JSC-61777-models/utils/sim_interface

# JSC Engineering Orbital Dynamics Simulation Engine Interface Model

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

**Package Release JEOD v5.1**

**Document Revision 2.0**
**July 2023**

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

JSC-61777-models/utils/sim_interface

# JSC Engineering Orbital Dynamics Simulation Engine Interface Model

**Document Revision 2.0**
**July 2023**

**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**

## Abstract

The Simulation Interface Model provides a generic bridge between the models which comprise JEOD and a simulation environment where they operate. The Simulation Interface Model uses abstract classes to specify contracts between JEOD models and message handling, memory allocation and integration services provided by the simulation environment. There is also a suite of preprocessor macros which provide the simulation environment access to protected fields within JEOD classes for the purposes of initialization and logging.

The Simulation Interface Model includes Trick-specific implementations of all required interfaces as well as conditional definitions for the macros appropriate for use with Trick. All JEOD simulations require instantiation of exactly one JeodSimulationInterface class in order to function.

# Contents

# List of Tables

# Chapter 1

# Introduction

## 1.1 Purpose and Objectives of the Simulation Engine Interface Model

The Simulation Interface Model provides a generic bridge between the other models that comprise JEOD and a simulation environment where those models are used. The Simulation Interface Model uses abstract classes to specify contracts between JEOD models and message handling, memory allocation and integration services provided by the simulation environment. There is also a suite of preprocessor macros which provide the simulation environment access to protected fields within JEOD classes for the purposes of initialization and logging.

The Simulation Interface Model includes Trick-specific implementations of all required interfaces as well as conditional definitions for the macros appropriate for use with Trick. All JEOD simulations require instantiation of exactly one JeodSimulationInterface class in order to function.

## 1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [6]

The Simulation Engine Interface Model forms a component of the utilities suite of models within JEOD v5.1. It is located at models/utils/sim_interface.

## 1.3 Document History

| Author | Date | Revision | Description |
|---|---|---|---|
| David Hammen | February 2012 | 2.0 | Jeod 2.2 release |
| Robert Shelton | September 2010 | 1.0 | JEOD 2.1 release |

## 1.4 Document Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [7].

The document comprises chapters organized as follows:

**Chapter 1: Introduction** -This introduction describes the objective and purpose of the Simulation Engine Interface Model.

**Chapter 2: Product Requirements** -The requirements chapter describes the requirements on the Simulation Engine Interface Model.

**Chapter 3: Product Specification** -The specification chapter describes the architecture and design of the Simulation Engine Interface Model.

**Chapter 4: User Guide** -The user guide chapter describes how to use the Simulation Engine Interface Model.

**Chapter 5: Inspections, Tests, and Metrics** -The inspections, tests, and metrics describes the procedures and results that demonstrate the satisfaction of the requirements for the Simulation Engine Interface Model.

# Chapter 2

# Product Requirements

*Requirement SIMINT_1: Top-level requirement*

**Requirement:**

The Simulation Interface Model shall meet the JEOD project requirements specified in the JEOD v5.1 JEOD top-level document.

**Rationale:**

This is a project-wide requirement.

**Verification:**

Inspection

*Requirement SIMINT_2: Hidden Data Visibility*

**Requirement:**

The Simulation Interface Model shall provide a generic (simulation engine agnostic) mechanism by which a model developer can make protected and private data visible to the simulation engine.

**Rationale:**

Simulation users at times need access to protected and private data. As one simulation user said, “Murphy’s Law laughs at and cosmic rays don’t obey data encapsulation. I need to see, and sometimes modify, your hidden data. Tell me I can’t have such access and my project won’t use your tool.”

Another motivating factor is the need to make JEOD checkpointable and restartable. The Trick simulation engine do most of the work with regard to checkpoint and restart if hidden data are visible to Trick. Making hidden data visible drastically reduces the burden of making JEOD checkpointable and restartable.

**Verification:**

Inspection, Test

*Requirement SIMINT_3: Allocated Data Visibility*

**Requirement:**

The Simulation Interface Model shall provide a generic (simulation engine agnostic) mechanism by which data allocated via the Memory Management Model can be made visible to the simulation engine.

**Rationale:**

The same factors that motivated requirement SIMINT_2 apply to this requirement.

**Verification:**

Inspection, Test

*Requirement SIMINT_4: Simulation Engine Interface*

**Requirement:**

With one exception, the Simulation Interface Model shall be the only JEOD model that directly invokes simulation engine functions or that directly uses simulation engine data constructs.

**Exception:**

The Dynamics Manager Model is allowed to use Trick 7 specific data constructs with regard to integration.

**Rationale:**

The need to support the Trick environment resulted in an ever-growing number of Trick-specific constructs sprinkled throughout JEOD. These widely dispersed Trick-specific constructs threatened to make JEOD unusable outside of the Trick environment. This growth in Trick-specific constructs and the associated risk of failing to be usable outside of Trick were the key factors that motivated the the creation of the Simulation Interface Model.

**Verification:**

Inspection, Test

*Requirement SIMINT_5: Integration Interface*

**Requirement:**

The Simulation Interface Model shall provide:

*5.1 Generic.* Simulation engine agnostic mechanisms to make JEOD integration operable within a generic simulation environment.

*5.2 Trick specific.* Trick-specific implementations of these mechanisms to make JEOD integration operable within the Trick simulation environment.

**Rationale:**

This is a derived requirement of requirement SIMINT_4.

JEOD integration can be used in simulation engines such as Trick that treat integration as an abstract process, but this requires communications between integration as performed by JEOD and integration as performed by the simulation engine.

**Verification:**
Inspection, Test

*Requirement SIMINT_6: Job Cycle*

**Requirement:**
The Simulation Interface Model shall provide a mechanism to obtain the time interval between successive calls by the simulation engine to the currently executing function.

**Rationale:**
This is a derived requirement of requirement SIMINT_4.

The physical processes modeled in some JEOD models require knowledge of the rate at which the process model is called.

**Verification:**
Inspection, Test

*Requirement SIMINT_7: Trick Message Handler*

**Requirement:**
The Simulation Interface Model shall provide a Trick-based implementation of the abstract class MessageHandler.

**Rationale:**
This is a derived requirement of requirement SIMINT_4.

None of the verification simulations would build without a functional message handler.

**Verification:**
Inspection, Test

*Requirement SIMINT_8: Checkpoint/ Restart*

**Requirement:**
The Simulation Interface Model shall provide mechanisms that enable JEOD-based simulations to be checkpointable and restartable in the Trick 10 environment.

**Rationale:**
This is a requirement mandated by external users of JEOD.

**Verification:**
Inspection, Test

*Requirement SIMINT_9: Address/Name Translation*

**Requirement:**
The Simulation Interface Model shall provide mechanisms to translate an address to a simulation engine symbolic name and to translate a symbolic name to an address.

**Rationale:**
This is a derived requirement of requirements SIMINT_4 and SIMINT_8, levied on the model by the Container Model.

**Discussion:**
This capability is only used when the simulation interface object indicates that checkpoint/restart are enabled. An implementation that does not support checkpoint/restart can provide dummy implementations of this functionality.

**Verification:**
Inspection, Test

*Requirement SIMINT_10: Multiple Integration Groups*

**Requirement:**
The Simulation Interface Model shall provide a multiple integration group capability in the Trick 10 environment. In particular,

*10.1 Multiple rates.* The model shall provide the ability to integrate different dynamic bodies at different rates / with different integration techniques.

*10.2 Switchability.* The model shall provide the ability to move a dynamic body from one integration group to another.

**Rationale:**
This is a requirement mandated by external users of JEOD.

**Verification:**
Inspection, Test

*Requirement SIMINT_11: Extensibility*

**Requirement:**
The Simulation Interface Model shall be architected in such a manner that allows it to be applied to simulation engines other than Trick.

**Rationale:**
That JEOD is to be usable outside of the Trick environment is a top-level JEOD requirement.

**Verification:**
Inspection

# Chapter 3

# Product Specification

## 3.1 Conceptual Design

### 3.1.1 Overview

JEOD must be usable in the Trick 13 environment, and should be usable in other environments as well. The decision to make the Simulation Interface Model was mostly an internal decision. The model consolidates the interfaces between JEOD and Trick, and potentially with other simulation engines, in one place.

The Simulation Interface Model is implemented in the form of header files that configure JEOD for use in a particular simulation environment, header files the define macros, and header and source files that define classes. The common theme that connects these disparate items is that they represent interfaces between JEOD and the simulation engine. These interfaces take a number of forms:

- Making a class's protected and private data members visible to the simulation engine.
- Making data items allocated by JEOD models visible to the simulation engine.
- Translating addresses to and from simulation engine symbolic names.
- Checkpointing and restarting data allocations and Container Model objects.
- Making JEOD integration work within the context of a simulation engine's integration scheme.
- Obtaining the rate at which the simulation engine calls the currently executing function.

The Trick environment is the primary implementation target of JEOD. A number of the model header and source files pertain specifically to the Trick simulation environment.

In addition to the above interfaces, the model provides a Trick 13-specific capability to support multiple rates of integration.

### 3.1.2 Configuration

Some of the header files provided with the model configure JEOD for use in a particular simulation environment. These configuration headers define a set of C++ macros. Some of these macros are used in other model header and source files. Others provide portable type definitions that can be used by model developers.

### 3.1.3 Exposing Protected/Private Data Members to the Simulation Engine

All JEOD models must be restartable from a previously created set of checkpoint files. JEOD models rely primarily upon Trick 13 mechanisms to provide this checkpoint / restart capability. Making the protected and private data members of a class are visible to Trick enables Trick to checkpoint a large portion of the members of all class instances that are visible to Trick.

To this end, the Simulation Interface Model provides two macros, `JEOD_MAKE_SIM_INTERFACES` and `JEOD_NOMINALLY_PRIVATE`. The former makes all of a class's data members visible to the simulation engine. All JEOD classes that have protected or private member data invoke this macro. The latter is for templates that are base classes of various JEOD classes.

### 3.1.4 Interfacing JEOD and Simulation Engine Integration

JEOD has a well-developed concept of how to integrate the equations of motion, but so do many simulation engines. The Simulation Interface Model provides mechanisms to make JEOD integration work within the context of how the simulation engine performs integration.

### 3.1.5 Interfacing JEOD and Simulation Engine Memory Management

The constructors and initialization-time functions of a number of the JEOD models allocate memory dynamically. These allocations, along with registrations of containers that need to be checkpointed and restarted, are the purview of the JEOD Memory Management Model. The allocations and type descriptions must be coordinated with the simulation engine. The Simulation Interface Model relays information about the allocations to the simulation engine and provides the memory model with simulation engine descriptions of data types.

### 3.1.6 Checkpoint/Restart

JEOD checkpoint/restart capabilities come in two basic forms: A generic checkpoint/restart text-based capability, and a Trick 13-specific checkpoint restart capability that uses this generic capability to augment Trick 13 checkpoint restart.

The generic capability extends C++ I/O to provide the ability to write and read a partitioned checkpoint file. Each partition in the file looks like a separate file from the perspective of start and end of file markers. This partitioning reduces the number of files that need to be managed.

The Trick 13-specific capability augments existing Trick 13 text-based checkpoint/restart, writing checkpoint data to a pair of partitions in a partitioned JEOD checkpoint file during checkpoint and restoring content from that checkpoint file on restart. While Trick 13 can checkpoint and restore the contents of externally allocated data, it cannot checkpoint and restore the allocations themselves. One of the partitions in the JEOD checkpoint file describes the data allocated by JEOD models. The Container Model Checkpointable objects used throughout JEOD are opaque to the simulation engine. The other partition in the JEOD checkpoint file contains the contents of those Checkpointable objects.

#### 3.1.7 Trick Message Handler

The `MessageHandler` class is an abstract class. The Simulation Interface Model provides a Trick-specific classes that derive from the `MessageHandler` and that are instantiable.

#### 3.1.8 Multiple Integration Groups

Providing the ability to support multiple rates of integration, with each rate group using its own integration technique, has been a long-standing desire within the JEOD project. Until recently the Trick environment limited the extent to which this could be accomplished and there was no external impetus to provide this ability. Trick 13 now provides the necessary infrastructure to support this feature, and an external request for this feature now exists. The Simulation Interface Model provides a Trick 13-specific multiple integration group capability.

## 3.2 Key Algorithms

#### 3.2.1 Checkpoint/Restart

The JEOD project assumed that the bulk of the checkpoint and restart efforts are provided by the simulation engine. The two exceptions to this are data allocated by JEOD and classes with data members that are opaque to the simulation engine.

The Simulation Interface Model addresses these issues by recording the allocations and opaque content in a segmented JEOD checkpoint file. The allocations and opaque content are restored at restart time based on the contents of the specified JEOD checkpoint file. The algorithms for checkpointing and restoring Checkpointable objects is specified in the Container Model.

#### 3.2.2 Trick Name Lookup

JEOD uses Trick name lookup to convert addresses to names at checkpoint time and to convert names to addresses at restart time. The mechanisms for performing these activities were the result of mutually agreed upon negotiations between the Trick and JEOD development teams.

#### 3.2.3 Trick Memory Registration

JEOD registers allocated memory and deregisters freed memory with Trick per interfaces described in the Trick documentation.

## 3.3 Interactions

### 3.3.1 JEOD Models Used by the Simulation Interface Model

The Simulation Interface Model uses the following JEOD models:

- *Container Model* [1].
  The Trick 13 memory interface maintains a registry of Container Model `JeodCheckpointable` objects. The model uses this registry as the basis for checkpointing the contents of the checkpointable objects at checkpoint time and for restoring the contents of the checkpointable objects at restart time.

- *Dynamics Manager Model* [2] and *Dynamic Body Model* [3].
  The JEOD-agnostic aspect of the Trick 13-specific multiple integration group capability integrates a collection of Trick 13 simulation objects. The JEOD-aware aspect of this capability augments this JEOD-agnostic capability, using a `DynamicsIntegrationGroup` object to integrate all of the `DynBody` objects contained in the subject simulation objects. The set of `DynBody` to be integrated is formed by interactions with the simulation's `DynManager` object.

- *Integration Model* [4].
  The Trick simulation engine treats integration a special class of jobs. Trick makes integration abstract. It repeatedly calls derivative class jobs and then integration class jobs until the integration jobs indicate completion. The Simulation Interface Model enables communication between the simulation engine and JEOD integration in the form of the abstract class IntegratorInterface and Trick-specific classes that derive from IntegratorInterface.

- *Memory Management Model* [5].
  Every JEOD-based simulation must contain exactly one instance of a class that derives from `JeodSimulationInterface`. This class must contain or construct an instance of a class that derives from `JeodMemoryManager`.

  The Simulation Interface Model reacts to calls from the memory manager regarding allocations/deallocations of memory and registrations/deregistrations of checkpointable objects.

- *Message Handler Model* [8].
  The Simulation Interface Model mandates that a compliant `JeodSimulationInterface` must either contain or construct an instance of a class that derives from `MessageHandler`. This message handler must be created prior to the memory manager described above.

  As do almost all other models, the Simulation Interface Model uses the message handler to generate messages.

- *Named Item Model* [9].
  The Trick 13-specific memory interface calls `NamedItem::demangle` to demangle type names.

- *Time Model* [11].
  The JEOD-aware aspect of the Trick 13-specific multiple integration group capability uses the simulation's `TimeManager` object to create the integration group and to update time at the start of an integration loop.

### 3.3.2 JEOD Models That Use the Simulation Interface Model

All JEOD models use the Simulation Interface Model to make data members visible to the simulation engine. In addition to these uses,

- *Container Model* [1].
  The Container Model uses the Simulation Interface Model to translate addresses to and from symbolic names.
- *Dynamics Manager Model* [2].
  The Dynamics Manager Model integrates the dynamic aspects of a typical JEOD-based simulation. It uses the Integration Model to perform the integration, with the simulation engine mediating the integration process. Interactions between JEOD simulation and the simulation engine are handled generically by the Simulation Interface Model class `JeodIntegratorInterface`.

  The `DynManager` class contains an element of type `JEOD_SIM_INTEGRATOR_POINTER_TYPE`. This macro defines a pointer to a simulation engine-specific integrator structure, `Trick::Integrator` in the case of a Trick-based simulation. The Trick integration class job that calls the `DynManager` instance's integrate function should supply this member as the `sup\_class\_data` for the integration class job.
- *Integration Model* [4].
  The Integration Model uses a Simulation Interface Model `JeodIntegratorInterface` object to coordinate integration as performed by JEOD with integration as performed by the simulation engine.
- *Memory Management Model* [5].
  The Memory Model uses the Simulation Interface Model to
  - Coordinate JEOD and simulation engine descriptions of data types,
  - Register/deregister allocated memory with the simulation engine, and
  - Register/deregister Container Model objects with the Simulation Interface Model.

## 3.4 Detailed Design

### 3.4.1 Configuration

The header file `config.hh` includes an installation-specific configuration file that defines a set of macros used in other model headers. The model provides installation-specific configuration files for Trick 13 and for the Trick-independent test harness.

Developers who wish to use JEOD in a non-Trick simulation environment must develop a custom installation-specific configuration file.

### 3.4.2 Exposing Protected/Private Data Members to the Simulation Engine

In a Trick environment, the macro `JEOD_MAKE_SIM_INTERFACES` declares `InputProcessor` as a friend class and declares the auto-generated `init_attr<class_name>` as a friend function. Declaring `InputProcessor` as a friend class signals Trick's ICG facility to process protected and private data members. Declaring the function generated by ICG as a friend function is needed to make that auto-generated code compilable.

Developers who wish to use JEOD in a non-Trick simulation environment that requires visibility to protected and private data must define a custom version of `JEOD_MAKE_SIM_INTERFACES`.

### 3.4.3 The JEOD Simulation Interface Object

Every JEOD-based simulation must have exactly one instance of a class that that inherits from the abstract class `JeodSimulationInterface`.

#### 3.4.3.1 Class JeodSimulationInterface

This abstract class defines the basis for the interface between JEOD and a simulation engine. A compliant derived class must contain one instance each of a class that derives from `MessageHandler` and a class that derives from `JeodMemoryManager`. The `MessageHandler` object must be constructed before the `JeodMemoryManager` object; destruction must be performed in reverse order.

#### 3.4.3.2 Class JeodSimulationInterfaceInit

This class defines configuration data that can be used to configure the `JeodSimulationInterface` object's message handler and memory manager data members.

#### 3.4.3.3 Class BasicJeodTrickSimInterface

This class implements the required capabilities of the generic `JeodSimulationInterface` in a Trick simulation environment. The class contains a reference to a message handler, a memory interface object, and a memory manager object. The message handler, being a reference, must be created before the `BasicJeodTrickSimInterface` object is constructed.

The Trick 13 `JEODSysSimObject` provided with JEOD contains a `BasicJeodTrickSimInterface` data member. The MessageHandler object, which defaults to a `TrickMessageHandler` object, is passed as an argument to the `BasicJeodTrickSimInterface` non-default constructor.

#### 3.4.3.4 Class JeodTrickSimInterface

This class derives from `TrickMessageHandlerMixin` and `BasicJeodTrickSimInterface`. The order of the inheritance ensures that the message handler is constructed prior to the memory interface and the memory manager members. Note that because the class `TrickMessageHandlerMixin` derives from the the TrickMessageHandler class, using a `JeodTrickSimInterface` precludes the use of a custom message handler. Developers who want to use their own message handler should use the `BasicJeodTrickSimInterface` rather than the `JeodTrickSimInterface`.

### 3.4.4 Interfacing JEOD and Simulation Engine Integration

#### 3.4.4.1 Configuration Macros

The configuration file should define a series of macros that encapsulate concepts related to integration. These integration configuration macros are:

`JEOD_SIM_INTEGRATOR_FORWARD` provides a forward declaration to the simulation engine's integration data structure. This macro does not need to be defined if such a forward declaration isn't needed. In a Trick setting, this macro is defined via
`#define JEOD_SIM_INTEGRATOR_FORWARD namespace Trick { class Integrator; }`

`JEOD_SIM_INTEGRATOR_POINTER_TYPE` defines a pointer type that points to an instance of the simulation engine's integration data structure. In a Trick setting, this macro is defined via
`#define JEOD_SIM_INTEGRATOR_POINTER_TYPE Trick::Integrator *`

`JEOD_SIM_INTEGRATOR_ENUM` defines an enumeration or integral type that identifies the type used by the simulation engine to identify an integration technique. In a Trick setting, this macro is defined via
`#define JEOD_SIM_INTEGRATOR_ENUM Integrator_type`

#### 3.4.4.2 Class JeodIntegratorInterface

This abstract class derives from the `er7_utils::IntegratorInterface` class. The base class defines interfaces that the ER7 Utilities Integration model requires of an interface to the simulation engine's integration module. The class `JeodIntegratorInterface` adds two additional pure virtual interfaces used within JEOD.

Developers who wish to use JEOD in a non-Trick simulation environment must define a custom class that derives from `JeodIntegratorInterface` to interface their simulation engine's concept of integration with JEOD's concept of integration.

### 3.4.5 Class TrickJeodIntegrator

This class extends the `Trick::Integrator` class for internal use by JEOD. The class is needed because the Trick integration loop requires that integration class jobs be associated with a Trick integration structure. The loop uses this structure for various control purposes such as determining whether the derivative jobs are to be called on the first pass through the integration loop.

The class `Trick::Integrator` declares two pure virtual member functions, `initialize` and `integrate`. Even though these functions are never used by JEOD integration, they must be defined to make the class instantiable. The class `TrickJeodIntegrator` implements these as dummy functions.

### 3.4.6 Class JeodTrickIntegrator

This class derives from the `JeodIntegratorInterface` class. It implements all pure virtual interfaces declared in and inherited by the parent class. The class also contains a `TrickJeodIntegrator` data member. This inheritance and containment enables a `JeodTrickIntegrator` instance to be used as a common basis for Trick and ER7 Utilities integration.

### 3.4.7 Interfacing JEOD and Simulation Engine Memory Management

#### 3.4.7.1 Class JeodMemoryInterface

This abstract class specifies several pure virtual methods that are needed to interface the JEOD memory manager with the simulation engine. These include

- Functions pertaining to the JEOD versus simulation engine representations of data types.
- Functions for registering/deregistering JEOD data allocations with the simulation engine.
- Functions for translating addresses to and from symbolic names.
- Functions for registering/deregistering JEOD container objects with the simulation interface.
- Functions for checkpointing/restoring those registered container objects at checkpoint or restart time.

Developers who wish to use JEOD in a non-Trick simulation environment must define a custom class that derives from JeodMemoryInterface to interface their simulation engine's concept of memory with JEOD.

#### 3.4.7.2 Class JeodTrickMemoryInterface

This class derives from `JeodMemoryInterface`. It provides implementations of all of the pure virtual functions declared in the parent class. Several of these are dummy implementations based on the assumption that a `JeodSimulationInterface` by default is not checkpointable/restartable.

#### 3.4.7.3 Class JeodTrick10MemoryInterface

This class derives from `JeodTrickMemoryInterface`. It overrides the dummy implementations provided in `JeodTrickMemoryInterface` to make a Trick 13 / JEOD-based simulation fully checkpointable and restartable.

### 3.4.8 Checkpoint/Restart

This subsection describes the classes that provide a generic checkpoint/restart capability to create and restore from a partitioned checkpoint file. A partitioned checkpoint file comprises comprises multiple sections delineated by section markers, something along the lines of the following:

```
// ++++++++++++++++++++++++++++++++++++++++++ Start of section foo
Contents of section foo (multiple lines)
```

```
// ------------------------------------------ End of section foo

// ++++++++++++++++++++++++++++++++++++++++++ Start of section bar
Contents of section bar (multiple lines)
// ------------------------------------------ End of section bar

// ++++++++++++++++++++++++++++++++++++++++++ Start of section baz
Contents of section baz (multiple lines)
// ------------------------------------------ End of section baz
```

A section within a checkpoint file looks like a "file" to the external users of this capability. The writers automatically create the section markers when an output section is opened and later closed. The readers make the individual sections look like a "file". The pseudo file appears to start just after the start of the section marker and end just before the end of section marker.

The classes that comprise this capability take advantage of the fact that C++ I/O streams and buffers, unlike the C++ containers, were designed for extensibility. Users of this capability can use the standard C++ stream insertion and stream extraction operators to write to and read from a section of the checkpoint file.

#### 3.4.8.1 Class CheckPointOutputManager

The class `CheckPointOutputManager` provides the ability to create an output partitioned checkpoint file. The non-default constructor creates a C++ output file stream to the specified checkpoint file. This stream is private to the `CheckPointOutputManager` instance. This stream is accessed indirectly via a `SectionedOutputStream` object. The `CheckPointOutputManager` member function `create_section_writer` creates a `SectionedOutputStream`.

#### 3.4.8.2 Class SectionedOutputStream

A `SectionedOutputStream` is a `std::ostream` that writes a section of a partitioned checkpoint file. This class automatically writes the start and end markers of the checkpoint file section.

The current implementation is a bit spare. It does not provide buffering, and it does not support `seekp` or `tellp` to reposition the stream. It does support the stream insertion operator. It is this operator that external users should use to write the contents of the checkpoint file section.

Note that most of the content of this class is private. This class is not extensible and is intended to be used within the context of a `CheckPointOutputManager`.

#### 3.4.8.3 Class SectionedOutputBuffer

A `SectionedOutputBuffer` is a `std::streambuf` that writes a section of a partitioned checkpoint file. This stream buffer is created and used by a `SectionedOutputStream` object.

Note that with the exception of the destructor and the inherited members from `std::streambuf`, everything in this class is private. This class is not extensible.

#### 3.4.8.4 CheckPointInputManager

A `CheckPointInputManager` provides the ability to read from a previously created partitioned checkpoint file. The non-default constructor creates a C++ input file stream to the specified checkpoint file. This stream is private to the `CheckPointInputManager` instance. This stream is accessed indirectly via a `SectionedInputStream` object. The `CheckPointOInputManager` member function `create_section_reader` creates a `SectionedInputStream`.

#### 3.4.8.5 SectionedInputStream

A `SectionedInputStream` is a `std::istream` that reads from a section in a partitioned checkpoint file. This class indicates EOF when the input pointer in the checkpoint file file buffer reaches the end of the section.

The current implementation is a bit spare. It does not provide buffering, it does not support operations that require buffering such as `peek`, `putback`, and `unget`, and it does not support `seekg` or `tellg` to reposition the stream. It does support `std::getline` the stream extraction operator. It is this function or this operator that external users should use to read the contents of the checkpoint file section.

Note that most of the content of this class is private. This class is not extensible and is intended to be used within the context of a CheckPointInputManager.

#### 3.4.8.6 SectionedInputBuffer

A `SectionedInputBuffer` is a `std::streambuf` that reads from a section in a partitioned checkpoint file. This class indicates EOF when the input pointer in the checkpoint file file buffer reaches the end of the section.

Note that with the exception of the destructor and the inherited members from `std::streambuf`, everything in this class is private. This class is not extensible.

### 3.4.9 Trick Message Handler

#### 3.4.9.1 TrickMessageHandler

This class is an instantiable class that derives from the `SuppressedCodeMessageHandler` class, which in turn derives from the `MessageHandler` class. The `TrickMessageHandler` provides an implementation of the `process_message` member function declared as pure virtual in the base `MessageHandler` class.

JEOD classes that contain container objects should in general register those containers with the simulation interface at construction time. The TrickMessageHandler cannot do that because it needs to be constructed prior to the memory interface and memory manager. The class instead provides the method `register_contents` which is called by the `BasicJeodTrickSimInterface` constructor.

#### 3.4.9.2 TrickMessageHandlerMixin

This class contains a `TrickMessageHandler` as a protected data member. The class `JeodTrickSimInterface` inherits from this class and from `BasicJeodTrickSimInterface`, in that order.

### 3.4.10 Multiple Integration Groups

The Simulation Interface Model provides a Trick 13-specific multiple integration group capability. In JEOD 2.2, this capability was split into a JEOD-agnostic and JEOD-aware parts. The JEOD-agnostic part is now a part of Trick 13. The only aspect left in JEOD is the JEOD-aware part of the JEOD 2.2 implementation.

#### 3.4.10.1 JeodDynbodyIntegrationLoop

This class inherits from the `Trick::IntegLoopScheduler` class and from the Integration Model `IntegrationGroupOwner` class to provide the JEOD-aware aspect of the multiple integration group capability. A `JeodDynbodyIntegrationLoop` object integrates all of the `DynBody` objects that are contained in the sim objects to be integrated by that `JeodDynbodyIntegrationLoop` object.

A DynBody object that is an element of a Trick sim object is automatically moved from one integration loop to another when the sim object that contains the DynBody is moved to a different integration loop. Note well: JEOD does not support the concept of DynBody objects that are not members of a Trick simulation object. DynBody objects that are dynamically allocated outside of the context of a containing Trick simulation object is undefined behavior.

A `JeodDynbodyIntegrationLoop` object also integrates all non-DynBody integrable objects managed by the loop. These non-DynBody integrable objects can be added to or removed from the integration loop directly via calls to `JeodDynbodyIntegrationLoop::add_integrable_object` and `JeodDynbodyIntegrationLoop::remove_integrable_object`.

The recommended practice is to instead associate integrable objects with a DynBody object. The integrable objects associated with a DynBody object are automatically added to the dynamics integration group that integrates the DynBody object. In a future release or patch to JEOD, these associated integrable objects will be moved along with the DynBody object as the DynBody object is moved from one integration loop to another.

## 3.5 Inventory

All Simulation Engine Interface Model files are located in `${JEOD_HOME}/models/utils/sim_interface`. Relative to this directory,

- Model header and source files are located in model `include` and `src` subdirectories. See table **??** for a list of these configuration-managed files.
- Model documentation files are located in the model `docs` subdirectory. See table **??** for a list of the configuration-managed files in this directory.

# Chapter 4

# User Guide

## 4.1 Instructions for Simulation Users

Simulation users interact with the model by means of the `jeod_sys` simulation object, which contains the singular simulation interface object, the singular message handler object, and the singular memory manager object. See section 4.2 for details on this object.

The members of this simulation object mostly function without user intervention. This section describes those cases where user intervention can be used.

### 4.1.1 Memory Manager

User control over the memory manager is limited to specifying the memory manager debug level. The minimal setting results in the memory manager reporting serious errors only. The maximal setting causes every transaction to be reported.

Refer to the User Guide chapter of *Memory Management Model* [5] for details.

### 4.1.2 Message Handler

Users can direct the message handler to suppress messages that are below some severity threshold and can to some extent control how messages are formatted. Refer to the User Guide chapter of *Message Handler Model* [8] for details.

The Trick-specific message handler provided with this model adds the ability to suppress messages by their code. For example, the message handler reports JEOD-allocated memory that has not been freed by the time the memory manager is destroyed with the code `utils/memory/corrupted_memory`. Messages tagged with this code can be suppressed from the Python input file via

```
jeod_sys.message_handler.add_suppressed_code("utils/memory/corrupted_memory")
```

### 4.1.3 Checkpoint/Restart

JEOD is checkpointable and restartable in the Trick 13 environment. The standard `jeod_sys` simulation object provided with JEOD registers jobs that cause the Simulation Interface Model to create a JEOD checkpoint file whenever a Trick checkpoint file is created. The Simulation Interface Model saves JEOD content to this JEOD checkpoint file as a part of the Trick checkpoint process.

The `jeod_sys` simulation object similarly registers jobs that cause the Simulation Interface Model to restore JEOD content from a JEOD checkpoint file as a part of the restart procress. The Simulation Interface Model restores JEOD content from a JEOD checkpoint file as a part of the Trick restart process.

Trick provides several mechanisms for creating checkpoint files and for restoring from a checkpoint. A set of post-initialization checkpoint files can be created via

```
trick_mm.mmw.set_post_init_checkpoint(1)
```

Post-initialization and end checkpoint files can be created via similar mechanisms.

This class of checkpoint files typically are not useful for restart. Checkpoint files suitable for restart are created by Trick events at scheduled to run at either the beginning or end of a Trick frame. The following is the code used to create a checkpoint file in the `SET_test/RUN_prop_checkpoint` run of the *Ephemerides Model* [10] simulation `SIM_prop_planet_10`.

```
# Import the JEOD checkpoint/restart module.
import sys
import os
sys.path.append ('/'.join([os.getenv("JEOD_HOME"), "lib/jeod/python"]))
import jeod_checkpoint_restart

... (omitted code)

# Drop a checkpoint 60 days into the simulation.
jeod_checkpoint_restart.create_checkpoint (60*86400, 86400*(365*150+30))
```

This uses the JEOD Python library file `jeod_checkpoint_restart.py` to do most of the work. The function `create_checkpoint` in that module takes two arguments. The first is the simulation time at which the checkpoint is to be created. The second is the simulation time at which a simulation restarted from this checkpoint file is to end.

The Trick and JEOD checkpoint files are named according to the time at which the checkpoint is created. In the above example, the checkpoint files will be named `chkpnt_5184000.000000` and `jeod_chkpnt_5184000.000000`. The `SET_test/RUN_prop_restart` run of the same simulation loads the checkpoint files via the following:

```
# Import the JEOD checkpoint/restart module.
import sys
import os
sys.path.append ('/'.join([os.getenv("JEOD_HOME"), "lib/jeod/python"]))
```

```
import jeod_checkpoint_restart

... (omitted code)

# Restart the sim from the checkpoint data in SET_test/RUN_prop_checkpoint.
jeod_checkpoint_restart.restore_from_checkpoint (
  "SET_test/RUN_prop_checkpoint", "chkpnt_5184000.000000")
```

## 4.2 Instructions for Simulation Developers

### 4.2.1 The jeod_sys Simulation Object

All JEOD-based simulations must contain an instance of a `JeodSimulationInterface` object. This object must be the first JEOD object that is constructed and the last JEOD object that is destroyed during the course of execution of the simulation.

Use

```
#include "JEOD_S_modules/jeod_sys.sm"
```

where `JEOD_S_modules` is a symbolic link to `$JEOD_HOME/sims/shared/Trick10/S_modules`. To use a MessageHandler class other than the TrickMessageHandler, `#define MESSAGE_HANDLER_CLASS` to the desired class name.

### 4.2.2 Integration Jobs

The Dynamics Manager Model provides the ability to monolithically integrate all of the `DynBody` objects that comprise a simulation. Refer to the User Guide chapter of *Dynamics Manager Model* [2] for details.

The Simulation Interface Model provides a Trick-specific multiple integration group capability. One way to use this capability is to use the JEOD S_module `integ_loop.sm`. This defines the `JeodIntegLoopSimObject` simulation object class. You will need to `#include` this S_module in your S_define file:

```
#include "JEOD_S_modules/jeod_sys.sm"
```

Unlike the other Trick 13 JEOD S_modules, this S_module does not create an instance of simulation object class. Your S_define must declare these instances via the `JeodIntegLoopSimObject` non-default constructor. The following example is from the *Ephemerides Model* [10] simulation `SIM_prop_planet_10`.

```
JeodIntegLoopSimObject fast_integ_loop (
   FAST_DYNAMICS,
   integ_constructor.selected_constr, integ_constructor.group,
   jeod_time.time_manager, dynamics.dyn_manager, env.gravity_manager,
   &sun, &jupiter, &saturn, NULL);

JeodIntegLoopSimObject slow_integ_loop (
   SLOW_DYNAMICS,
   integ_constructor.selected_constr, integ_constructor.group,
   jeod_time.time_manager, dynamics.dyn_manager, env.gravity_manager,
   NULL);
```

The only public constructor for the `JeodIntegLoopSimObject` is the non-default constructor used in the above example. This constructor takes six required arguments plus an indeterminate number of optional arguments. The six required arguments are

`integ_cycle` The time interval between calls to the integration loop's integrate function.

`integ_cotr_in` A pointer to the integration constructor that is to be used to create integration artifacts for the integration group and the objects integrated by the integration group.

`integ_group_in` An integration group that is to be used as a prototype for creating the integration loop's integration group object.

`time_manager_in` The simulation-wide JEOD time manager.

`dyn_manager_in` The simulation-wide JEOD dynamics manager.

`grav_model_in` The simulation-wide JEOD gravity model.

Another option is to build a custom simulation object class. This class must contain an object of type `JeodDynbodyIntegrationLoop`. That object needs to be constructed using the non-default constructor for that class. The simulation object class must register `initialization`, `derivative`, and `integ_loop` jobs as the `JeodIntegLoopSimObject` class does. Use the `JeodIntegLoopSimObject` as a guide for these efforts.

## 4.3 Instructions for Model Developers

This section describes the use of the model from the perspectives of a user of the STL sequence container replacements, a model developer who wishes to write a class that derives from one of the provided checkpointable/restartable classes, and a simulation interface developer who is porting the model outside of Trick.

### 4.3.1 Using the Simulation Interface Model

External users can selected public interfaces of the global simulation interface object. Most of these are well-described in the Simulation Interface Model API.

External users who wish to use the JEOD checkpoint/restart capability can either create/read from their own sections within the JEOD checkpoint file or create/read from their own partitioned checkpoint file.. This sections that follow describes both of these options.

### 4.3.2 Creating and Restoring From a Section in the JEOD Checkpoint File

External users can write to the JEOD checkpoint file in a function registered with Trick as a checkpoint or post_checkpoint job. The steps involved in creating a section in the JEOD checkpoint file are:

- Create a `SectionedOutputStream` object by calling the static function `get_checkpoint_writer` in the class `JeodSimulationInterface`.
- Activate the `SectionedOutputStream` object. This activation will fail if another such output stream is currently active.
- Write to the `SectionedOutputStream` object using the standard C++ stream insertion operator.
- Deactivate the `SectionedOutputStream` object.

An abbreviated version of the above follows.

```
SectionedOutputStream writer (
   JeodSimulationInterface::get_checkpoint_writer (section_id));
if (! writer.activate()) {
   // Handle error
}
while (! done_writing) {
   writer << more_stuff_to_write();
}
writer.deactivate();
```

Restoring from that section is accomplished in a function registered with Trick as preload_checkpoint or restart job. The steps involved in restoring from a section in the JEOD checkpoint file are:

- Create a `SectionedInputStream` object by calling the static function `get_checkpoint_reader` in the class `JeodSimulationInterface`.
- Activate the `SectionedInputStream` object. This activation will fail if another such input stream is currently active.
- Read from the `SectionedInputStream` object using `std::getline` or the C++ stream insertion operator.
- Deactivate the `SectionedInputStream` object.

An abbreviated version of the above follows.

```
SectionedInputStream reader (
   JeodSimulationInterface::create_checkpoint_reader (section_id));
if (! reader.activate()) {
      // Handle error
   }
   while (std::getline (reader, line)) {
      process_line(line);
   }
   reader.deactivate();
```

Three restrictions apply to those who wish to write to or read from the JEOD checkpoint file:

- The substring "JEOD" must not appear anywhere in the name of the section. These names are reserved for use by JEOD.
- You must "play nice". You need to close your output and input section streams when you are done using them. Streams that are left open will prevent JEOD from writing or reading its checkpoint file sections.
- The checkpoint file is only available during the checkpoint and restart processes. To write to the JEOD checkpoint file you must do this in the context of a checkpoint job with a phase greater than 0 or a post_checkpoint job with a phase less 65535. To read from the JEOD checkpoint file you must do this in the context of a preload_checkpoint job with a phase greater than 0 or a restart job with a phase less 65535.

### 4.3.3 Creating and Restoring From a Custom Partitioned Checkpoint File

An alternative to the above is to create your own partitioned checkpoint file. This alternative removes the constraints on using the JEOD checkpoint file but adds the burden yet another checkpoint file that needs to be managed.

The class `CheckPointOutputManager` provides the ability to create a partitioned checkpoint file. Programmatic users of this class should:

- Create a `CheckPointOutputManager` object via the non-default constructor. The non-default constructor takes three arguments: the name of the file, the string that denotes the start of a section, and the string that denotes the end of a section. This opens a C++ output stream to the specified checkpoint file. That stream is private; it is accessed indirectly by `SectionedOutputStream` objects created by the `CheckPointOutputManager` object.

  ```
  checkpoint_writer = new CheckPointOutputManager (
     output_file_name, section_start, section_end);
  ```

- For each section to be created,
  - Create a `SectionedOutputStream` object by calling the `CheckPointOutputManager` object's `create_section_writer` member function.
  - Activate the `SectionedOutputStream` object. This activation will fail if some other `SectionedOutputStream` is currently active.
  - Write to the `SectionedOutputStream` object using the standard C++ stream insertion operator.

  - Deactivate the `SectionedOutputStream` object.

- An abbreviated version of the above follows.

```
SectionedOutputStream writer (
   checkpoint_writer->create_section_writer (section_id));
if (! writer.activate()) {
   // Handle error
}
while (! done_writing) {
   writer << more_stuff_to_write();
}
writer.deactivate();
```

- The output file is closed with the destruction of the `CheckPointOutputManager` object.

The class `CheckPointInputManager` provides the ability to read from a previously created partitioned checkpoint file. Programmatic users of this class should:

- Create a `CheckPointInputManager` object via the non-default constructor. The non-default constructor takes three arguments: the name of the file, the string that denotes the start of a section, and the string that denotes the end of a section. This opens a C++ input stream to the specified checkpoint file. That stream is private; it is accessed indirectly by `SectionedInputStream` objects created by the `CheckPointInputManager` object.

```
checkpoint_reader = new CheckPointInputManager (
   output_file_name, section_start, section_end);
```

- For each section to be read,
  - Create a `SectionedInputStream` object by calling the `CheckPointInputManager` object's `create_section_reader` member function.
  - Activate the `SectionedInputStream` object. This activation will fail if some other `SectionedInputStream` is currently active.
  - Read from the `SectionedInputStream` object using `std::getline` or the C++ stream insertion operator.
  - Deactivate the `SectionedInputStream` object.

- An abbreviated version of the above follows.

```
SectionedInputStream reader (
   checkpoint_writer->create_section_reader (section_id));
if (! reader.activate()) {
   // Handle error
}
while (std::getline (reader, line)) {
   process_line(line);
}
reader.deactivate();
```

- The input file is closed with the destruction of the `CheckPointInputManager` object.

### 4.3.4 Extending the Simulation Interface Model

This section describes how to extend the data types defined by the Simulation Interface Model.

The classes `JeodSimulationInterface`, `JeodMemoryInterface`, and `IntegratorInterface` declares several pure virtual functions. Usable derived class must provide implementations of these pure virtual functions. Dummy implementations can be provided for several of these functions. For example, address to name and name to address translation is only used during checkpoint and restart. There is no need to provide a full-blown implementation of this functionality if the target usage does not require checkpoint/restart.

A compliant instantiable class that derives from `JeodSimulationInterface` must contain one instance of a class that derives from `MessageHandler`, one instance of a class that derives from `JeodMemoryInterface`, and one instance of a class that derives from `JeodMemoryManager`. These objects must be constructed in the above order and destructed in the reverse order.

# Chapter 5

# Inspections, Tests, and Metrics

## 5.1 Inspections

This section describes the inspections of the Simulation Interface Model.

*Inspection SIMINT_1: Top-level Inspection*

This document structure, the code, and associated files have been inspected. With the exception of the cyclomatic complexity of the `JeodTrickMemoryInterface::primitive attributes` method, the Simulation Interface Model satisfies requirement SIMINT_1. A waiver has been granted for this one exception.

*Inspection SIMINT_2: Design Inspection*

Table 5.1 summarizes the key elements of the implementation of the Simulation Interface Model that satisfy requirements levied on the model. By inspection, the Simulation Interface Model satisfies requirements SIMINT_2 to SIMINT_11.

Table 5.1: Design Inspection

| Requirement | | Satisfaction |
|---|---|---|
| SIMINT_2 | Hidden Data Visibility | The macro `JEOD_MAKE_SIM_INTERFACES` provides the required simulation engine agnostic capability. The implementation of this macro is simulation engine specific. Trick-specific implementations make protected and private data visible to Trick. |

Continued on next page

Table 5.1: Design Inspection (continued from previous page)

| Requirement | | Satisfaction |
|---|---|---|
| SIMINT_3 | Allocated Data Visibility | The member functions `register_allocation` and `deregister_allocation` in the abstract class `JeodTrickMemoryInterface` specify the simulation engine agnostic capability. The implementations of these methods in the class `JeodTrickMemoryInterface` make allocated data visible to Trick. |
| SIMINT_4 | Simulation Engine Interface | The only Trick dependencies outside of the Simulation Interface Model are the noted exception in the Dynamics Manager Model. All other simulation engine interfaces are encapsulated within the Simulation Interface Model. |
| SIMINT_5 | Integration Interface | The abstract class `IntegratorInterface` provides the required simulation engine agnostic capabilities. Trick-specific classes that derive from this abstract class provide implementations of the required functionality in both the Trick 7 and Trick 10 environments. |
| SIMINT_6 | Job Cycle | The function `JeodSimulationInterface::get_job_cycle` is the public interface to this required functionality. This invokes the pure virtual protected member function `get_job_cycle_internal`. The class `BasicJeodTrickSimInterface` implements this function in the context of a Trick-based simulation. |
| SIMINT_7 | Trick Message Handler | The Simulation Interface Model provides the class `TrickMessageHandler`, which derives from the class `SuppressedCodeMessageHandler` and which implements the functionality required of a `MessageHandler` using Trick's messaging system. |
| SIMINT_8 | Checkpoint/Restart | The Simulation Interface Model provides a generic checkpoint/restart capability in the form of classes that create and read from a sectioned checkpoint file. The class `JeodTrick10MemoryInterface` uses these generic capabilities to provide the required ability to make JEOD-based simulations checkpointable and restartable in a Trick 10 environment. |

Continued on next page

Table 5.1: Design Inspection (continued from previous page)

| **Requirement** | | **Satisfaction** |
|---|---|---|
| SIMINT_9 | Address/Name Translation | The functions `get_name_at_address` and `get_address_at_name` in the classes `JeodSimulationInterface` (static) and `JeodMemoryInterface` (pure virtual) are the public interfaces to this required functionality. The class `JeodTrickMemoryInterface` provides dummy implementations while `JeodTrick10MemoryInterface` provides functional implementations of these functions. |
| SIMINT_10 | Multiple Integration Groups | The Simulation Interface Model implements this requirement as a set of JEOD-agnostic classes (which may eventually be migrated out of JEOD) and the JEOD-aware class `JeodDynbodyIntegrationLoop`. |
| SIMINT_11 | Extensibility | The Simulation Interface Model was carefully designed to have the public interfaces be in the form generic macros and abstract, simulation engine agnostic classes. The Trick independent demonstration and the test harness used in the JEOD unit tests illustrate to some extent that the model can be extended for use outside of the Trick environment. |

## 5.2 Tests

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

*Test SIMINT_1: Simulation Interface Simulation*

**Background** This test is an extremely simple simulation which creates a Trick sim_object containing an instance of a JeodTrickSimInterface to be tested. One other Trick sim_object creates a test object with a single scheduled job. The test object calls JeodTrickSimInterface::get_job_cycle() as a scheduled job, thus the output can be compared to the cycle time of the scheduled job.

The test object also includes private fields which are initialized by the Trick input processor and are logged by Trick.

**Test directory** `verif`
This is a standard verification directory containing the simulation directory `SIM_sim_interface` along with src and include directories for the test class code.

**Success criteria** The simulation includes initialization and logging of private fields which also reflect the output of JeodTrickSimInterface::get_job_cycle(). The logged data should be identical to the reference data in the SET_test_val directory.

**Test results** Passed.

**Applicable requirements** This test demonstrates the satisfaction of requirements SIMINT_2, SIMINT_4, and SIMINT_6.

*Test SIMINT_2: Container Simulation*

**Background** This simulation, located in the Container Model verification directory, exercises the checkpoint/restart capabilities of the model. For a complete description of this test, see the *Container Model* [1] for details.

**Test Directory** `models/utils/container/verif/SIM_container_T10`

**Test Results** Passed.

**Applicable Requirements** This test demonstrates the satisfaction of requirements SIMINT_3, SIMINT_4, SIMINT_8, and SIMINT_9.

*Test SIMINT_3: Memory Simulation*

**Background** This simulation, located in the Memory Management Model verification directory, tests the ability of the Memory Management Model to allocate and deallocate memory. This in turn tests the ability of this model to make those allocations visible to the simulation engine. For a complete description of this test, see the *Memory Management Model* [5] for details.

**Test Directory** `models/utils/memory/verif/SIM_memory_T10`

**Test Results** Passed.

**Applicable Requirements** This test demonstrates the satisfaction of requirements SIMINT_3, SIMINT_4, and SIMINT_8.

*Test SIMINT_4: Message Handler Simulation*

**Background** This simulation, located in the Message Handler Model verification directory, exercises the Trick-based implementation of the abstract class MessageHandler. For a complete description of this test, see the *Message Handler Model* [8] for details.

**Test Directory** `models/utils/message/verif/SIM_message_handler_verif_T10`

**Test Results** Passed.

**Applicable Requirements** This test demonstrates the satisfaction of requirements SIMINT_4 and SIMINT_7.

*Test SIMINT_5: Propagated Planet Simulation*

**Background** This simulation, located in the Ephemerides Model verification directory, tests the ability of the Ephemerides Model to propagate a planet as an alternative to using an ephemeris model. The Trick 10 version was constructed to serve as a test case of the multiple integration group capability provided by this model. For a complete description of this test, see the *Ephemerides Model* [10] for details.

**Test Directory** `models/environment/ephemerides/verif/SIM_prop_planet_T10`

**Test Results** Passed.

**Applicable Requirements** This test demonstrates the satisfaction of requirements SIMINT_3, SIMINT_4, SIMINT_5, SIMINT_8, SIMINT_9, and SIMINT_10.

## 5.3 Requirements Traceability

Table 5.2 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the model.

Table 5.2: Requirements Traceability

| **Requirement** | | **Traces to** |
|---|---|---|
| SIMINT_1 | Top-level requirement | Insp. SIMINT_1 Top-level Inspection |
| SIMINT_2 | Hidden Data Visibility | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_1 Simulation Interface Simulation |
| SIMINT_3 | Allocated Data Visibility | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_2 Container Simulation |
| | | Test SIMINT_3 Memory Simulation |
| | | Test SIMINT_5 Propagated Planet Simulation |
| SIMINT_4 | Simulation Engine Interface | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_1 Simulation Interface Simulation |
| | | Test SIMINT_2 Container Simulation |
| | | Test SIMINT_3 Memory Simulation |
| | | Test SIMINT_4 Message Handler Simulation |
| | | Test SIMINT_5 Propagated Planet Simulation |
| SIMINT_5 | Integration Interface | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_5 Propagated Planet Simulation |
| SIMINT_6 | Job Cycle | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_1 Simulation Interface Simulation |
| SIMINT_7 | Trick Message Handler | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_4 Message Handler Simulation |
| SIMINT_8 | Checkpoint/ Restart | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_2 Container Simulation |
| | | Test SIMINT_3 Memory Simulation |
| | | Test SIMINT_5 Propagated Planet Simulation |
| SIMINT_9 | Address/Name Translation | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_2 Container Simulation |
| | | Test SIMINT_5 Propagated Planet Simulation |
| SIMINT_10 | Multiple Integration Groups | Insp. SIMINT_2 Design Inspection |
| | | Test SIMINT_5 Propagated Planet Simulation |
| SIMINT_11 | Extensibility | Insp. SIMINT_2 Design Inspection |

## 5.4 Metrics

Table 5.3 presents coarse metrics on the source files that comprise the model.

Table 5.3: Coarse Metrics

| | **Number of Lines** | | | |
|---|---|---|---|---|
| **File Name** | **Blank** | **Comment** | **Code** | **Total** |
| **Total** | 0 | 0 | 0 | 0 |

Table 5.4 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.4: Cyclomatic Complexity

| **Method** | **File** | **Line** | **ECC** |
|---|---|---|---|
| jeod::SectionedInputBuffer::~SectionedInputBuffer () | include/checkpoint_input_manager.hh | 94 | 1 |
| jeod::SectionedInputBuffer::operator ! () | include/checkpoint_input_manager.hh | 100 | 1 |
| jeod::SectionedInputBuffer::deactivate (void) | include/checkpoint_input_manager.hh | 119 | 1 |
| jeod::SectionedInputStream::operator ! () | include/checkpoint_input_manager.hh | 331 | 3 |
| jeod::CheckPointInputManager::std::create_section_reader (const std::string & tag) | include/checkpoint_input_manager.hh | 428 | 1 |
| jeod::CheckPointInputManager::operator ! () | include/checkpoint_input_manager.hh | 441 | 2 |
| jeod::CheckPointInputManager::have_active_reader () | include/checkpoint_input_manager.hh | 450 | 1 |
| jeod::CheckPointInputManager::std::SectionInfo (std::size_t start, std::size_t end) | include/checkpoint_input_manager.hh | 483 | 1 |
| jeod::SectionedOutputBuffer::~SectionedOutputBuffer () | include/checkpoint_output_manager.hh | 91 | 1 |
| jeod::SectionedOutputBuffer::operator ! () | include/checkpoint_output_manager.hh | 97 | 1 |
| jeod::SectionedOutputBuffer::deactivate (void) | include/checkpoint_output_manager.hh | 117 | 2 |
| jeod::SectionedOutputStream::operator ! () | include/checkpoint_output_manager.hh | 190 | 3 |
| jeod::CheckPointOutputManager::std::create_section_writer (const std::string & tag) | include/checkpoint_output_manager.hh | 289 | 1 |
| jeod::CheckPointOutputManager::operator ! () | include/checkpoint_output_manager.hh | 296 | 2 |
| jeod::CheckPointOutputManager::have_active_writer () | include/checkpoint_output_manager.hh | 305 | 1 |

Continued on next page

Table 5.4: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::TrickJeodIntegrator::~TrickJeodIntegrator () | include/jeod_trick_integrator.hh | 93 | 1 |
| jeod::TrickJeodIntegrator::integrate () | include/jeod_trick_integrator.hh | 103 | 1 |
| jeod::TrickJeodIntegrator::initialize (int, double) | include/jeod_trick_integrator.hh | 108 | 1 |
| jeod::JeodTrickIntegrator::~JeodTrickIntegrator () | include/jeod_trick_integrator.hh | 137 | 1 |
| jeod::JeodTrickIntegrator::er7_utils::interpret_integration_type (int integ_technique) | include/jeod_trick_integrator.hh | 142 | 1 |
| jeod::JeodTrickIntegrator::virtual::get_integrator () | include/jeod_trick_integrator.hh | 153 | 1 |
| jeod::JeodTrickIntegrator::get_dt () | include/jeod_trick_integrator.hh | 162 | 1 |
| jeod::JeodTrickIntegrator::get_first_step_derivs_flag () | include/jeod_trick_integrator.hh | 171 | 1 |
| jeod::JeodTrickIntegrator::set_first_step_derivs_flag (bool value) | include/jeod_trick_integrator.hh | 181 | 1 |
| jeod::JeodTrickIntegrator::reset_first_step_derivs_flag () | include/jeod_trick_integrator.hh | 191 | 1 |
| jeod::JeodTrickIntegrator::restore_first_step_derivs_flag () | include/jeod_trick_integrator.hh | 203 | 1 |
| jeod::JeodTrickIntegrator::set_step_number (unsigned int stepno) | include/jeod_trick_integrator.hh | 213 | 1 |
| jeod::JeodTrickIntegrator::set_time (double sim_time) | include/jeod_trick_integrator.hh | 222 | 1 |
| jeod::JeodSimulationInterface::get_mode (void) | include/simulation_interface.hh | 247 | 1 |
| jeod::SectionedInputBuffer::SectionedInputBuffer () | src/checkpoint_input_manager.cc | 36 | 1 |
| jeod::SectionedInputBuffer::activate (std::ifstream & stream, std::size_t spos, std::size_t epos) | src/checkpoint_input_manager.cc | 59 | 1 |

Continued on next page

Table 5.4: Cyclomatic Complexity (continued)

| **Method** | **File** | **Line** | **ECC** |
|---|---|---|---|
| jeod::std::underflow (void) | src/checkpoint_input_manager.cc | 80 | 5 |
| jeod::SectionedInputStream::SectionedInputStream () | src/checkpoint_input_manager.cc | 124 | 1 |
| jeod::SectionedInputStream::SectionedInputStream ( CheckPointInputManager * mngr, std::ifstream & ifstream, std::size_t spos, std::size_t epos) | src/checkpoint_input_manager.cc | 145 | 1 |
| jeod::SectionedInputStream::SectionedInputStream (const SectionedInput Stream & source) | src/checkpoint_input_manager.cc | 172 | 4 |
| jeod::SectionedInputStream::~SectionedInputStream (void) | src/checkpoint_input_manager.cc | 201 | 2 |
| jeod::SectionedInputStream::is_activatable (void) | src/checkpoint_input_manager.cc | 214 | 5 |
| jeod::SectionedInputStream::activate (void) | src/checkpoint_input_manager.cc | 237 | 5 |
| jeod::SectionedInputStream::deactivate (void) | src/checkpoint_input_manager.cc | 283 | 2 |
| jeod::CheckPointInput Manager::CheckPointInput Manager (const std::string & fname, const std::string & start_marker, const std::string & end_marker) | src/checkpoint_input_manager.cc | 308 | 2 |
| jeod::CheckPointInput Manager::initialize (void) | src/checkpoint_input_manager.cc | 340 | 13 |
| jeod::CheckPointInput Manager::create_section_reader (bool trick, const std::string & tag) | src/checkpoint_input_manager.cc | 411 | 6 |
| jeod::CheckPointInput Manager::create_trick_section_reader (void) | src/checkpoint_input_manager.cc | 462 | 2 |

Continued on next page

Table 5.4: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::CheckPointInputManager::register_reader (SectionedInputStream * reader) | src/checkpoint_input_manager.cc | 479 | 2 |
| jeod::CheckPointInputManager::deregister_reader (const SectionedInputStream * reader) | src/checkpoint_input_manager.cc | 498 | 2 |
| jeod::SectionedOutputBuffer::SectionedOutputBuffer () | src/checkpoint_output_manager.cc | 36 | 1 |
| jeod::SectionedOutputBuffer::activate (std::ofstream & stream) | src/checkpoint_output_manager.cc | 54 | 1 |
| jeod::std::overflow (std::streambuf::int_type ch) | src/checkpoint_output_manager.cc | 69 | 5 |
| jeod::SectionedOutputStream::SectionedOutputStream () | src/checkpoint_output_manager.cc | 112 | 1 |
| jeod::SectionedOutputStream::SectionedOutputStream (CheckPointOutputManager * mngr, std::ofstream & ofstream, const std::string & start_marker, const std::string & end_marker, const std::string & section_name) | src/checkpoint_output_manager.cc | 134 | 1 |
| jeod::SectionedOutputStream::SectionedOutputStream (const SectionedOutputStream & source) | src/checkpoint_output_manager.cc | 164 | 4 |
| jeod::SectionedOutputStream::~SectionedOutputStream (void) | src/checkpoint_output_manager.cc | 194 | 1 |
| jeod::SectionedOutputStream::is_activatable (void) | src/checkpoint_output_manager.cc | 205 | 5 |
| jeod::SectionedOutputStream::activate (void) | src/checkpoint_output_manager.cc | 228 | 6 |

Continued on next page

Table 5.4: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::SectionedOutput Stream::deactivate (void) | src/checkpoint_output_ manager.cc | 282 | 3 |
| jeod::CheckPointOutput Manager::CheckPoint OutputManager (const std:: string & fname, const std:: string & start_marker, const std::string & end_marker) | src/checkpoint_output_ manager.cc | 312 | 2 |
| jeod::CheckPointOutput Manager::create_section_ writer (bool trick, const std::string & tag) | src/checkpoint_output_ manager.cc | 340 | 5 |
| jeod::CheckPointOutput Manager::create_trick_ section_writer (void) | src/checkpoint_output_ manager.cc | 380 | 2 |
| jeod::CheckPointOutput Manager::register_writer ( SectionedOutputStream * writer) | src/checkpoint_output_ manager.cc | 397 | 2 |
| jeod::CheckPointOutput Manager::deregister_writer (const SectionedOutput Stream * writer) | src/checkpoint_output_ manager.cc | 416 | 2 |
| jeod::JeodMemoryInterface:: JeodMemoryInterface (void) | src/memory_interface.cc | 38 | 1 |
| jeod::JeodMemoryInterface::~ JeodMemoryInterface (void) | src/memory_interface.cc | 48 | 1 |
| jeod::JeodMemoryInterface:: JeodMemoryInterface (const JeodMemory Interface & src JEOD_UNU SED) | src/memory_interface.cc | 58 | 1 |
| jeod::JeodMemoryInterface:: operator= (const Jeod MemoryInterface & src JE OD_UNUSED) | src/memory_interface.cc | 69 | 1 |

Continued on next page

Table 5.4: Cyclomatic Complexity (continued)

| **Method** | **File** | **Line** | **ECC** |
|---|---|---|---|
| jeod::JeodSimulationInterface Init::JeodSimulation InterfaceInit (void) | src/simulation_interface.cc | 48 | 1 |
| jeod::JeodSimulation Interface::JeodSimulation Interface (void) | src/simulation_interface.cc | 70 | 2 |
| jeod::JeodSimulation Interface::~JeodSimulation Interface (void) | src/simulation_interface.cc | 90 | 2 |
| jeod::JeodSimulation Interface::configure (const JeodSimulationInterfaceInit & config) | src/simulation_interface.cc | 102 | 1 |
| jeod::JeodSimulation Interface::create_integrator_interface (void) | src/simulation_interface.cc | 124 | 2 |
| jeod::JeodSimulation Interface::get_job_cycle (void) | src/simulation_interface.cc | 147 | 2 |
| jeod::JeodSimulation Interface::get_memory_interface (void) | src/simulation_interface.cc | 170 | 2 |
| jeod::std::get_name_at_address (const void * addr, const JeodMemoryType Descriptor * tdesc) | src/simulation_interface.cc | 190 | 2 |
| jeod::JeodSimulation Interface::get_address_at_name (const std::string & name) | src/simulation_interface.cc | 217 | 2 |
| jeod::JeodSimulation Interface::get_checkpoint_reader (const std::string & section_id) | src/simulation_interface.cc | 242 | 2 |
| jeod::JeodSimulation Interface::get_checkpoint_writer (const std::string & section_id) | src/simulation_interface.cc | 263 | 2 |

Continued on next page

Table 5.4: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::JeodSimulation Interface::set_mode (Mode new_mode) | src/simulation_interface.cc | 284 | 15 |

# Bibliography

[1] Hammen, D. Container Model. Technical Report JSC-61777-utils/container, NASA, Johnson Space Center, Houston, Texas, July 2023.

[2] Hammen, D. Dynamics Manager Model. Technical Report JSC-61777-dynamics/dyn_manager, NASA, Johnson Space Center, Houston, Texas, July 2023.

[3] Hammen, D. Dynamic Body Model. Technical Report JSC-61777-dynamics/dyn_body, NASA, Johnson Space Center, Houston, Texas, July 2023.

[4] Hammen, D. Integration Model. Technical Report JSC-61777-utils/integration, NASA, Johnson Space Center, Houston, Texas, July 2023.

[5] Hammen, D. Memory Management Model. Technical Report JSC-61777-utils/memory, NASA, Johnson Space Center, Houston, Texas, July 2023.

[6] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, July 2023.

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

[8] Shelton, R. Message Handler Model. Technical Report JSC-61777-utils/message, NASA, Johnson Space Center, Houston, Texas, July 2023.

[9] Shelton, R. Named Item Model. Technical Report JSC-61777-utils/named_item, NASA, Johnson Space Center, Houston, Texas, July 2023.

[10] Thompson, B. Ephemerides Model. Technical Report JSC-61777-environment/ephemerides, NASA, Johnson Space Center, Houston, Texas, July 2023.

[11] Turner, G. Time Model. Technical Report JSC-61777-environment/time, NASA, Johnson Space Center, Houston, Texas, July 2023.