.
- **Support for Complex State Layouts**: The DSL can handle complex state structures like struct type, ensuring compatibility with various contract design patterns and avoiding potential issues associated with state layout changes.

3.1 Syntax and Grammar

The syntax of the DSL follows an EBNF grammar. Each policy is composed of one or more statements (*Stmt*), which specify actions to be applied to contract states. This grammar allows developers to define contract modifications in a precise and structured way, minimizing the risk of errors. The main grammar components are as follows:

$$
\begin{aligned}
\textit{Policy} :=& \textit{Stmt+}\\
\textit{Stmt } S :=& \textit{InsStmt} \mid \textit{DelStmt} \mid \textit{UpdStm} \mid \textit{CreStmt}\\
& \mid \textit{AltStmt} \mid S;S\\
\textit{AltOp } A :=& \textit{InsStmt} \mid \textit{DelStmt} \mid \textit{UpdStm} \mid A;A\\
\textit{InsStmt} :=& \texttt{INSERT}\ (i,t,v,m)\\
\textit{DelStmt} :=& \texttt{DELETE}\ (i,t,v,m)\\
\textit{UpdStmt} :=& \texttt{UPDATE}\ (i,t,v,m)\ \texttt{AS}\ (i,t,v,m)\\
\textit{CreStmt} :=& \texttt{CREATE}\ \{\textit{InsStmt+}\}\\
\textit{AltStmt} :=& \texttt{ALTER}\ \{A+\}\\
i \in \textbf{String} \quad t \in& \textbf{ Type} \quad v \in \textbf{Value} \quad m \in \textbf{Modifier}
\end{aligned}
$$

Each statement contains a modification type and the target state represented via four tuples, formally as (identifier, type, value, modifier). The modification types are as follows:

- **INSERT**: Adds new states to the contract, allowing developers to introduce new data fields without disrupting existing data.
- **DELETE**: Removes specified states from the contract, useful for cleaning up outdated or unnecessary data fields.
- **UPDATE**: Modifies attributes of existing states, which is often necessary when changing variable types or adjusting values in the contract.
- **CREATE**: Defines new struct types with associated states, providing a way to add complex data structures within the contract.
- **ALTER**: Refactors existing structs by updating their fields, supporting complex modifications in nested or compound data types.

3.2 Policy Examples

Below are examples of smart contracts and corresponding DSL-written policies, along with explanations to clarify each policy's function.

3.2.1 Example: OnePlanetToken

Old Contract Version:

```
contract OnePlanetToken {
    string public ETHUSD;
    string public tokenPrice;
    mapping(address => uint) balances;
    ...
}
```

New Contract Version:

```
contract OnePlanetToken {
    string public tokenPrice;
    uint public ethPrice;
    mapping(address => uint) balances;
    ...
}
```

Policy:

```
DELETE (ETHUSD, -, -, -);
INSERT (ethPrice, uint, -, -);
```

Explanation: This policy removes the outdated `ETHUSD` variable and replaces it with a new `ethPrice` variable of type `uint`.

3.2.2 Example: CreditDAO Contract

Old Contract Version:

```
contract CreditDAO {
    struct Election{
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        address maxVotes;
        uint nummaxVotes;
    }
}
```

New Contract Version:

```
contract CreditDAO {
    struct Election{
        mapping(address => Participant) userMap;
        address maxVotes;
        uint nummaxVotes;
    }
    struct Participant{
        bool isCandidate;
        bool hasVoted;
    }
}
```

Policy:

```
CREATE Participant{
  INSERT (isCandidate,bool, -, -);
  INSERT (hasVoted,bool, -, -);
};
ALTER Election{
  INSERT (userMap,mapping(address=>Participant), -, -);
  UPDATE (candidates, -, -, -) AS (userMap.isCandidate, -, -,
    -);
  UPDATE (userHasVoted, -, -, -) AS (userMap.hasVoted, -, -,
    -);
};
```

Explanation: In this policy, a new struct `Participant` is created with fields `isCandidate` and `hasVoted`. The `Election` struct is then modified to replace the individual mappings `candidates` and `userHasVoted` with a unified mapping `userMap` that points to `Participant`. The `UPDATE` statements map the old values into this new structure.