SystemC Checkpoint Library

July 20, 2021

Contents

Chapter 1

Introduction

The SystemC Checkpoint Library provides the capability to save and restore the state of SystemC devices. Using the provided API users may add checkpoint support to their devices. C++ and SystemC data types, user defined data types, and pointers may be saved in checkpoints, as well as generic payloads and STL containers.

Furthermore, the checkpoint library provides infrastructure for *state keepers*. These are useful when it is desireable for applications to handle device state separately. For example, this may be the case for the saving and restoration of large memory images.

The checkpoint library is integrated into the SystemC Library; this makes the saving and restoration of SystemC state analogous to the rest of Simics. The checkpoint library may also be used independently of Simics. Using the provided C++ API, a user may instruct the library to save the state to memory or to disk without involving Simics.

This manual details the use of the checkpoint library. In particular, it describes how to use the provided API to serialize SystemC modules and other user types. As the library evolves, more details will be added to this manual.

Chapter 2

Concept Overview

The checkpoint API relies heavily on the Boost C++ Libraries (`www.boost.org`). Specifically, it makes use of the Boost serialization library. On top of that, the SystemC checkpoint API is built on a small set of concepts that need to be understood by users. These concepts are briefly described here. Details can be found in the corresponding sections of this manual.

2.1 Serializers

All classes that contain state should implement a ***serialize*** function. This function is invoked when checkpoints are saved or loaded and should serialize any state within the class (see 3.1). Any subsequent objects contained within the class may in turn be serialized by invoking their ***serialize*** function.

Thus, SystemC modules are responsible for serializing the state which they contain and making sure that any underlying objects are serialized. SystemC modules themselves are serialized by the checkpoint library. The code that provides a module is responsible for instantiating a **Serializer**, which registers the module with the framework so that it may be automatically serialized. That is, a library providing modules is responsible for registering those modules, and a programmer creating a new module is responsible for registering it with the checkpoint library.

This is described in detail in Section 3.

2.2 Non-modules

Types that are not SystemC modules need to be explicitly serialized in the ***serialize*** function of the class that contains them. For these types, there is no need to create a corresponding **Serializer** object since their ***serialize*** functions are explicitly invoked by their parent.

The user does not have to provide ***serialize*** functions for all types, however. For instance, C++ data types are supported by the Boost serialization library out-of-the-box. Furthermore, the checkpoint library provide serialization support for some SystemC types and STL containers.

See Section 3 for details.

2.3 State Keepers

The checkpoint library provides State Keepers, which let users handle the saving and restoration of state for parts of their system. This is useful when dealing with large memory images where incremental checkpoints is important for performance.

All SystemC objects implementing the *state-keeper* interface will automatically be found by the checkpoint library. There is no need to register a ***serialize*** function for these objects. Instead, the methods in the *state-keeper* interface are used for checkpoint handling.

This is detailed in Section 4.

2.4 Threads

Although the checkpoint library does not save and restore the state of threads, the library provides the means to restart threads and restore the corresponding thread stacks. This is accomplished by a combination of using the ***serialize*** function of the parent module to restore the state of the thread and a following reset of the thread process. The checkpoint library guarantees that all ***serialize*** functions are called before threads are restarted.

See Section 5 for details.

2.5 Payloads

The checkpoint library also provides infrastructure for the restoration of generic payloads. In order to restore payloads, the objects that reference them must serialize them as part of their ***serialize*** functions, and a **PayloadUpdate** object should be provided to restore the contents of the payload.

This is described in Section 3.3

Chapter 3

Serialization

In order to interact with the Boost serialization library, the checkpoint library provides a custom *archive* along with a set of macros and functions that are used with the serialization framework to transform data structures to text and back.

With the help of this archive, the checkpoint library can serialize SystemC state and format the output as a JSON text file. It is important to note that only the state is saved and restored; the checkpoint library makes no attempt to restore the configuration that was used when a checkpoint was saved. This responsibility is entirely with the user. Thus, the user is responsible for making sure that the configuration when loading a checkpoint is the same as the one that was used when the checkpoint was created.

This section describes the part of the API that allows a user to checkpoint modules and custom types.

3.1 Serializing User Types

Classes that should be serialized implement a function that saves and restores any state within that class. For example, a checkpointable user module may look like this:

```
SC_MODULE(UserModule) {
  public:
    SC_CTOR(UserModule) : a_(0), b_(0) {}

  private:
    int a_;
    int b_;

    friend class cci::serialization::access;
    template <class Archive>
        void serialize(Archive& ar, const unsigned int version) {
        ar & SMD(a_);
        ar & SMD(b_);
    }
};
```

The ***serialize*** function takes an arbitrary archive as a parameter. When the configuration is saved, the members are read into the archive; when the configuration is restored, the members are read from the archive. If any of the serialized members in turn implement a ***serialize*** function it will be invoked by the framework. This way, the serialization propagates through the data structure.

The ***SMD*** macro is used to extract meta-data from the variables and must be used. This meta-data is currently made up of the variable name and its type, but more information may be extracted in the future.

In case the user wishes to explicitly name any serialized state with anything other than the variable name, ***create_smd*** can be used in place of the regular ***SMD*** macro. This is recommended for the purposes of backwards compatibility when the contents otherwise wrapped in the ***SMD*** macro are likely to change.

Please observe that it is forbidden to serialize the same instance multiple times. The result of serializing the same instance multiple times is undefined.

For more information about the Boost serialization framework, such as how to split the serialization into explicit save and load functions, or how to implement the ***serialize*** function outside of classes, please see the Boost serialization home page (`www.boost.org/libs/serialization/doc/`).

The ***SMD*** macro is defined in `systemc-checkpoint/serialization/smd.h`.

The ***create_smd*** function is defined in
`systemc-checkpoint/serialization/serializable_meta_data.h`.

3.2 Type Registration

When all modules that need to save and restore state have implemented ***serialize***, they must be registered in order for the checkpoint library to save and restore them at appropriate times. This is done by creating a corresponding **Serializer**:

```
static sc_checkpoint::serialization::Serializer<UserModule> um_serializer;
```

This is sufficient for instances of **UserModule** to be automatically saved and restored whenever the checkpoint library is instructed to do so, which is handled automatically by Simics or performed manually using the Checkpoint Control Interface (see 8). Note that the **Serializer** is created on a class-by-class basis; it is not necessary to instantiate a **Serializer** for every class instance (this will work, but will unnecessarily serialize the objects as many times as there are instances).

It is not necessary to instantiate a **Serializer** type for underlying types that are not modules; these types only need to implement ***serialize*** and be explicitly archived in the ***serialize*** function of the module that contains them:

```
SC_MODULE(A) {
  public:
    SC_CTOR(A) : a_(0) {}
    int a_;

    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & SMD(a_);
    }
};

class B {
  public:
    B() : b_(0) {}
    int b_;

    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & SMD(b_);
    }
};

SC_MODULE(C) {
  public:
    SC_CTOR(C) : a_("a") {}
    A a_;
    B b_;

    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        // no need to explicitly serialize a_ since it is a module
        ar & SMD(b_);
    }
};

static sc_checkpoint::serialization::Serializer<A> a_serializer;
```

```
// there is no serializer for 'B'
static sc_checkpoint::serialization::Serializer<C> c_serializer;
```

If using a third-party library that in turn uses the checkpoint library to provide checkpointable modules, that library is responsible for providing the necessary **Serializer** instances. Thus, it should not be required to instantiate **Serializer**s for these modules.

The **Serializer** type is defined in `systemc-checkpoint/serialization/serializer.h`.

3.3 Serializing Generic Payloads

The checkpoint library supports the serialization of generic payloads. Provided that the ***serialize*** functions of modules are implemented appropriately, references to generic payloads between modules are restored correctly.

In order to serialize payloads, `systemc-checkpoint/serialization/payload.h` must be included and payload members serialized as usual. However, a **PayloadUpdate** class must also be provided and instantiated. This class is responsible for restoring the data contained in the payload, which the checkpoint library does not know how to restore.

The base class that is used to create new payload updaters is **PayloadUpdate**. The **PayloadUpdate** class is templetized and takes the owner module (the module that 'owns' the payload in question) as an argument.

In order to create a new payload updater, derive the **PayloadUpdate** class and implement the required methods that restore payload data:

```
template <class T>
class PayloadUpdate : public PayloadUpdateInterface {
  public:
    PayloadUpdate() {
        PayloadUpdateRegistry::Instance()->registerUpdater(this);
    }
    virtual ~PayloadUpdate() {
        PayloadUpdateRegistry::Instance()->unregisterUpdater(this);
    }

    virtual void set_data_ptr(T *t, tlm::tlm_generic_payload *payload) {}
    virtual void set_byte_enable_ptr(T *t,
                                     tlm::tlm_generic_payload *payload) {}
    virtual void set_extensions(T *t, tlm::tlm_generic_payload *payload) {}
    virtual void set_memory_manager(T *t, tlm::tlm_generic_payload *payload) {}

  private:
    virtual bool isUpdatable(sc_core::sc_object *object) {
        return dynamic_cast<T *> (object) != NULL;
    }
    virtual void updateDataPtr(sc_core::sc_object *object,
                               tlm::tlm_generic_payload *payload) {
        set_data_ptr(static_cast<T *>(object), payload);
    }
    virtual void updateByteEnablePtr(sc_core::sc_object *object,
                                     tlm::tlm_generic_payload *payload) {
        set_byte_enable_ptr(static_cast<T *>(object), payload);
    }
    virtual void updateExtensions(sc_core::sc_object *object,
                                  tlm::tlm_generic_payload *payload) {
        set_extensions(static_cast<T *>(object), payload);
    }
    virtual void updateMemoryManager(sc_core::sc_object *object,
                                     tlm::tlm_generic_payload *payload) {
        set_memory_manager(static_cast<T *>(object), payload);
    }
};
```

For example, a SystemC module that saves and restores a generic payload, a reference to which may be saved and restored by other modules, with some data pointer must define a **PayloadUpdate** class:

```
SC_MODULE(PayloadModule) {
  public:
    SC_CTOR(PayloadModule) {
        payload_ = new tlm::tlm_generic_payload;
    }
    template <class Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & SMD(data_)
           & SMD(payload_);
    }

    unsigned char data_[10];
    tlm::tlm_generic_payload *payload_;
};

class PayloadUpdater : public sc_checkpoint::PayloadUpdate<PayloadModule> {
  public:
    PayloadUpdater() {}
    virtual void set_data_ptr(PayloadModule *m,
                              tlm::tlm_generic_payload *p) {
        p->set_data_ptr(m->data_);
    }
};

static sc_checkpoint::serialization::Serializer<PayloadModule> p_serializer;
static PayloadUpdater p_updater;
```

Please note that the **PayloadUpdater** is invoked when the payload is serialized, so any state in the module accessed by the **PayloadUpdate** must have been restored by that point.

The **PayloadUpdate** definition is found in `systemc-checkpoint/payload_update.h`.

3.4 STL Containers

Like the Boost serialization library, the checkpoint library supports serialization of STL containers. In order to serialize a certain container, a corresponding header from the checkpoint library should be included and the container serialized like any other variable.

Adhere to the following table when including files necessary for the serialization of a certain container:

Container	Header
bitset	systemc-checkpoint/serialization/bitset.h
deque	systemc-checkpoint/serialization/deque.h
list	systemc-checkpoint/serialization/list.h
map/multimap	systemc-checkpoint/serialization/map.h
pair	systemc-checkpoint/serialization/pair.h
queue/priority_queue	systemc-checkpoint/serialization/queue.h
set/multiset	systemc-checkpoint/serialization/set.h
stack	systemc-checkpoint/serialization/stack.h
valarray	systemc-checkpoint/serialization/valarray.h
vector	systemc-checkpoint/serialization/vector.h

3.5 Arrays

Serialization of C++ arrays is also supported. The header that facilitates serialization support of C++ arrays is `systemc-checkpoint/serialization/array.h`.

In order to serialize a static array, one may pass it to the archive like so:

```
template <class Archive>
void serialize(Archive& ar, const unsigned int version) {
    ar & SMD(static_array_);
}
```

The array, one-dimensional or otherwise, will be automatically recognized as an array and serialized accordingly. Alternatively, the ***make_array*** function may be used to serialize heap allocated arrays:

```
template<class Archive>
void save(Archive &ar, const unsigned int version) const {
    ar << SMD(size_)
       << sc_checkpoint::serialization::create_smd(
              "array", sc_checkpoint::serialization::make_array(array_, size_));
}

template<class Archive>
void load(Archive &ar, const unsigned int version) {
    delete[] array_;
    ar >> SMD(size_);

    array_ = new int[size_];
    ar >> sc_checkpoint::serialization::create_smd(
              "array", sc_checkpoint::serialization::make_array(array_, size_));
}

CCI_SERIALIZATION_SPLIT_MEMBER()
```

Please note that the user is responsible for the memory management of heap allocated arrays.

3.6 Base Classes

In order to serialize the base of a derived class, use the provided ***SMD_BASE_OBJECT*** macro. This will invoke the ***serialize*** function of the base class. For example, for a class derived from the base class **BaseClass**:

```
template <class Archive>
void serialize(Archive& ar, const unsigned int version) {
    ar & SMD_BASE_OBJECT(Base);
}
```

The ***SMD_BASE_OBJECT*** macro is found alongside the regular ***SMD*** macro in `systemc-checkpoint/serialization/smd.h`.

Chapter 4

State Keeper Interface

In some cases it can be useful to leave the saving and restoration of state to the user. For example, if large memories are part of the configuration it can be bad for the performance if the entire memory image is saved whenever a checkpoint is saved. Rather, it can be appropriate to only save iterative changes of affected pages to save precious memory.

For this use-case the checkpoint API provides an interface that can be used to register a 'state keeper' that handles the saving and restoration of some component in the system. State keepers and regular serialization can be used side-by-side.

```
class StateKeeperInterface {
  public:
    virtual ~StateKeeperInterface() {}

    /**
     * State in the implementer is saved incrementally using this function. The
     * function must produce a handle which the library can use to restore or
     * merge with other in-memory states.
     */
    virtual bool save(void **handleOut) = 0;

    /**
     * Arbitrary state is restored in the implementer when this function is
     * invoked with a corresponding handle.
     */
    virtual bool restore(void *handle) = 0;

    /**
     * This function merges two states that are identified by the handles. The
     * contents of handleRemove are appended to handlePrevious, and
     * handleRemove is subsequently removed rendering the handle invalid.
     * NOTE: handlePrevious could be NULL, in which case handleRemove is just
     * removed
     */
    virtual bool merge(void *handlePrevious, void *handleRemove) = 0;

    /**
     * Write the state to disk. Dir is the directory in which state shall be
     * saved and prefix is a directory unique identifier. Persistent is a
     * boolean that indicates whether or not only persistent state should be
     * saved to disk. If persistent is not set all state is saved.
     */
    virtual bool write(const std::string &dir,
                       const std::string &prefix,
                       bool persistent) = 0;

    /**
     * Write a standalone checkpoint to disk. That is, the written checkpoint
     * cannot be dependent on any previous checkpoints. Dir is the directory in
     * which state shall be saved and prefix is a directory unique
     * identifier. Persistent is a boolean that indicates whether or not only
     * persistent state should be saved to disk. If persistent is not set all

     * state is saved.
     */
    virtual bool write_standalone(
        const std::string &dir, const std::string &prefix, bool persistent) = 0;

    /**
     * Restore state from disk. Dirs is a list of checkpoint directories and
     * prefix is an identifier.
     */
    virtual bool read(const std::vector<std::string> &dirs,
                      const std::string prefix, bool persistent) = 0;
};
```

StateKeeperInterface provides three methods that manage in-memory checkpoints, meaning checkpoints that are not saved to disk. These checkpoints may be incremental and do not necessarily contain the complete state. This is preferable because saving incremental changes for memory images can save a substantial amount of memory. The methods in question work with 'handles' that are originally returned by ***save***, subsequently used to load a checkpoint using ***load***, and finally to merge existing checkpoints using ***merge***. The checkpoint infrastructure may at any time invoke ***merge*** in order to combine two checkpoints and limit memory use. If only one checkpoint exists and it needs to be removed, the ***merge*** function is invoked with a 0 for *handlePrevious*. In this case all state related to *handleRemove* has to be removed.

An additional three methods are provided to handle disk checkpoints. These functions work with directories, denoting where checkpoints should be written, and prefixes, which are unique identifiers within the directories. A boolean flag indicates whether or not only persistent state should be saved; persistent state is the state of devices that is non-volatile. Support for the exclusive saving of persistent state will be added in a future release (see Section 9). For more information about persistent state, see the *Simics User's Guide*.

The ***write_standalone*** method must save the complete state to disk, meaning that the produced checkpoint cannot depend on any other checkpoints, whereas the ***write*** method may produce iterative checkpoints that refer back to previous checkpoints. In this case, subsequent ***read*** invocations provide any dependencies in the list of directories.

All the methods described above should return a boolean to indicate if the operation was successful or not.

StateKeeperInterface is defined in
`systemc-checkpoint/state_keeper_interface.h`.

4.1 Maintaining Portability

When Simics, or the user in a stand-alone application, instructs the checkpoint library that a checkpoint should be saved to disk it provides state keepers with a directory and a unique prefix. This is sufficient information for the implementer to save state in any format, later to be retrieved using that same prefix and path.

However, pose that the directory containing the checkpoints is moved to another location. If the checkpoint uses absolute paths to files within the directory, these paths must be updated in order for the files to be found. To overcome this problem, it is advisable to refer to checkpoint files in a relative manner using the paths handed to the ***read*** function. For example, a list of filenames can be maintained with special syntax referring to entries in *dirs* given to ***read***.

It is recommended to maintain this list in the checkpoint. For example, this is useful if the list is appended with another entry, or if files are accessed in a lazy fashion. When a checkpoint is loaded, all serialization takes place before *StateKeeperInterface* implementations restore their state. Therefore, this list can be maintained using the regular checkpoint API.

Below is an example of how this relative syntax could work. The numbered segments surrounded by % are references to the list of paths provided to ***read***. Thus, %0% refers to the first path, %1%

to the second path, and so on. This syntax is chosen arbitrarily and can be represented in any way. With this syntax, consider that the ***write*** function is invoked three times with these paths:

a/x

a/y

a/z

Provided that the *StateKeeperInterface* implementation only writes a single file containing the state, it can maintain references to these files in the checkpoint:

%0%/state.diff

%1%/state.diff

%2%/state.diff

Consider now that the directory in which the checkpoints have been written is moved (a is replaced with b):

b/x

b/y

b/z

When the checkpoint is read, the *StateKeeperInterface* implementation can map the provided paths onto the list of references that it maintains. In this way, there is no need to update absolute paths in the checkpoint itself.

Chapter 5

Thread Stack Restore

The checkpoint library does not restore thread stacks directly. Instead, a two phase approach is used. First, the serialize function is called and state is restored. Then the thread is restarted and the thread stack is rebuilt. This second phase is allowed to use checkpointed state (as this is guaranteed to have been restored already) but is not alloweed to use any kind of thread synchronization, global state, or rely on a certain thread restart order. Thus, thread waits are only allowed locally in modules in order for the state to be checkpointable. For example, the state is not checkpointable if there is a wait on the other side of a ***b_transport*** function call.

The user of the library is responsible for identifying all locations where a thread can yield control to the kernel. The user is further responsible for making sure the thread will use checkpointed state to reach the correct yield point when the thread is restarted. Please see the example below to get an idea of how this should work in practice.

```
SC_MODULE(ThreadModule) {
    enum {
        WAIT_FOR_EVENT,
        WAIT_FOR_TIME,
        DONE
    };
    SC_CTOR(ThreadModule) : state_(WAIT_FOR_EVENT) {
        SC_THREAD(thread);
    }
    template <class Archive>
        void serialize(Archive& ar, const unsigned int version) {
        ar & SMD(state_);
    }
    void thread() {
        while (true) {
            switch (state_) {
            case WAIT_FOR_EVENT: {
                sc_core::wait(event_);
                state_ = WAIT_FOR_TIME;
            }
                break;
            case WAIT_FOR_TIME: {
                sc_core::wait(5, sc_core::SC_PS);
                state_ = DONE;
            }
                break;
            case DONE: {
                sc_core::wait();
            }
                break;
            }
        }
    }

    sc_core::sc_event event_;
    int state_;
};
```

Restoration of the kernel state includes a reset of static thread processes; dynamic thread processes are not supported. Resetting of the thread processes makes use of C++ exceptions. It is important that no **sc_unwind_exceptions** are caught. Because **sc_unwind_exception** inherits from **std::exception**, the same goes for the latter.

Chapter 6

Callbacks

For some devices it can be useful to perform some action before or after state is saved. For example, it can be useful to flush certain caches or prepare the execution of threads before they are started. To facilitate this, the checkpoint library provides a number of callbacks that can be implemented by the user. These callbacks are provided by *SerializeCallbackInterface*:

```
class SerializeCallbackInterface {
  public:
    enum State {
        State_Saving = 0x1,
        State_Loading,
        State_Persistent = 0x10,
        State_Saving_Persistent = State_Saving | State_Persistent,
        State_Loading_Persistent = State_Loading | State_Persistent
    };

    virtual ~SerializeCallbackInterface() {}

    /**
     * Invoked before any save/load/serialize function is called and before any
     * state keepers are invoked.
     */
    virtual void pre_serialize_callback(State state) = 0;

    /**
     * Invoked after any save/load/serialize function has been called, but
     * before any threads have been restarted.
     */
    virtual void post_serialize_callback(State state) = 0;
};
```

SystemC modules implementing this interface are invoked before and after modules are serialized. Furthermore, ***pre_serialize_callback*** is called before any state keepers are invoked and ***post_serialize_callback*** is called before any threads are started.

Chapter 7

DMI Handling

A model is responsible for its own memory management. Hence, memory addresses may change between different runs. Because of this, all memory addresses received via DMI are invalid as soon as a checkpoint has been restored. It is the responsibility of the model providing DMI addresses to invalidate those addresses at checkpoint load.

Chapter 8

Checkpoint Control

If the checkpoint library is used in a stand-alone application without Simics, the **CheckpointControl** class is used to dictate when checkpoints should be saved or loaded. The control object implements interfaces that combined provide the same methods as *StateKeeperInterface* (see 4):

```
class InMemoryStateInterface {
  public:
    virtual ~InMemoryStateInterface() {}
    virtual bool save(void **handle) = 0;
    virtual bool restore(void *handle) = 0;
    virtual bool merge(void *handlePrevious, void *handleRemove) = 0;
};

class OnDiskStateInterface {
  public:
    virtual ~OnDiskStateInterface() {}

    virtual bool write(const std::string &dir, bool persistent) = 0;

    virtual bool write_standalone(const std::string &dir, bool persistent) = 0;

    virtual bool read(const std::vector<std::string> &dirs,
                      bool persistent) = 0;
};
```

To use the checkpoint library in a stand-alone application, instantiate **CheckpointControl** after the SystemC hierarchy has been elaborated and use it to save and restore the state of your device. Using Simics there is no need to instantiate **CheckpointControl**; the saving and restoration of state is handled automatically alongside other state in Simics devices.

The **CheckpointControl** type can be found in `systemc-checkpoint/checkpoint_control.h`.

Chapter 9

Limitations

There are limitations to capability of SystemC checkpoints that must be considered. This section outlines all known limitations.

- Disk checkpoints are not yet platform independent; a checkpoint saved on a Windows platform cannot be restored on a Linux platform and vice versa
- The checkpoint library cannot automatically save and restore the stacks of threads, but it does provide the infrastructure to recreate them. Details about this infrastructure and the rules that must be adhered to when using it are outlined in Section 5
- Saving and restoring floating point values using the checkpoint library may lead to precision loss
- Persistent state is not yet supported
- The checkpoint library does not support dynamic processes
- The checkpoint library does not support dynamic events
- Checkpoint updaters, otherwise used by Simics to maintain backwards compatibility with old checkpoints, are not supported by the checkpoint library
- The only SystemC Kernel backend thread library that is supported is the QuickThreads library. Hence, Pthreads and fiber threads are currently not supported. This makes Windows a non-supported target
- Use of DMI require special handling. Please see Section 7 for details
- It is forbidden to serialize the same instance multiple times
- The SystemC specification does not make any guarantees about the order in which events are evaluated inside a delta cycle. The checkpoint library makes no effort to amend any indeterminism induced by this