COOPERATION

DISTRIBUTED MILS FOR DEPENDABLE INFORMATION AND COMMUNICATION INFRASTRUCTURES

Project Number 318772

D5.2 Configuration Compiler

Version 1.1
4 November 2014
Final

Public Distribution

The Open Group, Université Joseph Fourier

Project Partners: **Fondazione Bruno Kessler**, **fortiss**, **Frequentis**, **LynuxWorks**, **The Open Group**, **RWTH Aachen University**, **TTTech**, **Université Joseph Fourier**, **University of York**

Project Partner Contact Information

Fondazione Bruno Kessler Alessandro Ciamatti Via Sommarive 18 38123 Trento, Italy Tel: +39 0461 314320 Fax: +39 0461 314591 E-mail: cimatti@fbk.eu	**fortiss** Harald Ruess Guerickestrasse 25 80805 Munich, Germany Tel: +49 89 36035 22 0 Fax: +49 89 36035 22 50 E-mail: ruess@fortiss.org
Frequentis Wolfgang Kampichler Innovationsstrasse 1 1100 Vienna, Austria Tel: +43 664 60 850 2775 Fax: +43 1 811 50 77 2775 E-mail: wolfgang.kampichler@frequentis.com	**LynuxWorks** Yuri Bakalov Rue Pierre Curie 38 78210 Saint-Cyr-l'Ecole, France Tel: +33 1 30 85 06 00 Fax: +33 1 30 85 06 06 E-mail: ybakalov@lnxw.com
RWTH Aachen University Joost-Pieter Katoen Ahornstrasse 55 D-52074 Aachen, Germany Tel: +49 241 8021200 Fax: +49 241 8022217 E-mail: katoen@cs.rwth-aachen.de	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 894 5845 E-mail: s.hansen@opengroup.org
TTTech Wilfried Steiner Schonbrunner Strasse 7 1040 Vienna, Austria Tel: +43 1 5853434 983 Fax: +43 1 585 65 38 5090 E-mail: wilfried.steiner@tttech.com	**Université Joseph Fourier** Saddek Bensalem Avenue de Vignate 2 38610 Gieres, France Tel: +33 4 56 52 03 71 Fax: +33 4 56 03 44 E-mail: saddek.bensalem@imag.fr
University of York Tim Kelly Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325477 Fax: +44 7976 889 545 E-mail: tim.kelly@cs.york.ac.uk	

Contents

List of Figures

Document Control

Version	Status	Date
0.1	Doc outline, Exec Summary	16 March 2014
0.2	Initial text added	2 April 2014
0.3	Draft	14 October 2014
0.4	Draft for review	28 October 2014
1.0	Complete version	31 October 2014
1.1	Final version	4 November 2014

Executive Summary

This document presents the requirements, approach, theory, design, and implementation description of the *Distributed MILS Platform Configuration Compiler* (DMPCC), the configuration automation backend of the Distributed MILS (D-MILS) tool chain developed as part of the D-MILS Project.

The DMPCC begins with a declarative high-level system description expressed in the BIP language. This input is provided by a transformation procedure developed in WP3 that translates the MILS-AADL language into BIP. The DMPCC (or simply *configuration compiler*) performs model refinement downward through a hierarchy of abstract models that are increasingly detailed in terms of MILS implementation conventions and deployment choices and options. The refinement ends with a model that captures low-level configuration descriptions that can then be more or less directly transliterated by adapters into the languages of the initialization functions of the MILS foundational components comprising the D-MILS platform.

The refinement process is a search for viable D-MILS platform configuration alternatives and the selection of optimal ones, possibly involving user interaction. Configuration generation and evaluation is guided by constraints that are supplied either as features or annotations of the high-level model, or declared as an inventory of available hardware of the target D-MILS platform, and by knowledge and heuristics encoded in knowledge sources used within the DMPCC.

The DMPCC together with the configuration translation tools provided by the individual foundational components form the *configuration plane* for D-MILS systems supported by this tool chain. In the present D-MILS Project those components are instances of a LynxSecure Separation Kernel from LynuxWorks, and a MILS Networking System supporting a TTEthernet network from TTTech.

1 Introduction

This document, and the software it describes, constitute Deliverable D 5.2 of WP 5 (*Distributed MILS Platform Configuration Compiler*) of the EU FP7 *Distributed MILS for Dependable Information and Communication Infrastructures* (D-MILS) Project; Project Number 318772.

Distributed MILS (D-MILS) is an extended version of MILS that permits a MILS policy architecture to be distributed over a networked collection of MILS nodes.

Key high-level objectives of the D-MILS Project are:

- to provide a capability for high-level specification in declarative languages,
- automated support for architecture, design, and implementation artifacts,
- an integrated tool chain and process to generate low-level configurations from high-level system specifications through a series of provably correct transformations,
- to seamlessly integrate the components of a distributed system so that the high-level architectural view of the system may be separated from the details of its distribution over physical resources, compositional verification of desired properties, and,
- to provide integrated assurance for compositional system certification through an automated framework that uses evidence and correctness arguments from every phase of the system development and deployment.

The distributed MILS platform configuration compiler (DMPCC) is the part of the integrated tool chain that generates configurations, achieving a seamless integration of the components of the D-MILS system, and feeds back assurance information concerning the implementation artifacts to the compositional verification and compositional assurance activities.

1.1 Work Package 5 and its relation to other activities

The objectives of Work Package 5 are to:

- Develop the back-end of the Distributed MILS tool chain. This back-end is a compiler from the intermediate representation to the low-level input languages of the configuration tools of concrete target component implementations. This compiler is responsible for:
  - Semantics preserving transformations of the high-level constructs to the low-level resource configurations,
  - Satisfying constraints imposed by the target configuration tools and by the actual available resources.
- Develop generic ways of configuring TTEthernet configurations for Distributed MILS
- Specify the separation kernel configuration target and TTEthernet configuration target interfaces.
- Support incremental configuration by parties having different roles, e.g., architect, developer, integrator, deployment site administrator.

Work Package 5 is closely coupled with Work Package 3, as it utilizes the semantics-preserving transformations developed in that work package to get an internal representation of the MILS-AADL

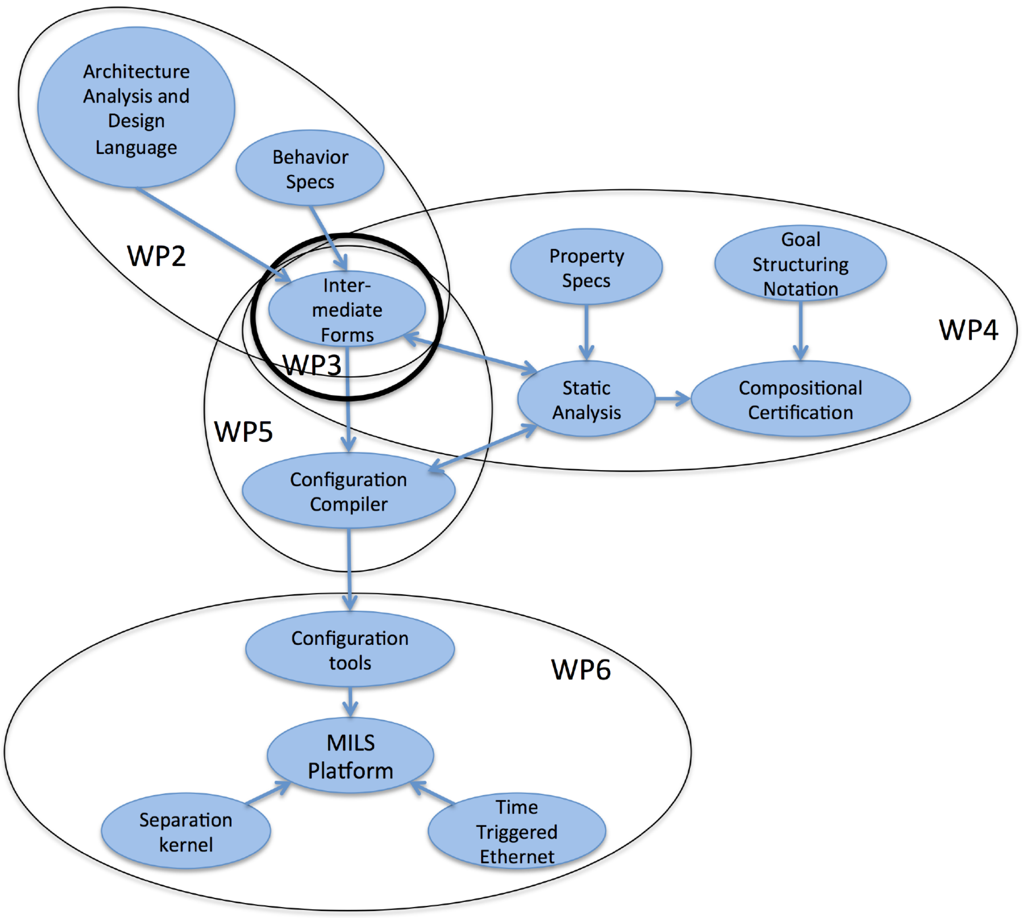

Figure 1: D-MILS Work Packages

system model. Figure 1 illustrates the central rôles of WP 3 and WP 5 in the D-MILS Project, and reflects also WP 5's close internal relationship with Work Packages 2 and 4. It interacts with and produces a product used by the D-MILS Platform implemented in Work Package 6. The interfaces described in Task T 5.1 and Task T 5.2, and documented in deliverable D 5.1, are the languages of the MILS foundational components targeted by the present implementation of the DMPCC.

Figure 2 illustrates the D-MILS Platform Configuration Compiler in the context of the overall D-MILS tool chain.

It should be noted that the version of the Distributed MILS Platform Configuration Compiler developed as part of this project targets a subset of the foundational components generally defined as potential components of the MILS platform. The D-MILS Platform in this project consists of a separation kernel and a MILS Network Subsystem (MNS) that includes support for Time-Triggered Ethernet (TTEthernet). It should be taken into consideration that future versions of the DMPCC should support all of the foundational components defined in the MILS Platform Specification [?], which include MILS Console Subsystem (MCS), MILS File Subsystem (MFS), MILS Audit Subsystem (MAS), and MILS Extended Attributes (MEA). Furthermore, in the future it is anticipated that "distributed MILS" will be considered part of MILS (as indeed it was initially defined), and thus

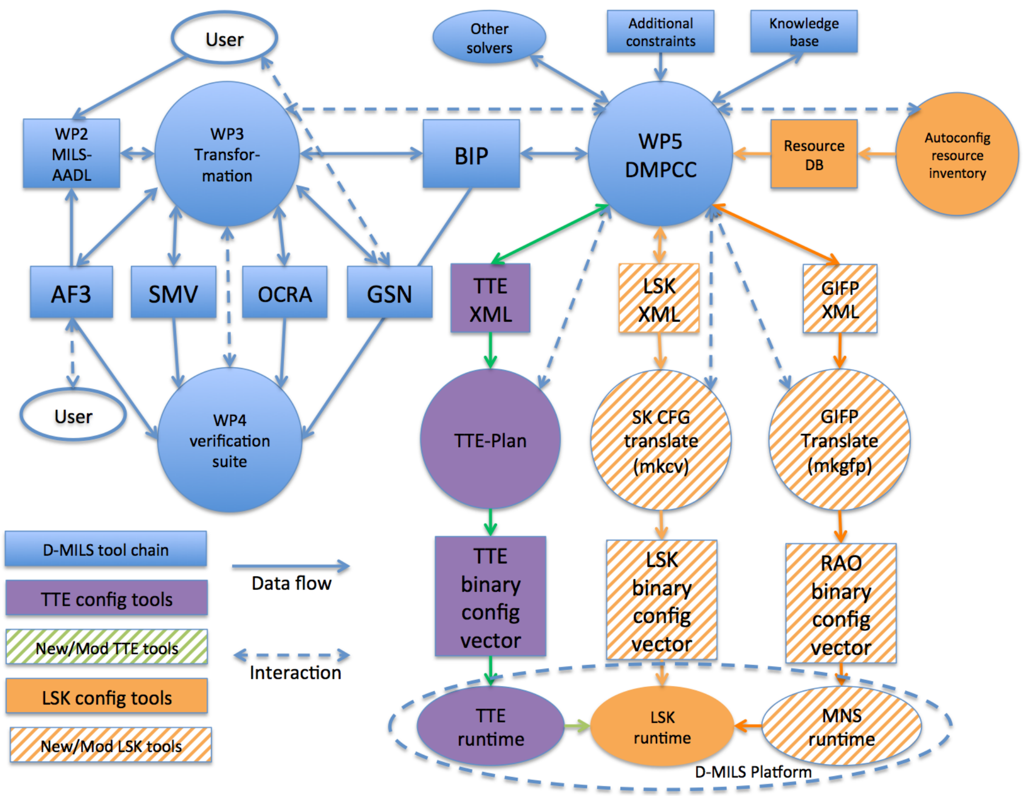

Figure 2: D-MILS Platform Configuration Compiler in tool chain context

the DMPCC will be considered simply as the MILS platform configuration compiler (MPCC), and therefore it is referred to as such equally as DMPCC on this project.

1.2 Overview of Requirements

This document elaborates derived requirements for the Distributed MILS Platform Configuration Compiler and explains how the requirements will be met.

Section **??** of *D1.3 Requirements for distributed MILS technology* [**?**] explains and enumerates the technology requirements for the D-MILS Configuration Compiler. For the convenience of the reader the D1.3 requirements are reproduced in Appendix A of the present document.

This document adds derived requirements based on, and in some cases an interpretation of, those of D1.3. The following is a summary list of the requirements described in more detail in Section 2:

1. **Interfaces**
   - D-MILS System Model from MILS-AADL via BIP
   - Configuration(s) in neutral format
   - Resource constraints
   - other constraints
   - user interaction

- external solvers

2. **Open Formats** – System Model as BIP representation of MILS-AADL, D-MILS Platform Model in neutral format, and other user and tool interactions according to open design specifications.
3. **Support for Proprietary Formats via Adapter Plug-Ins**
4. **Solve for Unspecified or Underspecified values**
5. **Provide multiple alternative solutions**
6. **Provide diagnostics for troublesome inputs**
7. **Support user interaction to guide search**
8. **Support for user-supplied "What If" values to override those specified in the System Model**
9. **Support for rating candidate configurations**
10. **Support for ranking multiple candidate configurations**
11. **Provide useful diagnostics on failure to find a satisfying configuration solution**
12. **Assure that provided configuration solutions are viable for the target platform**
13. **Assure that provided configuration solutions preserve semantic fidelity with the MILS-AADL System Model**

1.3 The MPCC as Open Technology

Two of the important principles that guide the approach to the design and implementation of the MPCC are:

- **"Community Source" implementation** – In support of dissemination and adoption of the D-MILS technology, the MPCC should have an open source implementation that can be maintained and further developed under the control of its community of users.
- **Neutrality of target platform interfaces** – In support of the general objective of the MILS Initiative to foster a marketplace of compatible MILS components, the MPCC should generate output that is easily adaptable to different implementations of the MILS foundational components.

Because of the need to keep the function of the MPCC target neutral, we have adopted the use of an *adapter*. The adapter converts from a neutral output of the MPCC's *configuration process* to a target-specific configuration vector syntax. Target-specific (and perhaps, proprietary) tools then perform the configuration vector conversion service from configuration vector syntax to a target-specific binary configuration vector suitable for load and consumption by component's initialization function.

Any proprietary information of a target's external configuration vector representation should be encapsulated in its adapter. Both the neutral and the target-specific forms may be represented externally in XML, since it is a common representation, though by different XML schemas. The adapters provide a way of insulating and decoupling changes to the neutral format from vendor updates to the individual target formats, which are expected to naturally happen independently.

It is expected that the neutral form produced by the MPCC will have more or less comparable detail to that of commercial partitioning and separation kernels. A future dissemination activity could be to instigate an industry working group to establish a standard for the external configuration language. That will take some time. The end result of such a process could serve as the "neutral format" in a future version of the MPCC. If the vendors adopted it as the input for their binary vector generation tools, then the needed adapters would be very simple, serving mostly to support different versions or target-specific extensions.

Per the MILS DCI and tailored D-MILS MPCC requirements, MPCC target adapters should perform invertible transformations, as should the entire MPCC, a requirement that exists to support validation and diagnostics. The target component is assumed to already provide an invertible semantics-preserving conversion and validation capability such as the required by the SKPP [**?**]: that is, one that "translates human-readable (e.g. ASCII) representations of configuration vectors into machine-readable (e.g. binary) format." And, "The configuration vector generation and validation capability shall be able to convert the configuration vectors from a human-readable form into a machine-readable form, and vice versa, such that the semantics of the data are preserved."

An adapter can be implemented by the target developer using any methods deemed appropriate to get from the neutral form to the target-specific form. On our project the MPCC team will help write the first adapters from neutral form to the XML representation used by LynxSecure's configuration tool and TTEthernet's configuration tool, and will provide a guide for other developers.

The neutral form should be sufficiently detailed to support future verification of the correctness of the configuration data down to the external representation handed off to the platform component configuration tools. The neutral form will likely have an XML-based external representation, and the MPCC must be able to import this format to support iterative incremental configuration refinement, and the ability to trace back to the higher abstraction levels and to MILS-AADL through the interfaces of the MPCC to the other components of the D-MILS tool chain.

1.4 Document Organisation

The remainder of this document is organised as follows:

Section 2 – Objectives and requirements

Section 3 – Concept of operation

Section 4 – Architecture Description

Section 5 – Module description, design and implementation

Section 6 – Assurance considerations

Appendix A – D-MILS configuration compiler requirements from D1.3

Appendix B – Internal representations

Appendix C – External representations

Appendix D – Command line interface

Appendix E – Examples

2 Objectives and Requirements

The following derived technical requirements for the D-MILS Platform Configuration Compiler are based on a tailored subset of the Multiple Independent Levels of Security (MILS) Delivery, Configuration, and Initialization (DCI) Requirements [**?**, **?**], which are summarized in the next section. The D-MILS requirements related to configuration are presented in the form of high-level Objectives, General Requirements, Configuration Requirements, and Distributed MILS Platform Configuration Compiler Requirements.

2.1 MILS Delivery, Configuration, and Initialization (DCI) Requirements

The MILS platform is composed of foundational components that work together to provide isolation and information flow control among the resources that each component exports. One may imagine that the challenge for the composition of these components is to have their interfaces, behaviors, and assurances combine seamlessly even though they may have been developed by different vendors, and indeed that is a challenge.

In addition however, to be successfully evaluated each of these components must provide trusted delivery, configuration and initialization functions [**?**, **?**] that perform their services entirely before[1] the components are actually executed,[2] and these functions, too, must compose seamlessly. It is essential to the technical and commercial objectives of MILS that they do. This is a substantial challenge that cannot be solved without deliberate forethought. Consequently, the problem has been investigated and documented, including a detailed set of MILS DCI requirements drawn up to guide design and implementation efforts.

In *Delivery, Configuration and Initialization of MILS Components and Integrations* [**?**], objectives and requirements for MILS DCI are enumerated. There are four high-level objectives for MILS DCI, nineteen general requirements, five delivery requirements, nine configuration requirements, two load requirements, and six initialization requirements.

In a static MILS context, MILS DCI must provide the *pre-execution-time composition* of the delivery, configuration, and initialization functions, and it must support the composition and incremental construction of configuration information in the supply chain, and the integrity of configuration information as it is constructed, delivered, and used to establish the initial state of a system. Moreover, in a dynamic MILS context, MILS DCI must additionally provide execution-time composition of configuration and initialization functions with MILS platform and MILS system operation. A design that meets all of these requirements *simultaneously* has yet to be developed. It is anticipated that incremental progress will occur as approaches to the separate implementation of the delivery, configuration, and initialization requirements are developed.

[1]Before, in *static* MILS. *Dynamic* MILS requires configuration and initialization to also be capable of being performed during operation.

[2] More specifically, before the components jointly enter operational mode, because the DCI functions of the components are also executed jointly at times.

2.2 D-MILS Adaptation of MILS DCI Requirements

The relevant D-MILS requirements are presented and described relative to the broader context of MILS DCI. The numbering of objectives, general requirements, and configuration requirements for D-MILS presented in the following subsections corresponds to that in MILS DCI so that one can readily relate D-MILS configuration to MILS DCI.

D-MILS levies configuration requirements and initialization requirements, but no delivery requirements. Trusted delivery is outside the scope of the D-MILS Project. Initialization of the separation kernel and the time-triggered network are already handled by the commercial LynxSecure and TTEthernet products. Some derived requirements for initialization of the D-MILS platform and the D-MILS system are presented.

Also, presented in Section 2.3 are requirements for the Distributed MILS Platform Configuration Compiler, which are new to D-MILS. They are derived from D-MILS technology requirements [?] and applicable MILS DCI requirements. The D-MILS technology requirements for the configuration compiler from [?] are reproduced in Appendix A for the convenience of the reader.

2.2.1 Objectives

These are abbreviated versions of the MILS DCI objectives.

- **O1 – Configuration Correctness** (best effort)
- **O2 – Maintenance of Configuration Integrity** *Prevention of unauthorized modification* is not a D-MILS objective. This objective is limited to well-formedness of configuration information.
- **O4 – Initial Secure State Transition** This is the ultimate step following initialization in which the trusted computing base transitions to operational mode.

2.2.2 General Requirements

These are abbreviated versions of the MILS DCI general requirements.

- **G1 – Establish Initial Secure State (ISS) of Composite** It should be borne in mind that establishing initial secure state is *the overarching MILS DCI requirement* but it is out of scope for D-MILS.
- **G2 – Accuracy of Initial Secure State** The comprehensive controls required for MILS DCI are not strictly required for D-MILS. Best effort should be applied to provide confidence that the use of the configuration information by the initialization functions results in the intended state.
- **G3 – Integrity of Entire Configuration** The guarantee of integrity and prevention of unauthorized modification required for MILS DCI are not strictly required for D-MILS. It should be borne in mind, however, that D-MILS may be applied in applications where this is a requirement.
- **G4 – Semantics of Configuration Information** Provide an unambiguous and demonstrably consistent set of definitions for the configuration items.
- **G10 – Evidence and Assurance of Dependable and Robust MILS DCI** This requirement is not strict for D-MILS, simply requiring evidence to support an explicit assurance case.

2.2.3 D-MILS Configuration Requirements

These are abbreviated versions of the MILS DCI configuration requirements.

- **C1 – Configuration Change –** D-MILS shall support incremental configuration for incremental composition. (A limited version of the MILS DCI requirement.)
- **C2 – Configuration Tool Design –** Shall provide configuration vector generation and validation capability and documentation. Shall provide human-readable form and internal form. Shall provide conversion between human-readable form and internal form, and vice versa. (A limited version of the MILS DCI requirement which also requires integrity seals on generated configuration vectors.)
- **C3 – Support for Static Configuration –** Shall support static (offline) configuration. (A limited version of the MILS DCI requirement which also requires dynamic configuration.)
- **C4 – Support for Incremental Configuration –** Shall provide capability to incrementally construct configuration vector or vectors over an indefinite number of configuration sessions. Shall provide a test for *perfected configuration*, i.e. configuration suitable as input for initialization.
- **C5 – Support for sets of configuration attributes –** Shall be flexible with respect to configuration attributes supported (which are component dependent). (A limited version of the MILS DCI requirement which requires support for arbitrary sets of configuration attributes.)
- **C7 – Support for intra-component configuration constraints –** Shall support application of constraints to establishment of configuration attribute values.
- **C8 – Support for inter-component configuration constraints –** Shall support application of constraints to establishment of configuration attribute values across component instances to achieve compositional consistency or desired compositional semantics.
- **C9 – Invertible Configuration Representation Transformation –** Transformations among human-readable and other representations shall be invertible.

2.2.4 D-MILS Initialization Requirements

These are abbreviated versions of the MILS DCI initialization requirements.

- **I1 – Establishment of Secure State –** The D-MILS platform shall be established in a secure state as defined by the configuration vector. The configuration data shall provide the Partitioned Information Flow Policy.
- **I2 – Trusted Initialization –** The D-MILS initialization function shall establish the D-MILS platform in a secure state consistent with the configuration vector. Shall verify integrity of code and data. Shall establish a security domain(s) for the D-MILS platform components and their configuration data. Shall not interact with the D-MILS platform after initialization.
- **I3 – Installation, Generation and Start-Up Procedures –** Procedures shall be documented.
- **I4 – Configuration Verification –** Verify the configuration information to be used. Verify the configuration information is available, complete, and well-formed before irreversible changes are made to the D-MILS platform state.
- **I5 – Initial Configuration Vector Selection –** At least one configuration vector shall be provided, and exactly one selected for initialization.

- **I6 – Initialization Sequence –** All D-MILS platform components and other trusted components shall be initialized before non-trusted components. Trusted components shall be initialized in a sequence determined by initialization sequence constraints specified in the configuration vector.

2.2.5 D-MILS Non-Requirements

For completeness of perspective on the MILS DCI context we list the MILS DCI objectives and requirements not applied to D-MILS.

The following MILS DCI Objectives are out of scope of D-MILS. These are abbreviated versions.

- **O3 – Correct Establishment of Operational Configuration** This is not a strict D-MILS objective because it is outside the scope of D-MILS. It should be borne in mind, however, that D-MILS may be applied in applications where this is an objective.

The following MILS DCI General Requirements are out of scope of D-MILS. These are abbreviated versions.

- *G5 – A priori Strict DCI Interoperability* (not required)
- *G6 – Integrated Modular DCI Solution* (not required)
- *G7 – Compositional DCI Solution* (not required)
- *G8 – Common DCI Realisation or Interoperable Realisations* (not required)
- *G9 – Secure Mobile DCI Execution Environment* (not required)
- *G11 – Robustness in the Face of Operational Maintenance* (not required)
- *G12 – Exclusively MILS DCI Standard Features* (not required)
- *G13 – Development Assurance* (not required)
- *G14 – Guidance Documents* (not required)
- *G15 – Life Cycle Support* (not required)
- *G16 – Ratings Maintenance* (not required)
- *G17 – Platform Assurance* (not required)
- *G18 – Testing* (not required)
- *G19 – Vulnerability Assessment* (not required)

The following MILS Configuration Requirements are out of scope of D-MILS. These are abbreviated versions.

- *C6 – Configuration policy specification and enforcement* – MILS DCI requires configuration functions to enforce a fine-grained configuration policy, where each configuration attribute or collection of attributes may be governed by a configuration policy, which can specify the principals, circumstances, and permissible values or conditions for the establishment of attribute bindings.

2.3 Requirements for Distributed MILS Platform Configuration Compiler

The MPCC Requirements were summarized in the Introduction. A more complete description is provided here:

- **MPCC-1 – Interfaces** The MPCC shall provide interfaces designated as *input*, *output*, or *partial*. Input interfaces require an input to be present. Output interfaces always provide an output. An interface that is designated partial may serve as input, output, or both. When the value presented at a partial interface is instantiated and a constant, it is treated as input; when the value is uninstantiated or partially instantiated, the instantiated part will be treated as input and the uninstantiated part may be used by the MPCC for output.
    - D-MILS System Model from MILS-AADL via BIP (*input* or *partial*)
    - Configuration(s) in neutral format (*partial*)
    - Resource constraints (*partial*)
    - other constraints (*partial*)
    - user interaction (*partial*)
    - external solvers (*partial*)
- **MPCC-2 – Open Formats –** MPCC shall provide open specification of its input and output formats: System Model: BIP representation of MILS-AADL; D-MILS Platform Model: neutral format per MPCC design specification; and other user and tool interactions according to open design specifications.
- **MPCC-3 – Proprietary format adapters –** The MPCC shall provide an interface for developer provided plug-in adapters to convert from the MPCC's neutral configuration format to proprietary formats.
- **MPCC-4 – Solve for unspecified or underspecified input values** The MPCC shall be able to provide values for unspecified or underspecified values on interfaces designated as partial.
- **MPCC-5 – Multiple configuration solutions –** The MPCC shall be able to provide multiple alternative solutions for unspecified or underspecified terms on designated partial interfaces.
- **MPCC-6 – Diagnose bad inputs –** The MPCC shall be able to provide diagnostic information on malformed input to designated input interfaces and on provided partial inputs to interfaces designated partial interfaces.
- **MPCC-7 – User interaction –** If not invoked in silent mode, the MPCC shall be able to interact with a user to guide the search for configuration solutions.
- **MPCC-8 – What If –** The MPCC shall be able to employ user-supplied override values that differ from those specified in the MILS-AADL system model to explore other deployment alternatives.
- **MPCC-9 – Solution rating –** The MPCC shall be able to evaluate configuration solutions according to user-specified criteria and provide a quantitative rating.
- **MPCC-10 – Solution ranking –** The MPCC shall be able to maintain a ranking of multiple configuration solutions according to evaluation rating results.
- **MPCC-11 – Failure to find solution –** The MPCC shall be able to report failure to find a satisfying configuration solution accompanied by a useful diagnostic report.
- **MPCC-12 – Viable configuration solutions –** The MPCC shall assure that provided configuration solutions are viable for the target D-MILS platform.

- **MPCC-13 – Semantic Correctness –** The MPCC shall assure that provided configuration solutions preserve semantic fidelity with the MILS-AADL System Model under reasonable human interpretations and platform constraints.

2.4 Design and Implementation Imperatives

The high-level requirements for D-MILS technology are presented in D 1.3 [**?**]. The objectives, general requirements, configuration requirements, and MPCC requirements presented above derive from D 1.3 and from contextual requirements represented by existing MILS objectives and requirements for delivery, configuration, and initialization (DCI), laying groundwork for future evolution of D-MILS proceeding in harmony with overarching MILS target outcomes.

The following imperatives for the design and implementation of the D-MILS Platform Configuration Compiler (MPCC) provide guidance on how the objectives and requirements above are to be realised. They take into consideration key strategic objectives both of the D-MILS Project, *viz.* dissemination and exploitation, ease of adoption, features and attributes to handle a reasonably large class of realistic use cases; and of the MILS initiative, seeking coherence with established and emerging MILS concepts and standards, while conservatively extending the scope and capability of MILS.

1. **Open design and implementation** – The MPCC will contain no proprietary code or data within its functional perimeter.

2. **Target neutral** – The MPCC will generate D-MILS platform configurations in a target-neutral format.

3. **Configuration assurance** – The MPCC should support strong assurance of configuration vector correctness.

4. **MILS compatible** – The MPCC should be compatible with existing MILS standards and conventions, and support future development of MILS delivery, configuration, and initialization, and integrate with dynamic reconfiguration of the MILS platform.

2.4.1 Open design and implementation

In keeping with the European Commission's emphasis on effective dissemination and exploitation, is intended that the D-MILS technology be easily adopted and applied by as broad a community of developers and industries as possible. The tool chain is intended to be as self contained as possible with respect to its core functions and not be dependent on proprietary components for those functions. Other projects should be able to adapt it and other research extend it. To achieve the independence from proprietary information while being able to utilize commercial MILS components, the MPCC provides for plug-ins *at its functional perimeter* that can be implemented by component developers to isolate proprietary conversions.

2.4.2 Target neutral

Further to the preceding imperative, the configurations as produced by the MPCC should include configuration information for distinct MILS platform components. Sufficient information should be available for use by the configuration functions of multiple components in order to achieve as seamless an integration of those components in the configuration realm as they are intended to have in the operational realm. This topic has been elaborated in prior MILS DCI research. The configuration information, while generic, should be of sufficient detail so as not to assume more capability on the part of individual component configuration mechanisms beyond a straightforward conversion of syntax (to be handled by an adapter plug-in to the MPCC) and conversion by the component specific tool to binary configuration vector format.

2.4.3 Configuration assurance

The MPCC should provide the level of detailed information necessary so that strong assurance evidence of configuration vector correctness, in the form of correspondence proofs down to the physical resource level, can be performed within the D-MILS tool chain and linked in to the GSN assurance case. This approach places little complexity burden upon the commercial components' configuration tools, facilitating the correctness proofs of those tools that are necessary to complete the comprehensive correspondence assurance case. In this way some of the most burdensome proofs can be done once-and-for-all for all targets.

More on the issue of assurance considerations may be found in Section 6.

2.4.4 MILS compatible

Numerous concepts and definitions are already established within the MILS community. Additional emerging standards are being developed that are needed for the achievement of long held community goals for MILS and the growth of MILS to be capable of handling more diverse use cases. Consequently, although not all of these considerations might be germane if D-MILS were being done in a vacuum, they are relevant because of the bigger picture and future plans to build upon Distributed MILS and to further augment the capabilities of MILS.

3 Concept of Operation

3.1 Input and Output Forms

The MPCC requires as input a model of the subject system in the BIP language. A front-end to the MPCC translates a user-created MILS-AADL model into BIP. Another front-end translates BIP to a Prolog clause representation, which can be directly read by Prolog.

The MPCC maintains several intermediate forms as it elaborates and completes a configuration that fulfills the input requirements while satisfying constraints imposed by the target. The resulting system configuration is expressed in a comprehensive common language called MILS configuration normal form. Plug-in modules to the MPCC, referred to as target adapters, translate the MILS configuration-normal-form (MCNF) into target-specific formats.

In the future, project members will seek to use the resulting version of the MCNF language to initiate an effort to establish an industry standard. To the extent that such a standard is adopted, target developers may be able to simplify the transformation needed to get from MCNF to a target-specific format, thus simplifying the associated adapter.

3.2 Designer / Deployer Interactions

Though it would be desirable to have the synthesis of a satisfying configuration be entirely automated and require no human interaction, this may not be possible in some cases. As a configuration is elaborated details must be supplied. To the extent these can be provided by predetermined rules no interaction is required. However, not all such situations can be anticipated during the initial development of the MPCC. We need to gain experience to learn how much automation can be achieved.

There must be mechanisms in place for the user to provide guidance to the MPCC. As such situations arise, new rules can be added to the knowledge base of the MPCC. With time and growth of the knowledge base the incidence of such events will decrease. However, it is not expected that the need for human judgement can be entirely eliminated by automation, as it will be necessary to evaluate alternative configuration choices in light of considerations that may not reasonably be formulated in the guidance rules on which the MPCC operates. Also, it is expected that some deployment parameters may need to be changed when, for example, the physical characteristics of the platform becomes known at deployment time, or requires changes at operation time.

Thus, there will always be a place for some human interaction with the MPCC and it is expected that the effectiveness of the interaction mechanisms will increase as the required interactions are better understood.

3.3 Intermediate Configuration Models

A considerable range of levels of abstraction may need to be bridged by the MPCC going from the system model to the target configuration(s). Depending upon the specificity of the system model, and aspects that may have been left underspecified, there will be more or less of a gap.

To make the process more manageable to the implementors of the MPCC and to make it more comprehensible to those who need to understand it in order to provide assurance arguments and evidence, it is expedient to define a series of intermediate models between the input and the output.

During requirements analysis, at least the following intermediate hierarchical models are considered to be useful:

- internal system model
- exported resource model
- MILS platform model
- physical resource model

In addition, the following, not necessarily hierarchical, models appear to be useful:

- hardware-specific platform model(s)
- software component-imposed constraints
- physical allocation constraints
- network model
- scheduling model(s)

More on this subject may be found in Appendix A, Distributed MILS CC Requirements.

3.4 Creating and Refining the Current Configuration

The configuration being constructed is referred to as the current configuration. The current configuration is iteratively refined by applying rewriting rules and solving constraint satisfaction problems. During intermediate stages of this process, the well-formedness of the configuration is maintained even though the configuration may be incomplete.

The representation of the current configuration contains decided, as yet undecided, and perhaps partially decided details. Undecided details are represented by uninstantiated placeholders, and partially decided details as partially instantiated placeholders within the configuration. As the configuration process proceeds more and more of the placeholders are replaced by decided details, unless it becomes necessary to undo decisions, by backtracking, to recover from a discovery that the current choices lead to an untenable or non-optimal configuration solution. Even so, the well-formedness of the configuration can be checked even while it contains placeholders because of typing of the placeholders.

When the configuration is fully instantiated, well-formed, satisfies all essential assessment tests, and can be transformed into a concrete configuration file for some target, it is said to be a perfected configuration.

3.5 Search: Generation and Evaluation of Alternative Configurations

The control of the configuration creation process is conducted as a heuristics-guided search for a fully-instantiated configuration solution that satisfies the requirements stated in the system model and meets the constraints imposed by the target platform.

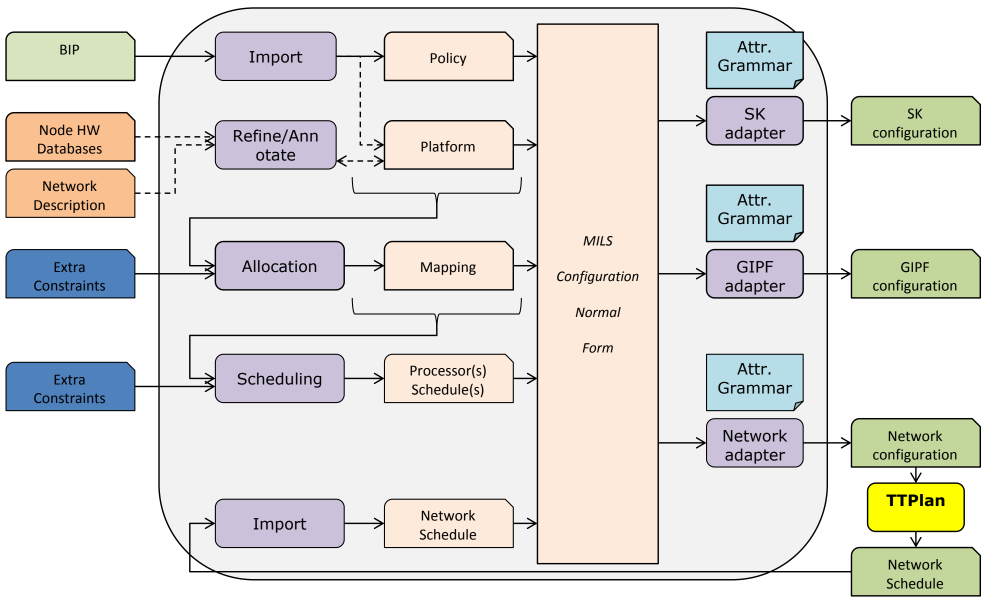

Figure 3: D-MILS Platform Configuration Compiler Flow

The flow of the configuration search process is depicted in Figure 3.

The compiler front-end constructs the configuration. It essentially implements an incremental search for a perfected configuration (if one exists) defined as the combination of several configuration artifacts that are:

- the mapping of components (subjects/objects) to nodes,
- the schedules for nodes handling multiple subjects and
- the schedule of the communication network.

The search for the mapping and for schedules is *stratified*, that is, the former has complete precedence over the second. The scheduling of subjects within nodes as well as the definition and scheduling of virtual links within the network can actually take place only when the mapping of nodes and virtual links is fully determined. Feedback mechanisms are nevertheless provided to constrain the allocation, whenever the scheduling fails. This way, whenever backtracking to find another set of mappings, the new candidate solutions will never fail scheduling again for exactly the same reason.

An arbitrary number of assessment tests may be applied to candidate choices, and/or future consequences of such choices, to decide among choices. Assessment tests result in a rating of the configuration choice. The choices may be ordered according to various criteria, constituting a ranking.

Finally, the compiler back-end produces the target-specific configurations. It extracts from the configuration normal form, which is target agnostic, the specific configuration files for the different separation kernels, MILS components, and for the communication network.

3.6 Knowledge Application

Arbitrary knowledge about the computational environment or about target systems or their configuration tools may need to be applied as configuration choices are considered. Such information should be encoded as Prolog rules that are consulted when the configuration process is being performed.

3.7 Internal and External Configuration Representation

Here we describe in general terms the representations used within the MPCC, and output from the MPCC. Details of the concrete representations and examples are contained in Appendices B and C.

3.7.1 Target Adapters

A D-MILS system may be composed from diverse MILS separation kernels and other MILS foundational components. Each must be separately and appropriately configured according to the conventions for the configuration of the component, but also consistently with the overall D-MILS system configuration.

One of the challenges of the back-end of the D-MILS tool chain is the anticipated diversity of target components. There is diversity of both kind and implementation, competing developers providing diverse implementations of the same component kind. The perfected internal model of the system configuration must be *projected* onto the components chosen for a particular D-MILS system deployment. The final syntactic and semantic gap to specific target components and systems is bridged by target-specific adapters that plug into the MPCC.

Target adapters have as their output the specific representations expected by configuration tools of the various target components. As a plug-in to the MPCC, an adapter has access to MPCC-internal data structures. The developer of an adapter has the choice of coding the adapter in the MPCC Prolog programming environment, where access to the internal form of MCNF is available, or the adapter can be a minimal plug-in that invokes an external program that may may access and transform the MCNF in its external XML representation.

A target adapter may be an ad hoc procedure or it may be structured around a pattern we refer to as a configuration generator that can be used for many different target languages. Assurance considerations for target adapters, discussed in Section 6, explains the benefits of configuration generator-based target adapters.

3.7.2 Configuration Generators

Apart from the Prolog clause form of the system model, derived from the BIP model and computed in part by the configuration process, external forms are generally defined by a configuration generator.

In the following, $Model$ is a model instance to be rendered using a configuration generator for a specific target format. $Generator$ is a configuration generator that targets some specific output format. $Sentence$ is a sentence in the language of the target of the configuration generator.

The application of a configuration generator is represented,

$$Model\ |_M\ Generator_T \Rightarrow Sentence_{MT}$$

The symbol $|_M$ indicates composition using an interface specific to the to the $Model$ type that is provided to the $Generator$. The generator may call upon this interface to obtain the resources needed by the configuration and the ground elements that serve as the values of terminal elements of the target language.

For short, the generic pattern is written,

$$M|G \Rightarrow S$$

which indicates the application of a generator to derive a sentence.

The principle is very general and is used in MPCC to generate both internal and external forms. That is, the target of a configuration generator can be the language of an external tool, or it can be an internal form. For example, the MCNF described in the next section is to be defined by a configuration generator. Also, among the generators we have defined is a configuration *generator-generator* (GG) for targets defined by XML schemata.

The pattern,

$$XS_T|GG \Rightarrow G_T$$

indicates the application of the generator-generator to an XML schema (XS) to derive a configuration generator (G) for the target described by the schema. In this case there are two "targets" involved: the model is the internal form of an XML schema for the external target language, the interface is a simple query interface to the schema, the "target" of the generator GG is the internal configuration generator structure, and the generated sentence is a configuration generator for the external target language. This GG will continue to be incrementally augmented to minimize the amount of manual adaptation that must be performed to complete the generated configuration generator for a target. It is expected that this effort will pay dividends in the future.

The resulting configuration generator is, in turn, composed with the internal system model instance of a D-MILS configuration through an interface that is specific to that model type. A configuration generator will also be used to generate from the internal system model the Global Information Flow Policy that is used by LynxSecure and the MNS to configure the inter-node flows of the D-MILS system.

Another configuration generator-generator we will develop is one to generate a configuration generator for the simple command line input to the LynxSecure *autoconfig* tool. Most extant separation kernels use an XML-based language to specify configuration of the kernel, so our XML CGG is expected to be useful in the future for targeting other separation kernels, and also as an example of how to generate in a more automated fashion configuration generators for other targets and rich target languages. However the LynxSecure Separation Kernel being used for D-MILS has the *autoconfig* tool that will accept a higher-level specification, represented as a sequence of parameters on its command line invocation. This enables a fairly simple generator that permits us to bypass the MCNF representation in the prototype implementation.

3.7.3 MILS Configuration Normal Form (MCNF)

The MCNF is a target-neutral format that spans the domains of many kinds of targets, separation kernels, MILS foundational components, MILS operational components, hierarchical system structure, and perhaps in the future extensibility for bespoke components. The MCNF for the internal model is built from a configuration generator for MCNF.

Externally, MCNF can be exported in two file formats: as a file of Prolog clauses that may be directly read back into the MPCC as the Prolog data structures, or as an XML-based representation that can be imported to reconstruct the internal data structures.

An XML schema for MCNF will be developed in the future, permitting the internal structure of MCNF to be transliterated into XML, allowing this as an alternative external representation. The schema can be used by other tools to process the MCNF.

3.8 Invertible Transformations

Transformations within the MPCC and to and from the input and output formats ideally are invertible transformations. This is desirable to support two objectives:

- make it possible to provide meaningful feedback to the user in the context of user-visible representation(s), based on traceability of exceptions and annotations attached to internal representations within the tool chain to user-visible representation(s), and,
- make it possible to provide rigorous correspondence demonstrations throughout the chain of representations to provide assurance that the tool chain preserves the semantics attached to user-visible representations in support of system assurance.

4 Architecture Description

The architecture of the MPCC is described, including identification and purpose of the subsystems and modules.

4.1 Subsystem and Module Decomposition and Dependencies

The MPCC is organized into subsystems that in turn represent groups of modules.

Figure 4 illustrates the subsystems and modules of the MPCC and the high-level interactions among the subsystems and modules.

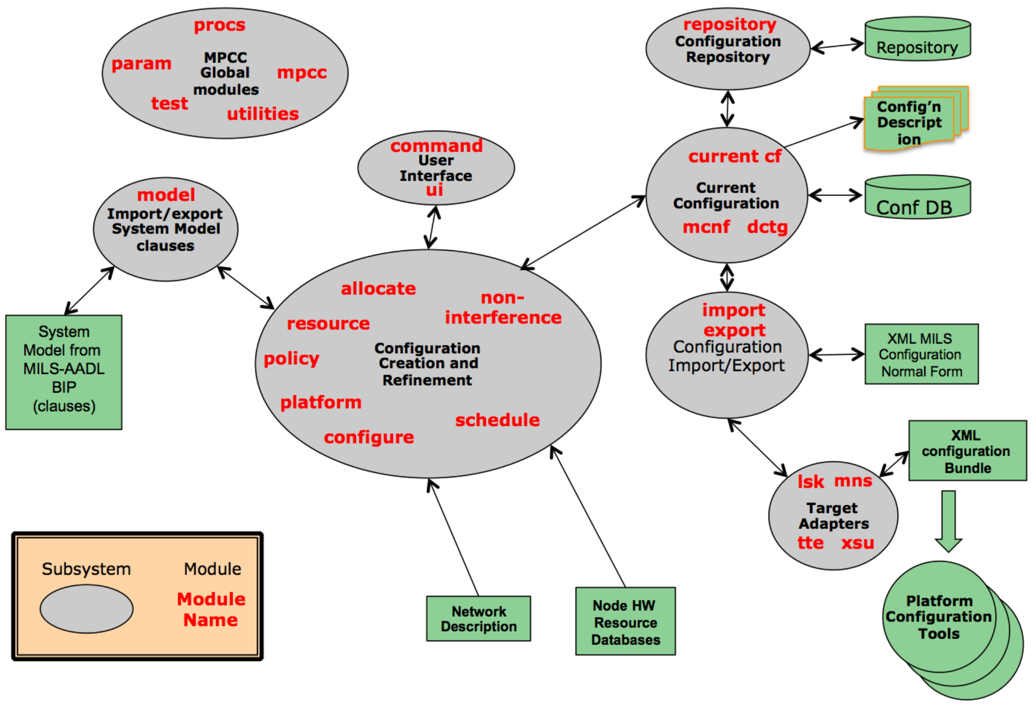

Figure 4: MILS Platform Configuration Compiler Subsystems and Modules

Some of the subsystems contain multiple modules as shown (module names in red).

4.2 System Model Import / Export Subsystem

This subsystem is on the front-end of the MPCC to bring in the high-level system model as defined by the user. The user writes and sees the model in MILS-AADL (or edits a graphical representation that is output from the editor in MILS-AADL) and annotations attached to the MILS-AADL. An external translator translates the MILS-AADL system model into a semantically equivalent model expressed in the BIP language.

4.2.1 BIP Model Import

We defined and implemented a policy extractor from BIP models into Prolog. This module has been realized using the existing BIP frontend. Once the BIP model is parsed, it dumps the policy-related information into a policy term, as described in section 5.12. That is, the policy contains information about the component inetrfaces, their connections, and related annotations for MPCC, if any. All information on component implementation (e.g., data, mode behavior, etc) is ommitted.

4.2.2 Extended Symbol Table

As part of the translation from MILS-AADL to BIP, the external translator builds an *extended symbol table* that is needed to reconstitute the MILS-AADL from the BIP translation. The information contained in the extended symbol table includes not only symbols and values, but also annotations attached to the model by the user or by earlier stages of the D-MILS tool chain. This information is retained so that it is possible to provide the user with a location in the MILS-AADL model associated with exceptions generated during the configuration synthesis process or with annotations representing configuration choices made in the MPCC that may be presented to the user or to the verification system. Information and configuration-related annotations in the extended symbol table associated with the incoming system model may also be used by the MPCC.

4.3 Current Configuration Subsystem

The Current Configuration subsystem holds the data structures that represent the configuration under construction and alternative configuration choices and representations.

The current configuration acts as a *blackboard* upon which modules in the Configuration Creation and Refinement Subsystem operate to construct the configuration to be exported. In addition to the content of the configuration, the blackboard may also contain metadata and control information recognized by primitives in the Creation subsystem and by agents that invoke those primitives.

A collection of predefined access predicates are provided to operate upon the current configuration.

4.4 Configuration Creation and Refinement Subsystem

This subsystem contains the primitives that construct the current configuration and the control flow for that construction process.

4.4.1 Configuration Generation and Instantiation

Configurations contain structural and deployment details that are not provided by the high-level system model. Yet, configurations must be represented in forms that have strict requirements on correctness of form. A configuration is generated from a template called *configuration generator* that provides the appropriate form needed for the particular kind of configuration or configuration section.

Configuration generators represent a well-formed configuration of the kind for which the generator has been created.

When a configuration generator is initially invoked, it establishes a structure that represents an uninstantiated configuration of the type of the generator. This structure is incrementally instantiated as the configuration creation process proceeds.

As the generator is used, the intermediate stages of the configuration development are to remain well-formed according to the generator's specification even though the configuration is not yet complete. It is a well-formed but non-terminal sentential form of the language of configurations defined by the generator. When the configuration is finally completed, it is a terminal sentence in the language, and thus it is (still) a well-formed configuration of the type.

4.4.2 Rating and Ranking

Alternative choices arise during the process of creating a configuration. These choices may be trivial decisions among alternative equally valid parameter values or allocation decisions. They may also represent tradeoffs among competing concerns, that include such factors as physical resource requirements, time, security, safety, risk, etc.

One or more evaluation functions may be provided to be use singly or together to rate alternative configuration choices quantitatively or qualitatively. The result of such ratings allow alternatives to be ranked in terms of suitability according to the vector of rating values generated by the evaluation functions applied to the set of alternatives.

4.5 User Interface Subsystem

The user interface acts on behalf of the user with respect to interactions with the core engine (the Configuration Creation and Refinement Subsystem) and the import and export functions. The user interface is conceptually independent of any specific user interface modality, thus a command line user interface or a graphical user interface could be used. The current user interface is a textual command line interface.

4.6 Solver Subsystem

The solver subsystem includes solvers and interfaces to external solvers of various kinds that may be needed for the completion of a configuration.

4.6.1 Constraint Solver

The constraint solver uses the integrated Prolog Constraint Logic Programming (CLP) package to perform tightly-coupled, frequently-iterated, or straightforward constraint problems that are most easily handled within the MPCC. If more difficult constraint satisfaction problems are posed, it may be necessary to convert the problem into the language of a more powerful external constraint solving tool, to invoke that tool, and to convert the result back into a usable internal form.

4.6.2 Other Solvers

These are external tools used to solve problems that arise while constructing a configuration. Such tools could include components in the D-MILS verification suite, such as SMT solvers, general scheduling support tools, or target-specific scheduling tools.

4.7 Configuration Information Repository Subsystem

The Configuration Information Repository subsystem, or simply "Repository", provides the organization and access to persistent storage of configurations, configuration fragments, configuration generators, and other metadata and intermediate forms of information used during the construction of a current configuration or to save a completed configuration.

The repository provides persistent storage of configurations that are under construction or completed, intermediate artifacts, and storage of configuration generators.

4.8 Configuration Import / Export Subsystem

The Configuration Import / Export subsystem can read and write external representations of configuration information, and in particular such information encoded in an XML format, though possibly other external representations.

4.8.1 XML Import/Export

Target-specific configuration vectors are typically represented in an XML-encoded format. The MCNF can also be stored in an XML-encoded form or a less cluttered form based on the external representation of Prolog clauses.

4.8.2 Target Adapters

Conversions to and from specific representations (XML-based or otherwise) that are used by various target platform components, from and to the internal representations of the MPCC are done by target adapters. Target adapters are plug-in modules developed to particular interfaces provided within the MPCC, whereby the internal structures of the current configuration may be accessed, and requests by the MPCC may be carried out by the adapter. A target adapter may use the current configuration access interface to obtain configuration information that is to be rendered in the language of external target-specific tools, or it may access the configuration information in the external MCNF and translate to the target-specific form, taking only service requests from the MPCC and providing status reports to the MPCC over the plug-in interface.

5 Module Description, Design and Implementation

This section describes the design and implementation of D 5.2, the D-MILS Platform Configuration Compiler.

Within this design and implementation section and in the code itself the software suite is referred to as the *MILS Platform Configuration Compiler*.

The configuration compiler framework being constructed now will go on to be extended in the future with other aspects of MILS, such as dynamic MILS. It implements the infrastructure for future extensions to handle all the anticipated aspects of MILS configuration that lies beyond the scope of the D-MILS project. Hence, it is appropriate that the compiler should be called MILS Platform Configuration Compiler (MPCC), because it will already support single node MILS as a part of distributed MILS, and will support other extensions in the future for trusted delivery, configuration and initialization.

5.1 Implementation Vehicle

The chosen implementation vehicle is Prolog, a language in the logic programming paradigm. It is a powerful symbolic programming language with extensibility and features that support rapid prototyping and exploratory programming. It is particularly well-suited to the implementation of the MPCC. The SWI Prolog implementation is being used, SWI-Prolog [**?**], is available without charge and is quite mature and reasonably well supported. It has a principal developer and maintainer, Jan Wielemaker, and many contributors. Among a rich set of libraries, SWI Prolog has one library that is of particular interest for this project, constraint logic programming over finite domains, denoted CLP(FD) [**?**].

5.2 Subsystems and Modules of the MPCC

The following modules are discussed in this section:

- Allocate Module (`allocate`)
- Command Module (`command`)
- Current Configuration Module (`current_cf`)
- Definite Clause Translation Grammar Module (`dctg`)
- Import/Export Module (`import_export`)
- MPCC Module (`mpcc`)
- Noninterference Analysis Module (`noninterference`)
- Parameters Module (`param`)
- Platform Module (`platform`)
- Policy Module (`policy`)
- Repository Module (`repository`)
- Resource Module (`resource`)
- Scheduling Module (`schedule`)
- Test Module (`test`)

- User Interface Module (`ui`)
- Utilities Module (`utilities`)

The modules are described in alphabetical order.

5.3 Allocate Module (`allocate`)

The `allocation` module provides the computation of possible deployments of a policy architecture into a D-MILS platform. Deployments are defined as static *mappings* from policy elements (that is, components and flows) towards platform structural elements (that is, devices and physical links). The `allocation` module focuses uniquely on allocation of components to nodes. The allocation of flows is considered in further configuration step and is done using specialized network configuration tools developed for TTTEthernet.

The mapping computation is currently subject to two families of constraints:

1. *primitive resource constraints*: these constraints are implicitely obtained by considering altogether the various resource categories requested (resp. available) for components (resp. nodes) within the policy architecture (resp. D-MILS platform) model. As concrete example, the amount of *memory* requested by components mapped on a node must not exceed the available memory of that node.

2. *additional user-defined constraints*: these constraints are explicitely added by the user of the MPCC in order to further restrict or guide the search process. Currently, the two following forms of explicit constraints are handled:

   - a component must / must not be allocated to a given subset of nodes
   - two (or more) components must / must not be allocated to the same node

The `allocation` module relies on the CLPFD library of Prolog to express the deployment constraints and to enumerate all the satisfactory solutions. The functionality is implemented using the predicates above.

allocate(*+Policy, +Platform, +Constraints, -Mapping*)

Find mapping of policy components subject to platform constraints and user-defined constraints. .

allocate_with_ranking(*+Policy, +Platform, +Constraints, :Ranking, -Mapping, -Rank*)

Find the mapping of policy components subject to platform constraints and user-defined constraints. The mappings are ranked according to a generic ranking predicate and returned in decreased order of their rank..

The `allocate` module computes the deployment of policy components onto the platform nodes. The result of the allocation are the component *mappings*, that is, explicit association of components into nodes $compo_i \mapsto node_k$, for all components $compo_i$.

The mapping computation is expressed as a constraint solving problem and solved using the CLPFD library. The various constraints derived from requests/availability of primitive resources or formulated by the user are expressed as arithmetic and/or logical constraints using CLPFD variables. The constraint resolution process is fully under the control of the Prolog engine and allow to enumerate all the satisfactory solutions.

The listing hereafter contains the concrete Prolog implementation.

```
:-module(allocate, [allocate/4,
                    allocate_with_ranking/6]).

:-use_module(library(pairs)).
:-use_module(library(clpfd)).

:-use_module(policy).
:-use_module(platform).

:-meta_predicate allocate_with_ranking(+,+,+,4,-,-).

%! allocate(+Policy, +Platform, +Constraints, -Mapping)

allocate(Policy, Platform, Constraints, Mapping) :-
      policy_components(Policy, Components),
      platform_nodes(Platform, Nodes),
      allocate_with_resource_constraints(Components, Nodes, memory, Xs),
      allocate_with_resource_constraints(Components, Nodes, cpu, Xs),
      allocate_with_explicit_constraints(Components, Nodes, Constraints, Xs),
      build_mapping(Components, Nodes, Xs, Mapping).

%! allocate_with_resource_constraints(+Components, +Nodes, +A, -Xs)

allocate_with_resource_constraints(Components, Nodes, A, Xs) :-
      maplist(component_needs_resource(A, exclusive), Rs, Components),
      maplist(component_needs_resource(A, shared), Us, Components),
      maplist(device_has_resource(A), Cs, Nodes),
      allocate_with_sum_max_capacity_constraints(Rs, Us, Cs, Xs).


% allocation under combined sum/max constraints

%! allocate_with_sum_max_constraints(+Rs, +Us, +Cs, -Xs)
% (\sum {x_i = j} r_i) + (\max_{x_i = j} u_j) <= c_j, forall j.

allocate_with_sum_max_capacity_constraints(Rs, Us, Cs, Xs) :-
      length(Rs, N), length(Us, N), length(Cs, M),
      length(Xs, N), Xs ins 1 ..M,
      numlist(1, M, Js),
      maplist(jth_sum_max_capacity_constraint(Rs, Us, Xs), Js, Cs).

jth_sum_max_capacity_constraint(Rs, Us, Xs, J, C) :-
      maplist(jth_sum_max_capacity_constraint_(J, MaxU), Us, Xs, Bs),
      scalar_product( [1 | Rs], [MaxU | Bs], #=<, C).

jth_sum_max_capacity_constraint_(J, MaxU, U, X, B) :-
      X #= J #==> U #=< MaxU, X #= J #<==> B.
```

```
% allocation with explicit constraints
%! allocate_with_explicit_constraints(+Components, +Nodes, +Constraints, -Xs)

allocate_with_explicit_constraints(Components, Nodes, Constraints, Xs) :-
      maplist(component_id, Components, CIds),
      maplist(device_id, Nodes, NIds),
      maplist(jth_explicit_constraint(CIds, NIds, Xs), Constraints).

jth_explicit_constraint(CIds, _, Xs, x_same(A, B)) :-
      nth1(I, CIds, A), nth1(I, Xs, Xi),
      nth1(J, CIds, B), nth1(J, Xs, Xj),
      Xi #= Xj.

jth_explicit_constraint(CIds, _, Xs, x_notsame(A, B)) :-
      nth1(I, CIds, A), nth1(I, Xs, Xi),
      nth1(J, CIds, B), nth1(J, Xs, Xj),
      Xi #\= Xj.

jth_explicit_constraint(CIds, NIds, Xs, x_notallsame([A|List])) :-
      member(B, List),
      jth_explicit_constraint(CIds, NIds, Xs, x_notsame(A, B)).

jth_explicit_constraint(CIds, NIds, Xs, x_within(A, List)) :-
      nth1(I, CIds, A), nth1(I, Xs, Xi),
      member(Z, List), nth1(J, NIds, Z),
      Xi #= J.

% rewrite Xs into (human readable?) mapping form

%! build_mapping(+Components, +Nodes, +Xs, -Mapping)

build_mapping(Components, Nodes, Xs, Mapping) :-
      maplist(component_id, Components, CIds),
      maplist(nth_device_id(Nodes), Xs, NIds),
      pairs_keys_values(Mapping, CIds, NIds).

% allocate with ranking
% get mappings ordered according to Ranking(+Policy, +Platform,
% +Mapping, -Rank)

%! allocate_with_ranking(+Policy, +Platform, +Constraints,
% :Ranking, -Mapping, -Rank)

allocate_with_ranking(Policy, Platform, Constraints, Ranking, Mapping, Rank) :-
      findall( M-R, (allocate(Policy, Platform, Constraints, M),
                  call(Ranking, Policy, Platform, M, R) ),Unsorted ),
      keysort( Unsorted, Sorted ),member(Mapping-Rank, Sorted).
```

5.4 Command Module (`command`)

The `command` module contains the top-level command interpreter loop, the definition of the valid commands and associated help text, and code to perform command reading, syntax checking, semantics checking, and referral of execution to the code that implements each command.. Also contained in the module is the code for some of some simple commands such as help, inspect, status, and demo. Finally, the code is here to execute stored command procedures and command script files.

The command line interpreter provides a simple, and fairly thin layer between the user and the MPCC. It provides some syntactic sugar, some syntax and semantic checking, and command scripting. If something is not implemented in the command line interpreter, it is possible to exit the command interpreter and directly issue Prolog queries for testing or other special purpose tasks, or one can escape from the command interpreter, using a prefix, to execute a single Prolog directive or query. Many of the commands translate directly to calls on subsystem interfaces.

5.4.1 Command Processing

The top-level command loop prints the MPCC banner defined in param and enters a failure-driven loop that terminates when the `quit` command is entered. The top-level loop prompts, reads a command, executes the command, and repeats, recognizing the difference between interactively executed commands and command procedures and scripts. If the command given is not known it issues an error message and repeats. The top-level loop prompts for a command by sending a string defined in param to the output stream. If the system is not configured to start the top-level loop automatically then it may be started by invoking the goal “mpcc.” at the Prolog prompt after loading the MPCC system into Prolog.

The following describes the command flow and failure handling of the `command` module. The command loop invokes `rd` to read the next command and `do` to execute the command. If the command entered fails to pass the checks performed by `rd`, then `rd` succeeds but the value `invalid` will be returned by `rd` as the next command. The interpreter `do` will see `invalid` not as one of the commands it knows about and will issue an appropriate message and succeed. Command failure is regarded as a failure during the execution, by `do`, of a command considered valid by `rd` and by `do`.

There are four predicates invoked from within the command loop that provide opportunities to define additional actions to be performed each time through the command loop:

- `pre_act` - action to be performed before the next command is read by `rd`. The default is no action.
- `mid_act` - action to be performed after the command has been read by `rd` but before it is executed, and may depend on the particular command. The default is no action.
- `post_act` - action to be performed after the command is executed by `do` (whether such execution resulted in success or failure, see description above of valid commands). The default `post_act` is to perform the `status` command if the system parameter `statprt` is `on`.
- `fail_act` - action to be performed if execution of the command action fails during `do`. The default is to write a failure message defined in param to the console.

There is some memory maintained among these actions to permit Prolog tracing to be turned on for the execution of just one MPCC command.

The predicate `rd` uses the built-in `read_history` to print the prompt string and read from the console, allowing the user to issue commands, possibly modified, from the command history. Also accepted are Prolog goals, preceded by `:-` or `?-`, to be executed by Prolog, which `rd` does directly using `call`, and then `rd` restarts, waiting for a new command without returning. If the entry read is not a Prolog goal, then it is checked for syntactic correctness as a MPCC command using `syntax_chk`, and if that succeeds then constraints on its parameters are checked using `semantics`. If both of these checks succeed, then the entry read is returned as a command, otherwise the value `invalid` is returned as the command.

For each command there should be at least one `help` fact clause. When help is printed for the user, in response to an explicit help query or some exception that triggers help, all of the lines defined in multiple such clauses for the subject command are printed. The first line (first clause) of the help text for each command should be a short statement of what the command does. Subsequent lines may describe the format and effects of parameters. See the existing definitions for the help predicate in the `command` module for examples and conventions.

The predicate `syntax_chk` checks a command from the input stream against the collection of facts that define the valid command forms. It constructs a new term (using the built-in, `functor`, twice) replacing the actual arguments with variables. Then it attempts to unify the result with one of the `syntax` facts that define the name and arity of each command variation.

When the programmer wishes to create a new command, a fact must be added to the `syntax` predicate. If the command can occur with different numbers of arguments, a fact must be created for each different arity. In the `syntax` facts, the arguments should appear as atoms with names that are descriptive of the argument. For example, in the fact

$$syntax(debug(on\_or\_off)).$$

the name of the argument placeholder suggests that the valid values for the argument would be `on` and `off`.

The `semantics` predicate is defined by a collection of clauses (rules) that will unify a command to be checked with a prototype term contained in the head of one of the rules. Variables in the prototype are instantiated to the actual arguments in the command. The body of the rule cuts choice points, committing to the chosen form, and proceeds to test constraints on the variables that hold the argument values. For example, the rule

$$semantics(debug(OnOff)) :- \ !\,,(OnOff == on\ ;\ OnOff == off),\ !\,.$$

constrains the single argument to be one of the two acceptable values. The initial cut commits to the form matched in the head, while the final cut prevents backtracking to the choicepoint created by the `;`. Most such rules only require the initial cut. The final cut could be eliminated by placing `nonvar(OnOff)` before the initial cut. The constraints to be checked are typically simple type checking, using Prolog built-ins to check that an argument is an atom, an integer, a non-variable, ground, or a specific value.

The do predicate unifies a checked command with a form in the head of one of the do clauses, instantiating variable in the head with the actual arguments in the command, and referring control to a command procedure for execution. For readability, the clauses defining the do predicate are kept very concise. The body usually eliminates choice points and calls a command procedure to carry out the command. In some cases, the command procedure is so simple that it can be represent on one line in the body of the selected do clause. In a very small number of cases, the body of the do clause is more than one line, in which alternative cases of the command are selected by distinct values of an argument.

5.4.2 Command Procedures and Scripts

Commands that can be entered from the console can also be executed from a script file or from a stored procedure defined in the procs module. The file or proc contains a list of commands as they would be typed at the console. The run_command_script predicate is the command procedure to carry out the script or proc command. The first argument to run_command_script is the name of the file containing the script. The second argument can have one of the values step, verbose, or none. The first two of these values may be given as an argument to the script command. The value none is supplied by do if the script command is entered without a second argument.

run_command_script reads the file into a list of terms, one command per term. run_command_script calls run_commands to execute the list of commands, pausing for user acknowledgment before executing each command if the argument was step, or printing the command and immediately proceeding to execute it if the argument was verbose.

As in interactive command execution, Prolog goals may be interspersed, and putative commands are checked for syntax and semantics before they are passed to do. Script and proc execution can be nested, that is, scripts and procs can contain the script and proc commands.

5.4.3 Adding a New Command

These are step-by-step instructions for adding a new command to the command module:

1. Add a clause (fact) to the syntax predicate for each form of the new command, e.g., with and without parameters. The single argument to syntax is an instance of a form of the command with a ground argument instance that is descriptive of the argument. There should be a distinct such fact for each form of the command with a different number of arguments.

2. Add a clause (rule) to the semantics predicate for each form of the new command for which semantics checking, such as parameter constraints, is to be performed. It is not necessary to add a semantics clause for commands that have no arguments. The semantics check succeeds by default for any command for which no semantics clause has been defined.

3. Add clauses (facts) to the help predicate for the new command according to the form and conventions presented where the help predicate is described above. Adding help text is optional but recommended.

4. Add a clause (fact) to the do predicate for each form of the new command, e.g., with and without parameters. The action of the command may be performed directly in the body of the do clause or the body may refer to a predicate defined later in the command module or elsewhere. For readability of the command module source it is suggested that do clauses be kept very simple, preferably only one line. Often the body of the do clause will consist simply of a goal invoking the command action predicate.

5. Add a definition for the predicate(s) called by do to perform the action of the new command. Simple miscellaneous commands may be implemented in the command module. The command action predicate for most complex commands will reside in other modules.

Documentation of the MPCC commands may be found in Appendix D.

5.5 Current Configuration Module (current_cf)

Procedures for clearing the current configuration, creating a new configuration or a new configuration generator, instantiating a configuration generator, generating a configuration sequence, and pretty-printing a tree that results from expanding a configuration generator grammar.

Comments in the module define configuration model representation concepts.

5.6 Definite Clause Translation Grammar Module (dctg)

The code in this module is adapted from work by Abramson, Dahl, and Paine [?, ?]. It is used internally by the MPCC to process the "grammar" of configuration languages targeted by the MPCC. It is not used to process the sequences of literal terminal symbols that make up the external form of a configuration language, but rather the sequences of non-terminals that make up non-terminal sentential forms of a configuration.

The dctg module converts a list of terms comprising grammar rules very similar in structure to extended BNF (*Backus-Naur Form* or *Backus Normal Form*), augmented by definitions of semantic attributes, into a set of executable Prolog clauses that operate as a parser (or generator) for the language defined by the grammar.

The dctg module provides predicates to: read grammar rules from a file (dctg_reconsult/1), read from a file containing a structure containing the grammar rules (dctg_s_reconsult/1), or read from a list of grammar rules (dctg_list_reconsult/1). In the MPCC, rules are taken from a list (dctg_list_reconsult) to build the parser program. A two argument variant dctg_list_reconsult/2 separately returns the start symbol of the grammar.

The file variant, dctg_reconsult/1, uses dctg_reconsult_1/1 to repeatedly process terms in the file. The list variant, dctg_list_reconsult/1, uses dctg_s_reconsult1/2 to repeatedly process terms in a list. The list-contained-in-a-structure-in-a-file variant just reads the structure and passes the list to dctg_list_reconsult/1.

Each grammar rule is represented as a Prolog term, and as they are read (from list or file) the term is passed to the predicate process_term/3, which determines form of the rule and, as appropriate, calls translate_rule/2 to actually perform the translation. The predicates t_lp and t_rp translate the left and right parts of a rule, respectively.

The dctg module also contains an interpreter for semantic attributes attached to grammar rules. The interpreter directly recognizes the form of the attribute terms using unification in the head of the rules making up the interpreter. When attributes are referenced, a list or rules defining the attribute names is searched for a rule defining the named attribute. When such a rule is found the body of the rule is executed as a goal, thus instantiating variables occurring in the attribute term. An exception is noted if there is no definition for a referenced attribute.

A number of auxiliary predicates used elsewhere in the module are also defined. The last part of the module is a test case bit_test_grammar for the dctg module's self test.

5.7 Import/Export Module (import_export)

Code that deals with the import and export of XML-based files, and extracting information from XML files and converting formats. Also some code for pretty-printing internal structures.

Intermediate structures are built in the import_export module, and the final result structures are placed in the current_cf module. Many of these intermediate structures have the same name as, but arity different to, the final structures in the current_cf.

5.8 MPCC Module (mpcc)

The mpcc module provides the main entry point to the MPCC, the most typical entry provided by invoking the mpcc/0 predicate. The predicate mpcc/1 provides several optional invocations that provide variations on self-test, regression test, and initialization.

5.9 Noninterference Analysis Module (noninterference)

The noninterference module constructs the Goguen-Meseguer non-interference assertions [?] from a policy architecture. These assertions are generated by the following predicates:

noninterference_assertions(+*Policy*, -*NIAssertions*)

Constructs the list of all noninterference assertions.

ni_assertions(+*Policy*, -*NIA1*, -*NIA2*)

Constructs the noninterference assertions as two separate lists, respectively for isolation and channel control.

The noninterference module provides the primitives for constructing the list of noninterference assertions corresponding to a policy architecture.

```
:-module(noninterference,
 [ noninterference_assertions/2,
   ni_assertions/3]).

:-use_module(policy).
```

```
:-use_module(utilities).
:-use_module(library(ugraphs)).
...

ni_assertions(P,NIAs1,NIAs2) :-
      policy_components(P,Cs), policy_flows(P,Fs), components_ids(Cs,Cids),
      policy_graph(P,G),
      findall(Ur:C,
(              member(C,Cids), delete(Cids,C,Rest),
               findall(R,
(                     member(R,Rest), unreachable(R,G,U,Cids), member(C,U)),
                     Ur)
)            ,
            NIAs1),
      findall([Cid]-Fid:N,
(              member(Cid,Cids), member(flow(Fid,Cid,Nextid),Fs),
               reachable(Nextid,G,N)
)            ,
            NIAs2).
```

5.10 Parameters Module (`param`)

The `param` module provides a single place for the setting of MPCC system parameters and the definition of strings, directories, and file names.

5.11 Platform Module (`platform`)

The `platform` module provides structural representation and access primitives for D-MILS platforms and associated HW constraints. The representation can be (1) auto-generated from detailed HW descriptions (such as the ones produced by the autoconfig tool of LynxSecure) or (2) defined manually by the system designer.

The proposed platform representation is structural. D-MILS platforms are represented by Prolog terms containing as subterms representations of respectively, platform devices (nodes or switches) and the physical links. For both, placeholders are defined to represent specific attributes and/or primitive resources (see the `resource` module hereafter). More formally, platform terms are generated by the following grammar:

platform	::=	platform([*node(s)*], [*physical-link(s)*], [*attribute(s)*])
device	::=	device(device-id, *device-family*, [*attribute(s)*], [*primitive-resource(s)*], [*port(s)*])
device-family	::=	node | switch
port	::=	port (port-id, port-category)
physical-link	::=	physical_link(link-id, *port-reference*, *port-reference*, [*attribute(s)*])
port-reference	::=	pr(device-id, port-id)

As an example, consider the following example which present a simple D-MILS platform consisting of two nodes (x and y) connected by using one switch (s) and two physical network links with some characteristics.

```
platform(
       [device(x, node, [],
          [resource(memory, 50), resource(cpu, 4)],
          [port(x1, p1), port(x2, p2)]),
        device(y, node, [],
          [resource(memory, 100), resource(cpu,2)],
          [port(y1, p1), port(y2, p2), port(y3, pmgmt)]),
        device(s, switch, [],
          [],
          [port(s1, p1), port(s2, p2)])],

       [physical_link(pl_x_s, pr(x, x1), pr(s, s1),
         [attribute(mediaType, copper) ]),
        physical_link(pl_s_y, pr(s, s2), pr(y, y2),
         [attribute(transmissionSpeed, 1000),
          attribute(mediaType, fiber)])],

       [attribute(transmissionSpeed, 1000)]).
```

The `platform` module provides all the necessary primitives to access information from platform terms as defined above:

platform_devices(+*Platform*, -*Devices*)
Extract the list of devices for a platform.

platform_nodes(+*Platform*, -*Nodes*)
Extract the list of nodes for a platform.

platform_switches(+*Platform*, -*Switches*)
Extract the list of switches for a platform.

platform_physical_links(+*Platform*, -*PhysicalLinks*)
Extract the list of physical links for a platform.

platform_attributes(+*Platform*, -*Attributes*)
Extract the list of attributes from a platform.

platform_has_attribute(+*AttributeName*, -*Value*, +*Platform*)
Extract the value of an attribute for a platform.

device_id(+*Device*, -*DeviceId*)
Extract the identifier of a device.

nth_device_id(+*Devices*, +*N*, -*DeviceId*)
Extract the identifier of the Nth device in the list.

device_attributes(+*Device*, -*Attributes*)
Extract the list of attributes of a device.

device_resources(+*Device*, -*Resources*)
Extract the list of resources of a device.

device_ports(+*Device*, -*Ports*)
Extract the list of ports of a device.

device_has_attribute(+*AttributeName*, -*Value*, +*Device*)
Extract the value of an attribute for a device.

device_has_resource(+*ResourceCategory*, -*Quantity*, +*Device*)
Extract the quantity of a resource category for a device.

physical_link_has_attribute(+*AttributeName*, -*Value*, +*PhysicalLink*)
Extract the value of an attribute for a physical link.

The `platform` module provides all the necessary primitives to access information from a platform term.

```
:-module(platform,
 [ platform_devices/2, platform_nodes/2, platform_switches/2,
   platform_physical_links/2, platform_attributes/2,
   platform_has_attribute/3,

   device_id/2, nth_device_id/3, device_attributes/2,
   device_resources/2, device_ports/2, device_has_attribute/3,
   device_has_resource/3,

   physical_link_has_attribute/3]).

:-use_module(resource).

% platform(devices, physical_links, attributes)

platform_devices(platform(Ds,_,_), Ds).
platform_nodes(platform(Ds,_,_), Ns) :-
      include(device_is_node, Ds, Ns).
platform_switches(platform(Ds,_,_), Ss) :-
      include(device_is_switch, Ds, Ss).
platform_physical_links(platform(_,PLs,_), PLs).
platform_attributes(platform(_,_,As), As).

%! platform_has_attribute(+A, -V, +Platform)

platform_has_attribute(A, V, platform(_,_,As)) :-member(attribute(A,V), As).

% device(id, {node|switch}, attributes, resources, ports)

device_id(device(Id,_,_,_,_), Id).
nth_device_id(Ds, I, Id) :-nth1(I, Ds, device(Id,_,_,_,_)).
device_is_node(device(_,node,_,_,_)).
device_is_switch(device(_,switch,_,_,_)).
device_attributes(device(_,_,As,_,_), As).
device_resources(device(_,_,_,Rs,_), Rs).
device_ports(device(_,_,_,_,Ps), Ps).

%! device_has_attribute(+A, -V, +Device)

device_has_attribute(A, V, device(_,_,As,_,_)) :-member(attribute(A,V), As).

%! device_has_resource(+R, -Q, +Device)
```

```
device_has_resource(R, Q, device(_,_,_,Rs,_)) :-member(resource(R, Q), Rs), !.
device_has_resource(_, 0, device(_,_,_,_,_)).

% physical_link(id, pr(devid, portid), pr(devid, portid), attributes)
%! physical_link_has_attribute(+A, -V, +PhyLink)

physical_link_has_attribute(A, V, physical_link(_,_,_,As)) :-
member(attribute(A,V), As).
```

5.12 Policy Module (`policy`)

The `policy` module provides a structural representation, access primitives and basic manipulation for annotated architecture policies. According to the MPCC flow, the clauses used to represent a policy are auto-generated by the `import_export` module from BIP representations.

The policy representation is structural. A policy is represented as a Prolog term, whose subterms represent the components and the flows between them. Components are represented at interface level (that is, no implementation details) by their set of ports and associated attributes. Flows are represented as binary connections between component ports. Both components and flow representations contain placeholders for representing primitives resources as well as specific attributes, if needed. Policy terms are generated by the following grammar:

policy	::=	policy([*component(s)*], [*flow(s)*], [*attribute(s)*])
component	::=	component(component-id, *component-family*, [*attribute(s)*], [*resource(s)*], [*port(s)*])
component-family	::=	subject | object
port	::=	port(port-id, *port-direction*, *port-family*, data-type-id, [*attribute(s)*])
port-direction	::=	in | out
port-family	::=	event | data | event data
flow	::=	flow(flow-id, *port-reference*, *port-reference*, [*attribute(s)*])
port-reference	::=	pr(component-id, port-id)

The listing below presents a concrete example, the red-crypto-black policy architecture.

```
policy(
      [component(red, subject, [], [],
          [port(out_header, out, event_data, header, []),
           port(out_data, out, event_data, data, [])]),
       component(bypass, subject, [], [],
          [port(in_header, in, event_data, header, []),
           port(out_header, out, event_data, header, [])]),
       component(crypto, subject, [], [],
          [port(in_data, in, event_data, data, []),
           port(out_data, out, event_data, encrypted_data, [])]),
       component(black, subject, [], [],
          [port(in_header, in, event_data, header, []),
           port(in_data, in, event_data, encrypted_data, [])])],

      [flow(f1, pr(red,out_header), pr(bypass,in_header), []),
      flow(f2, pr(red,out_data), pr(crypto,in_data), []),
      flow(f3, pr(bypass,out_header), pr(black,in_header), []),
      flow(f4, pr(crypto,out_data), pr(black,in_data), [])],

    []).
```

The `policy` module provides all the necessary primitives to access information from policy terms as defined above:

policy_components(+*Policy*, -*Components*)
Extract the list of components for a policy.

policy_flows(+*Policy*, -*Flows*)
Extract the list of flows for a policy.

policy_attributes(+*Policy*, -*Attributes*)
Extract the list of attributes for a policy.

policy_has_attribute(+*AttributeName*, -*Value*, +*Policy*)
Extract the value of an attribute for a policy.

policy_graph(+*Policy*, -*Graph*)
Build the policy graph for a policy.

component_id(+*Component*, -*ComponentId*)
Extract the identifier of a component.

components_ids(+*Components*, -*ComponentIds*)
Extract the list of identifiers from a list of components.

nth_component_id(+*Components*, +*N*, -*ComponentId*)
Extract the identifier of the Nth component in the list.

component_is_subject(+*Component*)
Test if the component is a subject.

component_is_object(+*Component*)
Test if the component is an object.

component_attributes(+*Component*, -*Attributes*)
Extract the list of attributes for a component.

component_resources(+*Component*, -*Resources*)
Extract the list of resources for a component.

component_ports(+*Component*, -*Ports*)
Extract the list of ports for a component.

component_has_attribute(+*AttributeName*, -*Value*, +*Component*)
Extract the value of an attribute for a component.

component_needs_resource(+*ResourceCategory*, -*Access*, -*Quantity*, +*Component*)
Extract the quantity and the access type requested for a resource category within a component.

flow_id(+*Flow*, -*FlowId*)

Extract the identifier of a flow.

flow_components(+*Flow*, -*ComponentId*, -*ComponentId*)

Extract the source and target component identifier of a flow.

flow_has_attribute(+*AttributeName*, -*Value*, +*Flow*)

Extract the value of an attribute for a flow.

The `policy` module provides all the necessary primitives to access information from a policy term.

```
:-module(policy,
 [ policy_components/2, policy_flows/2, policy_attributes/2,

   component_id/2, components_ids/2,
   nth_component_id/3, component_is_subject/1,
   component_is_object/1, component_attributes/2,
   component_resources/2, component_ports/2,
   component_has_attribute/3, component_needs_resource/4,

   flow_id/2, flow_components/3, flow_has_attribute/3,

   policy_graph/2]).

:-use_module(resource).

% policy(components, flows, attributes)

policy_components(policy(Cs,_,_), Cs).
policy_flows(policy(_,Fs,_), Fs).
policy_attributes(policy(_,_,As), As).

%! policy_has_attribute(+A, -V, +Policy)

policy_has_attribute(A, V, policy(_,_,As)) :-
      member(attribute(A, V), As).

% component(id, {subject|object}, attributes, resources, ports)
% obs: resources are the ones 'requested' by the component...

component_id(component(CId,_,_,_,_), CId).
components_ids(Cs, Cids) :-
      findall(Cid,member(component(Cid,_,_,_,_),Cs),Cids).
nth_component_id(Cs, I, CId) :-
      nth1(I, Cs, component(CId,_,_,_,_)).
component_is_subject(component(_,subject,_,_,_)).
component_is_object(component(_,object,_,_,_)).
component_attributes(component(_,_,As,_,_), As).
component_resources(component(_,_,_,Rs,_), Rs).
component_ports(component(_,_,_,_,Ps), Ps).

%! component_has_attribute(+A, -V, +Component)

component_has_attribute(A, V, component(_,_,As,_,_)) :-
      member(attribute(A, V), As).
```

```
%! component_needs_resource(+A, +S, -Q, +Component)

component_needs_resource(R, A, Q, component(_,_,_,Rs,_)) :-
      member(resource(R,Q,A), Rs), !.
component_needs_resource(_,_,0, component(_,_,_,_,_)).

% port(id, {in|out}, {event|data|eventdata}, type, attributes)

% flow(id, pr(compid,portid), pr(compid,portid), attributes)

flow_id(flow(FId,_,_,_), FId).
flow_components(flow(_, pr(X,_), pr(Y,_), _), X, Y).
flow_has_attribute(A, V, flow(_,_,_,As)) :-
      member(attribute(A,V), As).

%! policy_graph(+Policy, -Graph)

policy_graph(Policy, Graph) :-
      policy_components(Policy, Cs),
      policy_flows(Policy, Fs),
      components_flow_graph(Cs, Fs, Graph).

components_flow_graph(Cs, Fs, Graph) :-
      findall( CId-SIds,
(              member(C, Cs), component_id(C, CId),
               findall( SId,
                      component_flow_successor(CId, Fs, SId),
                      SIds) ),
             Graph).

component_flow_successor(CId, Fs, SId) :-
      member(F, Fs), flow_components(F, CId, SId).
```

5.13 Procedures Module (procs)

The procs module provides a place to define useful MPCC command procedures used in development, testing, and use of the MPCC.

5.14 Repository Module (repository)

The repository module provides the configuration repository subsystem. It defines the kinds of configuration files that may occupy the repository and manages the variations of the names of configuration kinds. It provides predicates to load/store from the repository, repository_load/3 and repository_store/3; check for the existence of a repository configuration or a configuration generator for a kind of configuration, exists_repository_config/2 and exists_conf_gen_for_kind/2; and manages an in-memory cache for open repository items.

5.15 Resource Module (`resource`)

The `resource` module provides an unified representation and functionality for handling primitives resources and attributes. These are used to annotate policy and platform elements. We define them as simple terms defined by the following simple grammar:

resource	::=	resource (resource-category, resource-quantity, *resource-access*?)
resource-access	::=	exclusive | shared
attribute	::=	attribute(attribute-name, attribute-value)

Primitives resources are defined by their category (an identifier, e.g., memory), the quantity available (a numerical value) and possible, the type of access provided/requested to them (exclusive, shared, etc.). Attributes are defined by their name (an identifier) and their associated value (identifier).

resource_category(+*Resource*, -*Category*)
Extract the category for a resource.

resource_quantity(+*Resource*, -*Quantity*)
Extract the quantity for a resource.

resource_access(+*Resource*, -*Access*)
Extract the access type for a resource.

attribute_name(+*Attribute*, -*Name*)
Extract the name for an attribute.

attribute_value(+*Attribute*, -*Value*)
Extract the value for an attribute.

The `resource` module provides the primitives for manipulation of primitive resources and attributes.

```
:-module(resource, [
  resource_category/2, resource_quantity/2, resource_access/2,
  attribute_name/2, attribute_value/2 ]).

% resource(category, quantity, [access={shared|exclusive}])

resource_category(resource(X,_), X).
resource_category(resource(X,_,_), X).

resource_quantity(resource(_,X), X).
resource_quantity(resource(_,X,_), X).

resource_access(resource(_,_,X), X).

% attribute(name, value)

attribute_name(attribute(X,_), X).
attribute_value(attribute(_,X), X).
```

5.16 Scheduling Module (`schedule`)

The `schedule` module provides the computation of possible schedules for running subjects mapped onto the same node. That is, whenever multiple subjects are deployed onto the same node, a schedule is needed to coordinate their access to (shared) computing resources (namely CPU).

The `schedule` module computes periodic schedules according to utilization constraints expressed for different subjects. In the current setting, we restricted ourselves to periodic subjects. Every subject j is characterized by two attributes, namely the period and the amount of CPU time requested during the period. Two scheduling algorithms have been implemented so far:

- simple scheduling policy, whenever the subjects have identical periods, simply choose an order for execution,
- earliest deadline first scheduling policy (EDF), subjects requests are unfolded (replicated) for the hyper-period[3] and scheduled according to the EDF strategy

Other scheduling algorithms can be defined and integrated as needed.

The interface of the `schedule` module is listed below. In all predicates above, *tasks* denote subject's periodic requests for execution and are characterized by their period P and execution time E.

schedule(+*Policy,* +*Platform,* +*Mapping,* -*Schedules*)

Find schedules for all platform nodes, given the policy and the mapping.

schedule1(+*Tasks,* -*Schedule,* -*Period*)

Find a schedule according to the simple scheduling strategy for a set of periodic tasks with identical periods.

schedule2(+*Tasks,* -*Schedule,* -*Period*)

Find a schedule according to the earliest deadline first scheduling strategy.

load(+*Tasks,* -*Load*)

Compute the load $\sum_{j \in Tasks} \frac{E_j}{P_j}$ corresponding to a set of periodic tasks.

The `schedule` module implements the computation of schedules according to various strategies. The fragment below contains the code of the earliest deadline first scheduling strategy.

```
:-module(schedule, [schedule/4,
                    schedule1/3,
                    schedule2/3,
                    load/2]).

:-use_module(library(pairs)).

:-use_module(policy).
:-use_module(platform).

%! schedule(+Policy, +Platform, +Mapping, -Schedules)
% - the policy
% - the platform
```

[3] least common multiple of existing periods

```
% - the mapping [cId-nId, ...]
% - schedules, if available used node [nId-[period, schedule], ...]

schedule(Policy, Platform, Mapping, Schedules) :-
      policy_components(Policy, Components),
      platform_nodes(Platform, Nodes),
      maplist(schedule_node(Components, Mapping), Nodes, Schedules).

schedule_node(Components, Mapping, Node, Schedule) :-
      device_id(Node, NId),
      findall(task(SId, ExecTime, Period),
(             member(Subject, Components),
              component_is_subject(Subject),
              component_id(Subject, SId),
              member(SId-NId, Mapping),
              component_needs_resource(schedule, exclusive,
                                 rate(ExecTime, Period), Subject) ),
            Tasks),
(      schedule2(Tasks, Sch, Period) -> Result = [Period, Sch] ; Result = fail),
      pairs_keys_values([ Schedule ] ,[ NId ], [ Result ]).

      /* ...*/

%! schedule2(+Tasks, -Schedule, -Period)
% - tasks as before
% - schedule as before
% - schedule period as before
% - edf scheduling, for arbitrary task periods

schedule2([], [], 0).

schedule2(Tasks, Schedule, Period) :-
      load(Tasks, L), L =< 1.0,
      maplist(period, Tasks, Ps), lcm(Ps, Period),
      maplist(unfold_task_jobs(Period), Tasks, Jobs),
      append(Jobs, UnsortedJobs),
      sort_jobs(UnsortedJobs, SortedJobs),
      edf_schedule(0, [], SortedJobs, Schedule).

                        % unfold jobs of a task, for some hyper period

unfold_task_jobs(0, task(_, _, _), []).

unfold_task_jobs(XP, task(Id, C, P), [job(Id, XPminusP, C, XP) | Jobs ]) :-
      XP > 0, 0 is XP mod P,
      XPminusP is XP - P,
      unfold_task_jobs(XPminusP, task(Id,C,P), Jobs).

      /* ...*/

%! edf_schedule (+CurrentTime, +ReadyJobs, +WaitingJobs, -Schedule).
% - jobs are characherized by (TaskId, ArrivalTime, ExecTime, Deadline)
% - the waiting list is ordered by arrival times (increasing)
% - jobs are preemptive

      % terminal case, empty problem

edf_schedule(_, [], [], []).

      % jobs arrival (move from waiting to ready)

edf_schedule(T, RJs, [job(J,T,E,D) | WJs], S) :-
      edf_schedule(T, [job(J,T,E,D) | RJs], WJs, S).

      % idle time (wait for the first job)

edf_schedule(T, [], [job(J,A,E,D) | WJs], [wait(X) | S]) :-
      T < A, X is A-T,
      edf_schedule(A, [job(J,A,E,D)], WJs, S).

      % schedule the ready queue (without waiting jobs)

edf_schedule(T, RJs, [], [exec(J,E) | S]) :-
      edf_select_job(RJs, job(J,_,E,D), RemRJs),
      T + E =< D, NextT is T + E,
      edf_schedule(NextT, RemRJs, [], S).

      % schedule the ready queue (until next waiting job...)

edf_schedule(T, RJs, [job(Jw,Aw,Ew,Dw) | WJs], [exec(J,X) | S]) :-
      T < Aw,
      edf_select_job(RJs, job(J,A,E,D), RemRJs),
      T + E =< D,
(        T + E =< Aw
      -> NextT is T + E, X is E,
         edf_schedule(NextT, RemRJs, [job(Jw,Aw,Ew,Dw) | WJs], S)
      ; NextT is Aw, X is Aw - T, RemE is E - X,
         edf_schedule(NextT, [job(J,A,RemE,D) | RemRJs], [job(Jw,Aw,Ew,Dw) | WJs], S)).

      /* ...*/
```

5.17 Target Adapter Module (`target`)

Contains the predicates to convert target-specific formats to MCNF and vice versa. Currently these are identity mapping stubs.

The `target` module uses an auxiliary module, the XML schema utilities module `xsu`, for loading and querying XML schemata files, used by the configuration generator-generator for XML-based languages. The module also has include directives for files from per-target directories containing target-specific utility procedures.

5.18 Test Module (`test`)

The `test` module provides a framework for self-test and regression tests.

5.19 User Interface Module (`ui`)

A small set of primitive actions for interacting with a user. To be expanded upon as needed.

5.20 Utilities Module (`utilities`)

The `utilities` module defines a small number of utility predicates that don't obviously belong somewhere else.

5.21 Implementation Standards and Conventions

5.21.1 Self Tests and Regression Tests

Self tests establish the integrity and minimal correct functional operating characteristics of the MPCC. Regression tests are comprehensive functional tests that establish that previously implemented and tested functionality is not broken by development work on the current version. The conventions for the implementation of each of these in MPCC is described in the following.

Self tests are intended to be performed upon startup, periodically, and on demand. To minimize the time required to perform startup and periodic tests, there is a $self\_test$ flag in the $param$ module that is checked for the value on before self tests are automatically initiated (the value off indicates that tests should not be performed automatically). Self tests may also be initiated on demand at any time by the command interpreter's $selftest$ command.

Regression tests are cumulative from one version of the MPCC to the next. Individual tests should be adapted when necessary to accommodate changes in functionality. Regression tests may be lengthy and would typically be performed on command during development and prior to committing changes to the code repository.

The top-level module $test$ in the file $test.pl$ provides the entry point for the initiation of all MPCC system-level tests. As deemed appropriate, other functional modules shall provide their own self tests and regression tests according to the conventions described here. Modules defining self tests are listed in $param : self\_test\_modules/1$. Modules defining regression tests are listed in $param : regression\_test\_modules/1$. Modules that are designated to be subject to self tests and/or regression tests shall contain the generic mechanisms necessary for running the tests, and shall be independent of the test cases, which are to be defined separately. Test cases are maintained separately so that the versioning of the functional module and its test cases are distinct.

The test cases shall be independently defined in a separate file that shall be incorporated in the associated module at compile time by occurrence of a Prolog $include$ directive for the file containing the test cases. If the module file is named, for example, $a\_module.pl$ then the regression test file shall be named $a\_module\_test.pl$. The module test files are contained in the directory $Test$ that is parallel to the source directory $CORE$ where the corresponding module source is contained.

Self tests to be performed on startup, periodically, and on command for a module may be configured in the module's test file, by defining the lists $a\_module\_startup\_tests$, $a\_module\_periodic\_tests$, and $a\_module\_demand\_tests$. Regression tests to be performed on demand for a module are enumerated in the list $a\_module\_regression\_tests$, defined in the test file for the module. Any defined test case may occur in more than one of these lists.

In addition to the self tests and regression tests, a functional module may contain examples of constructs defined in the module and of the use of functions defined in the module. (Cf. example policies and example platform defined in the $policy$ and $platform$ modules, respectively.) These are distinct from the self tests and regression tests, and if useful may be duplicated in the module's test file and integrated there into the automated testing mechanism.

6 Assurance Considerations

The goal of Task T 5.5 and Deliverable D 5.3 of WP 5 (*Distributed MILS Platform Configuration Compiler*) is "configuration correctness and definition of semantics-preserving transformations." That document more comprehensively treats the key assurance concerns associated with the MPCC. We provide here a few high-level observations and comments.

The only complete argument for configuration correctness would have to take into account the detailed static semantics of the configuration languages of each of the targets, the operational semantics of the processing of those representations to establish the initial state of each of the components, and by the operational semantics of the interpretation of that data by each component at runtime. This clearly involves many phases of processing that are beyond the scope, and IP accessibility, of this project.

It is not a goal of this effort to deliver a high-assurance implementation of the MPCC, which would have to meet stringent assurance requirements in addition to the functional requirements. Our goal is to develop a functioning prototype configuration framework. A benefit of MILS modular assurance is that a framework can be developed to ultimately support high assurance but that is initially populated by components that are not high-assurance, and would most likely require re-implementation to achieve high assurance. When such components are later replaced by "work-alike" high-assurance ones then neither the conceptual framework nor other surrounding components need be changed, provided care was taken to implement the same functional interfaces and behaviors in the prototype as are specified for the high-assurance version. The framework can be incrementally upgraded. By implementing a MPCC that is "plug compatible" with backend tools, an upgrade path to a future high-assurance MPCC has been established. As incremental improvements are made, the assurance case would reflect the incremental amelioration, or elimination, of assurance deficits.

The compositional assurance and verification efforts (WP 4) in the D-MILS project are developing a generic assurance case for D-MILS systems. Included in this assurance case, among other things, is argumentation and evidence supporting the goals claimed for the D-MILS tools, including the D-MILS Platform Configuration Compiler, and assumptions about the target tools and components. Similar arguments would need to be made for any additional components targeted in the future.

In safety-critical systems tools are often assessed from the point of view of what they could do wrong and what they could fail to do right. In support of the assurance case for the MPCC, the following describes important characteristics of its design and organization that mitigate risks associated with it.

6.1 Correspondence Among Model Representations

The policy architecture expressed as a MILS-AADL model passes through a sequence of two transformations prior to its use in the configuration compiler:

1. transformation from MILS-AADL to BIP

2. transformation from BIP to clausal representation in Prolog

The first transformation has been introduced in Deliverables D3.2 [?] and D3.3 [?] of WP3. This transformation covers the MILS-AADL language to a large extent. The operational semantics of the models is preserved from MILS-AADL to BIP. The translation structurally maps MILS-AADL components to BIP components, and their connections to interactions. The operational semantics rules defined for MILS-AADL are mimicked by the semantics of interactions in BIP.

The BIP model obtained is used as input for both the configuration compiler and for BIP verification and analysis tools developed in WP4. From the configuration compiler perspective, however, not all the information contained in the model is actually needed. The configuration compiler uses exclusively the structural information, that is, components and connections, with some specific annotations. The behaviour of components is not needed.

Henceforth, as far as only the structural information is concerned, the correspondence between the MILS-AADL model and the BIP model is rather trivial. The translation is structure-preserving: the same set of components and connections exists in the BIP model as in the original MILS-AADL model. The translation rules restricted to the structure are relatively simple and easy to implement.

The second transformation has been defined to achieve a simpler and more convenient input of the policy architecture model into Prolog. This transformation step actually extracts the useful structural information from the BIP model and rewrites it as a Prolog term. This step is essentially a simple rewriting step, from BIP to Prolog syntax, for the structural part.

6.2 Confidence in the Configuration Process

The search of a perfected configuration has been expressed as a constraint satisfaction problem and solved using the Prolog engine and specific constraint solving libraries.

The constraint programming approach lets us focus on *what* are the properties/constraints the configuration must satisfy and less on *how* it is computed. From this point of view, the confidence is related to

- whenever the configuration process succeeds, the ability to check (easily) that the resulting (alternative) configurations indeed satisfies all the constraints, expressed on different models (for the policy, the platform, etc).
- whenever the configuration process fails, the ability to produce evidence / justification for the unsatisfiability of constraints.

Obviously, both aspects are equally important for a sucessful configuration process based on constraint solving. The assurance argument can therefore focus on why the configuration, if one is obtained, is correct and not on the correctness of Prolog constraint solving, which is definitely out of the scope of D-MILS.

6.3 Syntactic and Semantic Correctness of Target Configurations

As described in Section 3, the MPCC has a front-end which synthesizes a target-agnostic (except for constraints expressed in the platform model) D-MILS configuration that is consistent with the MILS-AADL system model. The MPCC has a back-end that is responsible for projecting the target agnostic distributed system configuration onto expressions in the configuration languages of the components that appropriately represent the intended configuration for each component. Thus, one important issue is *diversity* of target representations.

The target-specific configuration languages are not only diverse, but they may be more-or-less *confidential*. Therefore, another important issue is the ability to *isolate* the details of the implementation of transformations to such representations.

The configuration languages of targets may be *complex*, with elaborate options and very low-level details, which must be consistently and correctly established for the generated configuration representation to be a viable input to the target's configuration internalisation tools.

Finally, all of the above issues must be addressed in a way that makes ongoing maintenance and upgrades feasible.

To meet the challenges of diversity and confidentiality/isolation we introduced the concept of a target adapter to specialise the configuration information to a target. A target adapter may be thought of as a "device driver" that permits output to be generated to a particular target. The implementations of target adapters are maintained outside of the MPCC core, though they interact with the core, or at least with data structures generated by the core. The MPCC core will operate with or without any target adapters present.

To meet the challenges of complexity and maintainability we defined a design pattern for target-adapters that we call configuration generators. The main elements of a configuration generator are a syntactic definition of the target language expressed in a formal metalanguage, a set of semantic constraints on the language expressed as attribute computations attached to the grammar, and the ability to build a data structure representation of the result as well as a textual one. For example, by building an internal XML "element" structure corresponding to the generated configuration data, standardized XML output libraries may be used to generate external XML format.

The syntactic and semantic definitions of the target language are used to guide and constrain the generation of the configuration information in the target language. To the extent that the syntax and semantics are correctly and sufficiently defined in the configuration generator, the generated target form will be correct by construction. The manner of representation of the grammar of the target within the configuration generator, as a well-understood declarative form (annotated grammar rules), permits validation of the syntax and semantics by direct inspection, rather than by having to try to infer what is generated by procedural code.

A Appendix: Distributed MILS Configuration Compiler Requirements (from D1.3)

The following requirements have been identified for the D-MILS research and development work of the Configuration Compiler:

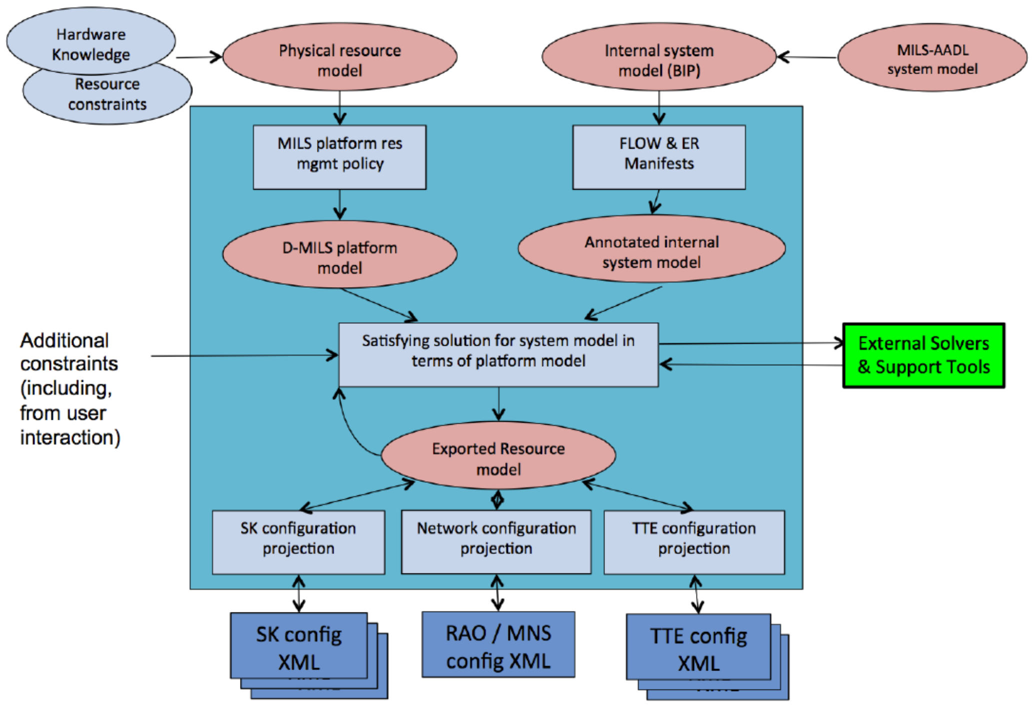
Hardware Knowledge
Resource constraints
Physical resource model
Internal system model (BIP)
MILS-AADL system model
MILS platform res mgmt policy
FLOW & ER Manifests
D-MILS platform model
Annotated internal system model
Additional constraints (including, from user interaction)
Satisfying solution for system model in terms of platform model
External Solvers & Support Tools
Exported Resource model
SK configuration projection
Network configuration projection
TTE configuration projection
SK config XML
RAO / MNS config XML
TTE config XML

Figure 5: D-MILS Platform Configuration Compiler models and flow

CC-WP5.1 MANDATORY: The Configuration Compiler shall provide representation for the intermediate models used in the compilation flow, as illustrated in figure 5.

CC-WP5.2 MANDATORY: All the models used by configuration compiler shall be represented using abstractions and/or annotations available in the AADL-MILS and D-MILS intermediate language, as provided in WP2 and WP3.

CC-WP5.3 MANDATORY: The Configuration Compiler shall produce configurations that enforce the policy architecture under given infrastructure constraints (physical platform(s), separation kernel(s) and network topology).

CC-WP5.4 FUTURE: The Configuration Compiler shall automatically build the physical resource model by importing physical platform constraints from the hardware database constructed by the LynxSecure auto-configuration tool.

CC-WP5.5 MANDATORY: An approach shall be provided to import a system specification described in the Fibex format and transform it into an appropriate model used by the Configuration Compiler.

CC-WP5.6 DESIRABLE: The interaction between TTE and LynxSecure shall be investigated. This could open opportunity to incorporate additional system-level constraint or optimization concerns (e.g., joint time partition and network traffic scheduling for end-to-end latency optimization).

CC-WP5.7 DESIRABLE: The Configuration Compiler shall handle additional optimization, security and user-interaction constraints.

CC-WP5.8 DESIRABLE: The Configuration Compiler shall produce generic XML configuration files for separation kernels.

CC-WP5.9 MANDATORY: The Configuration Compiler shall produce XML separation kernel configuration files, according to LCC-WP5-* requirements on LynxSecure target; these files shall be initially produced using the `hcv2bcv` tool.

CC-WP5.10 MANDATORY: The Configuration Compiler shall produce XML network configuration files, according to TCC-WP5-* requirements on TTE target.

CC-WP5.11 DESIRABLE: Whenever no satisfying configurations exist for the actual configuration inputs, the configuration compiler shall provide diagnostics and assistance to the user.

CC-WP5.12 MANDATORY: The Configuration Compiler shall provide incremental development of a satisfying configuration, through progressive refinement of the exported resource model.

CC-WP5.13 FUTURE: The Configuration Compiler shall import and build the exported resource model from an existing configuration file.

The following requirements have been identified for the D-MILS research and development work in semantic analysis and property checking of the D-MILS configuration:

SAC-WP5.1 MANDATORY: The exported resource model constructed by the Configuration Compiler shall be a provable correct refinement of the policy architecture model on the selected D-MILS platform.

SAC-WP5.2 MANDATORY: The model transformations used internally by the Configuration Compiler shall by manually proven correct, that is, preserving essential functional and security properties.

SAC-WP5.3 DESIRABLE: The output configurations shall have a fully formal semantics.

SAC-WP5.4 FUTURE: The Configuration Compiler shall provide a formal proof of correctness of the output configurations with respect to the semantics of the input models.

The following requirements have been identified for the D-MILS research and development work addressing the TTEthernet target for the Configuration Compiler:

TCC-WP5.1 MANDATORY: A specification shall be provided for the XML-based language accepted by the TTE configuration tool $^{TTE}Plan$ to allow the D-MILS Configuration Compiler to generate correct input.

TCC-WP5.2 MANDATORY: The semantics for each structure of the language shall be described in terms of how it is interpreted by the TTEthernet subsystem and/or how it affects the initialization or the runtime operation of TTE (operational semantics).

TCC-WP5.3 MANDATORY: A specification shall be provided for the format of the network configuration information. (Comment: this is also part of the XML language).

TCC-WP5.4 MANDATORY: A specification shall be provided for the communication of any additional constraints that the D-MILS Configuration Compiler must consider when planning the use of the network resources reported to be available in the physical D-MILS configuration.

The following requirements have been identified for the D-MILS research and development work addressing the LynxSecure target for the Configuration Compiler:

LCC-WP5.1 MANDATORY: A specification shall be provided for the XML-based language accepted by the LynxSecure configuration tool.

LCC-WP5.2 MANDATORY: The semantics for each structure of the language shall be described in terms of how it is interpreted by the LynxSecure runtime and/or how it affects the initialization or the runtime operation of LynxSecure (operational semantics).

LCC-WP5.3 MANDATORY: The specification shall make it possible for the D-MILS Configuration Compiler to generate correct XML for consumption by the LynxSecure configuration tool

LCC-WP5.4 DESIRABLE: The semantics of the generated XML files shall be validated against the specification before submitting them to the configuration tool.

LCC-WP5.5 MANDATORY: A specification shall be provided for the format of the hardware configuration database constructed by the automated hardware discovery procedure, i.e. each structure and element of the database shall be described and its interpretation given in terms of the device identifications and hardware attributes used in the respective documentation of each hardware component.

LCC-WP5.6 MANDATORY: The intermediate representation shall be adopted for the D-MILS Configuration Compiler to make possible the import of information about the available hardware resources.

LCC-WP5.7 DESIRABLE: A specification shall be provided for the communication of any other possible constraints that the D-MILS Configuration Compiler must consider when planning the use of remote resources reported to be available on one or more D-MILS nodes.

B Appendix: Internal Representations

B.1 Representation of the BIP model

The BIP models taken as input by the configuration compiler are produced from MILS-AADL by using the *milsaadl2bip* tool and the transformation described in [**?**, Sect. 5].

For complete details about the BIP framework, including the modeling language and associated tools, we refer to the overview included in [**?**, Sect. 3.2.2] and the public documentation available on the BIP web site, at `http://www-verimag.imag.fr/New-BIP-tools.html`.

We recall that the representation of MILS-AADL models in BIP is structural and cover a large part of the MILS-AADL language. That is, the (hierarchical) organization of a policy architecture as a network of interconnected components is fully preserved in BIP. Furthermore, behavior of components, expressed in terms of modes and mode-transitions is represented as well. Some limitations exist however, for representation of complex structured data types and for continuous and hybrid behaviour. Nonetheless, as explained next, these missing features have no impact on the configuration process, which requires only the structural information from the BIP model.

A concrete BIP model obtained by translation from MILS-AADL can be found in [**?**, App. A].

B.2 Clausal representation of BIP model

An intermediate clausal representation of BIP models has been defined in order to facilitate their input within the MPCC.

This representation allowed us to decouple the development of the MPCC from the development of intermediate representations and associated transformations from MILS-AADL and BIP.

From a BIP model (obtained from MILS-AADL as explained above) we extract a Prolog term representing the policy architecture. This term contains the key information about components (identifier, semantic category, attributes), their interfaces (event and data ports, with associated data types) and the various connections (event connections and data flows, with their attributes). The grammar of these policy terms has been formally introduced in Section 5.12.

The extraction of the clausal representation has been implemented as part of the MPCC frontend. The *bip2pl* translator uses the native BIP frontend for parsing and internal representation of BIP models. The Prolog term extraction reduces essentially to the pretty printing of useful information from the model using some specific syntax.

Finally, let us remark that this representation of BIP models in Prolog is restricted to structural information, and therefore partial. Behavioral aspects are completely ignored as they do not have any impact on the configuration process.

B.3 Clausal representation of derived model elements

The configuration procedure generates a number of intermediate model elements (or configuration artifacts). Among the most relevant, we recall the following:

- the mapping/deployment of policy components to D-MILS platform nodes
  Mappings are represented as lists of pairs *(component-id, node-id)*.
- the schedule for multiple subject components mapped onto the same platform node
  Schedules are represented as sequences of *(component-id, time)*.

B.4 Internal representations of external forms

Definition of the representation(s) that are in between the internal configuration model and the external representation(s).

C Appendix: External Representations

C.1 Generating external representations for arbitrary targets

There are a multitude of ways that one could use the internal form of the MPCC to generate the input for external configuration tools associated with various MILS components. To mention a few, one could export the internal form as Prolog clauses, which are syntactically simple, and could process them with bespoke tools for specific target languages. One could develop bespoke translation inside Prolog to output the form of the target language. One could build within Prolog an interaction between the MPCC and the tools that are specific to the target.

We sought an organization that would be amenable to many kinds of target languages and that would minimize the amount of work both for the initial transformations and for ongoing maintenance of the connection between the MPCC and a target, for example, to accommodate changes made to the target language by the target developers.

We also needed a way to isolate the specific details of proprietary targets from the rest of the MPCC code, so that it is not part of the MPCC source base, but can be linked with the MPCC on-the-fly when needed. We refer to this as a target adapter. This concept itself does not diminish the multitude of approaches for its implementation, but it does discourage mixing target-specific transformation functions with generic functions.

A significant concern is how to how these adapters are developed and how one can validate that the configuration information generated for a target will be acceptable to the target in all instances. To prevent ad hoc approaches to the implementation of adapters and to systematically approach the correctness of the final output representation, we introduce the notion of a configuration generator.

C.2 Configuration Generators

A configuration generator is a component of a target adapter that is intended to address some of the aforementioned concerns. A target adapter consists primarily of a configuration generator for the target, plus any needed additional target-specific utility functions that may be used before, during, or after the configuration generator is applied.

The configuration generator encodes the correct form of the target language in a grammar, and generates syntactically correct sentences in that language. This approach facilitates validation of the generated language and maintenance of the generator if the target language changes. Semantic constraints of the target language may also be associated with the syntax in the configuration generator. The completeness and correctness of the generator's grammar representation of the target language and semantic constraints is much easier to validate than would be ad hoc code created to generate the language.

As the configuration generator is applied, calls on an interface to the MPCC internal configuration representations are made to guide the elaboration of the configuration expression in the target language.

This approach can be applied more generically to the generation of configuration generators: a configuration generator to generate configuration generators may be applied to a data base that defines a particular configuration language. An XML schema is such a database. The generator-generator is provided with the top symbol to use as the root of the grammar to appear the the resulting configuration generator. The generator-generator queries the XML schema to obtain the definitions of the non-terminals as it descends to concrete syntax and defined primitive types, generating grammar rules, and deferred grammar rules as it proceeds.

In general, a configuration generator may reference the common internal representation of the configuration model through an open interface, and use that information to populate a correct sentence in the target language with the appropriate information from the model. We write

$$M|G \Rightarrow S$$

to indicate the application of a generator to a model derive a sentence. The symbol | indicates composition using an interface provided to the generator that is specific to the to the model type. The generator calls upon this interface to obtain the specific model elements that appear as the values of terminal elements of the target language.

D Appendix: Command Line Interface

The *command* module implements numerous commands. This section provides a brief reference to those that are currently used. Each command should be terminated by a full stop (".").

`configure` (policy id , platform id) Run configuration procedure. The first argument is a policy id. The second argument is platform id (ids are tags for a policy or platform examples stored in the module). This command is used to demonstrate the configuration procedure.

`demo` (demo id) Run the canned demonstration associated with demo identifier (currently, only configure or configure(policy,platform)).

`echo` (string) Print the string on the standard output.

`help` List the legal command forms.

`help` (command name) Provides help on the named command.

`import` (file spec) Import based on file spec (conf(F), model(F), or confgen(F)).

`inspect` (item) Inspect values of internal structures or variables based on item argument (settings, name, cfdb, ccf, current, model, model(M), xmlfacts, str, xml, schema, seq, cg, target(P,T), xrm, mpm, hpm, hwc, prm).

`load` (doc id [, again]) Load a configuration from the repository as the current configuration. The first argument is a configuration id (base name of the repository file). If the second (optional) argument is "again" a reload is performed.

`make` Recompile changed source files.

`newconf` (conf gen id [, new conf id [, new conf title]]) Create a new configuration as the "current configuration". The first argument is a configuration generator id (base name of the repository file). The second argument (optional but necessary if argument three is to be supplied) is a new configuration id that will override that given in the generator. The third argument (optional) is a new configuration title that will override the generic title given in the document generator.

`newgen` (conf gen id , new file) Create a new configuration generator. The first argument is the kind of the new generator and the second argument is the repository file name.

`newgen` (Conf_kind , Ck , CK , ConfigurationKind) Create a new configuration kind. Arguments 1-4 specify the different forms of the generator name as defined in the repository module.

`proc` (proc id [, `step` | `verbose`]) Run stored command procedure defined in procs module. The argument `step` causes the command to be printed followed by a pause for operator go-ahead, `verbose` causes commands to be printed and executed without pausing.

`quit` Terminate the MPCC top-level command loop or a command script.

`regtest` Run regression tests.

`script` (file [, `step` | `verbose`]) Run command script from a file. The argument `step` causes the command to be printed followed by a pause for operator go-ahead, `verbose` causes commands to be printed and executed without pausing.

`selftest` Run self tests.

`set (` name [`,` value] `)` Set flag `name` to `value`. Command with one argument displays the value of the named flag. Command with no arguments displays value of all settable flags.

`status` Display MPCC system status.

`traceoff` Turn Prolog tracing off.

`traceon` Turn Prolog tracing on.

`traceone` Turn Prolog tracing on for one MPCC command.

`version` Display MPCC current version number.

`versions` Display MPCC past versions with descriptions and current version number.

E Appendix: Examples

E.1 Mapping search

Consider the following policy architecture consisting of three components:

```
policy_example(4, policy([component(s1, subject, [],
                          [resource(memory, 6, exclusive),
                           resource(cpu, 3, exclusive)], []),

                        component(s2, subject, [],
                          [resource(memory, 5, exclusive),
                           resource(cpu, 2, exclusive)], []),

                        component(s3, subject, [],
                          [resource(memory, 7, exclusive),
                           resource(cpu, 2, exclusive)], [])],

                       [], [])).
```

Consider the following platform consisting of two nodes:

```
platform_example(1, platform([device(n1, node, [],
                              [resource(memory, 15),
                               resource(cpu, 8)], []),

                             device(n2, node, [],
                              [resource(memory, 15),
                               resource(cpu,2)], [])],
                             [], []))
```

The mapping search find two possible solutions satisfying the resource constraints:

```

?- policy:policy_example(4,Policy), platform:platform_example(2,Platform),
   allocate(Policy,Platform,[],Mapping).
Policy = ...
Platform = ...
Mapping = [s1-n1, s2-n1, s3-n2] ;
Policy = ...
Platform = ...
Mapping = [s1-n1, s2-n2, s3-n1] ;
false.
```

Nonetheless, by considering the additional constraint that $s2$ and $s3$ are mapped on the same node, no solution is found:

```
?- policy:policy_example(4,Policy), platform:platform_example(2,Platform),
   allocate(Policy,Platform,[x_same(s2,s3)],Mapping).
false.
```

E.2 Schedule search

Consider two periodic tasks a and b, both with execution time 1, and periods respectively 2 and 4. By running edf scheduling, we obtain

```
?- schedule2([task(a, 1, 2), task(b, 1, 4)], S, Length).
S = [exec(a, 1), exec(b, 1), exec(a, 1)],
Length = 4 ;
false.
```

That is, an unique periodic schedule of length 4, where a is first executed for 1 time unit, b is then executed for 1 time unit, and finally, a is executed again for one time unit.

E.3 Mapping and Scheduling search

Consider the following policy architecture consisting of two subjects and one object:

```
policy_example(1, policy([component(a, subject, [],
                             [resource(memory, 20, exclusive),
                              resource(schedule, rate(10, 10), exclusive)],
                             []),

                    component(b, subject, [],
                             [resource(memory, 40, exclusive),
                              resource(memory, 10, shared),
                              resource(schedule, rate(12, 20), exclusive)],
                             []),

                    component(c, object, [],
                             [resource(memory, 60, exclusive)],
                             [])],

                   [], [])).
```

Consider the following platform consisting of two interconnected nodes:

```
platform_example(1, platform([device(x, node,
                              [],
                              [resource(memory, 50),
                               resource(cpu, 4)],
                              [port(x1, p1),
                               port(x2, p2)]),

                       device(y, node,
                              [],
                              [resource(memory, 100),
                               resource(cpu,2)],
                              [port(y1, p1),
                               port(y2, p2),
                               port(y3, pmgmt)]),

                       device(s, switch,
                              [],
                                                    [],
                                                    [port(s1, p1),
                                                     port(s2, p2)])],

                                      [physical_link(pl_x_s, pr(x, x1), pr(s, s1),
                                                    [attribute(mediaType, copper) ]),

                                       physical_link(pl_s_y, pr(s, s2), pr(y, y2),
                                                    [attribute(transmissionSpeed, 1000),
                                                     attribute(mediaType, fiber)])],

                                      [attribute(transmissionSpeed, 1000)])).
```

The stratified search for mapping and scheduling had an unique solution:

```
?- configure:configure_test(1,1,[],Mapping,Schedules).
Mapping = [a-y, b-x, c-y],
Schedules = [x-[20, [exec(b, 12)]], y-[10, [exec(a, 10)]]] ;
false.
```