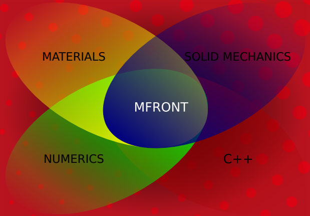
Overview of `TFEL-4.1` and `MGIS-2.1`



DE LA RECHERCHE À L'INDUSTRIE

MFront User Meeting

14/11/2021

T. Helfer[1], M. Wangermez[1] and (so) many others
[1] CEA, DES, IRESNE, DEC, SESC, LSC, Cadarache, FranceCEA

Outline

- **Documentation, tutorials and the MFront book**
- **New features of MFront in Version 4.1**
- **New features of MTest in Version 4.1**
- **MFront on the `GPUs`, an on-going work**
- **The MGIS project**
- **The MFrontGallery project**
- **Conclusions and perspectives**

- This talk is based on the following release-notes :
  - `https://thelfer.github.io/tfel/web/release-notes-4.1.html`
  - `https://thelfer.github.io/mgis/web/release-notes-2.1.html`

Documentation, tutorials and the MFront book

List of new tutorials

- Implicit integration of the constitutive equations of a polycrystal obtained by the Berveiller-Zaoui homogeneization scheme :
  - https://thelfer.github.io/tfel/web/ImplicitBerveillerZaouiPolyCrystals.html
- Implicit integration of finite strain behaviours based on a multiplicative decomposition of the deformation gradient :
  - https://thelfer.github.io/tfel/web/FeFpImplicitPlasticity.html
- Implementation of the Korthaus' behaviour for crushed salt
  - https://thelfer.github.io/MFrontGallery/web/CrushedSaltKorthausBehaviour.html
- Introducing small strain legacy `Abaqus/UMAT` implementations in `MFrontGallery`
  - https://thelfer.github.io/MFrontGallery/web/SmallStrainUmatWrapper.html

- The following chapters have been written :
  1. Material properties in MFront :
     - https://thelfer.github.io/tfel/web/material-properties.html
  2. Point-wise models in MFront

- The following chapters have been written :
  1. Material properties in MFront :
     - https://thelfer.github.io/tfel/web/material-properties.html
  2. Point-wise models in MFront
- Those chapters are meant for new users and introduces the main concepts behind MFront and MTest.

- The following chapters have been written :
  1. Material properties in MFront :
     - https://thelfer.github.io/tfel/web/material-properties.html
  2. Point-wise models in MFront
- Those chapters are meant for new users and introduces the main concepts behind MFront and MTest.
- The next chapter is about the role of mechanical behaviours and the fourth one is about isotropic (visco-)plastic behaviours.

- The following chapters have been written :
  1. Material properties in MFront :
     - https://thelfer.github.io/tfel/web/material-properties.html
  2. Point-wise models in MFront
- Those chapters are meant for new users and introduces the main concepts behind MFront and MTest.
- The next chapter is about the role of mechanical behaviours and the fourth one is about isotropic (visco-)plastic behaviours.
- Early feed-backs would be greatly appreciated.

New features of MFront Version 4.1

Selecting an unit system

```
@UnitSystem SI;
```

- The `@UnitSystem` keyword (common to all DSLs) allows to specify an unit system.
  - Currently, `SI` is is the only unit system supported.
  - `mfront --help-keyword=Default:@UnitSystem`

Selecting an unit system

```
@UnitSystem SI;
```

- The `@UnitSystem` keyword (common to all DSLs) allows to specify an unit system.
  - Currently, `SI` is is the only unit system supported.
  - `mfront --help-keyword=Default:@UnitSystem`
- Defining an unit system allows to retrieve *automatically* the physical bounds of variable from the `TFEL`' glossary.

Selecting an unit system

```
@UnitSystem SI;
```

- The `@UnitSystem` keyword (common to all DSLs) allows to specify an unit system.
  - Currently, `SI` is is the only unit system supported.
  - `mfront --help-keyword=Default:@UnitSystem`
- Defining an unit system allows to retrieve *automatically* the physical bounds of variable from the `TFEL`' glossary.
- The unit system can be retrieved by the `getUnitSystem` method of the `ExternalLibraryManager` class (in `C++` and in `python`).

Selecting an unit system

```
@UnitSystem SI;
```

- The `@UnitSystem` keyword (common to all DSLs) allows to specify an unit system.
  - Currently, `SI` is is the only unit system supported.
  - `mfront --help-keyword=Default:@UnitSystem`
- Defining an unit system allows to retrieve *automatically* the physical bounds of variable from the `TFEL`' glossary.
- The unit system can be retrieved by the `getUnitSystem` method of the `ExternalLibraryManager` class (in `C++` and in `python`).
- The `ExternalMaterialKnowledgeDescription` class has a new data member `unit_system` (in `C++` and in `python`).

Selecting an unit system

```
@UnitSystem SI;
```

- The `@UnitSystem` keyword (common to all DSLs) allows to specify an unit system.
  - Currently, `SI` is is the only unit system supported.
  - `mfront --help-keyword=Default:@UnitSystem`
- Defining an unit system allows to retrieve *automatically* the physical bounds of variable from the `TFEL`' glossary.
- The unit system can be retrieved by the `getUnitSystem` method of the `ExternalLibraryManager` class (in `C++` and in `python`).
- The `ExternalMaterialKnowledgeDescription` class has a new data member `unit_system` (in `C++` and in `python`).
- `mfront-query` has a new `--unit-sytem` query.

```
@Parameter temperature Ta = 600;
@Parameter strain p0 = 1e-8;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "UserDefinedViscoplasticity" {
    criterion : "Mises",
    E : 8.2,
    A : "8e-67 * exp(- T / Ta)",
    m : 0.32,
    vp : "A * (f ** E) / ((p + p0) ** m)",
    dvp_df : "E * vp / (max(f, seps))"
    // dvp_dp is evaluated by automatic differentiation (which is not recommended)
  }
};
```

- The `UserDefinedViscoplasticity` inelastic flow allows the user to specify the viscoplastic strain rate `vp` as a function of `f` and `p` where :
  - `f` is the positive part of the $\phi\left(\underline{\sigma}-\sum_i \underline{\mathbf{X}}_i\right)-\sum_i R_i\left(p\right)$.
  - `p` is the equivalent viscoplastic strain (optional).

```
@Parameter temperature Ta = 600;
@Parameter strain p0 = 1e−8;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "UserDefinedViscoplasticity" {
    criterion : "Mises",
    E : 8.2,
    A : "8e−67 ∗ exp(− T / Ta)",
    m : 0.32,
    vp : "A ∗ (f ∗∗ E) / ((p + p0) ∗∗ m)",
    dvp_df : "E ∗ vp / (max(f, seps))"
    // dvp_dp is evaluated by automatic differentiation (which is not recommended)
  }
};
```

- The `UserDefinedViscoplasticity` inelastic flow allows the user to specify the viscoplastic strain rate `vp` as a function of `f` and `p` where :
  - `f` is the positive part of the $\phi\left(\underline{\sigma}-\sum_i \underline{\mathbf{X}}_i\right)-\sum_i R_i\left(p\right)$.
  - `p` is the equivalent viscoplastic strain (optional).
- If required, the derivatives of `vp` with respect to `f` and `p` can be provided through the options `dvp_df` and `dvp_dp`. The derivatives `dvp_df` and `dvp_dp` can depend on two additional variables, `vp` and `seps` (a stress threshold under which the stress is considered null).

```
@Parameter stress R0 = 200e6;
@Parameter stress Hy = 40e6;
@Parameter real b = 100;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "UserDefined" {
      R : "R0 + Hy * (1 - exp(-b * p))",    // Yield radius
      dR_dp : "b * (R0 + Hy - R)"
    }
  }
};
```

- The `UserDefined` isotropic hardening rule allows the user to specify the radius R of the yield surface as a function of the equivalent plastic strain p.

Improvements to the `StandardElastoVisco-Plasticity` brick - II

```
@Parameter stress R0 = 200e6;
@Parameter stress Hy = 40e6;
@Parameter real b = 100;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "UserDefined" {
      R : "R0 + Hy * (1 - exp(-b * p))",    // Yield radius
      dR_dp : "b * (R0 + Hy - R)"
    }
  }
};
```

- The `UserDefined` isotropic hardening rule allows the user to specify the radius R of the yield surface as a function of the equivalent plastic strain p.
- If required, the derivative of R with respect to f and p can be provided through the option `dR_dp`. Note that the derivative $\dfrac{\partial R}{\partial p}$ can depend on the variable `R`.

Improvements to the `StandardElastoVisco-Plasticity` brick - III

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "Data" {
      values : {0 : 150e6, 1e−3 : 200e6, 2e−3 : 400e6},
      interpolation : "linear"
    }
  }
};
```

- The `Data` isotropic hardening rule allows the user to define an isotropic hardening rule using a curve defined by a set of pairs of equivalent strain and equivalent stress.

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "Data" {
      values : {0 : 150e6, 1e−3 : 200e6, 2e−3 : 400e6},
      interpolation : "linear"
    }
  }
};
```

- The `Data` isotropic hardening rule allows the user to define an isotropic hardening rule using a curve defined by a set of pairs of equivalent strain and equivalent stress.
- This isotropic hardening rule can be parametrised using three options :
  - `values` : which must a dictionnary giving the value of the yield surface radius as a function of the equivalent plastic strain.
  - `interpolation` : which allows to select the interpolation type. Possible values are `linear` (default choice) and `cubic_spline`.
  - `extrapolation` : which allows to select the extrapolation type. Possible values are `bound_to_last_value` (or `constant`) and `extrapolation` (default choice).

```
kinematic_hardening : "DRS" {
  C : 150.e9,  // kinematic moduli
  D : 1e2,     // back—strain callback coefficient
  f : 10,
  m : 5,
  Ec : {0.33, 0.33, 0.33, 1, 1, 1},
  Rs : {0.33, 0.63, 0.33, 1, 1, 1},
  Rd : {0.33, 0.33, 0.33, 1, 1, 1}  //
},
```

- The Delobelle-Robinet-Schaffler (DRS) kinematic hardening rule has been introduced to describe the viscoplasticity of Zircaloy alloys.

```
kinematic_hardening : "DRS" {
  C : 150.e9,  // kinematic moduli
  D : 1e2,     // back—strain callback coefficient
  f : 10,
  m : 5,
  Ec : {0.33, 0.33, 0.33, 1, 1, 1},
  Rs : {0.33, 0.63, 0.33, 1, 1, 1},
  Rd : {0.33, 0.33, 0.33, 1, 1, 1}  //
},
```

- The Delobelle-Robinet-Schaffler (DRS) kinematic hardening rule has been introduced to describe the viscoplasticity of Zircaloy alloys.
- It describes both dynamic and static recovery by the following law :

$$\frac{d\underline{\mathbf{a}}}{dt} = \dot{p}\,\underline{\underline{\mathbf{E}}}_{c}\,:\,\underline{\mathbf{n}} - D\,\dot{p}\,\underline{\underline{\mathbf{R}}}_{d}\,:\,\underline{\mathbf{a}} - f\left(\frac{a_{\mathrm{eq}}}{a_{0}}\right)^{m}\frac{\partial a_{\mathrm{eq}}}{\partial \underline{\mathbf{a}}}$$

with $a_{\mathrm{eq}} = \sqrt{\underline{\mathbf{a}}\,:\,\underline{\underline{\mathbf{R}}}_{s}\,:\,\underline{\mathbf{a}}}$ and $\dfrac{\partial a_{\mathrm{eq}}}{\partial \underline{\mathbf{a}}} = \dfrac{\underline{\underline{\mathbf{R}}}_{s}\,:\,\underline{\mathbf{a}}}{a_{\mathrm{eq}}}$

```
kinematic_hardening : "DRS" {
  C : 150.e9,  // kinematic moduli
  D : 1e2,     // back—strain callback coefficient
  f : 10,
  m : 5,
  Ec : {0.33, 0.33, 0.33, 1, 1, 1},
  Rs : {0.33, 0.63, 0.33, 1, 1, 1},
  Rd : {0.33, 0.33, 0.33, 1, 1, 1}  //
},
```

- The Delobelle-Robinet-Schaffler (DRS) kinematic hardening rule has been introduced to describe the viscoplasticity of Zircaloy alloys.
- It describes both dynamic and static recovery by the following law :

$$\frac{d\underline{\mathbf{a}}}{dt} = \dot{p}\,\underline{\underline{\mathbf{E}}}_{c}\,:\,\underline{\mathbf{n}} - D\,\dot{p}\,\underline{\underline{\mathbf{R}}}_{d}\,:\,\underline{\mathbf{a}} - f\left(\frac{a_{\mathrm{eq}}}{a_0}\right)^{m}\frac{\partial a_{\mathrm{eq}}}{\partial \underline{\mathbf{a}}}$$

with $a_{\mathrm{eq}} = \sqrt{\underline{\mathbf{a}}\,:\,\underline{\underline{\mathbf{R}}}_{s}\,:\,\underline{\mathbf{a}}}$ and $\dfrac{\partial a_{\mathrm{eq}}}{\partial \underline{\mathbf{a}}} = \dfrac{\underline{\underline{\mathbf{R}}}_{s}\,:\,\underline{\mathbf{a}}}{a_{\mathrm{eq}}}$

- The three fourth order tensors $\underline{\underline{\mathbf{E}}}_{c}$, $\underline{\underline{\mathbf{R}}}_{d}$ and $\underline{\underline{\mathbf{R}}}_{s}$ are assumed to have the same structure as the Hill tensor.

```
kinematic_hardening : "DRS" {
  C : 150.e9,  // kinematic moduli
  D : 1e2,     // back—strain callback coefficient
  f : 10,
  m : 5,
  Ec : {0.33, 0.33, 0.33, 1, 1, 1},
  Rs : {0.33, 0.63, 0.33, 1, 1, 1},
  Rd : {0.33, 0.33, 0.33, 1, 1, 1}  //
},
```

- The Delobelle-Robinet-Schaffler (DRS) kinematic hardening rule has been introduced to describe the viscoplasticity of Zircaloy alloys.
- It describes both dynamic and static recovery by the following law :

$$\frac{\mathrm{d}\underline{\mathbf{a}}}{\mathrm{d}t} = \dot{p}\,\underline{\underline{\mathbf{E}}}_{c} : \underline{\mathbf{n}} - D\,\dot{p}\,\underline{\underline{\mathbf{R}}}_{d} : \underline{\mathbf{a}} - f\left(\frac{a_{\mathrm{eq}}}{a_{0}}\right)^{m}\frac{\partial a_{\mathrm{eq}}}{\partial \underline{\mathbf{a}}}$$

with $a_{\mathrm{eq}} = \sqrt{\underline{\mathbf{a}} : \underline{\underline{\mathbf{R}}}_{s} : \underline{\mathbf{a}}}$ and $\dfrac{\partial a_{\mathrm{eq}}}{\partial \underline{\mathbf{a}}} = \dfrac{\underline{\underline{\mathbf{R}}}_{s} : \underline{\mathbf{a}}}{a_{\mathrm{eq}}}$

- The three fourth order tensors $\underline{\underline{\mathbf{E}}}_{c}$, $\underline{\underline{\mathbf{R}}}_{d}$ and $\underline{\underline{\mathbf{R}}}_{s}$ are assumed to have the same structure as the Hill tensor.
- The `f` and `a0` parameters are optional and defaults to 1.

Initialize functions for behaviours (experimental)

```
@InitializeFunction ElasticStrainFromInitialStress{
  const auto K = 2 / (3 * (1 - 2 * nu));
  const auto pr = trace(sig) / 3;
  const auto s = deviator(sig);
  eel = eval((pr / K) * Stensor::Id() + s / mu);
}
```

- The `@InitializeFunction` keyword introduces a code block that can be used to initialize internal state variables at the very beginning of the computation.
  - A behaviour can define many initialize functions that can be called individually by the calling solver.

Initialize functions for behaviours (experimental)

```
@InitializeFunction ElasticStrainFromInitialStress{
  const auto K = 2 / (3 * (1 - 2 * nu));
  const auto pr = trace(sig) / 3;
  const auto s = deviator(sig);
  eel = eval((pr / K) * Stensor::Id() + s / mu);
}
```

- The `@InitializeFunction` keyword introduces a code block that can be used to initialize internal state variables at the very beginning of the computation.
  - A behaviour can define many initialize functions that can be called individually by the calling solver.
- Initalize functions take sthe state at the beginning of the time step (all increments are null) and update the value of the state variables.

Initialize functions for behaviours (experimental)

```
@InitializeFunction ElasticStrainFromInitialStress{
  const auto K = 2 / (3 * (1 — 2 * nu));
  const auto pr = trace(sig) / 3;
  const auto s = deviator(sig);
  eel = eval((pr / K) * Stensor::Id() + s / mu);
}
```

- The `@InitializeFunction` keyword introduces a code block that can be used to initialize internal state variables at the very beginning of the computation.
  - A behaviour can define many initialize functions that can be called individually by the calling solver.
- Initalize functions take sthe state at the beginning of the time step (all increments are null) and update the value of the state variables.
- Initalize functions may also have dedicated intputs (called initialize function variables) introduced by the `@InitializeFunctionVariable`.
  - An initialize function variable can be common to several initialize functions.

Initialize functions for behaviours (experimental)

```
@InitializeFunction ElasticStrainFromInitialStress{
  const auto K = 2 / (3 ∗ (1 − 2 ∗ nu));
  const auto pr = trace(sig) / 3;
  const auto s = deviator(sig);
  eel = eval((pr / K) ∗ Stensor::Id() + s / mu);
}
```

- The `@InitializeFunction` keyword introduces a code block that can be used to initialize internal state variables at the very beginning of the computation.
  - A behaviour can define many initialize functions that can be called individually by the calling solver.
- Initalize functions take sthe state at the beginning of the time step (all increments are null) and update the value of the state variables.
- Initalize functions may also have dedicated intputs (called initialize function variables) introduced by the `@InitializeFunctionVariable`.
  - An initialize function variable can be common to several initialize functions.
- Initialize functions are only supported by the `generic` interface.

Post-processings of behaviours (experimental)

```
//! principal strains
@PostProcessingVariable tvector<3u,strain> ep;
ep.setEntryName("PrincipalStrain");
//! compute the principal strain
@PostProcessing PrincipalStrain {
  ep = eto.computeEigenValues();
}
```

- The `@PostProcessing` keyword introduces a code block that can be used to perform computations in a separate step of the behaviour integration.

Post-processings of behaviours (experimental)

```
//! principal strains
@PostProcessingVariable tvector<3u,strain> ep;
ep.setEntryName("PrincipalStrain");
//! compute the principal strain
@PostProcessing PrincipalStrain {
  ep = eto.computeEigenValues();
}
```

- The `@PostProcessing` keyword introduces a code block that can be used to perform computations in a separate step of the behaviour integration.
- The outputs of post-processings are stored in so-called *post-processing variables* declared by the `@PostProcessingVariable`.

Options to domain specific languages

```
@DSL Default{parameters_as_static_variables : true};
```

- Options to domain specific languages modify their default behaviour :
  - The goal is inhibit some features (for instance, the modification of the parameters from a text file).

Options to domain specific languages

```
@DSL Default{parameters_as_static_variables : true};
```

- Options to domain specific languages modify their default behaviour :
  - The goal is inhibit some features (for instance, the modification of the parameters from a text file).
  - Customize the compilation for performances (for instance, treating parameters as static variables).

Options to domain specific languages

```
@DSL Default{parameters_as_static_variables : true};
```

- Options to domain specific languages modify their default behaviour :
  - The goal is inhibit some features (for instance, the modification of the parameters from a text file).
  - Customize the compilation for performances (for instance, treating parameters as static variables).
  - Paves the way toward computations on GPUs and on the fly compilation of behaviours.

Options to domain specific languages

```
@DSL Default{parameters_as_static_variables : true};
```

- Options to domain specific languages modify their default behaviour :
  - The goal is inhibit some features (for instance, the modification of the parameters from a text file).
  - Customize the compilation for performances (for instance, treating parameters as static variables).
  - Paves the way toward computations on GPUs and on the fly compilation of behaviours.
- The list of available options for a DSL can be retrieved as follows :

```
$ mfront --list-dsl-options=RungeKutta
```

Options to domain specific languages - II

```
$ mfront --obuild --interface=generic                              \
         --behaviour-dsl-option=parameters_as_static_variables:true \
         --behaviour-dsl-option='overriding_parameters:{T:293.15}'  \
         Plasticity.mfront
```

- DSL options can be specified in a block after the definition of the DSL or on the command line (see `MFrontGallery` project) :
  - `--dsl-option`
  - `--material-property-dsl-option`
  - `--behaviour-dsl-option`
  - `--model-dsl-option`

Options to domain specific languages - II

```
$ mfront --obuild --interface=generic                                \
       --behaviour-dsl-option=parameters_as_static_variables:true \
       --behaviour-dsl-option='overriding_parameters:{T:293.15}' \
       Plasticity.mfront
```

- DSL options can be specified in a block after the definition of the DSL or on the command line (see `MFrontGallery` project) :
  - `--dsl-option`
  - `--material-property-dsl-option`
  - `--behaviour-dsl-option`
  - `--model-dsl-option`
- DSL Options can also gathered in an JSON-like file :

```
$ mfront --obuild --interface=generic          \
       --behaviour-dsl-options-file=options.json \
       Plasticity.mfront
```

where the file `options.json` file may look like :

```
overriding_parameters : {T : 293.15, dT : 0},
parameters_as_static_variables : true
```

Options to domain specific languages - III

- The following options are available :
  - `default_out_of_bounds_policy`
  - `out_of_bounds_policy_runtime_modification`
  - `parameters_as_static_variables`
  - `parameters_initialization_from_file`
  - `build_identifier`
  - `overriding_parameters`
  - `automatic_declaration_of_the_temperature_as_first_external_state_variable` (behaviours only)

Extension of the `Cast3M` interface to point-wise models

- The `Cast3M` interface has been extended to support point-wise models :
  - As supported by the `Model` DSL.
  - As generic behaviours without gradients, as supported by the `DefaultModel` DSL, the `RungeKuttaModel` DSL and the `ImplicitModel` DSL.

Extension of the `Cast3M` interface to point-wise models

- The `Cast3M` interface has been extended to support point-wise models :
  - As supported by the `Model` DSL.
  - As generic behaviours without gradients, as supported by the `DefaultModel` DSL, the `RungeKuttaModel` DSL and the `ImplicitModel` DSL.
- Proper support for models will land in `Cast3M` Version 2023. In the meantime, the generated models can be tested with `MTest`.

Miscellaneous improvements - I

```
constexpr auto R = PhysicalConstants::R;
```

- Improvements to the `MaterialProperty` DSL :
  - The physical constants defined in the `TFEL/PhysicalConstants` library are available through the PhysicalConstants type alias.

Miscellaneous improvements - I

```cpp
constexpr auto R = PhysicalConstants::R;
```

- Improvements to the `MaterialProperty` DSL :
  - The physical constants defined in the `TFEL/PhysicalConstants` library are available through the PhysicalConstants type alias.
- Improvements to the `Model` DSL
  - The physical constants defined in the `TFEL/PhysicalConstants` library are available through the PhysicalConstants type alias.
  - The keywords `@StateVariable` and `@ExternalStateVariable` are synomymous of the `@Output` and `@Input` keywords respectively for consistency with behaviours.
  - The keywords `@StateVariable` (`@Output`), `@ExternalStateVariable` (`@Input`) and `@Parameters` now allow to specify the type of the variables they define. Note that only scalar types are supported by the `Model` DSL.
  - Quantities are now fully supported in the `Model` DSL.

```cpp
constexpr auto R = PhysicalConstants::R;
```

- Improvements to the `MaterialProperty` DSL :
  - The physical constants defined in the `TFEL/PhysicalConstants` library are available through the PhysicalConstants type alias.
- Improvements to the `Model` DSL
  - The physical constants defined in the `TFEL/PhysicalConstants` library are available through the PhysicalConstants type alias.
  - The keywords `@StateVariable` and `@ExternalStateVariable` are synomymous of the `@Output` and `@Input` keywords respectively for consistency with behaviours.
  - The keywords `@StateVariable` (`@Output`), `@ExternalStateVariable` (`@Input`) and `@Parameters` now allow to specify the type of the variables they define. Note that only scalar types are supported by the `Model` DSL.
  - Quantities are now fully supported in the `Model` DSL.

```cpp
@LocalVariable derivative_type<stress, temperature, temperature> d2E_dT2;
```

- Extension of `derivative_type` to higher order derivatives.

Miscellaneous improvements - II

- `madnex file support`:
  - Add the contents of a `madnex` file to the search paths.
  - Automatic declaration of a madnex input file as a `madnex` search path.
- New domain specific language `ImplicitCZM`
- `generic` interface for material properties
- `generic` interface for point-wise models implemented using the `Model` domain specific language

New features of MTest in Version 4.1

Miscellaneous improvements

```
$ mtest ——behaviour=Plasticity ——test=UniaxialTensileTest       \
        ——@interface@=cyrano ——@behaviour@="'cyranoplasticity'"  \
        ——@library@="'src/libCyranoBehaviours.so'"               \
        Plasticity .mdnx
```

- Support of `madnex` file :

Miscellaneous improvements

```
$ mtest  --behaviour=Plasticity --test=UniaxialTensileTest      \
         --@interface@=cyrano --@behaviour@="'cyranoplasticity'" \
         --@library@="'src/libCyranoBehaviours.so'"             \
         Plasticity.mdnx
```

- Support of `madnex` file :
- Support for material properties generated with the `generic` interface.

Miscellaneous improvements

```
$ mtest  --behaviour=Plasticity --test=UniaxialTensileTest      \
         --@interface@=cyrano --@behaviour@="'cyranoplasticity'" \
         --@library@="'src/libCyranoBehaviours.so'"            \
         Plasticity.mdnx
```

- Support of `madnex` file :
- Support for material properties generated with the `generic` interface.
- Support for extended types (see the `MGIS` part).

Miscellaneous improvements

```
$ mtest --behaviour=Plasticity --test=UniaxialTensileTest      \
        --@interface@=cyrano --@behaviour@="'cyranoplasticity'" \
        --@library@="'src/libCyranoBehaviours.so'"             \
        Plasticity.mdnx
```

- Support of `madnex` file :
- Support for material properties generated with the `generic` interface.
- Support for extended types (see the `MGIS` part).
- Support for a boundary condition modelling the effect of a mandrel in pipe modelling.

MFront on the GPUs, an on-going work

Current status

- Partial port of `TFEL/Math` and `TFEL/Material`

Current status

- Partial port of `TFEL/Math` and `TFEL/Material`
- Some simple behaviours have been implemented to `GPUs` using several frameworks (`SYCL`, `CUDA`, `Kokkos`)

Current status

- Partial port of `TFEL/Math` and `TFEL/Material`
- Some simple behaviours have been implemented to `GPUs` using several frameworks (`SYCL`, `CUDA`, `Kokkos`)
- Requires a deep integration between `MFront` and `MGIS`

Current status

- Partial port of `TFEL/Math` and `TFEL/Material`
- Some simple behaviours have been implemented to `GPUs` using several frameworks (`SYCL`, `CUDA`, `Kokkos`)
- Requires a deep integration between `MFront` and `MGIS`
- Probably requires on the fly compilation of a dedicated kernel using DSL options to optimize the code.

Current status

- Partial port of `TFEL/Math` and `TFEL/Material`
- Some simple behaviours have been implemented to `GPUs` using several frameworks (`SYCL`, `CUDA`, `Kokkos`)
- Requires a deep integration between `MFront` and `MGIS`
- Probably requires on the fly compilation of a dedicated kernel using DSL options to optimize the code.
- A difficult part is error handling and error reporting which requires a deep overhaul of the `TFEL/Math` and `TFEL/Material` libraries and the code generated by `MFront`
  - Removal of exception usage
  - Generation of a dedicated function (on the host) per behaviour for retrieving error messages from an integer exit status.

Current status

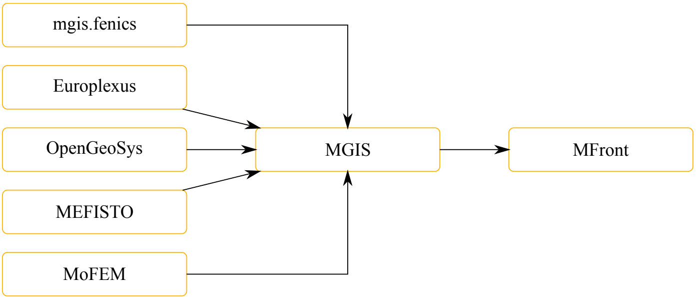
- Partial port of `TFEL/Math` and `TFEL/Material`
- Some simple behaviours have been implemented to `GPUs` using several frameworks (`SYCL`, `CUDA`, `Kokkos`)
- Requires a deep integration between `MFront` and `MGIS`
- Probably requires on the fly compilation of a dedicated kernel using DSL options to optimize the code.
- A difficult part is error handling and error reporting which requires a deep overhaul of the `TFEL/Math` and `TFEL/Material` libraries and the code generated by `MFront`
  - Removal of exception usage
  - Generation of a dedicated function (on the host) per behaviour for retrieving error messages from an integer exit status.
- A post-doctoral position is open.

Current status

- Partial port of `TFEL/Math` and `TFEL/Material`
- Some simple behaviours have been implemented to `GPUs` using several frameworks (`SYCL`, `CUDA`, `Kokkos`)
- Requires a deep integration between `MFront` and `MGIS`
- Probably requires on the fly compilation of a dedicated kernel using DSL options to optimize the code.
- A difficult part is error handling and error reporting which requires a deep overhaul of the `TFEL/Math` and `TFEL/Material` libraries and the code generated by `MFront`
  - Removal of exception usage
  - Generation of a dedicated function (on the host) per behaviour for retrieving error messages from an integer exit status.
- A post-doctoral position is open.
- A project named `tfel-gpu` may be released in the near future.

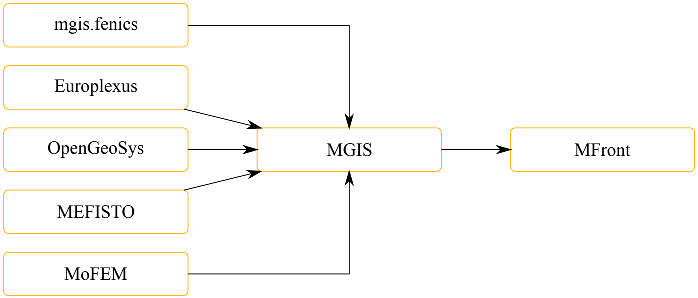
The MGIS project

Overview of the `MFrontGenericInterfaceSupport` project (MGIS)



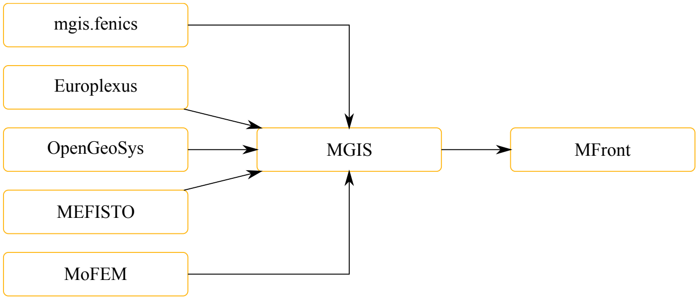
- The `MGIS` project provides classes on the solver side to retrieve **metadata** from an `MFront` behaviour and call the behaviour integration over a time step.

Overview of the `MFrontGenericInterfaceSupport` project (MGIS)



- The `MGIS` project provides classes on the solver side to retrieve **metadata** from an `MFront` behaviour and call the behaviour integration over a time step.

Overview of the MFrontGenericInterfaceSupport project (MGIS)



- The `MGIS` project provides classes on the solver side to retrieve **metadata** from an `MFront` behaviour and call the behaviour integration over a time step.
- Written in `C++`. Bindings exists for `C`, `Fortran2003`, `python`, `Julia`. And also used/tested in `XPer`, `Kratos Multiphysics`, `JuliaFEM`, `NairmMPM`, `esys.escript`, `DUNE`, `HELIX` (based on `MFEM`).

Support for extended types

```
$ python3
>>> import mgis.behaviour
>>> print(mgis.behaviour.getVariableTypeSymbolicRepresentation(780))
derivative_type<stensor<N, real>, tensor<N, real>>
```

- The `Variable::Type` enumeration may now hold the following values : `SCALAR`, `VECTOR`, `VECTOR_1D`, `VECTOR_2D`, `VECTOR_3D`, `STENSOR`, `STENSOR_1D`, `STENSOR_2D`, `STENSOR_3D`, `TENSOR`, `TENSOR_1D`, `TENSOR_2D`, `TENSOR_3D`, `HIGHER_ORDER_TENSOR` and `ARRAY`.

Support for extended types

```
$ python3
>>> import mgis.behaviour
>>> print(mgis.behaviour.getVariableTypeSymbolicRepresentation(780))
derivative_type<stensor<N, real>, tensor<N, real>>
```

- The `Variable::Type` enumeration may now hold the following values : `SCALAR`, `VECTOR`, `VECTOR_1D`, `VECTOR_2D`, `VECTOR_3D`, `STENSOR`, `STENSOR_1D`, `STENSOR_2D`, `STENSOR_3D`, `TENSOR`, `TENSOR_1D`, `TENSOR_2D`, `TENSOR_3D`, `HIGHER_ORDER_TENSOR` and `ARRAY`.
- The Variable class exposes an integer named `type_identifier`
  - The `getVariableTypeSymbolicRepresentation` returns a symbolic representation of a object using a C++-like representation from a type identifier
  - See `https://thelfer.github.io/tfel/web/mfront-types.html` for details.

Support for behaviours' `initialize` functions

```
auto d = BehaviourData{b};
//   initialize   the material properties and the external state variables
 ...
//  calling  an  initialize   function which requires an input
auto inputs = allocateInitializeFunctionVariables(b,  "StressFromInitialPressure");
inputs[0] = pr;
auto v = make_view(d);
executeInitializeFunction(v,  b,  "StressFromInitialPressure", inputs);
```

- The `Behaviour` class exposes a data member named `initialize_functions` which associates the name of `initialize` function and a small data structure containing :
  - the pointer to the initialize function
  - the list of inputs of the initialize function

Support for behaviours' `initialize` functions

```
auto d = BehaviourData{b};
//  initialize  the material properties and the external state variables
 ...
// calling an initialize  function which requires an input
auto inputs = allocateInitializeFunctionVariables(b, "StressFromInitialPressure");
inputs[0] = pr;
auto v = make_view(d);
executeInitializeFunction(v, b, "StressFromInitialPressure", inputs);
```

- The `Behaviour` class exposes a data member named `initialize_functions` which associates the name of `initialize` function and a small data structure containing :
  - the pointer to the initialize function
  - the list of inputs of the initialize function
- The following free functions are now available :
  - `getInitializeFunctionVariablesArraySize` returns the size of an array able to contain the inputs for an integration point.
  - `allocateInitializeFunctionVariables` returns an array able to store the inputs of an initialize function.
  - `executeInitializeFunction` executes an initialize function.

Support for behaviours' `initialize` functions

```
auto d = BehaviourData{b};
//   initialize   the material properties and the external state variables
 ...
//  calling  an  initialize   function which requires an input
auto inputs = allocateInitializeFunctionVariables(b,  "StressFromInitialPressure");
inputs[0] = pr;
auto v = make_view(d);
executeInitializeFunction(v,  b,  "StressFromInitialPressure", inputs);
```

- The `Behaviour` class exposes a data member named `initialize_functions` which associates the name of `initialize` function and a small data structure containing :
  - the pointer to the initialize function
  - the list of inputs of the initialize function
- The following free functions are now available :
  - `getInitializeFunctionVariablesArraySize` returns the size of an array able to contain the inputs for an integration point.
  - `allocateInitializeFunctionVariables` returns an array able to store the inputs of an initialize function.
  - `executeInitializeFunction` executes an initialize function.
- See the documentation for further details.

Support for behaviours' post-processings

```
auto m = MaterialDataManager{b, 2u};
//   initialize   the state  and perform the behaviour integration
 ...
// execute the post—processing
auto outputs = allocatePostProcessingVariables(m, "PrincipalStrain");
executePostProcessing(outputs, m, "PrincipalStrain");
```

- The `Behaviour` class exposes a `postprocessings` member which associates the name of a postprocessing and a small data structure containing :
  - the pointer to the postprocessing,
  - the list of outputs of the postprocessing.

Support for behaviours' post-processings

```
auto m = MaterialDataManager{b, 2u};
//   initialize   the state  and perform the behaviour integration
 ...
// execute the post−processing
auto outputs = allocatePostProcessingVariables(m, "PrincipalStrain");
executePostProcessing(outputs, m, "PrincipalStrain");
```

- The `Behaviour` class exposes a `postprocessings` member which associates the name of a postprocessing and a small data structure containing :
  - the pointer to the postprocessing,
  - the list of outputs of the postprocessing.
- The following free functions are now available :
  - `getPostProcessingVariablesArraySize` returns the size of an array able to contain the outputs a postprocessing for an integration point.
  - `allocatePostProcessingVariables` returns an array able to store the outputs of a postprocessing.
  - `executePostProcessing` executes a postprocessing.

Miscellaneous improvements

- Support for material properties and point-wise models.
- The `extractInternalStateVariable` function can now be used to extract the value of an internal state variable in a pre-allocated buffer.

The MFrontGallery project

The MFrontGallery projet

```
mfront_behaviours_library(Bentonite
  BentoniteMassin2017)
```

- MFrontGallery is an open-source projet which :
  - provides a cmake infrastructure to build, deploy and tests MFront implementations. This infrastructure is meant to be create dedicated child projects.
    - Those child projects can be generated by the CADEEX database.
  - capitalizes a set of well-implemented and documented MFront implementations, mostly for MFront tutorials (https://thelfer.github.io/tfel/web/gallery.html)
- https://thelfer.github.io/MFrontGallery/web/index.html
- https://thelfer.github.io/MFrontGallery/web/creating-derived-project.html
- https://github.com/thelfer/MFrontGallery/blob/master/docs/papers/joss/paper.md

Advances features of MFrontGallery projet

```
cmake ../MFrontGallery/ —DCMAKE_BUILD_TYPE=Release —Denable—c=ON —Denable—c++=ON —Denable—excel=ON —Denable—
    fortran=ON —Denable—python=ON —Denable—java=ON —Denable—octave=ON —Denable—generic—behaviours=ON —
    Denable—castem=ON —Denable—castem—behaviours=ON —Denable—aster=ON —Denable—cyrano=ON —Denable—ansys=
    ON —Denable—europlexus=ON —Denable—calculix=ON —Denable—abaqus=ON —Denable—zmat=ON —
    DZSET_INSTALL_PATH=/home/th202608/codes/ZeBuLoN/8.5/Z8.5 —DCASTEM_INSTALL_PATH=/home/th202608/codes/castem
    /2014/install —Denable—castem—pleiades=ON —Denable—diana—fea=ON —Denable—mfront—documentation—generation
    =ON —DCMAKE_INSTALL_PREFIX=/home/th202608/codes/MFrontGallery/master/install —DMFM_BUILD_IDENTIFIER=Mecanum —
    DMFM_TREAT_PARAMETERS_AS_STATIC_VARIABLES=true
```

- Automatic handling of dependencies between `MFront` files.
- Support for `MADNEX` files.
- Support for integration tests described by `mfm-test-generator`.
- Integration of DSL options (TFEL version 4.1) to customize builds :
  - Build identifiers,
  - Runtime modification of parameters,
  - etc...

Conclusions and perspectives

What's planned for TFEL Version 4.2

- Continue experiments on GPUs

What's planned for TFEL Version 4.2

- Continue experiments on GPUs
- Revamping error handling and error reporting in MFront

What's planned for TFEL Version 4.2

- Continue experiments on GPUs
- Revamping error handling and error reporting in MFront
- New interfaces :
  - Plaxis : https://github.com/thelfer/tfel/issues/272
  - Flac3D : https://github.com/thelfer/tfel/issues/271

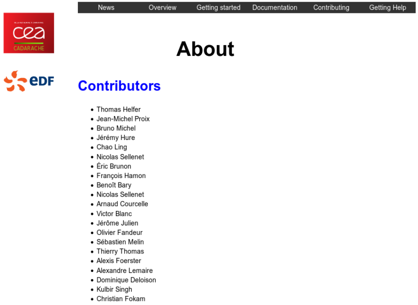
What's planned for TFEL Version 4.2

- Continue experiments on GPUs
- Revamping error handling and error reporting in MFront
- New interfaces :
  - Plaxis : https://github.com/thelfer/tfel/issues/272
  - Flac3D : https://github.com/thelfer/tfel/issues/271
- Automatically define some initialize functions and post-processing by the StandardElasticity and the StandardElastoViscoplascity bricks.

Futures developments in `MGIS`

- Support for material properties.
- Context aware "Just-In Time" compilation of the behaviour :
  - Values of parameters, uniform material properties and external state variables are turned into `static constexpr` variables.
- Port of `MFront/MGIS` on `GPU`.
- Data structures for `GPU`, Adaptative mesh refinements.
- Optional integration with `MEDCoupling` for exporting values at integration points.

How to contribute


News Overview Getting started Documentation Contributing Getting Help

About

Contributors

- Thomas Helfer
- Jean-Michel Proix
- Bruno Michel
- Jérémy Hure
- Chao Ling
- Nicolas Sellenet
- Éric Brunon
- François Hamon
- Benoît Bary
- Nicolas Sellenet
- Arnaud Courcelle
- Victor Blanc
- Jérôme Julien
- Olivier Fandeur
- Sébastien Melin
- Thierry Thomas
- Alexis Foerster
- Alexandre Lemaire
- Dominique Deloison
- Kulbir Singh
- Christian Fokam

- Citations and illustrations
- Feed-backs, feed-backs, and feed-backs!
  - Please use the forum.
  - Enhancement suggestions (code, documentation, algorithm, etc...)
- Submit new behaviours implementation and tests.
- Submit pages to the gallery.
- Code (for the braves)

Thank you for your attention.
Time for discussion !

https://tfel.sourceforge.net
https://www.researchgate.net/project/TFEL-MFront
https://twitter.com/TFEL_MFront
https://github.com/thelfer/

tfel-contact@cea.fr

The development of MFront is supported financially by CEA, EDF and Framatome in the framework of the PLEIADES project.