KIM API alpha version

(Knowledgebase of Interatomic Models Application Program Interface alpha version)

Prepared by: Valeriu Smirichinski, Ryan S. Elliott and Ellad Tadmor

June , 2011

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 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.

Content

KIM API concept and implementation:

1. The KIM repository contains **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. Structure of descriptor file and type of the variable in the 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 KIM API object, setting and getting data-pointers can be done through the KIM service routines
9. KIM installation, compilation, linking and running tests

Content (2)

Appendix

1. Every variable that needs to be communicated between **Tests** and **Models** must be in the descriptor file

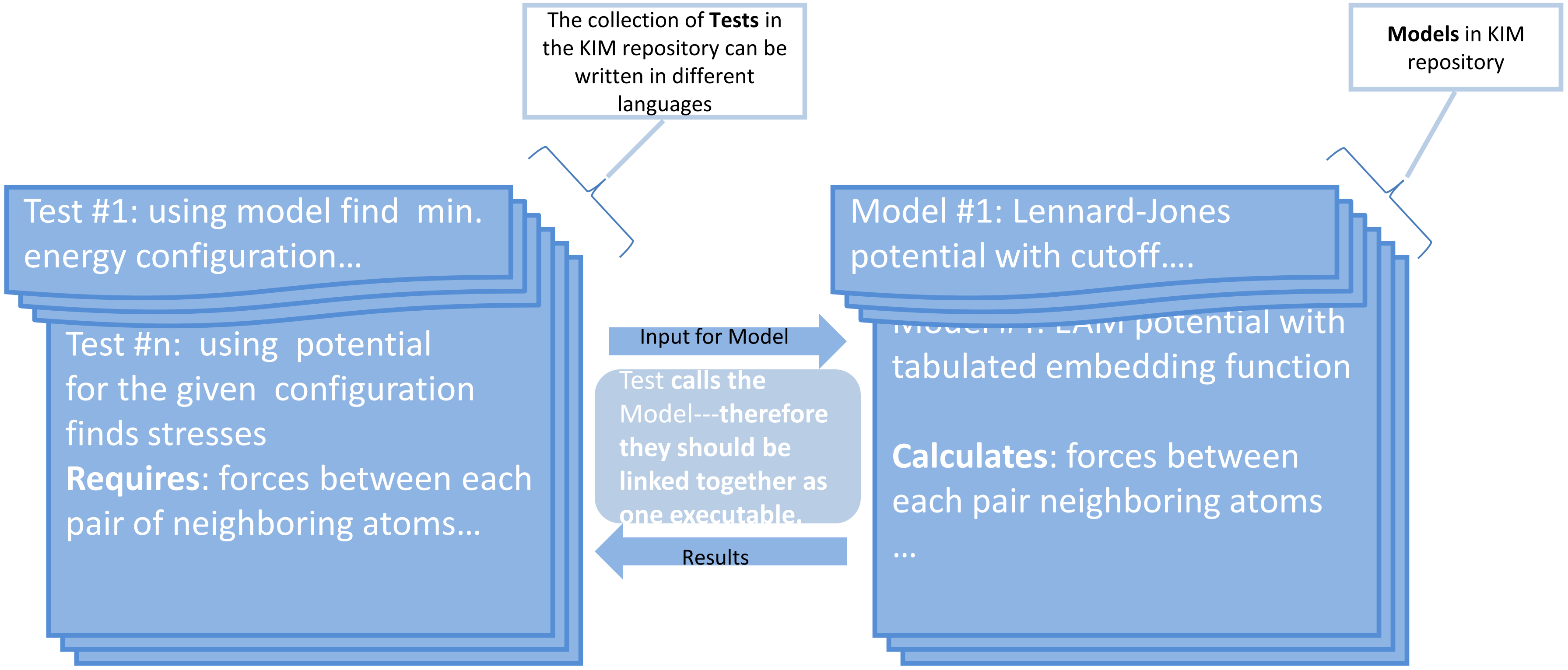
2. The KIM API directory structure

3. Number of models and test examples available in the current version of KIM API

4. The KIM API object is an array of Base data elements.
   Each Base data element can hold a pointer to any relevant data (scalar, array, method, etc.)

KIM API concept

The KIM repository contains Models and Tests



The collection of **Tests** in the KIM repository can be written in different languages

Models in KIM repository

Test #1: using model find min. energy configuration…

Test #n: using potential for the given configuration finds stresses
Requires: forces between each pair of neighboring atoms…

Input for Model

Test **calls the** Model---**therefore they should be linked together as one executable.**

Results

Model #1: Lennard-Jones potential with cutoff….

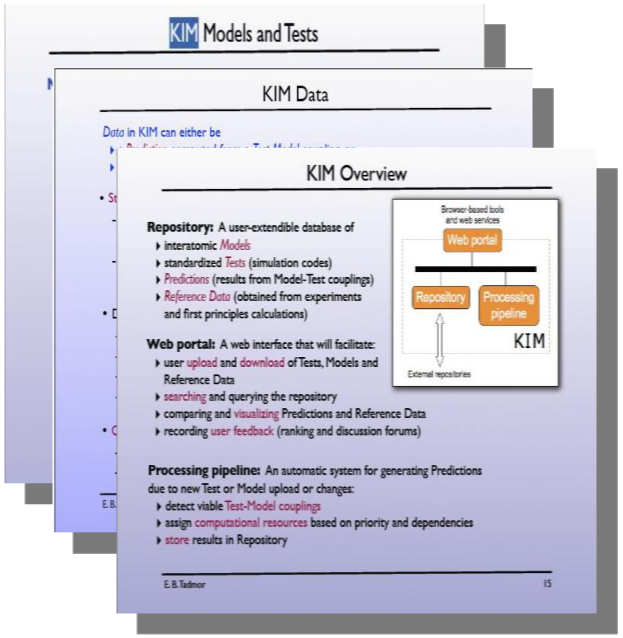
Model # ?: EAM potential with tabulated embedding function

Calculates: forces between each pair neighboring atoms
…

Users and developers will be able to download **Tests** and **Models** , then compile, link and run the resulting programs to produce new results.

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

KIM framework



KIM Models and Tests

KIM Data

Data in KIM can either be

KIM Overview

Repository: 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)

Web portal: 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)

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

E.B.Tadmor 15

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

Requirements:

•Multilanguage support (C, C++, FORTRAN 90, Python …)

• A variety of data structure 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

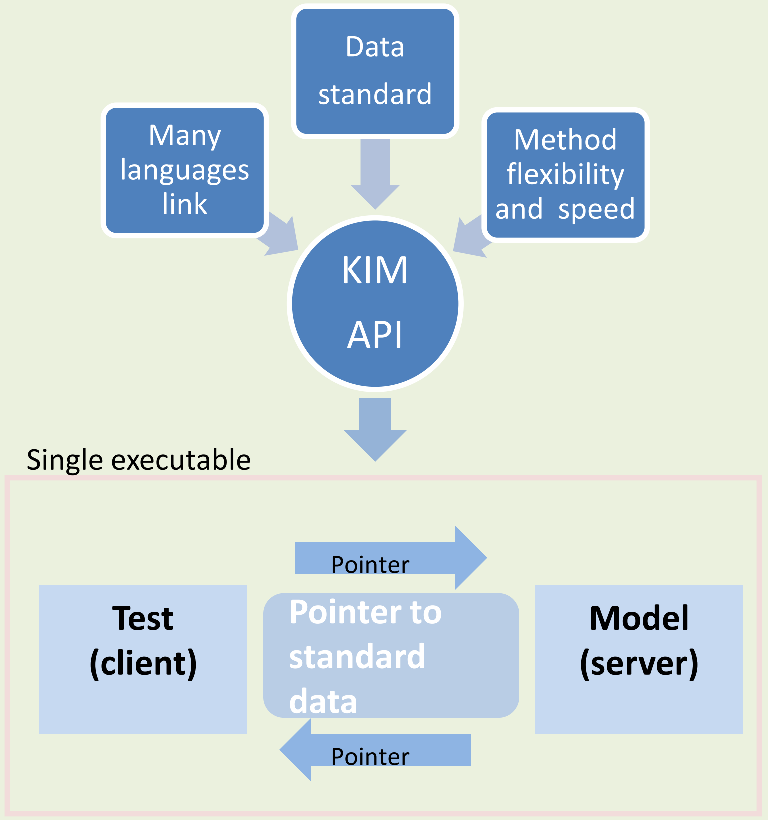
Need a simple interface : ideally just one argument per call

Source: KIM kickoff presentation

3

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

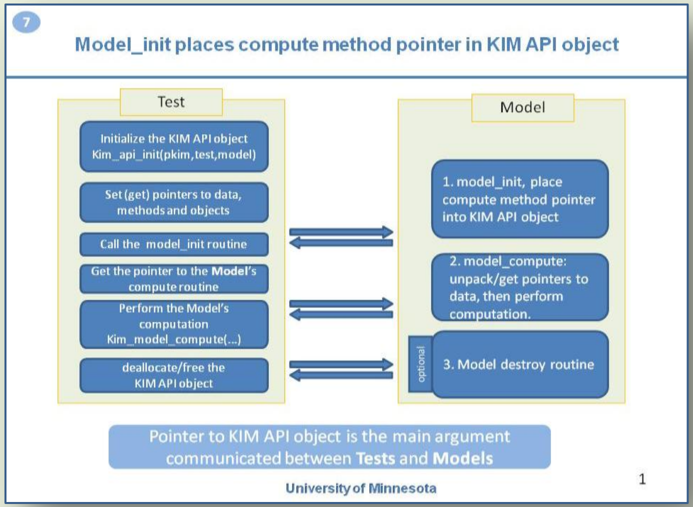
Concept



Data standard should accommodate every possible data set required for the model

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


7
Model_init places compute method pointer in KIM API object
Test
Initialize the KIM API object Kim_api_init(pkim,test,model)
Set (get) pointers to data, methods and objects
Call the model_init routine
Get the pointer to the Model's compute routine
Perform the Model's computation Kim_model_compute(...)
deallocate/free the KIM API object
Model
1. model_init, place compute method pointer into KIM API object
2. model_compute: unpack/get pointers to data, then perform computation.
optional
3. Model destroy routine
Pointer to KIM API object is the main argument communicated between Tests and Models
University of Minnesota
1

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

Sample_01_lj_cutoff_f77.kim

```
MODEL_NAME        := Sample_01_lj_cutoff_f77
SystemOfUnitsFix := fixed    #can work only with units system defined bellow
…
MODEL_INPUT:
# Name            Type          Unit       SystemU/Scale       Shape               requirements
numberOfAtoms     integer*8     none       none                []

coordinates       real*8        length     standard            [numberOfAtoms,3]

compute           method        none       none                []

neighObject       pointer       none       none                []
…

MODEL_OUTPUT:
# Name            Type          Unit       SystemU/Scale       Shape               requirements
energy            real*8        energy     standard            []

forces            real*8        force      standard            [numberOfAtoms,3]
…
```

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

Tests and models expose their required input/output variables that will be communicated through KIM API

Note: full .kim file shown here can be found in MODELs/Sample_01_cutoff_f77/

Structure of descriptor file and type of the variable in the descriptor file

Sections lines

```
SUPPORTED_ATOM/PARTICLES_TYPES

CONVENTIONS

MODEL_INPUT

MODEL_OUTPUT

MODEL_PARAMETERS
```

Data lines

```
* Species Data lines

* Dummy Data lines

* Argument Data lines
```

Model/test name and system of units lines

```
MODEL_NAME:=Sample_01_lj_cutoff_c

SystemOfUnitsFix := fixed
```

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 above 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 to define atomic species by providing symbol and integer code. Those lines are located in section SUPPORTED_ATOM/PARTICLES_TYPES.
* Dummy 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.

Each argument line in the descriptor file describes a variable and its properties

Sample_01_lj_cutoff_f77.kim

```
MODEL_NAME        := Sample_01_lj_cutoff_f77
SystemOfUnitsFix  := fixed    #can work only with units system defined bellow
….
compute              method     none       none                    []

MODEL_OUTPUT:
# Name               Type       Unit       SystemU/Scale          Shape               requirements
energy               real*8     energy     standard                []

energyPerAtom        real*8     energy     standard                [numberOfAtoms]    optional
….
```

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

Method means a subroutine or function pointer

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

Type of data in computer representation

Physical dimensions:

System of units: standard, SI, none

The shape of a variable describes its array properties . It specifies the number and size (range) of indices. For example, [] means a scalar (zero-dimensional array), [NumberOfAtoms] means a one-dimensional array and [N, 3] means a two-dimensional array of size N x 3.

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

Note: detailed description of all Types value , Unit, SystemU/Scale can be found in KIM_API/standard.kim file

Specifying atom types – species data lines

sample_model.kim

```
…
SUPPORTED_ATOM/PARTICLES_TYPES:
#
# The list of standard KIM atom/particle types
# The code listed in 'standard.kim' is the atomic
#number. However, this value is ignored and only
#the code value provided by the Model is retained.
#
# Symbol/name     Type     code
#
H                 spec     1      # Hydrogen
He                spec     2      # Helium
Li                spec     3      # Lithium
…
user01            spec     201    # user defined
user02            spec     202    # user defined
…
user20            spec     220    # user defined
```

Species data line type defines 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. The line consists of three (3) white-space separated (case sensitive) strings The three strings, in order, are as follows:

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

2) Type: This must be `spec'.

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

KIM_API_get_listAtomsTypes service routine allows to obtain list of all atom species used by the model during runtime.
Also KIM_API_get_atypeCode service routine allows to get the atom species integer code (see KIMserviceDescription.txt)
Example of using the service routines is in TESTs/Sample_01_lj_cutoff_NEIGH_PURE_H_f/

In order to define "conventions" of test models behavior, dummy data lines are reserved

sample_model.kim

```
CONVENTIONS:
# Name               Type

ZeroBasedLists       dummy #indexes for atoms are
                           #from 0 to numberOfAtoms-1
                           #(C-style)
…
CLUSTER              dummy # The Model needs only
                           #coordinates to compute
                           #no neighbor list needed
…
```

This line type defines a convention (or parameter), 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 (2) white-space separated (case sensitive) strings. The two strings, in order, are as follows:

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

2) Type: This must be `dummy'

KIM_API_allocate has no effect on "dummy " type variables, because they are not "data pointer holders".

For detailed description of all dummy lines see KIM_API/standard.kim file, also sample examples in MODELs and TESTs show how they are used.

5.4

Parameter variables are used to publish/access internal parameters of a model

sample_model.kim

```
MODEL_PARAMETERS:

# Name                    Type        Unit       SystemU/Scale          Shape     requirements

PARAM_FIXED_Sigma       real*8       length      standard                  []

PARAM_FREE_Epsilon      real*8       energy      standard                  []
 …
```

Parameter variable format in KIM descriptor file is the same as for argument data type

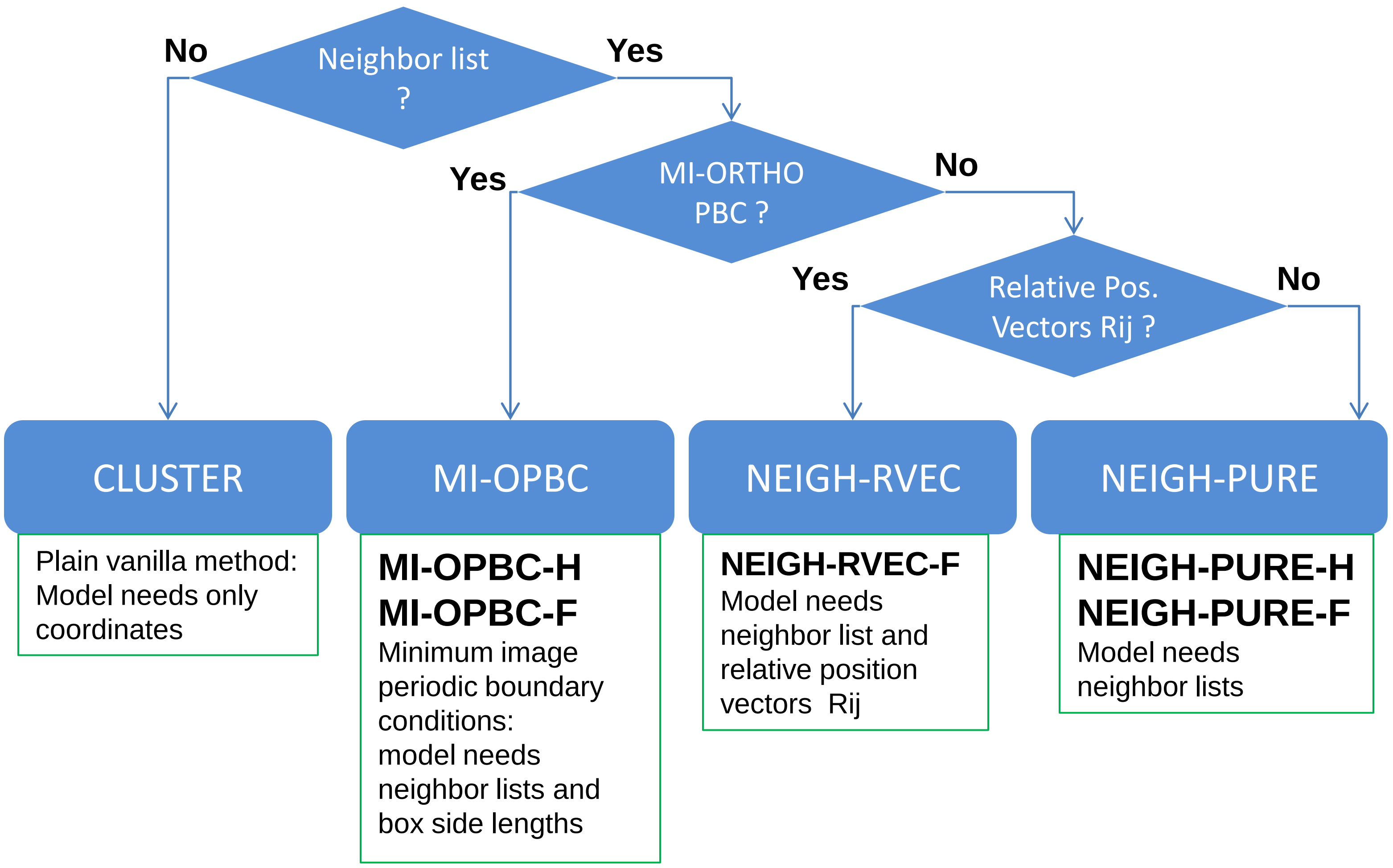
```
Two types of model parameters are allowed
 1) PARAM_FIXED_XXXXXX  - these should not be changed by the Test
 2) PARAM_FREE_XXXXXX   - these may be changed by the Test (which should then call the
    Model's_reinit() function to inform the model that its parameters have changed)


KIM_API_get_listParams() service routine will return list of all parameters in the object during
the runtime (as an array of text string
KIM_API_get_listFreeParams() service routines will return list of FREE parameters and
KIM_API_get_listFreeParams() will return list of FIXED parameters (see KIMserviceDescription.txt)
```

An example of using these service routines is in TESTs/Sample_01_lj_cutoff_NEIGH_PURE_H_f/

Names of parameter variables are not checked against standard.kim

6

Handling of Neighbor lists and Boundary Conditions – NBC methods



CLUSTER
Plain vanilla method: Model needs only coordinates

MI-OPBC
MI-OPBC-H
MI-OPBC-F
Minimum image periodic boundary conditions: model needs neighbor lists and box side lengths

NEIGH-RVEC
NEIGH-RVEC-F
Model needs neighbor list and relative position vectors Rij

NEIGH-PURE
NEIGH-PURE-H
NEIGH-PURE-F
Model needs neighbor lists

Note: NBC stands for Neighbor list and Boundary Conditions

Descriptions of the NBC methods

CLUSTER:
Receives a list of atoms and coordinates without additional information, such as neighbor lists or other boundary condition specifiers, and computes requested quantities under the assumption that the atoms form an isolated cluster. For example, if energy and forces are requested, it will compute the total energy of all the atoms based on the supplied atom coordinates and the derivative of the total energy with respect to the positions of the atoms.

MI-OPBC-[F|H]:
Receives a list of atoms and coordinates, the side lengths for the periodic orthogonal box and a neighbor list as detailed below. Assumes all atoms lie inside the periodic box. Side lengths of box must be at least twice the neighbor list range. Computes the requested quantities under the assumption that the atom s are subjected to minimum image , orthogonal, periodic boundary conditions.

Neighbor list requirements for MI-OPBC-[F|H]:

1. Minimum image conventions is applied during construction of the neighbor list consistent with the box size.
2. The neighbor list can be supplied in either full or half mode.
   Full neighbor list: All neighbors of an atom are stored
   Half neighbor list: For an atom i only the neighbors j>i are stored.

Calculated quantities for both –H and –F modes should be equivalent to those obtained were the model to compute its own neighbor list using the provided orthogonal periodic box side lengths.

Descriptions of the NBC methods (2)

NEIGH-RVEC-F:
Receives a list of atoms and coordinates, a full neighbor list and the relative position vectors Rij. The neighbor list and Rij vectors define the environment of each atom, from which the atom's energy is defined. 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 atoms based on their neighbor lists and relative position vectors and the derivative of the total energy with respect to the positions of the atoms. This method enables the application of general periodic boundary conditions, including multiple images. (This approach can fail with half neighbor lists and therefore the –H variant of the method does not exist.) A possible future extension to this method 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-PURE-[F|H]:
Receives a list of atoms and coordinates and a full or half neighbor list. The neighbor list defines the environment of each atom, from which the atom's energy is defined. 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 atoms based on their neighbor lists and the derivative of the total energy with respect to the positions of the atoms. This method can be used with codes that use ghost atoms to apply boundary conditions. The ghost atoms are treated as regular atoms by the model, and it is up to the calling code to discard some information such as the forces on the ghost atoms and to compute the appropriate total energy from per-atom energies of the physical atoms.

Example of NBC using methods in KIM file

Sample_01_compute_example_f.kim

```
…
CONVENTIONS:
# Name                    Type

OneBasedLists             dummy

NEIGH-PURE-H              dummy

NEIGH-PURE-F              dummy

NEIGH-RVEC-F              dummy

CLUSTER                   dummy
 …
```

NBC Methods

The example in TESTs/Sample_01_compute_example_f is designed to work with four different scenarios methods.

If the model also has possibility to work with different NBC methods and there are several matches, the first matched method listed in the model KIM file will have precedence.

KIM_API_init () routine will check that all needed lines for the chosen method are in KIM descriptor file.

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

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

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

MI-OPBC-H                  dummy
MI-OPBC-F                  dummy
NEIGH-RVEC-F               dummy
NEIGH-PURE-H               dummy
NEIGH-PURE-F               dummy

MODEL_INPUT:
# Name                     Type        Unit        SystemU/Scale     Shape        requirements
get_full_neigh             method      none        none                []
get_half_neigh             method      none        none                []
neighObject                pointer     none        none                []

boxlength                  real*8      length      unspecified         [3]
```

neighObject stores completely encapsulated neighbor list object Access to the object is done through methods **get_full_neigh** or **get_half_neigh** . Neighbor list object and the method to access are supplied by the test.

Interface to methods: get_half_neigh & get_full_neigh

get_half_neigh and get _full_neigh functions both have the same interface 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 atom whose neighbor list is requested

```
integer function  get_half_neigh(pkim,mode,request,atom,numnei,pnei1atom,pRij)
     implicit none
     integer(kind=kim_intptr),    intent(in)   :: pkim
     integer,                     intent(in)   :: mode
     integer,                     intent(in)   :: request
     integer,                     intent(out)  :: atom
     integer,                     intent(out)  :: numnei
     integer,                     intent(out)  :: pnei1atom
     integer,                                  :: nei1atom(1);  pointer(pnei1atom,nei1atom)
     double precision,            intent(out)  :: pRij
     double precision,                         :: Rij(3,*);     pointer(pRij,Rij)
end function  get_half_neigh
```

FORTRAN style

```
int get_half_neigh(void ** pkim, int * mode, int * request, int * atom,
                   int * numnei, int ** pnei1atom, double ** pRij) ;
```

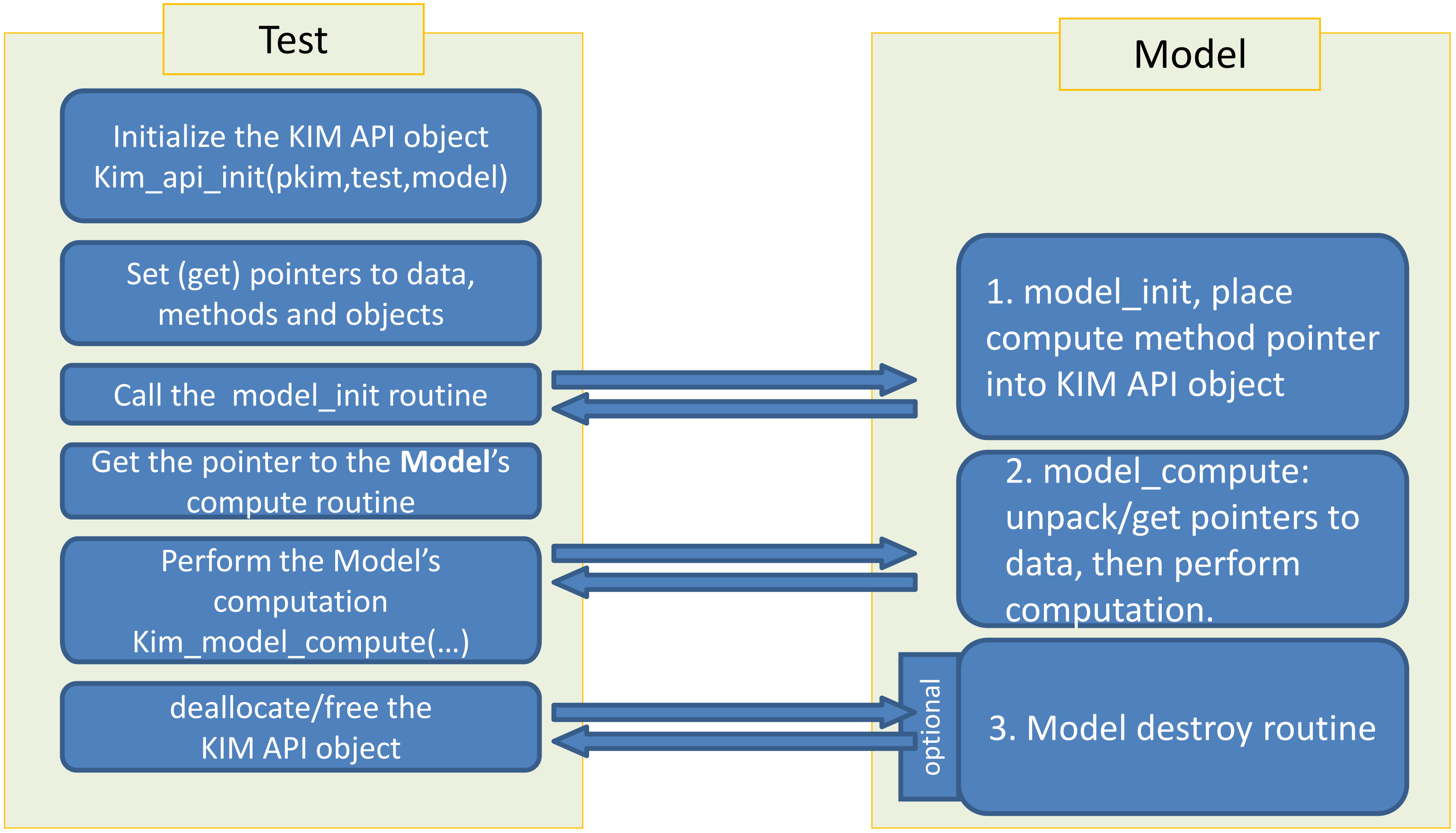
C style

atom	- the number of the atom whose neighbor list is returned
numnei	- number of neighbors returned
nei1atom	- integer array of neighbors of an atom which will point to the list of neighbors on exit.
Rij	- array of relative position vectors of the neighbors of an atom (including boundary conditions if applied) if they have been computed (NBC scenario NEIGH-RVEC-F only). Has NULL value otherwise (all other NBC scenarios).

The return value depends on the results of execution:
2 -- iterator has been successfully initialized
1 -- successful operation
0 -- iterator has been incremented past end of list
-1 -- or any negative value means unsuccessful operation (see KIM_API/KIMserviceDescription.txt)

Test must supply the get_half/full_neigh method and set it to KIM API object

Model_init places compute method pointer in KIM API object


Test
Initialize the KIM API object Kim_api_init(pkim,test,model)
Set (get) pointers to data, methods and objects
Call the model_init routine
Get the pointer to the **Model**'s compute routine
Perform the Model's computation Kim_model_compute(…)
deallocate/free the KIM API object
Model
1. model_init, place compute method pointer into KIM API object
2. model_compute: unpack/get pointers to data, then perform computation.
optional
3. Model destroy routine

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

8

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

KIMserviceC.h

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

int KIM_API_init(void * kimmdl, char * testname, 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);

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

…

//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);

…
```

Initialization is done by analyzing test and model configuration files

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

Call model compute routine by address stored in KIM API object

Directly place data pointer into the KIM API object

Description of all KIM API service routines are located in the file: **KIM_API/ KIMserviceDescription.txt**

8.1

Examples of using KIM_API_init and KIM_API_allocate service routines

TESTs/Sample_01_compute_example_f/lj_test.F90

```
...
 character*80 :: testname ="Sample_01_compute_example_f"
 character*80 :: modelname =""   !string for the model name
 integer(kind=kim_intptr) :: pkim
...
! initialize KIM_api object
 if(kim_api_init_f(pkim,testname,modelname).ne.1)&
     & stop 'not a test-model match'
...
!Allocate memory and associated it with the KIM API object
call kim_api_allocate_f(pkim,n,1,kimerr)
...
```

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, atom species (if any) , conventions and argument data lines

If match, then KIM API model object will be created. The object follows exactly model descriptor KIM file and can store all described data as pointers

TESTs/Sample_01_compute_example_c/lj_test.c

```
...
/* Initialized KIM API object */
 if (KIM_API_init(&pkim, testname ,modelname)!=1) return -1;
...
/* Allocate memory and associated it with KIM API object */
 KIM_API_allocate(pkim,n,ntypes,&kimerr);
...
```

KIM_API_allocate will allocate memory for all arrays, pointers stored in KIM API object

It is not mandatory to use KIM API allocate. Test can use its own data and set address of the data in KIM API object.

Examples of using KIM API get/set data

TESTs/Sample_01_compute_example_f/lj_test.F90

```
...
real*8::energy; pointer(penergy,energy)
real*8::cutoff; pointer(pcutoff,cutoff)
...
integer(kind=kim_intptr) :: one=1
...
penergy = kim_api_get_data_f(pkim,"energy",kimerr)
call kimerr_handle("energy",kimerr)
pcutoff = kim_api_get_data_f(pkim,"cutoff",kimerr)
call kimerr_handle("cutoff",kimerr)
...
if(kim_api_set_data_f(pkim,"neighObject",one,&
 &loc(neigh_both)).ne.1) stop' neighObjec not in kim'
...
```

KIM_API_get_data (or kim_api_get_data_f) will return address of data stored in the KIM API object.
kimerror will be equal 1 upon successful completion, otherwise it will be 0 or negative
(see **KIM_API/ KIMserviceDescription.txt)**

KIM_API_set_data (or kim_api_set_data_f) will place the address of data into KIM API object and will return integer error code : 1– success, 0 or negative – unsuccessful completion

TESTs/Sample_01_compute_example_c/lj_test.c

```
...
double * penergy;
double * penergy;
...
penergy=(double *) KIM_API_get_data(pkim,"energy",&kimerr);
...
pcutoff=(double *) KIM_API_get_data(pkim,"cutoff",&kimerr);
...
 /* Inform KIM API object about neighbor list object */
KIM_API_set_data(pkim,"neighObject",1,(void*) neighObject);
...
```

KIM_API_model init will call model initialize routine that in turn will place model compute into KIM object

TESTs/Sample_01_compute_example_f/lj_test.F90

```
...
if( kim_api_model_init(pkim).ne.1)&
 & stop ' model initialiser failed'
...
call kim_api_model_compute(pkim,kimerr)
call kimerr_handle("kim_api_model_compute",kimerr)
...
```

KIM API model init will call the model init routine . KIM_API_model_init utilizes the KIM standard naming convention in order to make the call. Name of the model init routine must have all lower case letters in the following format modelname_init_
in the following example we have:

$\underbrace{\texttt{sample\_01\_lj\_cutoff\_c}}_{\text{model name}}\texttt{\_init\_}$

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

MODELs/Sample_01_compute_example_f/lj_test.F90

```
...
/* Model Initiation routine */
void sample_01_lj_cutoff_c_init_(void * km){
 /* cast pointer for KIM API object */
 intptr_t * pkim = * ((intptr_t **)km);

 /* Provide KIM API object with function pointer*/
 /*      of compute routine */
  KIM_API_set_data(pkim,"compute",1,(void*)
                  &sample_01_lj_cutoff_c_calculate);
}
...
```

Place address of actual compute routine into the KIM API object

8.4

An example of using get_half_neigh methods through KIM API service routines

MODELs/ Sample_01_lj_cutoff_NEIGH_PURE_H_f / LJ_mod.F90

```
...
mode=0;   ! iterator mode
request=0;!reset neighbor iterator to beginning
retcode = kim_api_get_half_neigh(pkim,mode,request,&
                         &atom,numnei,pnei1atom,pRij)
...
retcode=1
do while (retcode .eq. 1)
   !increment iterator
    mode=0; request=1;
    retcode = kim_api_get_half_neigh(pkim,mode,request,&
                             &atom,numnei,pnei1atom,pRij)
    ...
    i=atom
    xi = x(:,i)
    do jj=1, numnei
       j=nei1atom(jj)
       xj = x(:,j)
       dx = xi-xj
...
```

Iterator mode -- reset iterator

Iterator mode -- increment iterator

KIM_API_get_half_neigh will call the method by address stored in KIM API object ("get_half_neigh") and supplied by test.

It will check if mode and request are set correctly, also will convert the result from oneBaseLists to zeroBaseLists (or vice versa) if necessary .

Details on interface and description of error codes are in **KIM_API/ KIMserviceDescription.txt**

KIM installation, compilation, linking and running tests

Instructions for installing, compiling and linking KIM:

1. In the desired directory, execute the command: ‘tar xzvf openkim-api-XX.XX.XX.tgz’

1. Set up environment variable (bash):
   > export KIM_DIR = /your_loacation/openkim-api/
   your_location is the correct path where the openkim-api dirrectory is located.
   (Make sure to include the trailing slash)

3. By default, all make files use the GNU compilers for 64 bit linux.
   In order to use the Intel compiler, define the environment variable KIM_INTEL
   bash:
   > export KIM_INTEL="yes".
   For using a 32 bit machine, define the environment variable KIM_SYSTEM32
   bash:
   > export KIM_SYSTEM32="yes"

4. change to the KIM_DIR directory and execute the commands:
   ‘make clean’
   ‘make’
   This will compile the KIM API and example **Tests** and **Models**.

5. To run a Test, change to the appropriate directory and run the executable. For example:
   > cd TESTs/Sample_01_compute_example_f
   > echo "Sample_01_lj_cutoff_f" | ./Sample_01_compute_example_f
   Each test sample (example Sample_01_compute_example_f and Sample_01_compute_example_c) reads the name of the model from its input stream, initiates the model and computes the total energy and forces using the coordinates of a configuration of atoms.

Note: refer to openkim-api/README for details on environment variables and other features

Appendix

A1

Every variable 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 variables in a **Test**'s descriptor file ("the test knows, a priori, what 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 variables/methods can be identified as optional. Optional variables/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_init_) use both **Test** and **Model** descriptor files to:

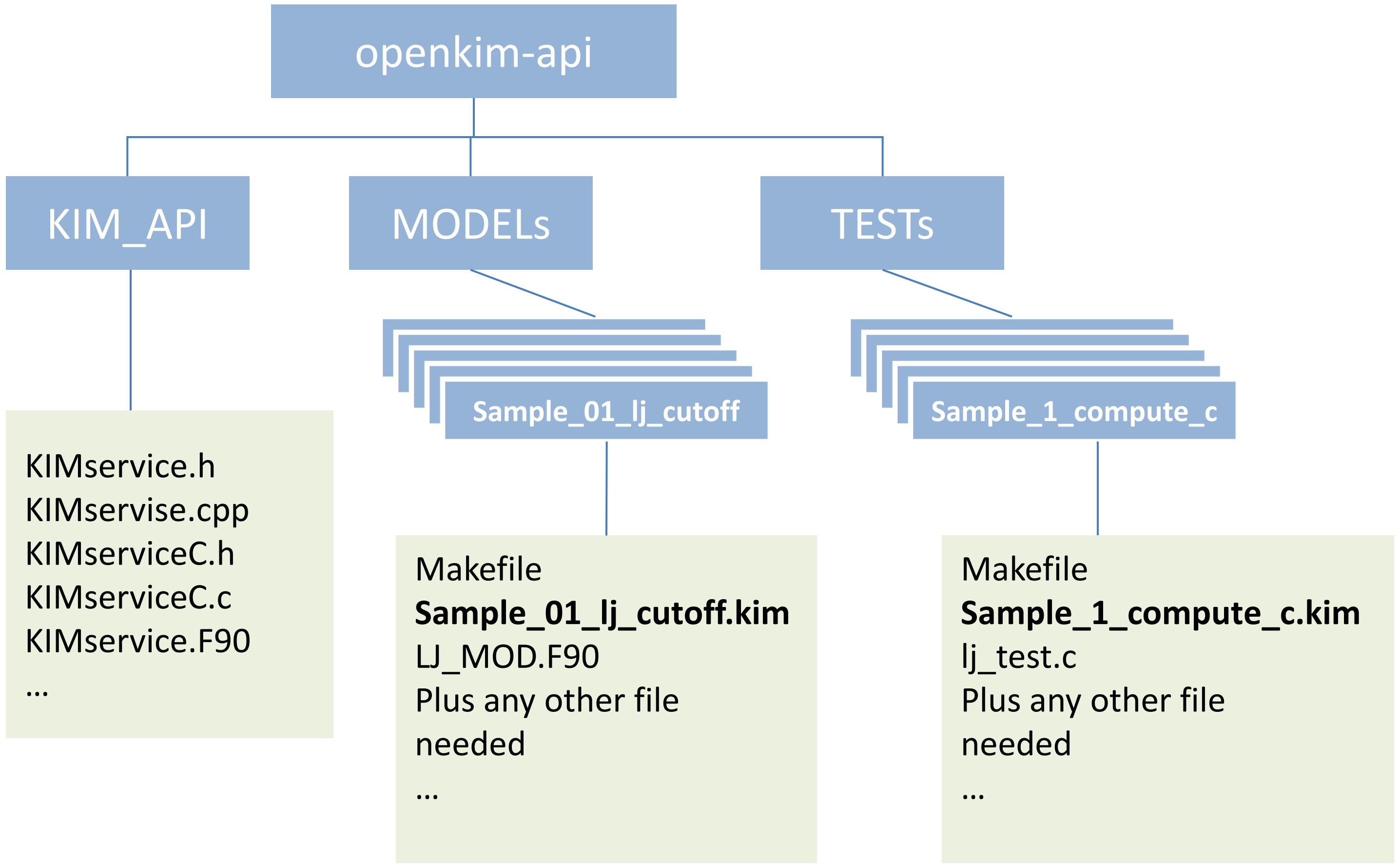
- Check if the **Model** and **Test** match, also check if their descriptor files conform to the KIM API standard
- If they do -- create a KIM API object to store all variables described in the **Model**'s descriptor file
- Mark each optional variable that is not used by the **Test** "uncompute" (i.e.,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

Other service routines are used to:

- Set (get) variable 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 a variable in the KIM_API obejct (kim_api_isit_compute)
- Execute the Model's compute method (kim_api_model_compute)
- etc…

A2

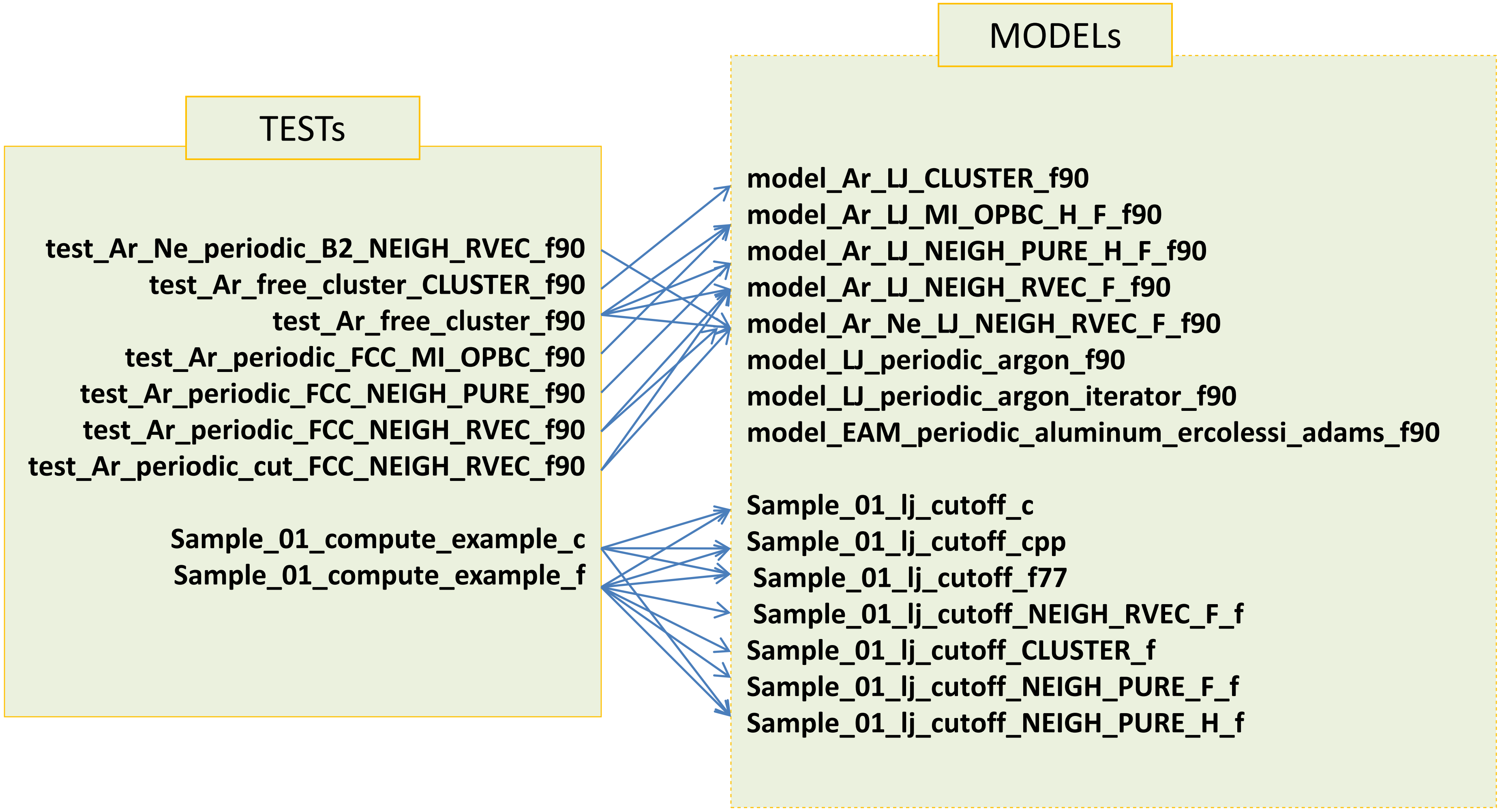
KIM API directory structure



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

A3

Number of models and test examples available in the current version of KIM API


TESTs
test_Ar_Ne_periodic_B2_NEIGH_RVEC_f90
test_Ar_free_cluster_CLUSTER_f90
test_Ar_free_cluster_f90
test_Ar_periodic_FCC_MI_OPBC_f90
test_Ar_periodic_FCC_NEIGH_PURE_f90
test_Ar_periodic_FCC_NEIGH_RVEC_f90
test_Ar_periodic_cut_FCC_NEIGH_RVEC_f90
Sample_01_compute_example_c
Sample_01_compute_example_f
MODELs
model_Ar_LJ_CLUSTER_f90
model_Ar_LJ_MI_OPBC_H_F_f90
model_Ar_LJ_NEIGH_PURE_H_F_f90
model_Ar_LJ_NEIGH_RVEC_F_f90
model_Ar_Ne_LJ_NEIGH_RVEC_F_f90
model_LJ_periodic_argon_f90
model_LJ_periodic_argon_iterator_f90
model_EAM_periodic_aluminum_ercolessi_adams_f90
Sample_01_lj_cutoff_c
Sample_01_lj_cutoff_cpp
Sample_01_lj_cutoff_f77
Sample_01_lj_cutoff_NEIGH_RVEC_F_f
Sample_01_lj_cutoff_CLUSTER_f
Sample_01_lj_cutoff_NEIGH_PURE_F_f
Sample_01_lj_cutoff_NEIGH_PURE_H_f

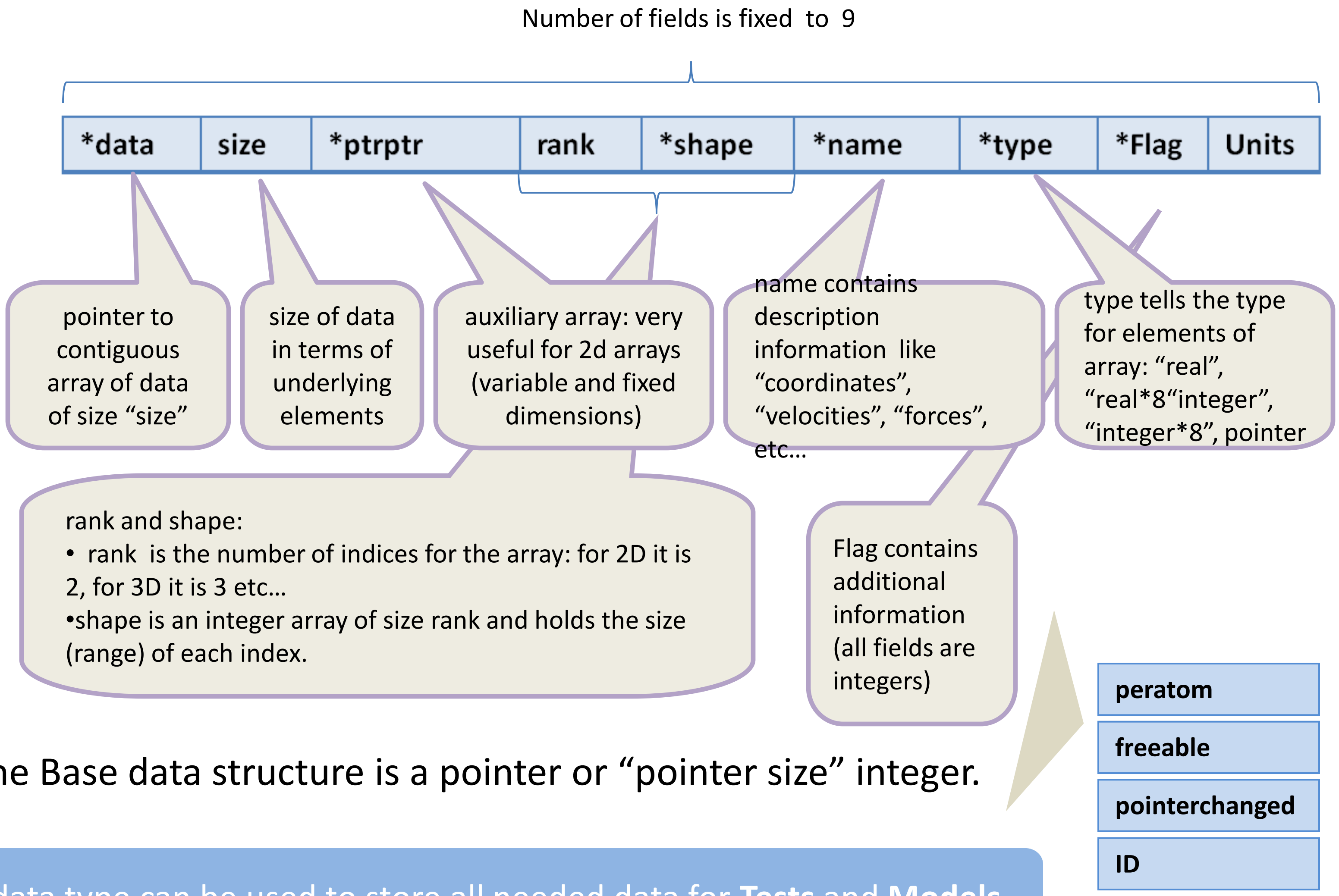
$\longrightarrow$ Indicates a **Test** can work (match) with a **Model** in the current KIM API version

A4

KIM API object is an array of Base data elements. Each Base data element can hold a pointer to any relevant data

Base data:

- Can hold any type of array: real, integer, pointer…
- Stores enough information for a complete description of the data



pointer to contiguous array of data of size “size”

size of data in terms of underlying elements

auxiliary array: very useful for 2d arrays (variable and fixed dimensions)

name contains description information like “coordinates”, “velocities”, “forces”, etc…

type tells the type for elements of array: “real”, “real*8“integer”, “integer*8”, pointer

rank and shape:
- rank is the number of indices for the array: for 2D it is 2, for 3D it is 3 etc…
- shape is an integer array of size rank and holds the size (range) of each index.

Flag contains additional information (all fields are integers)

Every field in the Base data structure is a pointer or “pointer size” integer.

Base data type can be used to store all needed data for **Tests** and **Models**