Anduril Maintenance Guide
February 27, 2015

Kristian Ovaska
Contact: kristian.ovaska@helsinki.fi

Contents

1 Introduction **1**

2 Architecture **1**

3 Component network **4**

4 Workflow execution **7**
- 4.1 Node states and the state file . . . 7
- 4.2 Network initialization and execution . . . 9
- 4.3 `switch-case` statements . . . 12
- 4.4 `@bind` annotations . . . 12

5 AndurilScript parser **15**
- 5.1 Symbol table . . . 15
- 5.2 Abstract syntax tree and conversion to network . . . 16
- 5.3 Implementing functions . . . 18

6 Unit testing the engine **20**
- 6.1 State initialization in various conditions . . . 21
- 6.2 State initialization for disabled nodes . . . 22
- 6.3 Network topology for nested functions . . . 23
- 6.4 Network topology for empty functions . . . 25
- 6.5 States in execution of `switch-case` statement . . . 26
- 6.6 Error check testing . . . 28
  - 6.6.1 Function definition . . . 28
  - 6.6.2 Invoking components and functions . . . 29
- 6.7 Other tests . . . 29

1 Introduction

This document describes the internal behaviour of the Anduril workflow engine. The document is intended for those maintaining the engine and interested in its design and detailed architecture.

Section 2 provides an overview of the system and explains the relationships between the AndurilScript parser, bundles, data models in workflows and the executing engine. Section 3 introduces the structures and the key properties of the workflows as networks of dependencies between the components. The execution of and the management of these networks is discussed in Section 4. Section 5 about the configuration parser is relevant for those implementing new features to AndurilScript. The last section is highly recommended for all maintainers of Anduril as it tells how to use tests to confirm that the system is working properly.

2 Architecture

High level architecture of Anduril is illustrated in Figures 1 and 2. Two central data structures are component repository and component instance network (Section 3). The repository stores static interfaces of components and associated data such as data types. These are read from XML files by reader classes. The repository is used to create HTML manual pages of components by manual writer classes. The component instance network represent a workflow and contains references to the component repository. The network is produced by AndurilScript parser from AndurilScript source code (Section 5). The parser uses abstract syntax trees (ASTs) and a symbol table as an intermediate form. The network is executed by the execution engine (Section 4), which also annotates component instances with dynamic state information, stored in a state file. The network is thus a bridge between the parser and the execution engine. The network is also visualized at runtime by the ConfigurationReport component that runs inside the engine.

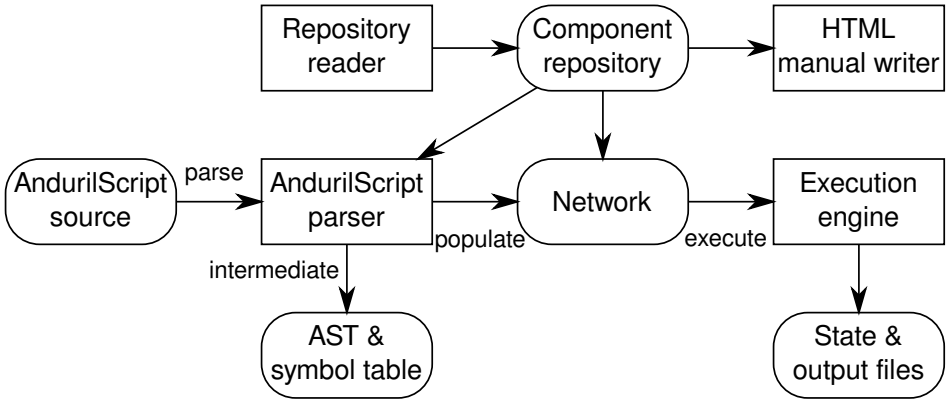


Figure 1: Flow of data in the Anduril framework. Ovals represent data structures and rectangles represent executable subsystems.

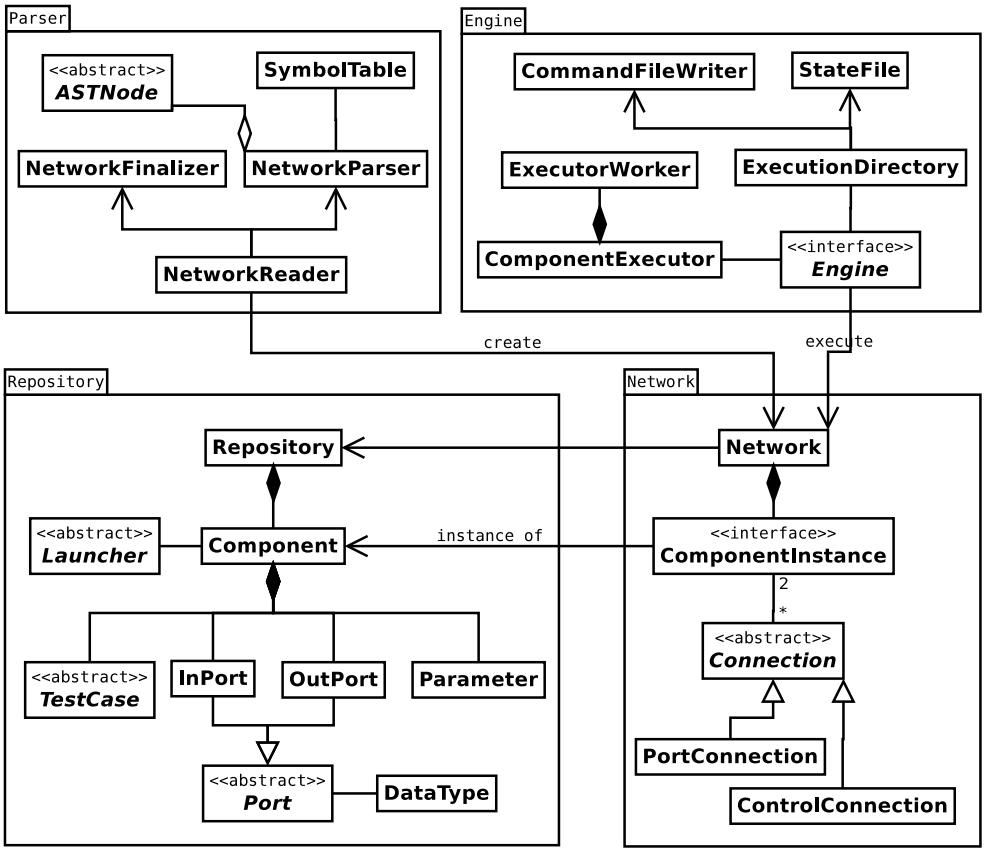


Figure 2: UML class diagram of the central structures of the Anduril engine. `Network` is a cental class that is executed by the engine and created by the parser. Network is a collection of `ComponentInstance` objects. Connections between component instances are represented by subclasses of `Connection`; each connection has a start point and an end point. The `Repository` class stores "static" component interfaces. Each `Component` object is a collection of input and output ports and simple parameters. Components also have a launcher and a number of test cases. `NetworkReader` is the gateway to AndurilScript parser. Behind the scenes, it uses ANTLR-derived `NetworkParser` and `NetworkFinalizer` to populate and initialize a network. The execution `Engine` executes a network with the help of `ComponentExecutor`, which is a singleton monitor that governs worker threads implemented in `ExecutorWorker`. The `ExecutionDirectory` class manipulates files in the execution directory.

3 Component network

A workflow is a directed acyclic graph (DAG) composed of interconnected component instances, or nodes for short. Each node in the network is an instance of a specific component and inherits the static interface of that component. Component interface is defined by input and output ports and simple parameters.

The can be two types of connections between nodes in the network: dependency and hierarchy connections. Of these, dependency connections are the defining feature of workflows: they determine the structure of the DAG. A directed dependency edge $A \rightarrow B$ indicates that A must be executed before B. Dependency connections have two subtypes, port connections and control connections. A port connection $A.out \rightarrow B.in$ indicates that the output of port $A.out$ is directed to the input of port $B.in$. Control connections are "pure" dependencies between nodes where the output of A is not (necessarily) used in B. However, nodes may have both port and control connections, in which case the control connections have no further effect on the dependency.

Definition. The directed dependency network $N = (V, D)$ consists of the set of nodes V and dependency edges $D \subset V \times V$. Nodes A and B have a dependency iff they have one or more port or control connections.

Definition. When edge $(A, B) \in D$, A is the *direct predecessor* of B and B is the *direct successor* of A. When then is a path from A to B, B is *reachable* from A and B *depends* on A. Nodes having no incoming dependencies (in-degree 0) are *source* nodes. Nodes having no outgoing dependencies (out-degree 0) are *sink* nodes.

Hierarchy connections are used to implement nested workflows, or `functions` in AndurilScript. They are also used in `switch-case` statements. The dependency network N is flat; the workflow engine does not care about the hierarchical structure of the workflow when it is executing the network. In fact, it is easier to execute a flat network. However, in some cases (e.g., workflow visualization), the structure must be reconstructed. The hierarchical structure of nodes is stored in hierarchy connections. This is a tree rather than a network. The tree is defined by parent-child links so that $P(A) = B$ if A belongs to function B (or branch node B). In function calls, the node B representing the function is a *virtual* node: it is only a placeholder and is not executed. Top-level nodes T have $P(T) = \emptyset$.

Definition. When $P(A) = B$, B is the *parent* of A and A is an *immediate child* of B. The set of all *children* of B are the immediate children and their children. The level $L(A) \geq 0$ denotes the position of node A in tree; it is the number of parents and grand-parents that the node has. A node with $L = 0$ is a top-level node.

The network contains a Port Substitution Table (PST) that shows which ports of nested nodes correspond to ports of the parent virtual node. Port substitutions of all virtual nodes are present in the global PST. PST is used for those nested nodes that are the entry and exit points of a function; nodes in between do not have entries in the PST. Together with hierarchy links, PST allows to reconstruct the hierarchical structure. The PST is a map from port connections to ports in virtual nodes. Effectively, a PST link is an edge between a port connection and a port of a virtual node.

Dependency connections are allowed between non-virtual/non-virtual and virtual/virtual nodes, but not between a non-virtual and a virtual node. Relationships between virtual and non-virtual nodes are recorded in the PST. Dependencies between virtual nodes do not affect network execution as virtual nodes are not executed; they are only used for propagating @enabled=false annotations. See Section 6.4 for an example of this.

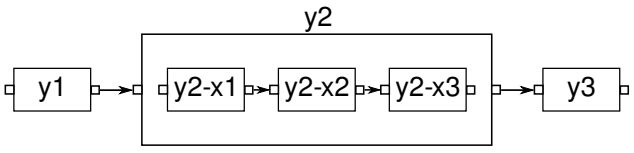
Figure 3 illustrates the dependency network and the hierarchy tree. On conceptual level, `y1` is connected to `y2` and it does not see the nested nodes of `y2`. On actual level, it is connected directly to `y2-x1` and `y2` is a virtual node. When the workflow is visualized, only one level of nodes is shown at a time; in this view, only `y1`, `y2` and `y3` are present on the top level. This structure can be reconstructed using the hierarchy tree. The PST shows that `y2-x1.in` corresponds to `y2.in` because the connection to `y2.in` in the conceptual level is routed to `y2-x1.in` in the actual view. Likewise, `y2-x3.out` is mapped to `y2.out`. If node `y3` were interested in the conceptual source port of its incoming connection, it would query the PST and see that the connection is mapped to the virtual port `y2.out`.

Port mapping is used to ensure that end points of port connections have compatible types. In the example, `y1.out` must have type CSV or its subtype; it is not enough to satisty the type of `y2-x1`, which may be a supertype of CSV. For instance, `y2-x1` could be a fully generic component with no type restrictions.

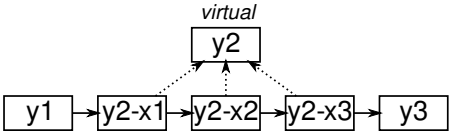
(a)

```
function F(CSV in) -> (CSV out) {
  x1 = SomeComp1(in)
  x2 = SomeComp2(x1.out)
  x3 = SomeComp3(x2.out)
  return x3.out
}
y1 = SomeComp4()
y2 = F(y1.out)
y3 = SomeComp5(y2.out)
```

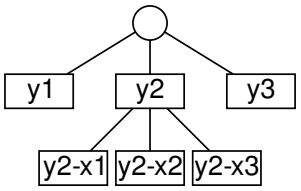
(b)



(c)



(d)



(e)

Connection	ToNode	ToPort	Type
`y1.out => y2-x1.in`	`y2`	`in`	in-port
`y2-x3.out => y3.in`	`y2`	`out`	out-port

Figure 3: *(a)* Example AndurilScript program. It is assumed that all components have one input port named `in` and one output port named `out`. *(b)* Conceptual view of the workflow. Nested nodes are named as `P(X)-X`, where `X` is the node in question and `P(X)` is the name of the parent node. *(c)* Actual view showing dependency edges (solid) and hierarchy edges (dotted). The `y2` node is a virtual placeholder that represents a call to function `F`. *(d)* Hierarchy tree of the workflow, illustrating parent nodes. Levels of `y1`, `y2` and `y3` are 0; children of `y2` are on level 1. *(e)* Port Substitution Table of the network. Note that in-ports are always substituted with in-ports and out-ports with out-ports.

4 Workflow execution

The workflow is executed by launching component instances in any order permitted by dependencies. Only those nodes whose configuration has changed since the last successful execution are re-executed, unless `--force` or `@execute=always` is given. The execution engine only considers dependency edges (D), not hierarchy links.

4.1 Node states and the state file

The execution engine maintains a dynamic state for each node that indicates whether the node has been executed, is waiting to be executed, or is inactivated. The state can be divided into two independent aspects: is the node up-to-date (U) and is it active (A). The state variables are described in Table 1. The state is a pair (U,A), with a total of $2\times 3=6$ combinations. The state A = DISABLED implements `@enabled=false`. The state A = SUSPENDED is used in `switch–case` constructs to suspend non-selected nodes until the branch is executed again. Suspended nodes can not be disabled because this would un-suspend them implicitly: if a suspended node has `@enabled=false`, its A state remains SUSPENDED. Legal state transitions are listed in Table 2.

States is stored in the *state file* located in the execution directory, to be read by the engine on the next run. Only some aspects of state need to be stored. For each node, it is recorded whether U is YES or NO and whether A is YES or SUSPENDED. The state A = DISABLED is not stored because it is a temporary state. If the `@enabled=false` annotation is removed, the node resets back to A = YES. A configuration digest of each

Variable	Value	Description
U	YES	The node has been successfully executed since the last configuration modification. Output files may or may not be present on disk; this is checked by the engine at runtime.
U	NO	Configuration has changed since the last successful execution, or all execution attempts have failed after configuration was changed, or this is a novel node that has never been executed.
A	YES	The node is active on this run and should be executed if U = NO.
A	DISABLED	The node is temporarily disabled on this run and must not be executed. On next run, the node is by default active. The node must be explicitly disabled on each run to keep it inactive.
A	SUSPENDED	The node is permanently inactive and must be explicitly re-activated in order to return it to execution.

Table 1: State variables of nodes. U describes whether the node is up-to-date and A whether it is active.

Var.	From	To	Description
U	-	NO	A novel node has not been executed.
U	NO	YES	The node is successfully executed.
U	YES	NO	Configuration of the node or any of its ancestors has been changed, or the user forced execution with `--force`, or the node has `@execute=always`. If `@execute=once`, the state is not changed. Transition also happens when output files are missing and immediate successor needs to be executed.
A	-	YES	A novel node is initially active.
A	YES	DISABLED	The active node has `disabled=true` annotation on this run.
A	DISABLED	YES	Disabled nodes are reset back to active at the end of network execution.
A	YES, DISABLED	SUSPENDED	The node is part of a switch-case structure and the branch node did not select this choice.
A	SUSPENDED	YES, DISABLED	The branch node begins re-execution and first sets all choice nodes to non-suspension. Nodes that have `disabled=true` on this run are set to DISABLED.

Table 2: State transitions.

node is also stored in the state file. This is a string encoding relevant information on the configuration and allows to observe which nodes have changed.

Annotations that disable nodes (`@enabled=false`) are propagated to child nodes and immediate successors nodes if port connections to mandatory in-ports are present. When node *n* is disabled, it recursively sets `@enabled=false` for all child nodes and all immediate successors for which *n* has a port connection to a mandatory in-port. If only control connections or port connections to optional in-ports are present, the immediate successor is not disabled. A disabled branch node also disables all choice and join nodes.

4.2 Network initialization and execution

Algorithm 1 specifies how node states are initialized before execution. In lines 35–43, the network is iterated from sink nodes to source nodes (following edges backwards) and nodes n with missing output files are handled. If immediate successors of n need to be executed, n must also be executed.

Algorithm 2 is used to execute the network. This algorithm is single-threaded; the multi-threaded version is similar but parallelizes the central `while` loop. The engine maintains four data structures in addition to node states: PRECOUNT, POSTCOUNT, READY and FAILED. The first three of these structures contain only information that could be deduced from node states; their purpose is to make network execution efficient by keeping the state information in a convenient format.

PRECOUNT is a map from nodes to integers that indicates how many immediate predecessors of each node are awaiting execution, i.e., have U = NO and A = YES. A node can only be executed when its PRECOUNT is 0. POSTCOUNT is a similar map for immediate successors. The key in this map is (node, output port). It is used to implement disk space optimization. When POSTCOUNT(n, P) reaches 0, the output files of port P of node n can be deleted if n is marked for disk space optimization. The READY priority queue maintains nodes that are ready for execution. A node n belongs to READY if and only if $U(n)$ = NO and $A(n)$ = YES and PRECOUNT(n) = 0 (and it has not been executed already). That it, the node is not up-to-date, is marked active, and results of all active predecessors are available. Nodes whose execution has failed are stored in the FAILED set; they have U = NO. The engine summarizes failed nodes at the end of network execution.

Algorithm 1 Initializing network state prior to execution. ALWAYS(n) and ONCE(n) denote that node n has `@execute=always` or `@execute=once`, respectively. When computing nodes reachable from a certain node, only active (A=YES) nodes are considered.

for all nodes n **do**
    U(n) = NO
    if n has `@enabled=false` $\vee$ n is virtual **then**
        A(n) = DISABLED
    else
        A(n) = YES
    end if
end for

CHANGED $\leftarrow \emptyset$

// Read state file (optional)
for all nodes n in state file **do**
    if stored A(n) = SUSPENDED in state file **then**
        A(n) $\leftarrow$ SUSPENDED
    end if
end for
for all nodes n in state file **do**
    if (configuration of n has changed $\wedge$ $\neg$ ONCE(n)) $\vee$ ALWAYS(n) **then**
        CHANGED $\leftarrow$ CHANGED $\cup$ (n and nodes reachable from n)
    else if stored U(n) = YES in state file **then**
        U(n) $\leftarrow$ YES
    end if
end for

// Handle `--forced` nodes (optional)
for all nodes n set to forced execution **do**
    CHANGED $\leftarrow$ CHANGED $\cup$ (n and nodes reachable from n)
    CHANGED $\leftarrow$ CHANGED $\cup$ (children of n)
end for

for all nodes n in CHANGED **do**
    if $\neg$ (ONCE(n) $\wedge$ U(n) = YES) **then**
        U(n) $\leftarrow$ NO
    end if
end for
for all nodes n having U(n) = YES, in depth-first order starting from sinks **do**
    if any output file of n is missing **then**
        if any immediate successor has U=NO and A=YES **then**
            U(n) $\leftarrow$ NO
        end if
    end if
end for

Algorithm 2 Executing the network. OPTSPACE(n) indicates that node n is marked for disk space optimization.

```
1:  PRECOUNT(n) ← 0 for all nodes n
2:  POSTCOUNT(n, P) ← 0 for all nodes n and out-ports P
3:  READY ← ∅
4:  FAILED ← ∅
5:  for all nodes n do
6:      if U(n) = NO ∧ A(n) = YES then
7:          for all direct successors s do
8:              PRECOUNT(s) ← PRECOUNT(s) + 1
9:          end for
10:     end if
11:     for all incoming port connections C do
12:         f := C.fromNode; P := C.fromPort;
13:         POSTCOUNT(f, P) ← POSTCOUNT(f, P) + 1
14:     end for
15: end for
16: for all nodes n do
17:     if PRECOUNT(n) = 0 ∧ U(n) = NO ∧ A(n) = YES then
18:         READY ← READY ∪ n
19:     end if
20: end for
21:
22: while READY ≠ ∅ do
23:     n ← pop node from READY according to priority
24:     execute n
25:     if n executed successfully then
26:         U(n) ← YES
27:         if OPTSPACE(n) ∧ n is sink node then
28:             delete output files of n
29:         end if
30:         update state file
31:         for all direct successors s do
32:             PRECOUNT(s) ← PRECOUNT(s) - 1
33:             if PRECOUNT(s) = 0 ∧ U(s) = NO ∧ A(s) = YES then
34:                 READY ← READY ∪ s
35:             end if
36:         end for
37:     else
38:         FAILED ← FAILED ∪ n
39:         if OPTSPACE(n) then
40:             delete output files of n
41:         end if
42:     end if
43:     for all incoming port connections C do
44:         f := C.fromNode; P := C.fromPort;
45:         POSTCOUNT(f, P) ← POSTCOUNT(f, P) - 1
46:         if POSTCOUNT(f, P) = 0 ∧ U(f) = YES ∧ OPTSPACE(f) then
47:             delete output files for f.P
48:         end if
49:     end for
50: end while
```

4.3 `switch-case` statements

Switch-case statements are executed dynamically so that a special *branch* component determines which *choices* are enabled for execution. A *join* node ends the branch. Figure 4 contains an example of this structure. Choice nodes are children of the branch nodes. In this example, abbreviated names of choice names are shown (i.e., `c2-x1` instead of `branch-c2-x1`).

When the network begins execution, PRECOUNT(`join`) = 3 at first. The branch node is executed using a special `ComponentInstance` subclass that first executes the actual code (e.g., R code) similar to regular components, and then it suspends nodes that belong to non-selected choices. The executable code returns a set of enabled choices in a special output port. In this case, possible choices are `c1` and `c2`; it is also possible to select both. If `c1` is selected and `c2` is not, the branch node sets A = YES for `c1` and A = SUSPENDED for all `c2` nodes, with the aid of its data structures. The execution engine provides facilities for modifying node states during runtime. When the engine sets A(`c2-x3`) to SUSPENDED, it also decrements PRECOUNT(`join`) to 2. Now `branch` exits and PRECOUNT(`join`) is further decremented to 1. Node `c1` is executed, which enables the execution of `join`, ending the branch. Output of the `switch-case` structure can be accessed using the join node; it is a normal component, visible to the rest of the network. Choice nodes (`c1` and `c2*`) are not visible to the rest of the network; they reside in their own namespace, only visible to the branch and join components.

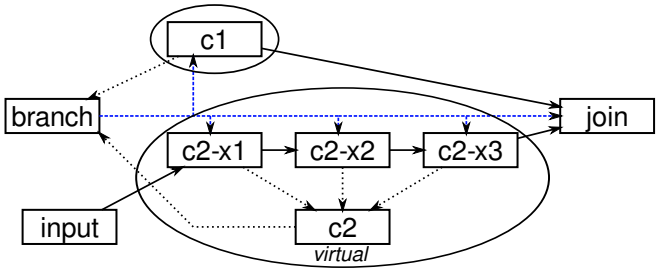
4.4 `@bind` annotations

Manual dependencies may be assigned between components using `@bind` annotations. The annotations are implemented using control connections. Figure 5 shows an example. The simplest, and fully functional, implementation is to include control connection from between all node pairs (A, B), where A appears in source set that has been assigned for the B. An optimized version considers existing port and control connections and adds only control connections between node pairs that are otherwise not connected.

(a)

```
function F(CSV in) -> (CSV out) {
  x1 = SomeComp1(in)
  x2 = SomeComp2(x1.out)
  x3 = SomeComp3(x2.out)
  return x3.out
}
input = SomeComp4()
branch = BranchComponent()
join = switch branch {
  case c1 = SompComp5()
  case c2 = F(input)
  return JoinComponent(c1.out, c2.out)
}
```

(b)



(c)

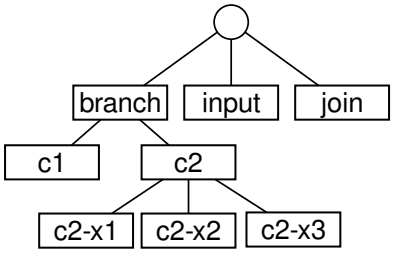
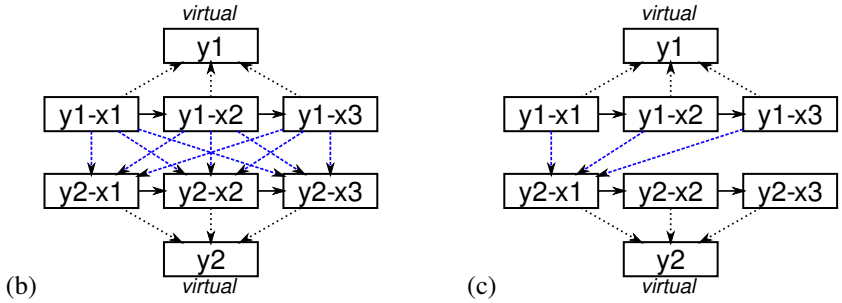


Figure 4: Example of a `switch-case` structure. *(a)* AndurilScript code for the structure. *(b)* The corresponding node network. The branch node has control connections (blue dashed) to all choice nodes (except virtual ones) and the join node. The control connection to the join node prevents execution of the join node before the branch node if outputs of the choices are not routed to the join node. The branch node maintains data structures that hold the set of nodes for each choice (circles). In this case, the set of nodes for choice `c1` is {`c1`} and for choice `c2` it is {`c2`, `c2-x1`, `c2-x2`, `c2-x3`}. Hierarchy links are shown with dotted black lines. *(c)* Hierarchy tree.

(a)

```
function F() -> (CSV out) {
  x1 = SomeComp1()
  x2 = SomeComp2(x1.out)
  x3 = SomeComp3(x2.out)
  return x3.out
}
y1 = F()
y2 = F(@bind=y1)
```



(b) (c)

Figure 5: Example of an `@bind` annotation. *(a)* AndurilScript code for the structure. *(b)* The corresponding node network. Blue dashed lines are control connections and black dotted lines are hierarchy links. *(c)* Optimized node network.

5 AndurilScript parser

Reading AndurilScripts into an executable format consists of two phases: (1) parsing, which creates an abstract syntax tree (AST) and (2) converting the AST into a network data structure. Parsing is done using the ANTLR library. The parser defines the syntax of AndurilScript. The AST is a high-level representation of the syntactic features of AndurilScript code. The AST is accompanied by a symbol table that holds information on entities that can be referred to by name, such as functions and component instances.

5.1 Symbol table

The symbol table stores contents of namespaces, or scopes. Each scope contains a number of named symbols that can refer to components, functions, records, out-ports or simple values (numbers, strings and Booleans). Scopes define the visibility of symbols. All symbols that belong to the same scope are visible in the same context. Scopes may be nested so if a name is not found in the current scope, the parent scope is examined in turn. A scope is defined by (1) a numeric scope ID, (2) ID of parent scope, if any, and (3) the set of symbols in the scope. A symbol is defined by name, type and value. Values are stored in classes that implement the `Value` interface; each type has its own value class.

Symbol types include COMPONENT, FUNCTION, RECORD, PORT, INT, FLOAT, BOOLEAN, STRING and NONE. A symbol of type COMPONENT has a value that is an instance of the `Component` class. Function references of type FUNCTION are defined as (1) the AST nodes that hold the syntactic structure of the function and (2) parent scope ID of the function. These items are encapsulated into `FunctionValue` objects. Simple values (numbers, Booleans and strings) are stored using types INT, FLOAT, BOOLEAN and STRING.

Records (type RECORD) are collections of key-value pairs and are implemented using map data structures. Records do not use the symbol table, although both contain key-value pairs: records are more general because they allow other key types than strings. Also, records do not implement hierarchical name spaces like the symbol table. Records that correspond to the output of a component instance hold a reference to the instance.

References to output ports of component instances in the network are stored in PORT symbols. They hold (1) a reference to the component instance (`ComponentInstance`) and (2) the specific output port (`OutPort`). These are encapsulated into the `PortValue` class.

Figure 6 contains an example AndurilScript program and the corresponding symbol table. The example illustrates that all function calls in AndurilScript are expanded, or inlined, into the main program. The global scope (ID=1) contains references to all available components, although only two are shown in the simplified example. Also, environment variables are imported into their own scope as strings so that they can be referred to as $NAME; this scope is not the parent of any other scope.

Not all names that can be referenced in AndurilScript need to be part of the symbol table. Data types (e.g., CSV in the example) are not present in the symbol table because they are used in a specific context (function definitions) and can be looked up explicitly in this context. Same holds for input ports and parameters of components.

5.2 Abstract syntax tree and conversion to network

Nodes in the AST are represented as instances of `ASTNode`. Subclasses of `ASTNode` represent syntactic structures of AndurilScript; each type of syntactic structure has its own `ASTNode` subclass. The final AST structure is a hierarchy of objects of different `ASTNode` subclasses.

Each AST node contains information of parent and child nodes (if any), source code location and the set of component instances that are created using the specific AST node. Most AST nodes are also instances of `Expression`; expressions have a type and a value. For example, `"abc"` is a `LiteralExpression` with type STRING and value `"abc"`. As another example, `2.5+x` is an `ArithmeticExpression` composed of two sub-expressions and an operator. The value and type of arithmetic expressions depend on its operands and are resolved when the AST is evaluated. Large syntactic structures are instances of `Statement`. In general, anything that can (syntactically) form a valid AndurilScript program when used alone is a statement. The most important statement is `AssignmentStatement` that is defined by a name and an expression. When evaluated, the value of the expression is inserted into the symbol table.

AndurilScript is parsed one top-level statement at a time. In Figure 6, there a five top-level statements. First, an AST is constructed that encapsulates the syntax of the statement. AST construction is done in the constructors of `ASTNode` subclasses. Then, the AST corresponding to the statement is evaluated using `ASTNode.evaluate(SymbolTable, Network)`. Evaluation updates the given network by adding new nodes or connections, or updates the symbol table by adding new symbols. Parsing then proceeds to the next top-level statement. When all statements have been evaluated, the textual representation of AndurilScript has been transformed into a network data structure using AST as an intermediate.

```
x1 = Randomizer(columns=5, rows=5)
function F(CSV in1, optional CSV in2, float f) -> (CSV out1, CSV out2) {
  rec = record(myfield=in1)
  result = CSVFilter(rec.myfield)
  if (in2 == null) myfloat = f + 5
  return record(out1=result.csv, out2=in1)
}
x2 = CSVFilter(x1.matrix)
x3 = F(x2.csv, f=3.2)
x4 = x3.out1
```

```
x1 = Randomizer(columns=5, rows=5)
x2 = CSVFilter(x1.matrix)
{
in1 = x2.csv
in2 = null
f = 3.2
rec = record(myfield=in1)
result = CSVFilter(rec.myfield)
if (in2 == null) myfloat = f + 5
EXPORT x3 = record(out1=result.csv, out2=in1)
}
x4 = x3.out1
```

Scope	Parent	Symbol	Type	Value
1	-	Randomizer	COMPONENT	Component: Randomizer
1	-	CSVFilter	COMPONENT	Component: CSVFilter
2	1	x1	RECORD	RecordValue: matrix (x1)
2	1	F	FUNCTION	FunctionValue: F, parent=2
2	1	x2	RECORD	RecordValue: csv (x2)
3	2	in1	PORT	PortValue: x2.csv
3	2	in2	NULL	null
3	2	f	FLOAT	3.2
3	2	rec	RECORD	RecordValue: myfield
3	2	result	RECORD	RecordValue: csv (x3-result)
3	2	myfloat	FLOAT	8.2
2	1	x3	RECORD	RecordValue: out1, out2 (x3)
2	1	x4	PORT	PortValue: x3-result.csv

Figure 6: *Top*: AndurilScript program. *Middle*: Pseudo code that shows how function calls are expanded (inlined) into the main program. Here, { } denote scope definition and `EXPORT` denotes inserting a symbol to the parent scope. The output of the function is exported to the main scope, while local variables inside the function are not. *Bottom*: Symbol table for the program, in the order that symbols are created.

Before the network can be executed by the engine, it needs to be finalized using `NetworkFinalizer`. Finalization performs modifications and error checking tasks that are not handled by the parser. For example, type parameters of generic components are inferred by the finalizer. See API documentation of `NetworkFinalizer` for details on the transformations.

5.3 Implementing functions

How `function` calls are implemented at AST-level is shown in Algorithm 3. The majority of the code is located in `CallExpression` which evaluates a stored `FunctionValue` instance. Local variables of the inlined function call, as well as function parameters, are stored into temporary scope *LOCAL*; results of the function call are stored in *RESULT*, which is a record that the function call defines. The symbol table maintains a call stack where each function call is represented by a `CallStackElement` object. When the body contains a `ReturnStatement`, the statement finds the result record (*RESULT*) using the call stack and inserts returned values into that record. Lines prefixed with `ReturnStatement:` are located in the `ReturnStatement` class.

Algorithm 3 Implementing function calls in AST.

```
1:  AST := current CallExpression AST node
2:  n ← new virtual component instance
3:  add new namespace LOCAL to symbol table (parent=FunctionValue.parent)
4:  RESULT ← new record value
5:  add n to network
6:  add RESULT to symbol table with type RECORD
7:
8:  for all parameters p of function do
9:      insert symbol p into LOCAL; value is from call expression or function default
10: end for
11: for all in-ports i of function do
12:     insert symbol i into LOCAL as PortValue; value is from call expression or null
13: end for
14:
15: push (AST, RESULT) to call stack
16: push LOCAL into current namespace stack
17:
18: for all statements s in body do
19:     recursively evaluate s
20:     if evaluating a ReturnStatement then
21:         ReturnStatement: (R_AST, R_RESULT) ← top of call stack
22:         for all fields f in return record do
23:             ReturnStatement: insert symbol f into R_RESULT
24:         end for
25:         interrupt evaluation of body
26:     end if
27: end for
28:
29: pop LOCAL from current namespace stack
30: pop (AST, RESULT) from call stack
```

6 Unit testing the engine

Unit tests are executed inside the core and have access to network and engine data structures. Unit tests supplement regular test networks by enabling more detailed validation of engine and network functionality. On the other hand, their construction and running requires writing Java code, which makes implementing them more tedious than regular test networks. The code is located in the core package `.core.unittest`. Unit tests are executed with `anduril unittest`. This section documents the central unit tests.

The unit test system uses only components and data types from a specific bundle, techtest, that has been constructed to support unit testing. The components in the bundle do not have any ”real” analysis functionality. Data types and components in techtest and shown in Figure 7 and Table 3. All components (except C4) can be set to fail with the `fail` parameter. The `dummy` parameter is used to change configuration digest. The `delay` parameter enables timing testing. The components write valid but dummy output to their output ports. C3 is a branch component with two choices, `c1` and `c2`. Which choices are enabled is controlled with Boolean `enable1` and `enable2` parameters.

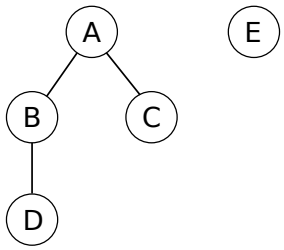


Figure 7: Data type hierarchy in the techtest bundle.

Name	Inputs	Outputs	Parameters	Notes
C1	in1 (*T*, optional)	out1 (*T*)	fail=false (boolean), dummy=0 (int), delay=0 (float)	Type parameter *T*
C2	in1 (B), in2–in5 (B, optional)	out1 (B)	See C1	Same parameters as C1; mandatory port
C3	in1–in5 (B, optional)	out1 (B)	See C1 & enable1=true (boolean), enable2=true (boolean)	Branch component
C4	in1 (B), in2 (E, optional)	out1 (B), out2 (E)	p1=0 (int), p2 (float)	Used in 6.6

Table 3: Components in the techtest bundle

6.1 State initialization in various conditions

Method: `StateInitTest.testStateInitialization`
Engine state initialization is tested with a very simple network consisting of two nodes, a and b, so that $a \rightarrow b$. The network is executed two times and up-to-date states right before the second execution are inspected. This is done after the state file has been read and the engine is ready for execution. Node attributes are manipulated so that different scenarios are simulated. Manipulations are:

- b.fail1: b fails at first execution
- a.changed2: configuration of a changes in second execution
- b.changed2: configuration of b changes in second execution
- a.keep: a is marked/not marked for space optimization in both executions
- b.execute: the @execute annotation of b is set to different values in both executions

Expected values of U(a) (top) and U(b) (bottom) at the beginning of second execution are shown below. Asterisk ($\star$) denotes any value.

b.fail1	a.changed2	b.changed2	a.keep	b.execute	U(a, 2)
$\star$	yes	$\star$	$\star$	$\star$	NO
yes	$\star$	$\star$	no	$\star$	NO
$\star$	$\star$	yes	no	$\star$	NO
$\star$	$\star$	$\star$	no	always	NO
no	$\star$	$\star$	no	changed	YES
no	$\star$	$\star$	no	once	YES
no	$\star$	yes	no	always	NO
no	$\star$	yes	no	changed	NO
no	$\star$	yes	no	once	YES
$\star$	no	$\star$	yes	$\star$	YES

b.fail1	a.changed2	b.changed2	a.keep	b.execute	U(b, 2)
no	no	no	$\star$	always	NO
no	no	no	$\star$	changed	YES
no	no	no	$\star$	once	YES
no	no	yes	$\star$	always	NO
no	no	yes	$\star$	changed	NO
no	no	yes	$\star$	once	YES
no	yes	$\star$	$\star$	always	NO
no	yes	$\star$	$\star$	changed	NO
no	yes	$\star$	$\star$	once	YES
yes	$\star$	$\star$	$\star$	$\star$	NO

6.2 State initialization for disabled nodes

Method: `StateInitTest.testDisabled`

State initialization with @enabled annotations are tested with a network consisting of a single node. The network is executed three times and state at the beginning of second (S_2) and third (S_3) executions are inspected. Here, S(n) denotes the combined state $U(n)/A(n)$. The node is manipulated as follows: disabled on first run (disabled1); disabled on second run (disabled2); configuration changed in second run (changed2). Expected states are as follows.

disabled1	disabled2	changed2	S_2	S_3
no	no	no	YES/YES	YES/YES
no	no	yes	NO/YES	YES/YES
no	yes	no	YES/DISABLED	YES/YES
no	yes	yes	NO/DISABLED	NO/YES
yes	no	$\star$	NO/YES	YES/YES
yes	yes	$\star$	NO/DISABLED	NO/YES

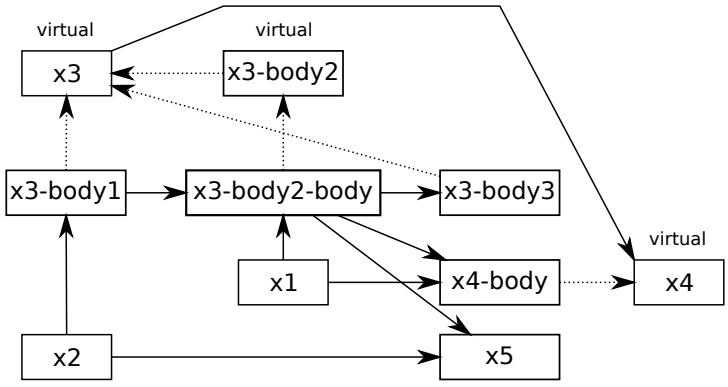
6.3 Network topology for nested functions

Method: `TopologyTest.testNestedFunctions`
The network below tests that connections, parent links and the Port Substitution Table are correct for a network that contains nested functions two levels deep. This network is not executed. The network contains links between nodes on different levels (levels $0 \to 0$, $0 \to 1$, $0 \to 2$, $1 \to 2$, $2 \to 0$ and $2 \to 1$). There is also a node (`x1`) that is used in a function without being routed through a port. Inside `F1`, output ports of `body1` and `body2` are referenced without explicit port name, while explicit port names are given in `x4` and `x5`.

Test network:

```
x1 = C1()
function F1(B in1) -> (B out1, B out2) {
    body1 = C2(in1)
    body2 = F2(body1)
    body3 = C2(body2)
    return record(out2=body2, out1=in1)
}
function F2(B in1) -> (B out1) {
    body = C2(in1, x1)
    return body.out1
}
x2 = C1()
x3 = F1(x2, @execute="once")
x4 = F2(x3.out2)
x5 = C2(in2=x3.out2, in1=x3.out1)
```

Expected network topology. Solid lines are port connections and dashed lines are parent links.



Expected Port Substitution Table:

Connection	ToNode	ToPort	Type
x2.out1 => x3-body1.in1	x3	in1	in-port
x2.out1 => x5.in1	x3	in1	in-port
x3-body1.ou1 => x3-body2-body.in1	x3-body2	in1	in-port
x3-body2-body.out1 => x4-body.in1	x4	in1	in-port
x2.out1 => x5.in1	x3	out1	out-port
x3-body2-body.out1 => x3-body3.in1	x3-body2	out1	out-port
x3-body2-body.out1 => x4-body.in1	x3	out2	out-port
x3-body2-body.out1 => x5.in1	x3	out2	out-port

Method: `TopologyTest.testNestedFunctionsAnnotations`
After basic topology has been validated, a second test is performed where `x2` is disabled by setting `@enabled=false`. Together with the `@execute` annotation of `x3`, these test that annotations are propagated correctly. Expected annotations are in the following table.

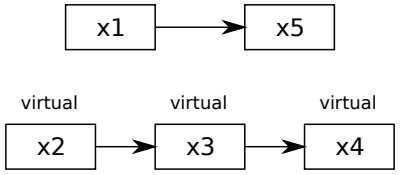
Node	@enabled	@execute
x1	true	changed
x2	false	changed
x3	false	once
x3-body1	false	once
x3-body2	false	once
x3-body2-body	false	once
x3-body3	false	once
x4	false	changed
x4-body	false	changed
x5	false	changed

Notice that disabling `x2` disables all nodes except `x1`, including the virtual nodes, even though there is no path from `x2` to any virtual node (parent links are followed backwards in disable propagation). The engine uses the PST to disable virtual nodes as it observes that the connection `x2.out1 => x3-body1.in1` has a substitution for the port `x3.in1`, which leads to disabling `x3` and its children as well as `x4`. Disabling virtual nodes does not have a sematic effect in this network, but it is important when a function contains a node that does not depend on other nodes in the network; it is disabled through the parent link.

6.4 Network topology for empty functions

Method: `TopologyTest.testEmptyFunctions`
The following network tests a sequence of empty functions with no body. The network is not executed. Notice that `x3` is not present in the PST because it only has connections with other virtual nodes. *Top*: Test network. *Middle*: Expected network topology. Note that there are no parent links. *Bottom*: Expected Port Substitution Table.

```
function Empty(B in) -> (B out) {
    return in
}
x1 = C1()
x2 = Empty(x1)
x3 = Empty(x2.out, @enabled=false)
x4 = Empty(x3.out)
x5 = C2(x4.out)
```



Connection	ToNode	ToPort	Type
`x1.out1 => x5.in1`	`x2`	`in`	in-port
`x1.out1 => x5.in1`	`x4`	`out`	out-port

This network also tests that @enabled=false annotations are propagated correctly. Connections between virtual nodes are elementary for correct propagation in this network. Expected annotations are as follows.

Node	@enabled
`x1`	true
`x2`	true
`x3`	false
`x4`	false
`x5`	false

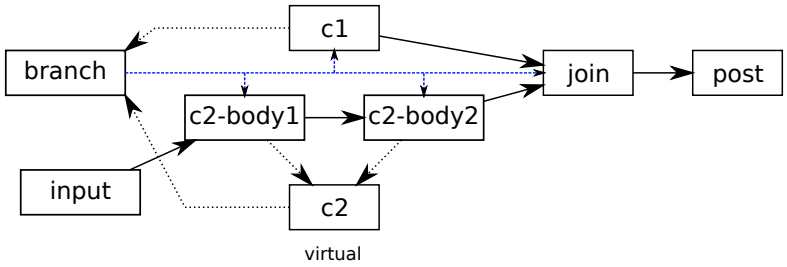
6.5 States in execution of `switch-case` statement

Method: `BranchTest.testTopology` and `BranchTest.testDynamic`
The following network tests the dynamics of `switch-case` statements. The network is executed several times and manipulated between rounds. The state of the previous round is used to initialize the next round. State of nodes are inspected at the beginning and end of each execution round.

Test network:

```
function F1(B in) -> (B out) {
    body1 = C2(in)
    body2 = C2(body1.out1)
    return record(out=body2.out1)
}
input = C1()
branch = C3()
join = switch (branch) {
  case c1 = C1()
  case c2 = F1(input)
  return C2(c1.out1, c2.out)
}
post = C2(join.out1)
```

Expected network topology. Control connections are shown with blue lines. Note that `branch` is a parent of `c1` and `c2`; full names of the choice nodes start with "`branch-`" but abbreviated names are shown.



Expected states are as follows. E=c1 denotes that choice `c1` is enabled by the branch node. S(c2⋆) denotes states of `c2-body1` and `c2-body2` but the state of `c2` itself is not checked for simplicity.

#	Manipulation	State (begin)	State (end)
1	Novel network, E=c1	S(⋆)=NO/YES	S(c2⋆)=NO/SUSPENDED, S(*other*)=YES/YES
2	No change	S(c2⋆)=NO/SUSPENDED, S(*other*)=YES/YES	S(c2⋆)=NO/SUSPENDED, S(*other*)=YES/YES
3	Conf. changed: input	S(input)=NO/YES, S(c2⋆)=NO/SUSPENDED, S(other)=YES/YES	S(c2⋆)=NO/SUSPENDED, S(*other*)=YES/YES
4	Conf. changed: c1	S(c1,join,post)=NO/YES, S(c2⋆)=NO/SUSPENDED, S(*other*)=YES/YES	S(c2⋆)=NO/SUSPENDED, S(*other*)=YES/YES
5	E=c2	S(branch)=NO/YES, S(c2⋆)=NO/SUSPENDED, S(c1,join,post)=NO/YES, S(*other*)=YES/YES	S(c1)=NO/SUSPENDED, S(*other*)=YES/YES
6	Conf. changed: input	S(input,c2⋆,join,post)=NO/YES, S(c1)=NO/SUSPENDED, S(*other*)=YES/YES	S(c1)=NO/SUSPENDED, S(*other*)=YES/YES
7	Conf. changed: c1	S(c1)=NO/SUSPENDED, S(*other*)=YES/YES	S(c1)=NO/SUSPENDED, S(*other*)=YES/YES
8	E=c1,c2	S(input)=YES/YES, S(c1)=NO/SUSPENDED, S(*other*)=NO/YES	S(*all*)=YES/YES

6.6 Error check testing

The following unit tests ensure that incorrect AndurilScript code is recognized and an error message is given. The error message must be attached to the network data structure; the parser must not crash.

6.6.1 Function definition

Method: `ErrorCheckTest.testFunctionDefinition`
Various possible errors in `function` definition are tested with the following test cases. The tests are based on the following simple network, that is modified by the test cases by introducing errors. There is one error in each test case.

```
function F(B in1, optional E in2, int p1, float p2=0)
  -> (B out1, E out2)
{ return record(out1=in1, out2=in2) }
x1 = C1() // Type parameter T gets value B
x2 = C1() // Type parameter T gets value E
x3 = F(x1, x2, p1=5)
x4 = C2(x3.out1)
```

Introduced errors are as follows:

Error description	Code
Optional in-port before mandatory in-port	`function F(optional E in2, B in1, int p1, float p2=0)`
Parameter before ports	`function F(int p1, B in1, optional E in2, float p2=0)`
Two in-ports with same name	`function F(B in1, optional B in1, int p1, float p2=0)`
Two parameters with same name	`function F(B in1, optional E in2, int p1, float p1=0)`
Two out-ports with same name	`-> (B out1, B out1, E out2)`
Default expression with invalid type	`function F(B in1, optional E in2, int p1, float p2="abc")`
Default expression for in-port	`function F(B in1, optional E in2=5, int p1, float p2=0)`
Optional parameter	`function F(B in1, optional E in2, int p1, optional float p2=0)`
Optional out-port	`-> (B out1, optional B out2)`
Invalid in-port type	`function F(NotFound in1, optional E in2, int p1, float p2=0)`
Invalid out-port type	`-> (NotFound out1, B out2)`
Elementary type for out-port	`-> (B out1, int out2)`
Missing entry in return record	`{ return record(out1=in1) }`
Extra entry in return record	`{ return record(out1=in1, out2=in2, extra=in1) }`
Invalid type in return record	`{ return record(out1=in2, out2=in1) }`
Invalid name in return record	`{ return record(x=in2, y=in1) }`
Elementary type in return record	`{ return record(out1=in2, out2=5) }`

6.6.2 Invoking components and functions

Method: `ErrorCheckTest.testInvoke`
Following tests introduce errors into component or function invocation. Test network is as follows:

```
x1 = C1() // Type parameter T gets value B
x2 = C1() // Type parameter T gets value E
x3 = X(x1, x2, p1=5, p2=1.5)
```

There are two variants of these tests, component and function invocation. In the former, `X = C4`, and in the latter, `X = F`, where `F` is the function defined in the previous section. Introduced errors are as follows:

Error description	Code
Empty call arguments	`x3 = X()`
Bare integer as argument	`x3 = X(1)`
Invalid type for in-port	`x3 = X(x1, x1, p1=5, p2=1.5)`
Missing connection to mandatory port	`x3 = X(in2=x2, p1=5, p2=1.5)`
Missing parameter value	`x3 = X(x1, x2, p2=1.5)`
Elementary type for port	`x3 = X(x1, x2=9, p1=5, p2=1.5)`
Double entry for port	`x3 = X(x1, in1=x1, p1=5, p2=1.5)`
Double entry for parameter	`x3 = X(x1, x2, p1=5, p2=1.5, p2=1.5)`
Parameter name omitted	`x3 = X(x1, x2, p1=5, 1.5)`
Parameter before port	`x3 = X(x1, p1=5, x2, p2=1.5)`
Invalid type for parameter	`x3 = X(x1, x2, p1=5, p2="abc")`
Connection into parameter	`x3 = X(in1=x1, in2=x2, p1=5, p2=x1)`
Missing expression for parameter	`x3 = X(in1=x1, in2=x2, p1=5, p2)`

6.7 Other tests

`VisualizerTest` uses networks from `TopologyTest` and `BranchTest` to test the GraphViz visualizer. See Javadocs for details.