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 N/A, last updated 2 May 2022

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.

AspectSol is a tool which is still in an alpha stage where development on the tool is still ongoing. However, source code is being released as open source to encourage independent extensions and offshoots to the tool.

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.

2.1 Defining Pointcuts

Two types of pointcuts are currently available, an Append Statement Pointcut and a Modification Pointcut. The difference between the two is that while the append statement should be used to add new code found in the pointcut body to the original smart contract, modification pointcuts should only be used to update the existing definition of the original smart contract.

Throughout this section the use of '`*`' is extensively used. This symbol dictates a wildcard usage and is used whenever we want a rule to pass irrespective of the input.

2.1.1 Append Statement Pointcuts

Append Statement Pointcuts are encoded in the following syntax:

```
<Placement> <Location> <Selector> <Sender> {
  ...
}
```

Listing 2: Append Statement Syntax.

Placement The placement property dictates where the scoped code should be placed, i.e. either before or after any of the matched selections:

- `before`: Using this placement property means that the code within the pointcut aspect is triggered before the matched selection is executed. Placing code before the matched selection is beneficial in cases where you want to ensure a condition before accessing a value.
- `after`: On the other hand, using this placement property means that the code within the pointcut is triggered after the matched selection is executed. Such a property can be used in situations where particular action needs to be carried out upon a sequence of logical steps.

Location The location property allows for the specification of call-to and execution-of:

- `call-to`: Dictates whether matched selection should be based on a specific call. Thus, placing scoped code before a specific call to a function would not enforce the scoped code to be executed before every call to the matched selection.
- `execution-of`: As opposed to call-to, this will enforce that scoped code is executed before each call to the matched selection.

Contract, Function & Interface Selectors For the purpose of offering a broader selection process, we have crated a number a selectors which can be used in a pointcut. These include contract, function, variable definition, variable getters and variable setters. All of these can be used interchangeably together to come up with the correct selection that would suite the users' needs.

- Fully Defined Selections: Targeting specific functions within a smart contract are handled by defining both smart contract name and function name in the selection definition. The syntax for this is defined as ⟨*contract-name*⟩`.`⟨*function-name*⟩`(`⟨*parameters*⟩`)`. For instance, to capture calls to the `PlaceBet` function in the `Casino` smart contract, we can write:
  `Casino.PlaceBet()`
- Partially Defined Selections: To handle more generic situations, the use of the wildcard symbol can be applied. The wildcard symbol can be placed instead of the ⟨*contract-name*⟩, ⟨*function-name*⟩ and the ⟨*parameters*⟩. For instance, to capture calls to all methods within the `Casino` smart contract, we can write:
  `Casino.*(*)`

  Similarly, to match all functions names `Withdraw` defined within any smart contract, we can write:
  `*.Withdraw(*)`

- Address Selections: It is also possible to be more specific and replace a smart contract name with a smart contracts' address such that the definition would look like ⟨*contract-address*⟩`.`⟨*function-name*⟩`(`⟨*parameters*⟩`)`. For instance, to capture all calls made to a specific contract address, we can write

  `0x71C7656EC7ab88b098defB751B7401B5f6d8976F.*(*)`

- Interface Selections: To further refine selections, the possibility of matching on implemented interfaces is also available. The syntax for this selector is ⟨*contract-name*⟩`.`⟨*function-name*⟩`::`⟨*interface-name*⟩`(`⟨*parameters*⟩`)`. For example, to match all functions implemented from the `ERC20` interface in the `CustomToken` smart contract, we can write:

  `CustomToken::ERC20.*`

- Mapped Selections: An extension of the wildcard symbol is the use of mappings. These selectors are identical to how partially defined selections work, with the difference that the smart contract and function names can be used within the scope of the aspect. As opposed to the `*` symbol for the wildcard selection, for mapped selections we need to write `[[`⟨*var-name*⟩`]]`. For instance, to match any function within any smart contract and also return and store the contract name, we can write:

  `[[`⟨*var-name*⟩`]].*(*)`

  The above can also be applied to function names, such that the function name is returned and stored in a variable.

  `*.[[`⟨*var-name*⟩`]](*)`

Parameter Selectors It is also possible to define the parameters functions should have to satisfy the selection defined in the pointcut. In such cases, the following options are available:

- No Parameters: If no parameters are provided in the function parenthesis then this would indicate that the selection should occur on those functions that do not have parameters defined. For example, to capture the start of execution of functions named `PlaceBet` within the `Casino` smart contract which do not define any parameters we can write:

  `before execution-of Casino.PlaceBet() { }`

- Fully Defined Parameters: If a parameter is passed within the parenthesis of the function, then the listed parameters need to be present in the function definition in order to satisfy the selection. For example, to capture the successful execution of the `BalanceOf` requiring a parameter of type `address` named `x` within any smart contract, we can write:

  `after execution-of *.BalanceOf(address x) { }`

- Partially Defined Parameters: To facilitate creation of generic aspects which can be reused more than once for various different smart contracts, we are also able to write partially defined parameters within pointcuts. For example, to capture the start of execution of the `Terminate` function in any smart contract which accepts any number of parameters, we can write:

  `before execution-of *.Terminate(*) { }`

Decorator Selectors In conjunction with the above selectors, AspectSol also offers the possibility to chain definition selectors with modifier decorators such as the following:

- `tagged-with` ⟨*tag-expression*⟩: In order to filter functions based on access modifiers (`public`, `private`, `internal`, `external`, `pure`, `view`), modifiers or the `payable` keyword, this pointcut filter allows for identifying what functions to capture based on a combination of tags. A tag expression is a boolean expression over tag names and the boolean operators of conjunction (`&`), disjunction (`|`) and negation (`!`). For example, to capture all public and external functions which use the modifier `byOwner`, but which may not receive funds, we can write:

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

- `returning (`⟨*var-type*⟩`)`: Compliments a function selector statement to indicate what the return types of functions should be. For example, to capture all methods in the `Casino` smart contract that return a `uint256`, we can write:

  `Casino.*(*) returning(uint256) { }`

- `in-interface` ⟨*interface-name*⟩: Compliments any selector statement indicating that matched selection should be defined within the provided interface. For example, to capture all implemented functions from the `ERC20` interface standard in the `CustomToken` smart contract, we can write:

```
CustomToken.*(*) in-interface ERC20 { }
```

- `not-in-interface` ⟨*interface-name*⟩: Compliments any selector statement indicating that matched selection should not be defined within the provided interface. For example, to capture all functions in the `CustomToken` smart contract which are not an implementation from the `ERC20` interface standard, we can write:

```
CustomToken.*(*) not-in-interface ERC20 { }
```

Variable Getter & Setter Selectors When it comes to filtering out variable access, get and set selectors are available. As the name implies, the get selector matches any variable access within a smart contract and the set selector matches any variable modification in the smart contract. Due to the nature of how you would need to carry out the selection of gets and sets, the selectors differ slightly from what has been previously explained above:

- Fully Defined Selections: Syntax for this selector should adhere to ⟨*selection-type*⟩ ⟨*variable-type*⟩ ⟨*contract-name*⟩`.`⟨*variable-name*⟩. The selection can be either `get` or `set`. For instance, if we want to match every time the `savings` variable in the `Wallet` contract is read, we can write:

```
get uint256 Wallet.savings { }
```

  Similarly, if we would want to match every time the `savings` variable in the `Wallet` contract is updated, we can write:

```
set uint256 Wallet.savings { }
```

- Partially Defined Selections: We can apply the same technique with the wildcard symbol as was applied in the contract and function selectors. In this case, the wildcard symbol can be interchanged with the ⟨*variable-type*⟩, ⟨*contract-name*⟩ or the ⟨*variable-name*⟩. For example, to match every time the a `uint256` variable was accessed in any contract, we can write:

```
get uint256 *.* { }
```

- Mapped Selections: Similar to how mapped selections are carried out for contract and function selectors, the wildcard selection is also extended further for variable access and modification statements. For example, to match the modification of the `balance` variable within any contract, without knowing the type from before-hand, we can write:

```
set [[x]] CustomToken.* { }
```

Sender The sender property allows users to specify the originator of the smart contract call to further narrow down the selection process. Variations for this property are:

- `originating-from` ⟨*address*⟩: Initiator of transaction matches the specified address, which can be 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 `owners[0]`:

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

  Note that the variable is evaluated inside the `depositMoney()` function (since `execution-of` is used). If the modality were `after`, the `originating-from` expression is evaluated at the end of the execution of the function.

- `originating-from` ⟨*contract name*⟩: Initiator of transaction matches the specified contract name.
- `originating-from` ⟨*function name*⟩: Initiator of transaction matches the specified function name.
- `originating-from this` and `originating-from !this`: The caller must be (respectively must not be) this same smart contract.

2.1.2 Modification Pointcuts

Modification Pointcuts are encoded in the following syntax:

```
<Type> <Selector> {
  ...
}
```

Listing 3: Modification Statement Syntax.

The type property indicates the type of modification that needs to be carried out on the matched selection.

- `add-to-declaration`: Add the code enclosed within the aspect scope to the matched selection.
- `update-definition`: Update the matched definition with the properties defined within the scope of the aspect. These properties can be one of the following:
  - `add-tag:` ⟨*some-modifier*⟩`;`
  - `remove-tag:` ⟨*some-modifier*⟩`;`
  - `implement:` ⟨*some-interface*⟩`;`

As for the selector property, contract and function selectors described in section 2.1.1 can also be applied here. Furthermore, modifier decorators explained in the same section can be used in conjunction with these selectors.

2.2 Defining Aspects

Aspects enable developers to combine a number of pointcuts meaning to achieve a common goal into one script. Variables can be defined within an aspect which are automatically injected to the smart contract prior to executing the pointcuts. Hence, such variables are made available within a pointcuts' definition and body.

```
aspect AspectExample {
  string type = "uint256";

  before set [[type]] Casino.balance {
    ...
  }
}
```

Listing 4: Aspect Definition Syntax.

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⟩
```

3.2 Errors & Warnings

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.

3.3 Use Cases

3.3.1 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.

```
pragma solidity >=0.7.0 <0.9.0;

contract CustomToken {

  public uint256 balance;
}
```

Listing 5: Non-Safe 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.

Can we do it for all variables i.e. update definitions for `CustomToken.*` but not update the functions? **/ Gordon**

Also, when capturing variable names, can we use them in a new variable name e.g. `set Wallet.[[v]] {` `old_[[v]] = v;` }? / **Gordon**

How does it know that the getter and setter are to be written to the `CustomToken` contract? Should there be an `add-to-declaration`? **/ Gordon**

```
aspect SafeVariableInjection {
  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 6: 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 7: Safe Variable Smart Contract.

3.3.2 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 ˆ0.5.11;

contract Casino{

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

  uint[] numbersGuessed;

  uint pot;

  uint tableID;
  uint tableOpenTime;

  address owner;

  constructor() public{
    owner = msg.sender;
  }

  function openTable() public{
    require(msg.sender == owner);
    require(tableOpenTime == 0);

    tableOpenTime = now;
    tableID++;
  }

  function closeTable() public{
    require(msg.sender == owner);
    require(pot == 0);

    delete numbersGuessed;
  }

  function timeoutBet() public{
    require(msg.sender == owner);
    require(now - tableOpenTime > 60 minutes);
    require(pot != 0);

    for (uint i = 0; i < numbersGuessed.length; i++) {
      uint l = placedBets[tableID][numbersGuessed[i]].length;

      for (uint j = 0; j < l; j++) {
        address payable better = placedBets[tableID][numbersGuessed[i]][l];
        better.transfer(potShare[tableID][better]);
        delete placedBets[tableID][numbersGuessed[i]];
      }
    }

    closeTable();
  }

  function placeBet(uint guessNo) payable public{
    require(msg.value > 1 ether);

    potShare[tableID][msg.sender] += msg.value;
    placedBets[tableID][guessNo].push(msg.sender);
    numbersGuessed.push(guessNo);
    pot += msg.value;
  }

  //we assume owner is trusted
  function resolveBet(uint _secretNumber) public{
    require(msg.sender == owner);

    uint l = placedBets[tableID][_secretNumber].length;
    if(l != 0){
      for (uint i = 0; i < l; i++) {
        placedBets[tableID][_secretNumber][i].transfer(pot/l);
      }
    }

    pot = 0;

    closeTable();
  }
}
```

Listing 8: 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.

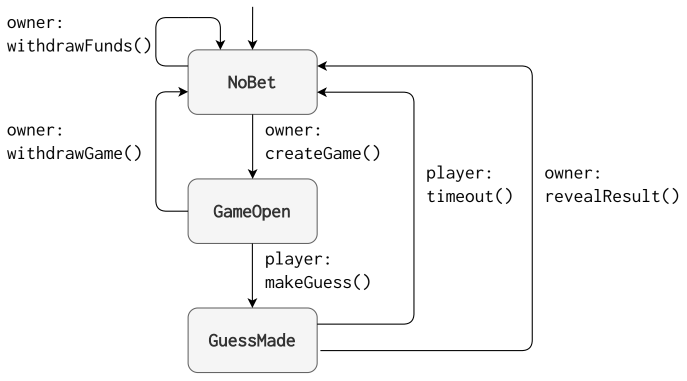

Figure 1: Casino business process

```
aspect OpenUntilResolution {
  enum States {
    TABLEOPEN,
    TABLECLOSED,
    BETPLACED,
    TABLECLOSEDURINGBET
  }

  States currentState;

  add-to-declaration Casino.openTable {
    assert(currentState == States.TABLECLOSED);
    currentState = States.TABLEOPEN;
  }

  add-to-declaration Casino.closeTable {
    assert(currentState == States.TABLEOPEN);
    currentState = States.TABLECLOSED;
  }

  add-to-declaration Casino.placeBet {
    assert(currentState == States.TABLEOPEN);
    currentState = States.BETPLACED;
  }

  add-to-declaration Casino.resolveBet {
    assert(currentState == States.BETPLACED);
    currentState = States.TABLEOPEN;
  }

  add-to-declaration Casino.timeoutBet {
    assert(currentState == States.BETPLACED);
    currentState = States.TABLEOPEN;
  }
}
```

Listing 9: Business Process Flow AspectSol Script

3.3.3 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 SafeReentrancy {
    private bool running = false;

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

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

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

Listing 10: Reentrancy Mitigation AspectSol Script.

4 Version History

0.1α released N/A (First release)