Technical manual

Contents

Chapter 1

Programming in the deck: UDQ

UDQ and ACTIONX are two keywords which offer a sort of *programming* in the input deck. UDQ is an acronym for *User Defined Quantity*, and the essence of the UDQ keyword is the ability to define arithmetic expressions based on the result vectors of the ongoing simulation. The quantites evaluated with UDQ can then be output as summary variables, and they can be used as controls in keywords like WCONPROD and GCONINJE. When used as controls the UDQ variables are called *User Defined Arguments* (UDA).

For both the UDQ and ACTIONX keywords evaluating an arithmetic expression based on the current results of the ongoing simulation is an important part of the concept, and they are often referenced in pair as UDQ/ACTIONX, this is slightly misleading as the two are fully independent and share very little both as concepts in flow and in the C++ implementation. The ACTIONX keyword is described in chapter 2.

The UDQ keyword is also documented in section 12.3.233 in the flow reference manual.

1.1 Defining a new UDQ keyword

New UDQ variables are defined with the UDQ keyword in the SCHEDULE section. The UDQ keyword is a sort of a super keyword with four additional subcommands: DEFINE, ASSIGN, UNIT and UPDATE. The DEFINE subcommand is the most important command, that is used to *define* an arithmetic expression for a UDQ variable, which is evaluated at the end of every simulator time step. ASSIGN is used to define a UDQ variable with a constant numerical value, in addition the ASSIGN subcommand is often used as a "forward reference" to enable using a UDQ keyword in another DEFINE expression. The UNIT command is used to assign a unit string to a UDQ variable, see section 1.5 for more details about UDQ variables and units. The UPDATE ON and UPDATE

OFF commands can be used to switch updating of UDQ variables on and off. The UDQ variables are created in mode ON.

All UDQ keywords must have the letter U as their second character, the first character should be 'F', 'G' or 'W' to indicate whether this is a field, group or well quantity[1].

The simplest way to define a UDQ is just using the ASSIGN subcommand of the UDQ keyword:

```
UDQ
  ASSIGN FUVAR1 123 /
  ASSIGN FUVAR2 456 /
/
```

This way we will assign to variables FUVAR1 and FUVAR2 with values 123 and 456 respectively. These values can output to the summary file, be used as control values in a control keyword like WELTARG and be used to recursively define another UDQ keyword.

The more interesting UDQ subcommand is DEFINE which is used to define an arithmetic expression for a UDQ variable, the arithmetic expression will be evaluated at the end of every timestep. The expressions are built from the normal arithmetic operators +,-,*,/, a predefined set of available functions (see 1.3) and results from the ongoing simulation.

In the example below we create UDQ variables FUWCT1 and FUWCT2 as user defined field water cut, one based on the summary variables FWPR and FOPR and one based on summing WOPR and WWPR over all wells

```
UDQ
  DEFINE FUWCT1 FWPR/(FWPR + FOPR) /
  DEFINE FUWCT2 SUM(WWPR)/(SUM(WWPR) + SUM(WOPR)) /
/
```

A UDQ variable can be redefined during the simulation, and also change from constant ASSIGN to variable DEFINE. Observe that UDQ values are always evaluated in order of occurence in the input deck. For instance for this input

```
UDQ
  ASSIGN FU1 100 /
  DEFINE FU2 FU1 + FOPR /
  DEFINE FU1 FWPR /
/
```

the evaluation order will be {FU1, FU2}[2]. ASSIGN statements take effect at input time, making the newly defined symbol immediately available for

[1] In Eclipse also the characters 'S', 'C', 'A' and 'B' are used to denote *segment*, *connection aquifer* and *block* variables respectively. None of these are supported in flow.

[2] Maintaining the input order and whether a certain UDQ symbol is now a ASSIGN or DEFINE keyword is also very important for the restart code.

reuse in a subsequent defintion. This is utilized in the UDQ keyword above where the symbol FU1 is referenced in the defintion of FU2. As used in this example the ASSIGN FU1 100 can be seen as a *forward reference*. When we have completed the first timestep and it is time to evalute the UDQ expressions they will be evaluated in order of *first appearance*, i.e. {FU1, FU2}. At this stage the active definition of FU1 is FU1 = FWPR, i.e. the value 100 from the initial definition FU1 = 100 is never actually used.

1.2 Different types of UDQ - field, group and well

The UDQ keywords can be of different types, flow supports *field*, *well* and *group* keywords[3]. The field keywords are scalar, whereas the well and group keywords are sets with values for every well/group.

The type of a UDQ keyword is inferred from the name in the same manner as the summary keywords, i.e. for this UDQ keyword

```
UDQ
  ASSIGN FU1 100 /
  ASSIGN WU1 200 /
  ASSIGN GU1 300 /
/
```

we will define one scalar keyword FU1, one well set WU1 and one group set GU1. The well sets will have one slot for each well in the model, it is not possible to create a well set with only a subset of wells, but the well set can have *undefined* value for a subset of wells. The set maintains a is_defined() status for each element and most operations only apply to the defined elements[4].

As indicated with ASSIGN WU1 200 you can assign a scalar value to a set keyword, then all elements in the set will have the same value. Assuming we have a model with wells OP1, OP2, WI and GI the WU1 will look like

```
  WU1 = {"OP1": 100, "OP2": 100, "WI": 100, "GI": 100}.
```

When defining a well UDQ it is natural to refer to well summary variables, in the example below we define WUBHP which is the bottom hole pressure for each well, WUOPR1 which is the oil production rate for a subset of wells and WUOPR2 which is the oil production rate for well OP1 *broadcasted to all wells*.

```
UDQ
  DEFINE WUBHP  WBHP     /
```

[3] Eclipse also supports connection, segment and aquifer variables.

[4] The C++ implementation of the UDQ value set is the equivalent of std::map<std::string, std::optional<double>>.

```
  DEFINE WUOPR1 WOPR 'OP*' /
  DEFINE WUOPR2 WOPR OP1 /
/
```

After evaluation these UDQ values will be[5]:

```
  WUBHP  = {"OP1": 20, "OP2": 30, "WI": 10,  "GI": 15}
  WUOPR1 = {"OP1": 13, "OP2": 17, "WI": [ ], "GI": [ ]}
  WUOPR2 = {"OP1": 13, "OP2": 13, "WI": 13,  "GI": 13}
```

Observe how well variables like WBHP and WOPR can be qualified with a well-name pattern. If no wellname is supplied the expression will be evaluated for all wells, as for WUBHP, for WUOPR1 the pattern 'OP*' will select wells {OP1, OP2} and leave the injectors {WI, GI} undefined. For WUOPR2 the well pattern 'OP1' specifies a well completely, i.e. this will be evaluated as a scalar, and then the scalar value WOPR:OP1 == 13 is broadcasted to all the wells in WUOPR2.

UDQ sets with different sets of defined wells can be combined, in most cases the operations will be applied to the intersection of all defined wells, consider for example:

```
UDQ
  DEFINE WUBHP    WBHP /
  DEFINE WUTHP    WTHP OP* /
  DEFINE WUPDIFF  WUBHP - WUTHP
/
```

When these UDQ statements are evaluated we will get:

```
  WUBHP   = {"OP1": 20, "OP2": 30, "WI": 10,  "GI": 15}
  WUTHP   = {"OP1": 13, "OP2": 17, "WI": [ ], "GI": [ ]}
  WUPDIFF = {"OP1":  7, "OP2": 13, "WI": [ ], "GI": [ ]}
```

As we see the undefined property for wells GI and WI in WUTHP is contagious when the two expressions are combined with WUPDIFF = WUBHP - WUTHP, see however section 1.3.4 for a collection of functions which operate on the union of values.

When an undefined variable is output to the summary file it will get a value given as item 3 of the UDQPARAM keyword. Summary output is the *only* point where the undefined value can be dereferenced, the numerical value given in UDQPARAM will not be available either for UDQ nor ACTIONX evaluations.

[5] The numerical values are arbitrary, just to illustrate that they are *defined*, in contrast to the [ ] which is used to illustrate an *undefined* value.

1.3 Functions available in UDQ defintions

flow supports all the UDQ functions available in Eclipse, for the future it would be a quite simple C++ task to extend the list of available functions, although that will affect Eclipse compatibility.

1.3.1 Ordinary binary functions

The UDQ framework supports all the ordinary arithmetic operators +,-,*,/ and ^ and the comparison operators $>, \geq, <, \leq, \neq, ==$. For all of these operations sets and scalars can be combined freely. When combining a scalar and a set the scalar will be promoted to a set with all values equal, i.e. for

```
UDQ
  ASSIGN WULIMIT 100 /
/
```

the numerical value 100 will be broadcasted to all wells:

```
  WULIMIT = {"OP1": 100, "OP2": 100, "WI": 100,  "GI": 100}.
```

In the case of set arguments the operations are only applied to the union of values which are defined in both argument sets. The comparison operators produce numerical 0 or 1 depending on the result of the comparison, the result from a comparison operation can be used numerically in further computations. In the following example FUWCNT will be the number of wells producing with oil production rate above 1000[6].

```
UDQ
  DEFINE FUWCNT SUM( WOPR > 1000 ) /
/
```

1.3.2 Functions over a set returning a scalar

The UDQ framework has several functions which iterate over all the defined members of a set and return a scalar. In the example

```
UDQ
  DEFINE FU1 SUM(WOPR 'OP*') /
/
```

The scalar variable FU1 will be equal to the sum of oil production rates for all wells matching the pattern OP*, the wells with a name not matching OP* will not contribute to the sum. The complete list of functions iterating over a set and returning a scalar are:

[6] Remember: 1000 has the units of the deck, i.e. stb/day in FIELD units and m^3/day in metric units.

SUM Sum all defined elements in the argument set and return a scalar.

AVEA The arithmetic average of the elements in the set.

AVEG The geometric average of the elements in the set.

AVEH The harmonic average of the elements in the set.

MAX The maximum element in the set.

MIN The minimum element in the set.

NORM1 The L^1 norm of the elements in the set.

NORM2 The L^2 norm of the elements in the set.

NORMI The L^∞ norm of the elements in the set.

PROD The product of all the elements in the set.

1.3.3 Elemental functions

There is a family of functions which take a vector or scalar as argument, run through all the elements in the argument and return a result of the same shape as the argument, where a function has been applied to all the defined elements. Assume that the `WU1` has the following value:

```
WU1 = {"OP1": -2, "OP2": -1, "WI": 1, "GI": []},
```

then `DEFINE WUABS ABS(WU1)` will give

```
WUABS = {"OP1": 2, "OP2": 1, "WI": 1, "GI": []}.
```

The available elemental functions of this kind come in different categories:

Mathematical functions

EXP Return a new set where all defined elements have been exponentiated.

ABS Return a new set with the absolute value of all elements.

LN Return a new set with the *natural logarithm* of all elements.

LOG Return a new set with $\log_{10}$ of all elements.

NINT Return a new set where all elements have been converted to the nearest integer.

Sorting functions

The UDQ functionality supports functions SORTA and SORTD to sort a set in ascending or descending order respectively. Observe that these functions do not sort the sets in place - the udqset does not have any notion of ordering, rather they create a permutation set with values 1,2,3, ... The following combination of UDQ and ACTIONX will use the SORTD function to close the two wells with the highest watercut:

```
UDQ
  DEFINE WUWCTS SORTD(WWCT OP*) /
/
```

assuming the WWCT looks like

```
  WWCT = {"OP1": 0.5, "OP2": 0.2, "OP3": 0.1, "OP4":0.7, "WI": 0, "GI": 0}
```

then the WUWCTS set will look like

```
  WUWCTS = {"OP1": 2, "OP2": 3, "OP3": 4, "OP4":1 "WI": [], "GI": []}
```

When this is combined with the ACTIONX(see chapter 2)

```
ACTIONX
CW /
WUWCTS <= 2 /
/

WELOPEN
'?'  'CLOSE' /
/

ENDACTIO
```

the two wells with highest watercut will be selected in the WUWCTS <= 2 statement and that will be expanded to {OP4, OP1} by the '?' expression in the WELOPEN keyword.

Random functions

The UDQ machinery has functionality to sample random numbers, there are one set of random number functions which are seeded deterministically by item 1 of UDQPARAMS and an alternative set which is seeded by the clock. The random number functions take a UDQ variable as argument, that is only to ensure that the shape of the result value is correct.

RANDN Random numbers from distribution $N(0, 1)$ seeded deterministically by item 1 in UDQPARAMS.

RANDU Random numbers from distribution $U(-1,1)$ seeded deterministically by item 1 in UDQPARAMS.

RRNDN Random numbers from distribution $N(0,1)$ seeded by the clock.

RRNDU Random numbers from distribution $U(-1,1)$ seeded by the clock.

Assuming the argument vector WU1 looks like

```
WU1 = {"OP1": -2, "OP2": [], "WI": 1, "GI": []},
```

the result of DEFINE WURAND RANDU(WU1) could be like

```
WURAND = {"OP1": 0.576, "OP2": [], "WI": -0.132, "GI": []}.
```

Work with defined status

The functions DEF, UNDEF and IDV can be used to inspect the defined/not defined status of the elements in a UDQ set.

DEF Return a set with value 1 for all defined elements.

UNDEF Return a set with value 1 for all undefined elements.

IDV Return a set with value 1 for all defined elements and value 0 for all undefined elements.

Assuming the input argument

```
WU1 = {"OP1": -2, "OP2": [], "WI": 1, "GI": []},
```

the UDQ assignments

```
UDQ
  DEFINE WUDEF   DEF(WU1) /
  DEFINE WUUNDEF UNDEF(WU1) /
  DEFINE WUIDV   IDV(WU1) /
/
```

will produce:

```
WUDEF   = {"OP1": 1,  "OP2": [], "WI": 1,  "GI": []},
WUUNDEF = {"OP1": [], "OP2": 1,  "WI": [], "GI": 1},
WUIDV   = {"OP1": 1,  "OP2": 0,  "WI": 1,  "GI": 0},
```

1.3.4 Union functions

There are a list of functions `Uxxx` which operate on the union of the values found in the two sets, i.e. a result is assigned if at at least one set has defined value for this well/group.

UADD Will add the items from the two sets.

UMUL Will multiply the items from the two sets.

UMAX The maximum value from the two sets.

UMIN The minumum value from the two sets.

In the case where only one of the sets has a defined value the operation will be performed *as if* the function `Uxxx` is the identity function.

As an example consider the two sets:

```
WU1     = {"OP1": 1, "OP2": [], "WI": 2,  "GI": []},
WU2     = {"OP1": 6, "OP2": 5,  "WI": [], "GI": []},
```

and the `UDQ` expression `DEFINE WUUADD WU1 UADD WU2`, then the resulting set `WUUADD` will be

```
WUUADD  = {"OP1": 7, "OP2": 5,  "WI": 2,  "GI": []}.
```

This in contrast to a normal + which only operates on the intersection of the two sets, only well OP1 would have a defined value in this case.

1.4 Used as a control: UDA

Probably one of the most important uses of the `UDQ` functionality is the ability to use a `UDQ` as control in e.g. the `WCONINJE` keyword. In the example below we calculate the produced liquid volume from a group of wells and use that as injection target for a water injector:

```
UDQ
  DEFINE FULPR  (WOPR P* + WWPR P*) * 1.25 /
/

...
...

WCONINJE
  WI  'WATER'  'OPEN'  'RATE'   'FULPR' /
/
```

In the following insane example we distribute the current oil production rate randomly among the producers and use that as target for the next timestep - do not try this at home:

```
UDQ
  -- Create a 0 / 1 mask with 0 for injectors and 1 for producers.
  DEFINE WUPROD WOPR > 0 /

  -- Create a vector of random numbers [0,1] for all producers
  DEFINE WURAND 0.5 * (RANDU(WUPROD) + 1) * WUPROD /

  -- Create a vector where all producers get a random fraction of
  -- the current total oil production rate
  DEFINE WUOPR FOPR * WURAND / SUM(WURAND) /
/

-- Need to have a well list or similar to select all producers for
-- the WCONPROD keyword.
WLIST
  'P' 'ADD' ..... /


WCONPROD
 '*P' 'OPEN' 'ORAT'  'WUOPR' /
/
```

One point about this example is that the `UDQ` variable `WUOPR` which is used as as control in the `WCONPROD` is a well set, the lookup machinery will automatically use the correct well index when assigning control value to a particular well.

From an implementation point of view the `UDA` functionality creates a significant complexity, because the actual rate to use in the simulation must be evaluated *just in time*.

1.5 UDQ units

The `UDQ` subcommand `UNIT` can be used to assign a string which is used as output unit when the `UDQ` variable is output to the summary file. This is *only* a string and does not induce any unit conversion. All `UDQ` evaluations are in terms of the corre deck units - irrespective of the `UNIT` subcommand. Consider the following

```
RUNSPEC

FIELD
```

```
...
...

SCHEDULE

UDQ
  DEFINE WUI WWIR - 100 /
/
```

1. The simulation runs in SI units; hence the water injection rate is calculated in m^3/s.

2. As part of the summary evaluation the `SummaryState` will contain the water injection rate converted to field units.

3. The `UDQ` variable `WUI` is evaluated as the water injection rate from `SummaryState` subtracted numerical value 100, *it is solely the users responsability that the numerical value 100 represents water injection rate in field units.*

Observe that the units are complicated, and non intuitive for `UDA` values. The `UDA` evaluation is based on the `SummaryState` class which is in deck units, the controls determined by `UDA` evaluation must therefor be converted to SI units just when it is passed to the simulator. The initial design ambition was to block the deck units at the IO boundary, keeping the internals fully in SI. The `UDA` concept is the exception which manages to inject deck units quite far into the code.

1.6 Implementation details

The significant part of the UDQ implementation is in classes located in `opm/input/eclipse/Schedule/UDQ`, in addition the restart output of `UDQ`/`UDA` values is in `opm/output/eclipse/AggregateUDQData.cpp`. Finally the `UDQ` parser makes use of the type `RawString` to treat literal "/" and "*" different from ordinary parsing where "/" signifies end of line and "*" is either a default or part of a multiplier expression.

`UDQConfig`

The class `UDQConfig` internalizes the parsing of the `UDQ` keywords from the input deck. The `UDQConfig` instance is time-versioned and managed by the `ScheduleState` class. The `UDQConfig` is immutable while the simulator is executing. The `UDQConfig` contains two main classes `UDQDefine` and `UDQAssign` in addition to book-keeping code to keep track of what type of

expression a UDQ keyword is at the time. A significant part of the bookkeeping is only required to be able to output restart files in Eclipse format.

UDQDefine and UDQASTNode

The UDQDefine manages a parsed UDQ expresssion along with a chunck of metadata. The parsed UDQ expression is managed in an instance of UDQASTNode. The UDQASTNode contains a parsed tree representation of the UDQ input expression. The parsing of UDQ expressions happens at input time, and for the rest of the simulation the UDQASTnode instances are immutable.

Many scalar UDQ types are defined in the file UDQEenums.hpp and in the namespace UDQ there are many small utility functions to work with these enums.

SummaryState

The SummaryState class is not part of the UDQ implementation, but it is a very important class for the UDQ functionality. At the end of every timestep the simulator will call the method evalSummary which will call into opm-common and evaluate all summary variables and store them in a SummaryState instance[7]. The SummaryState class manages a set of maps with well, group and field variables, when evaluating e.g. the WOPR the results will be stored in a two level map first indexed with keyword WOPR and then with well name. Afterwards the UDQ layer can fetch values with SummaryState::get_well_var(). The values in the SummaryState have been converted to output units, this is important for the UDA evaluation.

At the end of every timestep the UDQConfig::eval() method is called to evaluate all the UDQ expressions, the evaluated values will end up in the active SummaryState instance, i.e. for this UDQ

```
UDQ
  DEFINE FUOPR SUM(WOPR) /
  DEFINE WUGWR WGPR / WWPR /
/
```

we will get SummaryState entries for FUOPR and WUGWR for *all* wells in the model. These SummaryState values can then be output to the summary file, be used when evaluating ACTIONX keywords or to evaluate UDA control values.

In addition to the SummaryState variable there is an instance of type UDQState which is updated runtime, the UDQState instance holds on to the results of UDQ evaluations with more context than SummaryState and

[7] Observe that during initialization the UDQ expressions are inspected, and we make sure that all summary variables needed to evaulate UDQ expressions are evaluated in the Summary evaluation.

is used to output UDQ and UDA state to the restart files. While evaluating the simulator will create a UDQContext variable which will manage both the SummaryState, UDQState and also some UDQ parameters from the RUNSPEC section. The lifetime of the UDQContext instance is only one UDQ evaluation.

While evaluating the UDQ expressions the results will be instances of UDQSet which is a small container class which keeps track of well/group names and whether a value is defined or not. The UDQSet class overrides the arithmetic operators like UDQSet::operator+() so that expressions like 2 * WOPR 'OP*' can be easily evaluated in code.

Paralell awareness

The opm-common code where the UDQ functionality is implemented is totally unaware of parallel execution, so to be certain that this works for a parallel simulator care must be taken. In flow this is handled as:

1. The Schedule class as a whole is identical on all processes.
2. The state variables UDQState and SummaryState are distributed so they are equal on all processes before the UDQ evaluation. This communication is performed in the simulator.

The UDQ evaluation is invoked from opm-simulators/ebos/eclgenericwriter.cc function evalSummary().

Chapter 2

Programming in the deck: ACTIONX

The ACTIONX keyword is the most direct way to *program* in the deck. The ACTIONX functionality consist of the ACTIONX keyword itself, with some metadata and a condition and then a list of keywords which are injected into the in-memory representation of the SCHEDULE section at the point in time where the condition evaluates to true. The ACTIONX statement is evaluated at the end of every timestep, and if it evaluates to true the new keywords should take effect immediately. The ACTIONX conditions are less sophisticated than the expressions used in UDQ, this implies that a common pattern is to make involved calculations as UDQ expressions, and then use a simple test as ACTIONX condition.

The ACTIONX keyword is also documented in section 12.3.6 in the flow reference manual.

2.1 Structure of the ACTIONX keyword

The ACTIONX keyword itself consist of multiple records. The first record is metadata with name of the action, the number of times the action can be triggered and the minimum time elapsed before an action is eligible for a second run. The subsequent records are *conditions*, all the conditions are of the same form

```
lhs comparison rhs
```

and subsequent conditions are combined with AND or OR. The lhs is a field, well or group quantity, in addition you can use time variables DAY, MNTH and YEAR as left hand side[1]. As with the UDQ variables the well and group

[1] Eclipse supports a wider list of summary variables like region, block and aquifer quantities on both left and right hand side.

variables are *sets*, and the evaluation status is maintained individually for each well and group.

The comparison is one of the ordinary mathematical comparison operators `>,<,=,!=, <=` and `>=`. Numerical comparisons are done with the corresponding plain C++ operators, this is in contrast to the `UDQ` implementation where an epsilon defined in `UDQPARAMS` is used in floating point comparisons.

The `rhs` is a numerical scalar, or a field, well or group quantity. If your `rhs` is a well or group quantity the `lhs` and `rhs` must be of the same type. If you use the symbol `MNTH` as `lhs` you can compare with named months, i.e. the following will trigger on leap days

```
ACTIONX
  LEAP 1000 /
  MNTH=FEB AND /
  DAY=29 /
/
...
...
ENDACTIO
```

When there is a well/group quantity as `lhs` the evaluation status is maintained individually for each well/group. The complete condition evaluates to true if *any* of the wells/groups satisfy the condition. In the case of wells the wells matching the condition can subsequently be accessed with wellname '?' in the `ACTIONX` keywords, this is a quite common pattern to e.g. close the well with highest watercut.

If there are more conditions they must be joined with a trailing `AND` or `OR`, furthermore conditions can be grouped with paranthesis. The `ACTIONX` expressions can only contain the four arithmetic operators $+, -, *, /$ and not mathematical functions like log(), for more advanced expressions the natural approach is to first define a `UDQ` and then use the `UDQ` symbol in the `ACTIONX`, this is illustrated in section 1.4. When multiple conditions involving the same well set are evaluated, the list of matching wells available in '?' will contain all the wells from the final condition, i.e. for

```
 WWCT = {"OP1": 0.25, "OP2": 0.50, "OP3": 0.75}
```

and the action

```
ACTIONX
  WWCT /
  WWCT > 0.33 AND /
  WWCT < 0.66 /
/
...
ENDACTIO
```

the set of wells available for further use in '?' are *all* the wells matching the condition `WWCT < 0.66` i.e. OP1 and OP2 and *not* the wells matching the combined expression `0.33 < WWCT < 0.66`. In order to select wells in a range as attempted here you will have to create an indicator variable with `UDQ` first and then select based on that indicator - e.g. something like

```
UDQ
  DEFINE WUCTR (WWCT < 0.66) * (WWCT > 0.33) /
/

ACTIONX
  WUCTR /
  WUCTR = 1 /
/

...

ENDACTIO
```

The `ACTIONX` implementation is located in `opm/input/eclipse/Schedule/Action` and all the classes are in namespace `Action::`. As with the `UDQ` the input parser needs some special case to handle '/' and '*' as division operator and multiplier respectively, but that is the only code shared between the `UDQ` and the `ACTIONX` implementation[2].

The condition part of the `ACTIONX` keyword is internalized while the `SCHEDULE` section is parsed, the final product is maintained in a class `Action::ActionX` which has a `eval()` method waiting to be called. The keywords in the `ACTIONX` block are stored in the `Action::ActionX` keyword for future use. All of the `ACTIONX` keywords are stored in a container `Action::Actions` which will eventually manage the book keeping of which actions are eligible for evaluation.

2.2 The structure of the `Schedule` implementation

`flow` internalizes all keywords from the input deck and passes fully baked datastructures to the simulator, whereas our impression is that `Eclipse` works more like a reservoir model interepreter, executing one keywords at a time. Mostly the `flow` approach has worked out well, however for the `ACTIONX` functionality the difference in execution model is quite acute, and the nature of the `ACTIONX` keyword has had quite strong influence on the final Schedule implementation. Although not required for use of `ACTIONX` it

[2]It might be possible to share more code between the two, in particular both have an internal recursive descent parser, but both `UDQ` and `ACTIONX` have so much "personality" that at least initially separate implementations was the simplest.

is valuable to understand how the ACTIONX functionality has influenced the design of the Schedule class, that way you will hopefully better understand problems or bugs which might arise in the future.

At the very first pass the SCHEDULE section is split in *blocks*, with one block for each report step. The blocks are implemented with the class ScheduleBlock. Each block has a starting time and a list of keywords, the keywords are maintained in the input format DeckKeyword. Then the entire SCHEDULE section is internalized in the class ScheduleDeck which essentially contains a list of ScheduleBlock instances. Consider the SCHEDULE section

```
START
  1 'JAN' 2020 /

...
...

SCHEDULE

WELSPECS
'PROD''G1'10 10 8400 'OIL' /
'INJ''G1'1 1 8335 'GAS' /
/

COMPDAT
'PROD'10 10 3 3 'OPEN'1* 1* 0.5 /
'INJ'1 1 1 1 'OPEN'1* 1* 0.5 /
/

-- End of block 0

DATES
   1 'FEB' 2020 /
/

WCONPROD
'PROD' 'OPEN' 'ORAT' 20000 4* 1000 /
/

WCONINJE
'INJ''GAS''OPEN''RATE'100000 1* 9014 /
/

-- End of block 1
```

```
DATES
  1 'MAR' 2020 /
/

-- End of block 2
END
```

When this is internalized we get a `ScheduleDeck` instance with three `ScheduleBlock` values:

```
ScheduleDeck sched_deck = [
  ScheduleBlock {
    start = "2020-01-01",
    keywords = ["WELSPECS","COMPDAT"]
  },
  ScheduleBlock {
    start = "2020-02-01",
    keywords = ["WCONPROD","WCONINJE"]
  },
  ScheduleBlock {
    start = "2020-03-01",
    keywords = []
  }
]
```

The `Schedule` class has a `ScheduleDeck` member. The processed content of the `Schedule` class is managed in vector of `ScheduleState` instances, where one `ScheduleState` represents the complete dynamic input state at a particular report step. The processed `SCHEDULE` code is created with the method

```
Schedule::iterateScheduleSection().
```

When `Schedule::iterateScheduleSection(report_step)` is called it starts by clearing the vector of `ScheduleState` instances from `report_step` to the end of the simulation, and then recreates those by treating the `ScheduleBlock`. The advantage of this approach is that the `Schedule::iterateScheduleSection(report` method is idempotent - it can be called repeatedly, from an arbitrary point in the timeseries.

The implementation of the `ACTIONX` functionality is just to append the `ACTIONX` keywords in the `ScheduleBlock` instance corresponding to the current report step and then rerun the `Schedule::iterateScheduleSection()` from this report step.

2.3 Forward references of wells and groups

When a well or group is defined as an ACTIONX keyword and then unconditionally referenced in the deck we get a challenge at the first pass through the SCHEDULE section. In the example below a new well W1 is defined with the WELSPECS keyword when the action NEW_WELL evaluates to true. At 1.st of January 2025 the well W1 is opened with the WCONPROD keyword. The engineer making this model assumes that the NEW_WELL action will evaluate to true sometime before 1.st of January 2025, and thereby ensure that the well is fully defined when it is eventually opened with WCONPROD:

```
ACTIONX
   'NEW_WELL/
   WWCT OPX  > 0.75 /
/

WELSPECS
  'W1'  'OP'  1 1 3.33  'OIL' 7*/
/

ENDACTIO

TSTEP
  10*30 /

DATES
   1 'JAN' 2025 /
/

WCONPROD
 'W1'      'OPEN'      'ORAT'      0.000      0.000      0.000  5* /
/

TSTEP
  10*30 /
```

For flow this creates problems because the entire SCHEDULE section is parsed when the simulator starts, and at first pass the well W1 is unknown in the WCONPROD keyword. This is "solved" in the following way:

1. At first pass we inspect the keywords inside the ACTIONX block and if we discover WELSPECS we store the name of the well which will be defined at a later stage through ACTIONX.

2. When we parse further on as part of the first pass and said well is referenced e.g. in a WCONPROD keyword, we verify that the well will eventually appear via ACTIONX - we issue a warning and ignore the well in the WCONPROD keyword[3].

3. When the ACTIONX evaluates to true the well will be properly defined, and when reiterating over the Schedule keywords the WCONPROD keyword will now be properly internalized. If the ACTIONX never evaluates to true the WCONPROD keyword will never be applied, and the warning from point 2 will be the only trace of this well.

It should be mentioned that the functionality with forward referencing of well names is quite new[4], there might be well keywords in the SCHEDULE section where the implementation is not yet prepared for this. Furthermore the forward referencing is not at all implemented for groups. The relevant data structure is the member `Action::WGNames action_wgnames` in the `Schedule` class.

2.4 To enable a new keyword for ACTIONX

The keywords must be explicitly enabled to be available in an ACTIONX block, and enabling a new keyword requires recompiling `flow`. The keywords available as ACTIONX keywords are listed in the static method `ActionX::valid_keyword()` in `opm/input/eclipse/Schedule/Action/ActionX.cpp`. In principle it should just be to add the keyword to the `ActionX::valid_keyword()` method and rebuild `flow`, but experience has unfortunately shown that problems of various kinds have had a tendency to pop up when new keywords are tried out as ACTIONX keywords. Most commonly the problems have been in the interaction between the `Schedule` class in opm-common and the simulator - things have a tendency to go out of sync.

2.5 Running ACTIONX during simulation

The first part of the ACTIONX treatment is parsing the keyword and conditions and assemble a syntax tree which can be used to evaluate the conditions. This parsing takes place when the SCHEDULE section is parsed for the first time. This takes place fully within the realms of the opm-common codebase, and is quite mature.

When the simulation actually runs the ACTIONX behavior consists of three distinct parts, taking place in the simulator, in opm-common and again in the simulator. The simulator will manage an instance of `Action::State`

[3] If the well is not registered as "will appear through ACTIONX" there will be a runtime error with unknown well name when parsing WCONPROD.

[4] In January 2022

which will hold on to the time of last run and the latest results for the various actions.

2.5.1 Prepare and evaluate

As with UDQ the SummaryState instance is the most important variable to provide context to the ACTIONX evaluation, i.e. the SummaryState variable must be evaluated before ACTIONX. In addition the UDQ variables must be evaluated and available in the SummaryState instance before we invoke the ACTIONX functionality.

The evaluation of actions is called from the method applyActions() in eclproblem.hh. The method will evaluate which actions are eligible for running by inspecting the Action::Actions variable and call the ActionX::eval() method.

2.5.2 Recreate Schedule

When an ACTIONX has evaluated to true the simulator will call into the opm-common method Schedule::applyAction(report_step). That function will add the keywords from the ACTIONX keyword to the ScheduleBlock for the correct report step, and then reiterate through the SCHEDULE section to the end of the simulation.

This reiterate process will recreate all internal members in the Schedule class, i.e. if the simulator was holding on to a reference to an internal Schedule datastructure that will be invalidated.

While recreating the Schedule instance there is some book keeping as to which datastructures need to be recalculated in the simulator as a consequence of the ACTIONX. That information is maintained in the data structure Action::SimulatorUpdate which is returned back to the simulator.

2.5.3 Updating simulator data structures

The ACTIONX implementation is in the module opm-common, whereas when the simulation is proceeding it is the simulator code which is clearly in control. If an action has evaluated to true and new keywords are injected in the Schedule object the complete simulator state is updated. This is complex, and many of the bugs in ACTIONX functionality have been in the interaction between the simulator and opm-common, in particular when an action has evaluated to true.

An assumption permeating the simulator code is that changes to the well and group configuration only take place at report steps, and in between those the simulator "owns" the well and group data. Unfortunately this is no longer the case when ACTIONX is active, and depending on the keywords in the ACTIONX block we need to update the simulator data structures after ACTIONX has been evaluated to true. Some details of what is currently

updated is decsribed below. This update mechanism will probably need to be continously updated in the future.

The simulator code makes *copies* of many of the objects like wells and connections from the `Schedule` class, and also assembles many simulator specific data structures which to a large extent consist of extracts of information from the internals of the `Schedule` object. Some of the interaction between the simulator and the input layer could probably be simplified if the simulator would call the `Schedule` object when e.g. a well or connection is needed, instead of storing references or copies to the `Schedule` objects internally.

There are currently three categories of changes that can take place due to `ACTIONX`:

General changes in well status

When there is well related keyword in the `ACTIONX` block - e.g. `WELOPEN` to open or close a well or `WCONPROD` to adjust rates, the simulator is required update it's internal data structures with the updated input information. This will be communicated by flagging all the wells which need an update in the `Action::SimulatorUpdate` instance which is passed from the simulator.

Changes in geo properties

The grid keywords are in general not permitted in the `SCHEDULE` section, however there are a few geo multipliers like `MULTZ` and `MULTFLT` which are allowed in the `SCHEDULE` section, and thereby also in `ACTIONX`. If one of these keywords are encountered in the `ACTIONX` block we set the flag `Action::SimulatorUpdate::tran_update = true` to encourage the simulator to recalculate the transmissibilities.

WELPI

The `WELPI` keyword is quite complex from the outset, when it is included as an `ACTIONX` keyword it gets even more complicated. In order to support `WELPI` in `ACTIONX` the simulator needs to inspect the `ACTIONX` keywords before invoking them, and if `WELPI` is included the PI values must be assembled from the simulator and passed to the `Schedule::applyAction()`. This is implemented and works, but it is complex and the special treatment in order to support the combination `WELPI` + `ACTIONX` is quite considerable.

2.6 Problems in parallel

The operation environment for `ACTIONX` is quite similar to `UDQ` when it comes to parallel behavior - see section 1.6, in addition we need to be aware of

parallel challenges after the ACTIONX has completed. If the ACTIONX keyword changes the well structure there will be problems with with the parallel well distribution in the simulator. *As of Januarry 2022 this will fail undetected.*

2.7 ACTIONX restart output

As with the UDQ keyword some of the structure and complexity of the ACTIONX datastructures are there primarily to enable Eclipse compatible restart. Regarding restart of ACTIONX related data:

1. The restart output contains the result of parsing an ACTIONX condition in an intermediate representation which has been reverse engineered, this is complex and might not be 100% correct.

2. When restarting flow from an Eclipse formatted restart file the ACTIONX conditions are reparsed based on string data in the restart file and the intermediate representation mentioned in point 1 is not utilised.

Chapter 3

Programming in the deck: PYACTION

The PYACTION keyword is a flow specific keyword which allows for Python programming in the SCHEDULE section. The PYACTION keyword is inspired by the ACTIONX keyword, but instead of a .DATA formatted condition you are allowed to implement the condition with a general Python script. The ACTIONX keywords are very clearly separated in a condition part and an action part in the form of a list of keywords which are effectively injected in the SCHEDULE section when the condition evaluates to true. This is not so for PYACTION where there is only one Python script which can both evaluate conditions and apply changes. In principle the script can run arbitrary code, but due to the complexity of the SCHEDULE datamodel the "current best" way to actually change the course of the simulation is through the use of an additional dummy ACTIONX keyword.

In order to enable the PYACTION keyword flow must be compiled with the cmake switches -DOPM_ENABLE_EMBEDDED_PYTHON=ON and -DOPM_ENABLE_PYTHON=ON, the default is to build with these switches set to OFF. Before you enable PYACTION in your flow installation please read carefully through section 3.5 for security implications of PYACTION.

3.1 Python - wrapping and embedding

Python is present in the flow codebase in two different ways. For many of the classes in the flow codebase - in particular in opm-common, there are *Python wrappers* available. That means that you can invoke the C++ functionality in flow classes from Python - e.g. this Python script can be used to load a deck and print all the keywords:

```
import sys
from opm.io.parser import Parser
```

```
input_file = sys.argv[1]
parser = Parser()

deck = parser.parse_file(input_file)
for kw in deck:
    print(kw.name)
```

When used this way the Python interpreter is the main program running, and the `flow` classes like `Opm::Parser` are loaded to extend the Python interpreter. This can also be flipped around, the Python interpreter can be *embedded* in the `flow` executable. When Python is embedded, `flow` is the main program running, and with help of the embedded interpreter the `flow` program can be extended with Python plugins. The `PYACTION` keyword can be perceived as a Python plugin. To really interact with the state of the `flow` simulation the plugin needs to utilize the functionality which wraps the C++ functionality, so for `PYACTION` both wrapping and embedding is at play.

Exporting more functionality from C++ to Python in the form of new and updated wrappers is a quite simple and mechanical process. If you need a particular functionality which is already available in C++ also in Python it will probably be a quite limited effort for a developer who is already familiar with the code.

3.2 The PYACTION keyword

The `PYACTION` keyword is in the `SCHEDULE` section like `ACTIONX`. The first record is the name of the action and a string identifier for how many times the action should run, then there is a path to a Python module:

```
PYACTION
  PYTEST 'FIRST_TRUE' /
  'pytest.py' /
```

This keyword defines a `PYACTION` called `PYTEST` which will run at the end of every timestep until the first time a `true` value is returned. In addition to `FIRST_TRUE` you can choose `SINGLE` to run exactly once and `UNLIMITED` to continue running at the end of every timestep for the entire simulation. The second record is the path to a file with Python code which will run when this `PYACTION` is invoked. The path to the module will be interpreted relative to the location of the `.DATA` file.

The python module can be quite arbitrary, but it must contain a function `run` with the correct signature:

```
def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    print('Running python code in PYACTION')
    return True
```

The PYACTION machinery is not as robust as the simulator proper: while loading the PYACTION keyword flow will check that the Python module contains syntactically valid Python code, and that it contains a run() function, but it will *not* check the signature of the run() function. If the signature is wrong you will get a hard to diagnose runtime error.

When the Python module is loaded it does so in an environment where the path to the .DATA file has been appended to the Python load path by manipulating the internal sys.path variable.

3.2.1 The different arguments

The run() function will be called with exactly five arguments which your implementation can use. These arguments point to datastructures in the simulator, and is the way to interact with the state of the simulation. The five arguments are:

ecl_state: An instance of the Opm::EclipseState class - this is a representation of *all static properties* in the model, ranging from porosity to relperm tables. The content of the ecl_state is immutable - you are not allowed to change the static properties at runtime[1].

schedule: An instance of the Opm::Schedule class - this is a representation of all the content from the SCHEDULE section, notably all well and group information and the timestepping. Being able to change the SCHEDULE information runtime is certainly one of the main motivations for this functionality, however due to the complexity of the Opm::Schedule class (section 2.2) the recommended way to actually mutate the Opm::Schedule is through the use of a dummy ACTIONX keyword (section 3.3).

report_step: This is an integer for the report step we are currently working on. Observe that the PYACTION is called for every simulator timestep, i.e. it will typically be called multiple times with the same value for the report_step argument.

summary_state: An instance of the Opm::SummaryState class, this is where the current summary results of the simulator are stored. The SummaryState class has methods to get hold of well, group and general variables

[1] This could certainly be interesting, but this is beyond the scope of the PYACTION keyword.

```
# Print all well names
for well in summary_state.wells:
    print(well)

# Assign all group names to the variable group_names
group_names = summary_state.groups

# Sum the oil rate from all wells.
sum_wopr = 0
for well in summary_state.wells:
    sum_wopr += summary_state.well_var(well, 'WOPR')

# Directly fetch the FOPR from the summary_state
fopr = summary_state['FOPR']
```

The `summary_state` variable can also be updated with the `update()`, `update_well_var()` and `update_group_var()` methods.

`actionx_callback`**:** The `actionx_callback` is a specialized function which is used to update the `Schedule` object by applying the keywords from a normal `ACTIONX` keyword. This is described in detail in section 3.3.

3.2.2 Holding state

The `PYACTION` keywords will often be invoked multiple times, a Python dictionary `state` has been injected in the module - that dictionary can be used to maintain state between invocations. Let us assume we want to detect when the field oil production starts curving down - i.e. when $\partial_t^2 \text{FOPR} < 0$, in order to calculate that we need to keep track of the timesteps and the FOPR as function of time - this is one possible implementation:

```
def diff(pair1, pair2):
    return (pair1[0] - pair2[0], pair1[1] - pair2[1])

def fopr_diff2(summary_state):
    fopr = summary_state.get('FOPR')
    sim_time = summary_state.get('TIME')
    if not 'fopr' in state:
        state['fopr'] = []
    fopr_series = state['fopr']
    fopr_series.append( (sim_time, fopr) )

    if len(fopr_series) < 2:
       return None
```

```
    pair0 = fopr_series[-1]
    pair1 = fopr_series[-2]
    pair2 = fopr_series[-3]

    dt1, df1 = diff(pair0, pair1)
    dt2, df2 = diff(pair1, pair2)

    return 2*(df1/dt1 - df2/dt2)/(dt1 + dt2)

def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    fopr_d2 = fopr_diff2(summary_state)
    if not fopr_d2 is None:
       if fopr_d2 < 0:
           print('Hmmm - this is going the wrong way')
       else:
           print('All good - sky is the limit!')
```

3.3 Changing the Schedule object - using a "normal" ACTIONX

Before reading this section you should make sure to understand the Schedule design described in section 2.2. The initial plan when implementing the PYACTION keyword was to be able to make function calls like

```
schedule.close_well(w1, report_step)
schedule.set_orat(w2, 1000, report_step)
```

to close a well and set the oil rate of another well. Unfortunately it proved very complex to get good semantics for combining such runtime changes with the keyword based model for SCHEDULE section. The current recommendation is to apply changes to the SCHEDULE section using callbacks to ACTIONX keywords from Python code, this is illustrated in the example below[2].

The recommended way to achieve this is to create a normal ACTIONX keyword which is set up to run zero times, and then explicitly invoke that from the Python run() function. In the example below we create an ACTIONX CLOSEWELLS which will close all matching wells (the wellname '?')

```
ACTIONX
  CLOSEWELLS 0 /
```

[2]From a programmers point of view the solution seems very unsatisfactory, but it works and it plays nicely with the ACTIONX behavior. If/when the underlying Schedule implementation changes there is nothing per se in the PYACTION design which inhibits use of a better Schedule api in the future.

```
  /
/

WELOPEN
  '?' 'CLOSE' /
/

ENDACTIO
```

The CLOSEWELLS action is set up to run zero times, so the normal ACTIONX machinery will never run this action[3]. Then in the Python run function we go through all the wells and call the CLOSEWELL action to close those with OPR < 1000:

```
def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    close_wells = []
    for well in summary_state.wells:
        if summary_state.well_var(well, 'WOPR') < 1000:
           close_wells.append(well)

    if close_wells:
        actionx_callback('CLOSEWELLS', close_wells)
```

The implementation of this is quite complex with thread of execution going from C++ to Python, then invoking a callback to C++ which will call Schedule::iterateScheduleSection(), going back to Python to complete the run() method before the function pointers pops back to C++ and continues the simulator execution[4].

3.4 Implementing UDQ like behavior

The UDQ keyword has three different purposes - all based on defining complex quantities from the current state of the simulation:

1. Define a complex quantity to be used in a ACTIONX condition.
2. Define a complex quantity for reporting in the summary file.
3. Define a quantity which can used as a control in UDA.

All of these can be achieved by using the PYACTION keyword, although for the two latter alternatives you must specify the UDQ keyword in the deck first, but you can let the PYACTION implementation override the value:

[3]The CLOSEWELL action has an *empty condition*, the ACTIONX keywords with empty condition will always evaluate as false.

[4]This is documented in some detail as code comments of Schedule::applyPyAction() in the Schedule.cpp file.

```
-- Observe that this UDQ will be assigned from a PYACTION keyword,
-- the value used in the ASSIGN statement below is pure dummy.
UDQ
  ASSIGN WUGOOD 1 /
  ASSIGN FUGOOD 1 /
/
```

3.4.1 Using PYACTION instead of UDQ + ACTIONX

Towards the end of section 2.1 it is demonstrated how UDQ and ACTIONX can be combined to implement an action in case a complicated condition applies. As described in section 3.3 the best way to actually invoke changes on the SCHEDULE section is through the use of a dummy ACTIONX keyword, but PYACTION is very well suited to evaluate complex conditions. In the example below we close all wells which have consistently produced less than 1000 m^3/day for more than 60 days:

```
wopr_limit = 1000
time_limit = 60 * 3600 * 24

def init_state(summary_state):
    if 'closed_wells' in state:
        return

    state['closed_wells'] = set()
    bad_wells = {}
    for well in summary_state.wells:
        bad_wells[well] = None
    state['bad_wells'] = bad_wells


def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    shut_wells = []
    init_state(summary_state)
    for well in summary_state.wells:
        if well in state['closed_wells']:
           continue

        if summary_state.well_var(well, 'WOPR') < wopr_limit:
            elapsed = summary_state.elapsed()
            if state['bad_wells'][well] is None:
                state['bad_wells'][well] = elapsed
            else:
                bad_time = elapsed - state['bad_wells'][well]
```

```
                if bad_time > time_limit:
                    shut_wells.append(well)
                    state['closed_wells'].add( well )
        else:
            state['bad_wells'][well] = None

    if shut_wells:
       actionx_callback(shut_wells)
```

3.4.2 Using PYACTION to report to the summary file

The important point when using PYACTION to report complex results to the summary file is just that the summary_state argument to the run() function is *writable* with updata_xxx calls. Assuming dummy UDQ variables WUGOOD and FUGOOD have been defined as per the example above, we can use PYACTION to set variable FUGOOD to one for all wells with rate above a limit, and the FUGOOD variable can be the count of such wells:

```
def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    good_count = 0
    opr_limit = 1000
    for wname in schedule.well_names():
        if summary_state.well_var(wname, 'FOPR') > opr_limit:
            good_count += 1
            summary_state.update_well_var(wname, 'WUGOOD', 1)
        else:
            summary_state.update_well_var(wname, 'WUGOOD', 0)

    summary_state.update_var('FUGOOD', good_count)
```

3.4.3 Using PYACTION to set a UDA control

Using PYACTION to set UDA controls is quite simple. Again the UDQ keyword must have been defined with a dummy value in the SCHEDULE section, and the UDA keyword used in e.g. a WCONDPROD keyword. Then the run() function can just be used to assign to the UDQ variable. In the example below we use a UDA to control the oil production rate, and the value is set to the average value of the producing wells:

```
-- Define dummy UDQ WUOPR to be used as control in the WCONPROD
-- keyword. The actual value for this UDQ is assigned in a PYACTION
-- keyword
UDQ
   ASSIGN WUOPR 0 /
```

```
/
...
...
-- Need to define a well list with all the production wells.
-- This is to ensure that the WCONPROD keyword is only applied
-- to producers.
WLIST
  'PROD'  P1 P2 P3 .../

WCONPROD
   '*PROD'  'OPEN'  'ORAT'  'WUOPR' /
/
```

This can then be combined with the python code:

```
def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    num_prod_wells = 0
    for wname in schedule.well_names():
        if summary_state.well_var(wname, 'WOPR') > 0:
            num_prod_wells += 1

    fopr = summary_state['FOPR']
    new_rate = fopr / num_prod_wells
    for wname in schedule.well_names():
        summary_state.update_well_var(wname, 'WUOPR', new_rate)
```

3.5 Security implications of PYACTION

The PYACTION keyword allows for execution of arbitrary user supplied Python code, with the priviliges of the user actually running flow. If you have a setup where flow runs with a different user account than the person submitting the simulation you should be *very careful* about enabling the embedded Python functionality and the PYACTION keyword. As a scary example this script will wipe your disks:

```
import shutil

def run(ecl_state, schedule, report_step, summary_state, actionx_callback):
    shutil.rmtree('/')
```

If the user running flow has different security credentials than the user submits the job, this has significant security implications.