APPENDIX B

Description of Code

The full source code and Doxygen generated documentation for *PSTTools* is available online at

`http://pst.tools`.

The following should be considered a *broad strokes* description of the *PSTTools* code. *PSTTools* comprises many thousands of lines of code. Therefore, there is much detail that simply isn't feasible to include in this document. Flowcharts are only provided for illustrating high level program flow and are, again, to be considered *broad strokes* descriptions. Anyone who desires a more detailed understanding of how *PSTTools* works is encouraged to read the code for themselves, using this document as a guide.

B.1 Overview

The overall program flow of *PSTTools* is illustrated in the flowchart in Figure B.1 and is implemented by code in the files main.cpp, standard_execution.h and slave_execution.h. When *PSTTools* runs, it reads the contents of the input directory. Each input file in that directory is processed. The output for each job is written to a directory with the same name as the input file (sans file type suffix). This folder is created automatically by *PSTTools* in the output folder. *PSTTools* will also create the output folder if it doesn't find it where it expects to. The first line of the input file tells *PSTTools* what type of calculation the input file is for. *PSTTools* creates a calculation object corresponding to that type of calculation and passes the file path to the calculation object. Everything else, from parsing the input file through to writing the output, is then handled within that calculation object. The creation of this calculation object is represented by

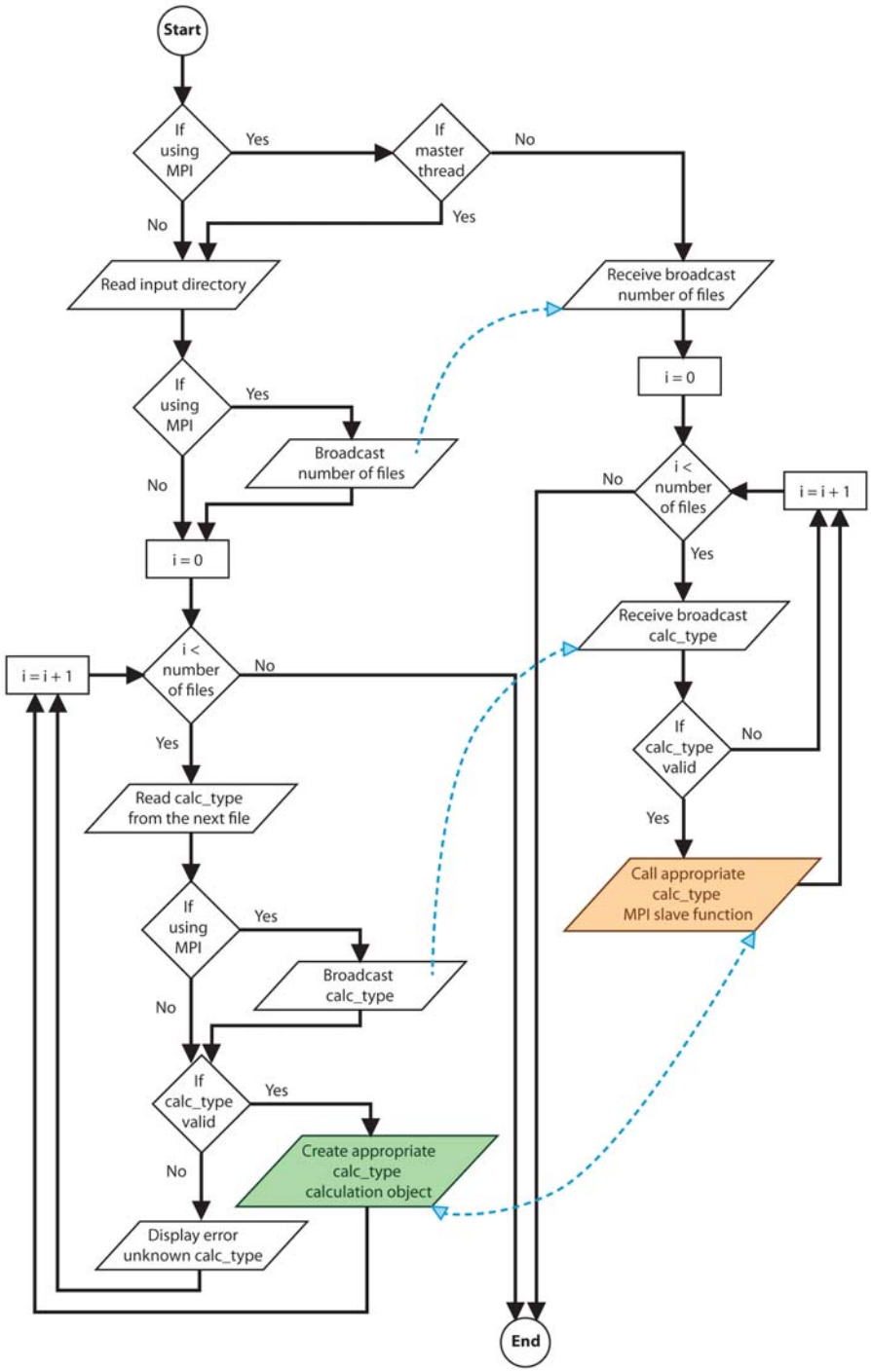

FIGURE B.1: Flowchart for overall *PST*$_{Tools}$ program flow. The pale blue dashed arrows illustrate data being sent between the slave threads and the master thread.

the green parallelogram in Figure B.1. The only exception to this is when running MPI jobs. In this case, the program must first determine whether it is running as the master thread or

not. If running as the master thread the logic above applies. Otherwise, the calculation type is passed in a message from the master thread to the slave thread and the corresponding MPI slave function (for now, class instantiation is the goal here too) is called by the slave thread. This is represented by the orange parallelogram in Figure B.1. If, according to *PSTTools*, the calculation type specified in the file does not exist, an error to that effect is displayed and *PSTTools* moves on to the next job.

There are, of course, many similarities between each type of calculation *PSTTools* does. *PSTTools* utilises a concept in C++ , called inheritance, to try to maximise code reuse. That is, we don't want to rewrite the same code for each calculation type, especially if it is identical each time. So, analogously to how one might write a function to prevent this, we write classes that inherit common functionality from others. There is a hierarchical logic to doing this. For example, all calculations will output some type of data and the code for creating the output directory is identical for all calculations. We only want to write that code once. We do that in the base calculation class, the at the top of the hierarchy; the root of the tree. All other calculation classes in *PSTTools* derive from the base calculation class and therefore inherit this functionality. The inheritance hierarchy structure for the calculation classes in *PSTTools* is shown in Figure B.2.

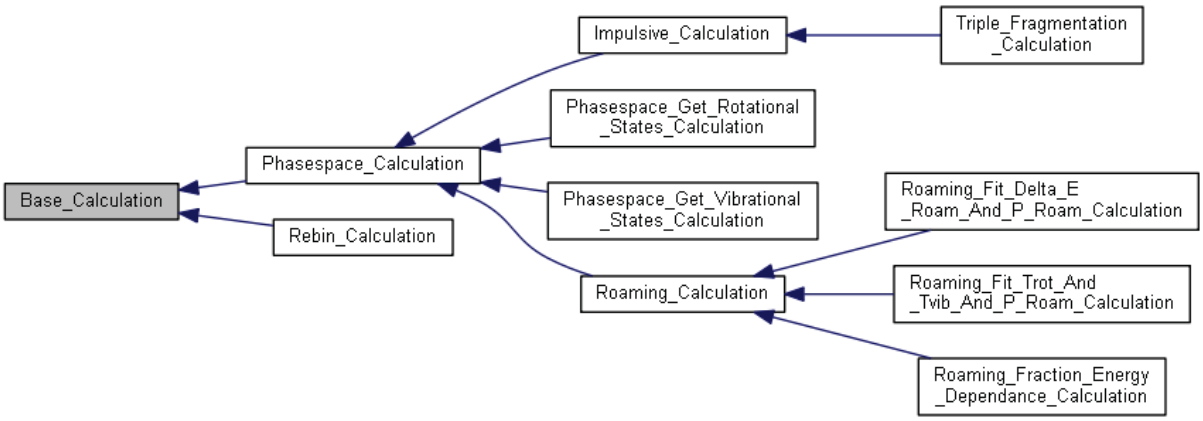

FIGURE B.2: Inheritance Diagram for Base_Calculation

Much of the code relating to molecular fragments is also common. There is an inheritance hierarchy for the classes that encapsulate code relevant to molecular fragments, which is shown in Figure B.3.

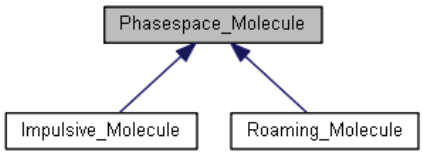

FIGURE B.3: Inheritance Diagram for Phasespace_Molecule

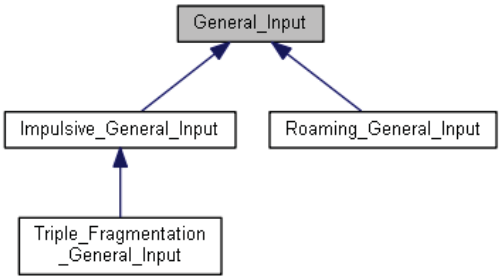

FIGURE B.4: Inheritance Diagram for General_Input

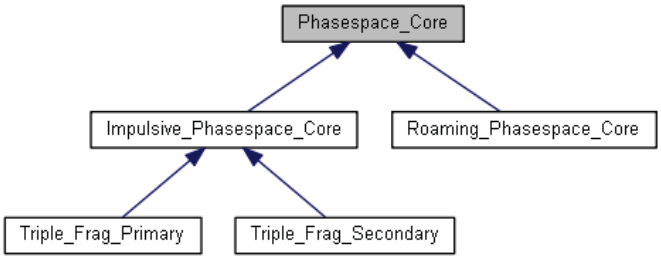

FIGURE B.5: Inheritance Diagram for Phasespace_Core

A similar inheritance hierarchy exists for the classes responsible for encapsulating; the code relating to general, non fragment specific, input data (Figure B.4), the classes that encapsulate the core PST code required to do a single PST state count (Figure B.5), the classes that encapsulate the input data passed into PST core objects (Figure B.6) and the classes that encapsulate the output of the PST core objects (Figure B.7) etc.

B.2 The Calculation Classes

For every input file processed by *PST*$_{Tools}$, a calculation object is created that executes all the code required to perform the type of calculation specified in the file. The creation of this object

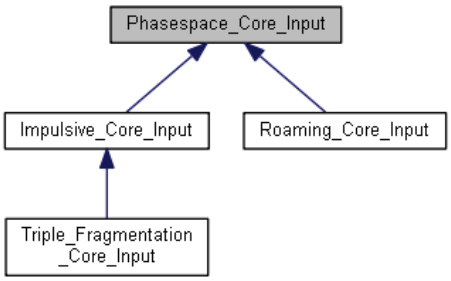

FIGURE B.6: Inheritance Diagram for Phasespace_Core_Input

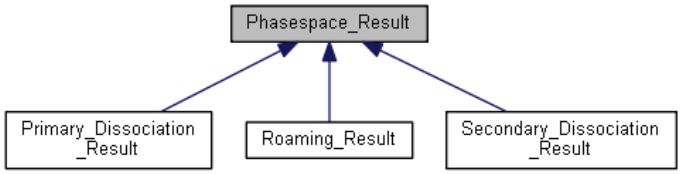

FIGURE B.7: Inheritance Diagram for Phasespace_Result

is represented by the green box in the flowcart in Figure B.1. What happens inside that green box is outlined in the following descriptions of the various *PST*$_{Tools}$ calculation classes.

B.2.1 Base Calculation

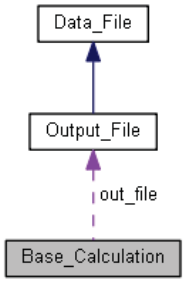

FIGURE B.8: Collaboration Diagram for Base_Calculation. The purple dashed arrow shows that the Base_Calculation class has a member called out_file that is an object of class Output_File. The solid blue arrow shows that the Output_File class is derived from the Data_File class.

Base_Calculation is defined in the file base_calculation.h. The Base_Calculation class is an abstract base class; it defines an interface and not a complete implementation. The base calculation class has a function, run(), that is pure virtual; it has no definition. Therefore, an object of type Base_Calculation cannot be created. Only derivatives of Base_Calculation, that define a

complete implementation of all functions, can be created. All other calculations will be derived from Base_Calculation or one of it's derivatives, see Figure B.2. The main output of the calculation will be written to an Output_File object, that is also a member of the Base_Calculation class, as illustrated in Figure B.8. The Base_Calculation class go() function, which is called to initiate the calculation, takes the input file name as an argument. The go() function sets a timer, calls the initialise routine, calls the run function and, if the run function completes successfully, displays the processing time on the screen and renames the input file so that this successfully completed calculation is not re-run the next time *PST*$_{Tools}$ starts. A flowchart describing the go function is shown in Figure B.9.

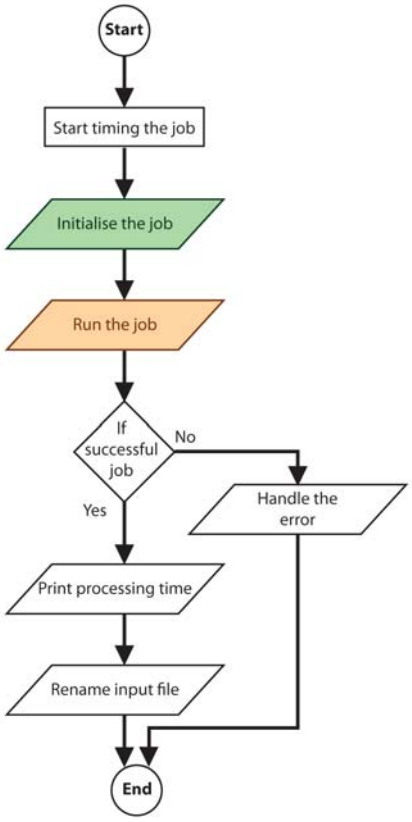

FIGURE B.9: Flowchart for the Base_Calculation class.

There will be many derivatives of this class (and it's derivatives), which will override one or more functions; At the very least the run() function must be overridden, as it is pure virual here. The run function is represented as an orange parallelogram in Figure B.9 and is where Phasespace_Core objects (or derivatives of the Phasespace_Core class) are created to perform

PST counts and the results of these counts are retrieved for use in further calculations and/or output. Initialise, represented as an green parallelogram in Figure B.9, will generally also be overridden to allow additional initialisation, specific to the task of the derived calculation. However, the overridden version of initialise must always call the base implementation first because the base implementation of initialise is where the output directory and main output file are created.

B.2.2 Phasespace Calculation

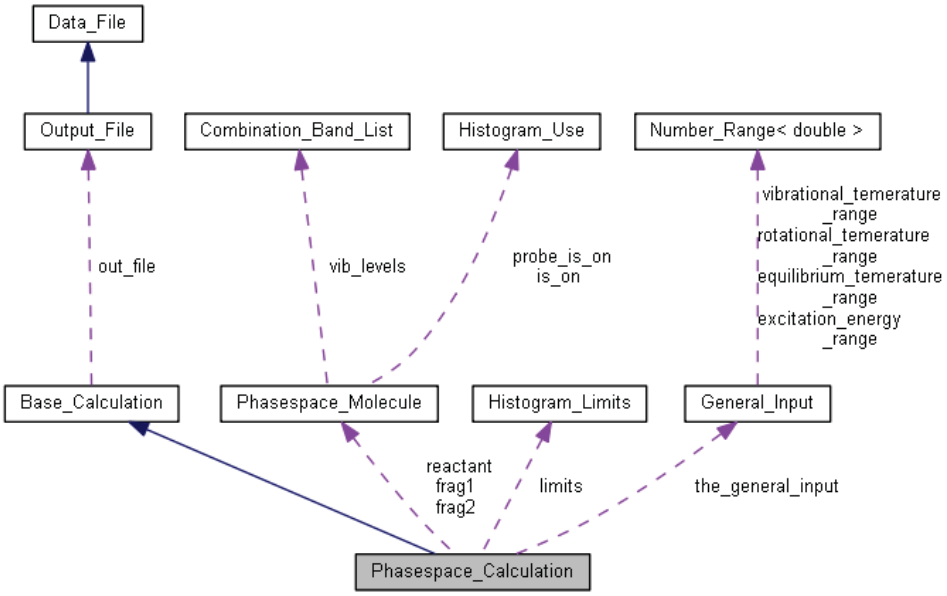

FIGURE B.10: Collaboration Diagram for Phasespace_Calculation

Phasespace_Calculation is defined in the file phasespace_calculation.h. A Phasespace_Calculation object models data for the two product fragments of a barrierless unimolecular dissociation. The Phasespace_Calculation class derives from the Base_Calculation class, as illustrated in Figures B.2 and B.10, and overrides the run() and initialise() functions of Base_Calculation. The collaboration diagram in Figure B.10 shows that in addition to the Output_File member, out_file, inherited from Base_Calculation, the Phasespace_Calculation class adds several new object members. Phasespace_Calculation has members that are General_Input and Phasespace_Molecule type objects, which, for the purposes of this class, parse the required input from the input file and store that data. The creation of those objects and the parsing of the

input file, etc., takes place in the overridden initialise() function. This class facilitates running a calculation using those objects and a Phasespace_Core object that this class will create in its overridden run() function and pass parameters to (including the Phasespace_Molecule objects).

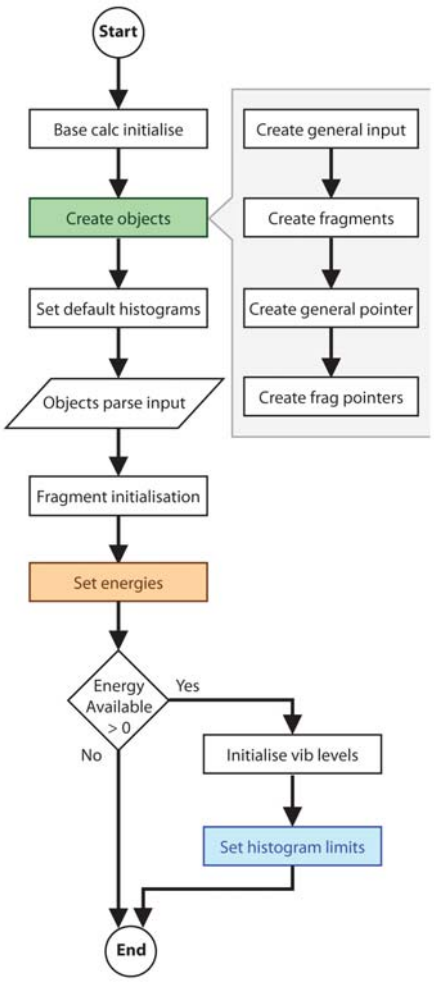

FIGURE B.11: Flowchart for the Phasespace_Calculation implementation of initialise()

The flowchart in Figure B.11 shows the program flow for the Phasespace_Calculation class implementation of the initialise() function, which corresponds to the green parallelogram in Figure B.9.

The flowchart in Figure B.12 shows the program flow for the Phasespace_Calculation class implementation of the run() function, which corresponds to the orange parallelogram in Figure B.9.

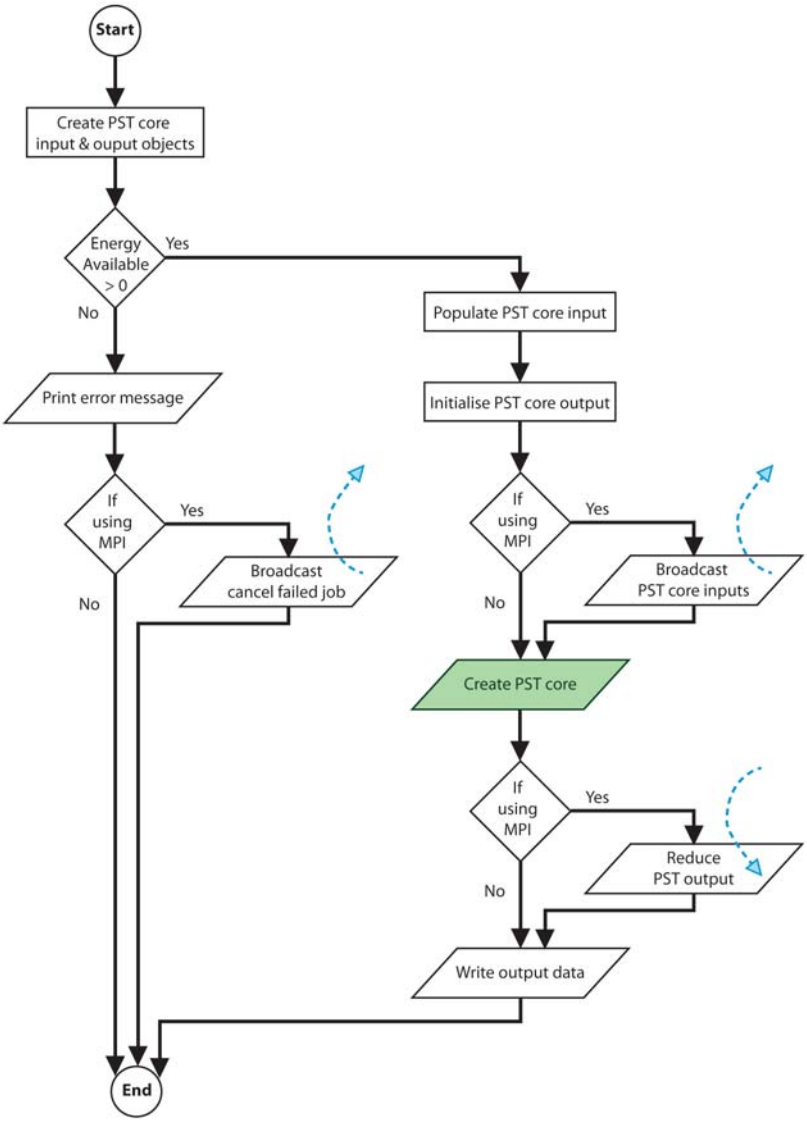

FIGURE B.12: Flowchart for the Phasespace_Calculation implementation of run(). The pale blue dashed arrows represent data flow between this master thread and the slave threads.

The run() and initialise() functions may be overridden in derivative classes. However, the overridden version of initialise must always call the Phasespace_Calculation implementation first. The Phasespace_Calculation implementation of initialise is where the functions that create the General_Input and Phasespace_Molecule type objects are called, after which their parse routines are initiated. set_limits() (blue box) set_energies() (orange box) can also be overridden ... create_objects() (green box) and the functions called within create_objects(), like create_general_input() (inset), are overridden in derived versions of this class when a more derived

General_Input and/or Phasespace_Molecule type object is required by the calculation. The create_general_input_pointer() and create_fragment_pointers() functions are empty here, but create dynamically cast pointers to the derivatives of the Phasespace_Molecule and General_Input classes created in the overridden versions of create_objects() or create_general_input() in derivatives.

create_objects() and create_general_input() may, in the future, be changed to use an "abstract factory pattern" to create the General_Input and Phasespace_Molecule type objects. However, I dont think that will change the special requirements outlined in the following paragraph because this base class will still be defined with pointers to the base General_Input and Phasespace_Molecule type objects.

Because the the_general_input, reactant, frag1 and frag2 members are pointers to their base classes, if they end up pointing to objects of more derived versions of their respective class types, the overridden versions of initialise and run() will need to dynamically cast them down to their derived class type, to avoid object slicing; to be able to access the functionality added to the derived versions. This dynamic_cast should always be safe because those member objects will have been created by the derived version of this class and so will always be of the correct type expected by that same derived version of this class. pointers to fulfill this requirement have been added as members of all the derived versions of this and the core PST classes.

Derivatives of the Phasespace_Core class, which also inherit members that are pointers to the base Phasespace_Molecule class, will also have to dynamically cast those pointers when their methods use features specific to the Phasespace_Molecule derivative. This should also always be safe because such Phasespace_Core derivatives will only be created by Phasespace_Calculation derivatives that create that same Phasespace_Molecule derivative. The Phasespace_Molecule derivatives created in the Phasespace_Calculation derivative are passed to the Phasespace_Core derivative. Therefore the dynamic cast will always be on a pointer to the correct Phasespace_Molecule derivative.

This design has been chosen becuase we do not want the interfaces of all the (potentially many) derived classes, designed for specific niche calculations, percolating up to the base classes. We

want to be able to present a subset of our code that performs one niche type of PST calculation, without revealing all of the interfaces required for other niche calculations that may be completely unrelated. At the same time, we do not want to duplicate code that will be common to all, or a subset of the calculation types etc. We recognise that dynamic_cast is potentially an indication of poor object oriented design. However, our (perhaps naive) concern is that the complexity required to achieve the above goals without using dynamic_cast (carefully) will make this code less accessible to people who are not familiar with object oriented design patterns.

This having been said; anyone extending this code should pay attention NOT to pass an object of the wrong type to their Phasespace_Core derivatives, lest they suffer the wrath of a dynamic cast gone wrong!

B.2.3 Impulsive Calculation

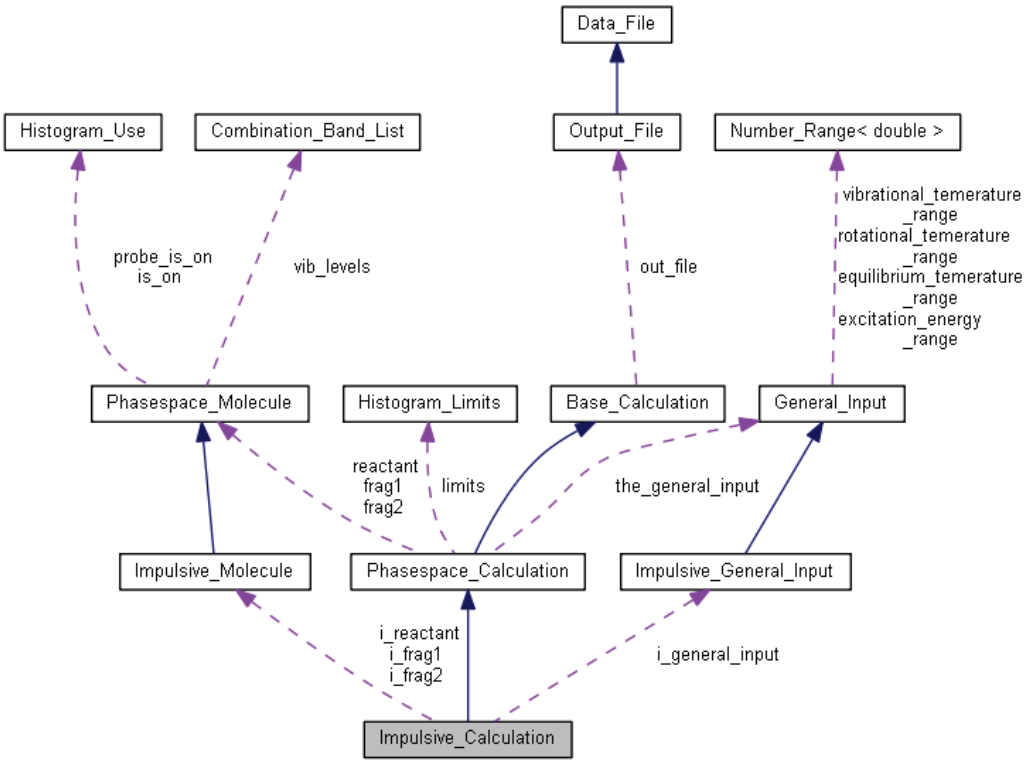

FIGURE B.13: Collaboration Diagram for Impulsive_Calculation

Impulsive_Calculation is defined in the file impulsive_calculation.h. An Impulsive_Calculation object models data for the two product fragments of a unimolecular dissociation with a barrier in the exit channel. The Impulsive_Calculation class derives from the Phasespace_Calculation class, as illustrated in Figures B.2 and B.13, and overrides the run(), initialise(), set_energies(), set_limits(), create_general_input(), create_general_input_pointer(), create_fragments() and create_fragment_pointers() functions of Phasespace_Calculation. The initialise() function is only initialised to explicitly call the Phasespace_Calculation implementation of initialise(). This is done so that any classes deriving from Impulsive_Calculation, that do need to add to the functionality of initialise(), can still follow the simple convention of calling the parent implementation first. The collaboration diagram in Figure B.13 shows that in addition to the members inherited from Phasespace_Calculation, the Impulsive_Calculation class adds 4 new object members.

The 4 new members, i_reactant, i_frag1, i_frag2 and i_general_input, are pointers to the derived versions of Phasespace_Molecule and General_Input that add logic specific to modelling impulsive reactions, see Figures B.3 and B.4. Because Impulsive_Calculation inherits members that are pointers to the base Phasespace_Molecule (reactant, frag1 and frag2) and General_Input (the_general_input) classes, one must dynamically cast those pointers to avoid object slicing and get access to the features specific to the derivative. The dynamically cast pointers are created in create_fragment_pointers() and create_general_input_pointer() and stored in i_reactant, i_frag1, i_frag2 and i_general_input. These pointers are used to access the objects throughout Impulsive_Calculation.

The general flow of run() for Impulsive_Calculation is the same as for Phasespace_Calculation, see Figure B.12. The significant difference is that an Impulsive_Core object is created rather than a Phasespace_Core object, and, that the fragments and input objects are also derivatives with additional logic for the impulsive component of the calculation.

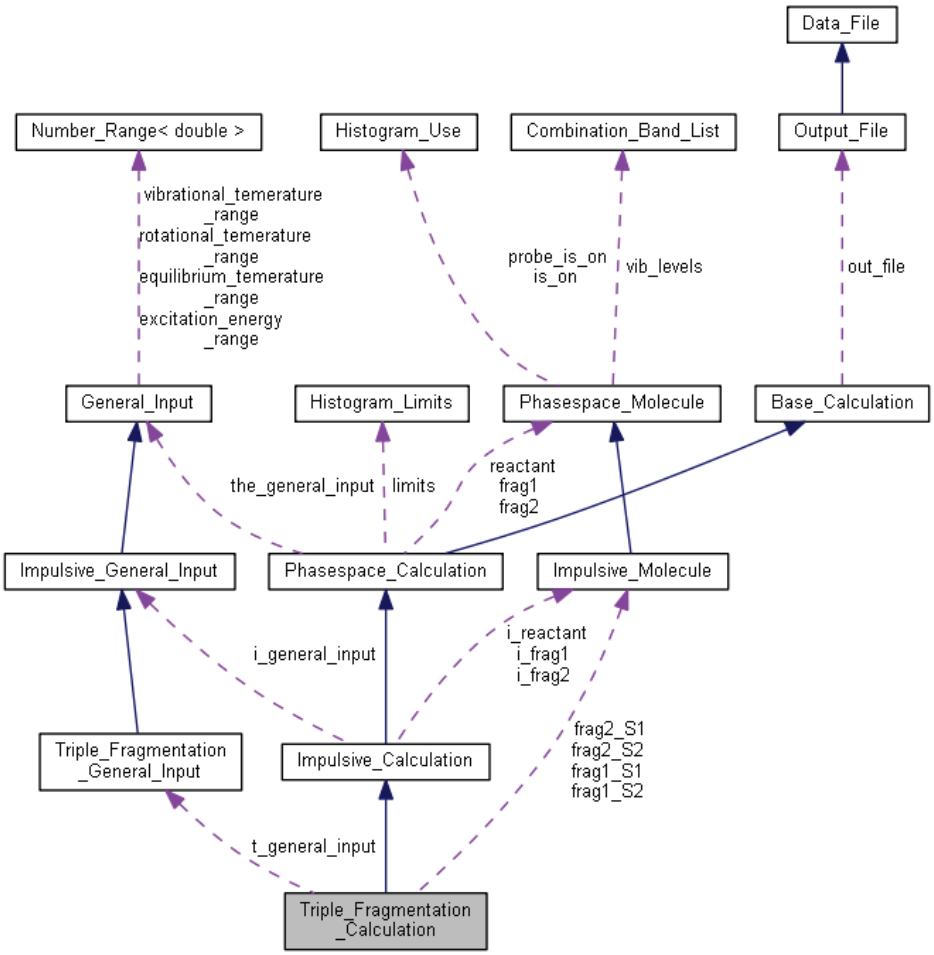

FIGURE B.14: Collaboration Diagram for Triple_Fragmentation_Calculation

B.2.4 Triple Fragmentation Calculation

Triple_Fragmentation_Calculation is defined in the file triple_fragmentation_calculation.h. A Triple_Fragmentation_Calculation object models data for the product fragments of stepwise unimolecular triple fragmentation. The Triple_Fragmentation_Calculation class derives from the Impulsive_Calculation class, as illustrated in Figures B.2 and B.14, and overrides create_general_input(), create_general_input_pointer(), create_fragments(), initialise() and run(). The collaboration diagram in Figure B.14 shows that in addition to the members inherited from Impulsive_Calculation, the Triple_Fragmentation_Calculation class adds 5 new object members; frag1_S1, frag1_S2, frag2_S1, frag2_S2 and t_general_input. In create_general_input_pointer(),

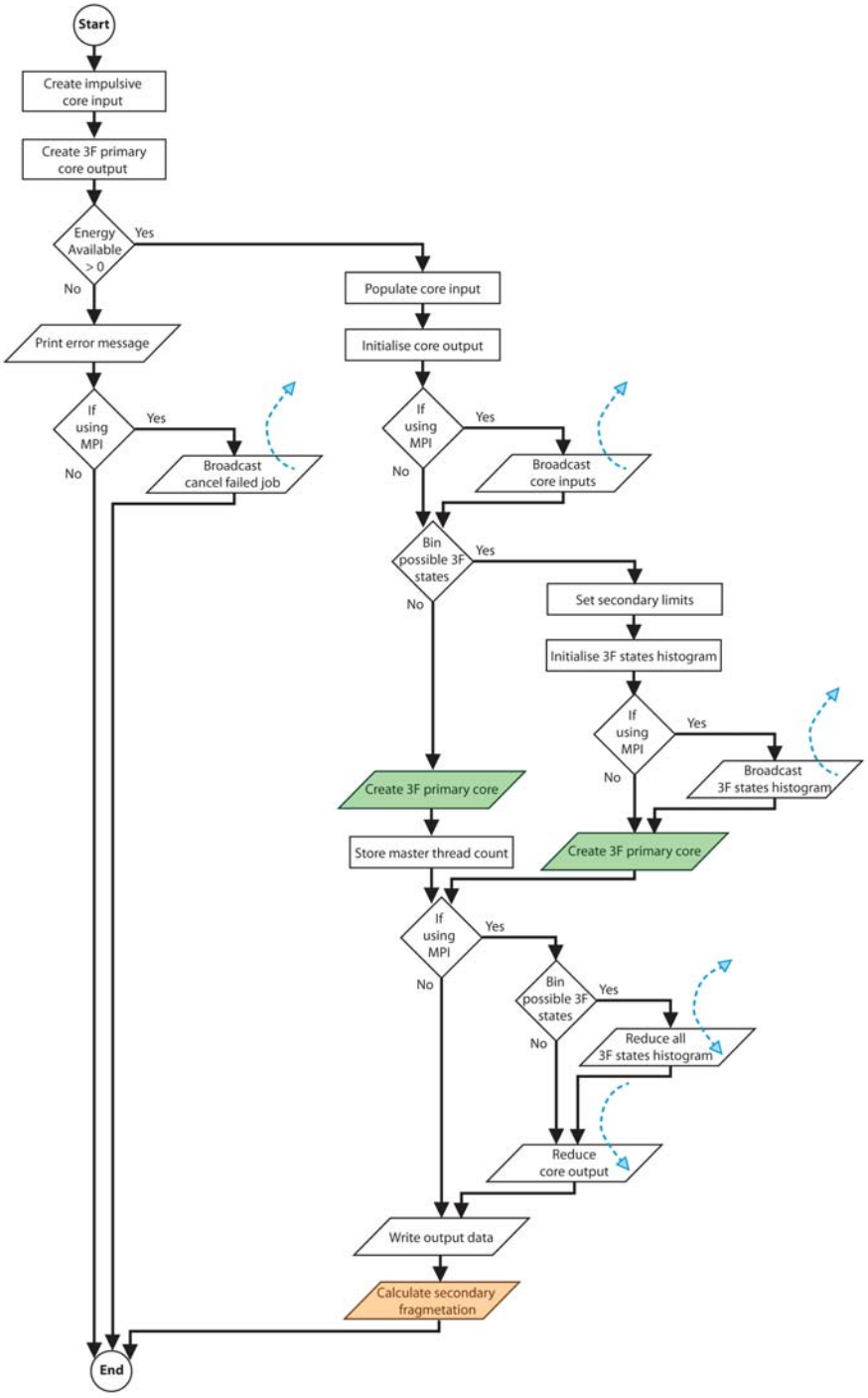

FIGURE B.15: Flowchart for the Triple_Fragmentation_Calculation implementation of run(). The pale blue dashed arrows represent data flow between this master thread and the slave threads.

t_general_input is dynamically cast to point to the Triple_Fragmentation_General_Input object created in create_general_input(). In the Triple_Fragmentation_Calculation implementation of create_fragments(), after calling the Impulsive_Calculation implementation of create_fragments() to create the reactant and primary dissociation product fragments, the four

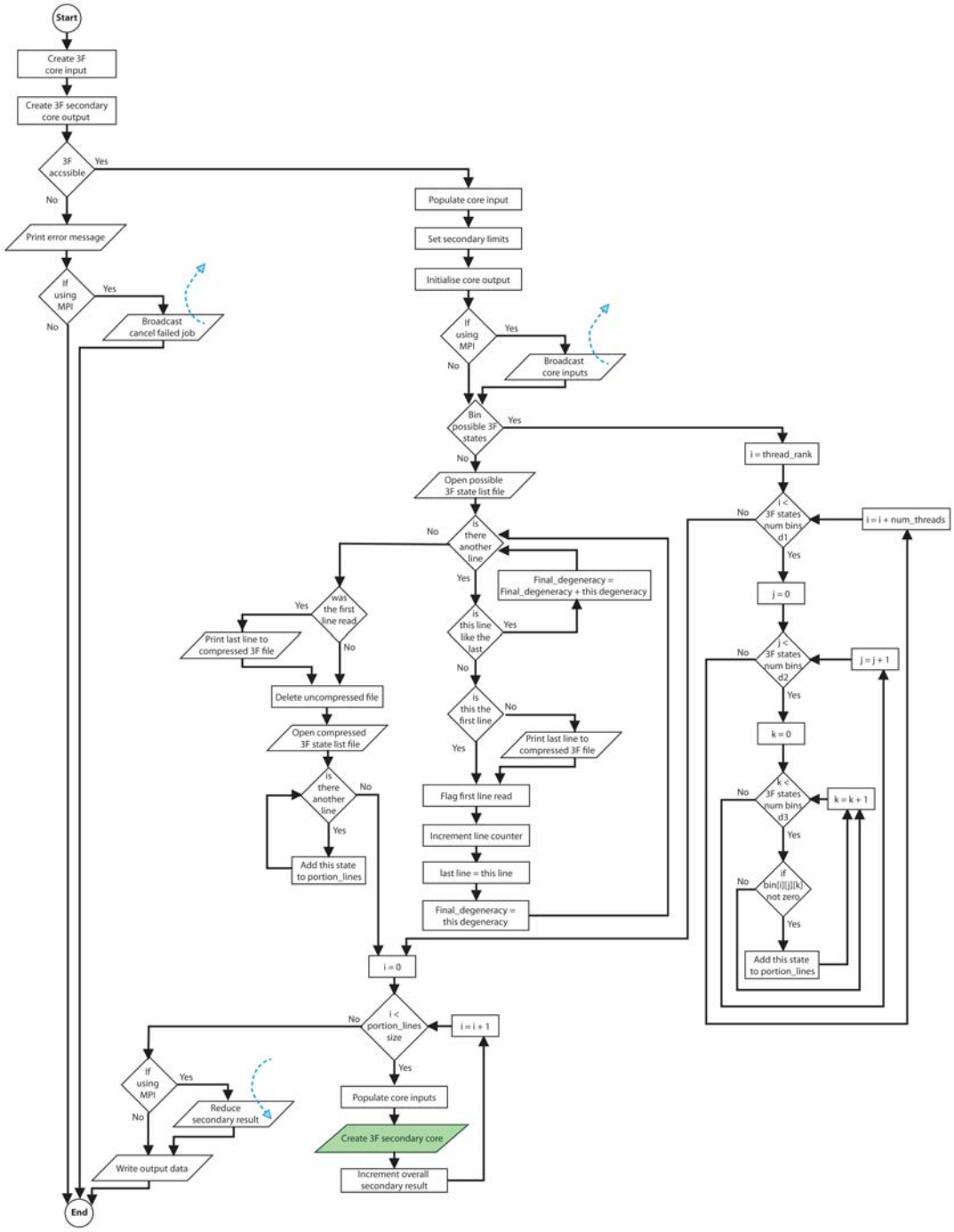

FIGURE B.16: Flowchart for the Triple_Fragmentation_Calculation implementation of secondary_fragmentation(). The pale blue dashed arrows represent data flow between this master thread and the slave threads.

possible secondary dissociation product fragments are created, frag1_S1, frag1_S2, frag2_S1 and frag2_S2. The Triple_Fragmentation_Calculation implementation of initialise() first calls

the Impulsive_Calculation implementation of initialise() and then passes the input file name to frag1_S1, frag1_S2, frag2_S1 and frag2_S2, so that they can attempt to parse any relevant data from the file.

The flowchart in Figure B.15 shows the program flow for the Triple_Fragmentation_Calculation class implementation of the run() function, which corresponds to the orange parallelogram in Figure B.9. There are two ways this calculation can store the potentially very large number of states of the product fragment of the primary dissociation that will undergo secondary dissociation. The simplest method writes each state out to a text file and the other method involves binning the three required values for each state, internal energy, angular momentum and velocity, in a three dimensional histogram (the 3F states histogram). The 3F states histogram method is recommended because it can significantly reduce the total number of secondary PST calculations required without significantly changing the result. In a MPI calculation the simple text file method simply requires each thread to write its own text file to the local HDD of the node, while the 3F states histogram is partially populated by each thread and reduced to all threads after the primary state count has completed.

The flowchart in Figure B.16 shows the program flow for the secondary_fragmentation() function, which corresponds to the orange parallelogram in Figure B.15. The secondary_fragmentation() function takes pointers to the 3F primary phasespace core output object, the molecule object representing the product fragment of the primary dissociation that undergoes secondary dissociation, the two molecule objects representing the products of the secondary dissociation and the 3F states histogram as arguments. It is called only once, passing the appropriate molecule objects. If the first primary product fragment (i_frag1) has a dissociation energy defined in the input file, then the program will pass i_frag1, frag1_S1 and frag1_S2 to the secondary_fragmentation() function. Otherwise, if the second primary product fragment (i_frag2) has a dissociation energy defined in the input file, then the program will pass i_frag2, frag2_S1 and frag2_S2 to the secondary_fragmentation() function.

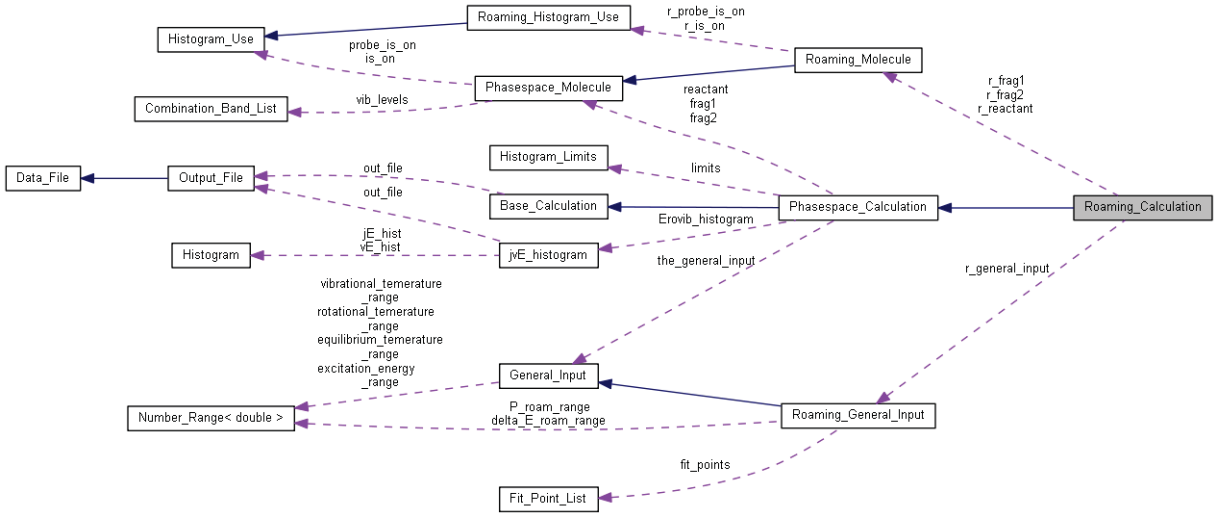

FIGURE B.17: Collaboration Diagram for Roaming_Calculation

B.2.5 Roaming Calculation

Roaming_Calculation is defined in the file roaming_calculation.h. Roaming_Calculation will eventually be used to perform a PST calculation, at a single photolysis energy, to produce product state distributions for products of both the barrierless bond breaking and roaming pathways. There is still a fair bit of work to be done on this calculation before that goal is reached. However, there are three fully functioning calculation classes that derive from Roaming_Calculation. There are several components that are common to all roaming calculations. These are implemented in Roaming_Calculation and inherited by the derivatives Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation,
Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation and
Roaming_Fraction_Energy_Dependance_Calculation.

The Roaming_Calculation class derives from the Phasespace_Calculation class, as illustrated in Figures B.2 and B.17, and overrides the run(), initialise(), set_energies(), create_general_input(), create_general_input_pointer(), create_fragments() and create_fragment_pointers() functions of Phasespace_Calculation. The collaboration diagram in Figure B.17 shows that in addition to the members inherited from Phasespace_Calculation, the Roaming_Calculation class adds 4 new object members.

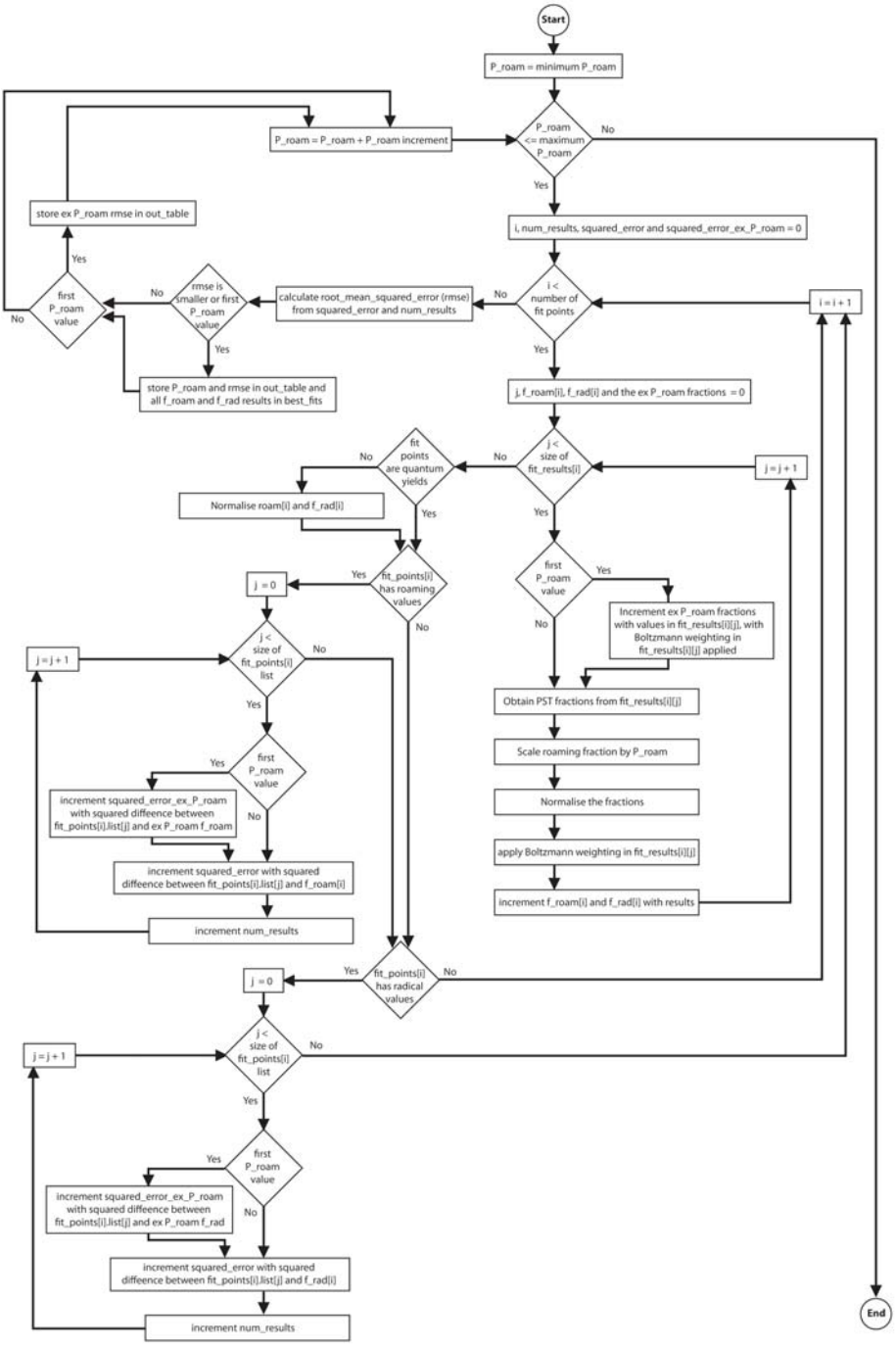

FIGURE B.18: Flowchart for the selectBestProam() function implemented in Roaming_Calculation

The 4 new members, r_reactant, r_frag1, r_frag2 and r_general_input, are pointers to the derived versions of Phasespace_Molecule and General_Input that add logic specific to modelling

roaming reactions, see Figures B.3 and B.4. Because Roaming_Calculation inherits members that are pointers to the base Phasespace_Molecule (reactant, frag1 and frag2) and General_Input (the_general_input) classes, one must dynamically cast those pointers to avoid object slicing and get access to the features specific to the derivative. The dynamically cast pointers are created in create_fragment_pointers() and create_general_input_pointer() and stored in r_reactant, r_frag1, r_frag2 and r_general_input. These pointers are used to access the objects throughout Roaming_Calculation.

The Roaming_Calculation class provides a function, selectBestProam(), that is used by the derivatives Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation and Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation calculation classes. The program flow for selectBestProam() is illustrated in Figure B.18. It applies each P_{roam} value, within the range specified in the input file, to the PST results for each fit point energy, defined in the input file, and stores the minimum overall RMSE value and the corresponding P_{roam} value, for all fit point roaming fractions defined in the input file.

B.2.6 Roaming Fraction Energy Dependance Calculation

Roaming_Fraction_Energy_Dependance_Calculation is defined in the file roaming_fraction_energy_dependance_calculation.h. A Roaming_Fraction_Energy_Dependance_Calculation object calculates the branching fraction and, when a non-zero reactant temperature is assigned, quantum yield for a bond fission channel and its associated roaming pathway, over a specified energy range, for a given reactant and products. The Roaming_Fraction_Energy_Dependance_Calculation class derives from the Roaming_Calculation class, as illustrated in Figure B.2, and overrides the run() function. The flowchart in Figure B.19 shows the program flow for the Roaming_Fraction_Energy_Dependance_Calculation class implementation of the run() function, which corresponds to the orange parallelogram in Figure B.9.

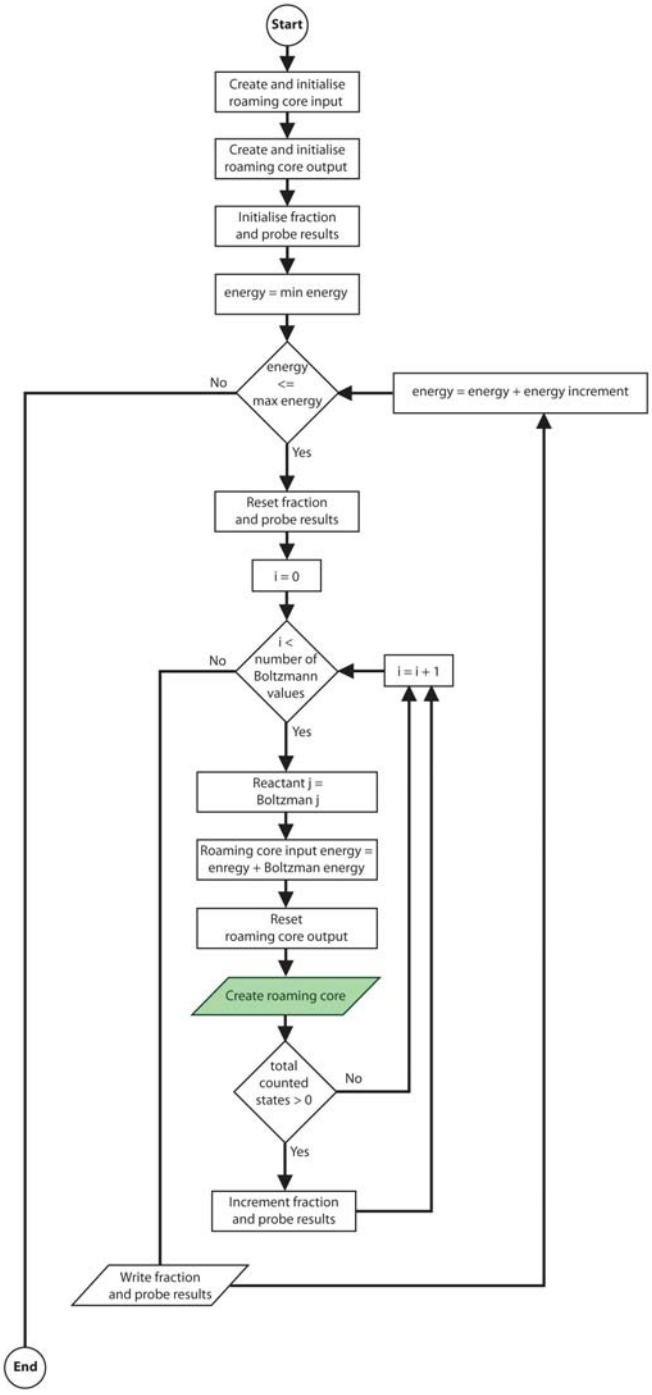

FIGURE B.19: Flowchart for the Roaming_Fraction_Energy_Dependance_Calculation implementation of run().

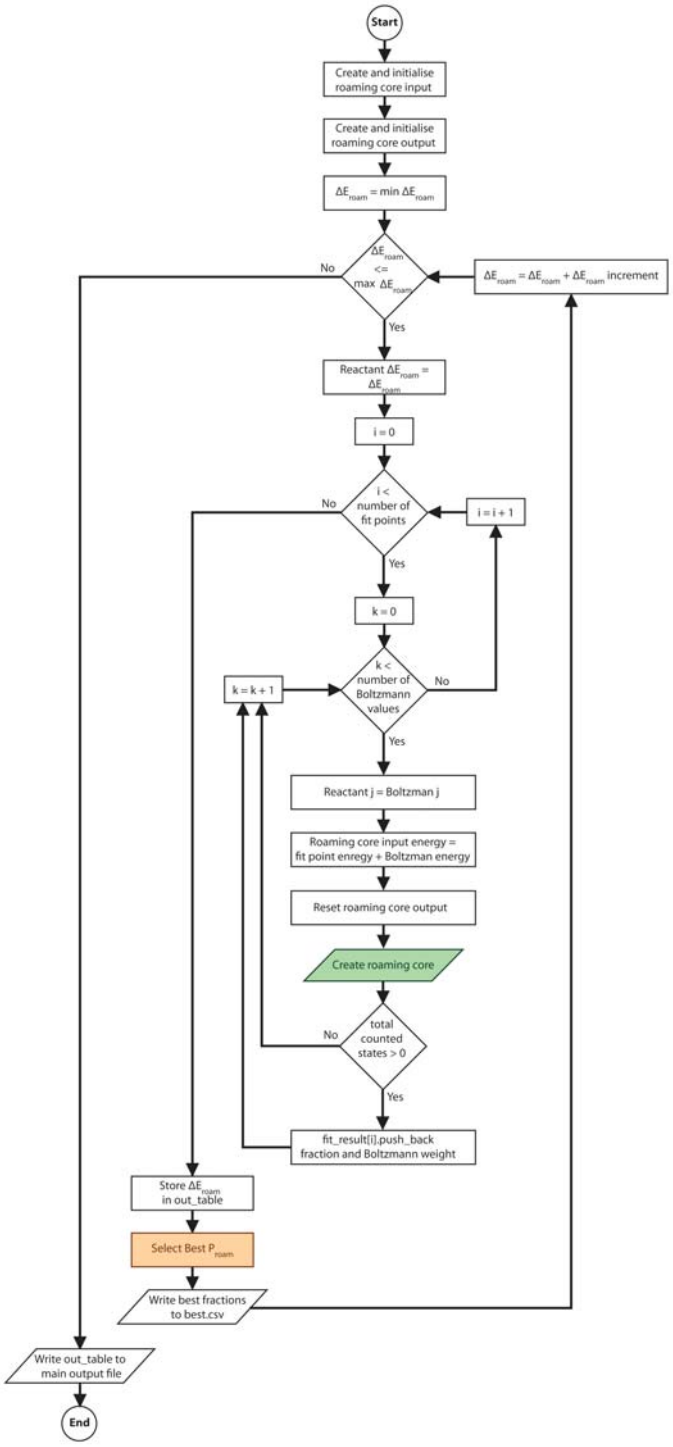

FIGURE B.20: Flowchart for the Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation implementation of run().

B.2.7 Roaming Fit ΔE_{roam} And P_{roam} Calculation

Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation is defined in the file roaming_fit_delta_E_roam_and_P_roam_calculation.h. A

Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation object calculates the branching fraction and, when a non-zero reactant temperature is assigned, quantum yield for a bond fission channel and its associated roaming pathway, over a specified ΔE_{roam} range, for a given reactant and products, at the energies defined by a set of experimental fit points defined in the input file. For each ΔE_{roam} PST result the best P_{roam} value is obtained, using the selectBestProam() function described in Section B.2.5 and Figure B.18. The output of this calculation comprises two files. The main output file includes a line for each ΔE_{roam} value, with the value of P_{roam} that best fits the fit points in the input file, the RMSE obtained for that P_{roam} value and the RMSE value (RMSE_EX_PROAM) where no P_{roam} scaling factor is applied (IE a P_{roam} value of 1). The best overall fit can be obtained by loading these results in excel and sorting them in ascending order by the RMSE column. The second output file, best.csv, writes out the actual branching fractions obtained at each fit point energy, for each ΔE_{roam} value, when the value of P_{roam} provides the best RMSE for all of the fit points in the input file, which is the same P_{roam} value specified in the main output file.

The Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation class derives from the Roaming_Calculation class, as illustrated in Figure B.2, and overrides the run() and set_energies() functions. The flowchart in Figure B.20 shows the program flow for the Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation class implementation of the run() function, which corresponds to the orange parallelogram in Figure B.9. The orange rectangle in Figure B.20 corresponds to the selectBestProam() function described in Section B.2.5 and Figure B.18.

B.2.8 Roaming Fit T_{rot} And T_{vib} And P_{roam} Calculation

Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation is defined in the file roaming_fit_trot_and_tvib_and_p_roam_calculation.h. A Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation object calculates the branching fraction and, when a non-zero reactant temperature is assigned, quantum yield for a bond fission channel and its associated roaming pathway, over specified T_{rot} and T_{vib} ranges, for a given reactant and products, at the energies defined by a set of experimental fit points defined in

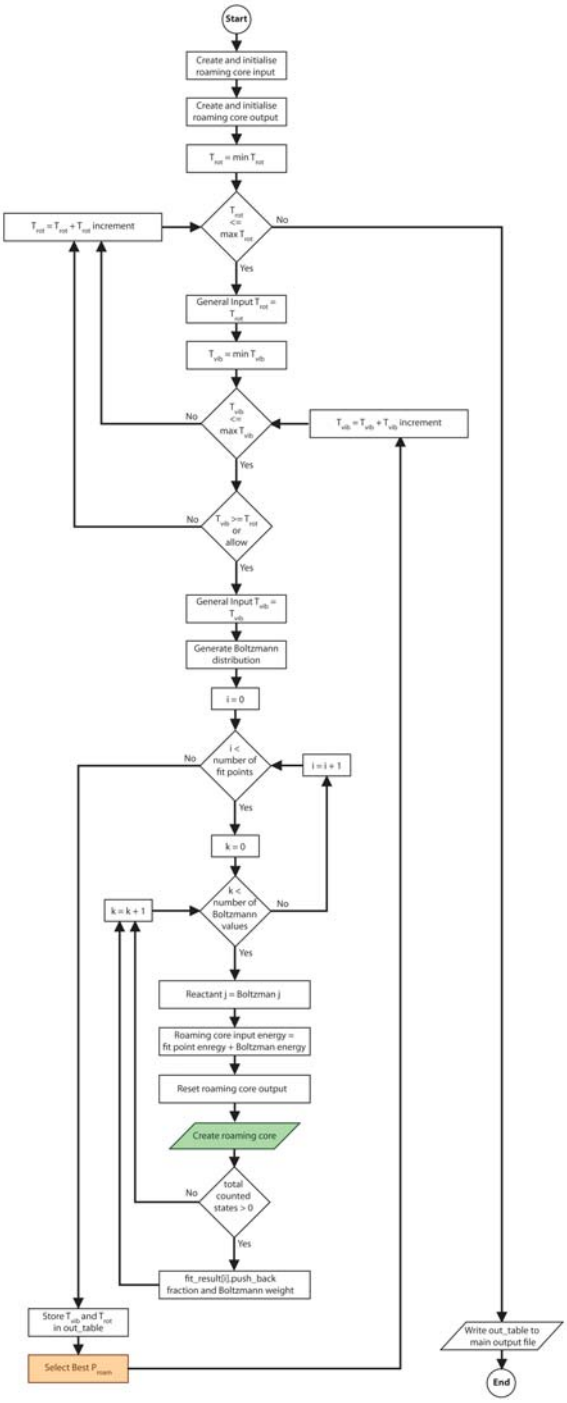

FIGURE B.21: Flowchart for the Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation implementation of run().

the input file. For each T_{rot} and T_{vib} PST result the best P_{roam} value is obtained, using the selectBestProam() function described in Section B.2.5 and Figure B.18. The output file includes a

line for each T_{rot} and T_{vib} combination, with the value of P_{roam} that best fits the fit points in the input file, the RMSE obtained for that P_{roam} value and the RMSE value (RMSE_EX_PROAM) where no P_{roam} scaling factor is applied (IE a P_{roam} value of 1). The best overall fit can be obtained by loading these results in excel and sorting them in ascending order by the RMSE column.

The Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation class derives from the Roaming_Calculation class, as illustrated in Figure B.2, and overrides the run() and jvE_initialise() functions. The flowchart in Figure B.21 shows the program flow for the Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation class implementation of the run() function, which corresponds to the orange parallelogram in Figure B.9. The orange rectangle in Figure B.21 corresponds to the selectBestProam() function described in Section B.2.5 and Figure B.18.

B.3 The Molecule Classes

PSTTools has an inheritance hierarchy for the classes that encapsulate code and data relevant to molecular fragments, which is shown in Figure B.3. An instance of these classes will be created for the reactant and any product fragments involved in each calculation.

B.3.1 Phasespace Molecule

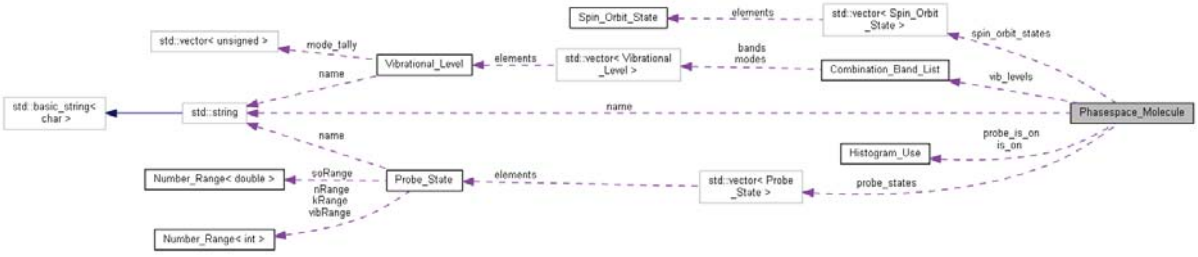

FIGURE B.22: Collaboration Diagram for Phasespace_Molecule

Phasespace_Molecule, Combination_Band_List, Vibrational_Level and Spin_Orbit_State are defined in the file phasespace_molecule.h. Probe_State and Histogram_Use are defined in the

files phasespace_probe.h and phasespace_result_data.h respectively. Phasespace_Molecule is the base molecule class, as illustrated in Figure B.3. The Phasespace_Molecule objects are created and used by the Phasespace_Calculation class and passed as constants to the constructor of the core PST state count class, Phasespace_Core. This class stores various data relating to a molecule or fragment. It has a number of functions that calculate values related to the fragment, which are used by Phasespace_Core and elsewhere, such as in the run functions of some calculation class derivatives. Phasespace_Molecule also has some functions that write output specific to the fragment they represent to a file. Each instance of this class knows how to parse an input file and populate all of the data in the file that is relevant to the fragment they represent. The logic for parsing the input file is in the parse_input() function.

The collaboration diagram in Figure B.22 shows that Phasespace_Molecule has several objects as members. The spin_orbit_states member stores a list of energies and angular momenta for all the spin obit coupling states that the fragment can occupy. The vib_levels member is an instance of the Combination_Band_List class, which contains two lists of Vibrational_Level objects, bands and modes. The modes list contains a Vibrational_Level object for each normal mode of the fragment, which are read from the input file. The bands list contains a Vibrational_Level object for each possible combination band for the fragment within the available energy. The bands list is generated from the modes list by the Combination_Band_List object, vib_levels, recursively, using the harmonic oscillator approximation.

The probe_states member of Phasespace_Molecule is a list of Probe_State objects, which define ranges for vibrational and rotational energy levels to probe, for a specified product fragment, within the total result. The Probe_State class also has provisions for probing specific spin orbit coupling states, however, this has not yet been fully implemented. The list of states to be probed is read from the input file and separate histograms are produced for each probe, with values only corresponding to states that fall within the ranges specified by the probe. Phasespace_Molecule has two Histogram_Use pointer members, is_on and probe_is_on. The is_on object stores flags that define which histograms are to be populated and output during the calculation and the probe_is_on object does the same for the histograms of any specific probed states. Default values for is_on and probe_is_on will be set by the calculation object that has

created the molecule object and these may be overridden by specifying which histograms are on and off in the input file.

Because the molecule classes in *PST*Tools are designed to be polymorphic, that is, derived objects will be created using base class pointers, the molecule classes doesn't use the constructor for initialisation and, instead, has a function construct(), which is called immediately after the object has been created. Derived versions of construct() will call the parent implementation first, so that initialisation routines for the inherited members are also executed. The reason this cannot be implemented using the constructors, is that the is_on and probe_is_on members are created in a function, create_histogram_use(), which is called in the Phasespace_Molecule implementation of construct() and is overridden in the Roaming_Molecule class to create derived Roaming_Histogram_Use versions of the Histogram_Use class required for roaming calculations. If create_histogram_use() is called from the constructor of Phasespace_Molecule, the Phasespace_Molecule implementation of create_histogram_use() will always be called and Histogram_Use objects will always be created. If the molecule being constructed is, in fact, a Roaming_Molecule, the Phasespace_Molecule constructor is still called before the Roaming_Molecule constructor, so the overridden implementation of create_histogram_use() would not yet exist, resulting in the wrong type of Histogram_Use class being created at is_on and probe_is_on. Having the initialisation routine called after the whole object has been created allows the correct overridden functions to be called. Another function that is necessary because of the polymorphic nature of the molecule classes is the clone() function. The clone function is overridden in all derived classes and is used instead of the copy constructor, so that an object of the correct derived type is created.

When MPI is used, molecule objects have to be sent between nodes. *PST*Tools makes use of the Boost.MPI library, which uses the Boost.Serialization library to serialise an object before sending is across the network. In order for this to work a class must define the boost::serialization::access class as a friend class and implement the serialize() function. The serialize() function defines all the members that are to be serialised. Any objects members that are to be serialised, such as vib_levels, spin_orbit_states, probe_states,is_on and probe_is_on must have the serialize() function defined in their class definitions too.

B.3.2 Impulsive Molecule

Impulsive_Molecule is defined in the file impulsive_molecule.h. The Impulsive_Molecule objects are created and used by the Impulsive_Calculation and Triple_Fragmentation_Calculation classes and passed as constants to the constructors of their respective PST cores. The Impulsive_Molecule class derives from the Phasespace_Molecule class, as illustrated in Figure B.3, and overrides construct(), clone(), serialize() and parse_input(). No collaboration diagram is presented here because there are no additional object members, only fundamental C++ data type members and functions.

B.3.3 Roaming Molecule

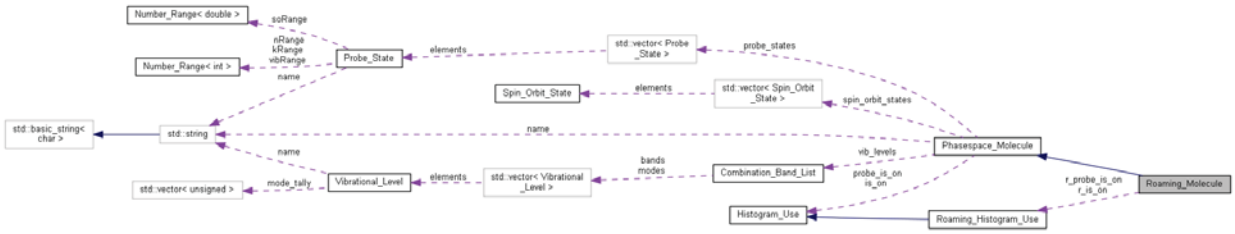

FIGURE B.23: Collaboration Diagram for Roaming_Molecule

Roaming_Molecule and Roaming_Histogram_Use are defined in the files roaming_molecule.h and roaming_result_data.h respectively. The Roaming_Molecule objects are created and used by the Roaming_Calculation and its derivatives and passed as constants to the Roaming_Phasespace_Core constructor. The Roaming_Molecule class derives from Phasespace_Molecule, as illustrated in Figures B.3 and overrides construct(), clone(), serialize(), create_histogram_use() and parse_input(). The collaboration diagram in Figure B.23 shows that Roaming_Molecule adds two new Roaming_Histogram_Use pointer members, r_is_on and r_probe_is_on. Because Roaming_Molecule inherits members that are pointers to the base Histogram_Use class (is_on and probe_is_on), one must dynamically cast those pointers to avoid object slicing and get access to the features specific to the derivative, Roaming_Histogram_Use. The dynamically cast pointers are created in create_histogram_use() and stored in r_is_on and r_probe_is_on. These pointers are used to detect which histograms are active throughout Roaming_Calculation and its derivatives.

B.4 The General Input Classes

PST$_{Tools}$ has an inheritance hierarchy for the classes that encapsulate code relevant to general input parameters, which is shown in Figure B.4. A single instance of one of these classes will be created in each calculation, that will parse and store any input parameters that are not related to specific molecules or fragments.

B.4.1 General Input

General_Input is defined in the file phasespace_input.h. General_Input is the base general input class, as illustrated in Figure B.4. The General_Input object is created and used by the Phasespace_Calculation class. The parse_input() function of the General_Input class parses the input file. This function parses general input data that is common to all calculations. Therefore, all derivatives of this class must call the parent implementation of parse_input() in their overridden parse_input() implementation.

B.4.2 Impulsive General Input

Impulsive_General_Input is defined in the file impulsive_input.h. The Impulsive_General_Input object is created and used by the Impulsive_Calculation class. The Impulsive_General_Input class derives from the General_Input class, as illustrated in Figure B.4, and overrides parse_input(). The Impulsive_General_Input implementation of parse_input() first calls the General_Input implementation and then parses the general input data that is specific to impulsive calculations.

B.4.3 Triple Fragmentation General Input

Triple_Fragmentation_General_Input is defined in the file triple_fragmentation_input.h. The Triple_Fragmentation_General_Input object is created and used by the Triple_Fragmentation_Calculation class. The Triple_Fragmentation_General_Input class derives from the Impulsive_General_Input class, as illustrated in Figure B.4, and overrides parse_input().

The Triple_Fragmentation_General_Input implementation of parse_input() first calls the Impulsive_General_Input implementation and then parses the general input data that is specific to triple fragmentation calculations.

B.4.4 Roaming General Input

Roaming_General_Input is defined in the file roaming_input.h. The Roaming_General_Input object is created and used by the Roaming_Calculation class. The Roaming_General_Input class derives from the General_Input class, as illustrated in Figure B.4, and overrides parse_input(). The Roaming_General_Input implementation of parse_input() first calls the General_Input implementation and then parses the general input data that is specific to roaming calculations.

B.5 The Phasespace Core Input Classes

PST$_{Tools}$ has an inheritance hierarchy for the classes that contain the general input parameters for the core PST state count classes, which is shown in Figure B.6. These core input classes can be distinguished from the general input classes. The general input classes above contain general input data for an entire calculation, which may involve many individual PST state counts, over ranges of input values. In contrast, the phasespace core input classes contain general input parameters required by a phasespace core to perform a single PST state count. Therefore there are only single values for each input parameter in a phasespace core input. A calculation that performs PST state counts over a range of input values will use loops to incrementally set those input parameter values in a core input object and then pass it to a new phasespace core object to obtain each PST state count result.

B.5.1 Phasespace Core Input

Phasespace_Core_Input is defined in the file phasespace_input.h. Phasespace_Core_Input is the base core input class, as illustrated in Figure B.6. The Phasespace_Core_Input object is

created and populated by the Phasespace_Calculation class and passed as a constant to the constructor of the Phasespace_Core class.

B.5.2 Impulsive Core Input

Impulsive_Core_Input is defined in the file impulsive_input.h. The Impulsive_Core_Input object is created and populated by the Impulsive_Calculation and Triple_Fragmentation_Calculation classes and passed as a constant to the constructors of the Phasespace_Core and Triple_Frag_Primary classes. Impulsive_Core_Input derives from the Phasespace_Core_Input class, as illustrated in Figure B.6.

B.5.3 Triple Fragmentation Core Input

Triple_Fragmentation_Core_Input is defined in the file triple_fragmentation_input.h. The Triple_Fragmentation_Core_Input object is created and populated by the Triple_Fragmentation_Calculation class and passed as a constant to the constructor of the Triple_Frag_Secondary class. Triple_Fragmentation_Core_Input derives from the Impulsive_Core_Input class, as illustrated in Figure B.6.

B.5.4 Roaming Core Input

Roaming_Core_Input is defined in the file roaming_input.h. The Roaming_Core_Input object is created and populated by the Roaming_Calculation class and passed as a constant to the constructor of the Roaming_Phasespace_Core class. Roaming_Core_Input derives from the Phasespace_Core_Input class, as illustrated in Figure B.6.

B.6 The Phasespace Result Classes

PST$_{Tools}$ has an inheritance hierarchy for the classes that contain the output from the core PST state count classes, which is shown in Figure B.7.

B.6.1 Phasespace Result

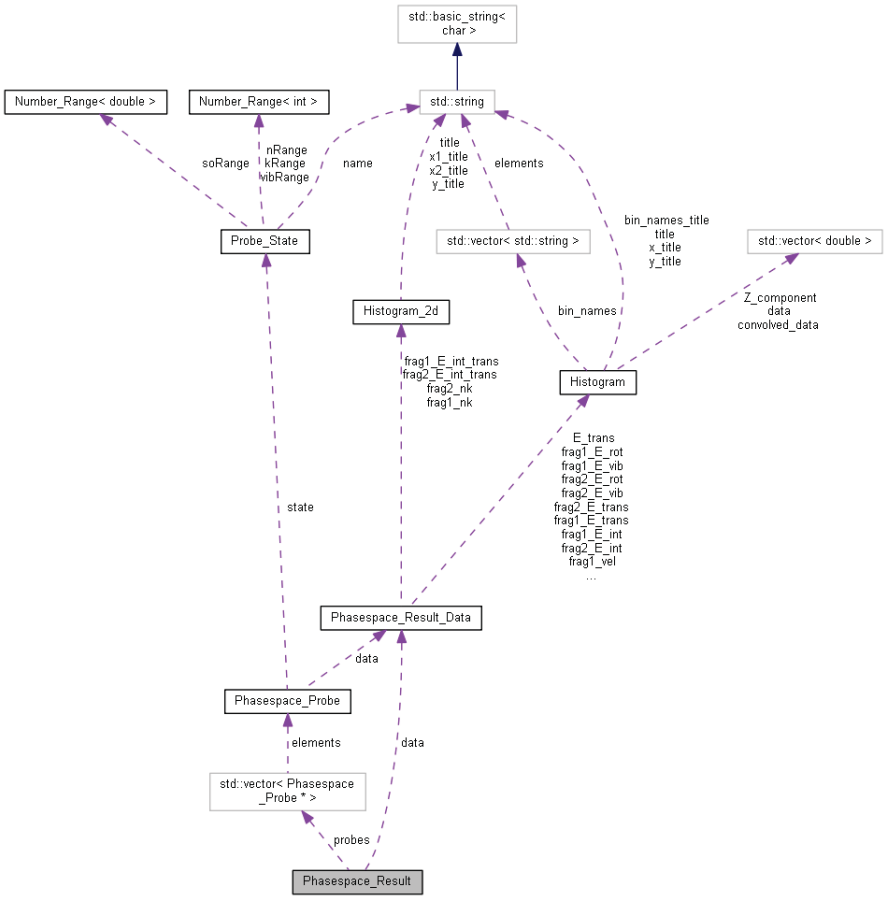

FIGURE B.24: Collaboration Diagram for Phasespace_Result

Phasespace_Result, Phasespace_Result_Data and Phasespace_Probe are defined in the files phasespace_result.h, phasespace_result_data.h and phasespace_probe.h respectively. Phasespace_Result is the base core output class, as illustrated in Figure B.7. The Phasespace_Result object is created and used by the Phasespace_Calculation and Impulsive_Calculation classes and passed to the constructor of their respective Phasespace_Core or Impulsive_Phasespace_Core classes. The Phasespace_Result object is populated with the results of the PST state count within the Phasespace_Core or derivative. The collaboration diagram in Figure B.24 shows that the Phasespace_Result class has several object members. Phasespace_Result has a Phasespace_Result_Data object member, data, which is used to store the overall results from the Phasespace_Core PST state count, including the total number of states and all the various histograms that *PST*$_{Tools}$ can output. Phasespace_Result also has a list of Phasespace_Probe

objects, called probes, with one Phasespace_Probe object for each state or range of states that have been defined as probes in the input file. Each of these Phasespace_Probe objects also has a Phasespace_Result_Data object member called data that is used to store the results for states that satisfy the probe criteria, including the total number of states and various histograms. The Phasespace_Core object evaluates whether a state satisfies the criteria of a Phasespace_Probe object by accessing its Probe_State object member, state, which defines the state or state range.

B.6.2 Primary Dissociation Result

Primary_Dissociation_Result is defined in the file triple_fragmentation_core.h.
The Primary_Dissociation_Result object is created and populated by the Triple_Fragmentation_Calculation class and passed to the constructor of the Triple_Frag_Primary class, which is a derivative of Impulsive_Phasespace_Core.
Primary_Dissociation_Result derives from Phasespace_Result and overrides the copy constructor, the overloaded = operator, the overloaded += operator, initialise() and serialize(). All the overridden functions call the parent implementation first.

B.6.3 Secondary Dissociation Result

Secondary_Dissociation_Result is defined in the file triple_fragmentation_core.h.
The Secondary_Dissociation_Result object is created and populated by the Triple_Fragmentation_Calculation class and passed to the constructor of the Triple_Frag_Secondary class, which is a derivative of Impulsive_Phasespace_Core.
Secondary_Dissociation_Result derives from Phasespace_Result and overrides the copy constructor, the overloaded = operator, the overloaded += operator, initialise(), reset() and serialize(). All the overridden functions call the parent implementation first.

B.6.4 Roaming Result

Roaming_Result, Roaming_Result_Data and Roaming_Probe are defined in the files roaming_result.h, roaming_result_data.h and roaming_probe.h respectively. The Roaming_Result

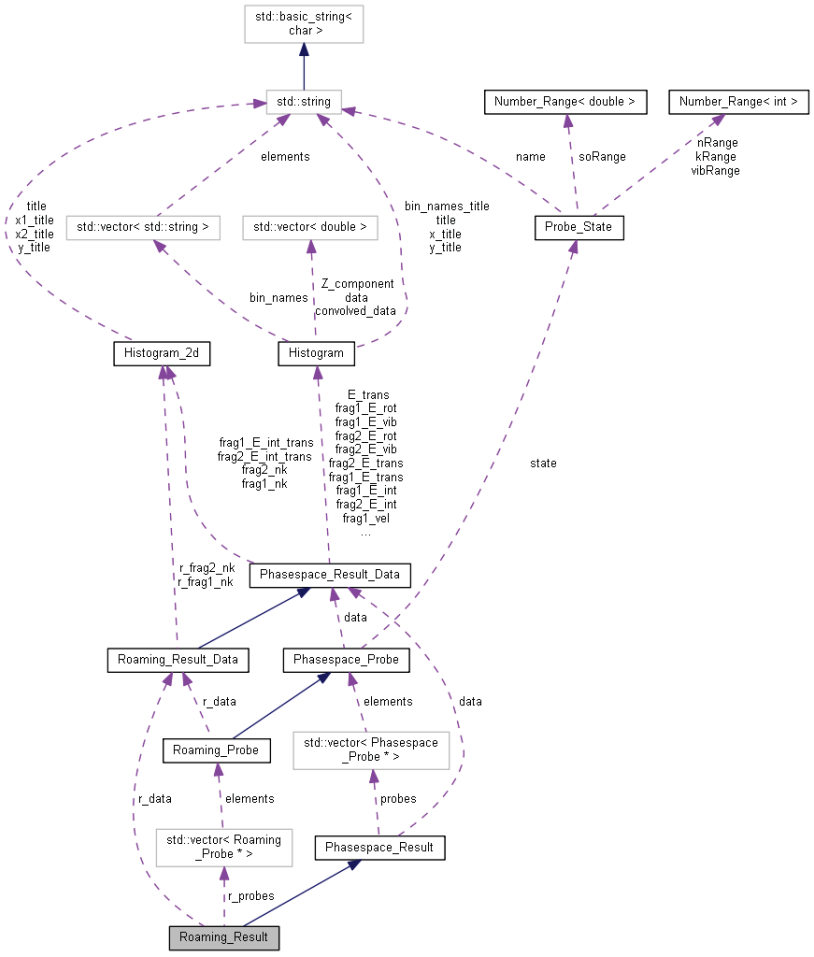

FIGURE B.25: Collaboration Diagram for Roaming_Result

object is created and populated by the Roaming_Calculation, Roaming_Fraction_Energy_Dependance_Calculation, Roaming_Fit_Delta_E_Roam_And_P_Roam_Calculation and Roaming_Fit_Trot_And_Tvib_And_P_Roam_Calculation classes and passed to the constructor of the Roaming_Phasespace_Core class.

The collaboration diagram in Figure B.25 shows that Roaming_Result adds a new Roaming_Result_Data pointer, r_data, and a list of Roaming_Probe pointers, r_probes, as members. Because Roaming_Result inherits members that are pointers to the base Phasespace_Result_Data and Phasespace_Probe classes (data and probes), one must dynamically cast those pointers to avoid object slicing and get access to the features specific to the derivatives Roaming_Result_Data

and Roaming_Probe. The dynamically cast pointers are created in create_data() for r_data and create_probes() for r_probes. These pointers are used to access the result and probe data throughout Roaming_Calculation and its derivatives.

B.7 The PST Core Classes

PSTTools has an inheritance hierarchy for the classes that contain the logic required to do a single PST state count, which is shown in Figure B.5. A single microcanonical PST state count is done at a discrete energy. Many of the calculations in *PSTTools* will perform a large number of single microcanonical PST state counts to obtain a result for output.

B.7.1 Phasespace Core

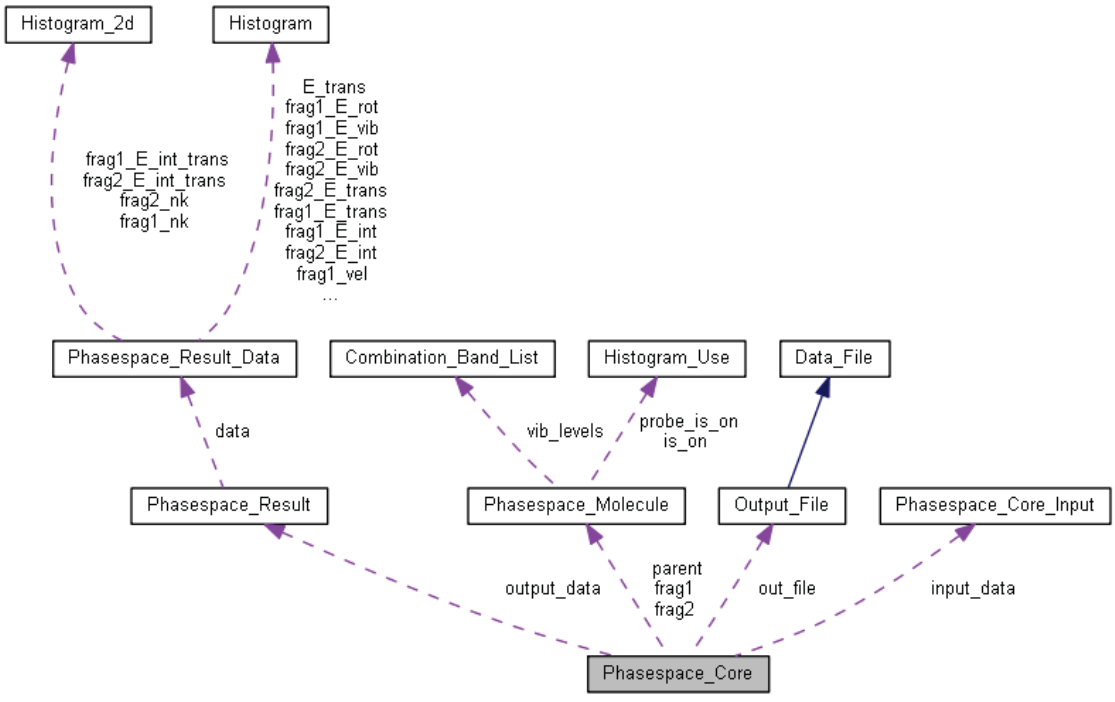

FIGURE B.26: Collaboration Diagram for Phasespace_Core

Phasespace_Core is defined in the file phasespace_core.h. Phasespace_Core is the base PST core class, as illustrated in Figure B.5. Phasespace_Core contains the core logic required to do

a single microcanonical PST state count for a two fragment system, at a discrete available energy, at infinite fragment separation, where the exit channel has no potential barrier. The Phasespace_Core object is created and used by the Phasespace_Calculation class. The collaboration diagram in Figure B.26 shows that the Phasespace_Core class has several object members. The Phasespace_Core_Input pointer, input_data, points to the general input parameters for the PST core, see section B.5.1 for more detail. The Phasespace_Result pointer, output_data, points to where the output from the PST core is stored, see section B.6.1 for more detail. The Phasespace_Molecule pointers, parent, frag1 and frag2, point to the objects that encapsulate code and data relevant to those molecular fragments, see section B.3.1 for more detail.

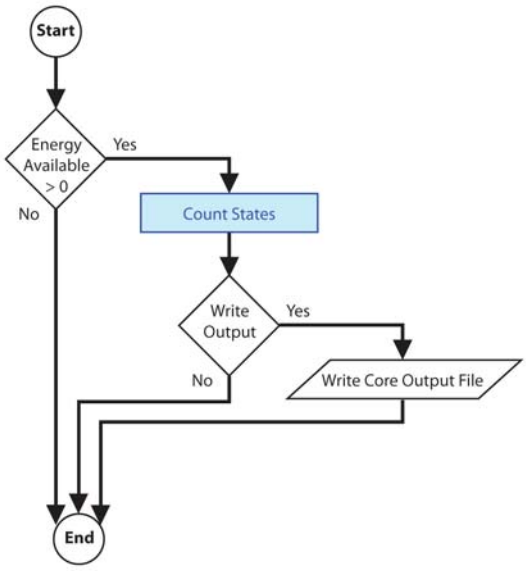

FIGURE B.27: Flowchart for the Phasespace_Core implementation of run().

The Phasespace_Core constructor calls initialise() then run(). As the name implies, initialise() performs any initialisation required before the PST state count is run. In any Phasespace_Core derivative, the initialise() function must be overridden, even if there are no additional initialisation statements required for the derived class. This is because the convention of calling the parent initialise() function implementation before adding derived initialisation code is followed for initialise(). If this convention is not adhered to in a derivative requiring no additional initialisation code, any derivatives of that class that do require additional initialisation will fail to call the

Phasespace_Core initialise() function when following this convention, missing out on the fundamental initialisation statements that are common to all PST state counts. The run() function is the function called to start the state count. The flowchart in Figure B.27 shows the program flow for the Phasespace_Core class implementation of the run() function. The count_states() function, represented as a blue rectangle in Figure B.27, is where the actual counting of states occurs. As of this *PSTTools* version, count_states() is not overridden by any Phasespace_Core derivatives. We do not consider a flowchart representation of count_states() feasible, considering the limited resolution available with A4 printed media, for which this document is intended. Inside the ten nested loops of count_states(), quantum states that can exist (because they conserve both energy and angular momentum) are processed by process_state() and then recorded in the histograms contained within the Phasespace_Result object, which was passed to the Phasespace_Core constructor. Process_state() is overridden in derivatives of Phasespace_Core, which require different treatments of allowed states. The functions create_input_pointer(), create_output_pointer() and create_fragment_pointers() are empty place holders that will be overridden in and called in the constructors of derivatives that use derived versions of the respective Phasespace_Core_Input, Phasespace_Result and Phasespace_Molecule classes, so that dynamically cast pointers can be assigned avoiding object slicing. A more detailed description of our use of dynamic cast to avoid object slicing is provided in some of the class descriptions above and below, see sections B.2.2, B.2.3, B.2.5, B.3.3, B.6.4, B.7.2.

B.7.2 Impulsive Phasespace Core

Impulsive_Phasespace_Core is defined in the file impulsive_core.h. Impulsive_Phasespace_Core contains the core logic required to do a single microcanonical PST state count for a two fragment system, at a discrete available energy, at infinite fragment separation, that may have a potential energy barrier in the exit channel. The Impulsive_Phasespace_Core object is created and used by the Impulsive_Calculation and Triple_Fragmentation_Calculation classes. The Impulsive_Phasespace_Core class derives from the Phasespace_Core class, as illustrated in Figure B.5, and overrides initialise(), run(), process_state(), create_input_pointer() and create_fragment_pointers(). The Impulsive_Phasespace_Core implementation of initialise() first

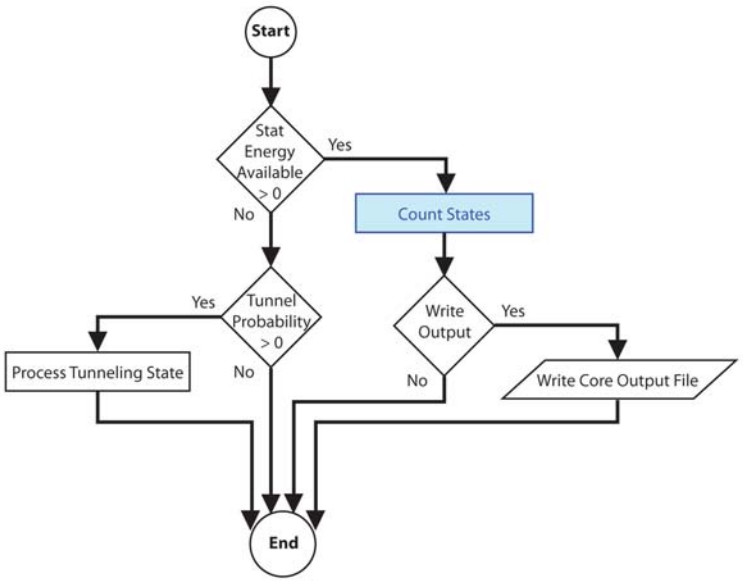

FIGURE B.28: Flowchart for the Impulsive_Phasespace_Core implementation of run().

calls the Phasespace_Core implementation and then performs the initialisation that is specific to Impulsive_Phasespace_Core. The flowchart in Figure B.28 shows the program flow for the Impulsive_Phasespace_Core class implementation of the run() function.

The collaboration diagram in Figure B.29 shows that in addition to the members inherited from Phasespace_Core, the Impulsive_Phasespace_Core class adds 4 new object members. The 4 new members, i_parent, i_frag1, i_frag2 and i_input_data, are pointers to the derived versions of Phasespace_Molecule and Phasespace_Core_Input that add logic specific to modelling impulsive reactions, see Figures B.3 and B.6 and sections B.3.2 and B.5.2. Because Impulsive_Phasespace_Core inherits members that are pointers to the base Phasespace_Molecule (reactant, frag1 and frag2) and Phasespace_Core_Input (input_data) classes, one must dynamically cast those pointers to avoid object slicing and get access to the features specific to the derivative. The dynamically cast pointers are created in create_fragment_pointers() and create_input_pointer() and stored in i_parent, i_frag1, i_frag2 and i_input_data. These pointers are used to access the objects throughout Impulsive_Phasespace_Core.

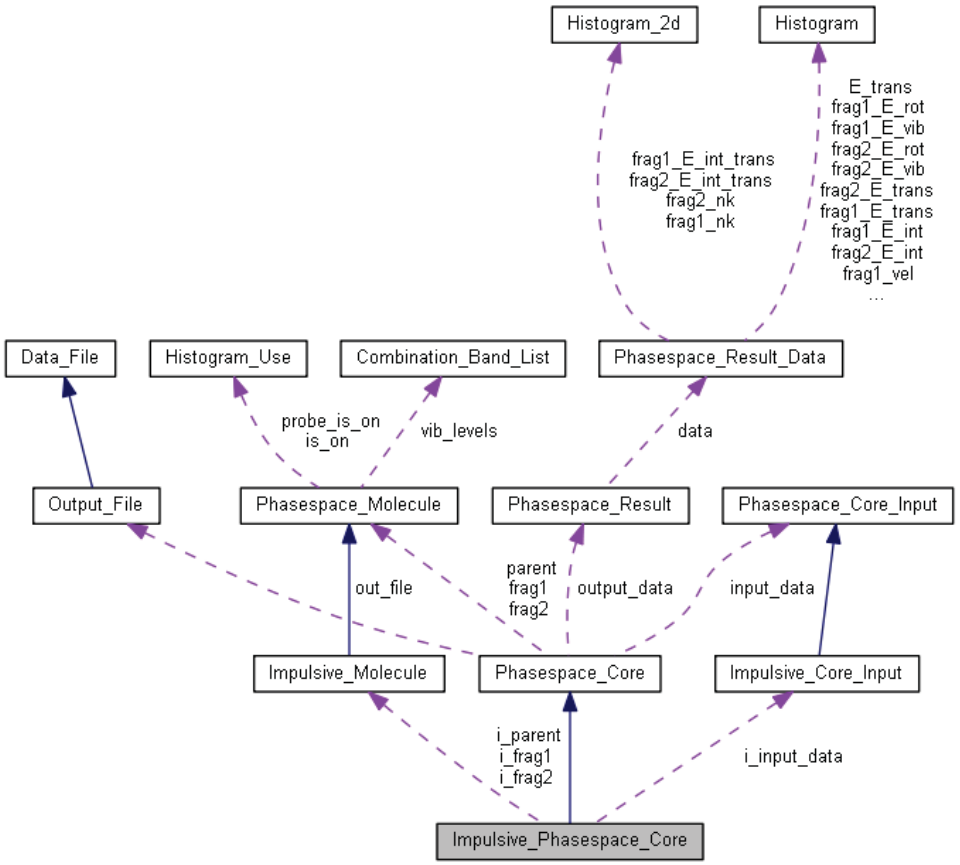

FIGURE B.29: Collaboration Diagram for Impulsive_Phasespace_Core

B.7.3 3F Primary Phasespace Core

Triple_Frag_Primary is defined in the file triple_fragmentation_core.h. Triple_Frag_Primary contains the core logic required to do a single microcanonical PST state count for a two fragment system, at a discrete available energy, at infinite fragment separation, that may have a potential energy barrier in the exit channel, where one fragment has been defined as potentially undergoing secondary fragmentation, and any states where that fragment has sufficient internal energy for secondary fragmentation to occur is logged in either an output file or a three dimensional array for further processing. The Triple_Frag_Primary object is created and used by the Triple_Fragmentation_Calculation class. The Triple_Frag_Primary class derives from the Impulsive_Phasespace_Core class, as illustrated in Figure B.5, and overrides initialise(), process_state() and create_output_pointer().

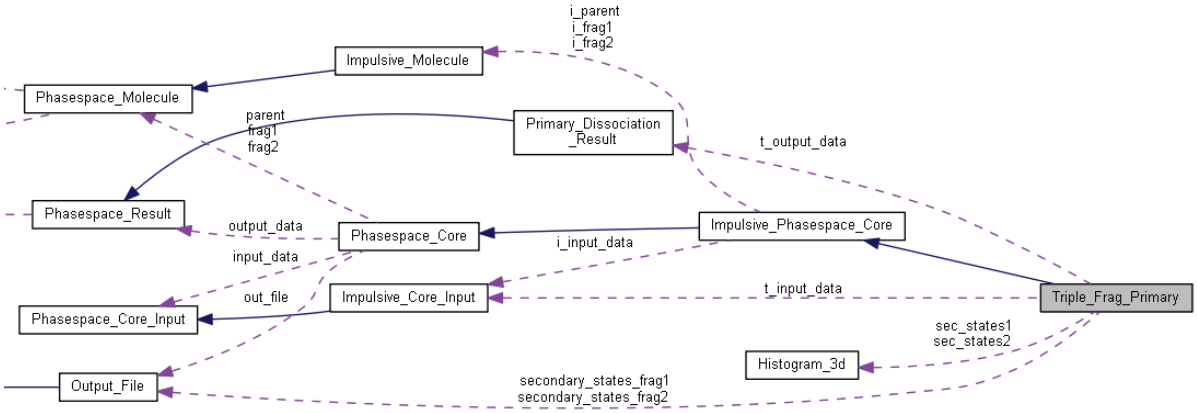

FIGURE B.30: Collaboration Diagram for Triple_Frag_Primary

The collaboration diagram in Figure B.30 shows that in addition to the members inherited from Impulsive_Phasespace_Core, the Triple_Frag_Primary class adds 5 new object members. The new member, t_output_data, points to a derived version of Phasespace_Result that adds logic specific to modelling the initial fragmentation step in a 3F reaction, see Figure B.24 and section B.6.2. The new members, sec_states1 and sec_states2, point to the 3 dimensional histograms that store the states with enough internal energy for secondary fragmentation to occur for each fragment product in the primary fragmentation step. The new members, secondary_states_frag1 and secondary_states_frag2, are output files that store the states with enough internal energy for secondary fragmentation to occur for each fragment product in the primary fragmentation step. Whether one of the 3 dimensional histograms or one of the output files is used to store the required states will depend on the input data.

B.7.4 3F Secondary Phasespace Core

Triple_Frag_Secondary is defined in the file triple_fragmentation_core.h. Triple_Frag_Secondary contains the core logic required to do a single microcanonical PST state count for a two fragment system, at a discrete available energy, at infinite fragment separation, that may have a potential energy barrier in the exit channel, where the results are scaled by a factor that takes in to account the degeneracy of the primary state from which this secondary fragmentation state derives and the total degeneracy of all secondary states produced by that primary state. Triple_Frag_Secondary object is created and used by the Triple_Fragmentation_Calculation

class. The Triple_Frag_Secondary class derives from the Impulsive_Phasespace_Core class, as illustrated in Figure B.5, and overrides initialise(), process_state(), create_input_pointer() and create_output_pointer().

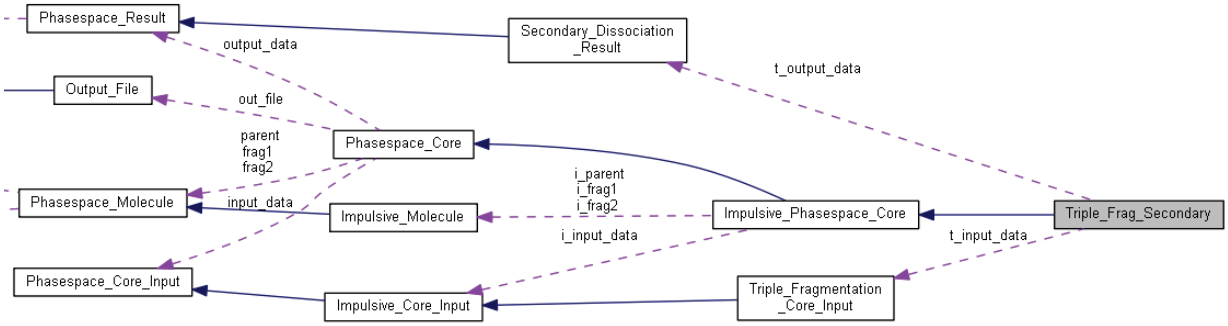

FIGURE B.31: Collaboration Diagram for Triple_Frag_Secondary

The collaboration diagram in Figure B.31 shows that in addition to the members inherited from Impulsive_Phasespace_Core, the Triple_Frag_Secondarys adds 2 new object members. The 2 new members, t_input_data and t_output_data, are pointers to the derived versions of Phasespace_Core_Input and Phasespace_Result that add logic specific to modelling the secondary fragmentation step in a 3F reaction, see Figures B.6 and B.24 and sections B.5.3 and B.6.3.

B.7.5 Roaming Phasespace Core

Roaming_Phasespace_Core is defined in the file roaming_core.h. Roaming_Phasespace_Core contains the core logic required to do a single microcanonical PST state count for a two fragment system, at a discrete available energy, where the exit channel has no potential barrier, at a fragment separation corresponding to the minimum bond length at which roaming may occur, categorising each state counted as either potentially roaming or dissociating via the corresponding barrierless channel. The Roaming_Phasespace_Core object is created and used by the Roaming_Calculation class and its derivatives. The Roaming_Phasespace_Core class derives from the Phasespace_Core class, as illustrated in Figure B.5, and overrides initialise(), process_state(), create_input_pointer(), create_output_pointer() and create_fragment_pointers().

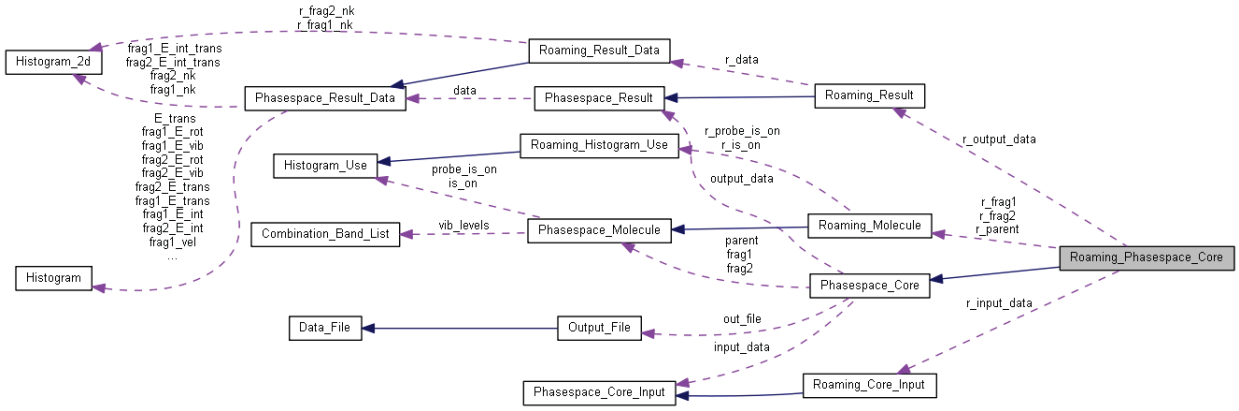

FIGURE B.32: Collaboration Diagram for Roaming_Phasespace_Core

The collaboration diagram in Figure B.32 shows that in addition to the members inherited from Phasespace_Core, the Roaming_Phasespace_Core class adds 5 new object members. The 5 new members, i_parent, i_frag1, i_frag2, i_input_data and i_output_data, are pointers to the derived versions of Phasespace_Molecule, Phasespace_Core_Input and Phasespace_Result that add logic specific to modelling roaming reactions, see Figures B.3, B.6 and B.24 and sections B.3.3, B.5.4 and B.6.4.