Advances in Bus Simulation, Software Debugging and SMP2 integration in the T-EMU Emulator

Dr. Mattias Holm[1], Joshua Whitty[2], Alberto Ferrazzi[2], and Mathilde Maury[2]

[1]*Terma B.V.*, Shuttersveld 9, 2316 XG Leiden, Netherlands, E-mail: maho@terma.com
[2]*Terma GmbH*, Europaplatz 5, 64293 Darmstadt, Germany, E-mail: jowh@terma.com, ablf@terma.com, mmau@terma.com

March 3, 2017

Abstract

Emulators and virtual platforms are essential in the development of any type of embedded platform. T-EMU is the Terma emulator framework supporting full system simulation. For full system simulation, not only does the emulator need to support CPUs and memories, but also several bus models to simplify the construction of larger systems (and enable plug and play between models). T-EMU is an evolving system and the latest release brings out support for several bus models (e.g. 1553, CAN and SpaceWire), source level software debugging and improvements for integration with SMP2 simulators.

1 Introduction

T-EMU is a full system simulation framework [3], providing several high performance[2] instruction set simulation models (emulators) of different SPARCv8 (and soon, also ARM) microprocessors. Current CPU models include the ERC32, LEON2, LEON3 and LEON4 processors. A common problem for users of emulators of such systems is that the systems are usually ASICs or SoCs with additional on-chip devices such as memory controllers, caches, UARTs and additional more complex busses such as MIL-STD-1553, SpaceWire and Ethernet. For the on-board software developer, it is a major problem when models for these devices are missing and they either have to wait for getting a full SVF or, write their own MMIO models for at least rudimentary simulating the hardware. With the current *commoditalisation* of hardware components, it makes it easier to provide a set of "standard device models" with the emulator that can be used by the user directly, without having to procure an additional simulator infrastructure.

T-EMU solves this by providing a set of fully virtual bus models (Serial, CAN, 1553, SpW and Signal) that works out of the box. The end user only need to provide the "remote terminals". The MMIO based controllers and the bus models are already provided by T-EMU. Consequently a large barrier to writing unit tests of e.g. software interacting with hardware directly has been removed and tests that previously was deferred to system level testing can now be done at a unit level.

Another problem often facing on-board software developers is the relatively poor state of software debuggers. Currently, the standard approach is to attach the GDB debugger via the Remote Serial Protocol (RSP) directly to either the emulator or a debugging stub on the actual hardware. There are several problems with this, one is that GDB is intrusive, even when the user would expect it to not be. It is also very inefficient for certain operations. When setting a watch point on a non-word size item, GDB will treat it as a software watch-point, which essentially force the debugger to resort to single-stepping the program and checking the watched data between every step. We have observed execution performance dropping from 100 MIPS to less than 1 MIPS due to this.

Emulators are often integrated in larger simulators, often based on the SMP2 standard[1]. Terma has prototyped an extension that simplifies event scheduling when an emulator is in the loop.

2 Microprocessors

The main improvement on the microprocessor side has been the large improvement in performance, increasing from 90 MIPS[2] to 110 MIPS. An improvement of 22%, increasing the performance lead over TSIM to 83% and the ESOC

emulator to 70% without an MMU and 340% with an MMU[1].

In addition to the performance improvements, support for the ARM architecture is about to be rolled out. This includes the ARMv7-R sub-architecture. ARM-support serves to make T-EMU more accessible to users outside the space sector, where SPARC processors are very rare. As T-EMU was designed from the start to support multiple architectures[3], the new ARMv7 support was relatively straight forward to add using the LLVM based tools that are used by Terma to implement CPU models.

3 Components

A fundamental advancement is the support of components in T-EMU. Components greatly simplifies the construction of complex systems. In T-EMU 2.1 and earlier, to create a system, one would execute one of the system configuration files using a command such as "exec leon2.temu", these files are relatively simple to create and is essentially just a list of commands to the T-EMU command line interface *CLI*. However, the scripts needed to create uniquely named objects, and for larger systems, these scrips tended to be quite long. The main problem was when the user wanted to create multiple systems, e.g. create two LEON2 processors and then connect these via a simulated bus. The user needed to duplicate the script and ensure that the two scripts used unique names.

The component support simplifies this by firstly providing a standard "component class", which tracks a set of common properties for components, such as a list of processors, objects and sub-components. In addition, an API was added that ensures unique naming of objects within a component, and the automatic assignment of a time source to every object. The object creation function for components also takes the object name as a printf style strings enabling simple application of loops to create multiple object instances with numerical identifiers.

Components also support the ability to delegate internal interfaces and properties as if they would belong to the component itself[2]. For example, a LEON2 system would normally consist of a CPU, the LEON2 on-chip devices, RAM and ROM. With the LEON2 component, the component embeds all those objects, and it delegates the UARTs from the on-chip devices. Thus the user can say:

```
  t-emu> connect a=l2comp0.uart_a b=tty:UartIface
# instead of:
  t-emu> connect a=l2soc0.uart0 b=tty:UartIface
```

As can be seen, the "l2comp0" has a delegated property ("uart_a") which is actually an alias for "l2soc0.uart0". While the example is trivial, it is easy to imagine a LEON processor with multiple UART models, each providing their own UART interface.

In addition, the components form natural machines and it is the intention that the components will in due time replace the machine objects used for multi-processor scheduling.

4 Device and Bus Models

A major new feature in T-EMU is the introduction of additional built in bus models. Bus models are transactional models that attempts to abstract a bus as much as is possible while being invisible for the software running in the emulator. For most busses, this abstraction is the frame or packet[3] level.

There are two primary aspects to modelling busses, the bus model itself and the controller models. The bus model include the interfaces and classes needed to simulate the data transfers. The controllers are typically memory mapped I/O devices that implement interface between the on-board software and the bus model. The bus models facilitate communication between different controllers.

As T-EMU is a general purpose product that should be usable out of the box, it needs to be standards independent. This way T-EMU can work with SMP2, System-C, and other simulation standards, without depending on any of them. For SMP2 side there are already several bus models provided for via auxiliary standards such as ISIS, REFA, and the ATB bus models.

To interface T-EMU with *foreign* bus models it is possible to implement either an adapter model that converts one API to another or to replace the T-EMU controller model with a custom one that interface directly with the bus model

[1] There have been discussions on improvements in performance in the ESOC emulator, but we have not seen any official figures for this yet

[2] Conceptually similar to delegation in UML components

[3] SpaceWire

standard in question. For T-EMU, the main difference is which interface is used to interact with the other busses, i.e. via the memory transaction interface or via the T-EMU bus interfaces.

There are several advantages of integrating bus models directly in the emulator. Firstly, T-EMU is relatively easy to learn and it is very simple to create for example simulated CAN terminal by following one of the examples provided in the documentation. Secondly, a major use case is the unit testing of hardware drivers, by providing the bus controllers with the emulator and a simple bus API, it is easy to create models that attach to the bus that can do various types of unit test operations (e.g. acting as a mock terminal, or asserting on expected frame data). Thirdly, with bundled bus models, the emulator eliminates the main technical need for a simulator built by a third party to re-implement the on-chip devices in different ASIC/SoCs. Instead, a more simple bus bridge can be created. That said, there is nothing that prevents the implementation or integration of replacements for either the bus models or the controller models.

There are two types of busses: point to point buses, and multi-node busses. Point to point busses such as serial and SpaceWire do not need a specific bus model, but only interfaces for sending data from one device to another one. Multi-node busses such as CAN, 1553 and Ethernet do on the other hand need a bus model object to facilitate distribution of messages.

4.1 CAN

The CAN bus is a multi-node bus. T-EMU provides a model of the CAN_OC controller from OpenCores (with GRLIB specific extensions) and a CAN bus model.

The CAN bus model itself implements an interface and it is possible for the user to replace the bundled bus model with their own bus model. This is useful in the cases where hardware based message filtering is used. An intelligent bus model could route messages to the relevant devices instead of broadcasting them. Thus, it is possible to improve performance of large CAN networks considerably. However, it is recommended to start with the bundled simple bus model before embarking on writing your own.

A remote terminal can easily be implemented using the CAN device interface:

```
typedef struct {
  uint8_t Data[8];
  uint32_t Flags;
  uint8_t Length;
  uint8_t Error;
} temu_CanFrame;

typedef struct {
  void(*connected)(void *Dev, temu_CanBusIfaceRef Bus)
  void(*disconnected)(void *Dev)
  void(*receive)(void *Dev, temu_CanFrame *Frame)
} temu_CanDevIface;

struct temu_CanBusIface {
  void (*connect)(void *Bus, temu_CanDevIfaceRef Dev);
  void (*disconnect)(void *Bus, temu_CanDevIfaceRef Dev);
  void (*send)(void *Bus, void *Sender, temu_CanFrame *Frame);
  void (*enableSendEvents)(void *Bus);
  void (*disableSendEvents)(void *Bus);
  void (*reportStats)(void *Bus);
  void (*setFilter)(void *Bus, temu_CanDevIfaceRef Dev, int FilterID,
                    uint32_t Mask, uint32_t Code);
};
```

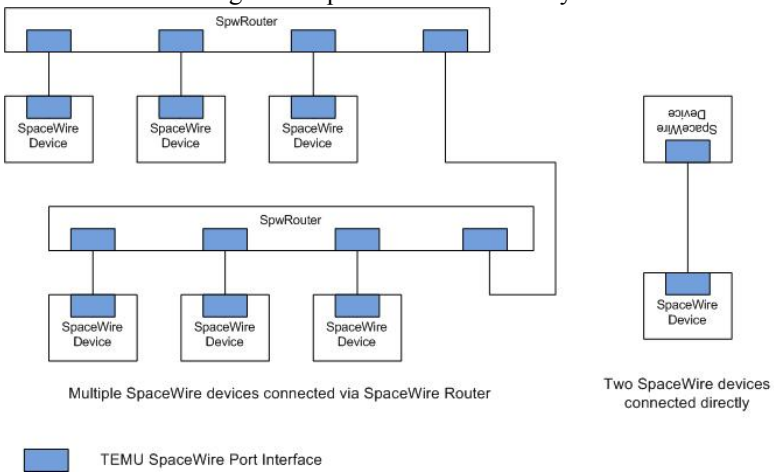
The connected and disconnected function in the device interface is called whenever the user connects or disconnects the device to the CAN bus object (i.e. a virtual cable insert / removal). The receive function is called whenever a frame is sent on the bus which needs to be delivered to your device.

4.2 SpaceWire

T-EMU 2.2 introduces support for SpaceWire network simulation. Two basic models are currently available to interconnect several SoC configurations via a virtual SpaceWire network and if that is not enough, the T-EMU API allows the user to easily create a custom model that can be connected to the virtual SpW network or as a bridge that connects the emulator to real hardware for hardware in the loop simulations.

The GRSPW2 model is available to connect the SoC to the SpaceWire network. Since SpaceWire is a point to point network a router is required to connect multiple ends together and T-EMU comes with a simple generic SpaceWire Router model. The router can be used to assemble complex networks as represented in Figure 1.

Figure 1: SpaceWire Connectivity



Note that the generic router is configurable only via the emulator's user interface and APIs, it is not configurable from embedded software running inside the emulator, for specific routers with e.g. RMAP configuration options, additional models must be implemented.

The concept of interface in T-EMU (which has both a name and a type) allows an object to have several interfaces of the same type provided that they are named differently (note that this differ from how languages such as Java and C++ provides interfaces). Interfaces have been used to model the SpaceWire ports so that a model can provide several SpaceWire ports. Two SpaceWire models can be hot plugged/unplugged using T-EMU commands that connect/disconnect the two related SpaceWire port interfaces. This capability to dynamically construct networks at run-time, after the simulated systems has been assembled and connected, is very important for T-EMU.

In order to connect a model to a virtual SpaceWire network, the model needs to implement the SpaceWire port interface for each desired port. The key part of it is to implement two methods: one for handling a received message, the other to handle a change into the other end SpaceWire state machine. The behaviour for the latter is device specific and therefore has to be supplied by the model developer.

T-EMU also provides an API for decoding RMAP packets, this was added as RMAP is a fundamental SpW protocol, and it made sense to simplify this activity for the end user.

As with all bus models in T-EMU, they model what is visible to the on-board software, and are designed for high performance. For SpaceWire, this implies that the bus model is packet based and that control characters are abstracted away using the link state interface. A single function call is all that is needed to transfer a whole packet to another device. A problem with that is that as packets are of arbitrary length, and they can be quite large. For larger networks where it is important to simulate network latency, there is a problem with how to allocate, dispose and pass on the packet to other nodes, and especially if a router needs to send a packet out on multiple ports. The problem here is essentially about buffer ownership and unnecessary copies.

To simplify this, instead of passing around a byte array in the SpW interface, the user passes a T-EMU buffer object. The buffer type has a copy on write semantic (COW) and can be cloned without having to allocate additional memory (copies are only made if there are multiple writers to the data). The buffers can also shrink from the head or tail without having to reallocate data or effect any clones of the buffer. This means that the SpW wormhole routing becomes very efficient as the first character can quickly be removed, even though the buffers are conceptually copied between devices and routers.

4.3 MIL-STD-1553

The T-EMU MIL-STD-1553 model simplifies development of bus controllers and remote terminals by handling a couple of features, such as message routing, bus phases, statistics reporting. By handling those at bus level, we reduce the need for protocol related processing on the terminals side, as the bus model does the work of keeping the bus into a healthy state. When a transmission happens on the bus that does not comply to the 1553 protocol, the operation is aborted and logged, and the bus automatically recovers. This way, messages are quickly transmitted by direct addressing, and are sent only to the relevant devices improving performance. A remote terminal can be created by implementing three functions: A connect and a disconnect call, that will automatically be called upon connection/disconnection on the bus of the remote terminal and a receive call which transmits any messages either specifically addressed to the remote terminal, or broadcasted.

The 1553 bus simulation is similar to the CAN model (see Section 4.1), except that the 1553 bus is phased. This comes from the ability to initiate RT-to-RT transactions on the bus. In a similar fashion, the user may implement their own bus using the same 1553 bus interface to be able to connect with any remote terminal already written for the bundled bus model. This makes it easy to create bus bridges for interfacing with other 1553 bus models.

4.4 Signal

The signal interface was added to T-EMU to simplify the implementation of GPIO devices. The signal interface is very simple, and requires two functions to be implemented:

```
struct {
  void (*raise)(void *Obj);
  void (*lower)(void *Obj);
} temu_SignalIface;
```

The two functions raise and lowers the signal. Multiple signals can be handled using interface arrays (or differently named interfaces). This differ from many other interface based standards which requires the user to add an additional class or object for every additional pin in this case.

4.5 Ethernet

The Ethernet interfaces are as for all the other bus models constructed around the software visible aspects of the protocol. That means that there are interfaces for Media Access Controllers (MAC), Physical controllers (PHY) and the Media Independent Interface (MII) which is used to configure PHYs from MACs, and an interface (and a model) for the media itself (as Ethernet is a multi-node bus). The Ethernet model is currently experimental.

5 Software Debugging

DWARF is a backronym of *Debugging With Arbitrary Records Format*, it was initially a pun on the ELF executable format. DWARF is a standard for how to provide debugging information in executables and libraries that can be read by debuggers. For example, it provides information on how to map instruction addresses to source lines (and the opposite direction), and it provides information on how to compute the location of named variables. For example, it is very common that variables move around, i.e. depending on the context they may be in memory or in a processor register. DWARF provides mechanisms that can encode this location information.

When loading an ELF image into the emulator, the DWARF info in it is parsed into a debugging context. It is possible to have several loaded debugging contexts (e.g. one for the boot software and one for the OBSW image) at the same time.

Although, while not 100% complete at the time of writing, the DWARF support enables source level debugging directly from the emulator's command line interface and via APIs that are exposed to the end user. This allows the user to avoid the use of GDB-RSP based interaction with third party debuggers (although the RSP protocol is still supported).

It is possible to step through source lines, instructions, set source level breakpoints and many other common debugger capabilities. Commands include:

```
t-emu> load file=obsw.elf
t-emu> disassemble func=foo
...
t-emu> break func=foo
t-emu> break line=foo.c:123
```

```
t-emu> list func=foo
   void foo() {
>*    int a = 42;
   }
t-emu> step-line
t-emu> show var=a
42
```

An interesting capability with the T-EMU debugging support is that there an API is available for interfacing with it. Consequently, it is possible to query the API for source locations, data locations, suitable breakpoint locations etc. These APIs could be used to provide source level debugging capabilities in existing simulators such as e.g. Simsat.

6 Asynchronous Events

A common problem for any simulator or emulator is how to handle integration with external hardware via for example TCP/IP or a SpaceWire card. On UNIX-like systems, the access is done via a socket or file descriptor. A standard solution for this is to do non-blocking polls on the descriptor, however, there are polling free solutions possible that eliminates the need to post polling events in the emulator event queue.

For this reason, T-EMU now have support for high performance asynchronous events, based on epoll on Linux and kqueue on BSD derived operating systems (including macOS). The capability allows the user to register a callback to be executed when there is data in the file/socket and the event is executed during the normal event execution in the emulator (i.e. after an instruction has finished executing). The result of this is that the entire T-EMU API that is allowed to be called from a normal T-EMU event handler, is now available for use in the asynchronous event handler. Note that the API has now replaced all polling based code that was used in some bundled T-EMU models.

The API for the new async event support is very simple:

```
int temu_asyncTimerAdd(void *Q, double T, unsigned Flags,
                       void(*CB)(void *), void *Data);
void temu_asyncTimerRemove (int Fd);
int temu_asyncSocketAdd(void *Q, int Sock, unsigned Flags,
                        void(*CB)(void *), void *Data);
void temu_asyncSocketRemove(int Fd, unsigned Flags);
void temu_eventPostAsync(void *Q, temu_ThreadSafeCb CB,
                         void *Data, temu_SyncEvent Sync);
```

The async timer functions allows for the posting of events that will be triggered within a certain number of wall clock seconds. These events are delivered whenever the emulator is running, and executed in the emulators normal event queue.

The async socket functions registers a callback to be called whenever there is data available for reading in a socket (or any other file descriptor, including pipes, on UNIX like systems).

7 SMP2 Compatibility

Experience from previous simulators has identified the need for a better scheduling approach when running models and an emulator synchronously in a simulation. The problem is that the emulator is scheduled from the Simulation Infrastructure scheduler along with other model events. This leads to four problems: 1. Events may need to wait to be executed while the emulator is running. 2. Events within the emulator are scheduled on its scheduler, therefore two schedulers exist and difficulties arise in synchronising simulation time between them. 3. Debugging of OBSW is very difficult to implement while maintaining non-intrusive and fully deterministic qualities that should be default behaviour in a simulator (this stem from there now being two schedulers and two systems wanting to control the advancement of time, the debugger and the simulator). 4. Emulator / OBSW cannot immediately detect interrupts raised by models.

Different solutions have attempted to solve each of these issues, but not all are optimal. For example, to ensure that the OBSW detects interrupts raised by models model developers have forced the emulator to run for a short periods of time after each model event has been executed with Pre or Post event notifications from the Simulation Infrastructure scheduler - but this solution is Simulation Infrastructure dependent. Another solution has been to put all events of models that interact with the processor on the emulator scheduler - but this solution leads to a hybrid system of two schedulers executing models, and these events are only executed while the emulator is running.

Another solution to ensure that events get executed at the exact time specified is to implement an SMP2 Global Interface that allows the Simulation Infrastructure scheduler to stop the execution of the emulator / OBSW to allow an event on the scheduler to execute. This has been implemented on previous SVFs but is found also not to be so optimal since the call can come from a separate thread than the one the emulator is executing from and the emulator cannot stop exactly when instructed to, but it needs to wait until it has completed the currently executing instruction window (which is usually in the range of 10000 instructions).

For debugging OBSW running in a simulator with multiple schedulers, many options exist, one approach has been to implement the GDB RSP protocol in an SMP2 model, however, again the scheduling of the emulator which must be controllable from the RSP server and the simulator is again a difficult to solve problem. In addition, emulators sometimes provide their own debugging APIs, and the use of RSP is not actually ideal for use in a simulator.

With the advent of multi-core processors (and in normal multi-processor systems as well), it will also not be possible to agree of a single time base between processors. Emulators with multi-core support splits up their own event schedulers into clock domains, where there are typically one domain per CPU core plus an additional “synchronised” domain. The different CPUs synchronise their schedules at regular intervals. Thus, to post in the synchronised domain typically has requires that the event is posted in the next synchronisation interval/quanta or later.

Terma’s solution is to have ONE single scheduler where all events are executed from within the emulator to best solve the issues identified.

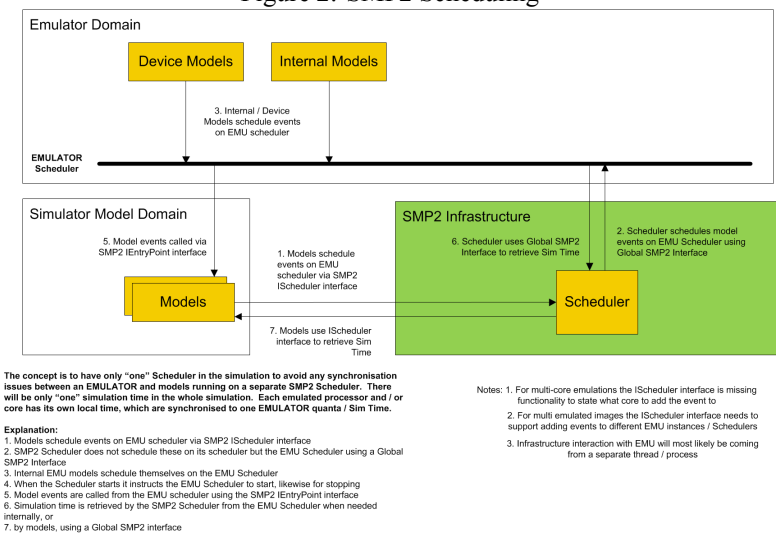
- All events can execute when at time specified
- OBSW can detect interrupts straight away from models
- Multi-core and multi-processor emulations ensure events are scheduled on the correct CPU (or clock domain)
- It becomes trivial to single step the CPU(s).

To test this approach a prototype using SIMSAT 4 was implemented where a SMP2 Global Interface was created to support an external scheduler (no changes to SMP2 standard were done). The SMP2 Global Interface IExternalScheduler interface was created to support:

- Start
- Stop
- GetSimulationTime
- AddSimulationTimeEvent (specifying a CPU ID for multi-core/-processor emulation)
- ModifySimulationTimeEvent

This interface is defined globally so both the simulation models and the Simulation Infrastructure can access it and is registered with the SMP2 Global Service Manager by the external scheduler (Emulator) in the simulation so that the Simulation Infrastructure can access it. Figure 2 illustrates how a single external scheduler can be supported in a Simulation Infrastructure and models alike in a simulation.

Figure 2: SMP2 Scheduling



One single scheduler does solve the issues that have been encountered when running models and an emulator synchronously in a simulation. There is no need for multiple calls between the Simulation Infrastructure scheduler and the emulator to determine how long the emulator can execute for determined by the time of the next event on the scheduler, and no need for the emulator to update the Simulation Infrastructure's simulation time. There is also no need for any time synchronisation and events can execute when specified and the emulator is not required to be stopped or wait for other events to finish. Another advantage is that the OBSW being emulated can immediately detect interrupts raised by models and not have to wait for the next time slice that the emulator is to be executed.

Note that the SMP2 standard does not currently support the external scheduler concept proposed by Terma. The prototyped example using SIMSAT4 showed that an external scheduler can be supported via SMP2 mechanisms e.g. using the SMP2 Global Interface Service, but this is not portable across different SMP2 Simulation Infrastructures since the interface is not part of the SMP2 standard.

8 Conclusions and Future Work

T-EMU has advanced quite considerably since being introduced to the world[3]. With the introduction of ARM support, components, many bus models and software debugging support. Further, Terma has solved the scheduling problems that often show up when integrating emulators inside existing simulators. The solution to the scheduling problem is flexible enough that it scales also to multi-core / -processor systems.

The near future will see the stabilisation of the DWARF support in the emulator. The next steps for T-EMU is to integrate support for dynamic binary translation, which should improve performance from the current 110 MIPS to over 400 MIPS. new models. For the 1553 bus, the Cobham Gaisler GR1553B is currently being modelled and it will feature BC and RT capabilities, send lists and interrupts.

T-EMU is also expected to become more aware of SMP2 and support the integration with SMP2 by helping out with the automatic wrapping of T-EMU interfaces (e.g. bus interfaces) as SMP2 bus adapter models.

A future direction is the T-EMU GUI or MMI. Such a tool could provide an integrated OBSW and simulation model debugging solution with visualisation of simulation state and OBSW state (e.g. partition, process and thread state, and ability to set breakpoints in the OBSW).

References

[1] ECSS. ECSS-E-TM-40-07 (SMP2 draft standard). `http://ecss.nl/`.

[2] Mattias Holm. Emulator Performance Study. In *Simulation & EGSE Facilities for Space Programmes*, 2015.

[3] Mattias Holm. The Terma Emulator Evolution. In *Simulation & EGSE Facilities for Space Programmes*, 2015.