

SUIT
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

B. Moran
T. Ibbs
G. Psimenos
ARM Limited
March 11, 2019

An Information Model for Behavioural Description of Firmware Update and
Related Operations

[draft-moran-suit-behavioural-manifest-01](#)

Abstract

This specification describes an approach to formally defining the behaviour of a system under firmware update and secure boot conditions. The behavioural documents described here can be used with [[Information](#)] to construct a firmware update manifest.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in [Section 4](#).e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1.	Introduction	3
2.	Conventions and Terminology	3
3.	Design Principles of the Behavioural Manifest	4
4.	Structure of a behavioural manifest	5
4.1.	Processing Steps	7
5.	Commands	7
5.1.	Verify Recipient Identity	8
5.2.	Verify Image Presence	9
5.3.	Verify Component Properties	9
5.4.	Verify System Properties	9
5.5.	Verify 3rd-party Authorisation	9
5.6.	Process sub-behaviours	9
5.7.	Process Dependencies	9
5.8.	Set Parameters	10
5.9.	Move an Image	10
5.10.	Invoke an Image	10
5.11.	Wait for an Event	10
6.	Parameters	11
6.1.	Strict Order	11
6.2.	Soft Failure	11
6.3.	Source List	11
6.4.	Processing Step Configurations	12
6.5.	Image Identifier	12
7.	ACLs/permissions	12
8.	Workflows	13
9.	Examples	15
9.1.	Example 1: Boot an image on an XIP processor	16
9.2.	Example 2: Download an image	16
9.3.	Example 3: Check compatibility, download, and boot	17
9.4.	Example 4: Check compatibility, download, load from external, and boot	17

9.5. Example 5: Check compatibility, download, load with decompress, and boot	18
9.6. Example 6: Check compatibility, download, install-from-external and boot	20
9.7. Example 7: Download and boot an image with a dependency .	21
9.8. Example 8: Download and boot an image with a dependency using override.	23
10. IANA Considerations	25
11. Security Considerations	25
12. References	25
12.1. Normative References	25
12.2. Informative References	25
Appendix A. Mailing List Information	27
Authors' Addresses	27

[1.](#) Introduction

Conventional hierarchical, descriptive documents, such as [draft-moran-manifest-03](#) imply the behaviour of the recipient without specifying that behaviour. This creates a situation where recipients must construct the assumed behaviour in accordance with a specification, handling many edge cases and introducing significant complexity. Capabilities are difficult to specify because they imply behaviours, rather than data, but the descriptive document only specifies data, not capabilities. This leaves the document author to interpret capabilities (supported behaviours) into allowable combinations of data. This disconnect demonstrates that devices require both an information model and a behavioural model.

This creates a situation where the behaviour of a system is imprecisely specified by the documents that it uses to perform secure boot and secure firmware update. In high security applications, precise specification of behaviour is beneficial, and can even be used for formal verification.

By specifying the behaviour of a device in a document rather than just the information, the gap between specified information and specified behaviour can be closed.

[2.](#) Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#) [[RFC2119](#)].

- SUIT: Software Update for the Internet of Things, the IETF working group for this standard.

- Image: A piece of information to be delivered. Typically Firmware for the purposes of SUIT.
- Document, Behavioural Document: The data that defines the behaviour of a recipient.
- Component: A target for storage of the Image
- Dependency: Another Behavioural Document upon which the current Document relies.
- Recipient: The system, typically an IoT device, that receives a Behavioural Document.
- Condition: A test for a property of the Recipient or its components.
- Directive: An action for the Recipient to perform.
- Command: A Condition or a Directive.

3. Design Principles of the Behavioural Manifest

In order to provide flexible behaviour to constrained devices, while still allowing more powerful devices to use their full capabilities, the SUIT manifest takes a new approach, encoding the required behaviour of a Recipient device, instead of just presenting the information used to determine that behaviour. This gives benefits equivalent to those provided by a scripting language or byte code, with two substantial differences. First, the language is extremely high level, consisting of only the operations that a device may perform during update and secure boot of a firmware image. The language specifies behaviours in a linearised form, without branches or loops. Conditional processing is supported, and parallel and out-of-order processing may be performed by sufficiently capable devices.

By structuring the data in this way, the manifest processor becomes a very simple engine that uses a pull parser to interpret the manifest. This pull parser consists of command handlers that evaluate a Condition or execute a Directive. Most data is structured in a highly regular pattern, which simplifies the parser.

The results of this allow a Recipient with minimal functionality to perform complex updates with reduced overhead. Conditional execution of commands allows a simple device to perform important decisions at validation-time, such as which differential update to download for a given current version, or which hash to check, based on the installation address.

Dependency handling is vastly simplified as well. Dependencies function like subroutines of the language. When a manifest has a dependency, it can invoke that dependency's commands and modify their behaviour by setting parameters. Because some parameters come with security implications, the dependencies also have a mechanism to reject modifications to parameters on a fine-grained level.

Developing a robust permissions system works in this model too. The Recipient can use a simple ACL that is a table of Identities and Component Identifier permissions to ensure that only manifests authenticated by the appropriate identity have access to define a component.

Capability reporting is similarly simplified. A Recipient can report the Commands and Parameters that it supports. This is sufficiently precise for a manifest author to create a manifest that the Recipient can accept.

Because the behavioural description is precise, and the machine definition upon which it relies is very simple it can be augmented with a proof that the effects of an update fall within a specified policy, in the same way as Proof Carrying Code. By combining this capability with formal verification of the document processor, it is possible to prove the result of a firmware update, prior to application, either on the target or on an intermediate system. The proof can be discarded before distribution to constrained nodes, creating no additional overhead.

The simplicity of design in the Recipient due to all of these benefits allows even a highly constrained platform to use advanced update capabilities.

4. Structure of a behavioural manifest

Behavioural manifests are divided into sections based on the behaviours of the Recipient. There are 8 conceptual sections of a behavioural manifest, listed below.

1. Document-global data
2. Common behaviour
3. Dependency Resolution behaviour
4. Image Acquisition behaviour
5. Image Application behaviour

6. System Validation behaviour
7. Image Loading behaviour
8. Image Invocation behaviour

Document-global data contains the information that is required to enable most behaviours along with a security parameter. The information contained is listed below.

1. Document Structure Version
2. Document Sequence Number
3. List of Dependencies
 1. List of Components affected by each Dependency
4. List of Components affected by this Document

Common behaviour is executed prior to each other behaviour. It is used to make common decisions for all other behaviours.

Dependency Resolution is used to ensure that all required documents have been collected prior to attempting to acquire any image. Where a document has no dependencies, this section is not required.

Image Acquisition is used to obtain images from local or remote sources and stage them for use by the Recipient. If a Document lists no affected components, then Image Acquisition is not required. If a device operates in a simultaneous Acquisition & Application mode (for example, streaming installation), then Image Acquisition should be discarded in favour of Image Application. Image Acquisition can be used in combination with several processing steps defined in [Section 4.1](#).

Image Application is used to place an image into its long-term storage. An image can be moved either from a staging area or from another source (including a remote) into its long-term storage. This can be done in combination with several processing steps defined in [Section 4.1](#).

System validation is used to ensure that all required dependencies are present and that all required images are present. This process is equivalent to that used in the validation portion of Secure Boot workflows.

Image Loading is used to ensure that all required images are moved from long-term storage to active use storage. This can include steps like copying from external Flash to RAM, as defined by Component information. This can be done in combination with several of the processing steps defined in [Section 4.1](#).

Image Invocation is used to finalise the manifest processor's behaviour and forward execution to a designated component. This is equivalent to Bootloader behaviour.

Some behaviours need only a single successful invocation. These behaviours can then be discarded, provided that the Document serialisation provides a mechanism to do so. Typically discarded behaviours are Dependency Resolution, Image Acquisition, and Image Application.

[4.1. Processing Steps](#)

Processing steps are the translation that is performed on an image prior to its execution. These steps typically include, in order, symmetric cryptographic operations, decompression operations, unpacking operations.

Each of these operation may need additional information, such as which algorithm is in use or arguments to that algorithm, such as key identifiers for cryptographic operations. This information can be encoded in Processing Step parameters, as described in [Section 6.4](#).

[5. Commands](#)

Behaviours are constructed as lists of commands, each of which may have arguments. The behaviours listed in any of the specified sections derives from a short list of commands. These commands are divided into two types, Conditions (verification operations) and Directives (action operations)

The lists of commands are logically structured into sequences of zero or more conditions followed by zero or more directives. The *logical* structure is described by the following CDDL:

```
Behaviour = [
  + {
    conditions => [ * Condition],
    directives => [ * Directive]
  }
]
```


The conditions form preconditions that MUST be true for the following sequence of directives to be executed.

However, this organisation could introduce significant complexity in a parser, so the structure MAY be flattened into the following:

Behaviour = [* (Condition/Directive)]

This does not alter the logical organisation of sequences of preconditions that precede sequences of directives, but it simplifies the consumption of commands in behaviours.

The Conditions are, broadly, those listed below.

1. Verify device identity
2. Verify image presence (correctness) or absence
3. Verify component properties
4. Verify system properties
5. Verify 3rd-party authorisation

The Directives are those listed below.

1. Process sub-behaviours
2. Process dependencies
3. Set parameters
4. Move an Image or Document
5. Invoke an Image
6. Wait for an event

5.1. Verify Recipient Identity

This is used to ensure that the document is being processed by the appropriate device and to eliminate incompatibility failures. Identity can include what sort of device is targeted, what software it uses, or who made it. Identity can also include the particular device that is targeted.

5.2. Verify Image Presence

This is used to ensure that a required image is present. This often includes the use of cryptographic checksums to validate the contents of an image contained in a component.

5.3. Verify Component Properties

This can be used to verify several properties of a targeted component, such as the current nominal version of its APIs, or the base address or offset that it will use.

5.4. Verify System Properties

This can be used to verify several properties of the system including, the current power state, such as battery level or presence of external power, the current time reported by the device, or the current state of a controlled piece of equipment.

5.5. Verify 3rd-party Authorisation

This can be used to ensure that some third-party has approved an action, in a system specific way. Options include checking a remote system for authorisation, looking for a cryptographic token, or invoking a user-interface.

5.6. Process sub-behaviours

In some use-cases, a decision must be made as to which of several behaviours must be invoked. To enable this use-case, sub-behaviours provide a mechanism to permit soft-failure of a Condition. A parameter of the sub-behaviour controls its response to a Condition check failure, allowing the command following the sub-behaviour to be the next to execute, or causing failure of the whole behaviour, depending on its value.

This allows the construction of a conditional behaviour. A sub-behaviour is invoked allowing condition checks to soft-fail. Once the conditions that inform the conditional behaviour have succeeded, the soft-failure parameter is switched to hard-failure, so that further condition failures will be detected.

5.7. Process Dependencies

Dependencies are processed by invoking two behaviours within the dependency; first the common behaviour is invoked, then the behaviour matching the current behaviour of the current document is invoked. So, if "Image Application" is active when "Process Dependencies" is

invoked, then each dependency's "Common" behaviour will be invoked, followed by its "Image Application" behaviour.

It can be advantageous to process dependencies in a particular order, so dependencies are processed in the order specified.

A failure of processing of any dependency results in a failure of the Process Dependencies behaviour.

[5.8.](#) Set Parameters

Many Commands are partially governed by configuration present in the form of parameters. Parameters control the source used for Image Acquisition, the processing steps applied to those images, the order in which commands are processed and more. See [Section 6](#) for more information.

Parameters can be set in one of three ways. They can be set-if-unset (the default), append-if-set (typically used for source lists), set-always (used for critical parameters). Parameters are either global or scoped by Component/Component Group.

[5.9.](#) Move an Image

This Command directs the document processor to acquire an Image or Document and store it to a specified Component or Document storage, respectively. The source can be local or remote, or a prioritised list of local and remote sources. The source or source list is specified by the source parameter. The Image or document can optionally be modified in transit by a sequence of processing steps, as defined in [Section 6](#).

[5.10.](#) Invoke an Image

This command forwards execution to the specified image in much the same way as a bootloader. As with bootloaders, the semantics of forwarding execution are application defined. An argument may be provided to the Image. The semantics of the argument are application-defined.

[5.11.](#) Wait for an Event

Frequently, a behaviour needs to wait for a property of the system to change. This may be a message from a remote, a time, a power state, a user-interaction, or some other system parameter.

6. Parameters

Available parameters may vary by implementation, but some core parameters are usually present.

Typical parameters are listed below.

1. Strict Order
2. Soft-Failure
3. Source List
4. Processing Step Configuration
5. Image Identifier

In some use-cases, device identity may also be configured in a parameter.

6.1. Strict Order

Some advanced devices may have particular requirements regarding command ordering within a behaviour. Others may enable parallel execution of commands. When the Strict Order parameter is set to False, these extended capabilities are enabled. An advanced device may then aggregate all successive commands up until the behaviour ends or the Strict Order parameter is returned to True and process those commands in parallel or reorder them as it requires. Strict Order defaults to True. If a device does not support command reordering or parallel processing, Strict Order = False has no effect.

6.2. Soft Failure

When a device invokes a sub-behaviour, any condition check failure and any directive failure causes the behaviour to immediately abort. However, if the Soft Failure parameter is True, then an abort due to a condition failure does not cause the sub-behaviour to report failure. If the Soft Failure parameter is True, indicating hard failure, then any abort causes the sub-behaviour to report failure as well.

6.3. Source List

The source list is scoped to an individual component or dependency. It is a prioritised search path for the Move command to use in order

to find an image or document. It can contain either local sources, such as other components, or remote ones, such as URIs.

6.4. Processing Step Configurations

Many processing steps require configuration to operate, or configuration informs whether or not to use them. Common processing steps include symmetric cryptography, compression or decompression operations, and packing or unpacking, for example relocation, differential compression, or hex file interpretation.

Processing step configuration is scoped to an individual component or document.

6.5. Image Identifier

In order to determine whether an image is present, the verify image presence condition requires an identifier for the image. This could be a version number or a cryptographic identity such as a digest.

7. ACLs/permissions

To manage permissions in documents, there are three models that can be used.

First, the simplest model requires that all documents are authenticated by a single identity. This mode has the advantage that only a single document needs to be authenticated, since each document's dependencies are uniquely identified in that document.

This simplest model can be extended by adding key delegation without much increase in complexity.

A second model requires an ACL to be presented to the device, authenticated by a trusted party or stored on the device. This ACL grants access rights for specific Components or Component Groups to the listed identities or identity groups. Any identity may verify that an image is present, but Moving an image into or out of a Component requires approval from the ACL.

A third model allows a Document Processor to provide even more fine-grained controls: The ACL lists the Component or Component Group that an identity may use, and also lists the commands that the identity may use in combination with that Component/Group.

8. Workflows

The two most common workflows are image installation and image invocation. Both of these workflows use a common component: `do_commands`.

`do_commands` uses the following pseudocode:

```
function do_commands(section, sequence)
  rc = SUCCESS
  foreach (command in $sequence)
    choose $command[Type]:
      case Sub-Behaviour:
        Load commands = $command[Argument][commands]
        Load parameters = System Parameters
        Load soft_failure = $parameters[Soft Failure]
        Call rc = do_commands(commands)
        if (soft_failure AND is_condition_failure(rc))
          rc = SUCCESS
        endif
      endcase
      case Process Dependency:
        ; Note Dependency selection can be done via argument or
        ; parameter. May process multiple dependencies in a list.
        Load Dependency
        Load common = $Dependency[Common Sequence]
        Call rc = do_commands(Common, $common)
        if (rc is SUCCESS?)
          Load $sequence = $Dependency[$section]
          Call rc = do_commands($section, $sequence)
        endif
      endcase
      case Set Parameters:
        Load parameter_list = $command[Argument][Parameter List]
        foreach (parameter in parameter_list)
          if (is_append(parameter?))
            Append argument value to $parameter
          elseif (is_set($parameter[Name]))
            Set $parameter[Name] = $parameter[Value]
          endif
        endfor
      endcase
      case Move:
        ; Note Component selection can be done via argument or
        ; parameter. May process multiple components in a list.
        ; Note Dependency selection can be done via argument or
        ; parameter. May process multiple dependencies in a list.
        ; Source
```



```

    Load component_list = $command[Argument][Component List]
    Load dependency_list = $command[Argument][Dependency List]
    foreach (target_list in [component_list, dependency_list])
        foreach (target in $target_list)
            Load $target parameters
            Set source = choose_best_source($parameters[Source
List])

            Acquire data from $source
            foreach (processing_step in $parameters[Processing Step
Configuration])

                rc = Process_Data($processing_step, $data)
            endforeach
            Store $data to $target
        endforeach
    endforeach
endcase
case Invoke:
    ; Note Component selection can be done via argument or
    ; parameter. May process multiple components in a list.
    Select component
    Load Argument = $command[Argument]
    Transfer execution to $component with $Argument;
endcase
case Wait:
    Load arguments = $command[Argument][Wait Arguments]
    Load type = $command[Argument][Wait Type]
    Wait ($type, $arguments)
endcase
case Device Identity:
    ; Note Device Identity selection can be done via argument or
    ; parameter. May process multiple Device Identities in a list.
    Load device_identity = $parameters[Device Identity]
    if ($device_identity is nil)
        Load device_identity = $command[Argument][Device Identity]
    endif
    rc = Compare $device_identity to $parameters
endcase
case Image Present/not Present:
    ; Note Component selection can be done via argument or
    ; parameter. May process multiple components in a list.
    ; Note Dependency selection can be done via argument or
    ; parameter. May process multiple dependencies in a list.
    Load component_list = $command[Argument][Component List]
    Load dependency_list = $command[Argument][Dependency List]
    foreach (target_list in [component_list, dependency_list])
        foreach (target in $target_list)
            Load $target parameters
            Set image_identifier = $parameters[Image Identifier]

```

```
        rc = Compare $component to $image_identifier;  
    endfor
```

```

        endfor
    endcase
    case Verify Authorisation
        Request authorisation
        Wait for authorisation
        rc = Check Response
    endcase
endchoose
endwhile (no)
endfunction

```

Installation, is represented by the following pseudocode.

```

function install(Document)
    Load $Document[Common Data] into parameters
    foreach (sequence in [Common, Dependency Resolution, Image Acquisition,
Image Application])
        rc = do_commands($sequence, $Document[$sequence]);
        if (rc is not SUCCESS)
            Abort
        endif
    endfor
endfunction

```

Image invocation is represented by the following pseudocode.

```

function invoke(Document)
    Load $Document[Common Data] into parameters
    foreach (sequence in [Common, System Validation, Image Loading, Image
Invocation])
        rc = do_commands($sequence, $Document[$sequence]);
        if (rc is not SUCCESS)
            Abort
        endif
    endfor
endfunction

```

Each operation represented here is already present in a device capable of firmware update or secure boot. This approach simply defines the mechanism by which these operations are orchestrated, and enforces that the behaviour of the system is defined by the Behavioural Document, rather than implied by it.

9. Examples

These examples demonstrate the serialisation of the behaviours of an update. They are serialised in JSON for readability, but JSON is not recommended for use on constrained devices.

9.1. Example 1: Boot an image on an XIP processor

```

{
  "structure-version" : 1,
  "sequence-number" : 1,
  "components": [
    {
      "id" : <Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "validate" : [
    {
      "condition-validate-image" : {"component" : 0}
    }
  ],
  "image-invocation" : [
    {
      "directive-run-component": {"component" : 0}
    }
  ]
}

```

9.2. Example 2: Download an image

```

{
  "structure-version" : 1,
  "sequence-number" : 2,
  "components": [
    {
      "id" : <Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "image-acquisition" : [
    {
      "directive-move" : {
        "source": "http://example.com/file.bin",
        "destination" : 0
      }
    }
  ]
}

```


9.3. Example 3: Check compatibility, download, and boot

```

{
  "structure-version" : 1,
  "sequence-number" : 3,
  "components": [
    {
      "id" : <Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "common" : [
    { "condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
    { "condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
  ],
  "image-application" : [
    {
      "directive-move" : {
        "source": "http://example.com/file.bin",
        "destination" : 0
      }
    }
  ],
  "validate" : [
    {
      "condition-validate-image" : {"component" : 0}
    }
  ],
  "image-invocation" : [
    {
      "directive-run-component": {"component" : 0}
    }
  ]
}

```

9.4. Example 4: Check compatibility, download, load from external, and boot

```

{
  "structure-version" : 1,
  "sequence-number" : 4,
  "components": [
    {
      "id" : <Flash Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],

```



```

    {
      "id" : <RAM Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "common" : [
    { "condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
    { "condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
  ],
  "image-application" : [
    {
      "directive-move" : {
        "source": "http://example.com/file.bin",
        "destination" : 0
      }
    }
  ],
  "validate" : [
    {
      "condition-validate-image" : {"component" : 0}
    }
  ],
  "load-image" : [
    {
      "directive-move" : {
        "source": 0,
        "destination" : 1
      }
    }
  ],
  "image-invocation" : [
    {
      "condition-validate-image" : {"component" : 1}
    },
    {
      "directive-run-component": {"component" : 1}
    }
  ]
}

```

9.5. Example 5: Check compatibility, download, load with decompress, and boot

```

{
  "structure-version" : 1,
  "sequence-number" : 5,
  "components": [

```



```
{
  {
    "id" : <Flash Component Identifier>,
    "digest": "<SHA256 of Compressed Image>",
    "size" : <Size of Compressed Image>
  },
  {
    "id" : <RAM Component Identifier>,
    "digest": "<SHA256 of Image>",
    "size" : <Size of Image>
  }
],
"common" : [
  {
    "condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe",
    "condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"
  }
],
"image-application" : [
  {
    "directive-move" : {
      "source": "http://example.com/file.bin",
      "destination" : 0
    }
  }
],
"validate" : [
  {
    "condition-validate-image" : {"component" : 0}
  }
],
"load-image" : [
  {
    "directive-move" : {
      "source": 0,
      "destination" : 1,
      "processing-step-compression-algorithm" : "gzip"
    }
  }
],
"image-invocation" : [
  {
    "condition-validate-image" : {"component" : 1}
  },
  {
    "directive-run-component": {"component" : 1}
  }
]
}
```


9.6. Example 6: Check compatibility, download, install-from-external and boot

```

{
  "structure-version" : 1,
  "sequence-number" : 6,
  "components": [
    {
      "id" : <External Flash Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    },
    {
      "id" : <Internal Flash Component Identifier>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "common" : [
    {"condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
    {"condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
  ],
  "image-acquisition" : [
    {
      "directive-move" : {
        "source": "http://example.com/file.bin",
        "destination" : 0
      }
    }
  ],
  "validate" : [
    {
      "sub-behaviour" : [
        "soft-failure" : True,
        "condition-validate-not-image" : {"component" : 1}
        "soft-failure" : False
        "condition-validate-image" : {"component" : 0}
      ]
    }
  ],
  "load-image" : [
    {
      "sub-behaviour" : [
        "soft-failure" : True,
        "condition-validate-not-image" : {"component" : 1}
        "soft-failure" : False
        "directive-move" : {
          "source": 0,

```



```

        "destination" : 1
      }
    ]
  }
],
"image-invocation" : [
  {
    "condition-validate-image" : {"component" : 1}
  },
  {
    "directive-run-component":{"component" : 1}
  }
]
}

```

[9.7.](#) Example 7: Download and boot an image with a dependency

```

[
  {
    "structure-version" : 1,
    "sequence-number" : 7,
    "components": [
      {
        "id" : <Component Identifier 0>,
        "digest":"<SHA256 of Image>",
        "size" : <Size of Image>
      }
    ],
    "common" : [
      {"condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
      {"condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
    ],
    "image-application" : [
      {
        "directive-move" : {
          "source": "http://example.com/file.bin",
          "destination" : 0
        }
      }
    ],
    "validate" : [
      {
        "condition-validate-image" : {"component" : 0}
      }
    ],
    "image-invocation" : [
      {
        "directive-run-component":{"component" : 0}
      }
    ]
  }
]

```



```
    }
  ]
},
{
  "structure-version" : 1,
  "sequence-number" : 8,
  "dependencies" : [
    {
      "digest" : "<SHA256 of Document 0>"
      "components" : [<Component Identifier 0>]
    }
  ],
  "components": [
    {
      "id" : <Component Identifier 1>,
      "digest": "<SHA256 of Image>",
      "size" : <Size of Image>
    }
  ],
  "dependency-resolution" : [
    {
      "directive-move" : {
        "source": "http://example.com/document0.bin",
        "destination" : <Document 0 ID>
      }
    },
    {
      "condition-validate-image" : {"dependency" : 0}
    }
  ],
  "image-application" : [
    {
      "directive-move" : {
        "source": "http://example.com/file1.bin",
        "destination" : 1
      }
    },
    { "process-dependency" : 0 }
  ]
  "validate" : [
    {
      "condition-validate-image" : {"dependency" : 0}
    },
    { "process-dependency" : 0 },
    {
      "condition-validate-image" : {"image" : 1}
    }
  ],
}
```



```

    "image-invocation" : [
      { "process-dependency" : 0 }
    ]
  }
]

```

9.8. Example 8: Download and boot an image with a dependency using override.

Override fetch location for dependency.

```

[
  {
    "structure-version" : 1,
    "sequence-number" : 7,
    "components": [
      {
        "id" : <Component Identifier 0>,
        "digest": "<SHA256 of Image>",
        "size" : <Size of Image>
      }
    ],
    "common" : [
      { "condition-vendor-id" : "fa6b4a53-d5ad-5fdf-be9d-e663e4d41ffe"},
      { "condition-class-id" : "1492af14-2569-5e48-bf42-9b2d51f2ab45"}
    ],
    "image-application" : [
      {
        "set-parameter" : {
          "component" : 0
          "source": "http://example.com/file.bin",
        }
      },
      {
        "directive-move" : {
          "destination" : 0
        }
      }
    ],
    "validate" : [
      {
        "condition-validate-image" : {"component" : 0}
      }
    ],
    "image-invocation" : [
      {
        "directive-run-component": {"component" : 0}
      }
    ]
  }
]

```



```

    ]
  },
  {
    "structure-version" : 1,
    "sequence-number" : 8,
    "dependencies" : [
      {
        "digest" : "<SHA256 of Document 0>"
        "components" : [<Component Identifier 0>]
      }
    ],
    "components": [
      {
        "id" : <Component Identifier 1>,
        "digest": "<SHA256 of Image>",
        "size" : <Size of Image>
      }
    ],
    "dependency-resolution" : [
      {
        "directive-move" : {
          "source": "http://example.com/document0.bin",
          "destination" : <Document 0 ID>
        }
      },
      {
        "condition-validate-image" : {"dependency" : 0}
      }
    ],
    "image-application" : [
      {
        "directive-move" : {
          "source": "http://example.com/file1.bin",
          "destination": 1
        }
      },
      {
        "set-parameter" : {
          "component" : 0
          "source": "http://other-host.com/file.bin",
        }
      },
      { "process-dependency" : 0 }
    ]
  },
  "validate" : [
    {
      "condition-validate-image" : {"dependency" : 0}
    },
  ],

```



```
        { "process-dependency" : 0 },
        {
            "condition-validate-image" : {"image" : 1}
        }
    ],
    "image-invocation" : [
        { "process-dependency" : 0 }
    ]
}
]
```

10. IANA Considerations

In any given serialisation of this approach, several registries will be required for:

- Standard Commands
- Standard Parameters

This document requires no action from IANA.

11. Security Considerations

This document describes the distribution of firmware updates and the invocation of complex behaviours on a device. As such, the contents of a document following the described approach to updates MUST be authenticated as described in [Section 7](#). A more detailed discussion about security can be found in the architecture document [\[Architecture\]](#).

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

12.2. Informative References

[Architecture]
Moran, B., "A Firmware Update Architecture for Internet of Things Devices", July 2018, <<https://tools.ietf.org/html/draft-ietf-suit-architecture-02>>.

[Information]

Moran, B., "Firmware Updates for Internet of Things Devices - An Information Model for Manifests", July 2018, <<https://tools.ietf.org/html/draft-ietf-suit-information-model-02>>.

12.3. URIs

- [1] <mailto:suit@ietf.org>
- [2] <https://www1.ietf.org/mailman/listinfo/suit>
- [3] <https://www.ietf.org/mail-archive/web/suit/current/index.html>

Appendix A. Mailing List Information

The discussion list for this document is located at the e-mail address suit@ietf.org [1]. Information on the group and information on how to subscribe to the list is at <https://www1.ietf.org/mailman/listinfo/suit> [2]

Archives of the list can be found at: <https://www.ietf.org/mail-archive/web/suit/current/index.html> [3]

Authors' Addresses

Brendan Moran
ARM Limited

E-Mail: Brendan.Moran@arm.com

Tony Ibbs
ARM Limited

E-Mail: Tony.Ibbs@arm.com

George Psimenos
ARM Limited

