JSC-61777-models/utils/container

JSC Engineering Orbital Dynamics Container Model

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

Package Release JEOD v5.1

Document Revision 1.1
July 2023

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

JSC-61777-models/utils/container

JSC Engineering Orbital Dynamics
Container Model

Document Revision 1.1
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 Container Model forms a component of the utilities suite of models within JEOD v5.1. It is located at models/utils/container.

The model has two overarching goals:

- Provide a generic framework for saving to and restoring from a checkpoint file those resources that are opaque to the typical simulation engine.
- Provide checkpointable and restorable replacements for selected Standard Template Library (STL) container class templates (list, vector, and set).

The Container Model implements these goals in the form of C++ classes and class templates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose and Objectives of the Container Model

The Container Model provides:

- A generic framework for defining checkpointable/restartable objects,
- Checkpointable/restartable replacements for the Standard Template Library (STL) containers used within JEOD (list, vector, and set), and
- Mechanisms to checkpoint and restart these STL containers.

1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [5]

The Container Model forms a component of the utilities suite of models within JEOD v5.1. It is located at models/utils/container.

1.3 Document History

Author	Date	Revision	Description
David Hammen	February 2012	1.1	Current version
David Hammen	June 2011	1.0	JEOD 2.2 beta release

1.4 Document Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [6].

The document comprises chapters organized as follows:

Chapter 1: Introduction -This introduction describes the objective and purpose of the Container Model.

Chapter 2: Product Requirements -The requirements chapter describes the requirements on the Container Model.

Chapter 3: Product Specification -The specification chapter describes the architecture and design of the Container Model.

Chapter 4: User Guide -The user guide chapter describes how to use the Container 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 Container Model.

Chapter 2

Product Requirements

Requirement Container_1: Top-level requirement

Requirement:
The Container Model shall meet the JEOD project requirements specified in the JEOD v5.1 top-level document.

Rationale:
This model shall, at a minimum, meet all external and internal requirements applied to the JEOD v5.1 release.

Verification:
Inspection

Requirement Container_2: Checkpoint/Restart

Requirement:
The Container Model shall provide a generic framework for checkpointing resources that would otherwise be opaque to simulation engines and for restoring those resources from a checkpoint file.

2.1 Extensibility. The basic checkpoint/restart capability shall provide a means for extension such that developers of models that contain opaque resources can have these resources checkpointed and restarted from a checkpoint.

2.2 Transparency. The implementation details of this extensibility framework shall be transparent to the models that perform the checkpointing of the checkpointable contents to a checkpoint file, and the restoration of resources from a previously recorded checkpoint file.

In other words, the implementation shall define a set of virtual functions that specify the checkpoint/restart interface. The checkpoint/restart framework calls the generic interfaces without concern of how the extensions implement the required behavior.

2.3 Undefined behavior. The effects of copying, swapping, or overwriting checkpoint information is undefined.

This means that extenders should not write a true swap() method and should be careful not to copy or overwrite checkpoint information in their copy constructors and assignment operators. Note that this is not a requirement on the model. It is a requirement on model extenders: Do not invoke undefined behavior.

Rationale:

The primary objective of the Container Model is to make all of JEOD checkpointable and restartable from a checkpoint. A standard framework enables the agents that perform the checkpointing and restoring to do so without knowledge of the contents to be checkpointed and restored. An extensible framework enables the developers of models with opaque content to specify the mechanisms for checkpointing and restoring such content.

Verification:

Inspection, Test

Requirement Container_3: STL Containers

Requirement:

The Container Model shall provide checkpointable and restorable replacements for the STL list, set, and vector class templates.

3.1 Full functionality. With the noted exceptions, each STL container replacement shall provide the full functionality of its STL counterpart as specified in the 2003 C++ Standard, ISO/IEC 14882:2003 [4].

Exceptions:

- Support for non-default Compare or Allocator classes is not required.
- The implementation shall provide a default constructor and a copy constructor for each STL container replacement. The implementation does not need to provide the full suite of constructors defined by the Standard for the STL containers.
- Where the standard itself is known to be buggy, the implementation should follow the recommended correction to the standard. This exception is defined as reported issues that have reached at least CD1 status with the C++ Standards Committee (ISO/IEC JTC1/SC22/WG21).
- The implementation shall provide the ability to swap the STL contents of a JEOD container with another JEOD or STL container of the same base type. The JEOD equivalent of the STL swap method shall not touch the internal checkpoint information and thus shall not be named swap. See requirement 2.3.

3.2 Transparency. Excluding the exceptions specified above, the JEOD equivalents of the STL containers shall use the function names and signatures as specified in the 2003 C++ Standard for the STL class templates.

Rationale:

This requirement exists because the STL containers were the number one culprit that formerly made JEOD unable to be fully checkpointed or restarted.

The functionality and transparency requirements mean that only the data types of those STL container data members need to be changed. There should be very little, if any, ripple effect resulting from the change in type.

Verification:

Inspection, Test

Chapter 3

Product Specification

3.1 Conceptual Design

The Container Model has two overarching goals:

- Provide a generic framework for saving to and restoring from a checkpoint file those resources that are opaque to the typical simulation engine.
- Provide checkpointable and restorable replacements for the STL container class templates.

This section provides insight into how the model achieves these goals.

3.1.1 Key Concepts

This section describes the key concepts that form the conceptual foundation of the Container Model.

3.1.1.1 Checkpoint/Restart

Checkpoint/restart addresses the issue of saving a description of a running application in a manner that the application can later be restarted from that saved description. Checkpoint/restart can be performed by the operating system or by the application itself, and the checkpoint file can be in text or binary form.

JEOD is primarily aimed at running in the Trick simulation environment, which provides a text-based, application-side checkpoint/restart capability. The basic Trick checkpoint/restart capability is, by design, incomplete. Trick provides the ability to register functions as checkpoint or restart jobs. A complete checkpoint/restart capability can be achieved by augmenting Trick's basic checkpoint/restart capabilities with the right combination of checkpoint and restart jobs.

This model does not provide the checkpoint/restart functions needed to make a Trick checkpoint complete. What this model does provide are data representations that simplify those checkpoint/restart functions in the Simulation Engine Interface Model model.

3.1.1.2 STL Containers

One of the key culprits that formerly made JEOD incapable of being checkpoint/restart complete was its extensive use of STL containers. Trick's data-driven checkpoint/restart mechanism does not jibe well with the complex implementation of STL containers in most compilers. The Trick project elected early on in the Trick 10 development process to exclude STL containers from its data-driven checkpoint/restart approach.

The first step in making JEOD's use of the STL containers checkpoint/restart-capable required providing functional equivalents of these STL containers. The model provides these function equivalents in the form of STL replacement class templates. These replacements do not re-implement the STL containers. They instead provide encapsulations of the containers. The interfaces to those replacement templates is by design a near-duplicate of the public interfaces to the STL containers.

3.1.1.3 Checkpointable/Restartable Objects

The above STL replacements by themselves do nothing to solve the checkpoint/restart problem. If anything, they exacerbate the problem by hiding the encapsulated STL object from the simulation engine.

The approach taken within the model is to provide an abstract framework for checkpoint and restart. Mixin class templates provided by the model augment the basic STL replacement class templates with this checkpoint/restart functionality. An added benefit of this scheme is that the same framework can also be used to address other checkpoint/restart issues such as checkpointing and restoring file descriptors.

3.1.2 Model Architecture

The Container Model is implemented in the form of C++ classes and class templates, described below.

`JeodCheckpointable` The base class for checkpointing and restarting data that are opaque to Trick, and presumably other simulation engines.

`SimpleCheckpointable` A simplified version of `JeodCheckpointable`.

`JeodSTLContainer` A non-checkpointable replacement for STL containers in general.

`JeodAssociativeContainer` A non-checkpointable replacement for STL associative containers.

`JeodSequenceContainer` A non-checkpointable replacement for STL sequence containers.

`JeodList` A non-checkpointable replacement for STL lists.

`JeodSet` A non-checkpointable replacement for STL sets.

`JeodVector` A non-checkpointable replacement for STL vectors.

`JeodContainer` A mixin class that adds checkpoint/restart capabilities to a non-checkpointable JEOD STL container replacement.

`JeodObjectContainer` A `JeodContainer`-derived class that addresses containers that contain object.

`JeodPointerContainer` A `JeodContainer`-derived class that addresses containers that contain pointers.

`JeodPrimitiveContainer` A `JeodContainer`-derived class that addresses containers that contain simple data.

`JeodPrimitiveSerializer` Class used by `JeodPrimitiveContainer` to serialize/deserialize a simple piece of data.

`JeodObjectList` A `Checkpointable` replacement for a list of objects.

`JeodObjectSet` A `Checkpointable` replacement for a set of objects.

`JeodObjectVector` A `Checkpointable` replacement for a vector of objects.

`JeodPointerList` A `Checkpointable` replacement for a list of pointers.

`JeodPointerSet` A `Checkpointable` replacement for a set of pointers.

`JeodPointerVector` A `Checkpointable` replacement for a vector of pointers.

`JeodPrimitiveList` A `Checkpointable` replacement for a list of primitives.

`JeodPrimitiveSet` A `Checkpointable` replacement for a set of primitives.

`JeodPrimitiveVector` A `Checkpointable` replacement for a vector of primitives.

3.2 Key Algorithms

The majority of the functions defined in the Container Model are simple pass-through functions to the encapsulated STL object. This section describes the key algorithms within the model in the broader context of the serialization and deserialization processes that form the basis of text-based checkpoint and restart.

3.2.1 Pointer Containers

From the perspective of this model, containers of pointers are by far the easiest to checkpoint and restart. The Simulation Engine Interface Model provides methods to convert a pointer to a named representation of the pointed-to object and to convert a named representation to a `void*` pointer. These methods provide a natural serialization/deserialization mechanism for containers of pointers.

3.2.2 Object Containers

Containers of objects are checkpointed by making a C-style array that is a copy of the contained objects. The simulation engine is responsible for checkpointing the contents of this array. The simulation engine is similarly responsible for restoring the array on restart. With this copy, the container itself is checkpointed and restarted by using the same serialization/deserialization mechanism used for pointer containers.

3.2.3 Primitive Containers

Containers of primitives are checkpointed and restarted by using the stream insertion and extraction operators for serialization and deserialization. Special care is needed for strings, which can contain special characters, and floating point numbers, which have special values to indicate the number is not-a-number (NaN) or infinity. During serialization, special characters in strings are backslash-escaped and special numbers are serialized as special-purpose strings. The deserialization restores the special characters and special numbers.

3.3 Interactions

3.3.1 JEOD Models Used by the Container Model

The Container Model uses the following JEOD models:

- *Memory Management Model* [2]. The class template `JeodContainer` contains a pointer to a Memory Management Model descriptor of the type of data in the container. The class template `JeodObjectContainer` uses the Memory Management Model to allocate and destroy an array of objects whose contents are checkpointed and restored by the simulation engine.

- *Simulation Engine Interface Model* [7]. All Container Model header files `#include` the Simulation Engine Interface Model header file jeod_class.hh. The class templates `JeodObjectContainer` and `JeodPointerContainer` use the Simulation Engine Interface Model to translate addresses to and from address specification strings.

3.3.2 Use of the Container Model in JEOD

The following JEOD models use the Container Model:

- *Memory Management Model* [2]. The Memory Management Model provides the means by which JeodCheckpointable objects are registered with the Simulation Engine Interface Model for subsequent checkpoint/restart operations.

- *Simulation Engine Interface Model* [7]. The Simulation Engine Interface Model provides the mechanisms that checkpoint and restore JeodCheckpointable objects.

- *Ephemerides Model* [8] and *Dynamic Body Model* [3]. The Ephemerides Model and Dynamic Body Model define subclasses of the SimpleCheckpointable class to restore opaque content defined by those models.
- Several models. Many JEOD models use the STL container replacements provided by the Container Model.

3.3.2.1 Simulation Engine Interface Model

3.3.2.1.1 Checkpoint. A JeodCheckpointable object is checkpointed by recording a series of triples, object identifier, action, and value, that collectively provide the information needed to later restore the contents of the object.

The following actions must be taken to checkpoint a JeodCheckpointable object:

1. If the string returned by object.get_init_name() is not empty, the checkpoint agent must record a triple comprising the object identifier, the string returned by get_init_name(), and the string returned by get_init_value().
2. Repeatedly record triples comprising the object identifier, the string returned by get_item_name(), and the string returned by get_item_value(). This should be done as the body of a loop that iterates over the object's checkpointable content.
3. If the string returned by object.get_final_name() is not empty, the checkpoint agent must record a triple comprising the object identifier, the string returned by get_final_name(), and the string returned by get_final_value().

```
const std::string & init_action = object.get_init_name();
if (! init_action.empty()) {
   record (identifier, init_action, object.get_init_value());
}

for (object.start_checkpoint();
     !object.is_checkpoint_finished();
     object.advance_checkpoint()) {
   record (identifier, object.get_item_name(), object.get_item_value());
}

const std::string & final_action = object.get_final_name();
if (! final_action.empty()) {
   record (identifier, final_action, object.get_init_value());
}
```

3.3.2.1.2 Restart. The JeodCheckpointable objects are restored to their checkpoint state by transforming by parsing the checkpointed contents into a set of identifier/action/value triples. For each triple, the identifier is mapped to a pointer to a checkpointable object and the object's `perform_action` method is called with the action and value supplied as arguments.

3.4 Detailed Design

The classes and methods of the Container Model are described in detail in the *Container Model API* [1]. This section describes architecture details that are not present in the API document.

3.4.1 Class Hierarchy

3.4.1.1 Overview

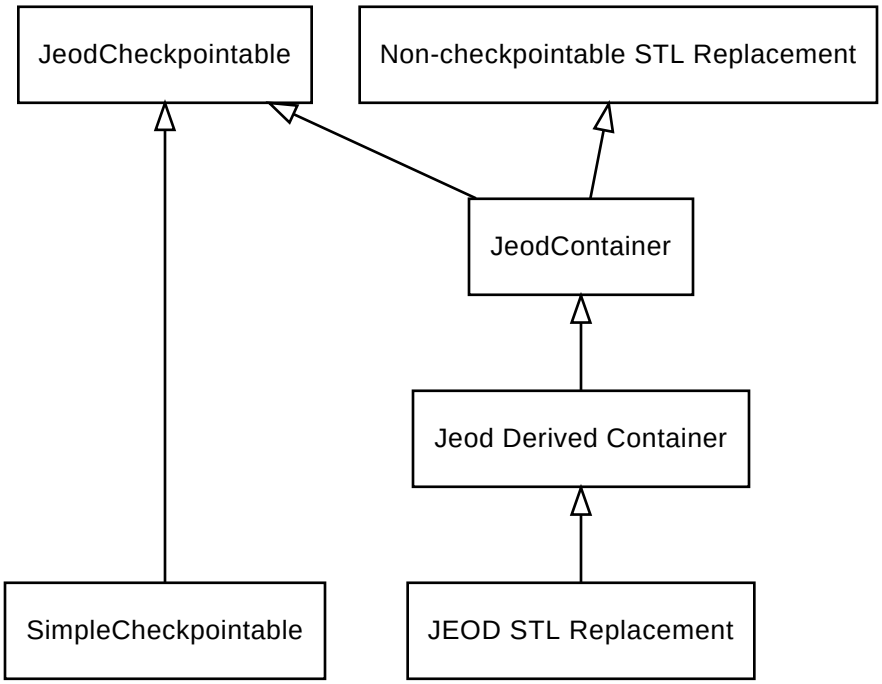

Figure 3.1: Class Hierarchy (Summary)

Figure 3.1 portrays a stylized depiction of the class hierarchy of the Container Model classes. The left side of the diagram portrays the classes SimpleCheckpointable and JeodCheckpointable. JeodCheckpointable is the base class that specifies in an abstract sense the interfaces needed to make an object checkpointable. The SimpleCheckpointable implements several of these behaviors but adds one new pure virtual interface that a derived class must implement.

On the right side of the diagram, the "Non-checkpointable STL Replacement" represents one of the class templates (three provided with this release) that encapsulate STL container objects. The class template JeodContainer augments a non-checkpointable STL replacement with a generic set of checkpoint/restart capabilities. The JeodContainer does not specify how to translate content to and from the textual representations used in a JEOD checkpoint file. This is the responsibility of one of the three "Jeod Derived Container" templates. Finally, "JEOD STL Replacement" represents one of the class templates designed as checkpointable replacements for the STL containers. There are nine such class templates provided with this release. The remainder of this section expands on the right side of figure 3.1.

3.4.1.2 Non-Checkpointable STL Replacements

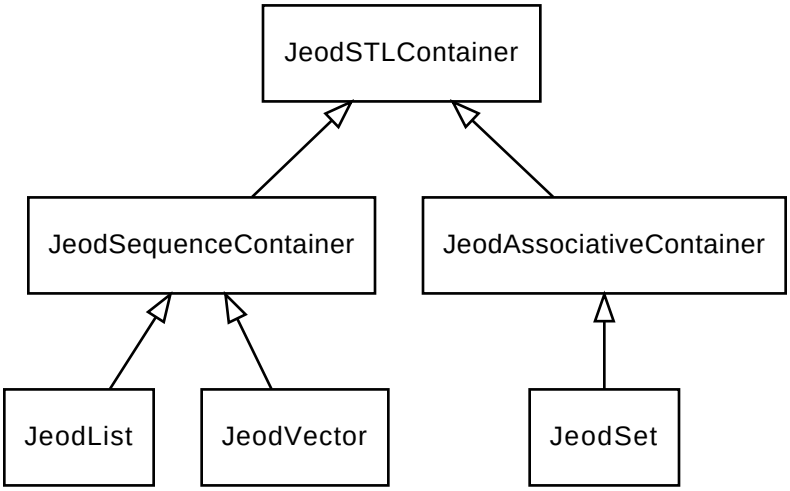

Figure 3.2: Non-Checkpointable STL Replacement Classes

Figure 3.2 expands upon the "Non-checkpointable STL Replacement" node in figure 3.1. That node in figure 3.1 represents one of the three nodes at the bottom of figure 3.2.

At the top of the diagram, JeodSTLContainer is the base class template for the non-checkpointable STL replacements. This class encapsulates an STL container object of the appropriate type and provides types and functions that are common to all STL containers. JeodSequenceContainer augments the JeodSTLContainer with methods that are not common to all STL containers but are common to all STL sequence containers (std::deque, std::list, and std::vector). JeodAssociativeContainer similarly augments the JeodSTLContainer with respect to STL associative containers (std::map, std::multimap, std::set, and std::multiset).

At the bottom of the diagram, JeodList, JeodVector, and JeodSet are functionally near-equivalents to std::list, std::vector, and std::set. These class templates add functionality unique to std::list, std::vector, and std::set, as appropriate. These templates invoke the parent class template in a way that specifies which STL container template is being replaced.

3.4.1.3 Checkpointable STL Replacements

Figure 3.3 expands upon the "JEOD STL Replacement" node in figure 3.1. That node in figure 3.1 represents one of the three nodes at the bottom of figure 3.3, each of which in turn represents three specific class template types. Nine end-user class templates are provided with this release, representing the set product {Object, Pointer, Primitive} × {List,Vector, Set}.

The class template JeodContainer is the central nexus in figure 3.3. JeodContainer inherits from JeodCheckpointable and from a "Non-checkpointable STL Replacement", one of JeodList, JeodVector, or JeodSet. Exactly which one of the three is specified as a template argument to JeodContainer. JeodContainer is a mixin class template. It adds checkpoint/restart capabilities to the functionality of a non-checkpointable STL replacement container.

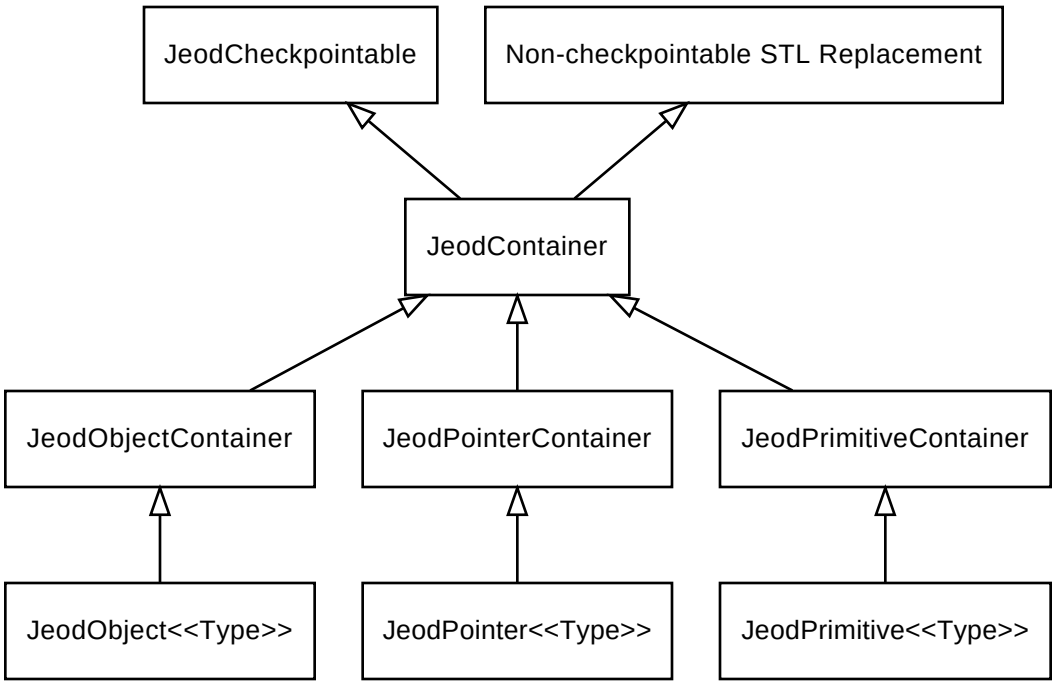

Figure 3.3: Checkpointable STL Replacement Classes (Generic)

JeodContainer implements all but one of the pure virtual methods declared in JeodCheckpointable and declares one new pure virtual method. These two unimplemented virtual methods address the problems of translating an element of a list, vector, or set to a string on checkpoint and restoring such an element from a string on restart. The type of data being stored (but not the type of the container) dictates how these translations to and from a string are performed. Each of the three template classes JeodObjectContainer, JeodPointerContainer, and JeodPrimitiveContainer provides implementations of these remaining virtual methods.

A user could use JeodObjectContainer et al. to specify the type of a checkpointable STL container replacement. Doing so would be very verbose and easy to get wrong. As a convenience, JEOD provides nine end-user templates that represent the set product {Object, Pointer, Primitive} $\times$ {List,Vector, Set}. Using these end-user template typedefs is the recommended technique for specifying the type of a JEOD STL container replacement object. These are collectively displayed as the three types at the bottom of figure 3.3. Figure 3.4 depicts the inheritance hierarchy for two of these nine class templates.

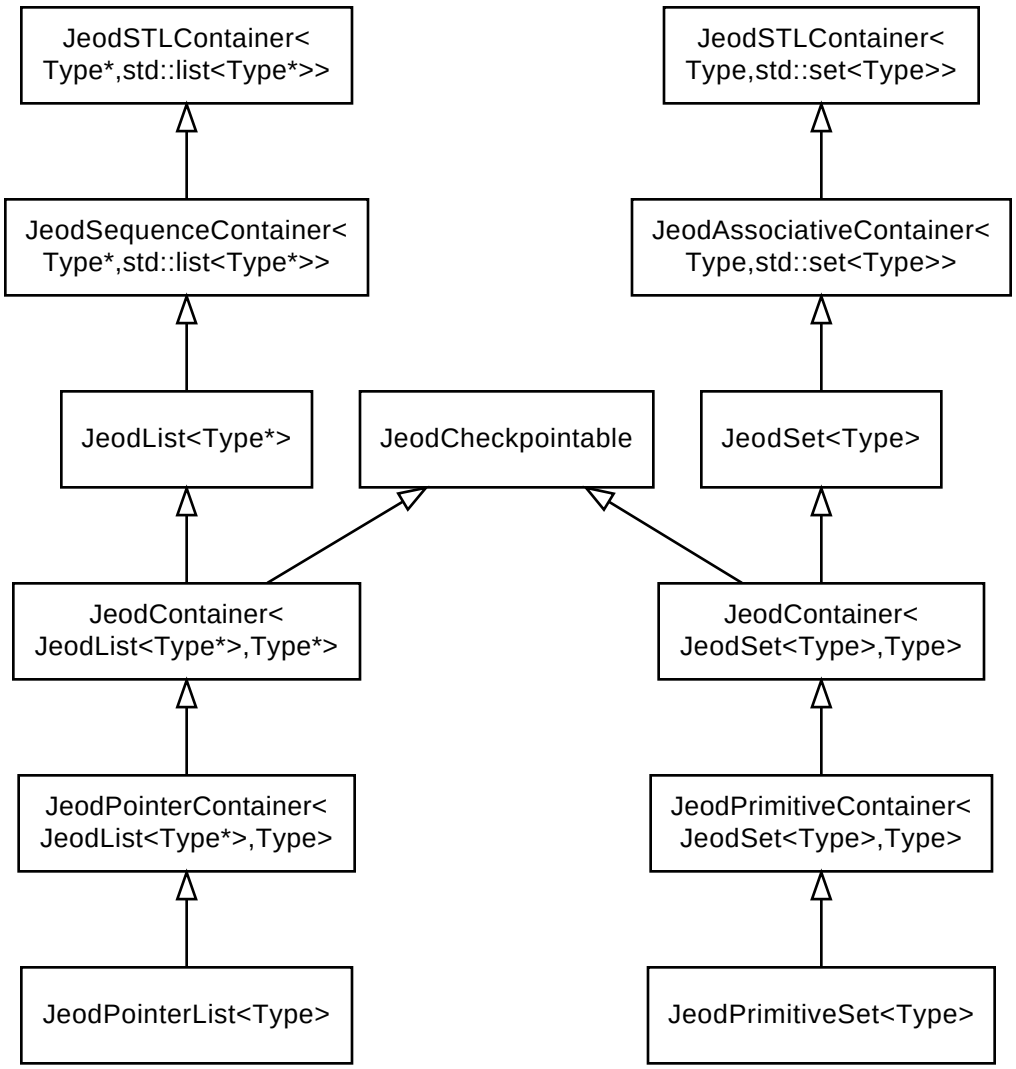

Figure 3.4: Checkpointable STL Replacement Classes (Examples)

3.4.2 Class JeodCheckpointable

JeodCheckpointable is the base class for all JEOD checkpointable objects. As currently implemented, the class contains no member data. The class defines a number of virtual methods, some pure virtual, others with an empty default implementation, organized below into five groups. These groups are:

- Initialization Method (one method).
- Checkpoint Control Methods (three methods).
- Checkpoint Content Methods (four methods).
- Restart Method (one method).
- Pre- and Post- Checkpoint and Restart Methods (four methods).

3.4.2.1 Initialization Methods

There is one method in this group, initialize_checkpointable.

3.4.2.1.1 initialize_checkpointable. All JeodCheckpointable objects must be registered to make the objects checkpointable and restartable. The registration machinery calls a JeodCheckpointable object's initialize_checkpointable method to signal that the object has been registered. The default implementation is to do nothing.

3.4.2.2 Checkpoint Control Methods

Three methods, start_checkpoint, is_checkpoint_finished, and advance_checkpoint, control the checkpoint generation process. The simulation engine interface dumps the contents of a JeodCheckpointable object via the following loop:

```
for (checkpointable->start_checkpoint();
     !checkpointable->is_checkpoint_finished();
     checkpointable->advance_checkpoint()) {
   // Record contents at the current checkpoint location, code elided.
}
```

Each of the methods in this group is pure virtual.

3.4.2.2.1 start_checkpoint. The simulation engine calls the start_checkpoint method to signal the checkpointable object to prepare for a checkpoint. Some examples:

- A state machine-based implementation will set some internal checkpoint state machine to its initial state.

- An iterator-based implementation will set some internal checkpoint iterator to the first item to be checkpointed.
- Other implementations such as the SimpleCheckpointable class will do nothing at all.

3.4.2.2.2 is_checkpoint_finished. The simulation engine calls the is_checkpoint_finished method to determine whether the checkpoint loop should be terminated. This method should return true when the checkpoint process is complete, false up until that point. Some examples:

- A state machine-based implementation will return true when the internal checkpoint state machine reaches its final state.
- An iterator-based implementation will return true when the internal checkpoint iterator reaches the end().

The checkpoint loop can be bypassed in its entirety by making the is_checkpoint_finished always return true. This is the approach taken by the SimpleCheckpointable class. The advance_checkpoint, get_item_name, and get_item_value methods will never be called in derived classes that employ this approach. Although never called, such classes will still need to provide implementations of these three methods because they are pure virtual.

3.4.2.2.3 advance_checkpoint. The simulation engine calls the advamce_checkpoint method to signal the checkpointable object to advance itself to the next item to be checkpointed. Some examples:

- A state machine-based implementation will advance the internal checkpoint state machine to the next state.
- An iterator-based implementation will increment the internal checkpoint iterator.

3.4.2.3 Checkpoint Content Methods

JeodCheckpointable declares four methods that the checkpoint machinery calls to dump the contents of a checkpointable object. Each of these methods must return a string.

3.4.2.3.1 get_init_name and get_init_value. The checkpoint machinery writes a checkpoint file entry for the checkpointable object that contains the values returned by these two methods, but only if the string returned by get_init_name is not empty. This occurs prior to the checkpoint loop described above. Some examples:

- get_init_name returns "clear" for the JEOD STL replacement classes. The corresponding restart action is to clear the contents of the object so that the follow-on actions will restore the contents of the STL object.
- get_init_name returns "restore" for the SimpleCheckpointable class. The corresponding restart action is to call the checkpointable object's simple_restore method (which must be implemented by the class that derives from SimpleCheckpointable.)

3.4.2.3.2 get_item_name and get_item_value. The checkpoint machinery writes a checkpoint file entry for the checkpointable object that contains the values returned by these two methods. These methods are called from within the checkpoint loop described above.

3.4.2.3.3 Constraints. The following constraints apply to the implementations of these get_ methods:

- get_init_name must return either the empty string or an alphanumeric string.
- get_item_name must return a non-empty alphanumeric string.
- The strings returned by get_init_value and get_item_value must not contain characters that force a newline or carriage return.

Violating these constraints results in undefined behavior.

Exception: The constraint on get_item_name does not apply if that method is guaranteed to never be called, *i.e.* the is_checkpoint_finished method always returns true. The get_item_name method must still be defined to make the class in question viable, but the implementation can be to return the empty string.

3.4.2.4 Restart Methods

There is one method in this group, perform_restore_action.

3.4.2.4.1 perform_restore_action. The restart machinery calls this method for every entry in the checkpoint file that pertains to the JeodCheckpointable object. This method takes two arguments, both of them const references to STL strings. The first argument, action_name, is a string returned by the get_init_name or get_item_name method. The second argument, action_value, is the corresponding value. A typical implementation will be in the form of an if/else if/else set of statements keyed on the action_name.

3.4.2.5 Pre and Post Checkpoint and Restart Methods

JeodCheckpointable defines four methods that are called before and after the checkpoint or restart proper. The default implementation for each does nothing.

3.4.2.5.1 pre_checkpoint. This method is called before any resources are checkpointed. Data allocated by JEOD_ALLOC in a pre_checkpoint method will be checkpointed. The JeodObjectContainer uses this capability to create a checkpointable copy of the STL container. JeodCheckpointable extenders may wish to emulate this behavior.

3.4.2.5.2 post_checkpoint. This method is called after checkpointing is complete. The array allocated by the JeodObjectContainer pre_checkpoint method is used only for checkpoint and restart. The JeodObjectContainer post_checkpoint method frees this array.

3.4.2.5.3 pre_restart. This method is called before any resources are restored. One envisioned use of this method is in a class that needs to free resources prior to restoring them to avoid leaks. Note: Allocating resources in this method is not recommended as doing so most likely will be a leak.

3.4.2.5.4 post_restart. This method is called after restart proper. One envisioned use of this method is in a class that needs to restore a resource that crosses class boundaries.

3.4.3 Class SimpleCheckpointable

The SimpleCheckpointable class provides a simple checkpoint/restart interface by which an object can complete the restart process. JEOD uses this class to restore the DE4xx ephemeris file stream and to restore the function pointers in a JEOD integrator. The class derives from JeodCheckpointable and implements all of the pure virtual methods declared in JeodCheckpointable.

The class SimpleCheckpointable declares one pure virtual interface, simple_restore. Classes that derive from SimpleCheckpointable must provide an implementation of this method to make the derived class complete. Derived classes should not override the methods declared in JeodCheckpointable and implemented in JeodCheckpointable. Derived classes can override the pre_ and post_checkpoint and the pre_ and post_restart methods.

3.4.4 Non-Checkpointable STL Replacement Class Templates

JEOD provides non-checkpointable STL replacement class templates for std::list (JeodList), std::set (JeodSet), and std::vector (JeodVector). These class templates, per self-imposed requirements and by design, are near-equivalents of the targeted STL class templates. The places where the JEOD class templates differ from their STL counterparts as specified in the 2003 C++ Standard (ISO/IEC 14882:2003 [4]) are:

- The destructors for the JEOD class templates, unlike their STL counterparts, are virtual. That STL containers have non-virtual destructors means a class cannot safely derive from an STL container. This is the key factor that drove the development of this part of the model.
- The constructors for the JEOD class templates, unlike their STL counterparts, do not take optional arguments. Users of the JEOD container class templates cannot provide an alternate Allocator class or an alternate Compare object.
- The standard specifies a number of constructors for the STL containers. The JEOD implementation provides only two of these: Default and copy.
- The constructors are protected rather than public. The non-checkpointable STL replacement class templates are not designed for use by the end-user.
- The swap function is protected rather than public. The checkpointable STL replacement class templates later exports this functionality in the form of the swap_contents function.

This function swaps the STL content of two objects but leaves the content related to checkpoint/restart untouched. (This in turn means swap_contents cannot be called "swap" per the standard committee's concept of being "swappable".)

- JeodSet implements the non-const iterator as a const_iterator and implements find, lower_bound, upper_bound, and equal_range as specified in table 69 of the 2003 C++ Standard rather than as specified in section 23.3.3 of the standard. (These implementations are consistent with the GNU implementations and with the recommendations of the C++ Standards Committee.)

To reduce the amount of replicated code, the JEOD implementations of these end-user class templates inherit from JeodSequenceContainer (JeodList and JeodVector) or JeodAssociativeContainer (JeodSet), both of which in turn inherit from JeodSTLContainer. The JeodSTLContainer class template defines items common to all STL containers. The class templates JeodSequenceContainer and JeodAssociativeContainer add items common to all sequence containers and associative containers. The end-user classes specify non-common content only.

In addition to defining items common to all STL containers, a JeodSTLContainer also contains a data member that is an STL container of the type being replaced. One way to look at the JEOD STL container replacement class templates is that they aren't truly STL container replacements. They are instead STL container encapsulators that transparently provide the functionality of the the encapsulated objects.

3.4.5 Checkpointable STL Replacement Class Templates

The ultimate goal of the STL container replacement aspect of this model is to make the non-checkpointable class templates described in section 3.4.4 checkpointable. The starting point for this effort is the class template JeodContainer. JeodContainer inherits from JeodCheckpointable and from one of the non-checkpointable STL container replacement class templates, specified as one of the template arguments to JeodContainer.

JeodContainer makes a non-checkpointable container object checkpointable without needing to know specifics of the type container that is being encapsulated. It accomplishes this by using three features common to all STL containers, including those not yet encapsulated by this model:

- Iterators that can be used to walk over the contents of the container object from beginning to end,
- A clear() method that empties the container object, and
- An insert(iterator, value) method that can be used to add some value to the end of the container object.

By taking advantage of these common features, the JeodContainer by itself comes very close to adding checkpoint/restart capabilities to a container. The class defines all but one of the functions declared as pure virtual in JeodCheckpointable.

What JeodContainer cannot do is translate the value of a container element to a string or recreate the element value given the string representation. JeodContainer does not implement the get_

item_value method declared as pure virtual in JeodCheckpointable and declares perform_insert_action as a new pure virtual method. The details of how containers of objects, pointers, and primitives are checkpointed and restored are handled by the three class templates JeodObjectContainer, JeodPointerContainer, and JeodPrimitiveContainer, each of which derives from JeodContainer and each of which implements those two remaining pure virtual methods in a different manner.

The class template JeodObjectContainer targets containers of objects. It achieves checkpoint/restart by creating a C-style array copy of the container. The array and its contents are checkpointed and restored using the simulation engine's native checkpoint/restart capabilities. The container is restored by copying the contents of the restored array back into the container.

The class template JeodObjectContainer targets containers of pointers. It's get_item_value and perform_insert_action methods use the Simulation Engine Interface Model to translate an address to and from a string representation.

The class template JeodPrimitiveContainer targets containers of items that can be translated to and from a string using the insertion and extraction operators. The class uses the template class JeodPrimitiveSerializer to assist in this serialization and deserialization.

A user could use JeodObjectContainer et al. to specify the type of a checkpointable STL container replacement. As mentioned above, this is verbose and error prone, and thus is not the recommended usage. JEOD provides nine end-user templates that represent the set product {Object, Pointer, Primitive} $\times$ {List,Vector, Set}.

3.5 Inventory

All Container Model files are located in `${JEOD_HOME}/models/utils/container`. 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

The typical JEOD model User Guide looks at the model at hand from the perspectives of a simulation user, a simulation developer, and a model developer. The Container Model is not the typical JEOD model User Guide. The model is designed to be transparent to the typical simulation user and simulation developer.

4.1 Instructions for Simulation Users

The container model is not intended for use by simulation users.

4.2 Instructions for Simulation Developers

The container model is not intended for use by simulation developers.

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 Model Header Files

The Container Model contains 23 header files, categorized into four major groups:

- Headers that define incomplete classes that form the basis for the checkpoint and restart. External users are free to create their own extensions of these incomplete classes.
- Headers that define non-checkpointable replacements for the STL sequence containers. These are incorporated into checkpointable replacements in other headers.

- Headers that define usable, but rather hard to name, checkpointable and restartable replacements for the STL containers. For example, a list that contains pointers to objects of type Foo is a JeodPointerContainer<JeodList<Foo*>, Foo>.
- Headers that simplify the process of defining checkpointable and restartable replacements for the STL containers. For example, to change a data member declared as std::list<Foo*> foo_ptr_list to its JEOD equivalent, change the type to JeodPointerList<Foo>::type and #include "utils/container/include/pointer_list.h" instead of <list>.

Only the first and last group are discussed in this User Guide. Table 4.1 describes these header files.

Table 4.1: Model Header Files

Header	**Defines**	**Description**
checkpointable.hh	JeodCheckpointable	Base class for something that can be checkpointed and restored from a checkpoint.
simple_ checkpointable.hh	SimpleCheckpointable	A simplified version of JeodCheckpointable. Extenders just need to define simple_ restore().
object_list.hh	JeodObjectList <ElemType>::type	Use in lieu of std::list<ElemType> for structured ElemType data.
object_set.hh	JeodObjectSet <ElemType>::type	Use in lieu of std::set<ElemType> for structured ElemType data.
object_vector.hh	JeodObjectVector <ElemType>::type	Use in lieu of std::vector<ElemType> for structured ElemType data.
pointer_list.hh	JeodPointerList <ElemType>::type	Use in lieu of std::list<ElemType*> (list of pointers).
pointer_set.hh	JeodPointerSet <ElemType>::type	Use in lieu of std::set<ElemType*> (set of pointers).
pointer_vector.hh	JeodPointerVector <ElemType>::type	Use in lieu of std::vector<ElemType*> (vector of pointer).
primitive_list.hh	JeodPrimitiveList <ElemType>::type	Use in lieu of std::list<ElemType> for primitive ElemType data.
primitive_set.hh	JeodPrimitiveSet <ElemType>::type	Use in lieu of std::set<ElemType> for primitive ElemType data.
primitive_vector.hh	JeodPrimitiveVector <ElemType>::type	Use in lieu of std::vector<ElemType> for primitive ElemType data.

4.3.2 Using the Container Model

This section describes how to use the data types defined by the Container Model in some other model.

4.3.2.1 STL Sequence Containers

Classes that contain one or more data members that are STL sequence containers are not checkpointable or restartable as-is. The approach taken by JEOD is to replace the STL types with their JEOD equivalents and to register the objects with JEOD. The code changes that need to be made to accomplish this are:

- In the header file for the class, replace the STL #includes and the STL sequence types with the JEOD equivalents. See the last set of entries in table 4.1 for the appropriate header files and data types. The following example illustrates a class that contains a list of pointers and a vector of doubles.

```
#include "utils/container/include/pointer_list.hh"
#include "utils/container/include/primitive_vector.hh"
class Foo;

class Bar {
public:
   Bar();
protected:
   JeodPointerList<Foo>::type foo_list;
   JeodPrimitiveVector<double>::type double_vector;
};
```

- In the constructor(s) for the class, register with the JEOD memory manager the class itself, the class in the list or vector (you don't need to register primitives), and the containers. The constructor for the class defined above is illustrated below.

```
#include "utils/memory/include/jeod_alloc.hh"
#include "../include/foo.hh"
#include "../include/bar.hh"

Bar::Bar ()
{
   JEOD_REGISTER_CLASS (Bar);
   JEOD_REGISTER_CLASS (Foo);
   JEOD_REGISTER_CHECKPOINTABLE (this, foo_list);
   JEOD_REGISTER_CHECKPOINTABLE (this, double_vector);
}
```

- In the destructor for the class, deregister the containers with the JEOD memory manager.

```
Bar::~Bar ()
{
   JEOD_DEREGISTER_CHECKPOINTABLE (this, foo_list);
   JEOD_DEREGISTER_CHECKPOINTABLE (this, double_vector);
}
```

4.3.3 Extending the Container Model

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

4.3.3.1 Extending the JeodCheckpointable Class

There are many things beyond STL containers that a simulation engine may not be able to checkpoint or restart without some additional help. One way to provide that additional help is to build a class that derives from JeodCheckpointable.

The class JeodCheckpointable declares several virtual methods with empty implementations and several pure virtual methods with no implementation at all. A usable extension to JeodCheckpointable must at a minimum provide implementations for these pure virtual methods. Section 3.4.2 discusses the class in detail, including descriptions of how to extend the class.

An instance of a class that derives from JeodCheckpointable will not checkpoint or restore itself. Checkpointing and restoring checkpointable objects is performed by the Simulation Engine Interface Model. A checkpointable object must be registered with JEOD via the macro JEOD_REGISTER_CHECKPOINTABLE to make the checkpointable object subject to checkpoint and restart.

4.3.3.2 Extending the SimpleCheckpointable Class

Directly extending JeodCheckpointable requires one to implement each of the JeodCheckpointable pure virtual methods. This is a bit much in the case of a simple, atomic checkpoint and restart process. The class SimpleCheckpointable provides a much simpler checkpoint/restart interface that may be used in the case of these easily restored objects.

For example, consider a model that uses random access to read binary data from some file. The underlying file stream is an opaque data type and is not directly checkpointable or restorable. What can be done is to maintain the name of the file and the location of the read pointer as data members. These are simple data types that the simulation engine can checkpoint and restore. A tiny bit of glue is needed to reopen the file stream. The example below uses a data member whose type derives from SimpleCheckpointable to make the model restartable.

```
#include <cstdio>
#include <string>
#include "utils/container/include/simple_checkpointable.hh"
```

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

class Foo {
public:

   // This class calls Foo's reopen method on restart.
   class Restart : public SimpleCheckpointable {
   public:
      // All checkpointable classes need a default constructor.
      Restart () {}

      // This constructor creates a viable Foo::Restart object.
      explicit Restart (Foo & foo) : foo_ptr(&foo) {}

      // simple_restore() is called by the restart machinery.
      virtual void simple_restore ()
      {
         foo_ptr->reopen();
      }

   private:
      Foo * foo_ptr; // -- Pointer to owner
   };

   // Constructor.
   Foo()
   {
      // All checkpointable objects must be registered with JEOD.
      // First register the containing class, then the object.
      JEOD_REGISTER_CLASS (Foo);
      JEOD_REGISTER_CHECKPOINTABLE (this, restart);
   }

   // Destructor.
   ~Foo()
   {
      // All registered checkpointable objects must be deregistered
      // when they go out of scope.
      JEOD_DEREGISTER_CHECKPOINTABLE (this, restart);
   }

   // Reopen the file on restart. Code elided.
   void reopen ();

protected:
   std::string fname; // -- The name of the file.
```

```
    long foffset; // -- The current offset within the file.

    std::FILE * fstream; // ** The input file stream.
    Restart restart; // ** The checkpoint/restart agent for this class.
};
```

In the above example, the embedded class Foo::Restart derives from SimpleCheckpointable. The class Foo contains an instance of Foo::Restart as a data member. This member acts as a restart agent that restores the containing Foo object's file pointer by calling that object's reopen method.

4.3.3.3 Which Class to Extend

A model developer that wishes to make some object checkpointable and restartable using the mechanisms provided by this model has several choices regarding which class to extend.

- If the object is one of the STL containers such as std::deque for which a JEOD implementation does not exist, the appropriate place to start is by extending the JeodSequenceContainer or JeodAssociativeContainer class as appropriate.
- If the object can be restored by extending SimpleCheckpointable, do so. One could also extend directly from JeodCheckpointable, but that entails a lot more effort.
- Finally, one can write an extension of JeodCheckpointable if all other options are off the table.

4.3.4 Implementing Checkpoint and Restart

The JEOD/Trick simulation interface model implements checkpointing and restarting of checkpointable objects in the Trick simulation environment. When JEOD is used outside of the Trick simulation environment, the implementation must provide an interface between JEOD and the simulation engine. If this new interface is to provide checkpoint/restart capabilities, it must do so in a manner consistent with the capabilities provided by this model. See the *Simulation Engine Interface Model* [7] for details on the implementation of the checkpoint/restart capability in the JEOD/Trick simulation interface.

Chapter 5

Inspections, Tests, and Metrics

5.1 Inspections

This section describes the inspections of the Container Model.

Inspection Container_1: Top-level Inspection

This document structure, the code, and associated files have been inspected. The Container Model satisfies requirement Container_1.

Inspection Container_2: Container Inspection

One purpose of the JEOD container replacements is to transparently encapsulate the C++ standard library list, vector, and set containers in a manner that enables checkpoint/restart. A design goal was to allow future extensions to sequence containers such as std::map and associative containers such as std::map that are not supported in this release.

Tables 65 and 66 in ISO/IEC 14882:2003[4] describe functionality common to all of the container classes. By inspection, the template class JeodContainer provides all types and methods described in these tables except for the Allocated-based `reference` and `const_reference` typedefs. (Support for Allocators is not required in JEOD 2.2.)

Tables 67 and 68 describe the base functionality of the sequence containers. Table 79 describes the base functionality of the associative containers. Items in tables 67 and 69 that share a common signature are implemented in template class JeodContainer. Items in tables 67 and 68 that are common to all sequence containers are implemented in class template JeodSequenceContainer. Items in table 69 are implemented in template class JeodAssociativeContainer.

Sections 23.2.2, 23.2.4, and 23.3.3 of the C++ standard describe the functionality of list, vector, and set. The functionality described in those sections is either implemented in one of the aforementioned base template classes; in the template classes JeodList (list), JeodVector (vector), and JeodSet (set); or are omitted because they relate explicitly to Allocator or Compare classes.

The constructors in the classes JeodList, JeodVector, and JeodSet are protected. The standard requires that the constructors be public. This is addressed in the template classes JeodObjectContainer, JeodPrimitiveContainer, and JeodPointerContainer, which provide the publicly visible constructors as required by the standard (sans the support for non-default Allocators).

By inspection, the Container Model satisfies requirement Container_3.

Inspection Container_3: Checkpoint/Restart Inspection

The primary purpose of the model is to provide a basis for checkpointing and restarting objects whose content is opaque to simulation engines such as Trick. The abstract class JeodCheckpointable defines the basis for these operations. The model provides a simple extension of this class, SimpleCheckpointable, which can be used to checkpoint and restart files.

The template class JeodContainer and its derived classes JeodObjectContainer, JeodPointerContainer, and JeodPrimitiveContainer provide the requisite functionality needed to make the STL container replacements checkpointable and restartable.

By inspection, the Container Model satisfies requirement Container_2.

5.2 Tests

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

Test Container_1: Basic Unit Test

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

Test Description This unit test exercises all of the required C++ container library capabilities except for the lexicological comparisons.

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` to run the test program. The test performs a number of operations on various containers. Each such operation is designed to produce an expected result. The test prints the expected and obtained results for each tested operation.

Success Criteria The success criteria involve analyzing the detailed output that results from running the program.

- The test must build and run to completion.
- The reported output must match expectations.

Test Results The test passes.

Applicable Requirements This test demonstrates the satisfaction of the requirement Container_3 except for the ability to compare objects.

Test Container_2: Pointer List Unit Test

Test Directory `models/utils/container/verif/unit_tests/pointer_list`

Test Description This unit test exercises all of the lexicological comparison operators with a focus on lists of pointers.

Success Criteria

- The test must build and run to completion.
- The reported output must match expectations.

Test Results The test passes.

Applicable Requirements This test demonstrates the satisfaction of the requirement Container_3 with respect to the ability to compare objects.

Test Container_3: Container Simulation

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

Test Description This unit test exercises the checkpoint / restart capabilities of the model.

To run the test, enter the test directory and type the command `CP build` to build the test program and then run the resultant executable with each of the four run directories. The non-restart run directories must be run prior to running the restart directories.

Success Criteria The success criteria involve analyzing the detailed output that results from running the program.

- The simulation must build and each run directory shall run to completion.
- The two checkpoint tests (SET_test/RUN_empty and SET_test/RUN_full) must produce Trick and JEOD checkpoint files that reflect the contents of the containers.
- the two restart tests (SET_test/RUN_empty_restart and SET_test/RUN_full_restart) must restore from the appropriate checkpoint file and must report the same content as reported by the corresponding checkpoint test.

Test Results The test passes.

Applicable Requirements This test demonstrates the satisfaction of the requirement Container_2.

5.3 Requirements Traceability

Table 5.1 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the model.

Table 5.1: Requirements Traceability

Requirement	**Traces to**
Container_1 Top-level requirement	Insp. Container_1 Top-level Inspection
Container_2 Checkpoint/Restart	Insp. Container_3 Checkpoint/Restart Inspection Test Container_3 Container Simulation
Container_3 STL Containers	Insp. Container_2 Container Inspection Test Container_1 Basic Unit Test Test Container_2 Pointer List Unit Test

5.4 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**
Total	0	0	0	0

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::JeodCheckpointable:: JeodCheckpointable (void)	include/checkpointable.hh	196	1
jeod::JeodCheckpointable::~ JeodCheckpointable (void)	include/checkpointable.hh	207	1
jeod::std::get_init_value (void)	include/checkpointable.hh	218	1
jeod::std::get_final_name (void)	include/checkpointable.hh	233	1
jeod::std::get_final_value (void)	include/checkpointable.hh	248	1
jeod::JeodCheckpointable:: pre_checkpoint (void)	include/checkpointable.hh	263	1
jeod::JeodCheckpointable:: post_checkpoint (void)	include/checkpointable.hh	278	1
jeod::JeodCheckpointable:: pre_restart (void)	include/checkpointable.hh	293	1
jeod::JeodCheckpointable:: post_restart (void)	include/checkpointable.hh	308	1
jeod::JeodCheckpointable:: initialize_checkpointable (const void * container JEOD_UNUSED, const std:: type_info & container_type JEOD_UNUSED, const std::string & elem_name JEOD_UNUSED)	include/checkpointable.hh	322	1
jeod::JeodCheckpointable:: undo_initialize_ checkpointable (const void * container JEOD_UNUSED, const std::type_info & container_type JEOD_UNUSED, const std::string & elem_name JEOD_UNUSED)	include/checkpointable.hh	343	1
jeod::JeodContainer::Jeod Container (void)	include/container.hh	106	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodContainer::Jeod Container (const this_ container_type & source)	include/container.hh	117	1
jeod::JeodContainer::Jeod Container (const stl_ container_type & source)	include/container.hh	133	1
jeod::JeodContainer:: ContainerType::operator= (const this_container_type & source)	include/container.hh	149	2
jeod::JeodContainer:: ContainerType::operator= (const stl_container_type & source)	include/container.hh	165	1
jeod::JeodContainer::~Jeod Container (void)	include/container.hh	179	1
jeod::JeodContainer:: ContainerType::swap_ contents (this_container_ type & other)	include/container.hh	185	1
jeod::JeodContainer:: ContainerType::swap_ contents (stl_container_type & other)	include/container.hh	195	1
jeod::JeodContainer::std:: perform_cleanup_action (const std::string & value J EOD_UNUSED)	include/container.hh	219	1
jeod::JeodContainer::std:: initialize_checkpointable (const void * container JEO D_UNUSED, const std:: type_info & container_type JEOD_UNUSED, const std::string & elem_name JE OD_UNUSED)	include/container.hh	231	2
jeod::JeodContainer::start_ checkpoint (void)	include/container.hh	249	1
jeod::JeodContainer::advance_ checkpoint (void)	include/container.hh	261	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodContainer::is_checkpoint_finished (void)	include/container.hh	273	1
jeod::JeodContainer::std::get_init_name (void)	include/container.hh	283	1
jeod::JeodContainer::std::get_item_name (void)	include/container.hh	294	1
jeod::JeodContainer::std::get_final_name (void)	include/container.hh	304	1
jeod::JeodContainer::std::perform_restore_action (const std::string & action_name, const std::string & action_value)	include/container.hh	316	4
jeod::JeodAssociativeContainer::~JeodAssociativeContainer (void)	include/jeod_associative_container.hh	149	1
jeod::JeodAssociativeContainer::key_comp (void)	include/jeod_associative_container.hh	157	1
jeod::JeodAssociativeContainer::value_comp (void)	include/jeod_associative_container.hh	166	1
jeod::JeodAssociativeContainer::base_container_type::count (const key_type & x)	include/jeod_associative_container.hh	184	1
jeod::JeodAssociativeContainer::base_container_type::find (const key_type & x)	include/jeod_associative_container.hh	193	1
jeod::JeodAssociativeContainer::base_container_type::find (const key_type & x)	include/jeod_associative_container.hh	202	1
jeod::JeodAssociativeContainer::base_container_type::lower_bound (const key_type & x)	include/jeod_associative_container.hh	211	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodAssociative Container::base_container_ type::lower_bound (const key_type & x)	include/jeod_associative_ container.hh	220	1
jeod::JeodAssociative Container::base_container_ type::upper_bound (const key_type & x)	include/jeod_associative_ container.hh	229	1
jeod::JeodAssociative Container::base_container_ type::upper_bound (const key_type & x)	include/jeod_associative_ container.hh	238	1
jeod::JeodAssociative Container::std::equal_range (const key_type & x)	include/jeod_associative_ container.hh	247	1
jeod::JeodAssociative Container::std::equal_range (const key_type & x)	include/jeod_associative_ container.hh	257	1
jeod::JeodAssociative Container::insert (Input Iterator first, InputIterator last)	include/jeod_associative_ container.hh	274	1
jeod::JeodAssociative Container::std::insert (const typename base_container_ type::value_type & new_ elem)	include/jeod_associative_ container.hh	289	1
jeod::JeodAssociative Container::base_container_ type::erase (typename base_ container_type::iterator position)	include/jeod_associative_ container.hh	300	1
jeod::JeodAssociative Container::base_container_ type::erase (typename base_ container_type::iterator first, typename base_ container_type::iterator last)	include/jeod_associative_ container.hh	311	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodAssociative Container::base_container_ type::erase (const key_type & x)	include/jeod_associative_ container.hh	324	1
jeod::JeodAssociative Container::JeodAssociative Container (void)	include/jeod_associative_ container.hh	342	1
jeod::JeodAssociative Container::JeodAssociative Container (const this_ container_type & src)	include/jeod_associative_ container.hh	350	1
jeod::JeodAssociative Container::JeodAssociative Container (const Container Type & src)	include/jeod_associative_ container.hh	358	1
jeod::JEOD_CONTAINER_G ENERATE_THREE_COM PARATORS (¡)	include/jeod_container_ compare.hh	157	1
jeod::operator ¡ (const ContainerType & x, const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	190	1
jeod::operator ¡ (const jeod:: JeodSTLContainer¡Elem Type, ContainerType¿ & x, const jeod::JeodSTL Container¡ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	204	1
jeod::operator == (const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & x, const ContainerType & y)	include/jeod_container_ compare.hh	219	1
jeod::operator == (const ContainerType & x, const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	233	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::operator == (const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & x, const jeod::JeodSTL Container¡ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	247	1
jeod::operator ¿ (const jeod:: JeodSTLContainer¡Elem Type, ContainerType¿ & x, const ContainerType & y)	include/jeod_container_ compare.hh	262	1
jeod::operator ¿ (const ContainerType & x, const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	276	1
jeod::operator ¿ (const jeod:: JeodSTLContainer¡Elem Type, ContainerType¿ & x, const jeod::JeodSTL Container¡ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	290	1
jeod::operator ¿= (const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & x, const ContainerType & y)	include/jeod_container_ compare.hh	305	1
jeod::operator ¿= (const ContainerType & x, const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	319	1
jeod::operator ¿= (const jeod::JeodSTLContainer¡ ElemType, ContainerType¿ & x, const jeod::JeodSTL Container¡ElemType, ContainerType¿ & y)	include/jeod_container_ compare.hh	333	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::operator != (const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & x, const ContainerType & y)	include/jeod_container_compare.hh	348	1
jeod::operator != (const ContainerType & x, const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & y)	include/jeod_container_compare.hh	362	1
jeod::operator != (const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & x, const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & y)	include/jeod_container_compare.hh	376	1
jeod::operator ¡= (const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & x, const ContainerType & y)	include/jeod_container_compare.hh	391	1
jeod::operator ¡= (const ContainerType & x, const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & y)	include/jeod_container_compare.hh	405	1
jeod::operator ¡= (const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & x, const jeod::JeodSTLContainer¡ElemType, ContainerType¿ & y)	include/jeod_container_compare.hh	419	1
jeod::JeodList::~JeodList (void)	include/jeod_list.hh	128	1
jeod::JeodList::jeod_stl_container_type::operator= (const this_container_type & src)	include/jeod_list.hh	136	1
jeod::JeodList::jeod_stl_container_type::operator= (const stl_container_type & src)	include/jeod_list.hh	146	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodList::merge (stl_ container_type & other)	include/jeod_list.hh	159	1
jeod::JeodList::merge (stl_ container_type & other, Compare comp)	include/jeod_list.hh	170	1
jeod::JeodList::push_front (const ElemType & elem)	include/jeod_list.hh	183	1
jeod::JeodList::pop_front (void)	include/jeod_list.hh	193	1
jeod::JeodList::remove (const ElemType & value)	include/jeod_list.hh	202	1
jeod::JeodList::remove_if (Predicate pred)	include/jeod_list.hh	211	1
jeod::JeodList::reverse (void)	include/jeod_list.hh	223	1
jeod::JeodList::jeod_stl_ container_type::splice (typename jeod_stl_ container_type::iterator position, stl_container_type & other)	include/jeod_list.hh	232	1
jeod::JeodList::jeod_stl_ container_type::splice (typename jeod_stl_ container_type::iterator position, stl_container_type & other, typename jeod_stl_ container_type::iterator other_pos)	include/jeod_list.hh	243	1
jeod::JeodList::jeod_stl_ container_type::splice (typename jeod_stl_ container_type::iterator position, stl_container_type & other, typename jeod_stl_ container_type::iterator first, typename jeod_stl_ container_type::iterator last)	include/jeod_list.hh	256	1
jeod::JeodList::sort (void)	include/jeod_list.hh	270	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodList::sort (Compare comp)	include/jeod_list.hh	279	1
jeod::JeodList::unique (void)	include/jeod_list.hh	291	1
jeod::JeodList::unique (BinaryPredicate comp)	include/jeod_list.hh	300	1
jeod::JeodList::JeodList (void)	include/jeod_list.hh	314	1
jeod::JeodList::JeodList (const this_container_type & src)	include/jeod_list.hh	319	1
jeod::JeodList::JeodList (const stl_container_type & src)	include/jeod_list.hh	326	1
jeod::JeodSequence Container::~JeodSequence Container (void)	include/jeod_sequence_ container.hh	129	1
jeod::JeodSequence Container::base_container_ type::back (void)	include/jeod_sequence_ container.hh	137	1
jeod::JeodSequence Container::base_container_ type::back (void)	include/jeod_sequence_ container.hh	146	1
jeod::JeodSequence Container::base_container_ type::front (void)	include/jeod_sequence_ container.hh	155	1
jeod::JeodSequence Container::base_container_ type::front (void)	include/jeod_sequence_ container.hh	164	1
jeod::JeodSequence Container::assign (Input Iterator first, InputIterator last)	include/jeod_sequence_ container.hh	176	1
jeod::JeodSequence Container::base_container_ type::assign (typename base_container_type::size_ type new_size, const Elem Type & new_elem)	include/jeod_sequence_ container.hh	190	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodSequence Container::base_container_ type::erase (typename base_ container_type::iterator position)	include/jeod_sequence_ container.hh	203	1
jeod::JeodSequence Container::base_container_ type::erase (typename base_ container_type::iterator first, typename base_ container_type::iterator last)	include/jeod_sequence_ container.hh	214	1
jeod::JeodSequence Container::base_container_ type::insert (typename base_container_type:: iterator position, Input Iterator first, InputIterator last)	include/jeod_sequence_ container.hh	231	1
jeod::JeodSequence Container::base_container_ type::insert (typename base_container_type:: iterator position, typename base_container_type::size_ type ncopies, const Elem Type & new_elem)	include/jeod_sequence_ container.hh	248	1
jeod::JeodSequence Container::base_container_ type::resize (typename base_container_type::size_ type new_size, ElemType new_elem = ElemType()	include/jeod_sequence_ container.hh	264	1
jeod::JeodSequence Container::push_back (const ElemType & elem)	include/jeod_sequence_ container.hh	277	1
jeod::JeodSequence Container::pop_back (void)	include/jeod_sequence_ container.hh	288	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodSequence Container::JeodSequence Container (void)	include/jeod_sequence_ container.hh	304	1
jeod::JeodSequence Container::JeodSequence Container (const this_ container_type & src)	include/jeod_sequence_ container.hh	312	1
jeod::JeodSequence Container::JeodSequence Container (const Container Type & src)	include/jeod_sequence_ container.hh	320	1
jeod::JeodSet::~JeodSet (void)	include/jeod_set.hh	118	1
jeod::JeodSet::jeod_stl_ container_type::operator= (const this_container_type & src)	include/jeod_set.hh	126	1
jeod::JeodSet::jeod_stl_ container_type::operator= (const stl_container_type & src)	include/jeod_set.hh	136	1
jeod::JeodSet::JeodSet (void)	include/jeod_set.hh	148	1
jeod::JeodSet::JeodSet (const this_container_type & src)	include/jeod_set.hh	153	1
jeod::JeodSet::JeodSet (const stl_container_type & src)	include/jeod_set.hh	160	1
jeod::JeodSTLContainer::~ JeodSTLContainer (void)	include/jeod_stl_container.hh	177	1
jeod::JeodSTLContainer:: operator= (const this_ container_type & src)	include/jeod_stl_container.hh	204	2
jeod::JeodSTLContainer:: operator= (const Container Type & src)	include/jeod_stl_container.hh	218	2
jeod::JeodSTLContainer::get_ allocator (void)	include/jeod_stl_container.hh	235	1
jeod::JeodSTLContainer:: begin (void)	include/jeod_stl_container.hh	247	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodSTLContainer:: begin (void)	include/jeod_stl_container.hh	256	1
jeod::JeodSTLContainer::end (void)	include/jeod_stl_container.hh	265	1
jeod::JeodSTLContainer::end (void)	include/jeod_stl_container.hh	274	1
jeod::JeodSTLContainer:: rbegin (void)	include/jeod_stl_container.hh	283	1
jeod::JeodSTLContainer:: rbegin (void)	include/jeod_stl_container.hh	292	1
jeod::JeodSTLContainer::rend (void)	include/jeod_stl_container.hh	301	1
jeod::JeodSTLContainer::rend (void)	include/jeod_stl_container.hh	310	1
jeod::JeodSTLContainer:: empty (void)	include/jeod_stl_container.hh	322	1
jeod::JeodSTLContainer:: max_size (void)	include/jeod_stl_container.hh	331	1
jeod::JeodSTLContainer::size (void)	include/jeod_stl_container.hh	340	1
jeod::JeodSTLContainer:: clear (void)	include/jeod_stl_container.hh	352	1
jeod::JeodSTLContainer:: insert (iterator position, const value_type & new_ elem)	include/jeod_stl_container.hh	361	1
jeod::JeodSTLContainer:: JeodSTLContainer (void)	include/jeod_stl_container.hh	383	1
jeod::JeodSTLContainer:: JeodSTLContainer (const this_container_type & src)	include/jeod_stl_container.hh	391	1
jeod::JeodSTLContainer:: JeodSTLContainer (const ContainerType & src)	include/jeod_stl_container.hh	399	1
jeod::JeodSTLContainer:: swap (this_container_type & other)	include/jeod_stl_container.hh	412	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodSTLContainer::swap (ContainerType & other)	include/jeod_stl_container.hh	422	1
jeod::JeodVector::~JeodVector (void)	include/jeod_vector.hh	124	1
jeod::JeodVector::jeod_stl_container_type::operator= (const this_container_type & src)	include/jeod_vector.hh	131	1
jeod::JeodVector::jeod_stl_container_type::operator= (const stl_container_type & src)	include/jeod_vector.hh	141	1
jeod::JeodVector::jeod_stl_container_type::capacity (void)	include/jeod_vector.hh	154	1
jeod::JeodVector::jeod_stl_container_type::reserve (typename jeod_stl_container_type::size_type n)	include/jeod_vector.hh	163	1
jeod::JeodVector::stl_container_type::operator[] (std::size_t n)	include/jeod_vector.hh	177	1
jeod::JeodVector::stl_container_type::operator[] (std::size_t n)	include/jeod_vector.hh	187	1
jeod::JeodVector::stl_container_type::at (std::size_t n)	include/jeod_vector.hh	197	1
jeod::JeodVector::stl_container_type::at (std::size_t n)	include/jeod_vector.hh	207	1
jeod::JeodVector::JeodVector (void)	include/jeod_vector.hh	220	1
jeod::JeodVector::JeodVector (const this_container_type & src)	include/jeod_vector.hh	225	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodVector::JeodVector (const stl_container_type & src)	include/jeod_vector.hh	232	1
jeod::JeodObjectContainer:: JeodObjectContainer (const JeodObject Container & source)	include/object_container.hh	98	1
jeod::JeodObjectContainer:: ContainerType::JeodObject Container (const typename ContainerType::stl_ container_type & source)	include/object_container.hh	111	1
jeod::JeodObjectContainer:: ElemType¿::operator= (const JeodObject Container & source)	include/object_container.hh	125	1
jeod::JeodObjectContainer:: ContainerType::operator= (const typename Container Type::stl_container_type & source)	include/object_container.hh	138	1
jeod::JeodObjectContainer::~ JeodObjectContainer (void)	include/object_container.hh	151	1
jeod::JeodObjectContainer:: ContainerType::pre_ checkpoint (void)	include/object_container.hh	157	3
jeod::JeodObjectContainer:: post_checkpoint (void)	include/object_container.hh	179	2
jeod::JeodObjectContainer:: post_restart (void)	include/object_container.hh	190	1
jeod::JeodObjectContainer:: ElemType¿::start_ checkpoint (void)	include/object_container.hh	198	1
jeod::JeodObjectContainer:: ElemType¿::advance_ checkpoint (void)	include/object_container.hh	209	1
jeod::JeodObjectContainer:: std::get_item_value (void)	include/object_container.hh	220	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodObjectContainer:: std::perform_insert_action (const std::string & value)	include/object_container.hh	232	1
jeod::JeodObjectContainer:: std::get_final_value (void)	include/object_container.hh	248	1
jeod::JeodObjectContainer:: std::perform_cleanup_action (const std::string & value)	include/object_container.hh	260	2
jeod::JeodPointerContainer:: JeodPointerContainer (void)	include/pointer_container.hh	83	1
jeod::JeodPointerContainer:: JeodPointerContainer (const JeodPointer Container & source)	include/pointer_container.hh	93	1
jeod::JeodPointerContainer:: ContainerType::Jeod PointerContainer (const typename ContainerType:: stl_container_type & source)	include/pointer_container.hh	105	1
jeod::JeodPointerContainer:: ElemType*¿::operator= (const JeodPointer Container & source)	include/pointer_container.hh	118	1
jeod::JeodPointerContainer:: ContainerType::operator= (const typename Container Type::stl_container_type & source)	include/pointer_container.hh	131	1
jeod::JeodPointerContainer::~ JeodPointerContainer (void)	include/pointer_container.hh	144	1
jeod::JeodPointerContainer:: std::initialize_ checkpointable (const void * container, const std::type_ info & container_type, const std::string & elem_name)	include/pointer_container.hh	149	2
jeod::JeodPointerContainer:: std::get_item_value (void)	include/pointer_container.hh	170	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodPointerContainer:: std::perform_insert_action (const std::string & value)	include/pointer_container.hh	183	1
jeod::JeodPrimitive Container::JeodPrimitive Container (void)	include/primitive_ container.hh	86	1
jeod::JeodPrimitive Container::JeodPrimitive Container (const Jeod PrimitiveContainer & source)	include/primitive_ container.hh	91	1
jeod::JeodPrimitive Container::ContainerType:: JeodPrimitiveContainer (const typename Container Type::stl_container_type & source)	include/primitive_ container.hh	102	1
jeod::JeodPrimitive Container::ElemType¿:: operator= (const Jeod PrimitiveContainer & source)	include/primitive_ container.hh	114	1
jeod::JeodPrimitive Container::ContainerType:: operator= (const typename ContainerType::stl_ container_type & source)	include/primitive_ container.hh	127	1
jeod::JeodPrimitive Container::~JeodPrimitive Container (void)	include/primitive_ container.hh	140	1
jeod::JeodPrimitive Container::std::get_item_ value (void)	include/primitive_ container.hh	145	1
jeod::JeodPrimitive Container::std::perform_ insert_action (const std:: string & value)	include/primitive_ container.hh	154	1
jeod::JeodPrimitiveSerializer Base::JeodPrimitive SerializerBase (void)	include/primitive_serializer.hh	82	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodPrimitiveSerializer Base::~JeodPrimitive SerializerBase (void)	include/primitive_serializer.hh	87	1
jeod::JeodPrimitiveSerializer:: JeodPrimitiveSerializer (void)	include/primitive_serializer.hh	112	1
jeod::JeodPrimitiveSerializer:: ~JeodPrimitiveSerializer (void)	include/primitive_serializer.hh	117	1
jeod::JeodPrimitiveSerializer:: std::to_string (const Type & val)	include/primitive_serializer.hh	122	1
jeod::JeodPrimitiveSerializer:: std::from_string (const std:: string & val)	include/primitive_serializer.hh	132	1
jeod::std::to_string (const std::string & val)	include/primitive_serializer.hh	158	1
jeod::std::from_string (const std::string & val)	include/primitive_serializer.hh	171	1
jeod::std::to_string (const float & val)	include/primitive_serializer.hh	183	1
jeod::JeodPrimitive Serializer¡float¿::from_string (const std::string & val)	include/primitive_serializer.hh	195	1
jeod::std::to_string (const double & val)	include/primitive_serializer.hh	207	1
jeod::JeodPrimitive Serializer¡double¿::from_ string (const std::string & val)	include/primitive_serializer.hh	219	1
jeod::std::to_string (const long double & val)	include/primitive_serializer.hh	231	1
jeod::double¿::from_string (const std::string & val)	include/primitive_serializer.hh	243	1
jeod::SimpleCheckpointable::~ SimpleCheckpointable (void)	include/simple_ checkpointable.hh	97	1
jeod::SimpleCheckpointable:: std::get_init_name (void)	include/simple_ checkpointable.hh	102	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::SimpleCheckpointable:: std::get_item_name (void)	include/simple_ checkpointable.hh	110	1
jeod::SimpleCheckpointable:: std::get_item_value (void)	include/simple_ checkpointable.hh	117	1
jeod::SimpleCheckpointable:: start_checkpoint (void)	include/simple_ checkpointable.hh	124	1
jeod::SimpleCheckpointable:: advance_checkpoint (void)	include/simple_ checkpointable.hh	130	1
jeod::SimpleCheckpointable:: is_checkpoint_finished (void)	include/simple_ checkpointable.hh	137	1
jeod::SimpleCheckpointable:: std::perform_restore_action (const std::string & action_ name, const std::string & action_value JEOD_UNUSE D)	include/simple_ checkpointable.hh	145	2
jeod::std::serialize_string (const std::string & val)	src/primitive_serializer.cc	41	5
jeod::std::deserialize_string (const std::string & val)	src/primitive_serializer.cc	86	5
jeod::std::serialize_float (const float & val)	src/primitive_serializer.cc	128	4
jeod::JeodPrimitiveSerializer Base::deserialize_float (const std::string & val)	src/primitive_serializer.cc	161	4
jeod::std::serialize_double (const double & val)	src/primitive_serializer.cc	190	4
jeod::JeodPrimitiveSerializer Base::deserialize_double (const std::string & val)	src/primitive_serializer.cc	223	4
jeod::std::serialize_long_double (const long double & val)	src/primitive_serializer.cc	252	4
jeod::JeodPrimitiveSerializer Base::deserialize_long_ double (const std::string & val)	src/primitive_serializer.cc	285	4

Bibliography

[1] Generated by doxygen. *Container Model Reference Manual*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics and Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, July 2023.

[2] Hammen, D. Memory Management Model. Technical Report JSC-61777-utils/memory, 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] ISO/IEC. *ISO/IEC 14882:2003: Programming languages: C++*. American National Standards Institute, New York, New York, second edition, 2003.

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

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

[7] Shelton, R. Simulation Engine Interface Model. Technical Report JSC-61777-utils/sim_interface, NASA, Johnson Space Center, Houston, Texas, July 2023.

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