

Knowledgebase of Interatomic Models Application Programming Interface (KIM API)

Valeriu Smirichinski, Ryan S. Elliott and Ellad B. Tadmor
Dept. of Aerospace Engineering and Mechanics, University of Minnesota

August 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 holds 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
9. KIM installation: compilation, linking and running Tests

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
3. Model and Test examples available in the current version of the 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 overview

KIM TEAM



PIs

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

Developers

Valeriu Smirichinski (U. Minnesota)
Daniel Karls (U. Minnesota)
Mihir Khadilkar (Cornell)
Alex Alemi (Cornell)
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)

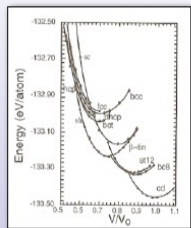
Molecular/atomistic simulations: tests and models

Tests

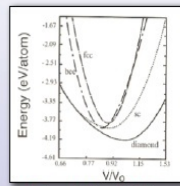
Test : a specific computer program which, when coupled with a suitable Model, calculates and returns a specific Prediction about a particular Configuration (or sequence of Configurations for dynamical properties).

Example of a Molecular Simulation

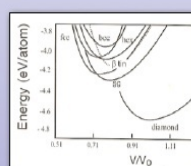
Transferability: Silicon bulk phases



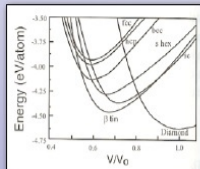
DFT (Needs and Mujica, 1995)



Stillinger-Weber (1985)



Tersoff (1988)



Ackland (1989)



fcc



hcp



bcc



β-tin



diamond

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

E. B. Tadmor

45

Models

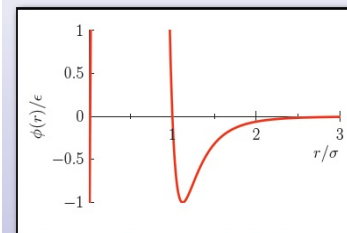
Model : Computer implementation representing a specific interaction between atoms, e.g. an interatomic potential or force field

$$\text{Total energy: } E_{\text{tot}} = \sum_i E_i(x_1, \dots, x_N)$$

$$\text{energy of atom } i \quad E_i = \frac{1}{2} \sum_{j \neq i} \phi(r_{ij})$$

distance between atoms i and j $r_{ij} = \|x_j - x_i\|$

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 (σ , ϵ)
- Designed for the noble gases (Ne, Ar, Kr, Xe).

Types of molecular modelers

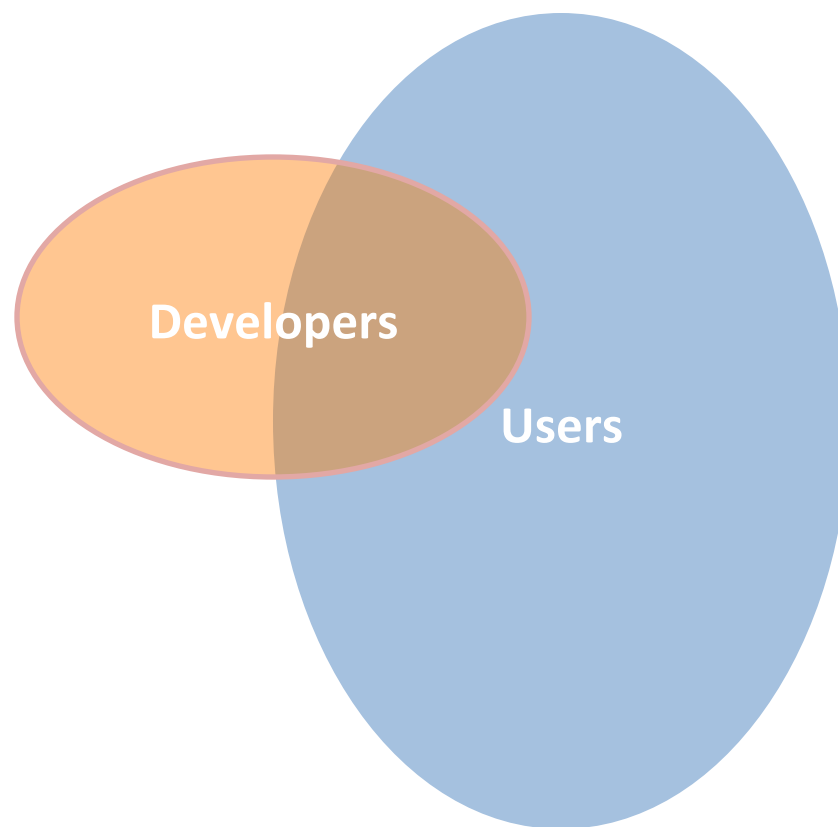
Very broadly speaking there are two types of *molecular modelers*:

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

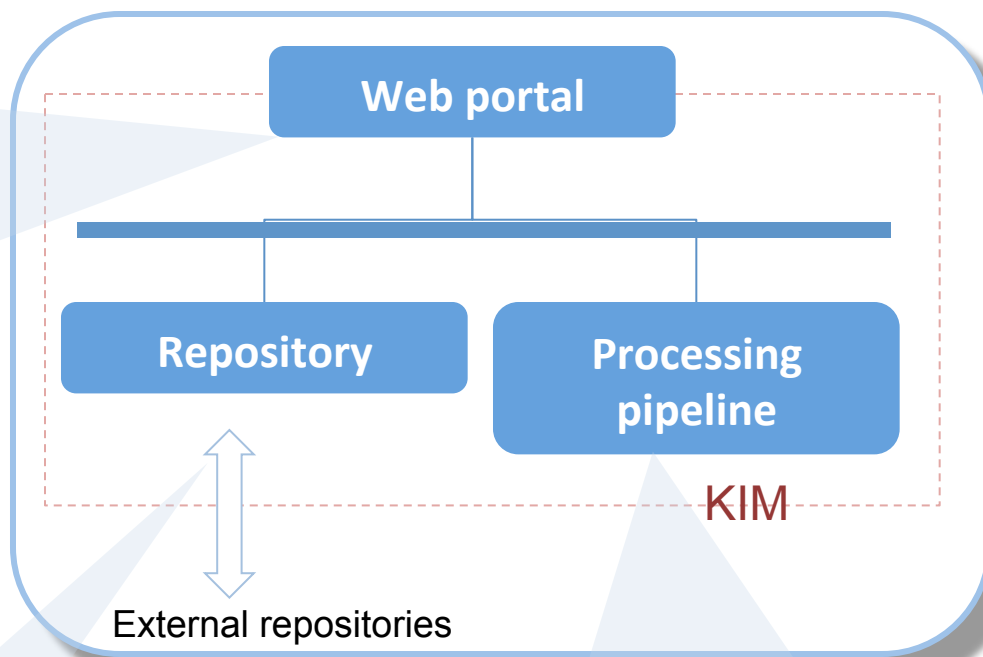
KIM framework

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

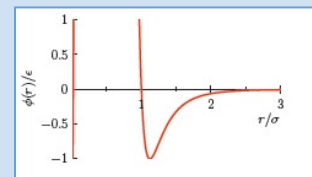
Model: Computer implementation representing a specific interaction between atoms, e.g. an interatomic potential or force field.

- Model Format

- Stand-alone Model (black box)
- Model Driver (e.g. Lennard-Jones)

+ Parameter Set (e.g. $\epsilon_{\text{Ar}} = 10.4 \text{ meV}$, $\sigma_{\text{Ar}} = 0.34 \text{ nm}$)

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



Lennard-Jones (pair)

- Ar parameterization

• ...

::

Morse (pair)

- Cu parameterization

• ...

::

Born-Mayer (ionic pair)

::

Stillinger-Weber (3-bdy)

- Si parameterization

• ...

::

MGPT (4-body)

- Mo parameterization

• Ta parameterization

• ...

::

CHARMM/AMBER

::

EAM/Finnis-Sinclair/glue

::

MEAM

::

Tersoff

::

EDIP

Brenner

::

Bond-order potentials

::

ReaxFF

::

GAP

::

- 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

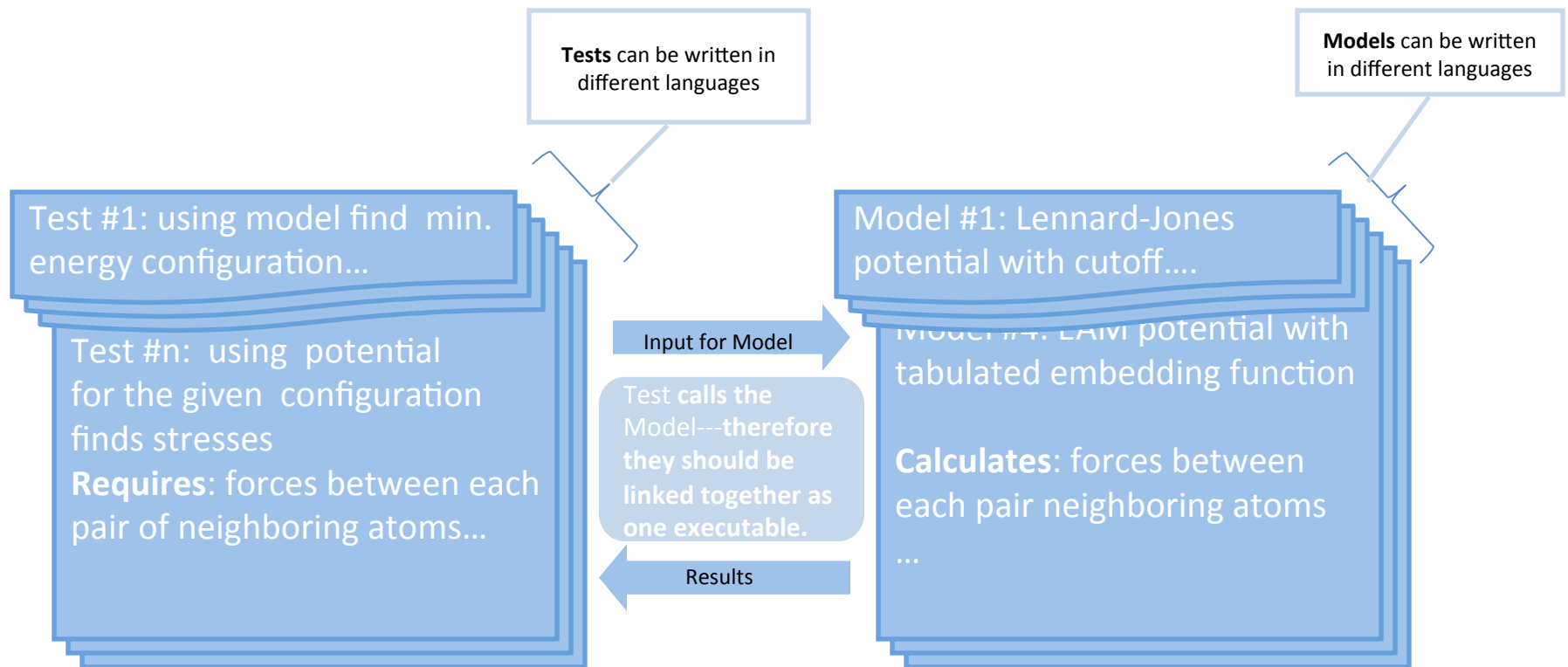
KIM API

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

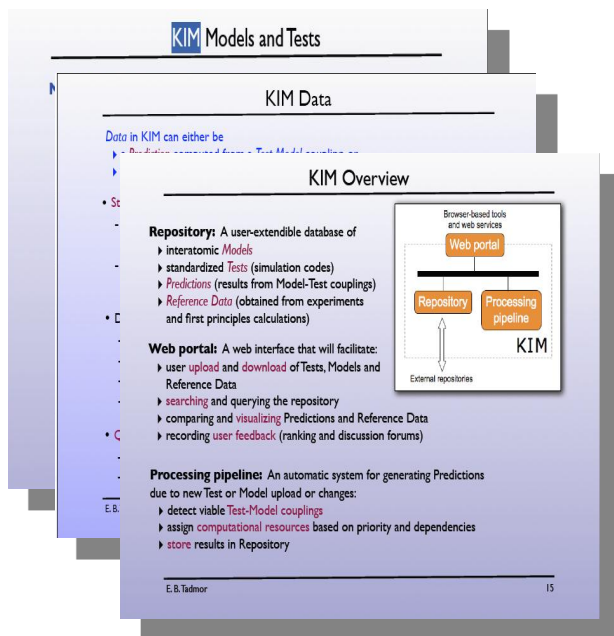
The KIM API facilitates communication between Models and Tests



Users and developers will be able to download **Tests** and **Models** (from openkim.org) , 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:

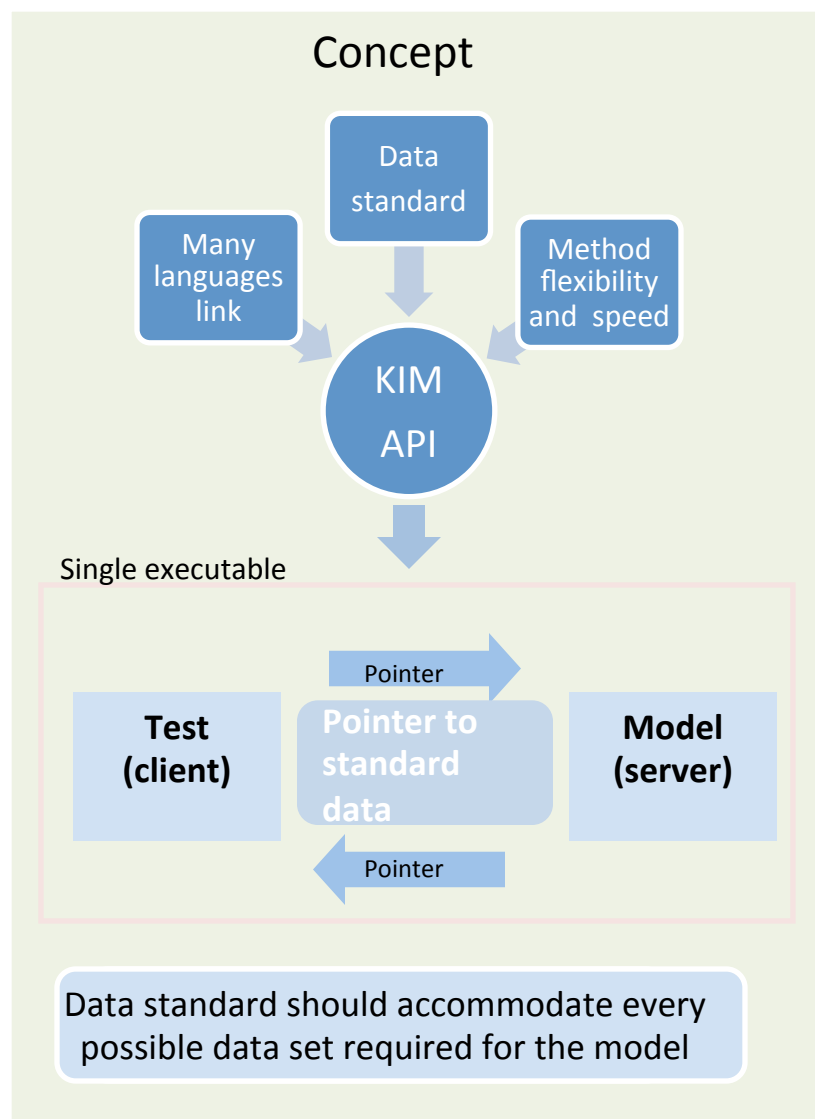
- Multilanguage support (C, C++, F77, FORTRAN 90, Python ...)
- A variety of data structures need to be accommodated: scalars, multidimensional arrays, variable size arrays, etc..
- Speed & performance are very important
- Standardized API, version tracking, etc...

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

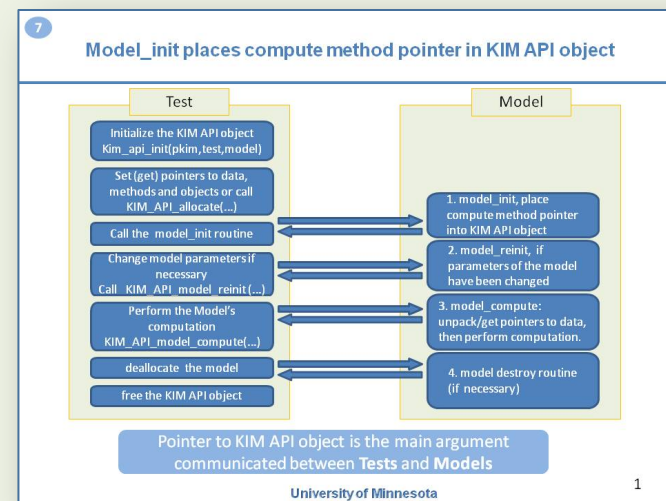
Need a simple interface : ideally just one argument per call

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 pointer in Fortran

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

FORTRAN 77 and Fortran 90/95 do not support C-style pointers directly, however essentially all compilers support the 'cray pointers' extension which provides this capability.

A cray pointer is an integer that can store a memory address. An example below shows the general syntax and usage of a cray pointer in Fortran compared with C.

FORTTRAN code

```
...  
double precision :: y=10.0d0  
double precision :: x  
pointer (px,x)  
...  
px = loc(y)  
print*, "x=", x  
...
```

Keyword pointer, followed
by two arguments

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

As soon as px holds an
address, access to that
address is done by pointee x

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

model_Ne_P_MLJ_NEIGH_PURE_H.kim

```
#####
MODEL_NAME := model_Ne_P_MLJ_NEIGH_PURE_H
SystemOfUnitsFix := fixed

#####
SUPPORTED_ATOM/PARTICLES_TYPES:
# Symbol/name          Type          code

Ne                      spec          1

...

MODEL_INPUT:
# Name                  Type          Unit          SystemU/Scale          Shape          Requirements

numberOfAtoms          integer*8     none          none                    []

numberAtomTypes        integer       none          none                    []

atomTypes              integer       none          none                    [numberOfAtoms]

...
```

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 `MODELS/model_Ne_P_MLJ_NEIGH_PURE_H/`

Structure of descriptor file

Model/Test name and system of units lines

```
MODEL_NAME:=model_Ar_P_Morse

SystemOfUnitsFix := fixed
```

Section lines

```
SUPPORTED_ATOM/PARTICLES_TYPES:

CONVENTIONS:

MODEL_INPUT:

MODEL_OUTPUT:

MODEL_PARAMETERS:
```

Data lines

- * Species Data lines
- * Dummy Data lines
- * Argument Data lines

Brief description of Section

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.

* 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 an argument and its properties

MODELS/model_Ar_P_MLJ_F90.kim

```
MODEL_NAME := model_Ar_P_MLJ_F90
SystemOfUnitsFix := fixed
...
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 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

System of units:
standard, SI, none

The shape of an argument 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 [numberOfAtoms,3] means a two-dimensional array of size numberOfAtoms 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: detailed description of all Types value , Unit, SystemU/Scale can be found in the file KIM_API/standard.kim

Specifying atom types – species data lines

MODELS/model_Ar_P_MLJ_F90.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_listAtomTypes()** service routine allows one to obtain a list of all atom species used by the model during runtime. Also the **KIM_API_get_atypeCode()** service routine allows one to get the atom species integer code (see KIMserviceDescription.txt).

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

DOCs/TEMPLATES/model_EI_P_Template.f.kim

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

OneBasedLists                        dummy

Neigh_IterAccess                     dummy

Neigh_LocaAccess                     dummy

NEIGH-RVEC-F                         dummy

NEIGH-PURE-H                         dummy

...
```

A dummy data line 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 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 `dummy`

KIM_API_allocate() has **no effect** on “dummy” type arguments, because they are not “data pointer holders”.

For a detailed description of all dummy lines see the file `KIM_API/standard.kim`. Also see template files in `DOCs/TEMPLATES/`.

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

model_Ar_P_MLJ_CLUSTER/model_Ar_P_MLJ_CLUSTER.kim

MODEL_PARAMETERS:

# Name	Type	Unit	SystemU/Scale	Shape	requirements
PARAM_FREE_sigma	real*8	length	standard	[]	
PARAM_FREE_epsilon	real*8	energy	standard	[]	
PARAM_FIXED_cutsq	real*8	area	standard	[]	
...					

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)

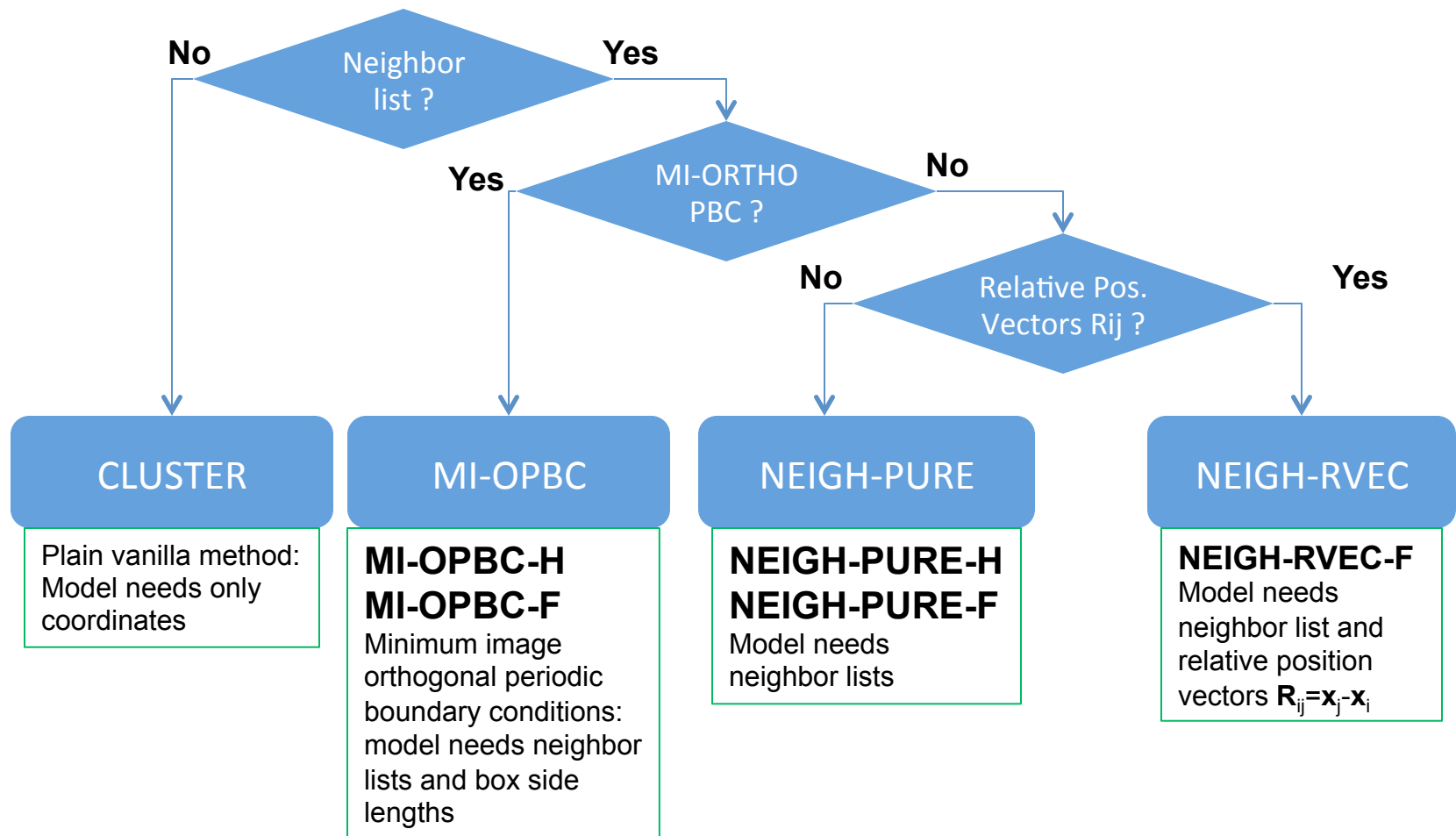
`KIM_API_get_listParams()` service routine will return a list of all parameters in the object during runtime (as an array of text strings).

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

`KIM_API_get_listFixedParams()` will return a list of FIXED parameters (see `KIMserviceDescription.txt`)

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

Handling of Neighbor lists and Boundary Conditions – NBC methods



Descriptions of the NBC methods

CLUSTER:

Receives the number 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 the number 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 cutoff range. Computes the requested quantities under the assumption that the atoms are subjected to minimum image, orthogonal, periodic boundary conditions.

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

1. The minimum image convention is applied during construction of the neighbor list consistent with the orthogonal 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-PURE-[F|H]:

Receives the number of atoms, 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, or to use a modified neighbor list to obtain the desired values.

NEIGH-RVEC-F:

Receives the number of atoms and coordinates, a full neighbor list and the relative position vectors \mathbf{R}_{ij} ($\mathbf{R}_{ij} = \mathbf{x}_j - \mathbf{x}_i$). The neighbor list and \mathbf{R}_{ij} 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.

Example of using NBC methods in KIM file

DOCs/TEMPLATES/model_EI_P_Template.f.kim

```
...  
CONVENTIONS:  
# Name                               Type  
OneBasedLists                        dummy  
  
NEIGH-RVEC-F  
  
NEIGH-PURE-H                        dummy  
  
NEIGH-PURE-F                        dummy  
  
...  
  
CLUSTER                             dummy
```

The template example in model_EI_P_Template.f.kim is designed to work with five 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 lines for the chosen method are in KIM descriptor file.

NBC Methods

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**. The neighbor list object and the method to access it 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

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).

```
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, intent(out) :: nei1atom(1); pointer(pnei1atom,nei1atom)
  double precision, intent(out) :: pRij
  double precision, intent(out) :: Rij(3,*); pointer(pRij,Rij)
end function get_half_neigh
```

FORTTRAN style

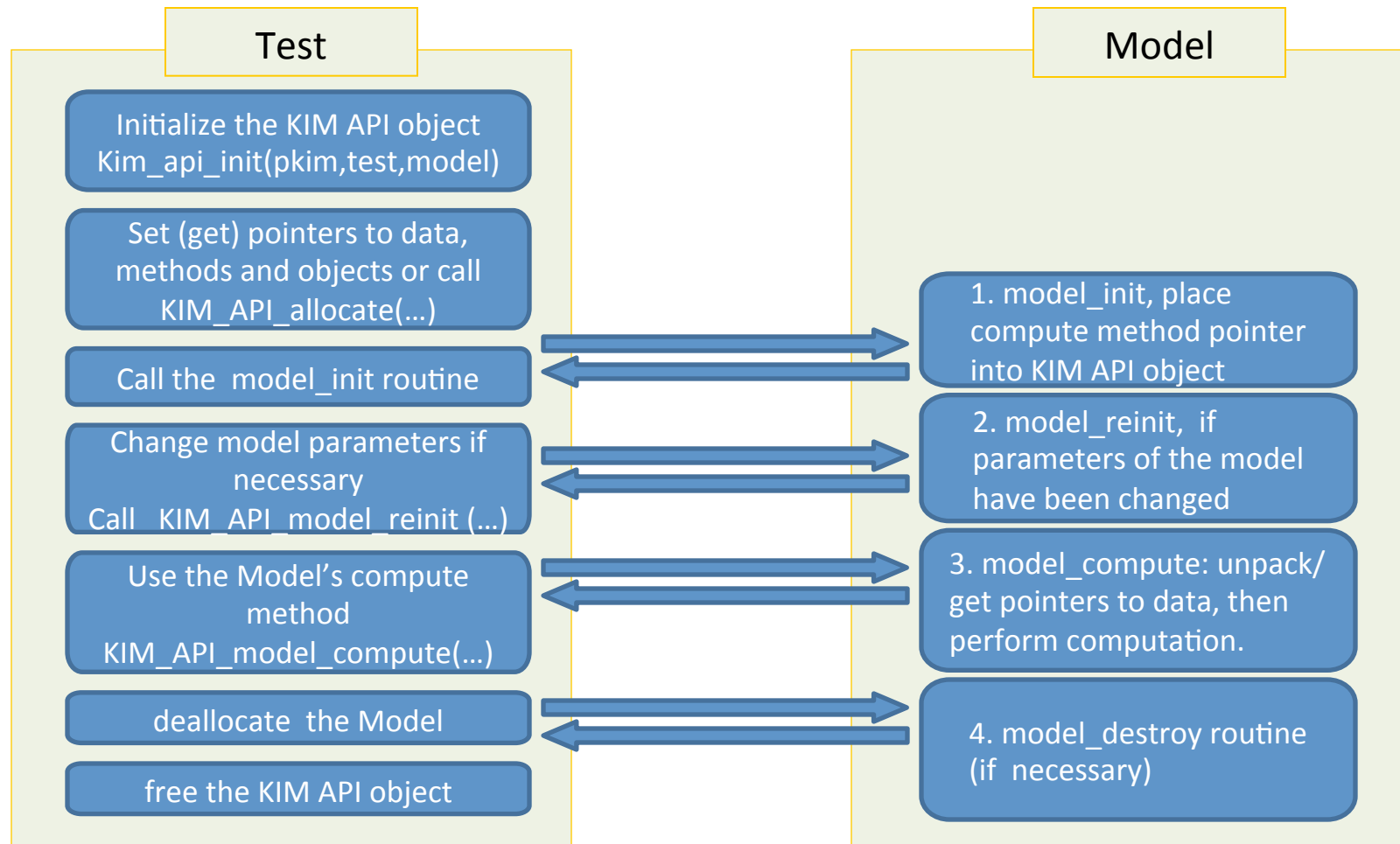
```
int get_half_neigh(void ** pkim, int * mode, int * request, int * atom,
                  int * numnei, int ** pnei1atom, double ** pRij);
```

C style

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 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

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 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

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

test_Ar_free_cluster_CLUSTER/test_Ar_free_cluster_CLUSTER.F90

```
...
! Initialize the KIM object
ier = kim_api_init_f(pkim, testname, modelname)
if (ier.le.0) then
  call report_error(__LINE__, "kim_api_init_f", ier)
  stop
endif
! Allocate memory via the KIM system
call kim_api_allocate_f(pkim, N, ATypes, ier)
if (ier.le.0) then
  call report_error(__LINE__, "kim_api_allocate_f", 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, atom 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

test_Ar_multiple_models/test_Ar_multiple_models.c

```
...
if (1 != (status = KIM_API_init(&pkim_periodic_model_0,
testname, argv[1])))
  report_error(__LINE__, "KIM_API_init() for MODEL_ZERO
for periodic", status);
...
```

KIM_API_allocate will allocate memory for all arguments 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 get/set data

test_Ar_free_cluster_CLUSTER/test_Ar_free_cluster_CLUSTER.F90

```
...
integer(kind=8) numberOfAtoms;
pointer(pnAtoms,numberOfAtoms)
...
! Unpack data from KIM object
!
pnAtoms = kim_api_get_data_f(pkim, "numberOfAtoms", ier);
if (ier.le.0) then
    call report_error(__LINE__, "kim_api_get_data_f", ier)
    stop
endif
...
```

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

test_Ar_multiple_models/test_Ar_multiple_models.c

```
...
/* Register memory */
/* model inputs */
status = KIM_API_set_data(pkim_periodic_model_0, "numberOfAtoms", 1, &numberOfAtoms_periodic);
if (1 != status) report_error(__LINE__, "set data", status);
status = KIM_API_set_data(pkim_periodic_model_1, "numberOfAtoms", 1, &numberOfAtoms_periodic);
if (1 != status) report_error(__LINE__, "set data", status);
...
```

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

8.3 KIM_API_model_init will call model initialize routine that in turn will place model compute into KIM object

test_Ar_multiple_models/test_Ar_multiple_models.c

```
...
/* call model init routines */
if (1 != (status = KIM_API_model_init+
(pkim_periodic_model_0))) report_error
(__LINE__, "KIM_API_model_init", status);
...
/* call compute functions */
KIM_API_model_compute(pkim_periodic_model_0, &status);
if (1 != status) report_error(__LINE__, "compute",
status);
...
```

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

Place address of actual compute routine into the KIM API object

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. In C the name of the model init routine must have all lower case letters in the following format modelname_init_, for example:
model_ar_p_mlj_cluster_init_
model name

DOCs/TEMPLATES/model_EI_P_Template.F90

```
...
subroutine model_<FILL element name>_P_<FILL model name>_init(pkim)
...
! store pointer to compute function in KIM object
if (kim_api_set_data_f(pkim, "compute", one, loc(Compute_Energy_Forces)) .ne.1) &
stop '* ERROR: compute keyword not found in KIM object.'
...

```

An example of using get_half/full_neigh methods through KIM API service routines

MODELS/ model_Ar_P_MLJ_NEIGH_PURE_H_F/ model_Ar_P_MLJ_NEIGH_PURE_H_F.F90

```
...
do i = 1,numberOfAtoms
  ! Get neighbors for atom i
  !
  atom = i ! request neighbors for atom i
  if (HalfOrFull.eq.1) then
    ier kim_api_get_half_neigh_f(pkim,1,atom,atom_ret,numnei, &
                                pneilatompRij_dummy)

  else
    ier = kim_api_get_full_neigh_f(pkim,1,atom,atom_ret,numnei, &
                                   pneilatompRij_dummy)

  endif
  if (ier.le.0) then
    call report_error(__LINE__, "kim_api_get_*_neigh", ier)
    return
  endif
  ! Loop over the neighbors of atom i
  !
  do jj = 1, numnei
    j = neilatompRij(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 an atom using half or full neighbor lists as requested.

KIM_API_get_half/full_neigh will call the method using the address stored in the KIM API object ("get_half_neigh" or "get_full_neigh"). These methods are supplied by the Test.

KIM_API_get_half/full_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/KIMserviceDescription.txt**

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.

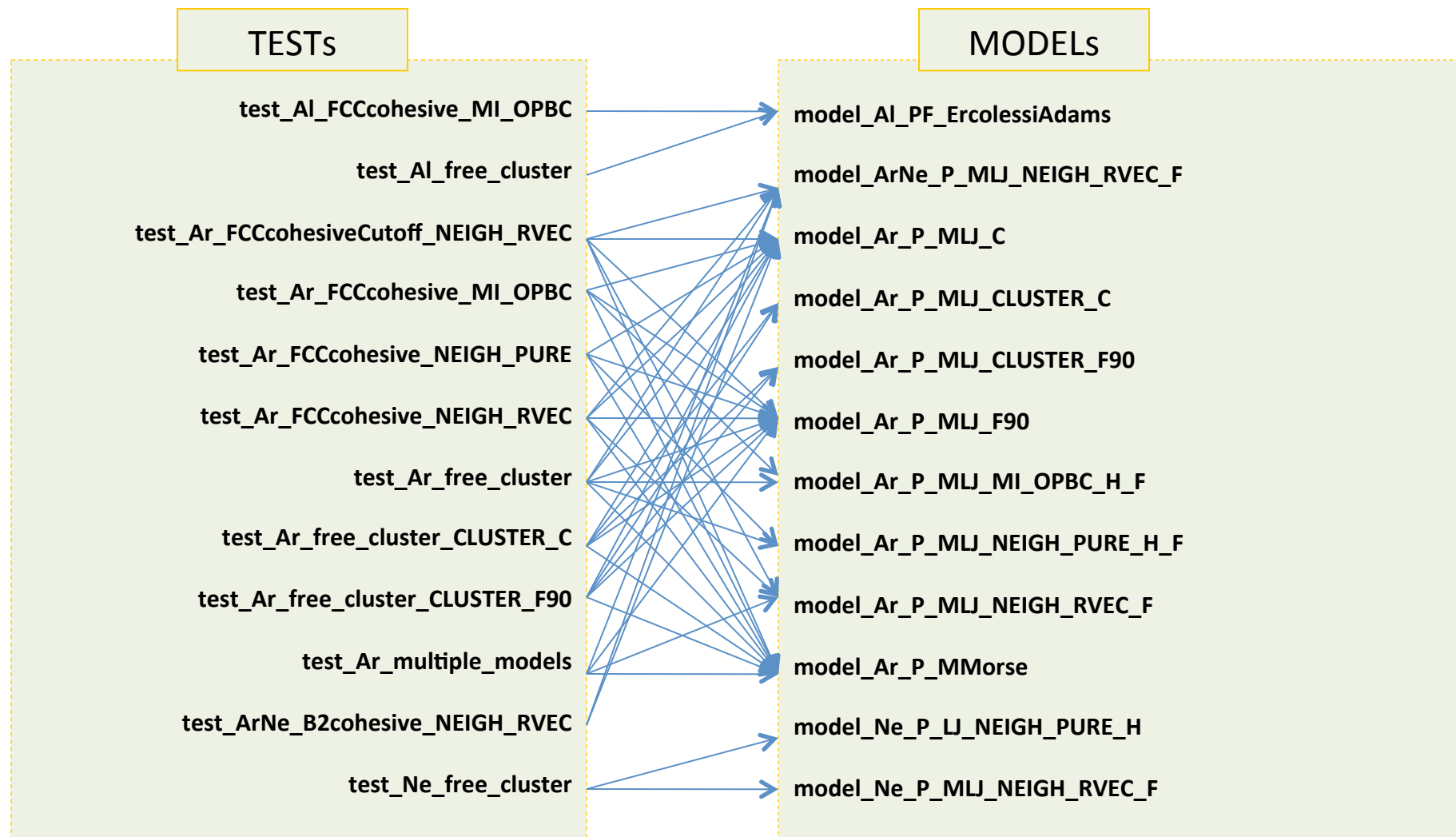
KIM service routines (such as `kim_api_init_`) use both **Test** and **Model** descriptor files to:

- 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** “uncompute” (i.e., 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_isit_compute`)
- Execute the Model's compute method (`kim_api_model_compute`)
- etc...

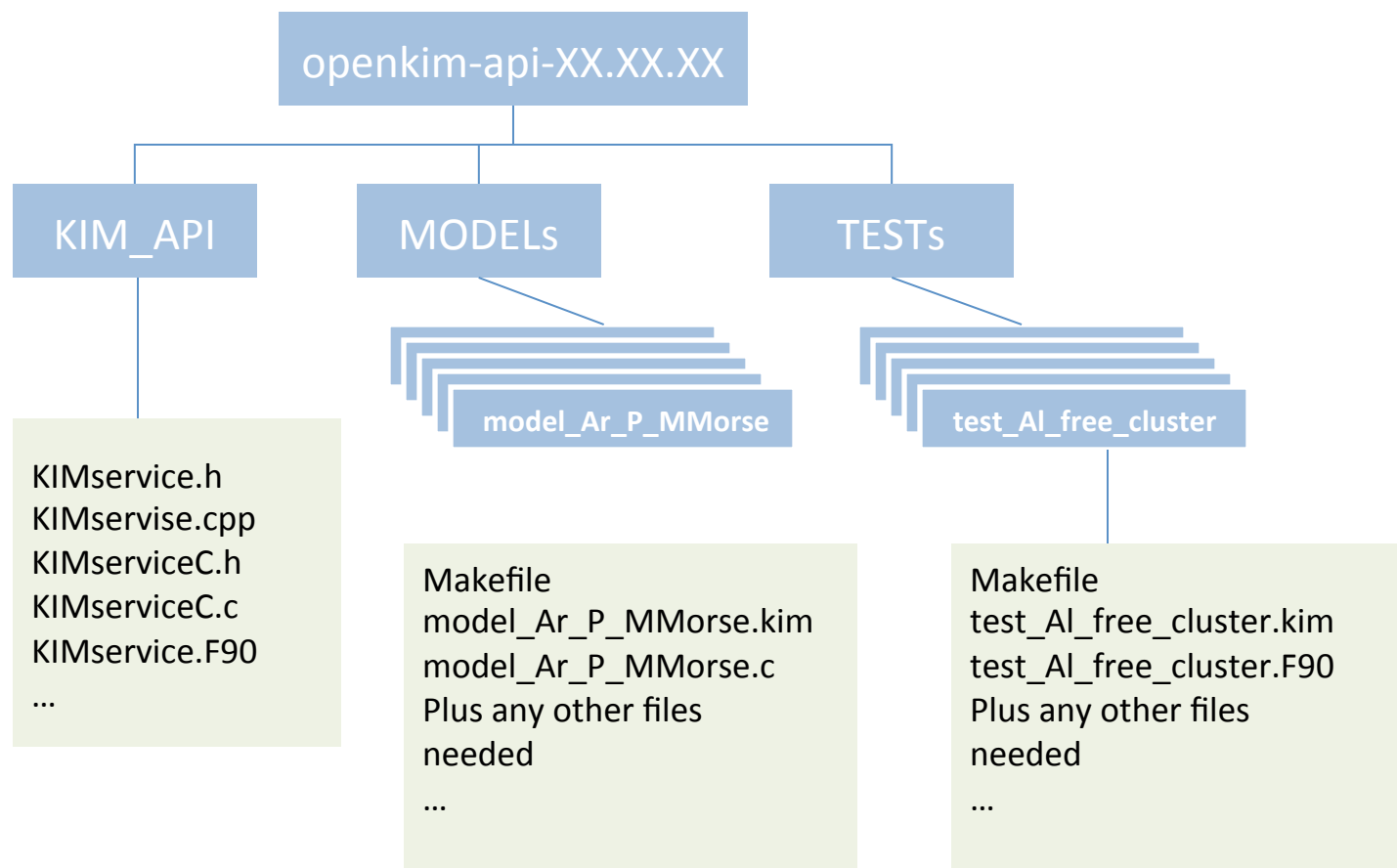
Model and Test examples available in the current version of KIM API



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

Description of the Models and Tests provided with the KIM API package are given in the files MODELS/EXAMPLES.README and TESTs/EXAMPLES.README.

KIM API directory structure



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

KIM installation: compilation, linking and running tests

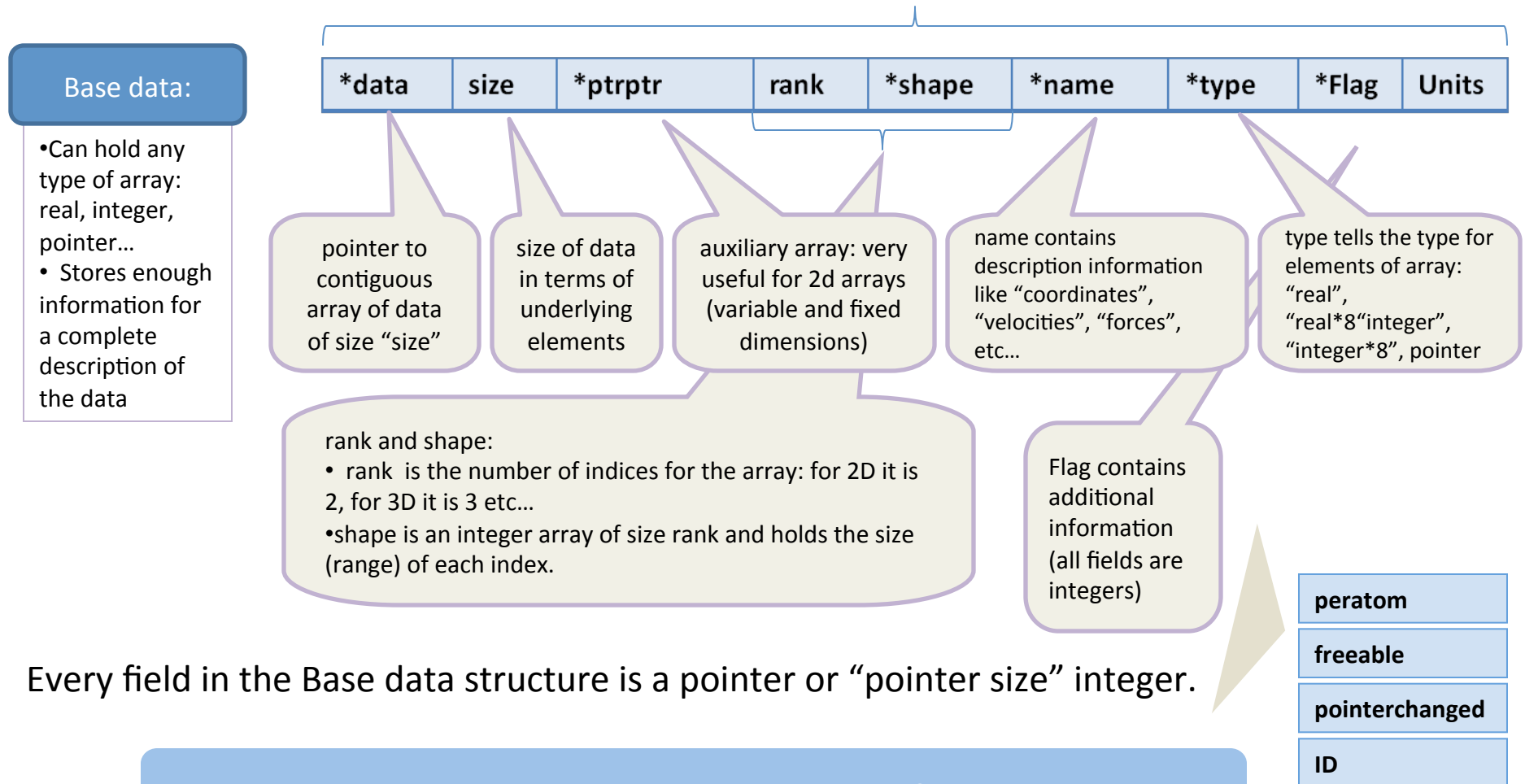
Instructions for installing, compiling and linking KIM:

1. In the desired directory, execute the command: `'tar xzvf openkim-api-XX.XX.XX.tgz'`
2. Set up environment variable (bash):
> `export KIM_DIR=/your_location/openkim-api/`
your_location is the correct path where the openkim-api directory 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 directory to `EXAMPLES_LEGO` type make: it will populate TESTs and MODELs directories with actual code.
4. change to the KIM_DIR directory and execute the commands:
 `'make clean'`
 `'make'`
 This will compile the KIM API and examples in **TESTs** and **MODELs**.
5. To run a Test, change to the appropriate directory and run the executable. For example:
 > `cd TESTs/<test_name>`
 > `echo "<model_name>" | ./<test_name>`

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

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

Number of fields is fixed to 9



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

The end