GreenSocket - Overview

Robert Günzel (GreenSocs)

Abstract

GreenSocket is a TLM2.0 convenience socket that tries to improve the interoperability offered by TLM2.0 and that provides automatic memory management of transactions and extensions for the user.This document will explain how we believe GreenSocket is able to provide those features. For a detailed description of the API and examples, please refer to the User's guide.

Contents

1 Aiming on Improved Interoperability

This section tries to explain why we thin the basic interoperability offered by TLM2.0 can still be further improved and how we want to achieve this.

1.1 TLM2.0 extensions and their impact on interoperability

TLM2.0 allows for using two different types of extensions, ignorable ones and non-ignorable ones. Ignorable ones may be added to the extension array of the generic payload by initiators and in case the target does not care about this extension, the communication is not negatively effected.

Non-ignorable ones are added by the initiator as well, but this time the target has to make use of it otherwise the communication will fail or significantly misbehave.

When using a non-ignorable extensions the user has to define his own traits class on which the socket has to be templated, so that it cannot be bound to sockets that have been templated on a the TLM2.0 standard traits class.

Example

Assume the user will use an extension that defines the priority of a request. Moreover the initiator setting this extension relies on the target dealing with the extension. The extension will be added as a `tlm_extension` to the transaction, so the type of transaction to be used still will be `tlm_generic_payload`. The user will only use the generic protocol (GP) phases, so the used phase type will be `tlm_phase`. So basically the user is using the normal types, but as he insists that the the extension is understood, he defines:

```
struct priority_extended_gp{
  typedef tlm::tlm_generic_payload tlm_payload_type;
  typedef tlm::tlm_phase  tlm_phase_type;
};
```

And templates his sockets on that struct. By doing this only sockets that use the same type for their sockets can be bound. So the type implicitly states that the other understands the extension.

There are a some potential catches with this approach. First of all, the traits class does not identify the actual extensions used. So it adds a level of indirection to find out about a module. Take the example above. What does `priority_extended_gp` tell you? What extensions are used? What are the types of the content of the extensions, etc...

Secondly, we believe that you cannot always draw a straight line between ignorable and non-ignorable. Again, let's take the example above. It is pretty simple to think of initiators whose correct function relies on the fact that the priorities are correctly evaluated by the interconnect, while others might just use the priority as some kind of hint to the interconnect. From a TLM2.0 point of view you will have to template the former on the priority traits class (see example above), but what about the one that is using the priority just as a hint? If you template it on the special traits class, you cannot connect it to GP interconnects anymore, although it would be no problem from a functional point of view. But if you template it on the standard TLM2.0 traits class, it cannot be connected to an interconnect that is templated on the special class. So either way around you loose interoperability with a family of interconnects or targets with which you basically are interoperable.

Net result: The priority extension cannot be defined as ignorable or non-ignorable a priory. It depends on the use case.

Moreover, it is conceivable to have modules that make use of a certain extension if the communication partner can handle it. This is not possible when using the traits classes, as they either say that you have an extension and you insist on using it, or that you have an extension that you actually do not care very much about. To enable such a switch in the use of extensions, runtime information is needed.

Another disadvantage of the traits class approach is, that it is pretty hard to develop generic modules. Imagine a class that shall represent a pretty abstract module that is simply delaying a transaction, regardless of phase or extensions. It just implements the transport functions and adds a delay to them that's all. Now the socket of this module will have to be templated on a traits class, by that outlawing connecting modules that use extensions that are not ignorable from their point of view, although the module would correctly forwards priority extended transactions as well as normal GP transactions.

Of course, this problem could be solved by templating the generic module on the traits class it is about to use, but this will prohibit the binary distribution of such modules.

1.2 The GreenSocket approach to extensions

As described above sometimes the nature of extensions WRT interoperability can only be defined at runtime. So when using GreenSocket we propose to configure the socket before the simulation starts with the extensions that shall be used with this socket. This information will include whether an extensions can be ignored or whether it is mandatory or optional for or even rejected by the module.

The following table summarizes the different bindability levels of extensions:

Before simulation starts, two connected GreenSockets will inform each other what extensions they use and check whether their configuration matches. Afterwards they will inform the user of the socket about the set of optional extensions that were present in both configurations. By that the user may then pull some runtime switches to turn off the use of some extensions that are not used by its connected socket. Using this information, a module designed for a protocol far richer than the GP could even switch to a GP fallback mode if its GreenSocket tells it that the other socket is mere GP.

Additionally, a GreenSocket can be told to be generic, by simple treating each extension as optional. That means the socket will simply adopt each mandatory extension used by the other socket, so that they are interoperable. This allows e.g. for the creation of such abstract delay modules as mentioned in the previous section.

Finally, the GreenSocket configuration will also include the phases used by a module, so that this information can be taken into account when interoperability is checked.

The back bone of this approach is that there is a common understanding of extensions. There is no gain in this approach if e.g. two sockets use plain GP with a priority extensions, but both defining their own extension class for that. Even if both mark the extension as ignorable, GreenSocket could not identify that they both have the same semantic.

For that we need a centralized and growing extensions library, from which protocol and module designer can pick the ones they need. So if designer A wants a protocol that is GP+priority and designer B wants a protocol that is GP+priority both should pick the same extension to express the priority, and they are interoperable.

Level	Description	Bindability Rules
Mandatory	The extension is vital for the correct operation of the module.	If the other socket is rejecting the extension, the bind attempt will fail. If the other socket treats the extension as optional or mandatory the bind attempt will succeed and the extension will become mandatory for both ends.
Optional	The module can operate correctly with or without the extension, but needs to know whether it is understood by the other side or not.	The bind attempt will always succeed. If the other socket is rejecting the extension, the extension will become rejected for both ends. If the other socket mandates the use of the extension, it will become mandatory for both ends. If the other socket also treats the extension as optional, the extension stays optional and the module will decide/has to detect whether to use the extension or not (also see a note section **Data extensions!**).
Rejected	The module will misbehave when dealing with the extension.	If the other socket mandates the use of the extension the bind attempt will fail. If the other socket treats the extension as optional the bind attempt will succeed and the extension will become rejected for both ends. If both ends reject the extension, the bind attempt will succeed and the extension will remain rejected.
TLM-2.0 ignorable	The module emits/receives this extension in a way as defined as 'ignorable' by the TLM-2.0 manual	The extension is not part of the interface (and bindability checks), since it is truly ignorable.

2 Memory management (MM) of extensions

GreenSocket will provide memory management of extensions for the user. However, GreenSocket does not know the type of extensions used at compile time and so an extension pool would have to operator with virtual functions. This of course is a definite no-go for pooling, as pools have to be as fast as possible. Basically, efficient pooling of types not know to the pool owner is not possible. So we have to find a way to avoid pooling. To this end let's have a look at the nature of extensions. Afterwards we'll see that there is no need for extension pooling.

2.1 Extension use cases and MM

1. **Guard extensions**

   There are extensions whose mere existence is the only information they carry, they do not have members. That means, they are basically like a boolean variable. If they are part of the transaction they represent a boolean *true* while their absence represents *false*. An example would be a extension called `cachable`. If it is there the transaction is cachable, if it is not there the transaction is not cachable.

   If transactions are reused it is important that such guards are removed when the transaction is done, otherwise when a transaction is reused and travels another path some module could see the extension and behave as if someone added the extensions which would be wrong, as no one did this time.

2. **Data extensions**

   We refer to such extension as pseudo transaction members. Such extension always carry data members. Once added they stay in the transaction (they are sticky from the TLM2.0 point of view). An example may be an extension called `priority` containing an integer which is only valid if a phase called `BEGIN_REQ_WITH_PRIO` is send (it would never be valid within b_transport). Whoever initiates that phase (an initiator or an interconnect) will just set the value of the extension. If it is already in the transaction the content will just be overridden, otherwise it will be added. Such extensions do not have to be removed from transactions, as they do not mean any harm if the transaction is reused and travels a path without the special `BEGIN_REQ_WITH_PRIO` phase. If it will travel a path with `BEGIN_REQ_WITH_PRIO` when it is reused, the one starting the phase will override the existing value and again there will be no harm to the correct function.

   Important Note:

If two connected GreenSockets treat a data extension as optional, after binding (see table above) it will stay optional. However, there is no chance of determining whether a data extension is valid or not, if there isn't a special mechanism for that (like the phase based approach as described above). So without such a mechanism, when receiving a transaction a module cannot know whether to use the extension or not. So when defining a data extension, make sure to either forbid treating it as optional, provide a mechanism to detect the validity, or specify a default (use/no use) in case the extension remained optional.

3. **Guarded data extensions**

   Finally, there are extensions that carry data and when present means the data inside is valid. An example could be an extension called `priority` containing an integer which is valid during the GP phase `BEGIN_REQ`. Whoever, gets `BEGIN_REQ` and knows about `priority` would test if it is there, and if it is check the value inside the extension. Just like simple guards, such extensions will have to be removed to avoid misbehavior when the transaction travels another path when being reused.

Now what about there memory management? Well, guard extension can be allocated statically and each transaction that has the guard enabled could point to the same extension, as the extension itself is not of interest. The information is just encoded in the fact that the entry in the `tlm_array` of the transaction is not `NULL`. When cloned, the extension will just return a pointer to itself. So there is no need for pooling guard extensions.

Data extensions are added sticky. Assuming transaction reuse, they are just `new`-ed when a transaction is not carrying it, but needs to. After some simulation runtime, the probability that a transaction carries the needed data extension is pretty high. So again there is no need to pool those extensions.

Finally what about guarded data extensions? Those shall be split into two extensions: one guard extension and one data extensions. The rule is that the data can only be considered valid, if the guard is present. Since we do not need to pool guard extensions or data extensions, we will not need to pool guarded data extensions as well.

3 GreenSocket services

This section will now illustrate what services GreenSocket will provide. It describes the most important services how they work.

3.1 Configuration

After creation a GreenSocket will virgin, i.e. not configured. It shall be an error if a GreenSocket did not get a valid configuration until the start of the simulation.

To configure GreenSocket WRT the used extensions and phases, one will instantiate a special GreenSocket configuration object, and tell it which extensions/phases are mandatory, optional or rejected. By default the configuration will treat unknown extensions (=extensions that are not part of the configuration) as rejected, but the user may change that to treat unknown extensions as optional. Afterwards, the configuration shall be provided to the socket. After the bindability check has been performed (before_end_of_elaboration) the GreenSocket will carry the resulting configuration (=the largest possible intersection of the configuration of the two connected sockets)

To create a generic GreenSocket that can adopt any configuration, just tell its config to treat unknown extensions as optional and nothing else. The resulting configuration will then contain all mandatory extension of the other socket as mandatory and all optional extensions as optional.

3.2 Bindability check

Before end of elaboration GreenSocket will determine, whether its connected socket is actually bindable or not. During this the socket will provide some calls into a derived socket to enable advanced bindability rules.

A plain GreenSocket will therefore not provide such callbacks into the owner, but at start_of_simulation the owner can safely check the resulting configuration of its GreenSocket.

The basic bindability check is as follows:

1. The socket checks whether it is currently in the process of checking its bindability. If that is the case it will abort this recursive check.

2. The socket checks whether it has a valid configuration. If not it assumes it will get one out of a before_end_of_elaboration callback or as a result of another socket that just got its configuration. Consequently, it will abort the check for now, and will redo it as soon as it gets a configuration.

3. The socket checks if its connected socket is a GreenSocket. If not GreenSocket assumes to be connected to a GP compliant socket. It assembles a GP compliant configuration. If it is connected to another GreenSocket, it will get the configuration from that one. That 'getting' of the configuration will start the other sockets bindability check (Note that this will lead to getting the config from the current socket but that is guarded by the very first check point).

4. The configuration created/acquired in the previous step is checked against the socket's configuration. If the binding fails, the simulation will abort reporting which binding failed for what reason. If the binding succeeds, the resulting configuration will become the socket's configuration.

5. The socket calls `bound_to` on itself. This call is virtual and may be overridden by derived sockets, so that they can react on the resulting configuration. The call provides a pointer to the bound socket, a string that identifies the type of the socket, and the index of the binding of the socket (for simple sockets that can only be bound once, the index will always b 0).

3.3 Transaction memory management

GreenSocket (the initiator and bidirectional variants only) contains a pool of transactions. The transactions carry all simple data extensions and the data parts of guarded data extensions that were configured to be used by this socket, when they come out of the pool. Since the pool is build before simulation starts, `new`-ing of extensions will never take place, when an initiator access the sticky extensions he configured its socket to use.

When a transaction is taken from the pool, and after some time is not of use any more, it must actively be put back into the pool. If one wants to reuse directly, he must check the reference count of the transaction and may only reuse if it is equal to one, if not he must return it to the pool and get a new one from the pool.

Still the user can choose to use his own external pool, or heap or stack allocate transactions as needed. GreenSocket does depend on getting a transaction out of its own pool (although this might improve performance).

3.4 GVI Extension access

GreenSocket does also offer simple functions to access extensions, although they still may be accessed manually through the transaction. Using the functions provided by GreenSocket the user can be sure that the semantics of guard, data and guarded data extensions are respected.

Moreover, via those access functions GreenSocket makes sure that an extension is never added twice to a transaction as an auto_extension (see the TLM-2 manual about set_auto_extension). Inexperienced users might do this when directly accessing the TLM-2 generic payload, which (in the worst case) will lead to a buffer overflow in an generic payload internal structure.

The access function will allow to

1. Get extensions from a transaction, just like the TLM-2 `get_extension` function. However the functions will make sure that, when accessing sticky extensions (data extensions or the data parts of guarded data extensions) you always get a valid extension handle, and you do not need to protect yourself against NULL pointers or some such.

2. Validating extensions in a transaction. That means activating the guard of a guard only or a guarded data extension. The functions hide as many memory management as possible from the user, like using `set_auto_extension` or `set_extension`, and like preventing buffer overflows (see above).

   Additionally, the validation functions return information, whether the user will have to remove the extensions manually (if there was no memory manager present) or if that will happen automatically. This allows for writing code that can easily operate in both memory managed and non memory managed environments.

3. Invalidating extensions in a transaction. That means deactivating the guard of guard only or a guarded data extension. This will have an immediate effect, since you either call that function because you are in

a non memory managed world and you have to remove extensions manually, or because you wanna change the state of a transaction **now** and not at a later time.

This simple get-validate-invalidate (GVI) scheme is as powerful as the set-set_auto-get-clear-release scheme, but is by far simpler.

Note: The GVI scheme is built on top of the set-set_auto-get-clear-release scheme. It does not modify or disrespect the TLM-2 standard.

3.4.1 Usage with blocking and non blocking transport

Just as normal for TLM2.0 if the user validates an extension before calling `b_transport` he has to invalidate it afterwards. In case of `nb_transport` the user will just validate.

There is a corner case, when a transaction comes into a bridge with an extension validated, and it shall go downstream without the extension (or vice versa). If the mutability rule for the extension allows it to be removed (so it is only valid with the very first phase you send), you can just invalidate the extension. If the mutability rules do not allow that, a transaction copy will have to take place.

3.5 Communication interface

The communication interface is equivalent to the passthrough sockets of the TLM2.0 kit. You can register callbacks for any of the TLM2.0 interface methods. No automatic blocking to non-blocking (and vice versa) conversion is performed. This task shall be left to sockets that are build on top of GreenSocket.