Muen - Toolchain

Reto Buerki Adrian-Ken Rueegsegger

February 7, 2017

無縁

University of Applied Sciences Rapperswil (HSR), Switzerland

Contents

List of Figures

1 Introduction

This document describes the process of configuring and building a component-based system running on the Muen Separation Kernel (SK).

2 Policy

A policy is a description of a component-based system running on top of the Muen Separation Kernel. It defines what hardware resources are present, how many active components (called subjects) the system is composed of, how they interact and which system resources they are allowed to access. The following properties are specified by the policy:

- Configuration values
- Hardware resources
- Platform descrption
- Kernel diagnostics device
- Physical memory regions
- Device domains
- Events
- Communication channels
- Components
- Subjects
- Scheduling plans

The policy serves as a static description of a Muen system. Since all aspects of the system are fixed at integration time the policy can be validated prior to execution, see also section 3.2.

2.1 Content

This section presents the different parts of a system policy and gives an overview what each section contains.

2.1.1 Configuration Values

This section of the policy is used to specify configuration values to parameterize a system. It allows to declare boolean, string and integer values, e.g. `<boolean name="iommu_enabled" value="true"/>`.

2.1.2 Hardware Resources

Systems running the Muen SK perform static resource allocation at integration time. This means that all available hardware resources of a target machine must be defined in the system policy in order for these resources to be allocated to subjects.

Data required by a hardware description includes the amount of available physical memory blocks including reserved memory regions (RMRR), the number of logical CPUs and hardware device resources.

The Muen toolchain provides a handy tool to automate the cumbersome process of gathering hardware resource data, see section 5.3.

2.1.3 Platform description

To enable an uniform view of the hardware resources across different physical machines from the system integrators perspective, the platform description layer is interposed between the hardware resource description and the rest of the system policy. This allows to build a Muen system for different physical target machines using the same system policy.

2.1.4 Kernel diagnostics device

The Muen SK can be instructed to output debugging information during runtime. The kernel diagnostics device specifies which I/O device the kernel is to use for this purpose.

2.1.5 Physical Memory Regions

This part of the policy specifies the physical memory layout of the system. Memory regions are defined by their size, caching, type and are placed by specifying a physical address. Additionally the content of the region can be declared as backed by a file or filled with a pattern.

2.1.6 Device Domains

The physical memory accessible by PCI devices is specified by so called device domains. Such domains define virtual mappings of physical memory regions for one or multiple devices. Device references select a subset of hardware devices provided by the platform.

Device domains are isolated from each other by the use of Intel VT-d. Thus they can only be specified and enforced on systems that provide at least one IOMMU[1].

2.1.7 Events

Events are an activity caused by a subject (source) that impacts a second subject (target) or is directed at the kernel. Events are declared globally and have a unique name to be unambiguous.

Subjects can use events to either deliver an interrupt, hand over execution to or reset the state of a target subject. The first kind of event provides a basic notification mechanism and enables the implementation of event-driven services. The second type facilitates suspension of

[1] Input/Output Memory Management Unit

execution of the source subject and switching to the target. Such a construct is used to pass the thread of execution on to a different subject, e.g. invocation of a debugger subject if an error occurs in the source subject. The third kind is used to faciliate the restart of subjects.

Kernel events are special in that they are targeted at the kernel. The currently supported events are system reboot and shutdown.

2.1.8 Communication Channels

Inter-subject communication is represented by so called channels. These channels represent directed information flows since they have a single writer and possibly multiple readers. Optionally a channel can have an associated notification event (doorbell interrupt).

Channels are declared globally and have an unique name to be unambiguous.

2.1.9 Components

A component is a piece of software executed by the SK. Similar terms are partition or container. They represent the building blocks of a component-based system.

The description of a component specifies the binary program file including the virtual memory location as well as the view of the expected execution environment. This environment is defined in terms of logical resources such as for example communication channels.

2.1.10 Subjects

Subjects are instances of components. A subject specification references a component and maps the declared logical resources to physical resources provided by the system.

Besides the component resource mappings, subjects can specify extra resources such as device and/or memory mappings. This is useful for subjects which are able to enumerate the available resources at runtime via configuration mechanisms like ACPI or the Subject Information Page.

Furthermore, subject specifications enable the declaration of events a subject is allowed to trigger and receive.

Subjects also have an associated profile (e.g. native or Linux) which determines properties of the execution environment provided by the kernel.

2.1.11 Scheduling Plans

The Muen SK performs scheduling of subjects in a fixed, cyclic and preemptive way according to a user-specified regime. Scheduling information is declared in so called scheduling plans. They specify in what order subjects are executed on which logical CPU and for how long. Multiple scheduling plans can be specified to enable the definition of different system execution profiles which can be switched during runtime.

A scheduling plan is specified in terms of frames. A *major frame* consists of a sequence of minor frames. When the end of a major frame is reached, the scheduler starts over from the beginning and uses the first minor frame in a cyclic fashion. This means that major frames are repetitive. A *minor frame* specifies a subject and a precise amount of time.

The `Mugenschedcfg` tool can be used to automatically generate scheduling plans from a given scheduling configration, see section 5.4.

2.2 Format

The system policy is specified in XML. There are currently three different policy formats:

- Source Format
- Format A
- Format B

The motivation to have several policy formats is to provide abstractions and a compact way for users to specify a system while simultaneously facilitate reduced complexity of tools operating on the policy.

The implementation of such tools is simplified by the absence of higher-level abstractions which would make the extraction of input data more involved. As an example, the page table generation tool can directly access all virtual memory mappings of a subject and must not concern itself with channels. The channel abstraction has already been broken down into the corresponding memory elements during the policy compilation step of the build process (see section 3.1).

2.2.1 Source Format

The user-specified policy is written in the source format. Constructs such as channels or events provide abstractions to simplify the specification of component-based systems. Many XML elements and attributes are optional and will be filled in with default values during later steps of the policy compilation process.

Kernel and $\tau0$ resources are not part of the source format since they are also automatically added by the policy expansion step.

Additionally the use of configuration values enables easy parametrization of the system policy.

2.2.2 Format A

Format A is a processed version of the source format where all includes are resolved and abstractions such as channels have been broken down into their underlying elements. For example, a channel is expanded to a physical memory region and the corresponding writer and reader subject mappings with the appropriate access rights.

In this format all implicit elements, such as for example automatically generated page table memory regions, are specified. The kernel and $\tau0$ configuration is also declared as part of format A.

The only optional attributes are addresses of physical memory regions.

2.2.3 Format B

Format B is equivalent to Format A except that all physical memory regions have a fixed location (i.e. their physical address is set).

3 Build Process

The build of a system is divided into the following steps:

- Policy compilation
- Policy validation
- Structure generation

- Image packaging

The toolchain is composed of several tools that operate on a user-specified system policy. Following the Unix philosophy ”A program should do only one thing and do it well” each of the tools performs a specific task. They work in conjunction to process a user-defined policy and build a bootable system image. An in-depth description of the involved tools is given in section 4 while figure 1 gives an overview of the whole build process.

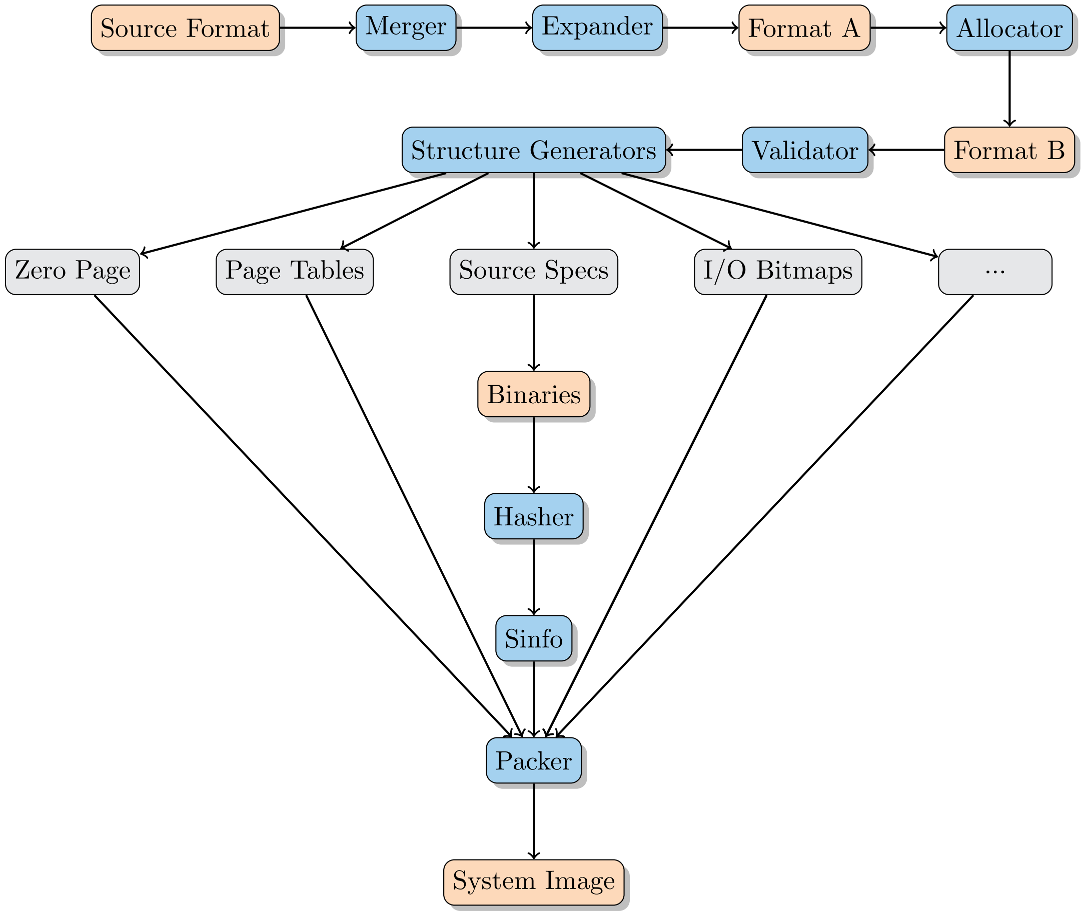

Figure 1: Build process

3.1 Policy Compilation

Policy compilation encompasses the tasks involved to transform the policy from source format to format A and finally to format B, which is the fully expanded format with no implicit properties.

The Merger tool is responsible to merge all XML files referenced by the user-specified system policy in format source. It is basically an implementation of the XML XInclude mechanism[2] with the additional benefit that the resulting policy is already well formatted to minimize the

[2] http://www.w3.org/TR/xinclude-11/

difference in the generated policies resulting from the subsequent tasks. This allows the user to easily review (`diff`) and therefore verify the results of each policy compilation task.

Using the XInclude mechanism, the policy writer is able to separate and organize the system policy as desired. Instead of specifying the whole policy in one file, the subject specifications could be put in separate files, or if different system descriptions share common parts, they could be extracted as well. See section 4.1 for more information about the Merger tool.

Expressions can be used to formulate (nested) boolean terms using the numeric equality/inequality and logical operators. They are evaluated to boolean config values prior to processing conditionals.

The use of conditionals enables selective activation of parts of the source policy depending on the value of a given config variable. This allows flexible customization of a system during policy compilation time by setting the value of a config variable or formulating an appropriate boolean expression.

After the merge task, the Expander tool takes care of completing the user-specified policy with additional information and abstractions only available in format source are resolved to low-level mechanisms.

For example, the concept of *channels* only exists in format source. Therefore a channel specified in format source must be expanded to shared memory regions with optional associated events in format A. Also, the Expander tool inserts specifications for the Muen kernel itself so the user is lifted from that burden. Generally, the aim of the expansion task is to make the life of a policy writer as easy as possible by expanding all information which can be derived automatically. Section 4.2 explains the Expander tool in detail.

The result of the expansion task is a policy in format A which is the input for the Allocator tool. This tool is responsible to assign a physical memory address to all memory regions which are not already explicitly stated. By querying the hardware section of the policy, the tool is aware of the total amount of available RAM on a specific system and allocates regions of it for memory elements with no explicit physical address. The Allocator tool also implements optimization strategies to keep the resulting system image as small as possible. For example, file-backed memory regions (e.g. a memory region storing a component executable) are preferably placed in lower physical regions. See section 4.3 for a description of the Allocator tool.

After the allocation task is complete, the policy is stored in format B. This format states all system properties explicitly and is used as input for the Validation step discussed in the following section.

3.2 Policy Validation

Before structures required to pack the final system image are generated, the policy must be thoroughly validated to catch errors in the system specification. Such errors might range from overlapping memory, undefined resource references to incomplete scheduling plans etc. The Validator task performs checks that assure the policy in format B is sound and free from higher-level errors that are not covered by XML schemata restrictions.

It is important to always run the Validator as the system could otherwise exhibit unexpected behavior. This is especially true if a policy writer decides to specify the system directly in format B which is also possible of course (but not advised). Section 4.4 explains the Validator tool and lists some example checks performed by the tool as illustration.

3.3 Structure Generation

The structure generation step encompasses various tools which extract information from a policy in format B and generate files in different formats (see figure 1).

While some generated files are directly linked into the Muen kernel (i.e. Source Specs, see 4.5.8), most of them are packed into the final system image by the packer tool.

For example, the tool responsible to generate page table structures queries memory mappings and the associated physical memory regions from the policy and creates page table structures in accordance to the format specified by the Intel Software Developer's Manual (SDM). The resulting files are packed into the system image and only applied by the kernel. The kernel itself does not care about memory management, all required tables are pre-built during system integration.

For more information about the structure generators, see section 4.5.

3.4 Image Packaging

The Packer tool assembles the final system image by first allocating a memory buffer which is initialized to zero. The size of the buffer is large enough to hold the complete system image, which consists of all file-backed memory regions specified in the policy:

- Kernel binary
- Kernel page tables
- I/O bitmaps
- MSR bitmaps
- MSR store
- Subject binaries
- Subject page tables
- VT-d tables
- ACPI tables for VM subjects
- Initial Ramdisks for Linux subjects
- Zero Page structures for Linux subjects
- Muen subject information structures

It then simply iterates over all file-backed memory regions and inserts the contents of the specified files into the allocated buffer. After performing various post-checks on the created image, it is written to a file. The resulting image can be booted by any Multiboot[3] compliant bootloader.

For more information about the Packer tool, see section 4.7.

4 Core Tools

This section describes the tools which form the core of the Muen toolchain.

[3] `https://www.gnu.org/software/grub/manual/multiboot/multiboot.html`

4.1 Merger

The Merger combines user-provided system policy files into a single XML document. It evaluates expressions to boolean configuration values and resolves conditionals.

Name
`mucfgmerge`
Input
System configuration as XML, Colon-separated list of include paths
Output
System policy in format source (merged)

This tool reads the system configuration and merges the specified system policy, hardware and platform files into a single file. It evaluates boolean expressions and resolves conditional parts of the policy. Included files are inserted at the corresponding locations in the merged file. The XML content is re-formatted so changes to the policy by subsequent build steps can be manually reviewed or visualized by diffing the files.

4.2 Expander

The expander completes the user-provided system policy by creating or deriving additional configuration elements.

Name
`mucfgexpand`
Input
System policy in format source
Output
System policy in format A (expanded)

The Expander performs the following actions:

- Pre-check the system policy to make sure it is sound
- Expand channels
- Expand device resources
- Expand device isolation domains
- Expand kernel sections
- Expand $\tau 0$ subject
- Expand additional memory regions
- Expand hardware-/platform-related information
- Expand additional subject information
- Expand profile-specific information
- Expand scheduling information
- Post-check resulting policy

4.3 Allocator

The Allocator is responsible to assign a physical address to all global memory regions.

Name
`mucfgalloc`

Input
System policy in format A

Output
System policy in format B (allocated)

First, the Allocator initializes the physical memory view of the system based on the physical memory blocks specified in the XML hardware section. It then reserves memory that is occupied by pre-allocated memory elements (i.e. memory regions with a physical address or device memory). Finally it places all remaining memory regions in physical memory. In order to reduce the size of the final system image file-backed memory regions are placed at the start of memory.

4.4 Validator

The Validator performs additional checks that go beyond the basic restrictions imposed by the XML schema validation. Currently over 110 checks are performed.

Name
`mucfgvalidate`

Input
System policy in format B

Output
None, raises exception on error

Examples of checks include:

- Assert that references between policy elements are correct (e.g. a physical memory region referenced by a virtual memory region exists)
- Assert that memory regions do not overlap
- Assert that device interrupts are unique
- Assert that no subject has access to system or kernel memory
- Assert that scheduling plan major frames have the same length on each CPU

4.5 Structure Generators

These tools do not change the policy and use it read-only.

4.5.1 Page Tables

Generate page tables for kernel(s) and subjects.

Name
`mugenpt`

Input
System policy in format B

Output
Page tables of kernels and subjects in binary format

Output format

- IA32-e paging structures, Intel SDM Vol. 3A, section 4.5
- EPT paging structures, Intel SDM Vol. 3C, section 28.2

The tool generates paging structures for subjects and kernels running on each CPU. These page tables are used to grant access to physical memory according to the virtual memory layout of the subject. The rest of physical and device memory is isolated from the subject.

An IA32-e page table is generated for each kernel running on a logical, active CPU. Depending on the subject profile either native 64-bit IA32-e or Extended Page Tables (EPT) are generated.

Page tables are used by the memory management unit (MMU) to enforce isolation of physical memory according to the system policy.

4.5.2 VT-d Tables

Generate VT-d tables for each device isolation domain.

Name
`mugenvtd`

Input
System policy in format B

Output
VT-d tables of device domains in binary format

Output format
VT-d tables according to Intel VT-d specification, section 9

The tool creates root, context and second-level address translation tables for Intel VT-d DMAR (DMA[4] remapping) hardware (see Intel VT-d specification, section 3). DMAR is used to restrict direct hardware device access to physical memory via DMA. Devices are put in so-called device security or device isolation domains and are only allowed to access physical memory as granted by the policy.

Interrupt remapping tables are also generated for Intel VT-d IR to ensure that physical devices can only generate interrupt requests as specified by the system policy.

4.5.3 I/O Bitmaps

Generate I/O bitmaps for each subject.

Name
`mugeniobm`

Input
System policy in format B

Output
I/O bitmaps of subjects in binary format

[4] DMA - Direct Memory Access

Output format
Intel SDM Vol. 3C, section 24.6.4

The tool generates I/O bitmaps for each subject. Access to device I/O ports is granted according to the device I/O port resources assigned to a subject.

I/O bitmaps are used by the hardware (VT-x) to enforce access to I/O ports according to the system policy.

4.5.4 MSR Bitmaps

Generate MSR bitmap for each subject.

Name
`mugenmsrbm`
Input
System policy in format B
Output
MSR bitmaps of subjects in binary format
Output format
Intel SDM Vol. 3C, section 24.6.9

The tool generates MSR bitmaps for each subject. Access to Model-Specific Registers (MSRs) is granted according to the MSRs assigned to a subject.

MSR bitmaps are used by the hardware (VT-x) to enforce access to Model-Specific Registers according to the system policy.

4.5.5 MSR Stores

Generate MSR store for each subject with MSR access.

Name
`mugenmsrstore`
Input
System policy in format B
Output
MSR store files of subjects in binary format
Output format
Intel SDM Vol. 3C, table 24-11

The tool generates MSR stores for each subject. The MSR store is used to save/load MSR values of registers not implicitly handled by hardware on subject exit/resumption.

MSR stores are used by hardware (VT-x) to enforce isolation of MSR (i.e. subjects that have access to the same MSRs cannot transfer data via these registers).

4.5.6 ACPI Tables

Generate ACPI tables for all Linux subjects.

Name
`mugenacpi`

Input
System policy in format B
Output
ACPI tables of all Linux subjects
Output format
Advanced Configuration and Power Interface (ACPI) Specification[5]

ACPI tables are used to announce available hardware to VM subjects. A set of tables consists of an RSDP, XSDT, FADT and DSDT table. See the ACPI specification for more information about a specific table.

4.5.7 Linux Zero Pages

Generate Zero Pages for all Linux subjects.

Name
`mugenzp`
Input
System policy in format B
Output
Zero pages of all Linux subjects
Output format
Linux Boot Protocol[6]
Zero Page[7]

The so-called Zero Page (ZP) exports information required by the boot protocol of the Linux kernel on the x86 architecture. The kernel uses the provided information to retrieve settings about its running environment:

- Type of bootloader
- Map of physical memory (e820 map)
- Address and size of initial ramdisk(s)
- Kernel command line parameters

4.5.8 Source Specifications

Generate source specifications used by kernel and subjects.

Name
`mugenspec`
Input
System policy in format B
Output
Source specifications in SPARK, C and GPR format

Gathers data from the system policy to generate various source files in SPARK, C and GNAT project file (GPR) format. Created output includes constant values for memory addresses, device resources, scheduling plans, etc.

[5] `http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf`
[6] `https://www.kernel.org/doc/Documentation/x86/boot.txt`
[7] `https://www.kernel.org/doc/Documentation/x86/zero-page.txt`

4.5.9 Component Source Specifications

Generate source specifications from component descriptions.

Name
: `mucgenspec`

Input
: System policy in format source, Name of component

Output
: Component source specifications in SPARK

The component spec generation tool reads the specification of a given component and generates Ada/SPARK packages containing constants of the declared logical component resources. The generated specifications can be used in the component source code to access the declared resources.

4.5.10 Subject Info

Generate subject information data for each subject.

Name
: `mugensinfo`

Input
: System policy in format B

Output
: Subject info data in binary format

Output format
: As specified in `common/musinfo/musinfo.ads`

The Sinfo page is used to export subject information data extracted from the system policy to VM subjects. Currently, information about available memory regions, communication channels and assigned PCI devices is provided.

4.6 Hasher

The Mucfgmemhashes tool is used to add memory integrity hashes to a given policy.

Name
: `mucfgmemhashes`

Input
: System policy in format B

Output
: System policy in format B with memory integrity hashes

The Mucfgmemhashes tool appends a hash to all memory regions with fill and file content. It must run after all files have been generated by the structure generator tools.

The actual hash is generated using the SHA-256 algorithm and is intended to be used to verify the integrity of memory regions during runtime.

Note that no hashes are generated for sinfo memory regions. Since the hash information will be exported via sinfo, and the sinfo region is itself part of the memory information of a subject, this hash would be self-referential.

The tool also replaces all occurrences of `hashRef` elements. A hash reference element instructs the tool to copy the hash element of the referenced memory region after message digest generation.

From an abstract point of view, the `hashRef` element is a way to link multiple memory regions by declaring that the hash of the content is the same. The hash may serve as an indicator on how to reconstruct the (initial) content of a memory region. This mechanism is heavily used by the subject loader (SL) during subject init and reset operation. The subject loader expander remaps writable memory regions of the loadee (the subject under loader control) to SL and replaces the original regions with new ones containing a hash reference to the associated pyhsical memory region. This way SL is able to determine the intended content of the target memory region by looking up the region in its sinfo page by using the hash value as key.

4.7 Packer

The Packer is responsible to assemble the final system image.

Name
`mupack`

Input
System policy in format B, Input directories, System image filename

Output
System image file

The Packer calculates the size of the resulting system image by querying the file-backed memory region with the highest physical memory address. It allocates a buffer of that size which is initially filled with zeros. It then iterates over all file-backed memory regions in the policy and adds the content of the files to the buffer. Before writing the buffer to a file specified on the command line, the packer tool performs post-checks on the buffer to make sure it is sound.

5 Additional Tools

This section lists additional helper tools which simplify the process of generating and validating a Muen system.

5.1 Kernel ELF Checker

The `Mucheckelf` tool enforces that the format of a given Muen kernel ELF binary matches the kernel memory layout specified in a system policy.

Size, VMA (Virtual Memory Address) and permissions of binary ELF sections are validated against kernel memory regions defined in the policy. The following table lists the correspondence of ELF section names to logical kernel memory region names.

ELF Section	Memory Name
.text	kernel_text
.data	kernel_data
.rodata	kernel_ro
.bss	kernel_bss

5.2 Stack Usage Checker

The `Mucheckstack` tool statically calculates the worst-case stack usage of a native Ada/SPARK component or the Muen kernel compiled with the -fcallgraph-info switch[8].

The tool takes a GNAT project file and a stack limit in bytes as input. All control-flow information (.ci) files found in the object directory of the main project and all of its dependencies are parsed. Once the control-flow graph is constructed the maximum stack usage of each subprogram is calculated and checked against the user-specified limit. The tool exits with a failure if a stack usage exceeding the limit is detected.

Note that the tool is not applicable to arbitrary software projects as it does not handle dynamic/unbounded stack usage and recursion. In the context of the Muen project these cases can not occur since they are prohibited by the following restriction pragmas:

- No_Recursion
- No_Secondary_Stack
- No_Implicit_Dynamic_Code

Additionally, the `-Wstack-usage` compiler switch warns about potential unbounded stack usage.

5.3 Hardware Config Generator

The `Mugenhwcfg`[9] tool has been created to automate the process of gathering all necessary hardware information. To collect data for a new target hardware all that is required is to run the tool on a common Linux distribution. See the project README for more information.

5.4 Scheduling Plan Generator

The `Mugenschedcfg`[10] tool generates scheduling plans for Muen based on a given scheduling configuration. The configuration allows the user to specify the following scheduling properties:

- Number of CPU cores
- The tick rate of the CPUs
- Security constraints to meet
  - Same CPU domains
  - Simultaneous execution domains
- Subject specifications
- Score functions
- Number of plans to generate
- Plans
  - Weighting of plan importance

[8] `https://www.adacore.com/uploads/technical-papers/Stack_Analysis.pdf`

[9] `https://git.codelabs.ch/?p=muen/mugenhwcfg.git`

[10] `https://git.codelabs.ch/?p=muen/mugenschedcfg.git`

  - Levels
  - Subjects of a plan
  - Chains with throughput metric

Consult the project's README and example plans on how to use the tool.