McStas — Monte Carlo Simulation of Triple Axis Spectrometre

Kristian Nielsen

June 4, 1998

1 Introduction

This document describes the first version of McStas as developed under the XENNI project. The goal of McStas is to automatically translate high-level specifications of triple-axis neutron spectrometer and similar instruments into executable algorithms for monte carlo ray-tracing simulations. The simulations will be used for widely varying tasks ranging from finding a good configuration of a particular instrument for a particular experiment to the design of new instruments for the next generation of neutron scattering sources.

In this, the initial phase of the project, the primary focus is on developing a prototype to serve as a testbed to determine the necessary future directions. In particular, a clarification is needed of the following points:

- What models of neutron-instrument interactions must be developed (such as monochromators, analysators, samples, guides etc.)
- What types of mathematical and statistical techniques are needed.
- What kind of user interfaces will be useful (for example graphical presentation of results, TASCOM/ODA interface).

In the following, the prototype is described in some detail, along with a collection of some ideas for the future development.

2 Overall System Architecture

2.1 Separating the Key Elements of Simulation

In a simulation of a neutron scattering experiment, we can identify four necessary key elements, which we refer to as system *layers*:

1. The modeling of the physical processes of neutron scattering, ie. the calculation of the fate of a neutron that passes through the individual components of the instrument (absorbtion, scattering at a particular angle, etc.)

2. The modeling of the overall instrument geometry, mainly consisting of the kind and placement of the individual components.

3. The proper calculation, using monte carlo techniques, of instrument properties such as resolution function from the result of ray tracing of a large number of neutrons. This includes estimating the accuracy of the calculation.

4. The presentation of the calculations, graphical or otherwise.

Though obviously interrelated, these four layers can be usefully treated independently, and this is reflected in the overall system architecture of McStas. The end user will in many situations be interested in knowing the details only in some of the layers. For example, one user will merely look at some results prepared by others, without worrying about the details of the calculation. Another user might want to simulate a new instrument without having to reinvent the code for simulating the individual components in the instrument. A third user may write an intricate simulation of a complex analysator such as the one in the RITA spectrometer, and expect other users to easily benefit from his or her work, and so on. McStas attempts to make it possible to work at any combination of layers in isolation by reflecting the layers in the basic program structure.

2.2 Modularity through Compilation

An important goal in McStas is to achieve a very high degree of modularity and flexibility in the construction of simulations. Simulations should be easy to build and flexible so that new ideas can quickly be tried; individual components should be simple to write and re-use by different users; and good debugging tools must be available if one is to have any kind of confidence in the system. To meet all of these goals, McStas is based on a *compiler*. Instruments and components are specified in a special McStas language that is tailored to the task of simulating neutron scattering instruments. McStas then reads the specifications and constructs a C program that implements the actual simulation.

Using a compiler-based approach has several advantages over writing a big monolithic program or a set of library functions. Specifications are much simpler to write and easier to read when the syntax of the specification language reflects the problem domain. For example, the geometry of instruments would be much more complex if it were specified in C code with static arrays and pointers. The compiler can also take care of the low-level details of interfacing the various parts of the specification with the underlying C implementation language and each other, so that users do not need to know about McStas internals to write new component or instrument definitions. This also means that it is possible to extend or change the McStas compiler without modifying specifications, for example to optimize the generated simulations or to generate a simulation that graphically displays neutron trajectories.

The high degree of modularity makes McStas well suited for doing "what if..." types of simultations. Once an instrument has been defined, questions such as "what if a slit was inserted", "what if a focusing monochromator was used instead of a flat one", "what if the sample was offset 2mm from the center of the axis" and so on are easy to answer; in a matter of minutes the instrument definition can be modified and a new simulation program generated. It also makes it simple to debug new components in isolation. A test instrument definition is written containing a source, the component to be tested, and whatever detectors are useful, and the component can be thoroughly tested before being used in a complex simulation with many different components.

3 Program Description

The first prototype deals mainly with the first two layers of the final system, that is the modeling of the instrument and of the individual components used in the instrument. This will make it possible to do some simple experiments to get an idea of what will be needed in the other two layers.

McStas is started with an argument that is the name of a file containing a high-level description of an instrument. The program reads this file, as well as the files describing the components used in the instrument definition, reporting any errors encountered. If no errors occur, the program outputs a C program containing an algorithm for simulating the instrument.

The following sections describe the format of the input and output. A complete example instrument description is found in the appendix.

3.1 Component Definitions.

The purpose of a component definition is to model the interaction of a neutron with the component. Given the state of the incoming neutron, the component definition calculates the state of the neutron when it leaves the instrument.

The state of the neutron is given by its position (x, y, z), its velocity (v_x, v_y, v_z), the time t, and the spin[1] s_1, s_2. In addition, the outgoing neutron has an associated weight p which is used to model fractional neutrons in the

[1]Spin is not currently correctly handled.

monte carlo simulation (so p=0.2 means that the neutron would have been absorbed 80% of the time).

The calculation of the interaction is performed by a block of C code, surrounded by delimiters "`%{`" and "`%}`". The delimiters should appear on a line by themselves.

A component definition looks as follows:

> `DEFINE COMPONENT` *name*

This marks the beginning of the definition, and defines the name of the component.

> `DEFINITION PARAMETERS` $(d_1, d_2, \ldots)$

This declares the definition parameters of the component. Definition parameters define properties of the component that cannot change for a given physical incarnation of the component, but which might vary among different components of the same type. Typical examples are physical dimensions, crystal plane distances, etc.

> `SETTING PARAMETERS` $(s_1, s_2, \ldots)$

This declares the setting parameters of the component. Setting parameters define the configuration of the component and typically changes during an experiment. An example is the position of a motor.

> `STATE PARAMETERS` $(x, y, z, v_x, v_y, v_z, t, s_1, s_2, p)$

This declares the parameters that define the state of the incoming neutron. The task of the component code is to assign new values to these parameters based on the old values and the values of the definition and setting parameters.

```
DECLARE
%{
        ... C code declarations ...
%}
```

This gives C declarations of global variables etc. that are used by the component code. This may for instance be used to declare a neutron counter for a detector component. This section is optional.

```
INITIALIZE
%{
        ... C code initialization ...
%}
```

This gives C code that will be executed once at the start of the simulation, usually to initialize any variables declared in the `DECLARE` section. This section is optional.

```
TRACE
%{
        ... C code to compute neutron interaction with compo-
nent ...
%}
```

This does the actual computation of the interaction between the neutron and the component. The C code should perform the appropriate calculations and assign the resulting new neutron state to the state parameters. The C code may also execute the special macro `ABSORB` to indicate that the neutron has been absorbed in the component.

```
FINALLY
%{
        ... C code to execute at end of simulation ...
%}
```

This gives code that will be executed when the simulation has ended. This might be used to print out results from components, eg. the value of the counter in a detector.

```
END
```

This marks the end of the component definition.

All C code may refer to the values of the definition and setting parameters, but the state parameters are only accessible to the `TRACE` code. Everything between the `%{` and `%}` delimiters is copied verbatim to the output and must therefore be legal C code. Outside the C code, everything from a "`%`" character to the end of a line is treated as a comment (though normal C style `/*` ... `*/` comments may also be used).

3.2 Instrument Definitions

The purpose of the instrument definition is to specify a sequence of components, along with their position and parameters, which together make up an instrument. Each component is given its own coordinate system which may be translated and rotated in any way relative to each other. The instrument definition is translated into a C program that will calculate the path of a neutron that passes through the components in the sequence given in the definition.

An instrument definition looks as follows:

`DEFINE INSTRUMENT` *name* $(a_1, a_2, \ldots)$

This marks the beginnign of the definition. It also gives the name of the instrument and the list of instrument parameters. Instrument parameters describe the configuration of the instrument, and usually correspond to setting parameters of the components. A motor position is a typical example of an instrument parameter.

> `DECLARE`
> `%{`
> ... C declarations of global variables etc. ...
> `%}`

As for component definitions, this gives C declarations that may be refered in the rest of the instrument definition. This section is optional.

> `INITIALIZE`
> `%{`
> ... C initializations. ...
> `%}`

This gives code that is executed when simulation starts. This section is optional.

After this comes the list of components that constitute the instrument, followed by "`END`" to mark the end of the definition.

Components are declared like this:

> `COMPONENT` *name* $=$ *comp*$(p_1 = v_1, p_2 = v_2, \ldots)$

This declares a component named *name* that is an instance of the component definition named *comp*. The parameter list gives the setting and definition parameters for the component. The values $v_1, v_2, \ldots$ can be constants. Setting parameters may also be assigned the values of instrument parameters. Definition parameters may be assigned the value of an arbitrary C identifier by use of the construct `EXTERN` *var*. The McStas program takes care to rename parameters appropriately in the output so that no conflicts occur between different component definitions or between component and instrument definitions. It is thus quite possible (and usual) to use a component definition multiple times in an instrument description.

By default, components are positioned in a coordinate system that is positioned at $(0, 0, 0)$ and is not rotated. This may be changed with the `AT` and `ROTATED` modifiers. By writing

> `AT` (x, y, z) `RELATIVE` *name*

The component is placed at position (x, y, z) in the coordinate system of the previously declared component *name*. Placement may also be absolute (not relative to any component) by writing

> `AT` (x, y, z) `ABSOLUTE`

Rotation is achieved similarly by writing

> `ROTATED` (ϕ_x, ϕ_y, ϕ_z) `RELATIVE` *name*

This will result in a coordinate system that is rotated first the angle ϕ_x around the X axis, then ϕ_y around the Y axis, and finally ϕ_z around the Z axis. Rotation may also be specified using `ABSOLUTE` rather than `RELATIVE`.

3.3 The Generated C Program

The result of running the McStas program on a valid instrument definition is to produce an output file containing the following C code:

1. Declarations of global variables for the various instrument and component parameters, along with the declarations from the users instrument and component definition files.

2. An `mcinit()` function that, when called, performs various initialization, including setting up the coordinate transformations necessary to translate between the coordinate systems of different components. This function also contains the code from the users `INITIALIZE` sections.

3. An `mcraytrace()` function that computes the path of a neutron through the entire instrument. This function contains the code from the `TRACE` sections of all the components in the instrument, as well as code to translate between the different coordinate systems of the components and handle the correct renaming of parameters. When called, this function replaces the initial neutron state with the final state when the neutron has passed all the way through the instrument.

4. A `mcfinally()` function that contains the code from the users `FINALLY` sections.

In this first prototype, no special code is generated to do monte carlo simulation or handle the user interface. Instead, such code must be placed by the user in the declarations section of the instrument definition, or placed in another C file that is linked with the generated code. A main objective with the prototype is to facilitate experimentation that will help determine what to generate to get a full and useful system.

4 The Next Step

At the time of writing, the necessary components and instruments definitions necessary to simulate the TAS1 spectrometer at Risø have been developed, and work is in progress on running some simulations and comparing them with experimental measurements. When this work is finished, the results will give an indication as to the accuracy that can be obtained using McStas. The experience obtained from using McStas for a real task will also help decide what kind of changes are needed in the McStas specification languages and in the user interface of the system. After this, the plan is to apply McStas to some more complex instruments such as the RITA spectrometer at Risø.

A A Complete Example

A.1 The instrument definition

```
/*******************************************************************************
* A simple TAS1 simulation of an Al2O3 powder sample:
*
* cold source-3 slits-monochromator-collimator
* Al2O3 powder sample
* 2-axis mode
*
* Written by: Kim Lefmann, November 4, 1997
* settings changed 8-12-97 by: AA,NBC,EML
* changed 4-2-98 AA, NBC, EML
* Rewritten for TAS1 simulation 20-5-98 KN.
*******************************************************************************/

DEFINE INSTRUMENT TAS1_2axis(OMM,TTM,OM,TT,C2,C3,NCOUNT)

DECLARE
%{

/* Number of neutrons to send through. */
int ncount;

/* 30' mosaicity used on monochromator and analysator */
double tas1_mono_mosaic = 30*MIN2RAD;
/* Q vector for bragg scattering with monochromator and analysator */
double tas1_mono_q = 1.87325*AA2MS;
double tas1_mono_r0 = 0.6;

/* Collimators */
double coll1_div = 30*MIN2RAD;
double coll2_div;
double coll3_div;


/* Global correction factor */
double MC_correction;

/* Our main() function. */
int
main(int argc, char *argv[])
{
  int run_num = 0;
  mcreadparams();
  mcinit();
  while(run_num < ncount)
  {
    mcsetstate(0, 0, 0, 0, 0, 1, 0, 0, 0, 1);
    mcraytrace();
    run_num++;
  }
  mcfinally();
  return 0;
}

%}

INITIALIZE
%{
  MC_correction = 1;  /* Scale factor is initially unity */
  coll2_div = C2*MIN2RAD;
  coll2_div = C3*MIN2RAD;
  ncount = NCOUNT;
%}

COMPONENT a1 = Axis()
  AT (0,0,0) ABSOLUTE

COMPONENT source = Source_aim(
        radius = 0.025,
        dist = 3.288,
        xw = 0.042, yh = 0.082,
        v0 = 977.9,     /* Which is 5.0 meV */
        dv = 20)                /* dE = 0.2 meV */
  AT (0,0,0) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT slit1 = Slit(
        xmin=-0.020, xmax=0.065,
        ymin = -0.075, ymax = 0.075)
  AT (0, 0, 1.1215) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT slit2 = Slit(
        xmin = -0.020, xmax = 0.020,
        ymin = -0.040, ymax = 0.040)
  AT (0,0,1.900) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT slit3 = Slit(
        xmin = -0.021, xmax = 0.021,
        ymin = -0.041, ymax = 0.041)
  AT (0,0,3.288) RELATIVE a1 ROTATED (0,0,0) RELATIVE a1

COMPONENT mono = Monochromator(
        zmin = -0.035, zmax = 0.035,
        ymin = -0.05, ymax = 0.05,
        mosaich = EXTERN tas1_mono_mosaic, mosaicv = EXTERN tas1_mono_mosaic,
        r0 = EXTERN tas1_mono_r0, Q = EXTERN tas1_mono_q)
  AT (0, 0, 3.56) RELATIVE a1 ROTATED (0, OMM, 0) RELATIVE a1

COMPONENT a2 = Axis()
  AT (0,0,0) RELATIVE mono ROTATED (0, TTM, 0) RELATIVE a1

/* Simulate 25cm collimator using a slit. */
COMPONENT c1_slit = Slit(
        xmin = -0.01, xmax = 0.01,
        ymin = -0.03, ymax = 0.03)
  AT (0, 0, 0.62) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT c1 = Collimator(
        xmin = -0.01, xmax = 0.01,
        ymin = -0.03, ymax = 0.03,
        divergence = EXTERN coll1_div)
  AT (0, 0, 0.87) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT mon = Monitor(
        xmin = -0.025, xmax = 0.025, ymin = -0.0375, ymax = 0.0375,
        psum = EXTERN mon_psum, Nsum = EXTERN mon_Nsum)
  AT (0, 0, 1.346) RELATIVE a2 ROTATED (0,0,0) RELATIVE a2

COMPONENT error_axis = Axis()
  AT (0, 0, 1.581) RELATIVE a2 ROTATED (0,OM,0) RELATIVE a2

COMPONENT sample = Powder1(
        radius = 0.006775,
        h = 0.00141,
        q = 1.8049,
        d_phi0 = 0.1,
        pack = 1, j = 6, DW = 1,
        F2 = 56.8,
        Vc = 85.0054, sigma_a = 0.463,
        target_x = -1000,  /* ToDo: GET_X(ana) */
        target_y = 0, target_z = 1000)
  AT (0.0015, 0, 0) RELATIVE error_axis ROTATED (0,0,0) RELATIVE error_axis

COMPONENT a3 = Axis()
  AT (0,0,1.581) RELATIVE a2 ROTATED (0, TT, 0) RELATIVE a2

COMPONENT c2 = Collimator(
        xmin = -0.025, xmax = 0.025,
        ymin = -0.05, ymax = 0.05,
        divergence = EXTERN coll2_div)
  AT (0, 0, 0.208) RELATIVE a3 ROTATED (0,0,0) RELATIVE a3

/* Simulate 20cm collimator using a slit. */
COMPONENT c2_slit = Slit(
        xmin = -0.025, xmax = 0.025,
        ymin = -0.05, ymax = 0.05)
  AT (0, 0, 0.408) RELATIVE a3 ROTATED (0,0,0) RELATIVE a3

COMPONENT a4 = Axis()
  AT (0, 0, 0.765) RELATIVE a3 ROTATED (0,0,0) RELATIVE a3
```

```
COMPONENT c3 = Collimator(
        xmin = -0.025, xmax = 0.025,
        ymin = -0.05, ymax = 0.05,
        divergence = EXTERN coll3_div)
  AT (0,0,0.107) RELATIVE a4 ROTATED (0,0,0) RELATIVE a4

/* Simulate 25cm collimator using a slit. */
COMPONENT c3_slit = Slit(
        xmin = -0.025, xmax = 0.025,
        ymin = -0.05, ymax = 0.05)
  AT (0, 0, 0.307) RELATIVE a4 ROTATED (0,0,0) RELATIVE a4

COMPONENT sng = Monitor(
        xmin = -0.0125, xmax = 0.0125,
        ymin = -0.05, ymax = 0.05,
        Nsum = EXTERN sng_Nsum, psum = EXTERN sng_psum)
  AT(0, 0, 0.43) RELATIVE a4 ROTATED (0,0,0) RELATIVE a4

FINALLY
%{
  printf("Finished simulation of %d neutrons.\n", ncount);
  printf("Correction factor: %g \n",MC_correction);
  printf("I = %g (%d neutrons detected)\n", sng_psum/MC_correction, sng_Nsum);
%}

END
```

A.2 Component Definition for “Axis”

```
% Component: Axis.
%

DEFINE COMPONENT Axis
DEFINITION PARAMETERS ()
SETTING PARAMETERS ()
STATE PARAMETERS ()
TRACE
%{
  /* An axis does not actually do anything, it is just there to set
     up a new coordinate system. */
%}
END
```

A.3 Component Definition for “Source_aim”

```
/* Component: Source_aim
 *
 * The routine is a circular neutron source, which aims at a square target
 * centered at the beam (in order to improve MC-acceptance rate).  The angular
 * divergence is then given by the dimensions of the target. The "cheating
 * factor" MC_correction is updated by the routine.  The neutron velocity is
 * uniformly distrubuted between v0-dv and v0+dv.
 *
 * ToDo: More flexible specification of v distribution.
 *
 * radius : Radius of circle in (x,y,0) plane where neutrons
 *          are generated.
 * dist   : Distance to target along z axis.
 * xw     : Width(x) of target
 * yh     : Height(y) of target
 * v0     : Mean velocity of neutrons.
 * dv     : Velocity spread of neutrons.
 *
 * Written by: KL, October 30, 1997
 */

DEFINE COMPONENT Source_aim
DEFINITION PARAMETERS (radius, dist, xw, yh, v0, dv)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
 double hdiv,vdiv;                    /* ToDo: Should be component local */
%}
INITIALIZE
%{
 hdiv = atan(xw/(2.0*dist));
 vdiv = atan(yh/(2.0*dist));
 MC_correction *= 4*PI/(4*hdiv*vdiv); /* Small angle approx. */
%}
TRACE
%{
 double theta0,phi0,chi,theta,phi,v,r;

 p=1;
 z=0;

 chi=2*PI*rand01();                           /* Choose point on source */
 r=sqrt(rand01())*radius;                     /* with uniform distribution. */
 x=r*cos(chi);
 y=r*sin(chi);

 theta0= -atan(x/dist);              /* Angles to aim at target centre */
 phi0= -atan(y/dist);

 theta=theta0+hdiv*randpm1();        /* Small angle approx. */
 phi=phi0+vdiv*randpm1();

 v=v0+dv*randpm1();                  /* Assume linear distribution */

 vz=v*cos(phi)*cos(theta);           /* Small angle approx. */
 vy=v*sin(phi);
 vx=v*cos(phi)*sin(theta);
%}
END
```

A.4 Component Definition for “Slit”

```
% RITA MC-resolution calculation%
% Component: Slit
%
% Written by: KL, HMR  June 16, 1997
%
DEFINE COMPONENT Slit
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
TRACE
%{
    PROP_Z0;
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;
%}
END
```

A.5 Component Definition for “Monochromator”

```
/*******************************************************************************
* RITA MC-resolution calculation
*
* Component: Flat monochromator
*
* Written by: KL, HMR  June 16, 1997
* Rewritten by: KL  Oct. 16, 1997
*
* Flat monochromator which uses a small-mosaicity approximation as well as
* the approximation vy^2 << vz^2 + vx^2.
* Mosaicity is given as FWHM.
* Second order scattering is neglected.
* Q is in units of velocity.
*******************************************************************************/

DEFINE COMPONENT Monochromator
DEFINITION PARAMETERS (zmin, zmax, ymin, ymax, mosaich, mosaicv, r0, Q)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
TRACE
  %{
    double dphi,tmp,vratio,phi,theta0,theta,v,cs,sn;

    PROP_X0;
    if (z>zmin && z<zmax && y>ymin && y<ymax)
    {
      /* First: scattering in plane */

      /* ToDo: This will not work if the incident angle > PI/2 !!!  */
      theta0 = atan2(vx,vz);           /* neutron angle to slab */
      v = sqrt(vx*vx+vy*vy+vz*vz);
      theta = asin(Q/(2.0*v));               /* Bragg's law */
      if(theta0 < 0)
```

```
        theta = -theta;
      tmp = 2*theta;
      cs = cos(tmp);
      sn = sin(tmp);
      tmp = cs*vx - sn*vz;
      vy = vy;
      vz = cs*vz + sn*vx;
      vx = tmp;
      tmp = (theta-theta0)/mosaich;
      p *= r0*exp(-tmp*tmp*4*log(2)); /* Use mosaics */

      /* Second: scatering out of plane.
                 Approximation is that Debye-Scherrer cone is a plane */

      phi = atan2(vy,vz);                              /* out-of plane angle */
      dphi = mosaicv/(2*sqrt(2*log(2)))*randnorm();  /* MC choice: */
                                        /* angle of the crystallite */
      vy = vz*tan(phi+2*dphi);
      vratio = v/sqrt(vx*vx+vy*vy+vz*vz);
      vz = vz*vratio;
      vy = vy*vratio;
      vx = vx*vratio;
    }

  %}
END
```

A.6 Component Definition for "Collimator"

```
/*******************************************************************************
* Component: Collimator
*
* Written by: KL, Oct. 8, 1997
*
* Collimator with rectangular opening. The transmission function is an average
* and does not utilize knowledge of the actual neutron trajectory.
* The divergence is given as w/l where w is the distance between coll. blades
* and l is the blade length. A zero divergence disables collimation (then the
* component works as a slit).
*******************************************************************************/


DEFINE COMPONENT Collimator
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, divergence)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
TRACE
%{
    double phi,factor;

    PROP_Z0;
    if (x<xmin || x>xmax || y<ymin || y>ymax)
      ABSORB;
    else if(divergence != 0.0)
    {
      phi = fabs(vx/vz);
      if (phi > divergence)
        ABSORB;
      else
      {
        factor = 1.0 - phi/divergence;
        p *= factor;
      }
    }
%}
END
```

A.7 Component Definition for "Powder1"

```
/*******************************************************************************
* RITA MC-resolution calculation
*
* Component: General powder sample
*
*
* SETTING parameters
*
* target_x, target_y, target_z: position of target to focus at (m)
*
* INPUT variables from McStas
*
* d_phi0    : Focussing angle corresponding to the vertical dimensions
*             of the detector placed at the right distance (rad)
* radius    : Radius of sample in (x,z) plane (m)
* h         : Height of sample y direction (m)
* pack      : Packing factor (no unit)
* Vc        : Volume of unit cell (AA^3)
* sigma_a   : Absorption cross section per unit cell (fm^2)
*
* j         : Multiplicity of reflection (no unit)
* q         : Scattering vector of reflection (AA^-1)
* F2        : Structure factor of reflection (fm^2)
* DW        : Debye-Waller factor of reflection (no unit)
*
*
* Variables calculated in the component
*
* my_s      : Attenuation factor due to scattering (m^-1)
* my_a      : Attenuation factor due to absorbtion (m^-1)
*
*
* Written by: EM,NB,ABA 4.2.98
* Rewritten by: KL, KN 20.3.98
*******************************************************************************/
```

```
DEFINE COMPONENT Powder1
DEFINITION PARAMETERS (d_phi0,radius,h,pack,Vc,sigma_a,j,q,F2,DW)
SETTING PARAMETERS (target_x, target_y, target_z)
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
%{
  double my_s_v2, my_a, q_v;     /* ToDo: Should be component local names. */
%}
INITIALIZE
%{
  MC_correction *= 2*PI/d_phi0;
  my_a = sigma_a / Vc;
  my_s_v2 = PI*PI*PI*AA2MS*AA2MS*pack*j*F2*DW/(Vc*Vc*q);
                                      /* Is not yet divided by v^2 */
  q_v = q*AA2MS;
%}
TRACE
%{
  double t0, t1, v, l_0, l, l_1, dt, d_phi, theta, my_s;
  double aim_x, aim_y, aim_z, axis_x, axis_y, axis_z;
  double tmp_vx, tmp_vy, tmp_vz, vout_x, vout_y, vout_z;

  if(cylinder_intersect(&t0, &t1, x, y, z, vx, vy, vz, radius, h))
  {
    if(t0 < 0)
      ABSORB;
    /* Neutron enters at t=t0. */
    v = sqrt(vx*vx + vy*vy + vz*vz);
    l_0 = v * (t1 - t0);                /* Length of full path through sample */
    dt = rand01()*(t1 - t0) + t0; /* Time of scattering */
    PROP_DT(dt);                /* Point of scattering */
    l = v*dt;                   /* Penetration in sample */

    d_phi = d_phi0/2*randpm1(); /* Choose point on Debye-Scherrer cone */
    theta = asin(q_v/(2.0*v));  /* Bragg scattering law */

    aim_x = target_x-x;         /* Vector pointing at target (anal./det.) */
    aim_y = target_y-y;
    aim_z = target_z-z;
    vec_prod(axis_x, axis_y, axis_z, vx, vy, vz, aim_x, aim_y, aim_z);
    rotate(tmp_vx, tmp_vy, tmp_vz, vx, vy, vz, 2*theta, axis_x, axis_y, axis_z);
    rotate(vout_x, vout_y, vout_z, tmp_vx, tmp_vy, tmp_vz, d_phi, vx, vy, vz);
    vx = vout_x;
    vy = vout_y;
    vz = vout_z;

    if(!cylinder_intersect(&t0, &t1, x, y, z,
                           vout_x, vout_y, vout_z, radius, h))
    {
      /* ??? did not hit cylinder */
      printf("FATAL ERROR: Did not hit cylinder from inside.\n");
      exit(1);
    }
    l_1 = v*t1;
    my_s = my_s_v2/(v*v);
    p *= l_0*my_s*exp(-(my_a+my_s)*(l+l_1));
  }
  else
    ABSORB;
%}

END
```

A.8 Component Definition for “Single_detector”

```
% Component: Monitor
%
% Written by: KL,  Oct. 4, 1997
%
% Sums neutrons (number and p) through the rectangular monitor opening.
% May also be used as a single detector.
%
DEFINE COMPONENT Monitor
DEFINITION PARAMETERS (xmin, xmax, ymin, ymax, psum, Nsum)
SETTING PARAMETERS ()
STATE PARAMETERS (x,y,z,vx,vy,vz,t,s1,s2,p)
DECLARE
  %{
    double psum;
    int Nsum;
  %}
INITIALIZE
  %{
    psum = 0;
    Nsum = 0;
  %}
TRACE
  %{
    PROP_Z0;
    if (x>xmin && x<xmax && y>ymin && y<ymax)
    {
      psum += p;
      Nsum++;
    }
  %}
FINALLY
  %{
    test_printf("Monitor count: %i %g.\n", Nsum, psum);
  %}
END
```