

# Knowledgebase of Interatomic Models Application Programming Interface (KIM API)

Ryan S. Elliott, Ellad B. Tadmor

Dept. of Aerospace Engineering and Mechanics, University of Minnesota

July, 2014

This document describes how KIM **Tests** and **Models** written in different languages work together. A unified interface, tuned for the specific needs of atomistic simulations, is presented. This interface is based on the concept of “descriptor files”. A descriptor file specifies all variables and methods required for communication between a particular **Model** and a **Test**. A “KIM API object” is created, based on the descriptor files, that hold all arguments (variable/data and method pointers) needed for **Test/Model** interaction. A complete set of KIM API service routines are available for accessing the various pointers in the KIM API object.

# Contents

---

## KIM overview

- Barriers faced by molecular modelers
- Knowledgebase of Interatomic Models (KIM) is proposed to overcome the barriers
- KIM framework
- KIM repository: Models
- KIM repository: Tests
- KIM repository: KIM data

## KIM API concept and implementation:

1. The KIM API facilitates communication between **Models** and **Tests**
2. The most challenging technical requirement is the need for multi-language support
3. The KIM API is based on exchanging pointers to data and methods
4. How can a **Test** know what type of input/output data is required by a **Model**?  
We have solved this problem by introducing the KIM API descriptor file
5. The structure of a descriptor file
6. Handling of Neighbor lists and Boundary Conditions – NBC methods
7. Test/Model coupling: The Model's initialization routine stores a pointer to the “compute” routine in the KIM API object
8. Initialization of a KIM API object, setting and getting data-pointers can be done through the KIM service routines

# Contents (2)

---

## Appendix

1. Every argument that needs to be communicated between **Tests** and **Models** must be in the descriptor file
2. The KIM API directory structure

---

## KIM overview

# KIM TEAM



## PIs

Ellad Tadmor (U. Minnesota)  
Ryan Elliott (U. Minnesota)  
James Sethna (Cornell)

## Developers

Alex Alemi (Cornell)  
Matt Bierbaum (Cornell)  
Woo Song Choi (Cornell)  
Daniel Karls (U. Minnesota)  
John Crow (Silicon Life Sciences)  
Trevor Wenblom (Silicon Life Sciences)

## Advisory Board

Graeme Ackland (U. Edinburgh)  
Michael Baskes (LANL)  
Chandler Becker (NIST)  
Noam Bernstein (NRL)  
Ioana Cozmuta (NASA)  
Karsten Jacobsen (Tech. U. Den.)  
Ronald Miller (Carleton)  
John Moriarty (LLNL)  
Sadasivan Shankar (Intel)  
Adri van Duin (Penn State)  
Gabriel Wainer (Carleton)

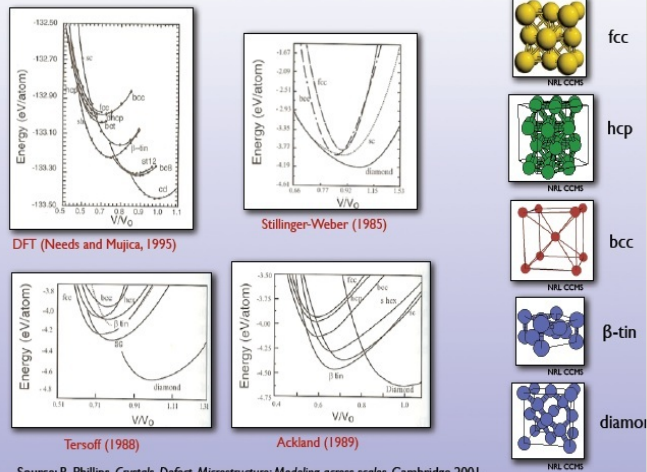
# Molecular/atomistic simulations: tests and models

## Tests

**Test** : a specific computer program which, when coupled with a suitable Model, calculates and returns a specific Prediction about a particular Configuration (or sequence of Configurations for dynamical properties).

### Example of a Molecular Simulation

#### Transferability: Silicon bulk phases



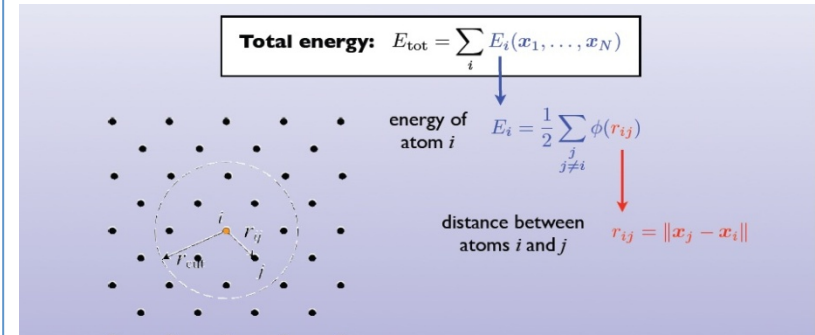
Source: R. Phillips, *Crystals, Defect, Microstructure: Modeling across scales*, Cambridge 2001.

E. B. Tadmor

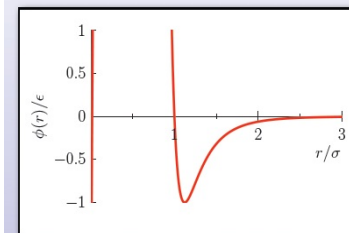
45

## Models

**Model** : Computer implementation representing a specific interaction between atoms, e.g. an interatomic potential or force field



The Lennard-Jones potential is a simple pair potential, which describes the interaction between two uncharged atoms:



$$\phi(r) = 4\epsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right]$$

repulsion due to overlapping electrons (Pauli principle)      van der Waals attraction between transient dipoles

- Two fitting parameters ( $\sigma$ ,  $\epsilon$ )
- Designed for the noble gases (Ne, Ar, Kr, Xe).

# Types of molecular modelers

---

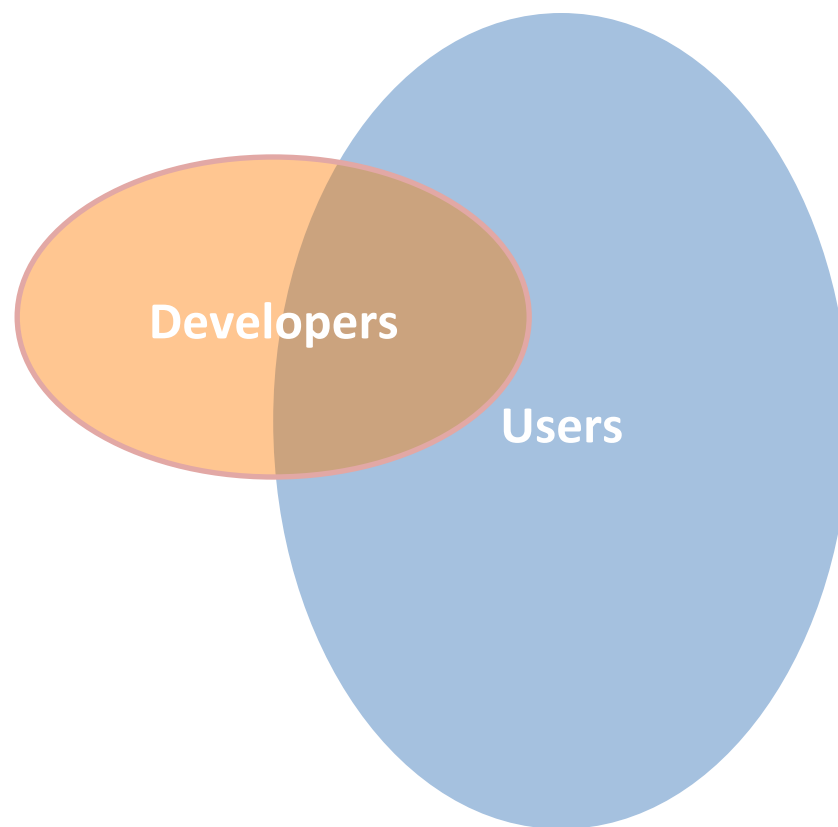
Very broadly speaking there are two types of *molecular modelers*:

## Developers

- Create new models
- Study materials physics and applications
- Create new knowledge

## Users

- Use models to study materials problems of scientific/technological importance
- Build sophisticated simulations to extract meaningful data
- Create new knowledge



## Barriers faced by molecular modelers

---

The difficulties faced by developers and users of interatomic models include:

1. No easy access to an extensive list of reliable *reference data* from experiments and first principles calculations for fitting.
2. No easy access to implementations of existing models with known *provenance* and *cross-language capability*.
3. No *standardized tests* for evaluating properties of molecular systems.
4. No framework for evaluating the *precision and transferability* of models and therefore no *rigorous guidelines* for choosing an appropriate model for a given application.



## Knowledgebase of Interatomic Models (KIM) is proposed to overcome the barriers

---

The *Knowledgebase of Interatomic Models (KIM)* project is based on a four-year NSF cyber-enabled discovery and innovation (CDI) grant. The KIM project is designed to overcome the barriers mentioned on the previous page. KIM has the following main objectives:

- Development of an *online open resource* for standardized testing and long-term warehousing of interatomic models (potentials and force fields) and data.
- Development of an *application programming interface (API)* standard for atomistic simulations, which will allow any interatomic model to work seamlessly with any atomistic simulation code.
- Fostering the development of a quantitative theory of *transferability* of interatomic models to provide guidance for selecting application-appropriate models based on rigorous criteria, and error bounds on results.
- Striving for the permanence of the KIM project, including development of a sustainability plan, and establishment of a long-term home for its content.

More information on KIM is available at the project website: <http://openKIM.org>  
University of Minnesota

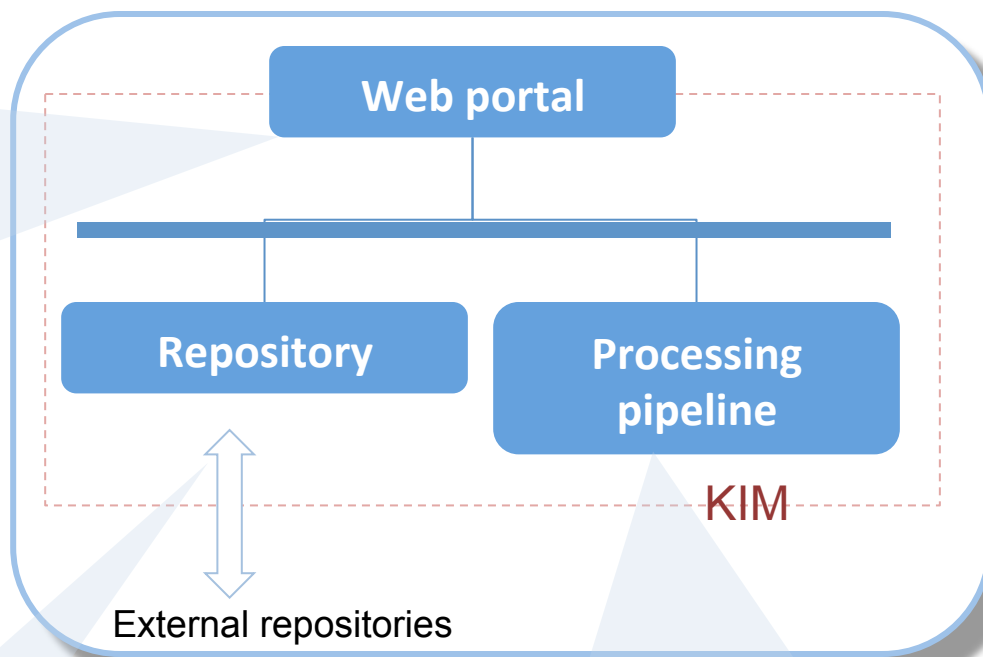
# KIM framework

## A web interface that will facilitate:

- user **upload** and **download** of Tests, Models and Reference Data
- **searching** and querying the repository
- comparing and **visualizing** Predictions and Reference Data
- recording **user feedback** (ranking and discussion forums)

## A user-extendible database of

- interatomic **Models**
- standardized **Tests** (simulation codes)
- **Predictions** (results from Model-Test couplings)
- **Reference Data** (obtained from experiments and first principles calculations)



## Processing Pipeline:

**An automatic system for generating Predictions due to new Test or Model upload or changes:**

- detect viable **Test-Model couplings**
- assign **computational resources** based on priority and dependencies
- **store** results in Repository
- requires an application programming interface (**API**) to be defined

# KIM repository: Models

## Models

## Tests

## Predictions

## Reference Data

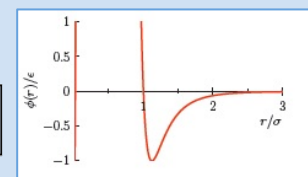
## KIM API

**Model:** Computer implementation representing a specific interaction between atoms, e.g. an interatomic potential or force field.

- Model Format

- Stand-alone Model (black box)
- Model Driver (e.g. Lennard-Jones )
- + Parameter Set (e.g.  $\epsilon_{\text{Ar}} = 10.4 \text{ meV}$ ,  $\sigma_{\text{Ar}} = 0.34 \text{ nm}$ )

$$\phi(r) = 4\epsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right]$$



<b>Lennard-Jones (pair)</b>	:	:	<b>EDIP</b>
• Ar parameterization	<b>Stillinger-Weber (3-bdy)</b>	<b>CHARMM/AMBER</b>	<b>Brenner</b>
• ...	• Si parameterization	:	:
:	• ...	<b>EAM/Finnis-Sinclair/glue</b>	<b>Bond-order potentials</b>
<b>Morse (pair)</b>	:	:	:
• Cu parameterization	<b>MGPT (4-body)</b>	<b>MEAM</b>	<b>ReaxFF</b>
• ...	• Mo parameterization	:	:
:	• Ta parameterization	<b>Tersoff</b>	<b>GAP</b>
<b>Born-Mayer (ionic pair)</b>	• ...	:	:

- Every model will have a **unique KIM ID** for referencing in papers.

# KIM repository: Tests

Models

Tests

Predictions

Reference Data

KIM API

**Test:** a specific computer program which when coupled with a suitable Model, possible including additional input, calculates and returns a specific Prediction about a particular Configuration (or sequence of Configurations for dynamical properties).

- *Prediction* of a Test will be a logical, scalar, tensor, graph, configuration or field, computed from a *Test-Model coupling*

## Scalars

- lattice constants
- cohesive energy
- vacancy formation energy
- surface energy
- grain boundary energy
- vacancy migration barrier
- dislocation mobility
- peierls stress
- melting temperature
- ...

## Tensors

- stress
- elastic constants
- ...

## Configurations

- dislocation core structure
- surface structure
- grain boundary structure
- nanocluster structure
- ...

## Graph

- phonon spectrum
- cohesive energy vs volume
- energy along transition path
- radial distribution functions
- ...

## Fields

- simulated TEM hi-res image
- gamma surface
- ...

- Popular codes (ddcMD, DL\_POLY, GROMACS, GULP, IMD, LAMMPS, NAMD, SPaSM, etc.) can be included in a library of tools for writing *Tests*.
- *Automatic test generation* by linking to external repositories of first principles results.

# KIM repository: KIM Data

Models

Tests

Predictions

Reference Data

KIM API

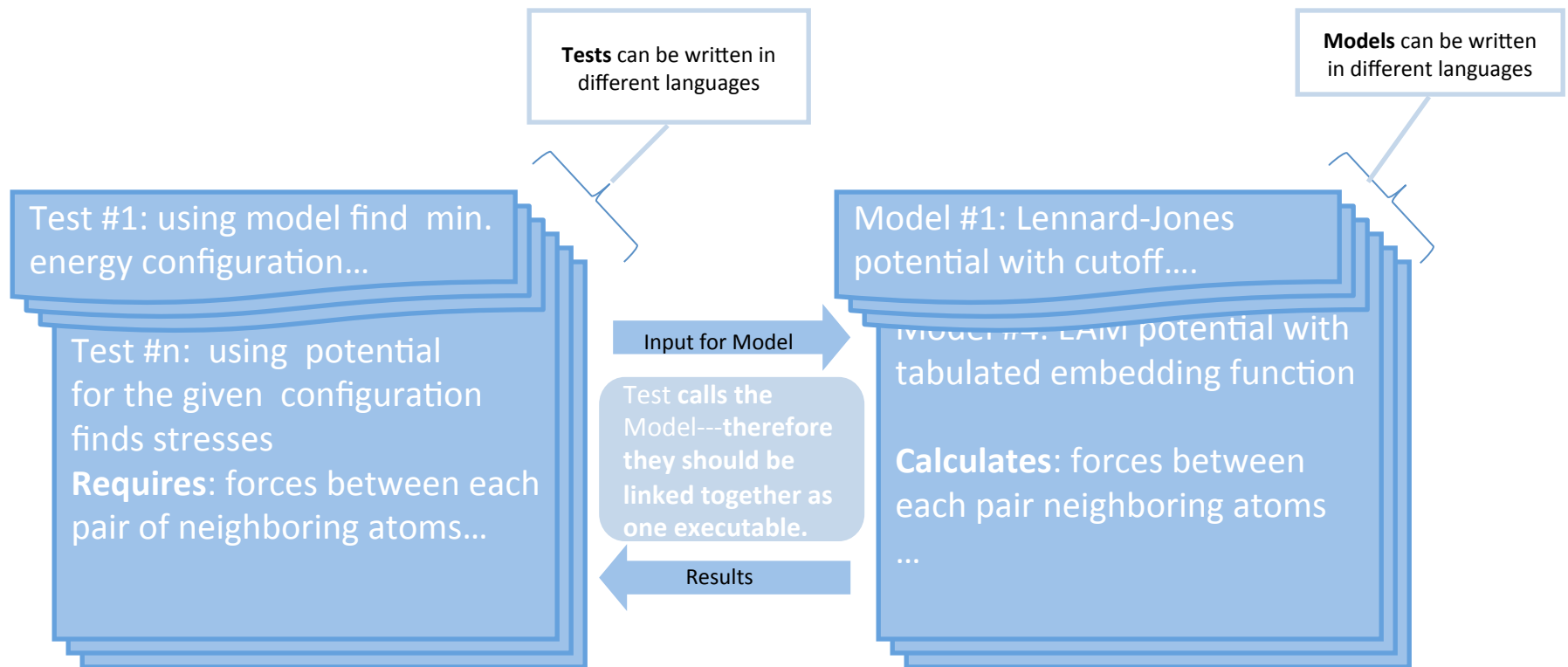
## Data in KIM can either be

- a *Prediction* computed from a Test-Model coupling, or
- *Reference Data* computed by first principles or measured experimentally.
- **Standardization** of Data
  - Identified in terms of a set of “descriptors” drawn from a standardized “dictionary” (similar to that used in the Protein Data Bank project)
  - Descriptors will be automatically generated when possible (for example, the “Space Group” descriptor will be automatically generated for a given crystal structure).
- **Data classes**
  - *Logical* (true/false result for a test, e.g. a given crystal phase is stable)
  - *Scalar or Tensor* (lattice constant, cohesive energy, elastic constants...)
  - *Graphs* (transition pathway energy, phonon spectrum, ...)
  - *Configurations* (relaxed defect core, surface structure, ...)
  - *Fields* (simulated hires TEM image, ...)
- **Quality** assurance
  - Acceptance of only “publication quality” data enforced by KIM Editor
  - “Data Provenance”

---

## KIM API concept and implementation

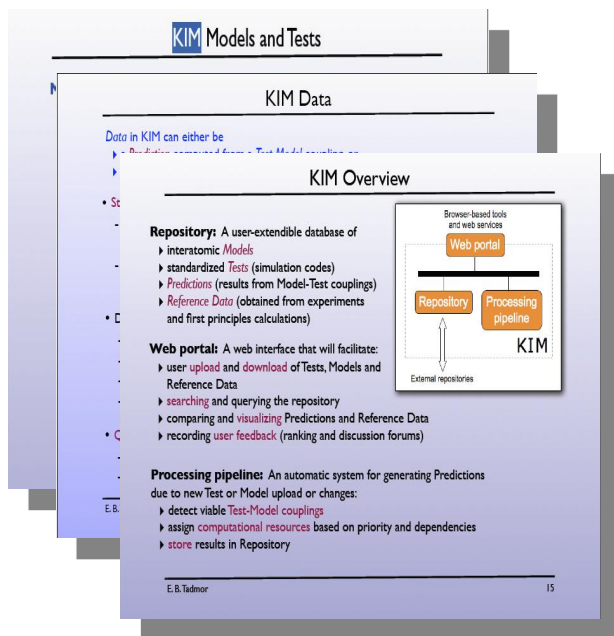
# The KIM API facilitates communication between Models and Tests



Users and developers will be able to download **Tests** and **Models** (from [openkim.org](http://openkim.org)) , then compile, link and run the resulting programs to produce new results.

# The most challenging technical requirement is the need for multi-language support

## openKIM.org framework



**Processing pipeline:** an automatic system for generating predictions when Tests or Models are uploaded or changed.

## Requirements:

- Multilanguage support (C, C++, F77, FORTRAN 90, Python ...)
- A variety of data structures need to be accommodated: scalars, multidimensional arrays, variable size arrays, etc..
- Speed & performance are very important
- Standardized API, version tracking, etc...

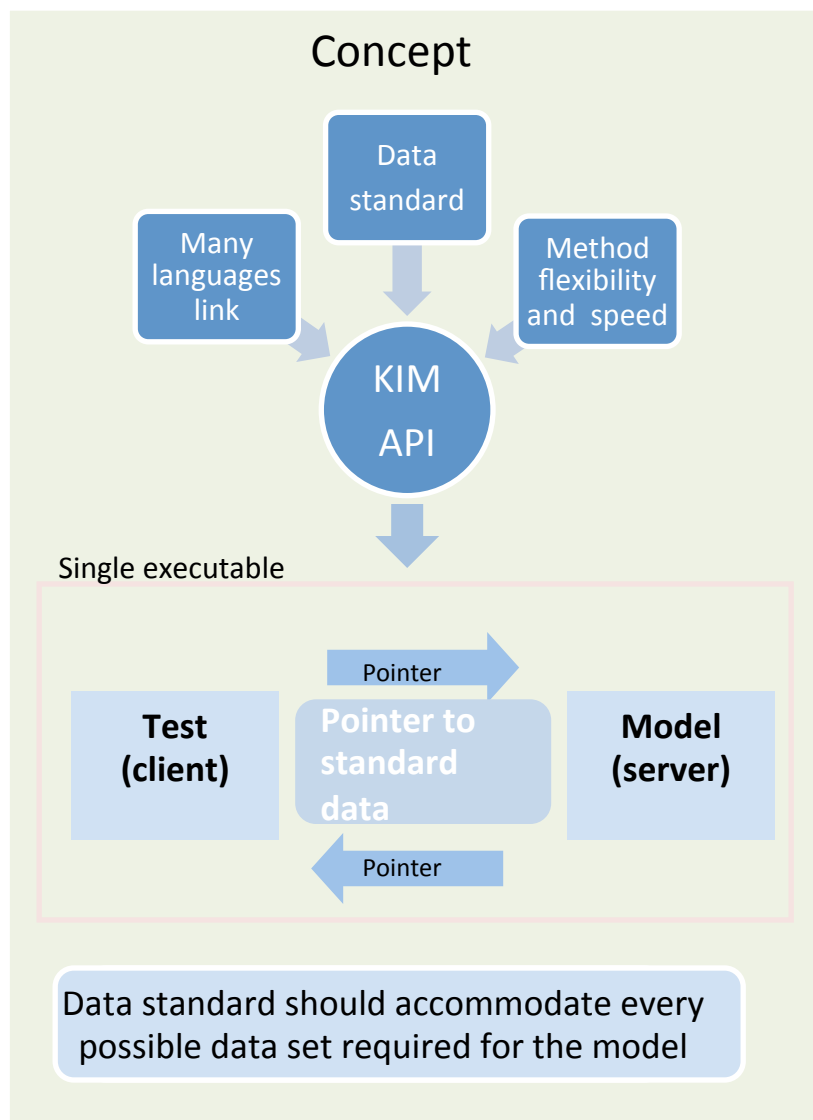
## Processing pipeline: sequence of actions

- detect a viable **Model/Test** coupling
- **build (compile and link) Tests against Model**
- **run probe-tests**
- assign computational resources
- run full-scale **Test** against **Model**
- analyze results ...
- store results in the repository

Need a simple interface : ideally just one argument per call

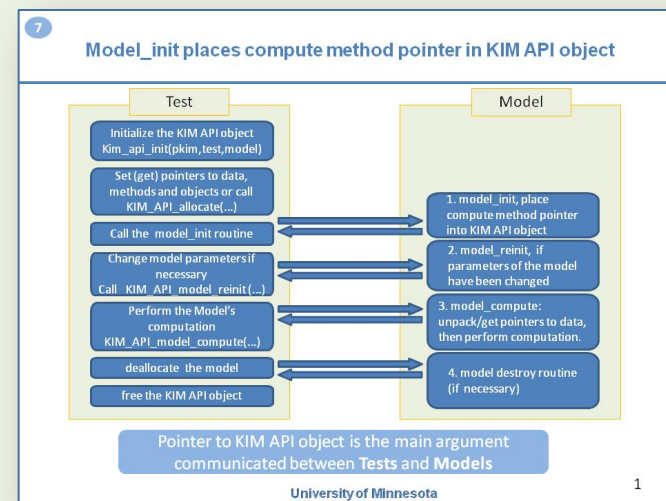


# The KIM API is based on exchanging pointers to data and methods



## Schematic of implementation

1. Data and method pointers are packed in one object. The Interface consists of exchanging one pointer to the KIM API object between a **Test** and a **Model**
2. All languages naturally support pointers:
  - FORTRAN (cray or 2003 standard)
  - C/C++
  - Java
  - Python



## Using C-style pointers in Fortran

In order to implement the KIM API concept in a cross-language environment, all languages have to work with C-style pointers.

FORTTRAN 77 and Fortran 90/95 do not support C-style pointers directly, however Fortran 2003 provides the `iso_c_binding` module for standard C/Fortran interoperability. An example below shows the general syntax and usage of a `iso_c_binding` in Fortran compared with C.

### FORTTRAN code

```
use, intrinsic :: iso_c_binding
...
real(c_double), target :: y=10.0d0
real(c_double), pointer :: x
type(c_ptr) px
...
px = c_loc(y)
call c_f_pointer(px, x)
print*, "x=", x
...
```

px - is a pointer (analog  
double \*x in C)  
x - is a pointee

### C code

```
...
double y=10.0;

double *x;

x = &y;
printf("*x=%f \n", *x);
...
```

## How can a Test know what type of input/output data is required by a Model? We have solved this problem by introducing the KIM API descriptor file

### ex\_model\_Ne\_P\_MLJ\_NEIGH\_PURE\_H.kim

```
#####
Unit_Handling      := fixed
...
#####
SUPPORTED_ATOM/PARTICLES_TYPES:
# Symbol/name      Type              code

Ne                  spec              1
...

MODEL_INPUT:
# Name              Type              Unit              Shape              Requirements

numberOfParticles   int              none              []
...
numberParticleTypes int              none              []
particleTypes       int              none              [numberOfParticles]
...
```

KIM API descriptor file defines all arguments that the model needs for computation including input and output arguments. Also on the test side, the .kim file defines what the Test can provide as input for the Model and what it expects from the Model as a result.

Tests and Models expose the required input/output arguments  
that will be communicated using the KIM API

Note: full .kim file shown here can be found in `EXAMPLES/MODELS/ex_model_Ne_P_MLJ_NEIGH_PURE_H/`

# Structure of descriptor file

## Model/Test name and system of units lines

```
Unit_Handling      := flexible
```

## Section lines

```
SUPPORTED_ATOM/PARTICLES_TYPES:
```

```
CONVENTIONS:
```

```
MODEL_INPUT:
```

```
MODEL_OUTPUT:
```

```
MODEL_PARAMETERS:
```

## Data lines

- \* Species Data lines
- \* Flag Data lines
- \* Argument Data lines

## Brief description of Section lines

These lines identify logically distinct sections within the KIM descriptor file.

All lines following a Section line, up to the next Section line or end of the file, will be assigned to the indicated section.

These sections may occur in any order within a KIM descriptor file, however the order given here is recommended. A section line may only occur once within a KIM descriptor file.

## Brief description of Data lines

These lines are used to specify the information that a Model (Test) will provide to and require from a Test (Model), as well as the conventions that the Model(Test) uses.

\* Species Data lines - allow for the definition of atomic species by providing a symbol and an integer code. These lines are located in section SUPPORTED\_ATOM/PARTICLES\_TYPES.

\* Flag Data lines - this line type defines a convention that can be used to ensure that Models and Tests are able to work together, and should only be used within the CONVENTIONS section of the KIM descriptor file.

\* Argument Data lines - the main KIM descriptor file line format, used within the MODEL\_INPUT, MODEL\_OUTPUT, and MODEL\_PARAMETERS sections.

## 5.1 Each argument line in the descriptor file describes an argument and its properties

### EXAMPLES/MODELS/ex\_model\_Ar\_P\_MLJ\_F03.kim

```
Unit_Handling      := fixed
...
compute            method      none      []

MODEL_OUTPUT:
# Name              Type        Unit      Shape      Requirements
energy              double      energy   []          optional
forces              double      force    [numberOfParticles,3] optional
```

All characters after a '#' are ignored  
(a comment field)

Method means a  
subroutine or function  
pointer

The name of an argument is its "key word". By using key words, the KIM service routines can pack/unpack data pointers from the KIM API object. Key words are standardized as part of the KIM API.

Type of data in computer representation

Physical dimensions

The shape of an argument describes its array properties. It specifies the number (rank) and size (range or extent) of indices. For example, [] means a scalar (zero-dimensional array), [numberOfParticles] means a one-dimensional array and [numberOfParticles,3] means a two-dimensional array of size numberOfParticles x 3.

The "requirements" field is only used in **Model** descriptor files. An empty field indicates that the argument is required. A value of "optional" indicates that the associated data will be computed only if the argument is in the **Test's** descriptor file and if the **Test** explicitly requests it.

Note: a detailed description of all Type values and Units can be found in the file DOCS/standard.kim

## Specifying particle types – species data lines

### EXAMPLES/MODELS/ex\_model\_Ar\_P\_MLJ\_F03.kim

```
...
#####
SUPPORTED_ATOM/PARTICLES_TYPES:
# Symbol/name          Type          code
Ar                      spec          1
#####.
..
```

Species data lines define the atom/particle types supported by the Test/Model and should only be used within the SUPPORTED\_ATOM/PARTICLES\_TYPES section of the KIM descriptor file. Each line consists of three white-space separated (case sensitive) strings. The three strings are as follows:

**code:** This is the integer that the Model uses internally to identify the atom/particle type. The value specified by a Test is ignored.

**Type:** This must be 'spec'.

**Name:** This string gives a unique name to the atom/particle type. This name is checked against the standard list in 'standard.kim'.

The **KIM\_API\_get\_partcl\_types()** service routine allows one to obtain a list of all particle types used by the model during runtime. Also the **KIM\_API\_get\_partcl\_type\_code()** service routine allows one to get the particle type integer code (see DOCS/KIM\_API\_Description.txt).

## In order to define “conventions” of test/model behavior, flag data lines are reserved

### EXAMPLES/MODELS/ex\_model\_Ar\_P\_MLJ\_F03.kim

```
#####
CONVENTIONS:
# Name                               Type

OneBasedLists                        flag
Neigh_IterAccess                     flag
Neigh_LocaAccess                     flag
NEIGH_RVEC_H                         flag
NEIGH_PURE_H                         flag
NEIGH_RVEC_F                         flag
...
```

A flag data line defines a convention, that can be used to ensure that **Models** and **Tests** are able to work together, and should only be used within the CONVENTIONS section of the KIM descriptor file. The line consists of two white-space separated (case sensitive) strings. The two strings, in order, are as follows:

**Name:** This string gives a unique name to the convention. This name is checked against the standard list in `standard.kim`

**Type:** This must be `flag`

**KIM\_API\_allocate()** has **no effect** on “flag” type arguments, because they are not “data pointer holders”.

For a detailed description of all flag lines see the file DOCS/standard.kim. Also see the files in DOCS/.

## Parameter arguments are used to publish/access internal parameters of a Model

### EXAMPLES/MODELS/ex\_model\_Ar\_P\_MLJ\_CLUSTER\_F03/ex\_model\_Ar\_P\_MLJ\_CLUSTER\_F03.kim

MODEL\_PARAMETERS:

# Name	Type	Unit	Shape	Requirements
PARAM_FREE_sigma	double	length	[]	
PARAM_FREE_epsilon	double	energy	[]	
PARAM_FREE_cutoff	double	length	[]	
...				

The format for parameter arguments in a KIM descriptor file is the same as that for argument data types.

Two types of model parameters are allowed

- 1) PARAM\_FIXED\_XXXXXX - these should not be changed by the Test
- 2) PARAM\_FREE\_XXXXXX - these may be changed by the Test (which must then call the Model's `reinit()` function to inform the model that its parameters have changed)

`KIM_API_get_params()` service routine will return a list of all parameters in the object during runtime (as an array of text strings).

`KIM_API_get_free_params()` service routine will return a list of FREE parameters and

`KIM_API_get_fixed_params()` will return a list of FIXED parameters (see `KIM_API_Description.txt`)

Names of parameter arguments are not checked against `standard.kim`



## Specifying units that model can handle: Units Handling and base units

EXAMPLES/MODELS/ex\_model\_Ar\_P\_MLJ\_F90/ex\_model\_Ar\_P\_MLJ\_F03.kim

```
...
#####

Unit_Handling      := fixed

Unit_length        := A
Unit_energy        := eV
Unit_charge        := e
Unit_temperature   := K
Unit_time          := ps

#####
...
```

For Models, a variable 'Unit\_Handling' specifies whether the Model can adjust its input and output to match a Test ('flexible') or can only work with one set of units ('fixed'). This information is ignored for Tests.

Base unit lines:

Five lines that describe a set of five base units from which all other units are derived in a consistent way:

```
Unit_length      := `A' | `Bohr' | `cm' | `m' | `nm'
Unit_energy      := `amu*A^2/(ps)^2' | `erg' | `eV' |
                  `Hartree' | `J' | `kcal/mol' | `kJ/mol'

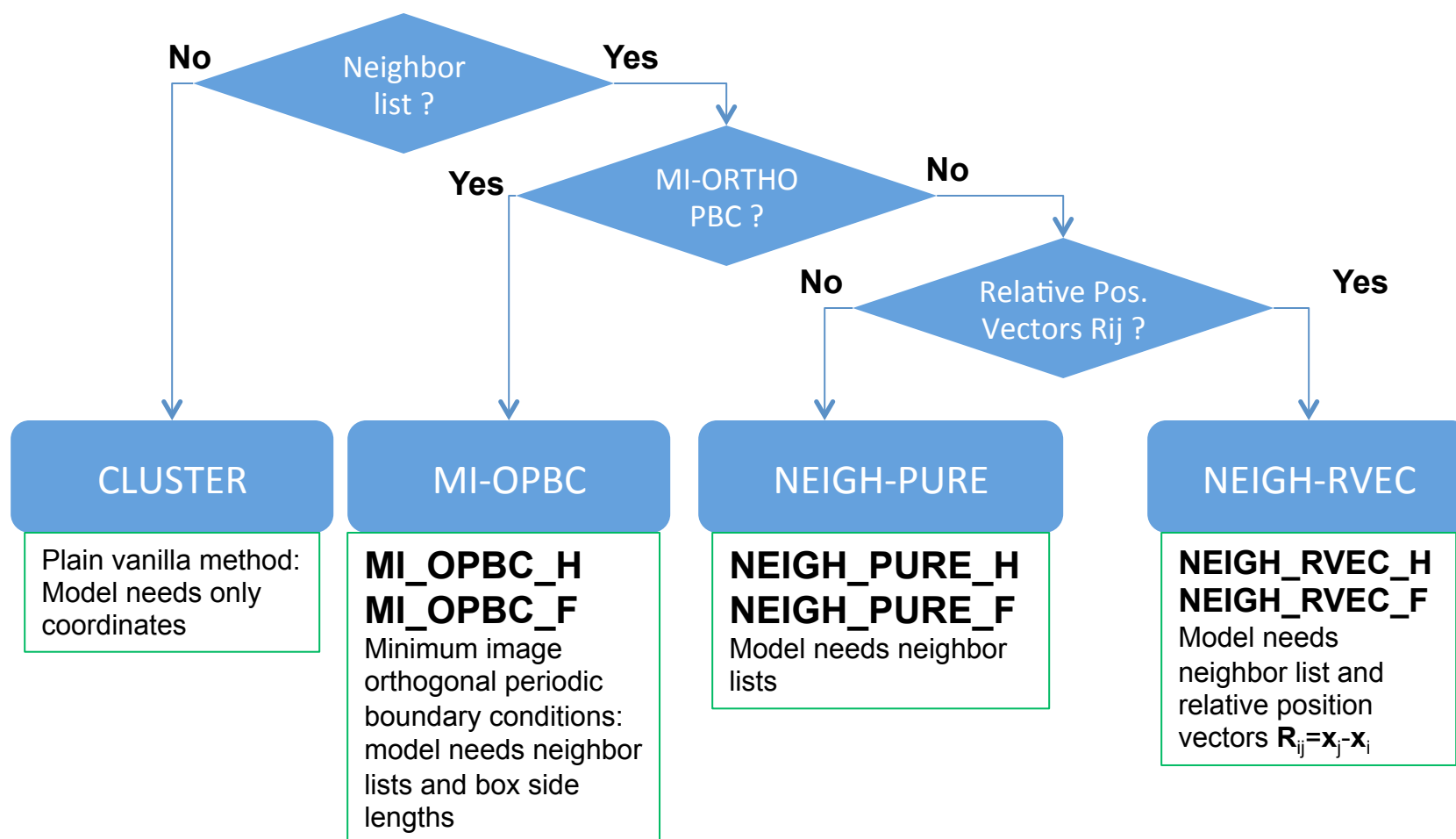
Unit_charge       := `C' | `e' | `statC'
Unit_temperature  := `K'
Unit_time         := `fs' | `ps' | `ns' | `s'
```

The list of recognized units above may be extended in the future.

There are several service routines related to units and units handling in KIM API:

**KIM\_API\_get\_unit\_handling()**, **KIM\_API\_convert\_to\_act\_unit()**, **KIM\_API\_get\_unit\_length()**, **KIM\_API\_get\_unit\_energy()**, etc...(see DOCS/KIM\_API\_Description.txt).

## Handling of Neighbor lists and Boundary Conditions – NBC methods



## Descriptions of the NBC methods

---

### **CLUSTER:**

In the CLUSTER method, the Model receives the number of particles and coordinates without additional information (such as neighbor lists or other boundary condition specifiers) and computes requested quantities under the assumption that the particles form an isolated cluster. For example, if energy and forces are requested, it will compute the total energy of all the particles based on the supplied particle coordinates and the derivative of the total energy with respect to the positions of the particles.

### **NEIGH\_PURE\_[H|F]:**

In the NEIGH\_PURE methods (NEIGH\_PURE\_H and NEIGH\_PURE\_F), the Model receives the number of particles, coordinates and a full or half neighbor list. The neighbor list defines the environment of each particle, from which the particles's energy is defined. In the case of a half list, the value of the argument ``numberContributingParticles'` indicates that the first ``numberContributingParticles'` contribute their energy to the total and the remaining particles do not contribute to the energy (they are "padding" particles). When ``numberContributingParticles'` is equal to ``numberParticles'` the half list is called "symmetric", otherwise it is called "unsymmetric." In the case of a full list, any particle that has one or more neighbors contributes its energy to the total and those particles with zero neighbors do not contribute to the total energy. The model computes the requested quantities using the supplied information. For example, if energy and forces are requested, it will compute the total energy of all the particles based on their neighbor lists and the derivative of the total energy with respect to the positions of the particles. This method can be used with codes that use padding particles to apply boundary conditions. The padding particles are treated as regular particles by the Model, and it is up to the calling code to discard some information such as the forces on the padding particles and to compute the appropriate total energy from per-particle energies of the physical particles, or to use a modified neighbor list to obtain the desired values.

#### **NEIGH\_PURE\_H:**

This is the Pure Half neighbor list method. The model needs ``coordinates'`, a half neighbor list (with data stored in the ``neighObject'` argument), the ``numberContributingParticles'`, and the ``get_neigh'` method supplied by the Test.

#### **NEIGH\_PURE\_F:**

This is the Pure Full neighbor list method. The model needs ``coordinates'`, a full neighbor list (with data stored in the ``neighObject'` argument), and the ``get_neigh'` method supplied by the Test.

## Descriptions of the NBC methods (2)

### NEIGH\_RVEC\_[H|F]:

In the NEIGH\_RVEC methods (NEIGH\_RVEC\_H and NEIGH\_RVEC\_F), the Model receives the number of particles, coordinates, a half or full neighbor list, and the relative position vectors  $r_{ij}$  ( $r_{ij} = x_j - x_i$ ). The neighbor list and  $r_{ij}$  vectors define the environment of each particle, from which the particle's energy is defined. In the case of a half list, a neighbor pair  $i$  and  $j$  (where  $i < j$ ) with relative position vector (RVEC)  $r_{ij}$  defines two pieces of information: (1)  $j$  is a neighbor of  $i$  with RVEC  $r_{ij}$  and (2)  $i$  is a neighbor of  $j$  with RVEC  $r_{ji} = -r_{ij}$ . Additionally, the value of the argument 'numberContributingParticles' indicates that the first 'numberContributingParticles' contribute their energy to the total and the remaining particles do not contribute to the energy (they are "padding" particles). When 'numberContributingParticles' is equal to 'numberParticles' the half list is called "symmetric", otherwise it is called "unsymmetric." In the case of a full list, any particle that has one or more neighbors contributes its energy to the total and those particles with zero neighbors do not contribute to the total energy. The model computes the requested quantities using the supplied information. For example, if energy and forces are requested, it will compute the total energy of all the particles based on their neighbor lists and the derivative of the total energy with respect to the positions of the particles. These methods enable the application of general periodic boundary conditions, including multiple images. A possible future extension to these methods is to allow the Test to provide a ForceTransformation() function for each neighbor, which would enable the application of complex boundary conditions such as torsion and objective boundary conditions.

### NEIGH\_RVEC\_H:

This is the Relative Vector Boundary Condition Full neighbor list method. The Model needs 'coordinates', a full neighbor list (with data stored in the 'neighObject' argument), and the 'get\_neigh' method supplied by the Test. The 'neighObject' argument must also contain the relative position vectors (RVEC) (which are returned by the 'get\_neigh' function).

### NEIGH\_RVEC\_F:

This is the Relative Vector Boundary Condition Full neighbor list method. The Model needs 'coordinates', a full neighbor list (with data stored in the 'neighObject' argument), and the 'get\_neigh' method supplied by the Test. The 'neighObject' argument must also contain the relative position vectors (RVEC) (which are returned by the 'get\_neigh' function).

## Descriptions of the NBC methods (3)

---

### **MI\_OPBC\_[H|F]:**

In the MI\_OPBC methods (MI\_OPBC\_H and MI\_OPBC\_F), the Model receives the number of particles and coordinates, the side lengths for the periodic orthogonal box and a neighbor list. It assumes all particles lie inside the periodic box. Side lengths of the box must be at least twice the cutoff range. This method computes the requested quantities under the assumption that the particles are subjected to the minimum image, orthogonal, periodic boundary conditions.

#### **MI\_OPBC\_H:**

This is the Minimum Image Orthogonal Periodic Boundary Condition Half neighbor list method. The Model needs 'coordinates', a half neighbor list (with data stored in the 'neighObject' argument), 'numberContributingParticles', the 'get\_neigh' method supplied by the Test, and the 'boxSideLengths' argument (which specifies the three side-lengths of the orthogonal simulation box).

#### **MI\_OPBC\_F:**

This is the Minimum Image Orthogonal Periodic Boundary Condition Full neighbor list method. The Model needs 'coordinates', a full neighbor list (with data stored in the 'neighObject' argument), the 'get\_neigh' method supplied by the Test, and the 'boxSideLengths' argument (which specifies the three side-lengths of the orthogonal simulation box).

## Example of using NBC methods in KIM file

### EXAMPLES/MODELS/ex\_model\_Ne\_P\_LJ/ ex\_model\_Ne\_P\_LJ.kim

```
...  
CONVENTIONS:  
# Name                               Type  
OneBasedLists                        flag  
  
Neigh_IterAccess                     flag  
  
Neigh_LocaAccess                     flag  
  
NEIGH_RVEC_H                         flag  
NEIGH_PURE_H                         flag  
NEIGH_RVEC_F                         flag  
NEIGH_PURE_F                         flag  
MI_OPBC_H                           flag  
MI_OPBC_F                           flag  
CLUSTER                             flag  
...
```

The example in ex\_model\_Ne\_P\_LJ.kim is designed to work with six different NBC methods.

If the Test can also work with multiple NBC methods and there are several matches, the first matched method listed in the Model's KIM file will have precedence.

The KIM\_API\_init () routine will check that all needed data lines for the chosen method are in the KIM descriptor file.

NBC Methods

## Neighbor list access methods: all related lines in the KIM descriptor files

**DOCS/standard.kim** (only related to Neighbor list access are shown here)

```
...
CONVENTIONS:
# Name                                Type
...
ZeroBasedLists                       flag    # presence of this line indicates that indexes
                                           # for particles are from 0 to numberOfParticles-1 (C-style)
OneBasedLists                         flag    # presence of this line indicates that indexes for
                                           # atoms are from 1 to numberOfParticles (Fortran-style)
Neigh_IterAccess                     flag    # works with iterator mode
Neigh_LocaAccess                     flag    # works with locator mode
Neigh_BothAccess                     flag    # needs both locator and iterator modes
```

```
NEIGH_PURE_H                         flag
NEIGH_PURE_F                         flag
NEIGH_RVEC_H                         flag
NEIGH_RVEC_F                         flag
MI_OPBC_H                           flag
MI_OPBC_F                           flag
```

**neighObject** stores completely encapsulated neighbor list object  
Access to the object is done through method **get\_neigh**. The  
neighbor list object and the method to access it are supplied by  
the Test.

```
MODEL_INPUT:
# Name                                Type      Unit      Shape      requirements
get_neigh                             method    none      []
neighObject                           pointer   none      []
boxSideLengths                        double    length    [3]
```

## Interface to get\_neigh method

get\_neigh function for access to the neighbor list object

here :

mode - operate in iterator or locator mode  
 mode = 0 : iterator mode  
 mode = 1 : locator mode

request - Requested operation

If mode = 0

request = 0 : reset iterator

request = 1 : increment iterator

If mode = 1

request = # : number of the particle whose neighbor list is requested

particle - the number of the particle whose neighbor list is returned

numnei - number of neighbors returned

nei1particle - integer array of neighbors of an particle which will point to the list of neighbors on exit.

rij - array of relative position vectors of the neighbors of a particle (including boundary conditions if applied) if they have been computed (NBC scenario NEIGH\_RVEC\_[H|F] only). Has NULL value otherwise (all other NBC scenarios).

```
integer(c_int) function get_neigh(pkim,mode,request,atom,numnei,pnei1atom,pRij) bind(c)
  use, intrinsic :: iso_c_binding
  implicit none
```

**FORTTRAN style**

!-- Transferred variables

type(c\_ptr), intent(in) :: pkim

integer(c\_int), intent(in) :: mode

integer(c\_int), intent(in) :: request

integer(c\_int), intent(out) :: atom

integer(c\_int), intent(out) :: numnei

type(c\_ptr), intent(out) :: pnei1atom

type(c\_ptr), intent(out) :: pRij

end function get\_neigh

```
int get_neigh(void ** pkim, int * mode, int * request, int * particle,
              int * numnei, int ** pnei1particle, double ** prij) ;
```

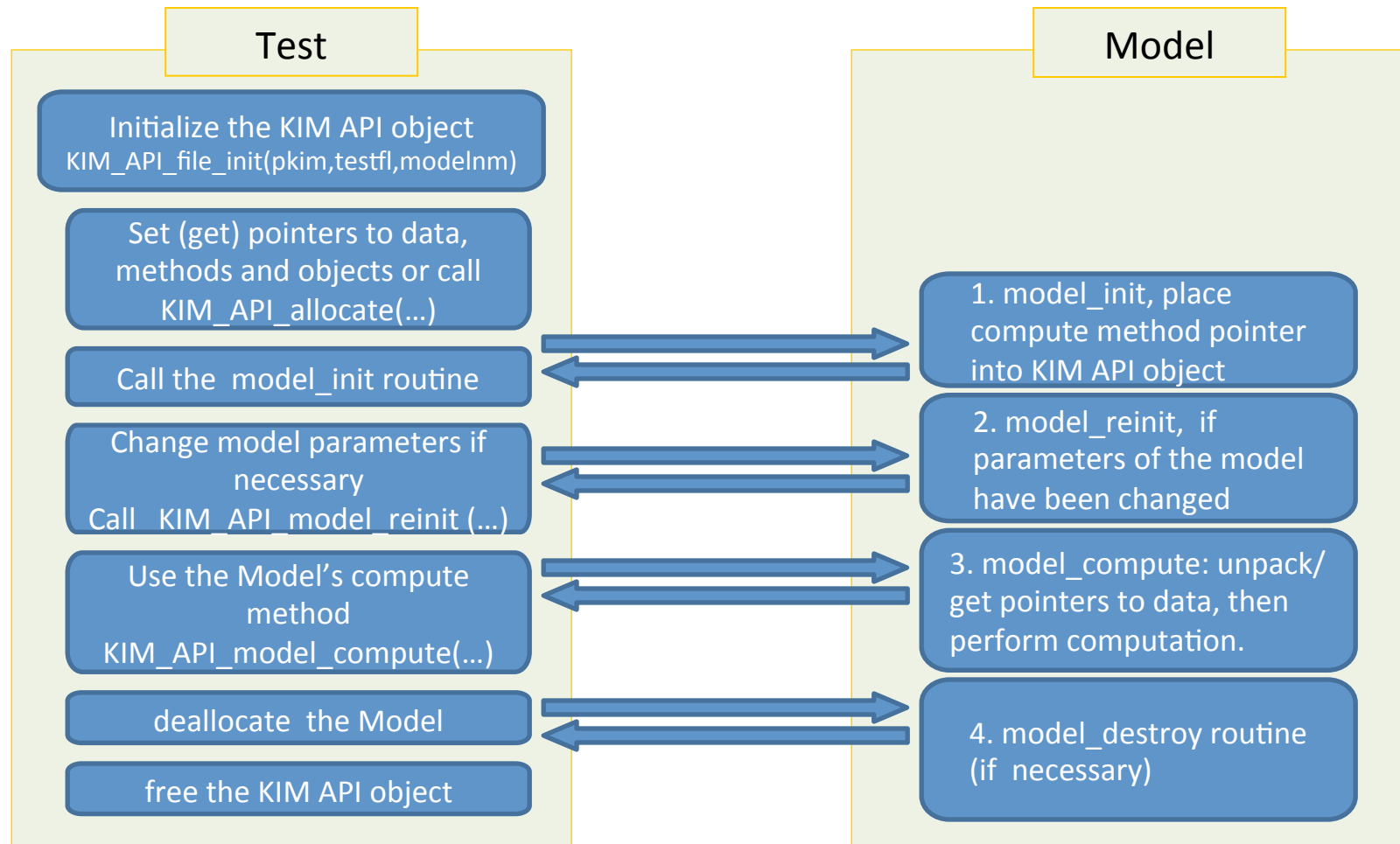
**C style**

The return value depends on the results of execution.  
 (see DOCs/KIM\_API\_Description.txt for details)

Test must supply the get\_neigh method and store a pointer to it in the KIM API object



## Model\_init places compute method pointer in KIM API object



Pointer to KIM API object is the main argument communicated between **Tests** and **Models**

## Initialization of KIM API object, setting and getting data-pointers can be done through the KIM service routines

### KIM\_API/KIM\_API\_C.h

```
#include <stdint.h>
#ifdef __cplusplus
extern "C" {
#endif
//global methods

int KIM_API_file_init(void * kimmdl, char * testkimfile, char * mdlname);

void KIM_API_allocate(void * kimmdl, intptr_t natoms, int ntypes);

void KIM_API_free(void * kimmdl, int * kimerror);

void KIM_API_print(void * kimmdl, int * kimerror);

int KIM_API_model_compute(void * kimmdl);
...
//element access methods
int KIM_API_set_data(void * kimmdl, char * nm, intptr_t size, void * dt);
void * KIM_API_get_data(void * kimmdl, char * nm, int * kimerror);
...
//multiple data set/get methods
//
void KIM_API_setm_data(void * kimmdl, int * error, int numargs, ... );
void KIM_API_getm_data(void * kimmdl, int * error, int numargs, ... );
```

Initialization is done by analyzing test and model descriptor files

One can use optional KIM service routine to allocate standard arguments and data

Call model\_compute routine by address stored in KIM API object

Directly place data pointer into the KIM API object

"Multiple" version of set/get data

Description of all KIM API service routines are located in the file:  
**DOCS/KIM\_API\_Description.txt**

## 8.1 Examples of using KIM\_API\_init and KIM\_API\_allocate service routines

.../ex\_test\_Ar\_free\_cluster\_CLUSTER\_F03/ex\_test\_Ar\_free\_cluster\_CLUSTER\_F03.F03

```
...
! Initialize the KIM object
ier = kim_api_file_init(pkim, testkimfile, modelname)
if (ier.lt.KIM_STATUS_OK) then
    idum = kim_api_report_error(__LINE__, THIS_FILE_NAME,
                               "kim_api_file_init", ier)

    stop
endif
! Allocate memory via the KIM system
call kim_api_allocate(pkim, N, ATypes, ier)
if (ier.lt.KIM_STATUS_OK) then
    idum = kim_api_report_error(__LINE__, THIS_FILE_NAME,
                               "kim_api_allocate", ier)

    stop
endif
...
```

KIM API init will check the consistency of KIM descriptor file (Test and Model) against standard.kim, after that will check if Test and Model match: NBC methods, particle species (if any), conventions and argument data lines

If the match is successful, then the KIM API object is created. This object conforms to the Model descriptor KIM file and can store all described data as pointers

.../ex\_test\_Ar\_multiple\_models/ex\_test\_Ar\_multiple\_models.c

```
...
status = KIM_API_file_init(&pkim_periodic_model_0, testkimfile, modelname0);
if (KIM_STATUS_OK > status){
    KIM_API_report_error(__LINE__, __FILE__, "KIM_API_file_init()", status);
    ...
}
```

KIM\_API\_allocate will allocate memory for all arguments with, fully specified shape, stored in the KIM API object

It is not mandatory to use KIM\_API\_allocate. A Test can use its own memory and set address of the data in the KIM API object.

## Examples of using KIM API getm/setm data ("multiple" version of get/set data)

.../ex\_test\_Ar\_free\_cluster\_CLUSTER\_F03/ex\_test\_Ar\_free\_cluster\_CLUSTER\_F03.F03

```
...
! Unpack data from KIM object
call kim_api_getm_data(pkim, ier, &
    "numberOfParticles",    pnAtoms,          1, &
    "numberParticleTypes",  pnparticleTypes,   1, &
    "particleTypes",        pparticleTypesdum, 1, &
    "coordinates",          pcoor,             1, &
    "cutoff",               pcutoff,           1, &
    "energy",               penergy,           1, &
    "virial",               pvirialglob,       1, &
    "forces",               pforces,           1)
if (ier.lt.KIM_STATUS_OK) then
    idum = kim_api_report_error(__LINE__, THIS_FILE_NAME, &
        "kim_api_getm_data", ier)
    goto 42
endif
...
```

KIM\_API\_getm\_data (or kim\_api\_getm\_data) will return pointers stored in KIM\_API object. "Multiple" version of get data routines allows to get several variable pointers from the KIM API object s at once.

KIM\_API\_setm\_data (or kim\_api\_setm\_data) allows to place (pack) several data pointers into KIM API objects See DOCs/KIM\_API\_Description.txt for the details

.../ex\_test\_Ar\_multiple\_models/ex\_test\_Ar\_multiple\_models.c

```
...
/* Register memory */
KIM_API_setm_data(pkim_periodic_model_0, &status, 8*4,
    "numberOfParticles",    1,    &numberOfParticles_periodic,    1,
    "numberParticleTypes",  1,    &numberParticleTypes,           1,
    ...
    "energy",               1,    &energy_periodic_model_0,       1);
if (KIM_STATUS_OK > status) KIM_API_report_error(__LINE__,
__FILE__, "KIM_API_setm_data", status);...
```

## 8.3 KIM\_API\_model\_init will call model initialize routine that, in turn, will place model compute into KIM object

.../ex\_test\_Ar\_multiple\_models/ex\_test\_Ar\_multiple\_models.c

```
...
/* call model init routines */
status = KIM_API_model_init(pkim_periodic_model_0);
if (KIM_STATUS_OK > status)
KIM_API_report_error(__LINE__, __FILE__, "KIM_API_model_i
...
/* call compute functions */
status = KIM_API_model_compute(pkim_periodic_model_0);
if (KIM_STATUS_OK > status)
KIM_API_report_error(__LINE__, __FILE__, "compute",
status);
...
```

KIM\_API\_model\_init will call the model\_init routine . The name of this routine is specified in the Model's Makefile

KIM\_API\_model\_compute calls the address of the model compute subroutine stored in KIM API object.

By the time KIM\_API\_model\_compute is called, the address is placed in KIM API object by model\_init\_ routine

.../ex\_model\_Ar\_P\_MLJ\_C/ex\_model\_Ar\_P\_MLJ\_C.c

```
/* store pointer to compute function in KIM object */
ier = KIM_API_set_method(pkim, "compute", 1, (void*) &compute);
if (KIM_STATUS_OK > ier){
    KIM_API_report_error(__LINE__, __FILE__, "KIM_API_set_method", ier);
    exit(1);
}
...
```

Place address of actual compute routine into the KIM API object

## An example of using get\_neigh method through KIM API service routines

.../ex\_model\_Ar\_P\_MLJ\_NEIGH\_PURE\_H\_F/ex\_model\_Ar\_P\_MLJ\_NEIGH\_PURE\_H\_F.F03

```
...
do i = 1,numberOfParticles
  ! Get neighbors for atom i
  !
  atom = i ! request neighbors for atom i
  ier= kim_api_get_neigh(pkim,1,atom,atom_ret,numnei,pneilatomb,&
                        pRij_dummy)
  if (ier.lt.KIM_STATUS_OK) then
    idum = kim_api_report_error(__LINE__, THIS_FILE_NAME,&
                                "kim_api_get_neigh", ier)

    return
  endif

  ! Loop over the neighbors of atom i
  !
  do jj = 1, numnei
    j = pneilatomb(jj)
    Rij(:) = coor(:,j) - coor(:,i) ! distance vector between i j
    Rsqij = dot_product(Rij,Rij)    ! compute square distance
    if ( Rsqij < model_cutsq ) then ! particles are interacting?
      r = sqrt(Rsqij)              ! compute distance
      call pair(model_epsilon,model_sigma,model_A,model_B,&
                model_C, r,phi,dphi,d2phi) ! compute pair potential
    endif
  enddo
enddo
...
```

Locator mode -- get neighbors of a particle using half or full neighbor lists as requested.

KIM\_API\_get\_neigh will call the method using the address stored in the KIM API object. These methods are supplied by the Test.

KIM\_API\_get\_neigh will check if the arguments are set correctly. It will also convert the result from OneBaseLists to ZeroBaseLists (or vice versa) if necessary .

Details on the interface and a description of error codes are in **DOCS/KIM\_API\_Description.txt**

## Computing quantities from the first derivative

### MODELS/ ex\_model\_Ne\_P\_fastLJ/ex\_model\_Ne\_P\_fastLJ.c

```
...
while (KIM_STATUS_OK == *ier)
{
    i = currentAtom + model_index_shift;;
    zi=i*DIM;

    /* loop over the neighbors of currentAtom */
    for (jj = 0; jj < numOfAtomNeigh; ++ jj)
    {
        ...

        /* process dEdr */
        if (comp_process_dEdr)
        {
            R = sqrt(Rsqij);
            double DE = fac*R;
            ier = KIM_API_process_dEdr(km, &DE, &R, &pdx, &i, &j);
        }
        ...
    }
}
...
```

This routine can be called by a Model to provide the Test with a contribution, dEdr, to the first derivative of the Model's energy with respect to the (scalar) distance  $r_{ij}$  between particle 'i' and particle 'j'. The Test can use this information to compute, via the chain-rule, many properties. Examples include forces, the virial, and other thermodynamic tensions. The KIM API performs automatic index conversion (based on ZeroBasedList and OneBasedList flag settings) before calling the Test's supplied process\_dEdr function. If the Test does not provide its own process\_dEdr routine, then the KIM API standard process\_dEdr routine is used. If the standard process\_dEdr routine is used, the KIM API ensures that any appropriate memory initializations are performed. This routine and currently supports the computation of 'virial' and 'particleVirial'.

void \*\*km -- pointer to KIM\_API\_model object

double \*dE -- pointer to the contribution to the first derivative of the energy with respect to the pair-distance  $r_{ij}$

double \*r -- pointer to  $r_{ij}$  -- the distance between particles i and j

int \*I,j -- pointers to particle index I and j.

double \*\*pdx -- pointer to the relative position vector of particle j relative to particle i (i.e.,  $r_{ij} = x_j - x_i$ ).

On details of interface using process\_dEdr see documentation in KIM\_API\_Description.txt and standard.kim

---

## Appendix



## Every argument that needs to be communicated between Tests and Models must be in the descriptor file

---

Each **Test** has its own descriptor file that describes the data it can supply to the **Model** and what data it expects the **Model** to compute. There are no optional arguments in a **Test**'s descriptor file (because the Test knows, a priori, what it will need to compute).

Each **Model** has its own descriptor file that describes the data it needs to perform its computations and what results it can compute. Some of the arguments/methods can be identified as optional. Optional arguments/methods are ones that the **Test** does not have to provide or are results that the **Model** will only compute if the **Test** explicitly requests it.

KIM service routines (such as KIM\_API\_file\_init) use both **Test** and **Model** descriptor files to:

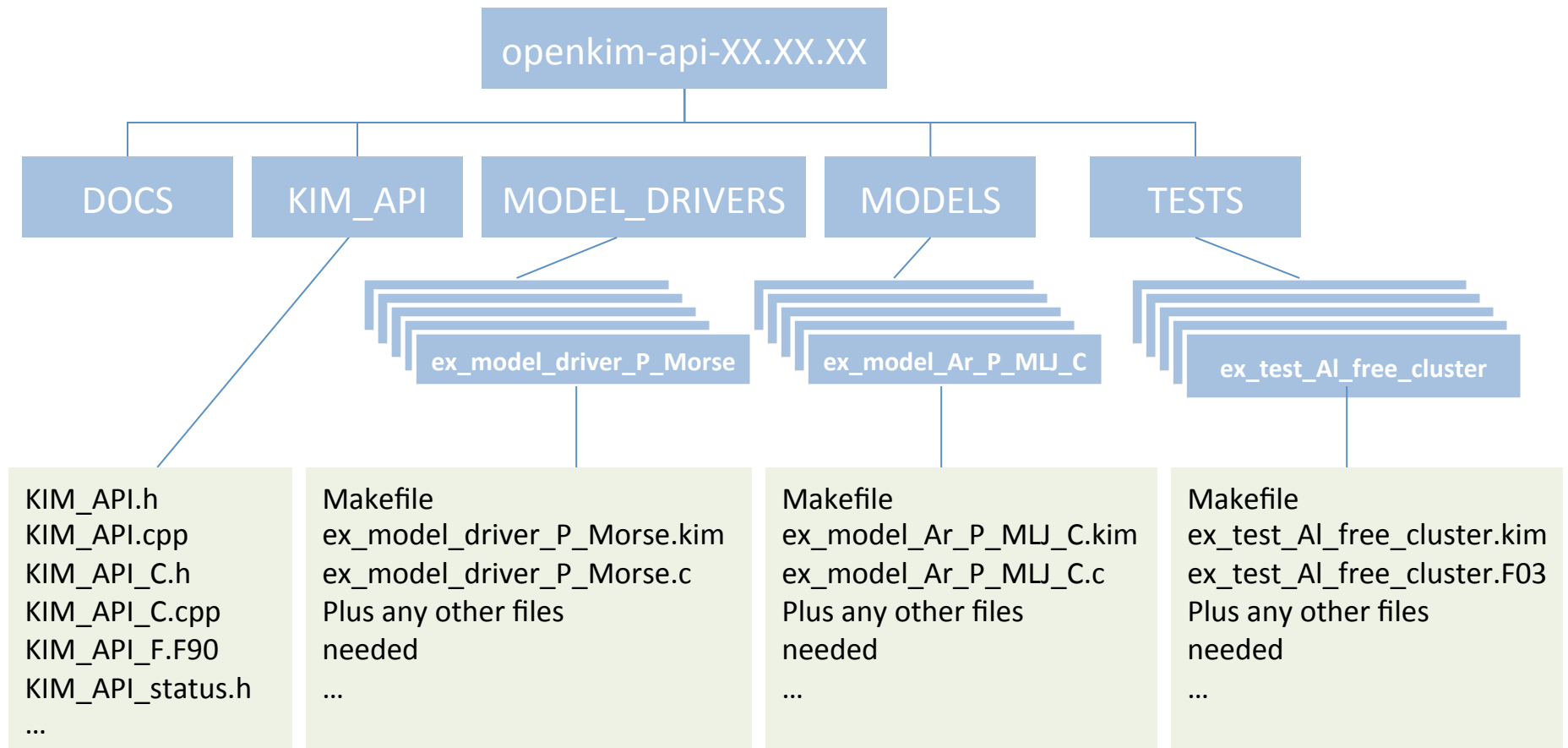
- Check if the **Model** and **Test** match, also check if their descriptor files conform to the KIM API standard
- If they do -- create a KIM API object to store all arguments described in the **Model**'s descriptor file
- Mark each optional argument that is not used by the **Test** “do not compute” (i.e., compute = false)

The flag here is an integer: KIM\_COMPUTE\_TRUE – compute, KIM\_COMPUTE\_FALSE – do not compute

Other service routines are used to:

- Set (get) argument or method pointers into (from) the KIM API object (e.g., KIM\_API\_set\_data, KIM\_API\_get\_data, etc.)
- Check if the “compute flag” is set to “compute” for an argument in the KIM API object (KIM\_API\_get\_compute).
- Execute the Model's compute method (KIM\_API\_model\_compute)
- etc...

# KIM API directory structure



Each **Test** and **Model** has its own descriptor file

---

**The end**