AspectSol v0.1α
User Manual

Gordon J. Pace and Joshua Ellul and Shaun Azzopardi and Ryan Falzon
{gordon.pace|joshua.ellul|shaun.azzopardi}@um.edu.mt, rfalzonryan@gmail.com

published 1 August 2022, last updated 31 July 2022

Contents

1 Overview

Aspect oriented programming is a software development paradigm normally used in conjunction to object oriented programming. The main aim of such languages is to separate software concerns from the business layer to further increase code modularization. Typical cases where Aspect programming is beneficial is for logging and transactional logic. Such cases are not domain specific to the business logic and, when abstracted out, can in theory be used in any software you develop.

Given the nature of immutability of how blockchains operate, it makes developing smart contracts a tedious process as one would need to reach a level of confidence that the contract is bug free. Even though the majority of smart contracts are unique in terms of their behaviour, certain common factors might exist, such as for example admin-only calls to certain functions or even perhaps a number of steps that need to be taken following updates done to a balance. Hence, it would be beneficial if developers have access to a tool that allows them to program such non-business logic once, test it, and inject it into as many smart contracts as they want.

AspectSol is an aspect-oriented language that aims at abstracting the implementation of features on top of already defined smart contracts to a higher level. This tool takes a script written in a domain language specified in this document and a Solidity smart contract, and produces a new Solidity smart contract reflecting the changes indicated in the aspect script. Such a tool allows for bulk updates of contracts where specific rules must be set on all of them.

The below is an example of the syntax used to formulate an AspectSol script. In this example we see how one can inject code into a smart contract to make the `terminate` function executable only by

the person who created the smart contract. Words such as `aspect`, `add-to-declaration`, `before` and `call-to` are all native keywords to the AspectSol domain language. These, together with more, will be further described in section 2 of this manual.

```
aspect OnlyOwnerInjection {
  address private owner;

  before execution-of Casino.constructor {
    owner = msg.sender;
  }

  before call-to Casino.terminate() {
    require(msg.sender == owner, "Caller is not owner");
  }
}
```

Listing 1: AspectSol Script Example.

2 Tool Syntax

The tool AspectSol allows the for the creation of short function-like scopes called pointcuts. Within these scoped functions, one can include code that should be factored into the original smart contract based on whether the pointcut rule is met. Similar to how classes are created in OOP languages, in AOP languages, an Aspect needs to be created. Therefore a single aspect may contain several pointcuts. Code within pointcuts is then applied to smart contracts based on the rules set out in the pointcut definition.

AspectSol provides two types of aspects: (i) *code-appending* aspects which add advice (the new code) at the location of the pointcut, and (ii) *modification* aspects which are used to update an existing definition of the original smart contract.

2.1 Selector expressions

In order to allow for the identification of pointcuts indicating where to append or modify the code, AspectSol uses selector expressions to identify a relevant unit of code.

$$\langle \textit{selector} \rangle \quad ::= \quad \langle \textit{contract-selector} \rangle \mid \langle \textit{function-selector} \rangle$$

Contract selectors. Contract selectors are used to identify a subset of contracts in the source files AspectSol is processing, with the following syntax (where the interface name is optional):

$$\langle \textit{contract-selector} \rangle \quad ::= \quad \langle \textit{contract-name} \rangle [:: \langle \textit{interface-name} \rangle]$$

A contract selector is made up of two components:

- **Contract:** The contract selector specifies the name of the contract to match. It can be (i) a contract name to be matched in the source files provided; or (ii) a concrete address on the Ethereum blockchain where that contract is to reside. One can use `*` to match any contract name.
- **Interface:** The interface specifies that a smart contract will only match if it is an instance of the specified interface name. If no such restriction is desired, one can simply leave out the interface name (and preceding double colon symbol).

For instance, to identify the contract `CustomToken` which is to be an instance of `ERC20`, one would write `CustomToken::ERC20`. The contract selector `CustomToken` would match any contract so named in the source files, while `*::ERC20` would match all contracts which are instances of the `ERC20` interface.

Function selectors. Function selectors revolve around named contracts, defined contract interfaces and particular functions within contracts. A full function selector takes the following form (where the `returning` clause is optional):

$$\langle \textit{function-selector} \rangle \quad ::= \quad \langle \textit{contract-selector} \rangle.\langle \textit{function-name} \rangle(\langle \textit{parameters} \rangle) \ [\texttt{returning} \ \langle \textit{type} \rangle]$$

For instance, to identify the `transfer` function in contract `CustomToken` which implements the `ERC20` interface, one would write:

```
CustomToken::ERC20.transfer(address _to, address _amount) returning (bool)
```

The different components of the selector are discussed below:

- **Function name:** This selector identifies which functions to match. If one wants to match *all* functions in the specified contract, one can use the `*` symbol.
- **Parameters:** The parameters of the function can be identified, together with their type. The parameter names allow us access to the values passed in the advice. We can use the `*` symbol instead of a single parameter which we do not care about e.g. `CustomToken.transfer(address _to, *)`, or to match the function with any number of parameters e.g. `CustomToken.transfer(*)`.
- **Return value:** The final part specifies the return type of the functions we want to match.[1] It can be left out altogether if one does not want to make any constraints.

Below are some examples of function selectors:

- To match the parameterless `placeBet()` function in the `Casino` smart contract (which can be found in the sources provided to AspectSol), one can use the selector `Casino.placeBet()`.
- To match all functions in the `Casino` smart contract, one can write: `Casino.*(*)`.
- If it is only parameterless functions in the `Casino` smart contract one is interested in, one would write: `Casino.*()`.
- In order to match any function called `withdraw` in any contract provided, one would use the selector: `*.withdraw(*)`.
- If one wants to identify functions of a smart contract already deployed ont he blockchain at a particular address, one would write: `0x71C7656EC7ab88b098defB751B7401B5f6d8976F.*(*)`.[2]

Sometimes, when one uses the wildcard selector `*`, one may want to access the actual name of that element. For instance, one may want to access the name of the function matching the selector `Casino.*()`. In order to do so, one can replace the wildcard with `[[f]]`, which binds the variable `f` with the name of the function in the rest of the pointcut (for instance to call the function again). Such bindings can be used with the contract and function names, including both in the same selector e.g. `[[cont]].[[func]](*)`.

In addition to selecting based on the contract and function names and parameters, AspectSol also allows filtering selectors based on function decorators e.g. user modifiers, access modifiers, payable, etc.

A selector can be followed by a decorator condition, written as: ⟨*selector*⟩ `tagged-with` ⟨*decorator-expression*⟩.

Decorators allowed are: (i) access modifiers (`public`, `private`, `internal`, `external`); (ii) function type modifiers (`pure` and `view`); (iii) the `payable` modifier; and (iv) user-defined modifiers. Decorator expressions are Boolean expressions over these base decorators, using `&` for conjunction, `|` for disjunction and `!` for negation.

For instance, to select all public and external functions in the `Casino` smart contract which use the user-defined modifier `byOwner`, but which may not receive funds, one would write:

```
Casino.*(*) tagged-with ((public | external) & byOwner & !payable)
```

Finally, AspectSol also provides means of filtering selectors to match only functions appearing (or not appearing) in a certain contract interface: writing ⟨*selector*⟩ `in-interface` ⟨*interface-name*⟩ and ⟨*selector*⟩ `not-in-interface` ⟨*interface-name*⟩.

For example, to select all public functions in the `CustomToken` smart contract which do not appear in the `ERC20` interface standard, one would write:

```
CustomToken.*(*) not-in-interface ERC20 tagged-with public
```

[1] **Limitation:** We currently provide no means of accessing the return value in the advice, and only the type of the return value is specified.

[2] Obviously, we cannot change the code of smart contracts already deployed, but we can use such selectors to identify, for instance, external calls to functions in such deployed contracts.

Variable getter and setter selectors. Apart from functions, AspectSol also allows selectors for global contract variable getters and setters. If such getters and setters are not defined in the code, AspectSol creates them and replaces access to these variable using the new functions. Getter and setter selectors are identified using the following syntax:

`get` ⟨*variable-type*⟩ ⟨*contract-selector*⟩.⟨*variable-name*⟩ [`tagged-with` ⟨*decorator-expression*⟩]
`set` ⟨*variable-type*⟩ ⟨*contract-selector*⟩.⟨*variable-name*⟩ [`tagged-with` ⟨*decorator-expression*⟩]

Note that getter and setter selectors can be filtered using a decorator expression, but limited to the access modifiers `public`, `private` and `internal`.

As in the case of function selectors, the contract name, variable type and variable name can also be replaced by a wildcard or variable to capture the name.

Below are some examples of selectors:

- To capture all getters to the `savings` variable (of type `uint256`) in the `Wallet` smart contract, one would write: `get uint256 Wallet.savings`.
- If one is interested in all setters to public variables named `savings` in any contract provided in the source, and of any type, one would write: `set * *.savings tagged-with public`.
- To select all getters to `uint256` variables which are private or internal in any smart contract provided in the code, but also be able to refer the variable name, one would write: `get uint256 *.[[var]] tagged-with (private | internal)`.

2.2 Identifying pointcuts

Code-appending pointcuts identify locations in the code where advice can be injected. These pointcuts are encoded using selectors as follows:

⟨*pointcut*⟩ ::= ⟨*placement*⟩ ⟨*location*⟩ ⟨*function-selector*⟩ ⟨*sender*⟩

The components of such a pointcut are explained below:

- **Placement:** The placement property dictates where the scoped code should be placed. Two options are available, `before` and `after`. The former triggers the advice code just *before* the matched selection is executed, while the latter triggers the advice code *after* the matched selection.

  ⟨*placement*⟩ ::= `before` | `after`

- **Location:** The location property dictates whether the advice is inserted on the caller's side or as part of the selection, providing two options: `call-to` and `execution-of`. This dictates the variables accessible from the advice.
  - `call-to` will inject the advice on the caller side of the matched selection (which has to be a function). The advice can thus access variables available on the caller's side.
  - `execution-of`: As opposed to `call-to`, this will execute the advice at the start of the matched selection, thus having access to variable local to that selection.

  ⟨*location*⟩ ::= `call-to` | `execution-of`

- **Sender:** The sender property, written as `originating-from` ⟨*origin*⟩, is used to specify the originator of the smart contract call i.e. `msg.sender`. It is important to note that this property is optional. Different ways of identifying the origin of the call are available:

  ⟨*sender*⟩ ::= `originating-from` ⟨*sender-source*⟩
  ⟨*sender-source*⟩ ::= ⟨*address*⟩ | `this` | `!this`
  | ⟨*contract-selector*⟩ | ⟨*function-selector*⟩

  An Ethereum address can be used to specify the initiator of the transaction, expressed as a constant or using variables and functions available from the point where the joinpoint is instrumented e.g. the following pointcut refers to the start of calls to `depositMoney()` in the `Wallet` smart contract with `msg.sender` being the first entry in an array of owners, `owners[0]`:

```
before execution-of Wallet.depositMoney() originating-from owners[0]
```

The expression (`owners[0]` in this case) is evaluated at the start of (`before`), and inside (`execution-of`) the `depositMoney()` function.

Checking whether or not the originating address is the same as the contract we are instrumenting can be checked using the `this` and `!this` sender identifier.

The sender filter can also be used to restrict function call pointcuts to arise from within a particular function or contract, both specified by their name. For instance, the following two pointcuts match at the end of function `depositMoney()` of the contract named `Wallet` as long as the call originated from the `Bank` smart contract (which is to be defined in the source code provided to AspectSol) or, more specifically, the `refund()` function in the `Bank` smart contract respectively:

```
after execution-of Wallet.depositMoney() originating-from Bank
after execution-of Wallet.depositMoney() originating-from Bank.refund()
```

While the above references function selectors, when it comes to variable access selectors, these can be only be used in conjunction with the placement property as follows:

```
after get uint256 Wallet.[[var_name]]
```

2.3 Code injection aspects

Using pointcuts, one can inject code at the locations by adding advice:

$$\langle \textit{code-injection-aspect} \rangle \quad ::= \quad \langle \textit{pointcut} \rangle \ \{\langle \textit{advice} \rangle\}$$

The code given in the advice is injected as is at matching pointcuts. For example, if one wants to keep count of how many times money is deposited in the `Wallet` smart contract using a `deposit` function and the running total of deposits one would write:

```
after execution-of Wallet.deposit(*) {
    count_deposits++;
    total_deposits += msg.value;
}
```

It is worth noting that any variable with a binding in the pointcut can be accessed in the advice using the same notation of the variable in double square brackets. All such occurrences in the advice are replaced by the term in the pointcut which matched. For instance, the following code emits an event whenever a `uint256` variable in the `Wallet` smart contract is updated:

```
after execution-of set uint256 Wallet.[[var_name]] {
  emit VariableRead("[[var_name]]", [[var_name]]);
}
```

2.4 Code modification aspects

Apart from adding executable code at a pointcut, AspectSol also provides means of modifying existing code in a controlled manner. These aspects are defined over a logical block of code (typically defined as a selector), typically a function or a contract, which can be modified with appropriate advice.

$$\begin{array}{lcl}
\langle \textit{modification-aspect} \rangle & ::= & \texttt{add-to-declaration}\ \langle \textit{selector} \rangle\ \{\langle \textit{declarations} \rangle\} \\
& | & \texttt{update-definition}\ \langle \textit{selector} \rangle\ \{\langle \textit{update-list} \rangle\} \\
\langle \textit{update-list} \rangle & ::= & \varepsilon \mid \langle \textit{update} \rangle\texttt{;}\ \langle \textit{update-list} \rangle \\
\langle \textit{update} \rangle & ::= & \texttt{implement}\ \langle \textit{interface-name} \rangle \\
& | & \texttt{add-tag}\ \langle \textit{decorator-name} \rangle \mid \texttt{remove-tag}\ \langle \textit{decorator-name} \rangle
\end{array}$$

As can be seen above, AspectSol provides two main modification types:

- `add-to-declaration`: Add the advice code as part of the declaration of the matched selection. One can, for instance, declare new variables, modifiers, functions, etc.
- `update-definition`: Allows updating the definition of the selected item using any combination of the following three directives:
  - `add-tag` is used to add the named decorator (such as user defined modifiers and access control decorators) to the selected item.
  - `remove-tag` is similarly used to remove the named decorator
  - `implement` is used when one wants to make a given smart contract an instance of a given interface. This is typically accompanied with another `add-to-declaration` modification aspect which adds the definitions of the interface functions.

Some examples of modification aspects are the following:

- In order to add a new variable to keep track of the number of times a function is called in a `Wallet` smart contract, one can use an `add-to-declaration` modification aspect:

```
add-to-declaration Wallet {
    uint internal counter = 0;
}
```

- If one would want to make all private functions in the `Wallet` smart contract internal, one can add the following aspects:

```
update-definition Wallet.*(*) tagged-by private {
   add-tag: internal;
   remove-tag: private;
}
```

- In order to make the `CustomToken` an instance of the ERC20 token standard, one would write the following:

```
update-definition CustomToken {
   implement: ERC20;
}
```

2.5 Defining Aspects

Aspects are defined as a sequence of pointcuts and advice meaning to achieve a common goal into one script. For instance, the following example shows how any changes to the variable keeping track of the owner and of whether the wallet is locked or not, can be guarded to ensure that they were initiated by the current wallet owner:

```
aspect SafeWallet {
   before set address Wallet.owner {
     require(msg.sender == owner);
   }

   before set bool Wallet.locked {
      require(msg.sender == owner);
   }
}
```

3 Tool Usage

The process which is being followed involves the parsing of the input AspectSol script file, followed by the parsing of the input Solidity smart contract file. This is followed by the AspectSol execution and the new Smart Contract generation.

1. AspectSol Parsing: The first input to the tool should be a script file containing the code written in the AspectSol scripting language. The script is firstly tokenized, after which the tokens are parsed to create an AST.
2. Smart Contract Parsing: The second input to the tool is a Solidity smart contract. This contract is passed through solc compiler in order to also be parsed into an AST. However, the solc compiler also ensures that going forward we do not work on a smart contract which contains errors. Any errors generated by the solc compiler are displayed in the console for the user to inspect.
3. AspectSol Execution: In this stage, we take the AspectSol AST and execute it based on the outputted smart contract AST. This is where the original smart contract is filtered and updated according to what was written in the AspectSol script.
4. Smart Contract Generation: Finally, once the smart contract AST has been updated accordingly, we again pass the smart contract AST back through solc to be able to re-generate a readable smart contract in Solidity.

3.1 Running AspectSol

After downloading the executable, the following is the procedure of running the tool:

1. Identify path to AspectSol script.
2. Identify path to origin smart contract solidity file.
3. Decide on the path to where the resulting smart contract solidity file should be written to.
4. Open up a terminal window and navigate to the root folder of where the executable was downloaded to.
5. Run the following command:
   `aspectsol.exe` ⟨*path-to-aspect-script*⟩ ⟨*path-to-solidity-file*⟩ ⟨*path-to-output-result*⟩

4 Use Cases

In this section we present three use cases illustrating the use of AspectSol.

4.1 Use Case 1: Safe Variables

One of the core principles of OOP is encapsulation. The general idea of this mechanism is that attributes of a class are kept private within the class. Read and write access to such attributes are made available from outside the class through functions. In OOP languages, such functions are more commonly referred to as getters and setters.

Consider the below solidity smart contract, where a custom token contract is created with a balance property. As it stands, the balance property is annotated with a public modifier, meaning anyone can read and update this property without any form of validation.

```
1  pragma solidity >=0.7.0 <0.9.0;
2
3  contract CustomToken {
4
5    public uint256 balance;
6  }
```

Listing 2: Unsafe Variable Smart Contract.

Let us consider that we want to change this so that reads and writes to the balance property need to go through a function first. Despite achieving the same thing, going through a function as opposed to directly accessing a property might be beneficial in cases were access needs to be given only to specific people, or when validation needs to be carried out prior to updating the value of a property.

```
aspect SafeVariableInjection {
    add-to-declaration CustomToken {
        function getBalance() public view returns(uint256) {
            return balance;
        }

        function setbalance(uint256 b) public {
            balance = b;
        }
    }

    update-definition CustomToken.balance tagged-with public {
        remove-tag: private
        add-tag: public
    }
}
```

Listing 3: Safe Variable AspectSol Script

Executing the above script would result in the original smart contract to be transformed to the below:

```
pragma solidity >=0.7.0 <0.9.0;

contract CustomToken {

  uint256 private balance;

  function getBalance() public view returns(uint256) {
    return balance;
  }

  function setbalance(uint256 b) public {
    balance = b;
  }
}
```

Listing 4: Safe Variable Smart Contract.

4.2 Use Case 2: Ensuring Adherence to Business Process Flow

Writing bug free smart contracts that adheres to their business process flow is of an utmost importance due to the nature of immutability that blockchains operate on. However, as important as it may be, this is very difficult to achieve. What we can do with AspectSol, however is, provide a means of injecting reparations that can be performed in case of a violation to the business process flow.

Consider a Casino smart contract that allows for users to bet a certain amount of cryptocurrency tokens. An example of this contract can be seen below. In such a scenario, a logical or run-time bug might irreparable consequences. Therefore, it would be beneficial post-deployment that reparation actions are exposed.

```
pragma solidity ^0.5.11;

contract Casino{

  mapping(uint => mapping (uint => address payable[])) placedBets;
  mapping(uint => mapping(address => uint)) potShare;

  uint[] numbersGuessed;

  uint pot;
```

```
11
12    uint tableID;
13    uint tableOpenTime;
14
15    address owner;
16
17    constructor() public{
18      owner = msg.sender;
19    }
20
21    function openTable() public{
22      require(msg.sender == owner);
23      require(tableOpenTime == 0);
24
25      tableOpenTime = now;
26      tableID++;
27    }
28
29    function closeTable() public{
30      require(msg.sender == owner);
31      require(pot == 0);
32
33      delete numbersGuessed;
34    }
35
36    function timeoutBet() public{
37      require(msg.sender == owner);
38      require(now - tableOpenTime > 60 minutes);
39      require(pot != 0);
40
41      for (uint i = 0; i < numbersGuessed.length; i++) {
42        uint l = placedBets[tableID][numbersGuessed[i]].length;
43
44        for (uint j = 0; j < l; j++) {
45          address payable better = placedBets[tableID][numbersGuessed[i]][l];
46          better.transfer(potShare[tableID][better]);
47          delete placedBets[tableID][numbersGuessed[i]];
48        }
49      }
50
51      closeTable();
52    }
53
54    function placeBet(uint guessNo) payable public{
55      require(msg.value > 1 ether);
56
57      potShare[tableID][msg.sender] += msg.value;
58      placedBets[tableID][guessNo].push(msg.sender);
59      numbersGuessed.push(guessNo);
60      pot += msg.value;
61    }
62
63    //we assume owner is trusted
64    function resolveBet(uint _secretNumber) public{
65      require(msg.sender == owner);
66
67      uint l = placedBets[tableID][_secretNumber].length;
68      if(l != 0){
69        for (uint i = 0; i < l; i++) {
70          placedBets[tableID][_secretNumber][i].transfer(pot/l);
71        }
72      }
73
```

```
74        pot = 0;
75
76        closeTable();
77      }
78    }
```

Listing 5: Casino Smart Contract.

Figure 1 illustrates how the state of the contract changes depending on which function is called and by whom it was called. This is encoded in the format `party:function`. We can also see how states can only be transitiioned to if and only if the smart contract is in a particular state. For instance, calling the `withdrawFunds()` function is only allowed to be called by the `owner` when the smart contract is in a `NoBet` state. Using AspectSol, we are able to encode this process flow in a script which would inject the necessary code to make this possible.

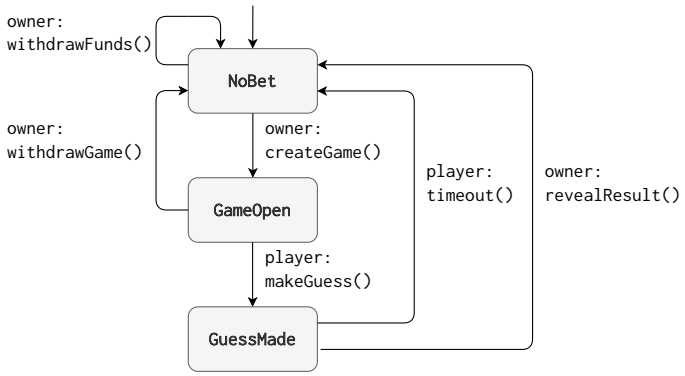

Figure 1: Casino business process

```
1     aspect OpenUntilResolution {
2       enum States {
3         TABLEOPEN,
4         TABLECLOSED,
5         BETPLACED,
6         TABLECLOSEDURINGBET
7       }
8
9       States currentState;
10
11      add-to-declaration Casino.openTable {
12        assert(currentState == States.TABLECLOSED);
13        currentState = States.TABLEOPEN;
14      }
15
16      add-to-declaration Casino.closeTable {
17        assert(currentState == States.TABLEOPEN);
18        currentState = States.TABLECLOSED;
19      }
20
21      add-to-declaration Casino.placeBet {
22        assert(currentState == States.TABLEOPEN);
23        currentState = States.BETPLACED;
24      }
25
26      add-to-declaration Casino.resolveBet {
27        assert(currentState == States.BETPLACED);
28        currentState = States.TABLEOPEN;
29      }
30
31      add-to-declaration Casino.timeoutBet {
32        assert(currentState == States.BETPLACED);
```

```
        currentState = States.TABLEOPEN;
      }
    }
```

Listing 6: Business Process Flow AspectSol Script

4.3 Use Case 3: Reentrancy Free Contract

Possibly one of the most fatal vulnerabilities for smart contracts is that of a reentrancy attack. Such a scenario would involve interaction between two contracts, a vulnerable one containing the reentrancy bug dubbed *Contract A*, and a malicious one, which is ready to exploit the other's vulnerability referred to as *Contract B*. *Contract A* would send funds to *Contract B*, which would trigger a fallback function in *Contract B* that calls back into *Contract A* before it is done updating balances.

A possible way of how you might go about mitigating this scenario, is by creating a boolean flag and use it to ensure that no code is executed while another transaction is currently being executed. Some choose to do this manually where and when needed, while others opt to create a modifier which does this manually every time a function is executed. Both approaches have their disadvantages: the former means that some instances may be unintentionally left out, whilst the latter is the equivalent of carpet bombing, disallowing calls between contract functions and unnecessarily increasing gas costs in functions which clearly do not yield control.

Using AspectSol we show how this can be achieved for invocations to the `transfer` function:

```
aspect SafeReenentrancy {
    private bool running = false;

    before execution-of *.* {
        require (!running);
    }

    before call-to *.transfer() {
        running = true;
    }

    after call-to *.transfer() {
        running = false;
    }
}
```

Listing 7: Reentrancy Mitigation AspectSol Script.

5 AspectSol Errors and Warnings

The following are errors and warnings which AspectSol may report:

10001 Failed to generate a valid token while compiling AspectSol script. You may encounter this error if your AspectSol script contains invalid syntax. Please refer to the line number provided with the error to help identify where the invalid syntax is located.

20001 Encountered a different token from what was expected while compiling the AspectSol script. The error message template for this is `'Expected [`⟨*expected-token*⟩`] but found: [`⟨*actual-token*⟩`]'`. You may encounter this error if your AspectSol script contains invalid syntax. Please refer to the line number provided with the error to help you identify where the invalid syntax is located.

30001 Placement token provided is not yet supported. Valid placement token values are `before` and `after`. Provided line number will indicate where the invalid placement token is located within the AspectSol script.

30002 Location token provided is not yet supported. Valid location token values are `call-to` and `execution-of`. Provided line number will indicate where the invalid location token is located within the AspectSol script.

30003 Variable access token provided is not yet supported. Valid variable access token values are `get` and `set`. Provided line number will indicate where the invalid variable access token is located within the AspectSol script.

30004 Variable visibility token provided is not yet supported. Valid variable visibility token values are `public`, `private` and `internal`. Provided line number will indicate where the invalid variable visibility token is located within the AspectSol script.

30005 Modification type token provided is not yet supported. Valid modification type token values are `add-to-declaration` and `update-definition`. Provided line number will indicate where the invalid modification type token is located within the AspectSol script.

6 Version History

0.1α released 1 August 2022 (First release)