JSC-61777-models/utils/memory

JSC Engineering Orbital Dynamics Memory Model

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

Package Release JEOD v5.3

Document Revision 2.0
February 2025

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

JSC Engineering Orbital Dynamics
Memory Model

Document Revision 2.0
February 2025

David Hammen

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Executive Summary

The Memory Model forms a component of the dynamics suite of models within JEOD v5.3. It is located at models/utils/memory.

This model provides mechanisms in the form of macros and classes for allocating and deallocating memory. Several factors drove the design of the Memory Model.

Environment. The model is to work within and outside of the Trick environment. (Only the Trick-based capabilities are used in JEOD 2.0.)

Flexibility. The model is to work with a wide variety of data types, including pointers, C++ primitive types, and classes.

C++ Concerns. The C++ allocation operators `new` and `new[]` have distinct semantics, as do `delete` and `delete[]`.

Ease of Use. All of the above concerns should be hidden from the programmers who use the model.

Contents

Chapter 1

Introduction

1.1 Purpose and Objectives of the Memory Model

The Memory Model provides mechanisms in the form of macros and classes for allocating and deallocating memory.

1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [1]

The Memory Model forms a component of the utilities suite of models within JEOD v5.3. It is located at models/utils/memory.

1.3 Documentation History

Author	Date	Revision	Description
David Hammen	December, 2009	1.0	Initial version
David Hammen	October, 2010	2.0	New memory model

1.4 Documentation Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [2].

The document comprises chapters organized as follows:

Introduction - This introduction describes the objective and purpose of the Memory Model.

Product Requirements - The requirements chapter describes the requirements on the Memory Model.

Product Specification - The specification chapter describes the architecture and design of the Memory Model.

User Guide - The user guide chapter describes how to use the Memory Model.

Inspection, Verification, and Validation - The inspection, verification, and validation (IV&V) chapter describes the verification and validation procedures and results for the Memory Model.

Chapter 2

Product Requirements

Requirement Memory_1: Top-level requirement

Requirement:
The Memory Model shall meet the JEOD project requirements specified in the JEOD v5.3 top-level document.

Rationale:
This model shall, at a minimum, meet all external and internal requirements applied to the JEOD v5.3 release.

Verification:
Inspection

Requirement Memory_2: Memory Allocation

Requirement:
The Memory Model shall provide the ability to allocate and initialize memory of the data types described below.

2.1 Array of pointers to some data type. The model shall provide the ability to allocate an array of a specified number of pointers to a specified type, with each element of the allocated array initialized to the null pointer.

2.2 Array of structured objects. The model shall provide the ability to allocate an array of a specified number of objects of a specified structured (non-primitive) type, with each element of the allocated array initialized using the default constructor for the specified type.

2.3 Array of primitive objects. The model shall provide the ability to allocate an array of a specified number of objects of a specified primitive data type, with each element of the allocated array initialized as zero.

2.4 Single structured object. The model shall provide the ability to allocate a single instance of a specified structured type, with the allocated object initialized using the constructor that matches a specified set of constructor arguments.

2.5 Single primitive object. The model shall provide the ability to allocate a single instance of a specified primitive type, with the allocated object initialized to a specified value.

2.6 String duplication. The model shall provide the ability to create a copy of a specified null-terminated string.

Rationale:

The primary objective of the Memory Model is to make allocated memory visible to the simulation engine (e.g., Trick) in a manner that enables the simulation engine's data input and output mechanisms to operate on the memory. The simulation engine must be notified of memory allocations to the use of these input and output mechanisms. Encapsulating memory allocations is the first step in accomplishing this primary objective.

Verification:

Inspection, Test

Requirement Memory_3: Memory Registration

Requirement:

The Memory Model shall use the capabilities provided by the Simulation Interface Model to register all memory allocated using the capabilities described in requirement Memory_2 with the simulation engine.

Rationale:

This requirement is the second step toward accomplishing the primary objective of making allocated memory visible to the simulation engine.

The reason for using the Simulation Interface Model is that JEOD 2.1 encapsulates all interactions with the simulation engine in that model.

Verification:

Inspection, Test

Requirement Memory_4: Memory Deallocation

Requirement:

The Memory Model shall provide the ability to destroy and release memory previously allocated via the capabilities described in requirement Memory_2.

Rationale:

Memory allocated by JEOD models must eventually be released to avoid memory leaks.

Verification:

Inspection, Test

Requirement Memory_5: Freeing from a Base Class Pointer

Requirement:
The Memory Model shall properly destroy and release an object previously allocated by the model given a base class pointer to the object in question.

Rationale:
Derived class and base class pointers to the same object do not necessarily point to the same location in memory. The Memory Model machinery must be capable of handling this fact of life.

Verification:
Inspection, Test

Requirement Memory_6: Memory Deregistration

Requirement:
The Memory Model shall use the capabilities provided by the Simulation Interface Model to remove the registration of memory that is deallocated using the capabilities described in requirement Memory_4.

Rationale:
Memory that is released can be reused for other purposes. Failing to remove the registration for deallocated memory would make the simulation engine's memory model invalid.

Verification:
Inspection, Test

Requirement Memory_7: Thread Safety

Requirement:
The implementation of the Memory Model shall be thread-safe when used in a POSIX-complaint environment.

Rationale:
Trick makes extensive use of threads, as do many other simulation engines.

Verification:
Inspection, Test

Requirement Memory_8: Memory Tracking

Requirement:
The Memory Model shall keep track of all outstanding memory allocations made with the model (memory allocated using the capabilities described in requirement Memory_2 and not yet freed using the capabilities described in requirement Memory_4).

Rationale:

This capability is a nice-to-have for the purposes of debugging memory leaks but will be essential in future releases of JEOD to fully support checkpoint/restart.

Verification:

Inspection, Test

Requirement Memory_9: Memory Reporting

Requirement:

The Memory Model shall optionally report individual memory transactions and shall optionally report outstanding memory allocations at simulation end time.

Rationale:

This capability provides a debugging aid. This capability needs to be optional as the output is of little interest except when needed.

Verification:

Inspection, Test

Requirement Memory_10: Checkpoint/Restart (FUTURE)

Reserved.

Chapter 3

Product Specification

3.1 Conceptual Design

Several factors drove the design of the Memory Model.

Environment. The model is to work within and outside of the Trick environment.

Flexibility. The model is to work with a wide variety of data types, including pointers, C++ primitive types, and classes.

C++ Concerns. The C++ allocation operators `new` and `new[]` have distinct semantics, as do `delete` and `delete[]`. The Memory Model needs to provide analogous capabilities.

Ease of Use. The Memory Model should be easy to use. Details of the above concerns should be hidden from the programmers who use the model.

Backward Compatibility. While the JEOD 2.1 Memory Model represents a marked change in functionality, the extermal interfaces to the model remain unchanged.

Thread Safety. The JEOD 2.0 implementation of the model was solely in the form of macros. Ensuring that JEOD was thread-safe fell upon the simulation engine. Adding substance to the model required making thread safety a foremost concern in the design and implementation of the model.

3.1.1 Interactions

3.1.1.1 JEOD Models Used by the Memory Model

The Memory Model uses the following JEOD models:

- *Message Handler Model* [3]. The Memory Model uses the Message Handler Model to report error conditions and optionally, to generate debug output.
- *Named Item Model* [4]. The Memory Model uses the Named Item Model to demangle the type name in a std::type_info object.
- *Simulation Engine Interface Model* [5]. The Memory Model uses the Simulation Engine Interface Model as the mechanism via which memory allocations and deallocations are registered with the simulation engine.

3.1.1.2 Use of the Memory Model in JEOD

All JEOD models that dynamically allocate memory must use the Memory Model for those allocations if the allocated memory needs to be visible to the simulation engine for any reason (user input, logging, checkpoint/restart, ...). Most JEOD models use the Memory Model because of this mandate.

3.1.2 Macros

The primary external interface to the model are the Memory Model macros intended for external use. The signatures (names and arguments) of these macros remain unchanged from JEOD 2.0. The implementations of these macros differs significantly from JEOD 2.0.

The bodies of the external macros expand into invocations of Memory Model macros intended for internal use only. Because these internal macros are clearly designated as such, the arguments and even the names of these internal macros do differ from their JEOD 2.0 counterparts. In JEOD 2.1, these internal macros expand into creation of Memory Model objects and calls to Memory Model member functions.

3.1.3 Classes

The Memory Model comprises several classes and templates. The JeodMemoryManager class is a singleton class that allocates, keeps track of, and deallocates memory. The manager keeps track of allocated memory and data types. The manager represents each block of allocated memory as an instance of the JeodMemoryItem class. Instances of template classes that derive from the JeodMemoryTypeDescriptor class represent data types. Each unique data type is represented by a different generated class.

3.1.3.1 JeodMemoryManager

The JeodMemoryManager class provides the interface between the Memory Model macros and the rest of the JEOD memory model. All nonstatic member functions and all member data are private. The public interface is via the publicly visible static member functions. Each public static member function relays the method call to the singleton memory manager via a correspondingly-named private member function.

Singleton. The class is intended to be a singleton. The constructor ensures that at most one instance of the class exists. The publicly visible static member functions ensure that an instance of the class does exist. Each public static member function invokes a correspondingly-named private non-static member function on this singleton instance.

Maps. The memory manager uses map and table objects to track extant allocated memory, to store information about the types of data that have been allocated, and to store information about where in the code those allocations occurred.

Thread Safety. The maps and tables, along with other data members of the class, must be accessed and updated in a thread-safe manner. To ensure the constraint is satisfied, access to these elements is protected by means of a mutex and is limited to a small number of member functions. Thread safety is further ensured by encoding these potentially unsafe member functions in a systematic way.

3.1.3.2 JeodMemoryItem

The model uses the JeodMemoryItem class to represent blocks of allocated memory. Each block of allocated memory is represented by an instance of this class. The memory manager maintains an STL map that maps the address of a memory block to the JeodMemoryItem object that describes that block. A JeodMemoryItem object contains information about the size and type of an allocated block of memory. It also contains a unique identifier that will eventually be used when checkpoint/restart capabilities are added to the model.

3.1.3.3 JeodMemoryTable

One challenge with recording information about each allocated block of memory is that using 64 bit addresses can make for rather high overhead. A JeodMemoryItem object occupies only 16 bytes in part due to the use of classes that derived from JeodMemoryTable. A memory table indirectly maps keys to values. A memory table comprises an STL map and an STL vector of values. The STL map object maps keys to index numbers into the vector. A JeodMemoryItem object stores index numbers rather than keys, resulting in a significant reduction in overhead on a 64 bit machine.

3.1.3.4 JeodMemoryTypeDescriptor

The model uses the JeodMemoryTypeDescriptor class to represent data types. The class JeodMemoryTypeDescriptor is a pure virtual class. Two template classes derive from this base class, one for non-structured data and the other for structured data. Each data type is represented by an instance of a class generated from one of these two template classes. A JeodMemoryTypeDescriptor contains information about the size of an instance of the data type represented by the type descriptor and the name of the type. Key member functions of the class address the issue of deleting and destroying allocated memory and the issue of determining whether a pointer to some interior address is in fact a base class pointer.

3.1.4 Simulation Assumptions and Limitations

This model places certain limitations on the architecture of a JEOD-based simulation.

1. The JeodMemoryManager destructor uses the simulation's message handler to report errors discovered during destruction and may eventually use the simulation's simulation engine memory interface to revoke the registration of memory allocated by JEOD that has not been freed. This in turn means that:
   (a) The simulation's message handler and simulation engine memory interface must be destructed after destructing the memory manager.
   (b) The destructors for those objects cannot use the memory manager.
2. The JEOD memory allocation and deallocation macros expand into calls to memory manager member functions. The memory manager must be viable (constructed and not yet destructed) for these calls to function properly. This in turn means that the memory manager must be constructed very early in the overall construction process and destructed very late in the overall destruction process.
3. The supported solution to both of these issues is to use a compliant derived class of the JeodSimulationInterface class and to ensure that this composite object is created early and destroyed late. In a Trick-07 simulation, this can be accomplished simply by placing a declaration of an object of type JeodTrickSimInterface near the top of an S_define file. The recommended placement is just after the Trick system sim object.

This model also makes certain assumptions regarding the behavior of the simulation engine.

1. The simulation engine will not spawn threads that use the Memory Model to allocate memory until after the JeodSimulationInterface object that contains the memory manager object has been constructed.
2. The simulation engine will join all threads that use the Memory Model prior to destroying the JeodSimulationInterface object.

The Trick-07 simulation engine satisfies these assumptions given a simulation built according to the above limitations.

3.2 Mathematical Formulations

N/A

3.3 Detailed Design

3.3.1 Memory Model Macros

The primary external interface to the model is in the form of macros that are explicitly marked as externally-usable. These in turn expand into invocations of internal macros. The internal macros

in turn expand into calls to JeodMemoryManager static member functions.

Tables 3.1and 3.2 list the externally-usable and the underlying primitive JEOD memory management macros. See the model API documentation for details.

Table 3.1: Externally-Usable Memory Model Macros

Macro	Description
JEOD_ALLOC_CLASS_MULTI_POINTER_ARRAY	Allocate array of pointers
JEOD_ALLOC_CLASS_POINTER_ARRAY	Allocate array of pointers
JEOD_ALLOC_CLASS_ARRAY	Allocate array of objects
JEOD_ALLOC_PRIM_ARRAY	Allocate array of primitives
JEOD_ALLOC_CLASS_OBJECT	Allocate one structured object
JEOD_ALLOC_PRIM_OBJECT	Allocate one primitive object
JEOD_STRDUP	Duplicate a string
JEOD_IS_ALLOCATED	Was memory allocated by this model?
JEOD_DELETE_ARRAY	Delete array of pointers
JEOD_DELETE_OBJECT	Delete allocated object

Table 3.2: Primitive Macros

Macro	Description
JEOD_MEMORY_DEBUG_INTERNAL	Debug level cast to enum value
JEOD_ALLOC_OBJECT_FILL	Fill pattern for structured types
JEOD_ALLOC_PRIMITIVE_FILL	Fill pattern for primitive types
JEOD_ALLOC_POINTER_FILL	Fill pattern for pointer types
JEOD_ALLOC_TYPE_IDENTIFIER	Allocate type identifier for a type
JEOD_ALLOC_CLASS_IDENTIFIER	Allocate class identifier for a type
JEOD_ALLOC_CREATE_MEMORY	Allocate and register memory
JEOD_ALLOC_ARRAY_INTERNAL	Allocate an array of objects
JEOD_ALLOC_OBJECT_INTERNAL	Allocate a single instance
JEOD_DELETE_INTERNAL	Delete memory

3.3.2 JeodMemoryManager

3.3.2.1 Public Interface

The public interface to the memory manager is in the form of static member functions. These static member functions test whether the singleton memory manager instance exists, and if it does, operates on that method.

3.3.2.2 Thread Safety

Thread safety was a major concern in the design and implementation of the model. The following JeodMemoryManager data members could become corrupted in a multi-threaded environment without adequate protection:

- `alloc_table` - Maps memory addresses to memory items.
- `type_table` - Maps RTTI names to type descriptors.
- `string_table` - Maps unique strings to themselves.
- `cur_data_size` - Current size of allocated data.
- `max_data_size` - Maximum of the above.
- `max_table_size` - Maximum allocation table size.
- `allocation_number` - Number of allocations made.

Several precautions are taken to protect against such corruption. Access to these protected data is limited to three groups of member functions.

1. The non-default constructor, the destructor, and member functions called only by the constructor and destructor (`generate_shutdown_report` in the current implementation). These member functions have free access to the protected data. This unfettered access will not result in corrupted data if the simulation and the simulation engine are constructed and behave according to the assumptions and limitations specified in section 3.1.4.
2. Methods whose names end in `_atomic`. These member functions use the mutex lock and unlock member functions (see section 3.3.2.3) to ensure exclusive access to sensitive data. Calls to other JeodMemoryManager member functions by members of this group of member functions are limited to the mutex lock and unlock member functions and the `_nolock` methods (next group), with the latter only called within a protected block of code. These `_atomic` member functions must not and do not call one another as doing so would lead to either a lock error or deadlock.
3. Methods whose names end in `_nolock`. These member functions also have unfettered access to the sensitive data. Members of either of the two previous groups can call these member functions. Currently there is only one such member function, `get_type_descriptor_nolock`.

Figure 3.1 depicts the access to the protected data by the `_atomic` member functions. The figure also depicts the member functions that directly or indirectly call these atomic member functions. Member functions are depicted as ovals, member data as parallelograms, function calls as solid lines, and data access as dashed lines.

3.3.2.3 Exclusive Access

The JeodMemoryManager contains a POSIX mutex data member whose purpose is to ensure exclusive access to the data that could otherwise become corrupted in a multi-threaded environment. The mutex is created by the constructor and destroyed by the destructor. (This is another reason for the constraints on the simulation architecture and the simulation engine.)

Only two other member functions directly access this mutex. These are the `begin_atomic_block` and `end_atomic_block` member functions. The former function obtains a lock on the mutex,

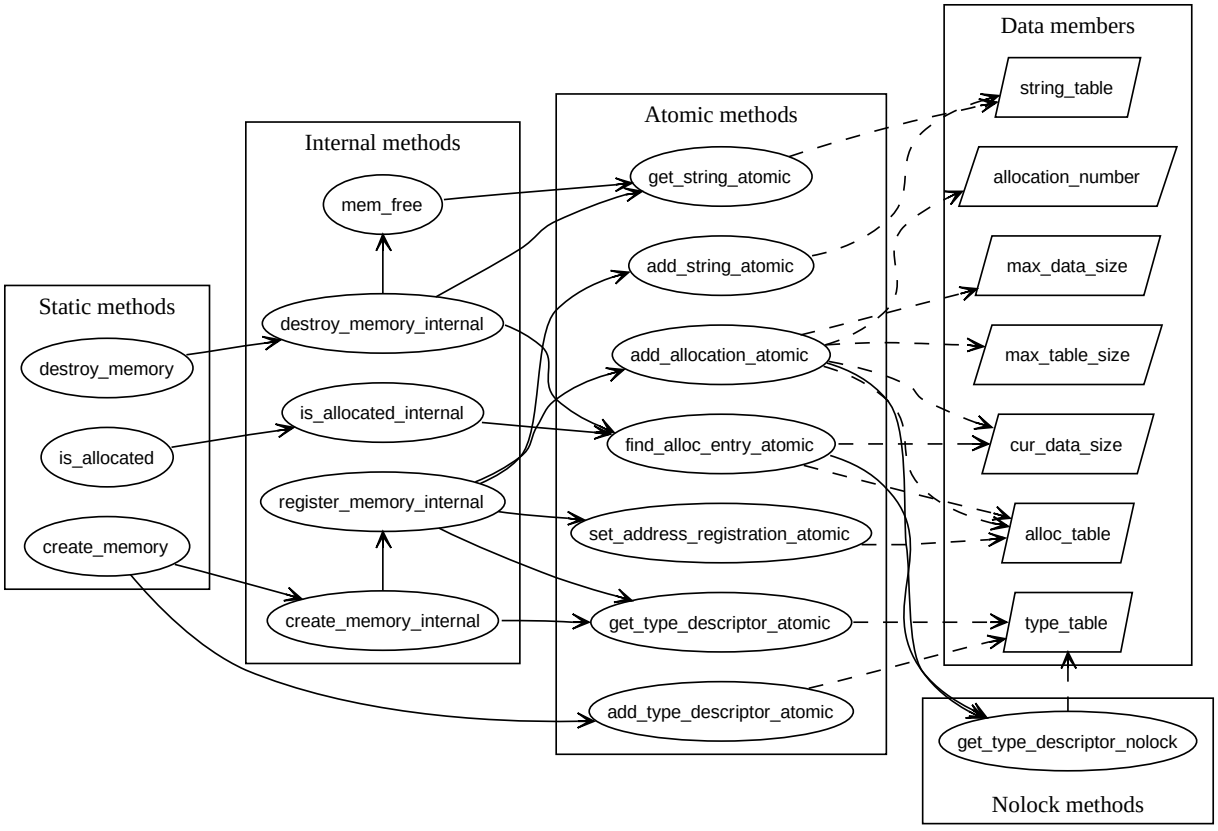

Figure 3.1: Protection of Thread-Sensitive Data

preventing any other thread from becoming active until the lock is released. The latter function releases the lock, once again allowing the system to switch threads to optimize performance. One key finding of the model peer review was that these methods should nominally return only if the request was successful. Failure to obtain or release the lock constitutes a fatal error. This simplifies the design of the functions themselves and that of the calling functions. The function `begin_atomic_block` obeys this dictum strictly. The function `end_atomic_block` takes an argument that indicates whether it should or should not ignore errors. When the argument is false (nominal usage) `end_atomic_block` checks for errors and returns only if the lock is successfully released. When the argument is true the function attempts to release the lock but does not check the status of the request. Calling functions use this latter capability when a fatal error has already been detected.

The only member functions that call `begin_atomic_block` and `end_atomic_block` are the `_atomic` functions. All of the `_atomic` member functions are written as follows:

```
try {
   begin_atomic_block ();
   // Operate on member data at will
   if (<error_condition>) {
      end_atomic_block (true);
      MessageHandler::fail (...);
      return;
   }
   // Continue operating on member data at will
   end_atomic_block (false);
}
catch (...) {
   end_atomic_block (true);
   throw;
}
```

3.4 Waivers

The data member `JeodMemoryManager::mutex` is `mutable`, a forbidden word per the JEOD coding standards. The coding standards allow for waivers to the standards if the exception is justified. This section provides the explanation needed to enable the use of that word in this case.

The `mutable` keyword tells the compiler to ignore modifications to mutable elements in an otherwise `const` method. The `mutex` data member is qualified as being mutable because, athough its value does change with a successful lock, it is restored to its prelock value with an unlock. A method that could otherwise qualify as a `const` method can still be a const method by marking the mutex as mutable. Mutexes are one of the well-accepted types of data that typically are designated as mutable.

3.5 Inventory

All Memory Model files are located in the directory `${JEOD_HOME}/models/utils/memory`. Relative to this directory,

- Header and source files are located in the model `include` and `src` subdirectories. Table 3.3 lists the configuration-managed files in these directories.
- Verification files are located in the model `verif` subdirectory. See table 3.5 for a listing of the configuration-managed files in this directory.
- Documentation files are located in the model `docs` subdirectory. See table 3.4 for a listing of the configuration-managed files in this directory.

Table 3.3: Source Files

File Name
include/class_declarations.hh
include/jeod_alloc.hh
include/jeod_alloc_construct_destruct.hh
include/jeod_alloc_get_allocated_pointer.hh
include/memory_attributes_templates.hh
include/memory_item.hh
include/memory_manager.hh
include/memory_manager_hide_from_trick.hh
include/memory_messages.hh
include/memory_table.hh
include/memory_type.hh
src/cmake_file_list.cmake
src/memory_item.cc
src/memory_manager.cc
src/memory_manager_protected.cc
src/memory_manager_static.cc
src/memory_messages.cc
src/memory_type.cc

Table 3.4: Documentation Files

File Name
docs/memory.pdf
docs/refman.pdf

Continued on next page

Table 3.4: Documentation Files (continued from previous page)

File Name
docs/tex/atomic_methods.pdf
docs/tex/change_history.tex
docs/tex/exec_summary.tex
docs/tex/intro.tex
docs/tex/ivv.tex
docs/tex/ivv_inspect.tex
docs/tex/ivv_test_basic.tex
docs/tex/ivv_test_threads.tex
docs/tex/ivv_trace.tex
docs/tex/makefile
docs/tex/memory.bib
docs/tex/memory.sty
docs/tex/memory.tex
docs/tex/model_name.mk
docs/tex/reqt.tex
docs/tex/spec.tex
docs/tex/user.tex

Table 3.5: Verification Files

File Name
verif/SIM_memory/S_define
verif/SIM_memory/S_overrides.mk
verif/SIM_memory/SET_test/RUN_checkpoint/input.py
verif/SIM_memory/SET_test/RUN_restart/input.py
verif/SIM_memory/SET_test_val_rh8/RUN_checkpoint/input.py
verif/SIM_memory/SET_test_val_rh8/RUN_restart/input.py
verif/unit_tests/CMakeLists.txt
verif/unit_tests/makefile
verif/unit_tests/memory_item_ut.cc
verif/unit_tests/memory_manager_protected_ut.cc
verif/unit_tests/memory_manager_static_ut.cc
verif/unit_tests/memory_manager_ut.cc
verif/unit_tests/memory_type_ut.cc

Continued on next page

Table 3.5: Verification Files (continued from previous page)

File Name
verif/unit_tests/basic/CMakeLists.txt
verif/unit_tests/basic/main.cc
verif/unit_tests/basic/makefile
verif/unit_tests/threads/CMakeLists.txt
verif/unit_tests/threads/main.cc
verif/unit_tests/threads/makefile

Chapter 4

User Guide

This chapter describes of the use of the Memory Model.

4.1 Analysis

The Memory Model is not an analytic tool. It can however be used to identify memory usage problems by defining `JEOD_MEMORY_DEBUG` and `JEOD_MEMORY_GUARD` in the `TRICK]_CFLAGS` environment variable. The following settings are available:

- `-DJEOD_MEMORY_DEBUG=0` (default setting). The Memory Model reports only serious errors in this mode.
- `-DJEOD_MEMORY_DEBUG=1`. Level 0 plus summary. This setting makes the model generate a summary report at shutdown time.
- `-DJEOD_MEMORY_DEBUG=2`. Reserved.
- `-DJEOD_MEMORY_DEBUG=3`. Blow-by-blow account of all transactions.
- `-DJEOD_MEMORY_GUARD=0` (default setting). The model does not guard against memory overwrites.
- `-DJEOD_MEMORY_GUARD=1` If both debugging and guarding are enabled, the model guards against simple memory overwrites by allocating 24 extra bytes around each allocation. Any detected modifications to these guard bytes are reported as errors.

4.2 Integration

Any JEOD-based simulation must contain an object of a class that derives from the JeodSimulationInterface class. A compliant version of a class that derives from JeodSimulationInterface (e.g., the JeodTrickSimInterface class recommended for use in all Trick-based simulations that use JEOD) must contain a JeodMemoryManager object. Following the instructions for use of the *Simulation Engine Interface Model* [5].

4.3 Programmatic Use of the Model

Model developers are free to use the externally-usable macros in their own models.

4.3.1 Primitive Types

To allocate/deallocate primitive types, or pointers to primitive types, all that is needed is to include the model header file and use the appropriate macros. For example, suppose the class `MyModel` has a `double **` member data element named `double_arr`. It is to be a $2 \times N$ array, where N is a to-be-supplied parameter. Displayed below are the `initialize` and `cleanup` methods for this class.

```
// Jeod includes
#include "utils/memory/jeod_alloc.hh"

// Model includes
#include "../my_model.hh"

void
MyModel::initialize (
   unsigned int nelem_in)  // In: -- Number of elements
{
   // Allocate double_arr
   nelem = nelem_in;
   double_arr    = JEOD_ALLOC_PRIM_ARRAY (2, double *);
   double_arr[0] = JEOD_ALLOC_PRIM_ARRAY (nelem, double);
   double_arr[1] = JEOD_ALLOC_PRIM_ARRAY (nelem, double);

   // Populate double_arr (not shown)
}

void
MyModel::cleanup ()
{
   JEOD_DELETE_ARRAY (double_arr[0]);
   JEOD_DELETE_ARRAY (double_arr[1]);
   JEOD_DELETE_ARRAY (double_arr);
}
```

4.3.2 Structured Types

Using the model with structured types in a Trick setting requires a bit more work. The user must declare as an external reference the automatically-generated `ATTRIBUTES` array for the types. Suppose that a pointer to a `Foo` object is added as a data member to `MyModel` class. The `Foo`

object is to be constructed with a non-default constructor that takes a reference to the MyModel object as an argument.

```
// Trick includes
#include "sim_services/include/attributes.h"
extern ATTRIBUTES * attrFoo;

// Jeod includes
#include "utils/memory/jeod_alloc.hh"

// Model includes
#include "../my_model.hh"

void
MyModel::initialize (
   unsigned int nelem_in)  // In: -- Number of elements
{
   // Allocate double_arr
   nelem = nelem_in;
   double_arr    = JEOD_ALLOC_PRIM_ARRAY (2, double *);
   double_arr[0] = JEOD_ALLOC_PRIM_ARRAY (nelem, double);
   double_arr[1] = JEOD_ALLOC_PRIM_ARRAY (nelem, double);

   // Populate double_arr (not shown)

   // Allocate and construct the Foo object.
   foo = JEOD_ALLOC_CLASS_OBJECT (Foo, (*this));
}

void
MyModel::cleanup ()
{
   JEOD_DELETE_OBJECT (foo);
   JEOD_DELETE_ARRAY (double_arr[0]);
   JEOD_DELETE_ARRAY (double_arr[1]);
   JEOD_DELETE_ARRAY (double_arr);
}
```

4.3.3 Macro Descriptions

4.3.3.1 Macro `JEOD_ALLOC_CLASS_MULTI_POINTER_ARRAY(nelem,type,asters)`

Purpose Allocate an array of `nelem` multi-level pointers to the specified `type`. The `asters` are asterisks that specify the pointer level. The allocated memory is initialized via `new`.

Returns Allocated array of specified type.

Arguments

`nelem` Size of the array.
`type` The underlying type, which must be a structured type.
`asters` A bunch of asterisks.

Example Allocate two pointers-to-pointers to the class `Foo`. Note that this does not allocate either the `Foo` objects or pointers to the `Foo` objects.

```
Foo *** foo_array = JEOD_ALLOC_CLASS_MULTI_POINTER_ARRAY(2,Foo,**);
```

4.3.3.2 Macro `JEOD_ALLOC_CLASS_POINTER_ARRAY(nelem,type)`

Purpose Allocate an array of `nelem` pointers to the specified `type`. The allocated memory is initialized via `new`.

Returns Allocated array of specified type.

Arguments

`nelem` Size of the array.
`type` The underlying type, which must be a structured type.

Example Allocate an array of two pointers to the class `Foo`. Note that this does not allocate the `Foo` objects themselves.

```
Foo ** foo_array = JEOD_ALLOC_CLASS_POINTER_ARRAY(2,Foo);
```

4.3.3.3 Macro `JEOD_ALLOC_CLASS_ARRAY(nelem,type)`

Purpose Allocate an array of `nelem` instances of the specified structured `type`. The default constructor is invoked to initialize each allocated object.

Returns Allocated array of specified type.

Arguments

`nelem` Size of the array.
`type` The underlying type, which must be a structured type.

Example Allocate an array containing two objects to the class `Foo`.

```
Foo ** foo_array = JEOD_ALLOC_CLASS_ARRAY(2,Foo);
```

4.3.3.4 Macro `JEOD_ALLOC_PRIM_ARRAY(nelem,type)`

Purpose Allocate `nelem` elements of the specified primitive `type`. The allocated array is zero-filled.

Returns Allocated array of specified type.

Arguments

`nelem` Size of the array.
`type` The underlying type, which must be a C++ primitive type.

Example Allocate an array of two doubles.

```
double * double_array = JEOD_ALLOC_PRIM_ARRAY(2,double);
```

4.3.3.5 Macro `JEOD_ALLOC_CLASS_OBJECT(type,constr)`

Purpose Allocate one instance of the specified class. The supplied constructor arguments, `constr`, are used as arguments to `new`. The default constructor will be invoked if the `constr` argument is the empty list; a non-default constructor will be invoked for a non-empty list.

Returns Pointer to allocated object.

Arguments

`type`	The underlying type, which must be a structured type.
`constr`	Constructor arguments, enclosed in parentheses.

Example Allocate a new object of type Foo, invoking the `Foo::Foo(bar,baz)` constructor.

```
Foo * foo = JEOD_ALLOC_CLASS_OBJECT(Foo,(bar,baz));
```

4.3.3.6 Macro `JEOD_ALLOC_PRIM_OBJECT(type, initial)`

Purpose Allocate one instance of the specified `type`. The object is initialized with the supplied `initial` value.

Returns Pointer to allocated primitive.

Arguments

`type`	The underlying type, which must be a C++ primitive type.
`initial`	Initial value.

Example Allocate a double and initialize it to π.

```
double * foo = JEOD_ALLOC_PRIM_OBJECT(double, 3.14159265358979323846);
```

4.3.3.7 Macro `JEOD_IS_ALLOCATED (ptr)`

Purpose Determine if `ptr` was allocated by some `JEOD_ALLOC_xxx_ARRAY` macro.

Returns true if `ptr` was allocated by the model, false otherwise.

Arguments

`ptr`	Memory to be checked.

Example Check that memory was allocated by JEOD before freeing it.

```
if (JEOD_IS_ALLOCATED (name)) {
   JEOD_DELETE_ARRAY(name);
}
```

4.3.3.8 Macro `JEOD_DELETE_ARRAY (ptr)`

Purpose Free memory at `ptr` that was earlier allocated with some `JEOD_ALLOC_xxx_ARRAY` macro.

Arguments

`ptr` Memory to be released.

Example Allocate a chunk of memory and then free it.

```
Foo * foo1 = JEOD_ALLOC_CLASS_ARRAY(2,Foo);
...
JEOD_DELETE_ARRAY(foo1);
```

4.3.3.9 Macro `JEOD_DELETE_OBJECT (ptr)`

Purpose Free memory at `ptr` that was earlier allocated with some `JEOD_ALLOC_xxx_OBJECT` macro.

Returns N/A

Arguments

`ptr` Memory to be released.

Example Allocate a chunk of memory and then free it.

```
Foo * foo1 = JEOD_ALLOC_CLASS_OBJECT(Foo,());
...
JEOD_DELETE_OBJECT(foo1);
```

4.4 Extension

The Memory Model is not designed for extensibility.

Chapter 5

Inspection, Verification, and Validation

5.1 Inspection

This section describes the inspections of the Memory Model.

Inspection Memory_1: Top-level Inspection

This document structure, the code, and associated files have been inspected. With the exception of the use of the *mutable* keyword (forbidden per the coding standards), the Memory Model satisfies requirement Memory_1. The use of the *mutable* keyword in the model has been granted a waiver.

Inspection Memory_2: Design Inspection

Table 5.1 summarizes the key elements of the implementation of the Memory Model that satisfy requirements levied on the model. By inspection, the Memory Model satisfies requirements Memory_2 to Memory_9.

Table 5.1: Design Inspection

Requirement	Satisfaction
Memory_2 Memory Allocation	The Memory Model externally-usable memory allocation macros ultimately expand into calls to memory manager methods that allocate and initialize memory.
Memory_3 Memory Registration	The memory manager uses its memory interface object (established via the non-default constructor) to register all allocated memory with the simulation engine.

Continued on next page

Table 5.1: Design Inspection (continued from previous page)

Requirement	Satisfaction
Memory_4 Memory Deallocation	The externally-usable memory deallocation macros ultimately expand into calls to memory manager methods that destroy and deallocate memory.
Memory_5 Free from Base Pointer	The memory manager accommodates for the fact that a derived class pointer upcast to a base class pointer does not always point to the same location in memory as the original derived class pointer. Memory allocated as a derived class can be deleted from a base class pointer to the allocated memory.
Memory_6 Memory Deregistration	The memory manager uses its memory interface object to inform the simulation engine that previously allocated memory is being released.
Memory_7 Thread Safety	The memory manager was designed, implemented, inspected, and tested with thread safety being one of the key goals.
Memory_8 Memory Tracking	The memory manager maintains a table of all extant memory allocations.
Memory_9 Memory Reporting	The memory manager provides the ability to report memory transactions and to report memory that has not been freed at the time the memory manager goes out of scope.

Inspection Memory_3: Peer Review

The Memory Model underwent an extensive peer review on July 29, 2010. While the primary focus of the peer review was thread safety, the review covered the entirety of the model. Findings were captured in the JEOD issue management system. Key findings included

- Eliminate use of the volatile keyword; many compilers do not handle this keyword properly.
- Make the errors detected in the sensitive sections of begin_atomic_block and end_atomic_block fatal errors. Making these errors non-fatal might lead to deadlock.
- Handle exceptions that might be thrown in a function called from within a sensitive block of code.

With the implementation of these findings, the Memory Model satisfies requirement Memory_7.

5.2 Validation

This section describes various tests conducted to verify and validate that the Memory Model satisfies the requirements levied against it. The tests described in this section are archived in the JEOD directory `models/utils/memory/verif`.

Test Memory_1: Basic

Test Directory `models/utils/memory/verif/unit_tests/basic`

Test Description This unit test exercises the basic ability of the Memory Model to allocate and deallocate memory. The test allocates and later deletes the following types of data:

1. An object of a primitive type via `JEOD_ALLOC_PRIM_OBJECT`,
2. Arrays of a primitive type via `JEOD_ALLOC_PRIM_ARRAY`,
3. An array of pointers to structured data via `JEOD_ALLOC_CLASS_POINTER_ARRAY`,
4. An array of a structured type via `JEOD_ALLOC_CLASS_ARRAY`,
5. Objects of a structured type via `JEOD_ALLOC_CLASS_OBJECT`.

To run the test, enter the test directory and type the command `make build` to build the test program and then type the command `./test_program -verbose` to run the test program with debugging enabled.

Success Criteria The success criteria involve analyzing the debug output that results from running the program in debug mode:

- Each memory allocation must indicate that an appropriately-sized block of memory is allocated, the allocated memory is registered the simulation interface, the allocated memory is registered with the Memory Model, and the allocated memory is initialized as expected.
- Each memory deallocation must indicate that the allocated memory: is deregistered with the simulation interface, is properly destructed and freed, and is deregistered with the Memory Model.
- The summary output must report that all memory has been freed.

Test Results The test passes.

Applicable Requirements This test demonstrates the partial or complete satisfaction of the following requirements:

- Memory_2. The `JEOD_ALLOC` macros properly allocate and initialize blocks of memory.
- Memory_3. The model registers allocated memory with the simulation engine.
- Memory_4. The `JEOD_DELETE` macros properly destruct and delete blocks of memory.
- Memory_5. The `JEOD_DELETE` macros function properly when the target of the deletion is a base class pointer with a different address than that of the allocated memory.
- Memory_6. The model deregisters released memory with the simulation engine.

Test Memory_2: Threads

Test Directory `models/utils/memory/verif/unit_tests/threads`

Test Description This unit test exercises the Memory Model in a multithreaded environment. The program spawns twelve threads. Each thread operates at a specific rate and allocates and deletes multiple chunks memory of a specific type. The collisions that would result from the memory operations performed by these threads would result in the model becoming corrupted were it not for the thread-safe features of the model.

To run the test, enter the test directory and type the command `make`. The test driver exercises the test program four times:

- Test #1 tests nominal behavior. Each thread allocates and deallocates memory. Eventually all memory allocated by a thread is deallocated. When all threads have finished there should be no extant allocated memory.
- Test #2 tests the ability to detect simple fence post errors, the most common type of memory error. Each thread writes just outside of one piece of allocated memory. The Memory Model should detect and report these as corrupted memory.
- Test #3 tests the ability to detect leaks. Each thread fails to release one piece of allocated memory. The Memory Model should detect these as leaks.
- Test #4 combines tests #2 and #3. The Memory Model should detect and report both the corrupted memory errors and the leaks.

Success Criteria Test criteria are embedded in the test program. The output for each test must indicate that the test passes.

Test Results Each of the four test passes its test criteria.

Applicable Requirements This test demonstrates the partial or complete satisfaction of the following requirements:

- Memory_2. Memory is allocated and initialized as expected.
- Memory_4. Memory is destructed and freed as expected.
- Memory_5. Freeing from a base class pointer works as expected.
- Memory_7. The model performs in a thread-safe manner.
- Memory_8. Memory allocations are tracked as evidenced by properly reporting leaks.
- Memory_9. Leaks and corrupted memory are reported upon request.

5.3 Metrics

Table 5.2 presents coarse metrics on the source files that comprise the model.

Table 5.2: Coarse Metrics

File Name	Number of Lines			
	Blank	**Comment**	**Code**	**Total**
include/class_declarations.hh	11	57	7	75
include/jeod_alloc.hh	42	381	105	528
include/jeod_alloc_construct_destruct.hh	24	143	61	228
include/jeod_alloc_get_allocated_pointer.hh	18	119	30	167
include/memory_attributes_templates.hh	19	107	58	184
include/memory_item.hh	49	184	93	326
include/memory_manager.hh	98	337	147	582
include/memory_manager_hide_from_trick.hh	11	68	12	91
include/memory_messages.hh	27	113	25	165
include/memory_table.hh	56	241	163	460
include/memory_type.hh	83	338	211	632
src/cmake_file_list.cmake	2	0	12	14
src/memory_item.cc	23	59	72	154
src/memory_manager.cc	87	302	460	849
src/memory_manager_protected.cc	100	339	428	867
src/memory_manager_static.cc	45	208	178	431
src/memory_messages.cc	17	35	17	69
src/memory_type.cc	31	72	112	215
Total	743	3103	2191	6037

Table 5.3 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.3: Cyclomatic Complexity

Method	File	Line	ECC
jeod::JeodAllocHelper ConstructDestruct:: construct(std::construct (std::size_t nelem JEOD_U NUSED, void * addr)	include/jeod_alloc_construct_ destruct.hh	104	1
jeod::JeodAllocHelper ConstructDestruct::destruct (std::destruct (std::size_t nelem JEOD_UNUSED, void * addr JEOD_UNUSE D)	include/jeod_alloc_construct_ destruct.hh	116	1
jeod::JeodAllocHelper ConstructDestruct:: construct(std::construct (std::size_t nelem, void * addr)	include/jeod_alloc_construct_ destruct.hh	132	1
jeod::JeodAllocHelper ConstructDestruct::destruct (std::destruct (std::size_t nelem JEOD_UNUSED, void * addr JEOD_UNUSE D)	include/jeod_alloc_construct_ destruct.hh	145	1
jeod::JeodAllocHelper ConstructDestruct:: construct(std::construct (std::size_t nelem, void * addr)	include/jeod_alloc_construct_ destruct.hh	161	1
jeod::JeodAllocHelper ConstructDestruct::destruct (std::destruct (std::size_t nelem, void * addr)	include/jeod_alloc_construct_ destruct.hh	173	2
jeod::jeod_alloc_construct_ array(std::jeod_alloc_ construct_array (std::size_t nelem, void * addr)	include/jeod_alloc_construct_ destruct.hh	195	1
jeod::jeod_alloc_destruct_array (std::jeod_alloc_destruct_ array (std::size_t nelem, void * addr)	include/jeod_alloc_construct_ destruct.hh	208	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodAllocHelper AllocatedPointer::cast (T * pointer)	include/jeod_alloc_get_allocated_pointer.hh	94	1
jeod::JeodAllocHelper AllocatedPointer::cast (T * pointer)	include/jeod_alloc_get_allocated_pointer.hh	114	1
jeod::std::jeod_alloc_get_allocated_pointer (T * pointer)	include/jeod_alloc_get_allocated_pointer.hh	133	1
jeod::JeodSimEngine Attributes::JeodSimulation Interface::attributes (bool)	include/memory_attributes_templates.hh	93	1
jeod::JeodSimEngine Attributes::JeodSimulation Interface::attributes (bool is_exportable = true)	include/memory_attributes_templates.hh	112	1
jeod::JeodSimEngine Attributes::JeodSimulation Interface::attributes (bool)	include/memory_attributes_templates.hh	132	1
jeod::JeodSimEngine Attributes::JeodSimulation Interface::attributes (bool is_exportable = true)	include/memory_attributes_templates.hh	150	2
jeod::JeodMemoryItem::get_nelems ()	include/memory_item.hh	224	1
jeod::JeodMemoryItem::get_unique_id ()	include/memory_item.hh	233	1
jeod::JeodMemoryItem::get_alloc_index ()	include/memory_item.hh	242	1
jeod::JeodMemoryItem::get_descriptor_index ()	include/memory_item.hh	251	1
jeod::JeodMemoryItem::get_placement_new ()	include/memory_item.hh	260	1
jeod::JeodMemoryItem::get_is_array ()	include/memory_item.hh	269	1
jeod::JeodMemoryItem::get_is_guarded ()	include/memory_item.hh	278	1
jeod::JeodMemoryItem::is_structured_data ()	include/memory_item.hh	287	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryItem::get_is_registered ()	include/memory_item.hh	296	1
jeod::JeodMemoryItem::get_checkpointed ()	include/memory_item.hh	305	1
jeod::JeodMemoryManager::TypeEntry (uint32_t num, const JeodMemoryTypeDescriptor * desc)	include/memory_manager.hh	262	1
jeod::JeodMemoryManager::get_type_descriptor_nolock (const JeodMemoryItem & item)	include/memory_manager.hh	553	1
jeod::JeodMemoryTable::JeodMemoryTable ()	include/memory_table.hh	142	1
jeod::JeodMemoryTable::~JeodMemoryTable ()	include/memory_table.hh	156	2
jeod::JeodMemoryTable::std::find (const std::string & key)	include/memory_table.hh	179	2
jeod::JeodMemoryTable::begin ()	include/memory_table.hh	209	1
jeod::JeodMemoryTable::end ()	include/memory_table.hh	217	1
jeod::JeodMemoryTable::std::add (const std::string & key, const ValueType & val)	include/memory_table.hh	225	2
jeod::JeodMemoryTable::std::del (const std::string & key)	include/memory_table.hh	255	3
jeod::JeodMemoryTable::JEOD_THROW(std::get (unsigned int idx)	include/memory_table.hh	291	4
jeod::JeodMemoryTableClonable::JeodMemoryTableClonable ()	include/memory_table.hh	349	1
jeod::JeodMemoryTableClonable::clone (const ValueType & value)	include/memory_table.hh	363	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryTable Copyable::JeodMemory TableCopyable ()	include/memory_table.hh	383	1
jeod::JeodMemoryTable Copyable::clone (const ValueType & value)	include/memory_table.hh	397	1
jeod::JeodMemoryReflective Table::JeodMemoryTable Copyable¡std::JeodMemory ReflectiveTable ()	include/memory_table.hh	415	1
jeod::JeodMemoryReflective Table::std::add (const std:: string & keyval)	include/memory_table.hh	436	1
jeod::JeodMemoryType Descriptor::set_check_for_ registration_errors (bool val)	include/memory_type.hh	102	1
jeod::JeodMemoryType Descriptor::std::get_typeid ()	include/memory_type.hh	136	1
jeod::JeodMemoryType Descriptor::std::get_name ()	include/memory_type.hh	145	1
jeod::JeodMemoryType Descriptor::std::get_size ()	include/memory_type.hh	154	1
jeod::JeodMemoryType Descriptor::get_attr ()	include/memory_type.hh	163	1
jeod::JeodMemoryType Descriptor::get_register_ instances ()	include/memory_type.hh	172	1
jeod::JeodMemoryType Descriptor::std:: dimensionality ()	include/memory_type.hh	183	1
jeod::JeodMemoryType Descriptor::std::buffer_size (unsigned int nelems)	include/memory_type.hh	192	1
jeod::JeodMemoryType Descriptor::std::buffer_size (const JeodMemoryItem & item)	include/memory_type.hh	202	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryType Descriptor::buffer_end (const void * addr, unsigned int nelems)	include/memory_type.hh	212	1
jeod::JeodMemoryType Descriptor::buffer_end (const void * addr, const JeodMemoryItem & item)	include/memory_type.hh	222	1
jeod::JeodMemoryType Descriptor::destroy_memory (bool placement_new, bool is_array, unsigned int nelem, void * addr)	include/memory_type.hh	235	3
jeod::JeodMemoryType DescriptorDerived:: Attributes::JeodMemory TypeDescriptorDerived (bool is_exportable = true)	include/memory_type.hh	389	1
jeod::JeodMemoryType DescriptorDerived::Jeod MemoryTypeDescriptor Derived (const Jeod MemoryTypeDescriptor Derived & src)	include/memory_type.hh	399	1
jeod::JeodMemoryType DescriptorDerived::clone ()	include/memory_type.hh	413	1
jeod::JeodMemoryType DescriptorDerived::std::is_ structured ()	include/memory_type.hh	424	1
jeod::JeodMemoryType DescriptorDerived:: construct_array(std:: construct_array (std::size_t nelem, void * addr)	include/memory_type.hh	433	1
jeod::JeodMemoryType DescriptorDerived::most_ derived_pointer (const void * addr)	include/memory_type.hh	441	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryType DescriptorDerived::most_ derived_pointer (void * addr)	include/memory_type.hh	451	1
jeod::JeodMemoryType DescriptorDerived::delete_ array (void * addr)	include/memory_type.hh	462	1
jeod::JeodMemoryType DescriptorDerived::delete_ object (void * addr)	include/memory_type.hh	474	1
jeod::JeodMemoryType DescriptorDerived:: destruct_array(std:: destruct_array (std::size_t nelem, void * addr)	include/memory_type.hh	485	1
jeod::JeodMemoryTypePre DescriptorDerived::Jeod MemoryTypePreDescriptor Derived (bool exportable = true)	include/memory_type.hh	541	1
jeod::JeodMemoryTypePre DescriptorDerived::Jeod MemoryTypePreDescriptor Derived (const Jeod MemoryTypePreDescriptor Derived & src)	include/memory_type.hh	549	2
jeod::JeodMemoryTypePre DescriptorDerived::~Jeod MemoryTypePreDescriptor Derived ()	include/memory_type.hh	561	1
jeod::JeodMemoryTypePre DescriptorDerived::get_ref ()	include/memory_type.hh	569	1
jeod::JeodMemoryTypePre DescriptorDerived::std::get_ typeid ()	include/memory_type.hh	588	1
jeod::JeodMemoryTypePre DescriptorDerived::get_ descriptor ()	include/memory_type.hh	597	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryItem::construct_flags (bool placement_new, bool is_array, bool is_guarded, bool is_structured)	src/memory_item.cc	39	5
jeod::JeodMemoryItem::JeodMemoryItem (bool placement_new, bool is_array, bool is_guarded, bool is_structured, unsigned int nelems_in, unsigned int type_idx, unsigned int alloc_idx)	src/memory_item.cc	74	1
jeod::JeodMemoryItem::set_unique_id (uint32_t id)	src/memory_item.cc	101	3
jeod::JeodMemoryItem::set_is_registered (bool value)	src/memory_item.cc	128	2
jeod::JeodMemoryManager::JeodMemoryManager (JeodMemoryInterface & interface)	src/memory_manager.cc	58	2
jeod::JeodMemoryManager::~JeodMemoryManager ()	src/memory_manager.cc	124	4
jeod::JeodMemoryManager::generate_shutdown_report ()	src/memory_manager.cc	172	5
jeod::JeodMemoryManager::restart_clear_memory ()	src/memory_manager.cc	265	4
jeod::JeodMemoryManager::restart_reallocate (const std::string & mangled_type_name, uint32_t unique_id, uint32_t nelements, bool is_array)	src/memory_manager.cc	309	2
jeod::JeodMemoryManager::create_memory_internal (bool is_array, unsigned int nelems, int fill, const TypeEntry & tentry, const char * file, unsigned int line)	src/memory_manager.cc	374	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryManager:: register_memory_internal (const void * addr, uint32_t unique_id, bool placement_ new, bool is_array, unsigned int nelems, const Type Entry & tentry, const char * file, unsigned int line)	src/memory_manager.cc	398	8
jeod::JeodMemoryManager:: is_allocated_internal (const void * addr, const char * file, unsigned int line)	src/memory_manager.cc	512	2
jeod::JeodMemoryManager:: destroy_memory_internal (void * addr, bool delete_ array, const char * file, unsigned int line)	src/memory_manager.cc	538	8
jeod::JeodMemoryManager:: set_mode_internal (Jeod SimulationInterface::Mode new_mode)	src/memory_manager.cc	661	1
jeod::JeodMemoryManager:: allocate_memory (std::size_t nelems, std::size_t elem_size, bool guard, int fill)	src/memory_manager.cc	689	4
jeod::JeodMemoryManager:: free_memory (void * addr, std::size_t length, bool guard, unsigned int alloc_ idx, const char * file, unsigned int line)	src/memory_manager.cc	760	9
jeod::JeodMemoryManager:: begin_atomic_block ()	src/memory_manager_ protected.cc	80	2
jeod::JeodMemoryManager:: end_atomic_block (bool ignore_errors)	src/memory_manager_ protected.cc	113	3
jeod::std::get_string_atomic (unsigned int idx)	src/memory_manager_ protected.cc	147	3

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryManager:: add_string_atomic (const std::string & str)	src/memory_manager_ protected.cc	179	2
jeod::JeodMemoryManager:: get_type_index_nolock (const JeodMemoryType Descriptor & tdesc, uint32_t * idx)	src/memory_manager_ protected.cc	227	2
jeod::JeodMemoryManager:: get_type_entry_atomic (JeodMemoryTypePre Descriptor & tdesc)	src/memory_manager_ protected.cc	261	5
jeod::JeodMemoryManager:: get_type_descriptor_atomic (const std::type_info & typeid_info)	src/memory_manager_ protected.cc	324	3
jeod::JeodMemoryManager:: get_type_entry_atomic (Jeod MemoryManager::Name Type name_type, const std:: string & type_name)	src/memory_manager_ protected.cc	363	6
jeod::JeodMemoryManager:: get_type_descriptor_atomic (unsigned int idx)	src/memory_manager_ protected.cc	432	3
jeod::JeodMemoryManager:: get_alloc_id_atomic (const char * file, unsigned int line)	src/memory_manager_ protected.cc	483	4
jeod::JeodMemoryManager:: reset_alloc_id_atomic (uint32_t unique_id)	src/memory_manager_ protected.cc	529	3
jeod::JeodMemoryManager:: find_alloc_entry_atomic (const void * addr, bool delete_entry, const char * file, unsigned int line, void *& found_addr, Jeod MemoryItem & found_item, const JeodMemoryType Descriptor *& found_type)	src/memory_manager_ protected.cc	558	8

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryManager:: add_allocation_atomic (const void * addr, const JeodMemoryItem & item, const JeodMemoryType Descriptor & tdesc, const char * file, unsigned int line)	src/memory_manager_ protected.cc	687	8
jeod::JeodMemoryManager:: delete_oldest_alloc_entry_ atomic (void *& addr, Jeod MemoryItem & item, const JeodMemoryType Descriptor *& type)	src/memory_manager_ protected.cc	789	5
jeod::JeodMemoryManager:: check_master (bool error_is_ fatal, int line)	src/memory_manager_ static.cc	51	3
jeod::JeodMemoryManager:: set_debug_level (Debug Level level)	src/memory_manager_ static.cc	80	2
jeod::JeodMemoryManager:: set_debug_level (unsigned int level)	src/memory_manager_ static.cc	94	2
jeod::JeodMemoryManager:: set_guard_enabled (bool value)	src/memory_manager_ static.cc	110	2
jeod::JeodMemoryManager:: is_table_empty ()	src/memory_manager_ static.cc	124	2
jeod::JeodMemoryManager:: register_class (JeodMemory TypePreDescriptor & tdesc)	src/memory_manager_ static.cc	146	2
jeod::JeodMemoryManager:: get_type_descriptor (const std::type_info & typeid_ info)	src/memory_manager_ static.cc	174	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryManager:: get_type_descriptor (Jeod MemoryManager::Name Type name_type, const std:: string & type_name)	src/memory_manager_ static.cc	198	2
jeod::JeodMemoryManager:: create_memory (bool is_ array, unsigned int nelems, int fill, const TypeEntry & tentry, const char * file, unsigned int line)	src/memory_manager_ static.cc	224	2
jeod::JeodMemoryManager:: is_allocated (const void * addr, const char * file, unsigned int line)	src/memory_manager_ static.cc	258	2
jeod::JeodMemoryManager:: destroy_memory (void * addr, bool delete_array, const char * file, unsigned int line)	src/memory_manager_ static.cc	284	2
jeod::JeodMemoryManager:: register_container (const void * container, const std:: type_info & container_type, const std::string & elem_ name, JeodCheckpointable & checkpointable)	src/memory_manager_ static.cc	310	3
jeod::JeodMemoryManager:: deregister_container (const void * container, const std:: type_info & container_type, const std::string & elem_ name, JeodCheckpointable & checkpointable)	src/memory_manager_ static.cc	356	3
jeod::JeodMemoryManager:: set_mode (JeodSimulation Interface::Mode new_mode)	src/memory_manager_ static.cc	402	2
jeod::std::initialize_type_name (const std::string & type_ name)	src/memory_type.cc	51	3

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodMemoryType Descriptor::pointer_ dimension (const std::string & demangled_name)	src/memory_type.cc	71	3
jeod::JeodMemoryType Descriptor::base_type (const std::string & demangled_ name)	src/memory_type.cc	93	10
jeod::JeodMemoryType Descriptor::JeodMemory TypeDescriptor (const std:: type_info & obj_typeid, const JEOD_ATTRIBUTE S_TYPE & type_attr, std:: size_t type_size, bool is_ exportable)	src/memory_type.cc	164	1
jeod::std::type_spec (const JeodMemoryItem & item)	src/memory_type.cc	185	2

5.4 Requirements Traceability

Table 5.4 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the model.

Table 5.4: Requirements Traceability

Requirement	**Inspection or test**
Memory_1 Project Requirements	Insp. Memory_1 Top-level Inspection
Memory_2 Memory Allocation	Insp. Memory_2 Design Inspection Test Memory_1 Basic Test Test Memory_2 Threading Test
Memory_3 Memory Registration	Insp. Memory_2 Design Inspection Test Memory_1 Basic Test
Memory_4 Memory Deallocation	Insp. Memory_2 Design Inspection Test Memory_1 Basic Test Test Memory_2 Threading Test
Memory_5 Free from Base Pointer	Insp. Memory_2 Design Inspection Test Memory_1 Basic Test Test Memory_2 Threading Test
Memory_6 Memory Deregistration	Insp. Memory_2 Design Inspection Test Memory_1 Basic Test
Memory_7 Thread Safety	Insp. Memory_2 Design Inspection Insp. Memory_3 Peer Review Test Memory_2 Threading Test
Memory_8 Memory Tracking	Insp. Memory_2 Design Inspection Test Memory_2 Threading Test
Memory_9 Memory Reporting	Insp. Memory_2 Design Inspection Test Memory_2 Threading Test

Bibliography

[1] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, February 2025.

[2] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[3] Shelton, R. Message Handler Model. Technical Report JSC-61777-utils/message, NASA, Johnson Space Center, Houston, Texas, February 2025.

[4] Shelton, R. Named Item Model. Technical Report JSC-61777-utils/named_item, NASA, Johnson Space Center, Houston, Texas, February 2025.

[5] Shelton, R. Simulation Engine Interface Model. Technical Report JSC-61777-utils/sim_interface, NASA, Johnson Space Center, Houston, Texas, February 2025.