
MAENAD

Grant Agreement 260057

# Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles

## Report type — Deliverable D3.2.1

## Report name — Analysis and Synthesis Concepts Supporting Engineering Scenarios

| | |
|---|---|
| **Dissemination level** | **PU** |
| **Status** | **Final (4th iteration)** |
| **Version number** | **4.0** |
| **Date of preparation** | **2014-02-20** |

## Authors


| Editor | E-mail |
|---|---|
| Martin Walker | Martin.Walker@hull.ac.uk |

| Authors | E-mail |
|---|---|
| Martin Walker | Martin.Walker@hull.ac.uk |
| Sandra Torchiaro | Sandra.Torchiaro@crf.it |
| DeJiu Chen | chen@md.kth.se |
| Tahir Naseer | tnqu@md.kth.se |
| Mark-Oliver Reiser | Mark-Oliver.Reiser@tu-berlin.de |
| David Parker | D.J.Parker@hull.ac.uk |
| Henrik Lönn | Henrik.Lonn@volvo.com |
| Sara Tucci | Sara.Tucci@cea.fr |
| David Servat | David.Servat@cea.fr |
| Lei Feng | leifeng@md.kth.se |
| Nidhal Mahmud | N.Mahmud@hull.ac.uk |
| Luís Azevedo | L.P.Azevedo@2012.hull.ac.uk |

| Reviewer | E-mail |
|---|---|
| Andreas Abele | Andreas.Abele@continental-corporation.com |
| Renato Librino | Renato.Librino@4sgroup.it |


**The Consortium**

Volvo Technology Corporation (S) Centro Ricerche Fiat (I)
Continental Automotive (D) Delphi/Mecel (S) 4S Group (I)
MetaCase (Fi) Pulse-AR (Fr) Systemite (SE) CEA LIST (F)
Kungliga Tekniska Högskolan (S) Technische Universität Berlin (D) University of Hull (GB)

## Revision chart and history log

| Version | Date | Reason |
|---|---|---|
| 0.1 | 2010-12-01 | Initial outline. |
| 0.9 | 2011-05-09 | Final draft |
| 1.0 | 2011-05-27 | Final review |
| 1.0.1 | 2011-07-1 | Optimisation discussion |
| 1.0.2 | 2011-07-6 | Timing analysis update |
| 1.0.3 | 2011-07-12 | Added details on the optimisation architecture elements. |
| 1.0.4 | 2011-08-31 | Period 1 refinements |
| 1.1.0 | 2012-03-30 | Second iteration |
| 1.1.1 | 2012-09-15 | Draft version for review release of second iteration |
| 2.0.0 | 2012-10-04 | Release version of second iteration |
| 3.0.0 | 2013-01-16 | Third iteration |
| 3.9.0 | 2014-02-02 | Fourth iteration draft for review |
| 4.0.0 | 2014-02-20 | Final version for release |

| Approval | Date |
|---|---|
| Henrik Lönn | 2014-02-20 |

## List of Abbreviations

| | |
|---|---|
| AADL | Architecture and Analysis Design Language |
| ARS | Assigned Requirement for Safety |
| ASIL | Automotive Safety Integrity Level |
| ASR | Assigned Safety Requirement |
| BDA | Behavioral Description Annex (for EAST-ADL) |
| CCF | Common cause failure |
| COF | Constrained Output Failure |
| DRS | Driver Reaction System |
| DSR | Derived Safety Requirement |
| ETA | Event Tree Analysis |
| FEV | Fully Electric Vehicle(s) |
| FM | Failure Mode |
| FMEA | Failure Modes and Effects Analysis |
| FTA | Fault Tree Analysis |
| GSPN | Generalised Stochastic Petri Net |
| HAZOP | Hazard & Operability Analysis |
| HiP-HOPS | Hierarchically-Performed Hazard Origin & Propagation Studies |
| HRC | Heteorogenous Rich Components |
| MC | Markov Chain |
| MoC | Model of Computation |
| M(A) | Model (Airfix) |
| PROMELA | Process Meta-Language |
| RBD | Reliability Block Diagram |
| RAS | Replenishment-at-Sea |
| RSA | Required Safety Assignment |
| SAFORA | State Automata to Fault-trees extended with temporal information (ORA in Greek) |
| SAR | Search-and-Rescue |
| SM | State Machine |
| SPEEDS | Speculative and Exploratory Design in Systems Engineering |
| SPIN | Simple PROMELA Intepreter |
| SRA | Safety Requirement Allocation |
| SRD | Safety Requirement Derivation |
| TFT | Temporal Fault Tree |
| UML | Unified Modeling Language |

## Table of contents

Authors....2

Revision chart and history log....3

List of Abbreviations....4

Table of contents....5

1 Introduction....8

1.1 Goals....8

1.2 Structure....9

2 Analysis and synthesis concepts to support ISO 26262....11

2.1 Summary of the ISO 26262 safety design workflow....11

2.2 Hazard analysis in ISO 26262....13

2.3 Risk assessment in ISO 26262....14

*2.3.1 Evaluation Criteria....15*

*2.3.2 ASIL (Automotive Safety Integrity Levels)....16*

2.4 ASIL decomposition and allocation in ISO 26262....18

*2.4.1 ASILs in ISO 26262....18*

*2.4.2 ASIL Decomposition & Allocation Algorithm....23*

*2.4.3 HiP-HOPS and Tool Support for ASILs....35*

2.5 Discussion & Further Work....37

*2.5.1 Support for ASILs....37*

2.6 Optimisation algorithm for ASIL Decomposition....40

*2.6.1 The ASIL Optimisation Process....40*

*2.6.2 Solution Representation....40*

*2.6.3 Cost Calculation....41*

*2.6.4 Tabu Search Algorithm for Optimisation....41*

*2.6.5 Implementation & Performance....45*

3 Model-based system analysis techniques....47

3.1 The Scope of Analysis support by EAST-ADL....47

3.2 Enhanced language support for behaviour-centric analysis....48

*3.2.1 The usages of EAST-ADL native behaviour constraint specification....48*

*3.2.2 Key content of EAST-ADL native behaviour constraint specification....50*

*3.2.3 Analysis of EAST-ADL native behaviour constraint specification through to external tools....59*

3.3 Modelling support for dependability analysis....63

*3.3.1 Dependability and error modelling in EAST-ADL....63*

*3.3.2 Safety Analysis with HiP-HOPS Technology....64*

*3.3.3 Multi-perspective analysis of EAST-ADL models....70*

*3.3.4 Dynamic SM-based Safety Analysis....74*

*3.3.5 Tool specification and exchange formats* ..... *85*
*3.3.6 Tool Integration* ..... *86*
*3.3.7 Semantic Mapping Transformation* ..... *88*
*3.3.8 Representation Transformation* ..... *89*
3.4 Timing Analysis ..... 91
*3.4.1 EAST-ADL/TADL for schedulability analysis* ..... *92*
*3.4.2 MARTE for Schedulability Analysis* ..... *94*
*3.4.3 The 'analysis gap' between EAST-ADL/TADL and MARTE for schedulability* ..... *97*
*3.4.4 Timing analysis tooling* ..... *97*
*3.4.5 Timing analysis limitations* ..... *99*
3.5 Analysis and Synthesis concepts to support Electrical Vehicles ..... 99
*3.5.1 Identified Needs* ..... *99*
*3.5.2 Component count and cost analysis* ..... *100*
*3.5.3 Cable Length* ..... *103*
*3.5.4 Power Consumption* ..... *103*
*3.5.5 Current analysis* ..... *103*
*3.5.6 Power Distribution Analysis* ..... *104*
3.6 Application of analysis techniques to the E/E lifecycle ..... 104
*3.6.1 Vehicle Level* ..... *105*
*3.6.2 Analysis Level* ..... *105*
*3.6.3 Design Level* ..... *106*
*3.6.4 Implementation Level* ..... *106*
*3.6.5 Optimisation* ..... *106*
4 Automatic multi-objective optimisation of system models ..... 107
4.1 Brief description of optimisation definitions and concepts ..... 107
4.2 Comparison of optimisation algorithms ..... 108
*4.2.1 Genetic Algorithms (GAs)* ..... *108*
*4.2.2 Tabu Search* ..... *110*
*4.2.3 Ant Colony* ..... *111*
*4.2.4 Simulated Annealing* ..... *111*
4.3 Tool support for automatic optimisation ..... 112
*4.3.1 Variability in HiP-HOPS* ..... *112*
*4.3.2 HiP-HOPS Optimisation Algorithm* ..... *113*
*4.3.3 Achieving full multi-objective optimisation in MAENAD* ..... *115*
*4.3.4 Optimisation Architecture Elements* ..... *117*
*4.3.5 Product Line Optimisation* ..... *122*
5 Implementation support for variability management ..... 127
6 Summary and conclusions ..... 129
7 References ..... 131

8 Appendix A — Library of Architectural Patterns....134
8.1 Base function type definition....134
8.2 Duplicate with Combination Block....134
8.3 Primary/standby recovery block....135
8.4 K-out-of-N Voter....136
8.5 Triple redundant standby-recovery block....136
8.6 Pair & Spare....137
8.7 Base hardware component type definition....138
8.8 Duplicated hardware component....138
8.9 System-level allocation....139

# 1 Introduction

## 1.1 Goals

The primary challenge of the MAENAD project is to develop methods and techniques to support the engineering and design of FEV (Fully Electric Vehicles). To achieve this, it is necessary to define language support in EAST-ADL for the required modelling, analysis, and optimisation concepts, and furthermore to develop algorithms and specify tools that can be implemented and used to analyse and optimise those models. The extensions to EAST-ADL to support the additional modelling concepts are covered in D3.1.1, while the specification of analysis and optimisation algorithms is described in this document, D3.2.1.

In particular, the D.3.x.1 deliverables are intended to support the following project objectives:

- O1-1: Support in EAST-ADL for the safety process of ISO 26262 and representation of safety requirements.

The ISO 26262 standard contains a number of different procedures and design techniques to ensure the safety of an automotive system. To achieve the project goals, EAST-ADL must be extended to support the different modelling approaches present in ISO 26262 and algorithms or tools should be developed to support the analyses required by ISO 26262. The additional analysis concepts and tool support required for ISO 26262 are described in this document.

- O1-2: Automatic allocation of safety requirements (ASILs).

One of the features of ISO 26262 is the concept of ASILs (Automotive Safety Integrity Levels) and the way they can be allocated to different system elements to meet safety requirements. The automatic allocation of safety requirements also has potential applications beyond simply supporting ISO 26262 and could be applied to similar types of qualitative requirements in other domains. While the language concepts necessary to support ASILs and safety requirements are described by D3.1.1, the algorithms for the analysis and allocation of ASILs are specified in this document.

- O2-1: Dependability analysis of EAST-ADL models (with capabilities for multi-mode and temporal analysis of failures, together with integrated assessment of HW-SW design perspectives).

To fully support ISO 26262, it must also be possible to perform various dependability analyses at different stages of the design process, ranging from more simple conceptual analysis to more detailed architectural analysis. Such analyses include hazard analyses, risk analyses, criticality analyses, Failure Modes and Effects Analysis (FMEA), and Fault Tree Analysis (FTA).

- O2-2: Behavioural Simulation of EAST-ADL models.

As well as dependability analysis, another goal of MAENAD is to introduce execution and compositional behaviour semantics into EAST-ADL to allow for behavioural simulation and analysis of EAST-ADL models. The modelling concepts necessary for behavioural simulation are described in D3.1.1 while the analysis and tool support necessary is specified in this document.

- O2-3: Timing Analysis of EAST-ADL models.

The third type of model-based analysis included MAENAD is timing and performance analysis. As with the other forms of analysis, the necessary language concepts are described in D3.1.1 while the algorithm and tool support is specified here in D3.2.1. In the case of timing analysis, this will mean developing ways of linking EAST-ADL models to external timing analysis tools like Qompass.

- O3-1: Extension of EAST-ADL with semantics to support multi-objective optimisation for product lines.

In addition to analysis of EAST-ADL models of FEVs, another goal of MAENAD is to support model-based optimisation. This involves both the extension of existing concepts and the development of new concepts in EAST-ADL to allow for substitution of one design element for another, with the aim of creating a design space that can be explored by optimisation algorithms (such as genetic algorithms). The notion of substitutability and the definition of alternative model elements that can be explored by the optimisation algorithm are achieved via extension of the variability capabilities of EAST-ADL.

The modelling concepts necessary for optimisation, including variability, are described in D3.1.1, while the algorithmic support is specified here in D3.2.1.

- O3-2: Definition of a library of standard architectural patterns that can be automatically applied on an un-optimised EAST-ADL model in order to improve dependability and performance.

Finally, one technique for supporting model-based optimisation is to allow different architectural patterns to be applied to a design model. Such patterns exhibit different dependability and performance characteristics, and the automatic selection and substitution of such patterns can allow different design candidates to be rapidly conceived and evaluated. Some example patterns forming a simple architectural library can be found at the end of this document in Appendix A.

## 1.2 Structure

The above objectives are developed in MAENAD by several different working groups that each focus on different aspects of the project goals: ISO 26262 support (XG-I), system analyses (XG-A), optimisation (XG-O), and variability (XG-V). Both D3.x.1 deliverables are therefore organised along these lines, and therefore the overall structure of D3.2.1 is as follows:

**Section 2** describes the requirement for additional analysis techniques and tools necessary to fully support the ISO 26262 design process, including a brief summary of the ISO 26262 workflow itself. ASIL decomposition is covered in this section.

**Section 3** describes the concepts and algorithms behind the various system analyses and tools developed in MAENAD, including subsections for both dependability analysis to support ISO 26262 and each of the other types of analysis of EAST-ADL models, namely behavioural simulation (via the new Behavioral Description Annex or BDA) and timing analysis. There is also a section on support for FEV analysis concepts. The specifications of the analysis concepts and algorithms in D3.2.1 is a valuable input to the tool implementation activities of WP5, as well as feeding back into development of EAST-ADL itself.

**Section 4** covers the development of concepts and algorithms for the automatic optimisation of EAST-ADL models, particularly with respect to dependability and timing/performance objectives. The development of different substitutable architectural patterns (with the ultimate aim of allowing them to be used in the optimisation process) is ongoing, and these will be fully specified in an appendix instead rather than the main body of this report.

**Section 5** briefly covers the analysis and tool support developed in MAENAD for the modelling of variability. Some of the variability concepts are also inevitably discussed as part of the optimisation section.

**Section 6** summarises the different analysis and synthesis concepts described in D3.2.1 and relates them to the objectives set out in this introduction, describing how each is fulfilled (or not) by the algorithms and tools developed in MAENAD.

**Appendix A** contains a library of simple architectural patterns that can be used to specify extra variability in a model and how components or functions can be substituted with more dependable alternatives, thereby facilitating optimisation.

# 2 Analysis and synthesis concepts to support ISO 26262

This section describes the requirements of the ISO 26262 standard and how we aim to support those requirements in MAENAD.

## 2.1 Summary of the ISO 26262 safety design workflow

The ISO 26262 requires the application of a “functional safety approach”, starting from the preliminary vehicle development phases and continuing throughout the whole product life-cycle. This approach allows for the design of a safe automotive system. Furthermore, it provides an automotive specific risk-based approach for determining risk classes named ASILs (Automotive Safety Integrity Levels). The new standard uses ASILs for specifying the item's necessary safety requirements for achieving an acceptable residual risk, and provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved [1].

The main phases of ISO 26262 are:

- Concept phase (ISO 26262 - Part 3):
    - Item definition: during this phase the Item has to be identified and described. To have a satisfactory understanding of the item, it is necessary to know about its functionality, interfaces, and any relevant environmental conditions. Moreover the boundary of the item has to be defined.
    - Initiation of safety lifecycle: the safety lifecycle is initiated based on Item definition and by determining the development category (e.g. current production, new, modified, etc). In the case of modification, an impact analysis shall be performed.
    - Hazard analysis and risk assessment: to evaluate the risk associated with the item under safety analysis, a risk assessment is carried out. A risk assessment considers the functionality of the item and a relevant set of scenarios (operating conditions & environmental conditions). For each identified hazardous event the ASIL level has to be determined. Moreover, after the assessment of the risk, the safety goal (top level safety requirements) shall be defined.
    - Functional Safety Concept: based on the safety goals, the functional safety requirements have to be specified. The functional safety concept shall include:
        - Functional safety requirements → to define in a complete way the functional safety requirements, the following attributes have to be defined for each hazardous event: safety goals, operating modes, fault tolerant time interval, possible safe state, transitions to and from the safe state, emergency operation interval, functional redundancies, driver warning, degraded operation and driver’s actions;
        - Item functional description and requirement allocation;
        - Interaction description with vehicle systems;
        - Functional specifications to achieve the safety goals;
        - Description of external measures to avoid or mitigate the effects of hazards.

- System level development – specification, (ISO 26262 - Part 4)
    - Initiation of product development at system level: the functional safety activities during the individual sub-phases of system development have to be determined and

planned. The safety plan shall include the Item integration and testing plan, the validation plan and the functional safety assessment plan.

- Specification of technical safety requirements: the technical safety requirements shall be specified by refining the functional safety concept, considering both the functional concept and the preliminary architectural assumptions. Moreover the consistency and traceability between functional safety requirements and technical safety requirements shall be verified.
- System design: the system design and the technical safety concept shall be defined to comply with the functional requirements, and the technical safety requirements specification of the item shall be defined. Methods and measures to avoid systematic failures have to be applied during system design. Furthermore, methods to avoid random hardware failures during system operation shall be applied in this phase.
- Comply with the technical safety requirements specification: it is necessary to verify that the system design and the technical safety concept comply with the technical safety requirements specification.

- Hardware and Software level development, (ISO 26262 - Part 5 and Part 6)
  - Hardware and Software development - safety technical design: based on the system design specification, the item is developed from the hardware and software level perspective.
- System level development – integration and validation (ISO 26262 - Part 4)
  - Item integration: the compliance with each safety requirement in accordance with its specification and ASIL classification shall be tested. Moreover it is necessary, during this phase, to verify that the "System design" covering the safety requirements is correctly implemented by the entire item. The item integration starts with the integration of the hardware and software of each element that the item comprises. After the integration of the elements into a full system, the final step is the integration of the item with other systems within a vehicle and with the vehicle itself.
  - Safety validation: the evidence of compliance with the safety goals and evidence that the functional safety concepts are appropriate for the functional safety of the item shall be provided. Moreover the evidence that the safety goals are correct, complete and fully achieved at the vehicle level shall be provided.
  - Functional safety assessment: it is necessary to assess that the functional safety is achieved by the item.
  - Release for production: the release for production criteria at the completion of the item development shall be specified. The release for production confirms that the item complies with the requirements for functional safety at the vehicle level.
- Production and Operation Level (ISO 26262 - Part 7)
  - Production: a production process for safety-related elements or items that are intended to be installed in road vehicles has to be developed and maintained. Functional safety during the production process by the relevant manufacturer or the person or organisation responsible for the process shall be achieved.
  - Operation, service and decommissioning: in order to maintain the functional safety after the item’s release for production, the customer information, maintenance and repair instructions, and disassembly instructions regarding the item, system or element shall be specified.

- Management of functional safety (ISO 26262 - Part 2): in this phase the requirements for management of functional safety are specified. These requirements cover all the phases of the safety lifecycle, and are specified for:
    - Overall safety lifecycle management;
    - Safety management during the concept phase and the product development;
    - Safety management after the item's release for production.

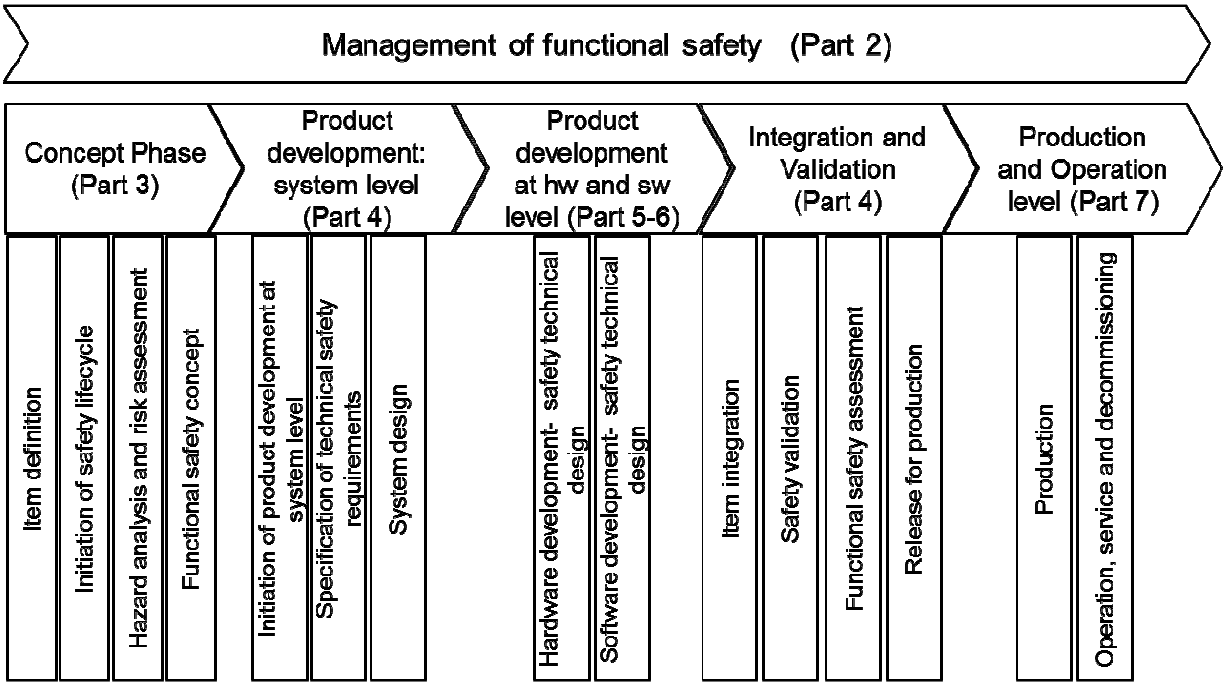



**Figure 1 - ISO 26262 Safety Lifecycle**

## 2.2 Hazard analysis in ISO 26262

- To identify the hazards, it is essential to define the malfunctions and eventual maintenance condition(s) related to the item.

All possible functional anomalies derivable from each foreseeable source shall be defined.

In general, for each *target function (*the function description in terms of output(s) behaviour) it is always possible to define, in an exhaustive manner, the malfunctions in terms of anomalies of function activation:

- Unwanted activation
- Missed activation
- Degraded activation (vs. value or timing)
- Incoherent activation (in presence of more than one function's output)
- Unwanted deactivation
- Missed deactivation
- Degraded deactivation (vs. value or timing)
- Incoherent deactivation (in presence of more than one function's output)

## 2.3 Risk assessment in ISO 26262

In order to perform a correct and complete risk assessment, a proper set of realistic scenarios (all variables and/or states that characterise the functions or affect them) must be defined.

A complete description of a scenario includes:

Operative conditions → selection of the factors related to vehicle usage scenarios to be considered for situation analysis and hazard identification. Examples in the vehicle dynamic control field are:

- *Dynamic driving state*
  - ✓ Speed
  - ✓ Longitudinal acceleration
  - ✓ Lateral acceleration
  - ✓ …
- Manoeuvres
  - ✓ Curve
  - ✓ Overtaking
  - ✓ Braking
  - ✓ Lane change
  - ✓ …
- Driver condition
  - ✓ Attention level
  - ✓ Hands on steering
  - ✓ …

- Environmental conditions → selection of the factors related to environmental variables to be considered for situation analysis and hazard identification. Typical environmental conditions include:
  - *Location*
    - ✓ Urban road
    - ✓ Suburban road
    - ✓ Motorway or highway (with lateral barriers)
    - ✓ …
  - *Street condition*
    - ✓ Adherence
    - ✓ Road surface type (smooth, rough, etc.)
    - ✓ Slope
    - ✓ …
  - *Traffic situation*
    - ✓ Presence of other vehicle(s) (lateral, oncoming, following, etc.)
    - ✓ Presence of cycles, pedestrians, etc.

The hazard under safety analysis, when applied to the operational situations (operative & environmental conditions), triggers the so called “hazardous event".

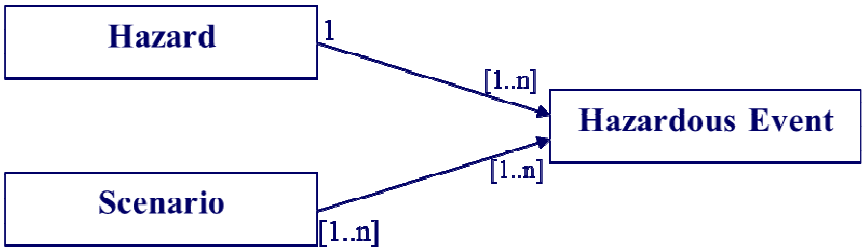



**Figure 2 - Hazardous Events**

For each identified hazardous event, the severity, controllability and exposure values should be ranked to determine the associated Automotive Safety Integrity Level (ASIL), representing the level of risk.

Parameters for risk determination according to ISO 26262:

- *Controllability [C]:*

  Avoidance of a specified harm or damage through timely reactions of the persons involved.

- *Severity [S]:*

  Measure of the expected degree of harm to an endangered individual in a specific situation.

- *Exposure time [E]:*

  State of being in an operational situation that may be hazardous if coincident with the malfunction under consideration

### 2.3.1 Evaluation Criteria

**Controllability**

The controllability shall be assigned to one of the controllability classes C0, C1, C2 and C3 in accordance with the table below:

| Class | Description |
|---|---|
| **C0** | Controllable in general |
| **C1** | Simply controllable |
| **C2** | Normally controllable |
| **C3** | Difficult to control or uncontrollable |

**Table 1 - Controllability**

It is important to note that the controllability levels assigned to the various situations should be assessed through specific testing on the road, fault injection, etc.

**Severity**

The severity shall be assigned to one of the severity classes S0, S1, S2 or S3 in accordance with the table below.

| Class | Description |
|---|---|
| **S0** | No injuries |
| **S1** | Light and moderate injuries |
| **S2** | Severe and life-threatening injuries (survival probable) |
| **S3** | Life-threatening injuries (survival uncertain), fatal injuries |

**Table 2 - Severity**

**Exposure time**

The probability of exposure shall be assigned to one of the probability classes E1, E2, E3 and E4 in accordance with the table below.

| Class | Description |
|---|---|
| **E1** | Very low probability |
| **E2** | Low probability |
| **E3** | Medium probability |
| **E4** | High probability |

**Table 3 - Exposure**

### 2.3.2 ASIL (Automotive Safety Integrity Levels)

Safety Integrity Levels (SILs) are abstract classification levels that can be used to indicate the level of safety required of safety-critical systems (or elements thereof). SILs have been adopted as part of safety standards such as IEC 61508 and - in the automotive domain - ISO/FDIS 26262. In the context of the upcoming ISO 26262, SILs are known as ASILs - Automotive Safety Integrity Levels - and form a major part of the standard: ASILs are used to specify the necessary safety requirements for achieving an acceptable residual risk, as well as providing requirements for validation and confirmation to ensure the required levels of safety are being achieved.

Safety requirements in these standards are intended to ensure the system being designed is free from unacceptable risk (assuming the requirements are met) and are derived through a process of analysis and risk assessment. The aim of the process is to determine the critical system functions - those which have the potential to be hazardous in the instance of failure - and the requirements necessary to mitigate the effects or reduce the likelihood of those hazards. These safety requirements are often associated with integrity requirements that apply to those critical functions to indicate, in essence, what level of contribution they have towards the overall system safety and thus what level of safety they should implement to avoid system failures. A low ASIL therefore indicates that the element is not a major contributor to severe system failures, while a high ASIL indicates that it is potentially is a major contributor, and this allows a means of verifying that system safety requirements are being achieved by ensuring that the ASILs allocated to system elements are also being met.

Therefore ASILs play a dual role in the development of safety-critical systems: they allow for top-down allocation of safety requirements to different elements of the system according to their contribution to risk, and they allow for bottom-up verification to show that the safety requirements are being met by the developed system.

ASILs are divided into one of four classes to specify the item's necessary safety requirement for achieving an acceptable residual risk, with D representing the highest and A the lowest class. The ASIL-Level shall be determined for each hazardous event using the estimation parameters severity (S), probability of exposure (E) and controllability (C) in accordance with the table below. QM (Quality Management) can be applied to non-safety critical elements to indicate that there are no specific safety requirements in place.

| | | **C1** | **C2** | **C3** |
|---|---|---|---|---|
| **S1** | **E1** | QM | QM | QM |
| | **E2** | QM | QM | QM |
| | **E3** | QM | QM | A |
| | **E4** | QM | A | B |
| **S2** | **E1** | QM | QM | QM |
| | **E2** | QM | QM | A |
| | **E3** | QM | A | B |
| | **E4** | A | B | C |
| **S3** | **E1** | QM | QM | A |
| | **E2** | QM | A | B |
| | **E3** | A | B | C |
| | **E4** | B | C | D |

**Table 4 - Determining ASILs**

ASIL = QM (Quality Management) → the function has no impact on safety - it is not necessary to define any safety requirement

ASIL = A → the function has a minimal impact on safety

ASIL = B → the function has an impact on safety – considerable damage

ASIL = C → the function has impact on safety – relatively high damage associated with medium-high probability of being in a situation of risk

ASIL = D → the function is critical - very significant damage associated with a high probability of being in a situation of risk

**Top-level safety requirements**

During the concept phase a *safety goal* shall be defined for each hazardous event. This is a fundamental task, since the safety goal is the top level safety requirement, and it will be the base on which the functional and technical safety requirements are defined. The safety goal leads to item characteristics needed to avert the hazard or to reduce risk associated with the hazard to an acceptable level. Each safety goal is assigned an ASIL value to indicate the required integrity level according to which the goal shall be fulfilled. For every safety goal a *Safe state*, if applicable, shall be identified in order to declare a system state to be maintained or to be reached when the failure is detected, so to allow a failure mitigation action without any violation of the associated safety goal. For each safety goal and safe state (if applicable) that are the results of the risk assessment, at least one safety requirement shall be specified.

## 2.4 ASIL decomposition and allocation in ISO 26262

One of the benefits of ASILs is that they can be decomposed across different elements and subsystems of the system architecture so that the highest constituent ASIL need not apply equally to the entire system; instead, those parts of the system that are most critical and that contribute most to potential failures are allocated higher ASILs and are thus subject to the strictest safety requirements, while less important parts that have little or no bearing on potential failures are allocated lower ASILs and are subject to lower safety requirements. This is important as high levels of safety are much more expensive to implement than lower levels of safety, and if the required level of safety can be achieved for a lower cost, then it is normally desirable to do so.

However, although ISO 26262 defines guidelines and procedures to be followed during ASIL decomposition and allocation, the process is generally a manual and unaided one. This can be problematic in situations where multiple safety-critical functions are delivered by complex, networked architectures. As part of the MAENAD project, we aim to provide both language and tool support for a more systematic approach to ASILs, including the potential for automatic decomposition of ASILs. Specifically, objective O1-1 states that EAST-ADL should be extended to support the safety process of ISO 26262 and provide representation of safety requirements, while objective O1-2 explicitly states a goal of developing support for the automatic allocation of ASILs. The intention is to ensure that the use of ASILs as described in ISO 26262 can be carried out more easily and efficiently even in larger and more complex systems.

### 2.4.1 ASILs in ISO 26262

ISO 26262 sets out a comprehensive methodological framework for the design and development of electronic engineering (EE) systems in the automotive domain. ASILs - and the associated acts of determining and upholding safety requirements - form a major part of this process as they are the mechanism by which the required levels of safety are represented and maintained throughout the design of the system. As such, ISO 26262 contains a considerable amount of information on ASILs and how they should be used. For detailed information, the reader should refer to ISO 26262 itself, but a general overview of the more relevant parts will be given here.

#### 2.4.1.1 Hazard Analysis and definition of ASILs

ASILs are initially derived during the process of defining the safety requirements, i.e., what must be done to ensure a reasonable freedom from unacceptable risk. They are obtained through a combination of functional hazard analysis and risk assessment techniques, starting at an abstract, high-level concept stage (corresponding to EAST-ADL's Vehicle level). These are then developed further and verified at subsequent stages of the design, as represented in increasingly lower, more concrete design views (corresponding to EAST-ADL's Analysis and Design levels).

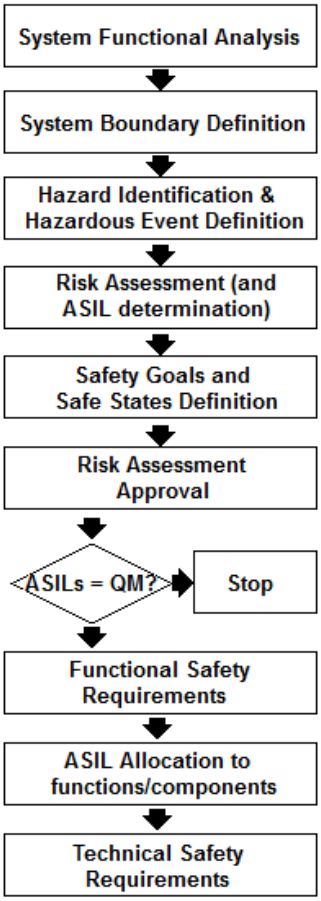



**Figure 3 - Overview of the ISO 26262 safety workflow**

The first step is the identification of hazards and the definition of hazardous events. This initially consists of assessing the 'target feature' of the item under safety analysis and determining its possible malfunctions, i.e., possible anomalies in its output (also known as *feature flaws*). All possible functional anomalies - whether internal, external (e.g. due to misuse), or as a result of maintenance (or the lack thereof) - are considered. These malfunctions form the basis of the hazards - the potential sources of harm that the item can cause. In addition, the various possible operating and environmental scenarios are also considered in which these hazards may occur; taken together, they form the definition of the *hazardous events*, i.e., the occurrence of a given hazard in a particular scenario.

Hazardous Events are then classified in terms of its risk by assigning it an ASIL. This is arrived at by considering the hazardous event from three perspectives: controllability, severity, and exposure, each of which is graded into four categories. The level of risk is determined as a function of the frequency and severity of the hazardous event, and the frequency in turn is determined by the controllability and exposure, so the ASIL can be derived from these factors, as shown in Table 4.

The ASIL itself is captured as part of the *safety goal* corresponding to the hazardous event. For all elements not classified as QM (and it is important to note that QM can only be assigned to elements that are not considered to be safety relevant), a safety goal should be defined. These can result in top-level (i.e., system-level) safety requirements that describe the characteristics the item needs to avert or otherwise mitigate the effects of the hazard in order to achieve an acceptable level of safety. In addition, for every safety goal, a safe state should be defined,

indicating what state the system should maintain or transition to when the hazardous event is detected in order to mitigate that event and achieve the safety goal.

It should also be noted that for a Safety Element out of Context (SEooC), much of this hazard analysis is not possible (due to the lack of contextual information), and so an assumed ASIL is typically assigned instead.

#### 2.4.1.2 ASIL Decomposition

Once defined, the ASILs of the top-level safety goals are propagated throughout the development process as the safety requirements are refined. Each derived safety requirement inherits the ASIL from the safety goal or requirement it was derived from, and so the functional and technical safety requirements that are ultimately allocated to architectural elements all possess a corresponding ASIL.

It is also possible for ASILs to be defined independently of the safety goals when looking at Safety Elements out of Context (SEooC). A SEooC can be a system, subsystem, software or hardware component, or any other part of a system that is safety-critical and designed to be generic and used within more than one system. In this case it is possible to analyse the SEooC with regard to safety, without knowing the context in which the element may operate, by assuming ASIL values and using those values in later decomposition.

These ASILs can be tailored to fit the architecture of the design through the process of *ASIL decomposition*, in which different ASILs are allocated to different elements while ensuring that the overall ASILs as defined by the system-level safety requirements are still being achieved. This process takes advantage of architectural decisions - such as the inclusion of independent or redundant architectural elements - to implement the safety requirements across these elements and thus (potentially) assigning a lower ASIL to the decomposed safety requirements that apply to each individual element without affecting the ASIL that applies to those elements taken together. It is important to note that this can only be achieved where there is a sufficient degree of independence between elements and no critical dependencies (and thus dependent failures) to influence them; this can exclude instances of homogenous redundancy (situations where the same element is duplicated to achieve some measure of redundancy) where elements may suffer from common flaws even if no explicit dependencies exist between elements. ISO 26262 suggests that when deciding the matter of independence, the following should be considered:

- Similar or dissimilar redundant elements;
- Different functions implemented with identical software or hardware elements;
- Functions and their respective safety mechanisms;
- Partitions of functions or software elements;
- Physical distance between hardware elements, with or without a barrier;
- Common external resources.

ISO 26262 also distinguishes between independence (which requires freedom from both cascading/propagating failures as well as common cause failures, and which affects the viability of decomposition) and interference (which requires only freedom from cascading failures and which affects the allocation of QM to sub-elements).

Figure 4 shows how the ASILs can be decomposed such that two (or more) lower ASILs can be equivalent to a higher ASIL. Note that the original ASIL is retained in brackets afterwards. Decomposition can also be applied to elements that are not safety-critical, assuming there is no link between those elements and safety-relevant elements such that failures can cascade from one to the other; in these cases, the non-critical elements can be allocated QM while the safety-related element gains a true ASIL. Decomposition can be applied during multiple stages of the design process, including the concept phase (corresponding to EAST-ADL analysis level and

FAA), system phase (corresponding to the EAST-ADL design level, with FDA for functions and HDA for platform), and software and hardware phases (corresponding to Implementation level and AUTOSAR).

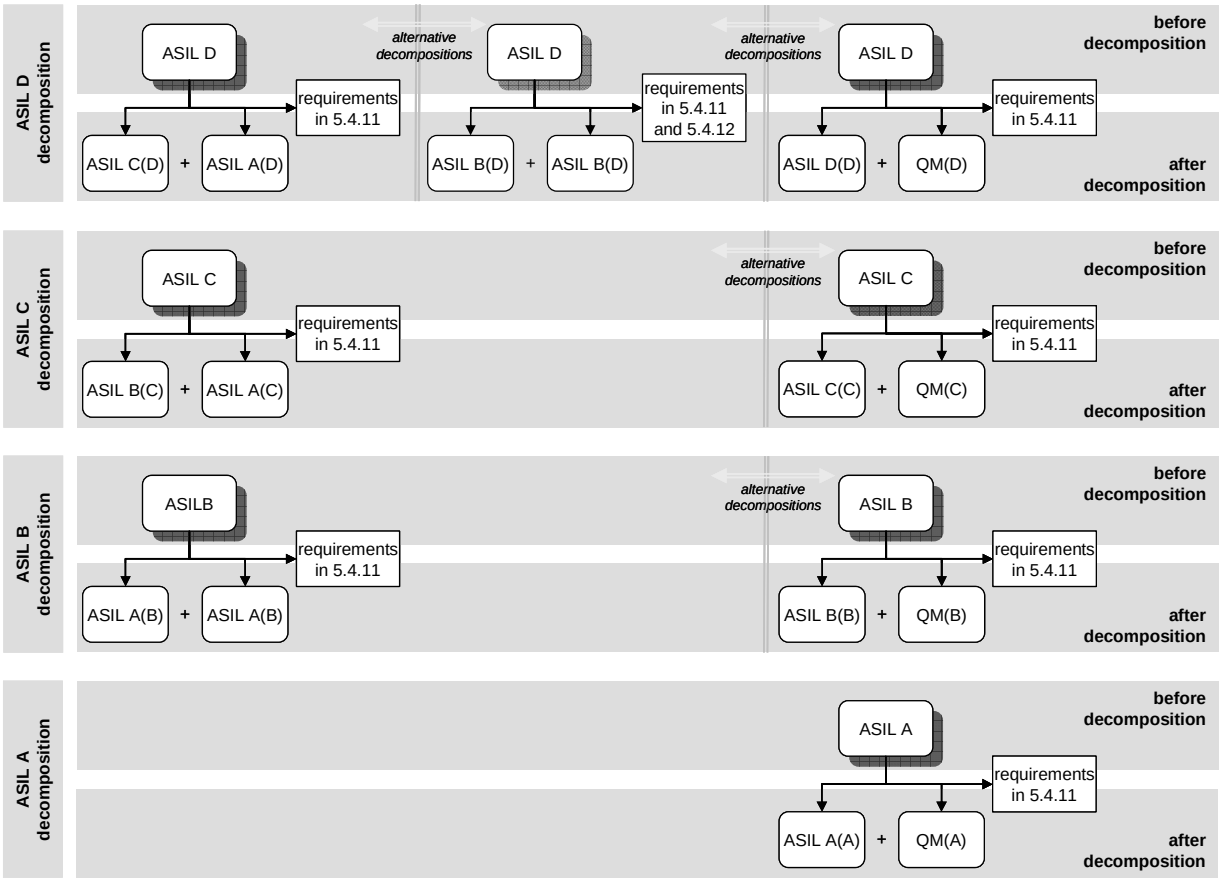



**Figure 4 - Decomposition Algebra**

Figure 5 shows how ASILs can be decomposed across independent channels in an architecture. The modelled system as a whole has ASIL C, indicating there is a relatively severe hazard that must be avoided. Parts of the system that contribute directly to the system failure (such as the Dedicated CAN and the outputs) receive the ASIL directly too. When there is a disjunction, e.g. two or more components can all cause the system failure individually, they all receive the full ASIL. Only when components must fail together in conjunction to cause the system failure is the ASIL diluted. This can be seen in the simple case for Macro-block3 in the diagram, which collectively receives an ASIL C; the subcomponents within (ultimately SWS1-4) may not have the full ASIL C, however, and there are a number of different possible combinations of ASILs that would fulfil the ASIL C for the subsystem as a whole (e.g. A + A + A + A).

A more complex case arises with Macro-block1 and Macro-block2. Both must fail in conjunction to cause system failure, thus both together receive a share of the ASIL C. Because of the AND failure logic (caused by multiplexers) in the "VCS" component, the received ASIL C can be decomposed over Macro-block 1 and Macro-block 2 such that Macro-block 1 receives ASIL A and Macro-block 2 receives ASIL B (which together meet ASIL C), since both are required to cause the VCS to fail. By contrast, Macro-block 3 inherits ASIL C directly since it is single-handedly capable of causing the VCS to fail. It is also worth noting that the ASILs of each of these three Macro-blocks could be decomposed further within those blocks too. For example, in Macro-block1, a failure of either output is all that is necessary, whereas in Macro-block2, a failure of both outputs is needed, so in Macro-block1, all components would require ASIL A (i.e., the ASIL of Macro-block1) since any individual failure is sufficient, but in Macro-block2, both originating contributors (NO and

NC in the EPB Button subsystem) could receive ASIL A (rather than ASIL B, the ASIL of Macro-block2) as both need to fail for Macro-block2 to fail.

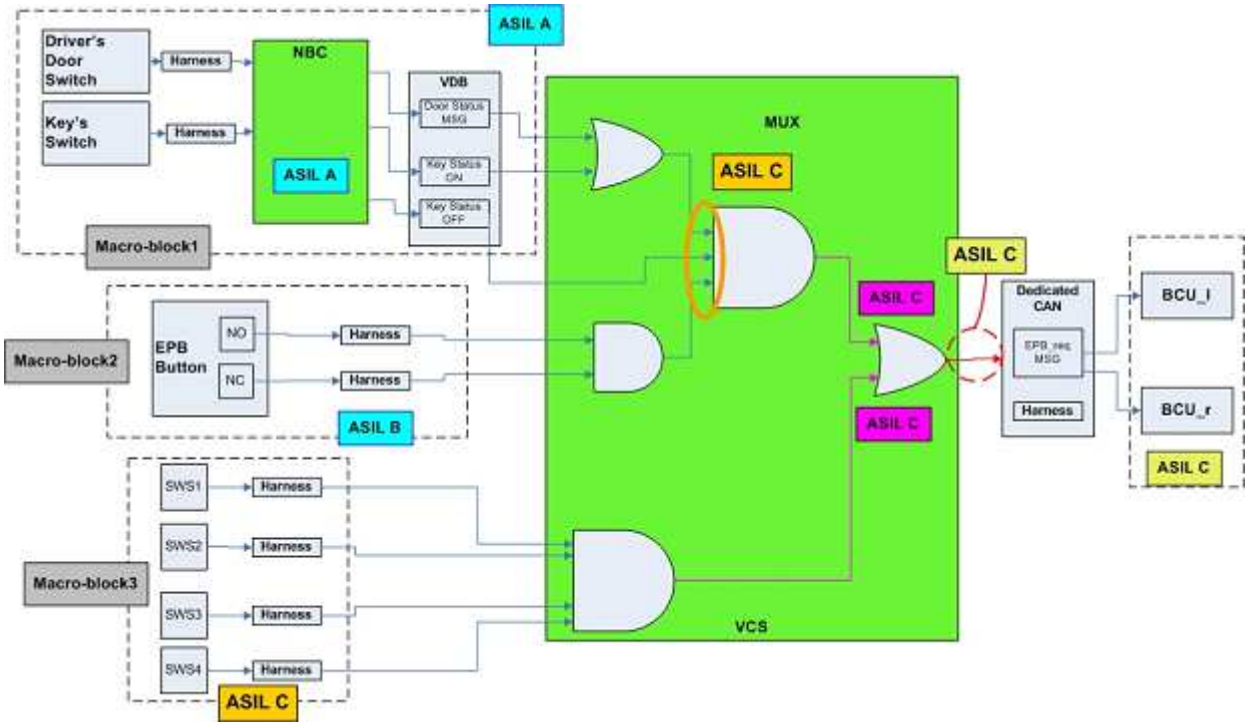



**Figure 5 - ASIL Decomposition example**

2.4.1.3 Safety Analysis in support of ASILs

An assessment of the failure logic of the system and thus a verification of whether elements of the system are indeed independent (without any dependent or common cause failures) can be carried out by a variety of safety analysis techniques. ISO 26262 suggests that qualitative methods should be applied during the abstract system phase and software phase, while quantitative methods can be applied during the hardware phase to allow for the quantification of random hardware failures (although it also suggests common cause failures should still be estimated on a qualitative basis due to the difficulty in accurately quantifying them). Any analysis should take into account:

- Random hardware failures (which may be quantified)
- Development/design faults (e.g. due to errors in design or requirements)
- Installation faults
- Repair/maintenance faults
- Environmental factors (e.g. temperature, pressure, corrosion, electromagnetism)
- Failures of common external resources (e.g. power, communication buses)
- Stress due to specific situations (e.g. wear & tear)

The objective of the safety analyses in these cases is to examine the consequences of the failures on the elements of the system, providing information on whether the safety goals & requirements are being met and also determining whether or not ASIL decomposition will be possible. Analysis may also discover new hazards. ISO 26262 suggests the following analysis techniques:

- Qualitative & quantitative FMEA
- Qualitative & quantitative FTA
- Qualitative & quantitative ETA
- HAZOP (qualitative)
- Markov models (quantitative)
- RBDs (quantitative)

FTA in particular is well suited to performing analysis for the purposes of ASIL decomposition since it can take into account the effects of multiple failures, which e.g. FMEA cannot. Thus it can determine whether or not two or more input failures must occur to cause an output failure (which may suggest the possibility for ASIL decomposition in that case).

In the case of random hardware failures, it is possible to apply quantitative analysis if the appropriate failure data is available. In these cases, quantitative target values for the maximum probability of the violation of each safety goal can be used; ISO 26262 provides a table with one such possible set of values:

| ASIL | PROBABILITY |
|---|---|
| D | $< 10^{-8}\ h^{-1}$ |
| C | $< 10^{-7}\ h^{-1}$ |
| B | $< 10^{-7}\ h^{-1}$ |

**Table 5 - Possible target probabilities for different ASILs**

However, it should be noted that these are only suggestions and can be tailored to fit the circumstances, particularly if e.g. trusted field data is available.

### 2.4.2 ASIL Decomposition & Allocation Algorithm

#### 2.4.2.1 ASIL Decomposition with multiple hazards

As stated in the previous section, one of the primary objectives of MAENAD is not merely to provide support for the ISO 26262 safety design process but also to support the automatic decomposition of ASILs. Because of the large role ASILs play in ISO 26262, having tool support - and particularly a degree of automation - may be very important. While simple examples that deal with only a single hazard - like that in Figure 5 - can be decomposed manually as they do not have too many elements or too complex logic, when multiple hazards are in play, this becomes significantly more difficult, even for small, seemingly simple systems like that in Figure 6. In this example, there is a system (S) that provides two separate outputs, F1 and F2. The six sub-elements of the system (e.g. functions, components etc.), E1 to E6, are the components necessary to provide these two outputs. Each output can fail in one of two ways - omission (lack of provision of output) and commission (unexpected or unwanted provision of output), and thus there are four hazards (omission of F1, commission of F1, omission of F2, and commission of F2) which interact at multiple points.

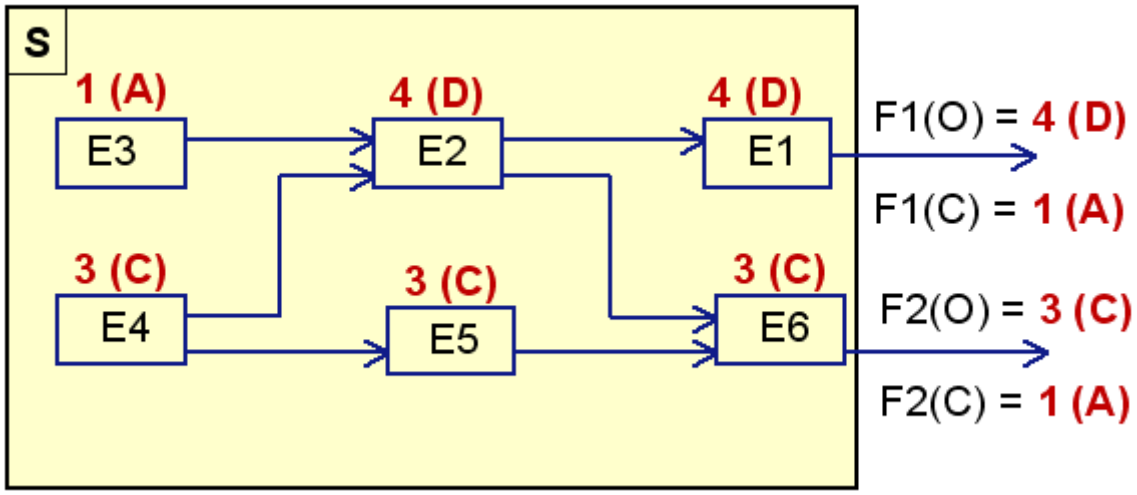



**Figure 6 - Dealing with multiple hazard constraints**

In most examples, there are many alternative ASIL assignment strategies that could be used (as in the case of Figure 5, for example). However, when there is more than one hazard and thus more than one system safety requirement, the competing demands of these multiple requirements can be used to reduce the number of possible assignments. In the more abstract example above, there is in fact only one possible assignment (there are other valid ones, but they are redundant), though this is definitely not apparent at first sight.

Firstly, each of the four hazards has its own assigned ASIL:

- F1(O) - Omission of output from F1, with ASIL D
- F1(C) - Commission of output from F1, with ASIL A
- F2(O) - Omission of output from F2, with ASIL C
- F2(C) - Commission of output from F2, with ASIL A

As stated above, due to the connecting failure logic amongst the six sub-elements in the system, it transpires that there is only one real possible ASIL assignment. To see why, it is necessary to go into more detail. First, the causes of each hazard must be explored. The failure logic of the system is better shown below in Figure 7. In this figure, the propagation of failure causes for the omission failures and propagation of failure causes for the commission failures are shown separately. The black dots represent internal failures of the components they are in and are numbered according to the element they are in, thus failure event FE3 is a failure of element E3. Furthermore, for the sake of simplicity, we assume that omission failures only cause other omission failures and commission failures only cause other commission failures, though in the general case this is not always true, and failures of one type can cause failures of another type.

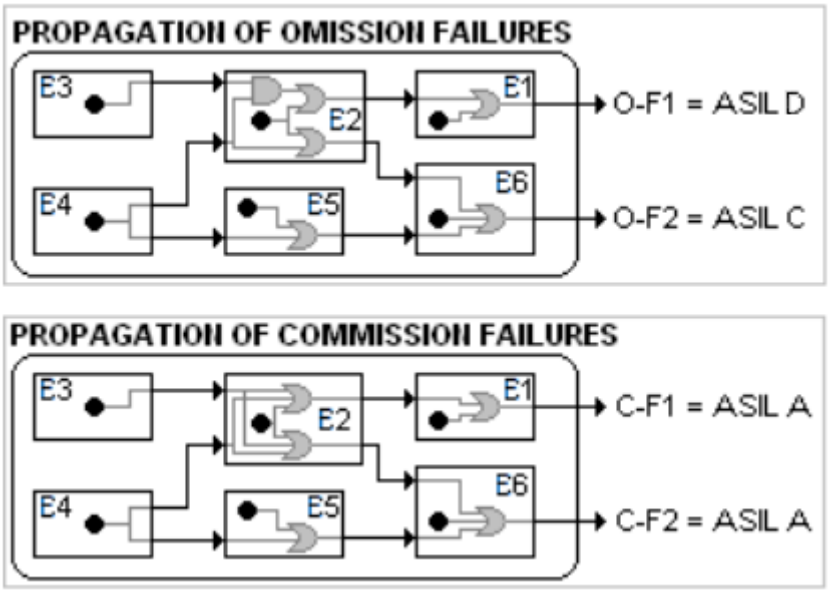



**Figure 7 - Failure logic in the example system**

As an example, take F1(O). Its initial cause is an omission from component E1, which uses OR logic - it can be caused either by an internal failure mode FE1 (the black circle) or by an omission of input. Since FE1 is directly capable of causing the output failure F1(O), it receives ASIL D. Omission of input to E1 is caused solely by an omission from E2, which in turn is caused either by an internal failure mode (FE2) or by an omission of input at *both* inputs. FE2 therefore receives ASIL D as well. However, omission of input to E2 is caused by a conjunction of omissions from both E3 and E4, so they each contribute some share of the ASIL D. There are five possible assignments for these two components:

- FE3 = QM (0), FE4 = D (4) Total = 4 (D)
- FE3 = A (1), FE4 = C (3) Total = 4 (D)
- FE3 = B (2), FE4 = B (2) Total = 4 (D)
- FE3 = C (3), FE4 = A (1) Total = 4 (D)
- FE3 = D (4), FE4 = QM (0) Total = 4 (D)

Assignments where ASILs for both FE3 and FE4 are *more* than sufficient to meet the safety requirement, e.g. FE3 = ASIL C and FE4 = ASIL C, are also possible, but not considered unless necessary (i.e. unless both FE3 and FE4 directly contribute to an ASIL C hazard somewhere). Furthermore, since both FE3 and FE4 can contribute to the system failure, neither should really be allocated QM according to ISO 26262. However, by assigning each ASIL an appropriate value (e.g. D = 4, A = 1), it can be seen that all of these alternatives are equivalent to ASIL D when summed and so all are at least theoretically valid.

To reduce the number of combinations, it is necessary to look at the other hazards and their ASILs. It is determined through decomposition that commission of F1 (i.e., F1 (C)), with ASIL A, is caused by any internal failure mode of E1, E2, E3, and E4; thus each of those failure modes (FE1-FE4) must be at least ASIL A. That firmly eliminates two possibilities for FE3 and FE4 (namely, where one is ASIL D and the other is QM). Next, decomposition of F2(O) – with ASIL C – shows that there are four possible causes: FE2, FE4, FE5, or FE6. Each therefore requires a minimum of ASIL C. This removes two more possibilities from the list, leaving only one possible allocation: FE3 = ASIL A, FE4 = ASIL C.

In this way, ASIL decomposition using multiple hazards allows us to determine which failure modes contribute to which hazards, and by process of elimination, determine what ASIL each

failure mode should have (or, more frequently, determine a set of many possible ASIL allocations for those failure modes). The new algorithm developed in MAENAD (and originated in ATESST2) supports this kind of multi-hazard ASIL decomposition process.

#### 2.4.2.2 Using FTA as the basis of ASIL decomposition

Before ASILs can be decomposed, it is necessary to understand which parts of the system are independent and whether or not failures can cascade from one element to another. FTA - particularly component-based FTA like that performed by the HiP-HOPS tool (see Section 3.3.2) - is ideal for this, as a fault tree naturally analyses the propagation and combinations of failures throughout a system. The failure logic in the previous example, as shown in Figure 7, is effectively a set of interconnected fault trees: there are four top events (one for each hazard), but only six basic events (the internal failures of the six components). Each failure can therefore contribute to more than one top event, and it is this fact that is the basis of the multiple-hazard ASIL decomposition. The actual fault trees used are presented (in a rather unconventional interconnected form) in Figure 8.

Looking at the figure, it is possible to see how FE2 for example contributes to all four top events; in this case it would have to receive the highest of the top event ASILs (in this case, ASIL D). FE4 contributes to all four events, but one of them (O-F1) is via an AND gate, meaning that it is now subject to multiple constraints:

- It must have at least the same ASIL as C-F1 (A)
- It must have at least the same ASIL as C-F2 (A)
- It must have at least the same ASIL as O-F2 (C)
- Together with FE3, it must add up to the same ASIL as O-F1 (D)

Taken together, the most likely assignment here is to give FE4 ASIL C and then FE3 can have ASIL A, assuming it is not subject to any higher ASIL from different branches.

This system of overlapping ASIL constraints both adds complexity to the process and also adds value to it, since it allows us to reduce the total number of possible ASIL assignments (and in general, there are a lot). Furthermore, fault trees are one of the best ways of being able to model these overlapping constraints. Other techniques that deal only with single failures, like FMEAs, would be unsuitable for this as they cannot show the effects of conjunctions of failures, which is what makes decomposition possible.

In particular, compositional FTA (the HiP-HOPS approach) - in which the system fault trees (e.g. O-F1, C-F2 etc.) are synthesised by connecting the individual fault trees of the separate components (E1, E2 etc.) - is ideally suited for this and can readily create such fault trees.

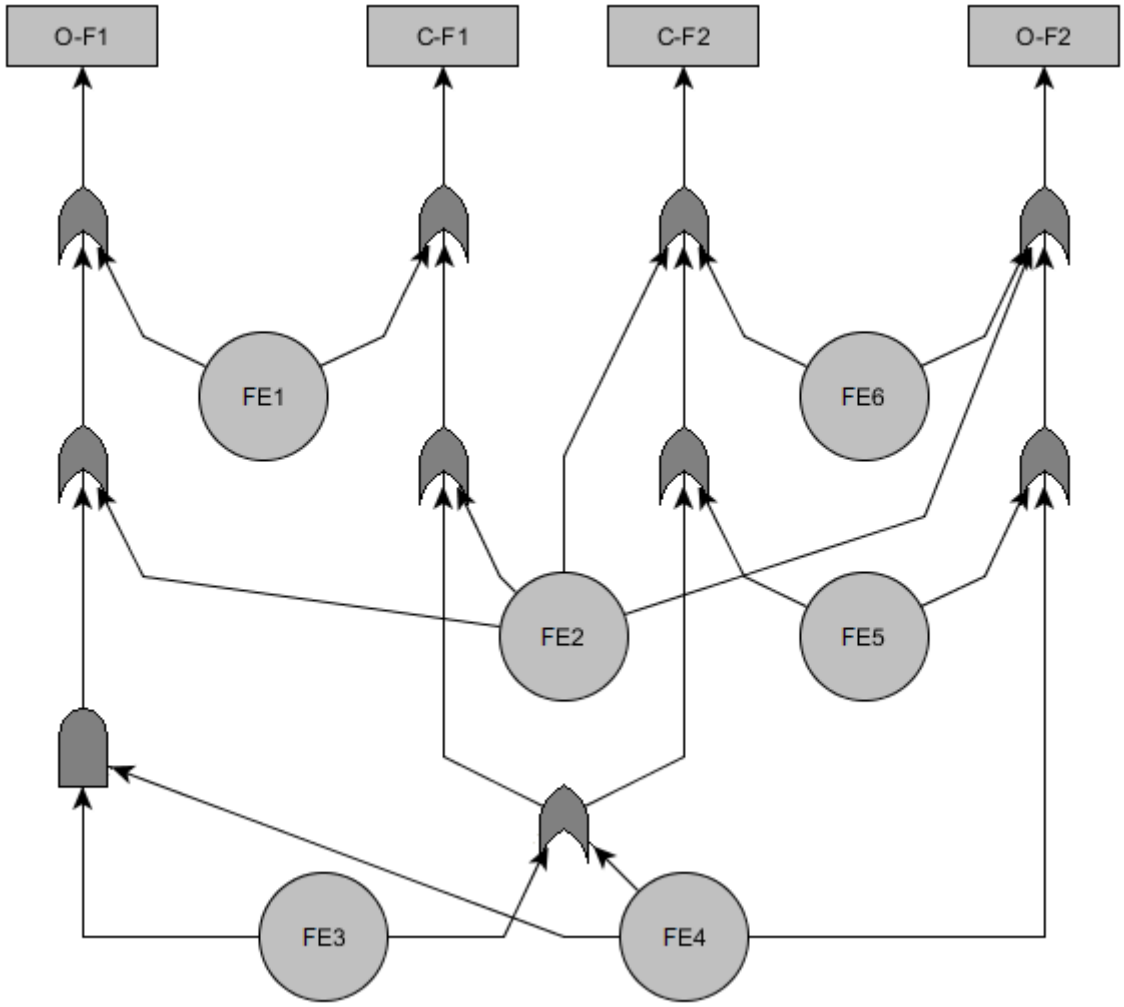



**Figure 8 - Interconnected FTs**

In fact, FTA has an even bigger advantage in that what really matters are the minimal cut sets of the fault trees, i.e., the set of smallest possible combinations of basic events capable of causing the system failures (or hazards in this case). Since these are calculated directly by FTA, it is much easier to see the relationships between the basic events and the hazards they cause. For example (where the black dot represents a conjunction):

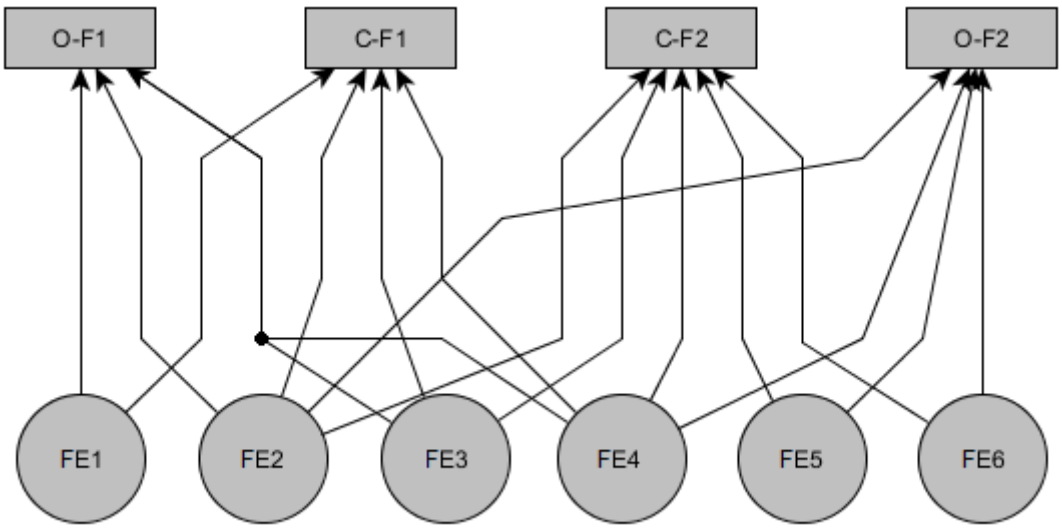



**Figure 9 - Direct connections from basic events to hazards**

On the basis of these minimal cut sets, we can allocate the hazards' ASILs directly to the cut sets, and then (if applicable) decompose them across the basic events that comprise those minimal cut sets.

#### 2.4.2.3 Decomposition Algorithm

The first step in the algorithm is to ensure that a set of *safety constraints* (i.e., ASILs applying to one or more output failures in the system) have been defined. These safety constraints, and their assigned ASILs, may have been produced as a result of a prior hazard analysis of the system (in which case the ASILs are derived from the hazardous events and safety goals) or may simply be assumed in the case of a SEooC. These safety constraints (and their ASILs) constitute one input to the algorithm; the other is a logical representation of the failure behaviour of the system, e.g. as described by the results of a fault tree analysis. Typically, one fault tree must be generated for each safety constraint, modelling the causes of the *constrained output failure* (COF).

Thus:

**Inputs:**

- Set of safety constraints, each of which should have an ASIL and a fault tree generated for its constrained output failure.
- Set of minimal cut sets (from the fault trees modelling the logical failure behaviour of the system) that cause the constrained output failures.

**Outputs:**

- Set of possible ASIL assignments for all basic events in the form of *Safety Requirement Assignments*; there will typically be many of these.

The important entities in the algorithm are as follows:

- **Safety Constraint (SC)**

Represents an ASIL applied to an output failure (malfunction) of an element of the system (or of the system itself). The output failure becomes the top event of a fault tree and its assigned ASIL serves as the basis of the decomposition.

- **Cut Set (CS)**

Cut sets are the results of the FTA. They contain sets of basic events (BEs) and are responsible for causing the COFs.

- **Basic Event (BE)**

A basic event (or BE) is an individual component or function failure in the system. Individually it may or may not have any effect on the system, but if it is a member of a cut set, it will contribute to at least one COF. Basic events are the primary targets for assignment of ASILs from the safety constraints they affect (whether directly or indirectly). When allocated an ASIL from a Derived Safety Requirement (or DSR - see next), the DSR also 'locks' them at that ASIL so that only DSRs with higher ASILs can change them until the original DSR releases the lock. This prevents BEs being incorrectly allocated lower ASILs, so a BE can always be allocated a higher ASIL, but never a lower one.

- **Derived Safety Requirement (DSR)**

In the context of the algorithm, a Derived Safety Requirement describes an ASIL value that a set of basic events (i.e., a minimal cut set) must implement to achieve required system risk levels. DSRs are produced by the ASIL decomposition and allocation process. More than one DSR may apply to the same basic event, in which case the highest ASIL from all the DSRs that apply to that BE is used.

- **Safety Requirement Allocation (SRA)**

A Safety Requirement Allocation is the output of the algorithm and represents a set of possible ASIL assignments to the basic events of the system that will meet the initial safety requirements as defined by the Safety Constraints. Typically many SRAs will be generated.

The algorithm itself is described in pseudo-code below.

### ASIL Decomposition Algorithm

```
for each safety constraint (SC):
{
  for each cut set (CS) in SC:
  {
    Generate new 'Derived Safety Requirement' or DSR (contains events and ASIL from SC)
  }
  Sort the DSRs - least basic events first, then highest ASIL first.
}

Recurse for every derived safety requirement (DSR):
{
  while there remain unallocated DSRs:
  {
    if this DSR applies to only one basic event:
    {
      Assign DSR's ASIL to the only BE
      If the assignment succeeded and BE's ASIL is >= DSR's ASIL:
      {
        BE is locked at that ASIL by this DSR {*1*}
        Recurse for next DSR
      }
      else
      {
        Recurse for next DSR
      }
      Unlock BE
    }
    else (DSR applies to more than one BE)
    {
      Iterate through all possible assignments of DSR's ASIL for these BEs {*2*}
      {
        If this assignment meets the DSR's ASIL:
        {
           Attempt to assign ASILs to BEs {*1*}
           If it worked and the DSR is achieved, recurse for next DSR
        }
      }
      Unlock all BEs
    }
  }
```

```
  else (there are no remaining DSRs)
  {
    Store the current configuration of BEs & ASILs as one possible assignment (SRA) {*3*}
    Backtrack up recursion stack to try new combinations
  }
}
```

The above algorithm describes the overall process, but it is worth describing certain aspects (numbered) in more detail:

**1) Allocating ASILs to Basic Events**

The key to the algorithm is the priority locking mechanism used by the Basic Events. When a Derived Safety Requirement attempts to allocate an ASIL to a Basic Event, this operation only succeeds if one of the following is true:

- The Basic Event is not locked
- The BE is locked, but the DSR is the one who holds the most recent lock
- The DSR's ASIL is higher than the ASIL of the current lock holder

When the assignment succeeds, the DSR that allocated it locks the Basic Event and registers itself with the Basic Event as the current lock holder (the BE retains a stack of lock holders). This ensures that a Basic Event can only be given an equal or higher ASIL than the one it currently holds, never a lower one (for this purpose, an ASIL is equated to a numeric value, e.g. A = 1, B = 2, C = 3 etc.). If the DSR's ASIL is lower, then the new assignment fails and the algorithm will potentially backtrack to try a new combination of assignments.

**2) Iterating through possible combinations of assignments for a set of BEs**

There can be many possible combinations of assignments for any given set of BEs. The theoretical maximum number of permutations is given by the formula:

$$p = (m + 1)^n$$

where $p$ is the number of permutations, $m$ is the maximum ASIL value (4 in this case, but then we add 1 because we include 0/QM), and $n$ is the number of basic events constrained by the SR. So for a 2 event cut set, there is a maximum of 25 possible assignments; for 3 events, there are 125.

Each assignment is determined mechanically, so it will start with e.g. 0/0/0, then 0/0/1, then 0/0/2 etc., and assign each of these values to the targets. However, not all of these permutations are used. Firstly, those that do not meet the current DSR are immediately discarded; for example, if the required ASIL value is 3 and the current permutation is 0/0/1 (sum = 1, which is less than 3), this would not even be tested. Secondly, it is possible for one or more of the BEs to already have been locked by earlier DSRs; in these cases, permutations in which those BEs have lower ASIL values are not even enumerated and are thus skipped. This helps to improve the efficiency of the process. Note however that assignments that *more* than meet the DSR are kept at this stage, so 1/2/3 (total = 6) would be kept even if the required total was only 3.

**3) Storing possible assignments**

Once all safety requirements have been examined and all of their ASILs assigned to basic events in such a way that all of the DSRs are still being achieved (i.e., no combination of basic event ASILs add up to less than the ASILs of any of the hazards they cause), that assignment of ASILs and corresponding basic events is stored as a Safety Requirement Allocation (SRA). Safety Requirement Allocations are nothing more than dictionaries using basic events as keys and allocated ASILs as their corresponding values.

However, there can be a great many Safety Requirement Allocations and it is important to cut down the number wherever possible. Therefore, we can do some redundancy checking at this stage. The main form of checking is to see whether one allocation renders another redundant. For example, if basic event BE1 is assigned ASIL C and BE2 is assigned ASIL B, and that is a valid allocation (i.e., all DSRs are met), then an allocation in which BE1 is assigned ASIL D and BE2 is assigned ASIL C would be redundant. To borrow an optimisation term, it is *dominated* by the first allocation. It is important to note that this is only the case if *all* of the ASILs are higher than (or equal to) another allocation; if BE1 was ASIL B and BE2 was C, then it would not be redundant.

To extend the example, assuming two BEs in total, BE1 and BE2, and four allocations added in the following order:

| | BE1 | BE2 | TOTAL | |
|---|---|---|---|---|
| 1) | C (3) | B (2) | D (5) | |
| 2) | C (3) | A (1) | D (4) | - Makes #1 redundant, so #1 is removed |
| 3) | D (4) | A (1) | D (5) | - Is made redundant by #2, so not added |
| 4) | B (2) | B (2) | D (4) | - Is *not* redundant |

In this case there are two non-redundant allocations (#2 and #4) and two redundant ones (#1 and #3). By keeping the number of allocations to a minimum throughout the process, we reduce the amount of checking we need to do when we add later SRAs. Even so, this can still be an expensive process, as every new SRA needs checking against every existing SRA. There are potential optimisations possible here, however.

#### 2.4.2.4 Example

To see how the algorithm works when applied to the previous 6-component example in Figure 6, we can work through the algorithm manually (albeit skipping the sorting for illustrative purposes).

First, we need a list of minimal cut sets, together with the hazards they cause and associated ASILs:

**O-F1 - ASIL D (4)**

{FE1} {FE2} {FE3.FE4}

**C-F1 - ASIL A (1)**

{FE1} {FE2} {FE3} {FE4}

**O-F2 - ASIL C (3)**

{FE2} {FE4} {FE5} {FE6}

**C-F2 - ASIL A (1)**

{FE2} {FE3} {FE4} {FE5} {FE6}

We can now generate a set of Derived Safety Requirements, one for each cut set. This tells us what different ASILs apply to each cut set. We can represent these in the form of equations, for clarity:

| | | | | |
|---|---|---|---|---|
| {FE1=4} | {FE2=4} | {FE3 + E4=4} | | |
| {FE1=1} | {FE2=1} | {FE3=1} | {FE4=1} | |
| {FE2=3} | {FE4=3} | {FE5=3} | {FE6=3} | |
| {FE2=1} | {FE3=1} | {FE4=1} | {FE5=1} | {FE6=1} |

Next we have to recurse through all of these DSRs, attempting to lock the Basic Events at each point. To make it clear what's happening, the status of the BEs (and their locks) is shown at each point. Bold event values indicate a successful lock; italic values indicate failure. We shall only examine the first full recursion.

| | **FE1** | **FE2** | **FE3** | **FE4** | **FE5** | **FE6** |
|---|---|---|---|---|---|---|
| 1. FE1=4 | **4 (#1)** | - | - | - | - | - |
| 2. FE2=4 | 4 (#1) | 4 (#2) | - | - | - | - |
| 3. For 3, we have two events, so we need to iterate through all permutations that add up to at least 4. We shall consider these separately. | | | | | | |
| 3a: FE3=0, FE4=4: | | | | | | |
| | 4 (#1) | 4 (#2) | **0 (#3a)** | **4 (#3a)** | - | - |
| 4. FE1=1 | *4 (#1)* | 4 (#2) | 0 (#3a) | 4 (#3a) | - | - |
| 5. FE2=1 | 4 (#1) | *4 (#2)* | 0 (#3a) | 4 (#3a) | - | - |
| 6. FE3=1 | 4 (#1) | 4 (#2) | **1 (#6)** | 4 (#3a) | - | - |
| 7. FE4=1 | 4 (#1) | 4 (#2) | 1 (#6) | *4 (#3a)* | - | - |
| 8. FE2=3 | 4 (#1) | *4 (#2)* | 1 (#6) | 4 (#3a) | - | - |
| 9. FE4=3 | 4 (#1) | 4 (#2) | 1 (#6) | *4 (#3a)* | - | - |
| 10. FE5=3 | 4 (#1) | 4 (#2) | 1 (#6) | 4 (#3a) | **3 (#10)** | - |
| 11. FE6=3 | 4 (#1) | 4 (#2) | 1 (#6) | 4 (#3a) | 3 (#10) | **3 (#11)** |
| 12. FE2=1 | 4 (#1) | *4 (#2)* | 1 (#6) | 4 (#3a) | 3 (#10) | 3 (#11) |
| 13. FE3=1 | 4 (#1) | 4 (#2) | *1 (#6)* | 4 (#3a) | 3 (#10) | 3 (#11) |
| 14. FE4=1 | 4 (#1) | 4 (#2) | 1 (#6) | *4 (#3a)* | 3 (#10) | 3 (#11) |
| 15. FE5=1 | 4 (#1) | 4 (#2) | 1 (#6) | 4 (#3a) | *3 (#10)* | 3 (#11) |
| 16. FE6=1 | 4 (#1) | 4 (#2) | 1 (#6) | 4 (#3a) | 3 (#10) | *3 (#11)* |

We have now run out of DSRs and so can store this as one possible SRA for the system:

FE1 = 4, FE2 = 4, FE3 = 1, FE4 = 4, FE5 = 3, FE6 = 3

As it turns out, the next recursion gives us a better solution. When we backtrack, we go back to the last time we have more than one basic event in a SR and were iterating through possible permutations. In this case, that SR was #3, so we can skip back to there and continue with #3b instead:

3b: FE3=1, FE4=3:

| | | | | | | |
|---|---|---|---|---|---|---|
| | 4 (#1) | 4 (#2) | **1 (#3b)** | **3 (#3b)** | - | - |
| 4. FE1=1 | *4 (#1)* | 4 (#2) | 1 (#3b) | 3 (#3b) | - | - |
| 5. FE2=1 | 4 (#1) | *4 (#2)* | 1 (#3b) | 3 (#3b) | - | - |
| 6. FE3=1 | 4 (#1) | 4 (#2) | *1 (#3b)* | 3 (#3b) | - | - |
| 7. FE4=1 | 4 (#1) | 4 (#2) | 1 (#3b) | *3 (#3b)* | - | - |
| 8. FE2=3 | 4 (#1) | *4 (#2)* | 1 (#3b) | 3 (#3b) | - | - |
| 9. FE4=3 | 4 (#1) | 4 (#2) | 1 (#3b) | *3 (#3b)* | - | - |
| 10. FE5=3 | 4 (#1) | 4 (#2) | 1 (#3b) | 3 (#3b) | **3 (#10)** | - |
| 11. FE6=3 | 4 (#1) | 4 (#2) | 1 (#3b) | 3 (#3b) | 3 (#10) | **3 (#11)** |
| 12. FE2=1 | 4 (#1) | *4 (#2)* | 1 (#3b) | 3 (#3b) | 3 (#10) | 3 (#11) |
| 13. FE3=1 | 4 (#1) | 4 (#2) | *1 (#3b)* | 3 (#3b) | 3 (#10) | 3 (#11) |
| 14. FE4=1 | 4 (#1) | 4 (#2) | 1 (#3b) | *3 (#3b)* | 3 (#10) | 3 (#11) |
| 15. FE5=1 | 4 (#1) | 4 (#2) | 1 (#3b) | 3 (#3b) | *3 (#10)* | 3 (#11) |
| 16. FE6=1 | 4 (#1) | 4 (#2) | 1 (#3b) | 3 (#3b) | 3 (#10) | *3 (#11)* |

Notice how this time, no overriding took place. Our allocation this time is:

FE1 = 4, FE2 = 4, FE3 = 1, FE4 = 3, FE5 = 3, FE6 = 3

and when we compare that to the previous one, we find the only difference is that this one has a lower ASIL for FE4. Therefore, this new allocation renders the previous one redundant, so the old one is removed and the new one is kept. As it turns out, in this example, this is the only undominated allocation, and it would render all other allocations (like FE1=4, FE2=4, FE3=2, FE4=3, FE5=3, FE6=3) redundant.

In cases where there are more results, these will typically be sorted (e.g. by total ASIL value, or by weighted total where A=1, B=10, C=100, D=1000 etc.) to make it easier for the analyst to absorb the results by presenting the most attractive possibilities first. However, only the analyst can decide for sure what is a good allocation and what is not.

#### 2.4.2.5 Scope of the decomposition

Although the algorithm decomposes and allocates ASILs directly to internal faults of system functions/components, it is also possible - once this has been done - to assign ASILs to other things, including propagated failures (or deviations), the ports that define the interface of a function/component, or even the function/component as a whole (e.g. to represent a process fault). This is done simply by taking the highest ASIL that propagates through a given port or the highest ASIL that passes through a component. The raw information provided by ASILs allocated to basic events is very flexible and this information can be used to derive allocations to virtually any other attribute or part of the system model.

### 2.4.3 HiP-HOPS and Tool Support for ASILs

#### 2.4.3.1 Current HiP-HOPS Support

The algorithm described in Section 2.4.2 is already implemented as part of HiP-HOPS, though only in prototype form and likely not bug-free. It can be activated by including a new command line option: "decomposeSILs=true".

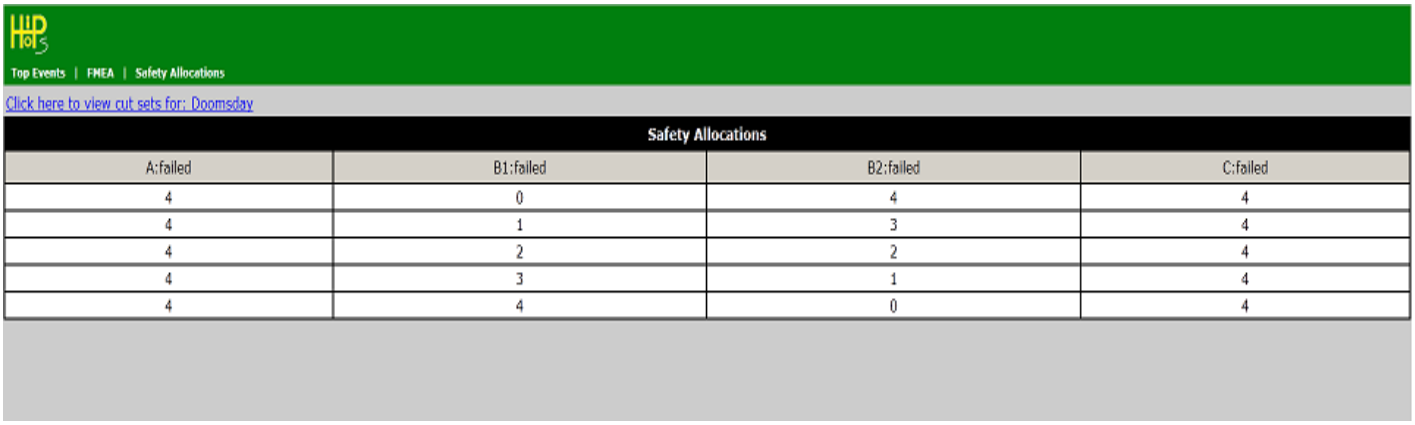
The main input to the process is the Hazard, which is included as part of the new XML interface to HiP-HOPS. It is intended for Hazards to replace the older System Outport style of system failure definition, as they are more flexible and more useful (e.g. to support ASIL decomposition).

The XML *Model* element contains an optional *Hazards* element, which in turn contains any number of *Hazard* elements defining the system-level hazard failures. Each Hazard contains:

- Name (string)
- Description (string, optional)
- Safety Requirement (integer, optional) - represents the ASIL. Uses integers, so ASIL D = 4, ASIL C = 3, ASIL B = 2, ASIL A = 1, QM = 0.
- Severity (double, optional)
- A Hazard Expression (string) - a logical expression consisting of one or more output deviations in any perspective of the model connected by AND or OR operators. Advanced operators (NOT, temporal) are not applicable here (yet).

This is all the information that HiP-HOPS needs to commence ASIL decomposition, as it obtains the fault trees and cut sets itself by analysing the model it is given. The algorithm has not been tested on large models and it is anticipated that the process will be relatively slow for larger numbers of basic events and safety requirements (due to the combinatorial explosion nature of enumerating all possible valid allocations of ASILs). More control over the process (e.g. timeouts, logging information) could be provided by adding further command line parameters.

At the other end, the output can be included (assuming ASIL decomposition takes place) as part of the HTML-style output currently produced by HiP-HOPS:


Top Events | FMEA | Safety Allocations

Click here to view cut sets for: Doomsday

**Safety Allocations**

| A:failed | B1:failed | B2:failed | C:failed |
|---|---|---|---|
| 4 | 0 | 4 | 4 |
| 4 | 1 | 3 | 4 |
| 4 | 2 | 2 | 4 |
| 4 | 3 | 1 | 4 |
| 4 | 4 | 0 | 4 |

**Figure 10 - Example ASIL decomposition output**

The output shows each basic event in a different column and then each row contains a different ASIL allocation that should meet the original ASILs defined in the Hazard. However, it may be that a more formalised output needs to be decided on, particularly if the results are to be imported back into the model (see below).

#### 2.4.3.2 Plugin Extension

To provide rudimentary support for ASIL decomposition, the EAST-ADL/HiP-HOPS plugin should be extended to support the output of hazards in the newer HiP-HOPS XML input format. This information is derived from the SafetyConstraints in the EAST-ADL model. It also needs to be possible to activate the process by sending the new command line parameter (when appropriate).

#### 2.4.3.3 Expanding the scope of decomposition

As currently implemented, HiP-HOPS theoretically allows ASILs to be assigned to any cut set-capable event, thus including things like input and output deviations. However, due to the way HiP-HOPS handles these extended node types (they do not ordinarily appear in the results and in fact usually get stripped out early in the analysis process), actually performing decomposition and allocation with them is not necessarily a straightforward proposition.

In practice, the simplest solution may in fact be to perform decomposition as it stands and then go back to the original trees and propagate the ASILs upwards from the basic events. This would ensure that all input/output deviations receive the correct ASILs. In case of conflicts, the highest ASIL received would be retained and, in theory, the ASILs of the system output deviations should match the ASILs of the hazards they cause. Doing this would also allow us to assign ASILs to ports relatively easily by taking the highest ASIL of any deviation at that port.

Allocation of ASILs to components and subsystems is substantially easier since we already know which basic events belong to which components. By simply taking the highest ASIL of all the basic events and deviations of a component, we can 'allocate' that ASIL to the component. Or, in the case of subsystems, do the same thing but take the highest ASIL of its subcomponents instead.

#### 2.4.3.4 Storing the results

During the York meeting in May 2011, it was decided that the goal for the first year of MAENAD would be to present the ASIL decomposition results in HiP-HOPS form only: the aim is to present a list of possible Safety Requirement Allocations (as in Figure 10) and allow the user to select one; this would then present a new display showing a tree-based view of the system and how the ASILs

are assigned to its constituent elements (specifically, the basic events, the input/output deviations, and the component/process fault ASIL).

More substantial work would be needed to be able to import the results back into the EAST-ADL model. To begin with, the decision would have to be made whether *all* possible SRAs would be stored, or whether the user would select the most appropriate allocation and only that one would be stored. In the latter case, the existing EAST-ADL language elements, e.g. the SafetyConstraint, would likely be sufficient to store the result (unless we also wished to be able to allocate ASILs to other parts of the model too), but some kind of selection dialogue would be necessary. However, in the former case, we would need to create some kind of storage mechanism to be able to map the different possible ASIL assignments to their targets without interfering with each other.

On a technical level, we would also need to import the results from the HiP-HOPS output files back into whichever modelling tool (e.g. Papyrus) was being used. This has not yet been done even for ordinary fault tree results, however. It would likely involve a consolidation of HiP-HOPS output into a single XML file to make it easier for the file to be imported back into the EAST-ADL model. The plugin may also need to perform some further model transformation on these results to know where they should be stored.

## 2.5 Discussion & Further Work

### 2.5.1 Support for ASILs

ASIL decomposition and allocation is an important objective for MAENAD and a major requirement in order to be able to fully support ISO 26262-compatible safety-driven design. This section details a newly-developed algorithm that enables the automatic decomposition and allocation of ASILs across independent elements of the system by building upon earlier work on FTA; ASILs assigned to hazards can then be decomposed to the minimal cut sets that cause those hazards. By enumerating the different permutations of those ASILs assigned to multi-event cut sets in a recursive process, it is possible to determine all possible valid ASIL allocations for the basic events of a system while ensuring that the resulting allocations are still capable of meeting the original safety requirements. EAST-ADL language support is relatively mature and language elements for both hazard analysis and ASILs are present. However, it may be that these need tweaking or extending to streamline the process in response to practical experience.

#### 2.5.1.1 Existing Support for ASIL decomposition

At present, much of the infrastructure required to support ASIL decomposition and analysis is already present in both EAST-ADL and HiP-HOPS. In particular, EAST-ADL supports:

- Hazard analysis and definition of Hazards, HazardousEvents, and SafetyGoals. HazardousEvents and SafetyGoals can both store ASIL values.
- SafetyConstraints can be used to assign ASILs to elements of the error model.
- SafetyConstraints may also provide a mechanism for linking the resulting ASIL allocations back to the faults & failures of the error model after decomposition.

The automatic ASIL decomposition and allocation algorithm has been implemented in HiP-HOPS, along with preliminary support for input and output. This may need extending in the future but is sufficient for early testing and experimentation. The EAST-ADL/HiP-HOPS plugin requires

extending to support this capability but the impact on the input interface to HiP-HOPS is hopefully relatively minor. HiP-HOPS can also produce output in its existing HTML-style format.

#### 2.5.1.2 Further work and potential changes

Although an algorithm has been developed to enable automatic ASIL decomposition, and although a lot of the foundation for supporting that algorithm now exists both in the EAST-ADL language and in the HiP-HOPS tool, there is still plenty of scope for further work in all areas. Much of that work revolves around the following issues.

#### 2.5.1.3 Linking the language support to the tools

EAST-ADL has sufficient language support to enable the decomposition and the algorithm has been implemented in the analysis tool. The main obstacle at present is the link between the two: namely, a plugin to an EAST-ADL modelling environment, such as Papyrus. The existing HiP-HOPS plugin for Papyrus would need extending to enable the output of hazards and ASIL information to HiP-HOPS and allow the starting of the decomposition process.

However, a prototype ASIL analysis capability has been implemented in the HiP-HOPS export plugin for EPM. This will enable us to begin testing the decomposition of EAST-ADL models with the algorithm, which will highlight bugs to be fixed and other areas for further work (e.g. any streamlining or clarification of the existing language elements, any additions necessary to the language, what sort of bugs exist in the tools, and how efficient and scalable the algorithm is).

#### 2.5.1.4 Presentation and storing of the results

Currently the results are presented as part of the standard HTML-style HiP-HOPS results. It is also sorted in a rudimentary fashion. However, some additional forms of filtering and/or sorting could be applied to the results as well, e.g. using a point-based heuristic that favours more lower ASILs over fewer higher ASILs. Also, the results currently only display what is essentially the raw assignment information: the direct assignment of ASILs to basic events. More development is necessary to be able to display the results in a more refined form, or display assignments of ASILs to other elements (e.g. output/input failures).

A secondary issue is that of storing the results back in the model. This is a general unsolved issue not specific to ASILs (e.g. the fault tree results are currently completely external as well) but is something to be investigated further.

#### 2.5.1.5 Scope of decomposition

At present, the algorithm focuses solely on allocating the ASILs to the basic events (component/function failures) of the system. This is done partly because it links in most closely with the way FTA and its analysis of system failure propagation works, and partly because it also offers a very sound foundation for further expansion into assignment to other model or system elements (e.g. ports or entire components).

The goal for MAENAD is to expand this to also allow assignment to input/output failures and entire components (to represent process faults). This will require an extension of the prototype implementation in HiP-HOPS as well as a new method of displaying the results. However, at the same time, it should not impede preliminary testing with existing support for ASILs.

#### 2.5.1.6 Scalability of the algorithm and optimisation

Due to the nature of the problem - the number of possible permutations increases very rapidly with the number of AND gates and basic events, leading to combinatorial explosion - it is unlikely that the algorithm is very scalable. Quite how scalable it is has yet to be fully tested. In any case, it can almost certainly be made more efficient in certain areas and there may even be other approaches that help overcome its difficulties. This does warrant some further investigation.

Furthermore, a more radical alternative may be to re-frame the process of ASIL allocation as an optimisation problem and employ tabu search to solve it. This would enable us to employ existing HiP-HOPS technology and may result in a more efficient process. However, it would obviously also mean investigating how best to go about doing this and would entail both theoretical and practical work on HiP-HOPS and its optimisation capabilities.

At the York meeting in May 2011 it was decided to attempt to develop prototype ASIL optimisation support by September 2011 to determine the feasibility of the process. This algorithm is described below in section 2.6.

#### 2.5.1.7 ASIL Decomposition and Hardware

During ASIL decomposition, it is not possible to manage hardware the same way as the functionality it hosts. Rather, the entire set of hardware components supporting a given ASIL X safety requirement (i.e., a safety constraint with ASIL = X) must be considered together. This is because the failure rate goal for hardware components is provided as a numerical range for each ASIL level in ISO 26262. Assuming that a large number of components are used to decompose an ASIL, their overall failure rate should still be less than the goal failure rate indicated by the ASIL. This is particularly important if the components are connected in series, such that any single failure will cause a system hazard to occur, as in that case the *combined* failure rate needs to be less than the failure rate indicated by the ASIL.

As a result, hardware errors need to be analysed quantitatively to ensure that both the components and the overall architecture meet the ASIL failure rate requirements. Nor can the hardware be considered only in isolation — failure propagation and therefore the quantitative analysis results are dependent on the allocation of functions to hardware components. Only by using a combined error model showing the propagations from both hardware elements and the software functions allocated to them can an accurate estimation of the overall failure probability, and thereby determining whether the ASIL is being met.

#### 2.5.1.8 Examples and Case Studies

As always, it would be highly beneficial to have some larger scale examples to work with, both for bug hunting purposes and to ensure that we are properly supporting an ISO 26262 compatible ASIL decomposition. Furthermore, small but realistic examples would be very valuable as case studies to use in papers etc. for dissemination purposes.

At the York meeting in May 2011, VTEC agreed to develop a brake-by-wire example for use with ASIL decomposition & allocation. This is currently in progress but awaiting full tool support to be able to undertake an ASIL decomposition/allocation.

## 2.6 Optimisation algorithm for ASIL Decomposition

The ASIL allocation and decomposition problem can also be solved as an optimisation process. This differs to architectural optimisation (discussed in section 4) in that the domain is the set of possible ASIL allocations to the basic events (or other elements) of the system, rather than the set of possible architectural configurations in the model. Furthermore, the scope of the evaluation is much more limited since the other factors that count are whether or not an ASIL allocation is valid, i.e., whether or not it meets its requirements, and how 'expensive' the allocation is.

### 2.6.1 The ASIL Optimisation Process

The inputs to the optimisation are the same as to the standard ASIL allocation/decomposition algorithm: namely, the set of constrained output failures (COFs) and the safety requirements derived (DSRs) from them that apply to the causes of those output failures:

**Inputs:**

- Set of safety constraints, each of which should have an ASIL and a fault tree generated for its constrained output failure.
- Set of minimal cut sets (from the fault trees modelling the logical failure behaviour of the system) that cause the constrained output failures.

Similarly, the output is also the same – a set of safety requirement assignments (SRAs):

**Outputs:**

- Set of possible ASIL assignments for all basic events in the form of *Safety Requirement Assignments*.

The difference is in how these SRAs are obtained. Whereas the standard algorithm essentially performs a brute force search through the entire optimisation space to find all valid solutions, the optimisation will perform a more randomised search, guided by heuristics, in an attempt to obtain a reasonable set of solutions without having to search through every possible valid solution. The technique applied in this project is called Tabu Search (TS), an algorithm inspired by the human memory mechanism initially presented by Glover [37][38].

### 2.6.2 Solution Representation

The solution representation/encoding used for the ASIL decomposition optimisation is much simpler than that the one used for the architectural optimisation. Rather than having a hierarchical tree structure to represent the makeup of the system, the ASIL optimisation simply uses the set of relevant basic events, i.e., Failure Modes (FMs) and assigns each of them an ASIL value between 0 (QM) and 4 (ASIL D). Thus each solution can in fact be represented as a string of numbers between 0 and 4.

**Figure 11 - Example of Tabu Search ASIL Solution Representation**

### 2.6.3 Cost Calculation

The optimisation needs to be aware on cost varies with component ASIL-dependant-implementation to steer the search. However, at the time of design, components may be completely new developments, and ASIL-dependant prices may not be available. The technique introduced here overcomes that limitation by allowing the user to define ASIL cost heuristics to guide the search based on information of average costs information from previous experience. An example of such cost heuristic could be a logarithmic one:

- QM = 0 points
- ASIL A = 10 point
- ASIL B = 100 points
- ASIL C = 1000 points
- ASIL D = 10000 points

For an illustrative ASIL assignment solution with 5 FMs, the cost calculation, using the logarithmic heuristic, is shown below.

| FM1 | FM2 | FM3 | FM4 | FM5 | |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 2 | 0 | |
| ASIL A | ASIL C | ASIL D | ASIL B | ASIL QM | |
| 10 | 1000 | 10000 | 100 | 0 | = 11110 |

**Figure 12 - ASIL assignment cost calculation example**

The other steering parameter of the optimisation technique is the feasibility of an ASIL assignment solution. Each cut set in the FTA results is subject to one or more DSRs – derived safety requirement – that indicates the minimum ASIL that cut set must achieve in order to meet the overall system safety requirements. The TS technique never visits solutions which do not meet all the DSRs.

### 2.6.4 Tabu Search Algorithm for Optimisation

**Overview:**

The technique introduced in 2.4.2.3 searches performs an exhaustive search of the ASIL decomposition solution space. Although it has suffered important improvements over the time, due to the combinatorial nature of the problem, scalability remains an issue. The fact that there is no standardized ASIL cost function complicates the matter of formulating a specialized optimization algorithm. Thus, we directed our research to meta-heuristic techniques that are known to be robust to tackle problems with different characteristics. Moreover, meta-heuristics are more efficient for large-scale problems than exhaustive techniques and often than deterministic algorithms, such as Branch and Bound [39].

Meta-heuristics do not guarantee finding the global optimal solution. However, ASIL Decomposition is intrinsically an iterative process, and therefore we aim at providing near-optimal solutions efficiently, whilst still offering advantages over manual approaches, thus contributing to a more competent development of dependable systems

As mentioned previously, a Tabu Search technique is employed in this work. The TS metaheuristic is an algorithm based on a single solution being changed iteratively through neighboring moves. General traces to the different TS implementations include the incorporation of a local search strategy and the use of memory mechanisms to avoid re-visiting past solutions [13].

Here we make use of a variant called Steepest Ascent Mildest Descent (SAMD) initially introduced in [40] and applied to the optimization of system reliability by Hansen and Lih [41]. SAMD is meant for maximization problems; costs imposed by ASILs, on the other hand, are to be minimized; in that sense we have adapted SAMD to a minimization version: the Steepest Descent Mildest Ascent (SDMA). In SDMA, the steepest descent direction is pursued until a minima is reached; subsequently the mildest ascent route is used. Reverse moves are forbidden for a pre-defined number of iterations *p* to avoid returning to the local optima.

**Finding the steepest descent:**

It can be assumed that every cost heuristic formulated for the ASIL decomposition problem is non-decreasing which means that a reduction of a failure's ASIL will always result in the decrease of the total system ASIL-dependant cost. Accordingly, the steepest descent direction is followed by decrementing the ASIL of the failure for which this results in the highest system cost reduction. When the steepest descent direction is calculated, FMs whose ASIL reduction means violating DSRs are not inspected.

Consider the following ASIL assignment solution at a given iteration *k*, and the respective deliberations on the cost impact of decrementing the ASILs of its FMs. In this example, it is considered that reducing the ASIL of FM5 to QM (0) results in not fulfilling a DSR, and therefore that move is not involved in the calculation of the steepest descent.

| | FM1 | FM2 | FM3 | FM4 | FM5 |
|---|---|---|---|---|---|
| $Iter^k$ | 2 | 4 | 3 | 2 | 1 |

**Figure 13 – ASIL assignment solution at iteration *k***

- FM1 and FM4 (ASIL B to ASIL A) – cost reduction: 100 – 10 = 90
- **FM2 (ASIL D to ASIL C) – cost reduction: 10000 – 1000 = 9000;**
- FM3 (ASIL C to ASIL B) – cost reduction: 1000 – 100 = 900;

Decrementing the ASIL of FM1 represents the highest decrease in cost; that action is therefore undertaken for iteration *k*+1.

| | FM1 | FM2 | FM3 | FM4 | FM5 |
|---|---|---|---|---|---|
| $Iter^{k+1}$ | 2 | 3 | 3 | 2 | 1 |

**Figure 14 – ASIL assignment solution at iteration *k* + 1**

**Finding the mildest ascent:**

The steepest descent direction if followed until every descent move represents violating DSRs; when that situation occurs, a minima has been identified. The mildest ascent route is then pursued by incrementing the ASIL of the failure which results in the lowest system cost growth. Assuming the following solution that represents a *minima* at iteration *j*:

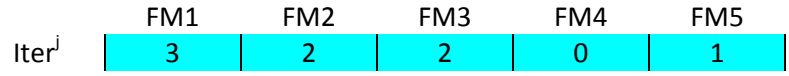



**Figure 15 - ASIL assignment solution at iteration *j*.**

The mildest ascent move is calculated as below:

- FM1 (ASIL C to ASIL D) – cost increase: 10000 – 1000 = 9000
- FM2 and FM3 (ASIL B to ASIL C) – cost increase: 1000 – 100 = 900;
- **FM4 (ASIL QM to ASIL A) – cost increase: 10 – 0 = 10;**
- FM5 (ASIL A to ASIL B) – cost increase: 100 – 10 = 90.

This results in the following solution at iteration *j*+1:

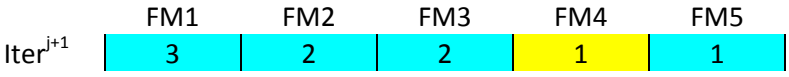



**Figure 16 - ASIL assignment solution at iteration *j* + 1**

**The memory mechanisms:**

A variable, $f_i$, stores the number of iterations to forbid decrementing the ASIL of failure *i* after an ascent move has been taken; this is to avoid returning to local optima. For iteration *j*+2, taking the steepest descent would mean returning to the *minima* of iteration *j* and it is therefore forbidden. The mildest ascent is calculated from the remaining considering the remaining FMs (see Figure 18).

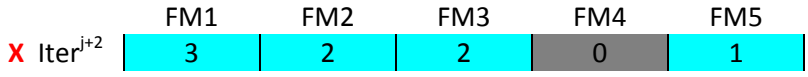



**Figure 17 – Forbidden ASIL assignment solution for iteration *j* + 2**


**Figure 18 - ASIL assignment solution at iteration *j* + 2**

Like Hansen and Lih, we also forbid the increase of a failure's ASIL after a descent move, for a number of iterations *p'*. Consider the following ASIL assignment solutions for two consecutive iterations.

| | FM1 | FM2 | FM3 | FM4 | FM5 |
|---|---|---|---|---|---|
| $Iter^{l}$ | 3 | 1 | 3 | 2 | 1 |
| $Iter^{l+1}$ | 3 | 0 | 3 | 2 | 1 |

**Figure 19 - ASIL assignment solutions at iterations *l*, *l* + 1**

Assuming that at iteration *l*+1, for which the ASIL of FM2 is decremented, a *minima* is identified, the calculation of the mildest ascent would indicate that FM2 should be incremented in iteration *l*+2 (least cost increase - the cost jump from ASIL QM (0) to A (1) is 10). That would mean returning to the same solution of iteration *l*, and therefore to further increase diversity in the search process such move is forbidden. As a result, the ASIL of FM5 is increased instead (least cost increase after the ASIL of FM2).

| | | | | | |
|---|---|---|---|---|---|
| X $Iter^{l+2}$ | 3 | 1 | 3 | 2 | 1 |

**Figure 20 – Forbidden ASIL assignment solution for iteration *l* + 2**

| | | | | | |
|---|---|---|---|---|---|
| ✓ $Iter^{l+2}$ | 3 | 1 | 3 | 2 | 2 |

**Figure 21 – Allowed ASIL assignment solution for iteration *l* + 2**

In regards to the memory parameters, *p'* was defined independently of p; moreover $f'_i$, in contrast to $f_i$, is not decreased in every iteration but only when further ASIL reductions are accomplished. In this way, ascent moves are forbidden for longer and diversification of the search direction is encouraged [41]. The values of *p* and *p'* are dynamically altered, as this reduces the sensitivity of the algorithm to the selection of such parameters. p is changed between 0 and the total number of FMs within a system, *n*, and p' between 0 and 0.4n; both reset their count after the upper bounds have been reached. Single increments are performed to *p* and *p'* at every 3rd and 4th iteration, respectively.

**Aspiration criteria:**

We have taken Hansen and Lih's approach a step further; our algorithm permits a tabu move overruling if this means obtaining a better solution than those found so far. Conditions to disregard tabu restrictions are known as aspiration criteria [42].

**How to start and finish:**

The search starts from an all ASIL D solution to ensure that the search starts from a point where no DSRs are violated. Diversity is introduced in the search by randomly selecting the FM whose is ASIL is changed, when multiple moves represent the same best cost variation.

The search is stopped after a pre-defined number of repetitions, *rep*, with no improvement to the best known solution.

**Algorithm Flowchart:**

The implemented algorithm is illustrated below with a Flowchart for better understanding. We formalize ASIL decomposition optimization as the search for the vector of ASILs *SRA*, corresponding to the *n* failure modes of a system, which minimizes the total ASIL-dependant cost, *C*, while respecting the all DSRs.

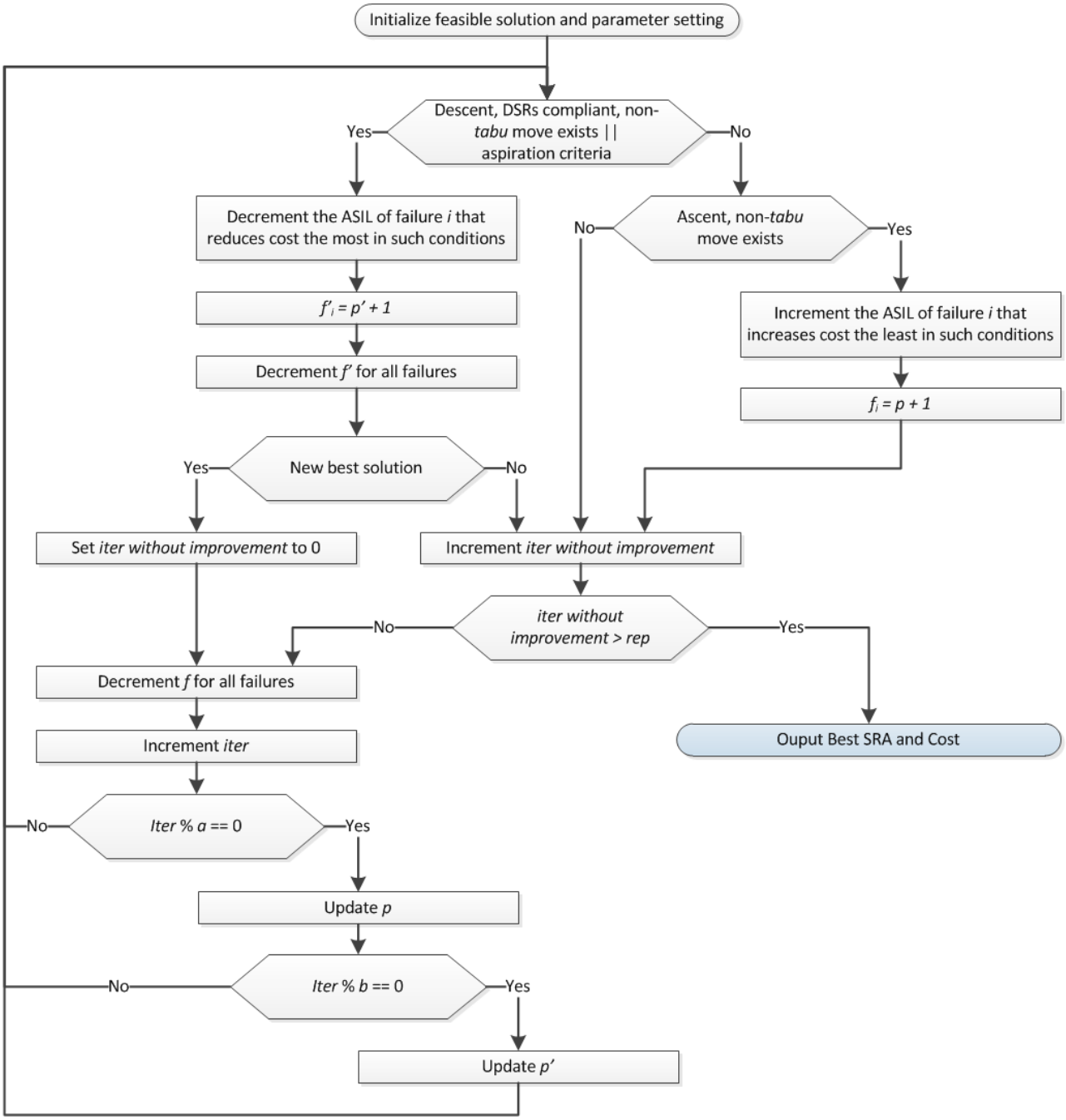



**Figure 22 – Tabu Search Algorithm Flowchart**

### 2.6.5 Implementation & Performance

The optimisation is implemented in the HiP-HOPS tool [44], building on its existing support for optimisation and the ASIL decomposition algorithm described earlier. As a result, performance between the standard exhaustive algorithm and the optimisation approaches can be directly compared. The central advantage of the optimisation approach is that it does not require HiP-HOPS to consider every possible solution, meaning that it performs much, much quicker; the trade-off is that it will not discover every solution in each case. The aim is simply to obtain a reasonable number of good, valid solutions.

For the small six component example described earlier, both the exhaustive algorithm and the optimisation algorithms are very quick (<1 second), and in the case of the optimisation, it also finds the valid solution. For larger examples, the exhaustive algorithm performs much worse, but will usually find all valid solutions eventually (unless the algorithm runs out of memory or is cancelled early). The TS scales far better but is not guaranteed to find optimal solutions. In one large-sized example, TS took 3 minutes to find the best known solution for that instance, whereas the exhaustive algorithm was stopped after 3 months without finishing the model. Continued testing and development is required to try to improve the success rate of the optimisation algorithms. Their advantage in performance is clear as optimisation is far more scalable than standard exhaustive approaches; however, to be more practical, they need tweaking further to yield more valid solutions. In addition, further developments of the standard exhaustive algorithm have resulted in improved performance, although it is doubtful that it will ever be fast enough to be applied to large systems.

# 3 Model-based system analysis techniques

This section presents an overview of model-based analysis techniques in the context of the MAENAD project and related analysis techniques.

## 3.1 The Scope of Analysis support by EAST-ADL

To clarify the exact analytical leverage of EAST-ADL for the development of embedded systems in FEV (Fully Electrical Vehicle), the WP2 of MAENAD is working on the elicitation of related language requirements.

In general, the overall landscape of architectural analysis can be reasoned according to the notions of views and viewpoints as in IEEE1471/ISO42010. This is shown in Figure 23. While a system description constitutes a system view based on one or multiple models, a viewpoint specifies the content of system description. An analysis support is always motivated by certain engineering concerns of stakeholders. This means that, for some particular analysis support, an analytical viewpoint stipulates the related analysis, verification, and validation concerns and the system artefacts in scope. The scope of analysis can range from requirements, to system design and properties, and to the design of verification and validation cases for an entire system or its subsystems. Moreover, according to the phases of system development, the scope of analysis can be related to certain levels of abstraction. To conduct analysis, models provide the descriptions of artefacts for formal or informal reasoning and thereby satisfy the related analytical viewpoints. For example, the analysis of safety constraints in Figure 23 denotes an analytical viewpoint motivated by the concerns about system failure modes, the levels of risks, and the related safety goals, functional and technical safety requirements, etc. The analysis relies on the views given by formal or semiformal descriptions of the assumed system operational situations, anomalies, and safe states.

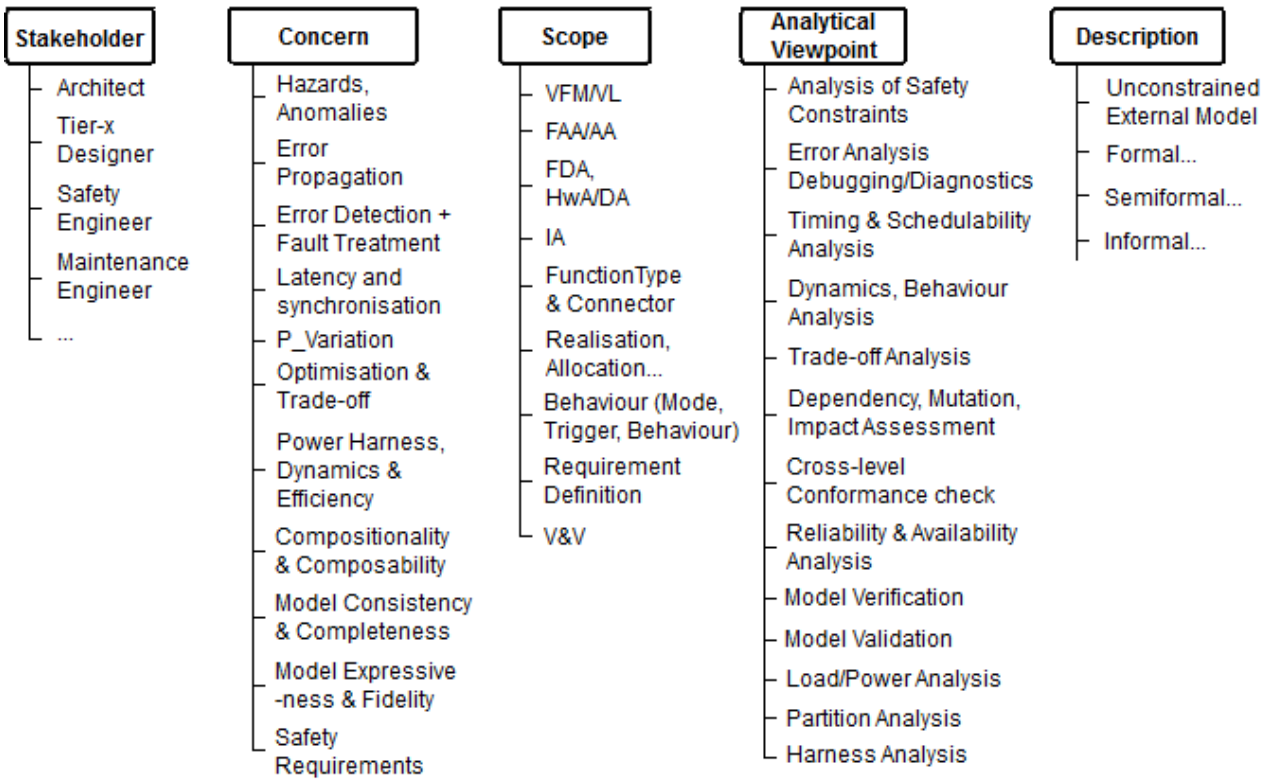



**Figure 23 – A structure of the landscape of potential analysis support by EAST-ADL.**

In particular, we consider that two categories of analysis can be directly benefited by EAST-ADL based system architecture description:

- **Class I – Analysis of the satisfactions of quality constraints:** the objective of this type of analysis is related to the constraint satisfaction problem (CSP), where the variables are given by the quality attributes annotated in EAST-ADL, such as timing budget and safety integrity level. An analysis in this category evaluates if the values assigned to these variables satisfy the constraints that are derived from the related system requirements, such as end-to-end timing, safety oriented partition-isolation. With EAST-ADL, the compositions of such quality constraints can be either declared separately or derived through the architecture specific topological relations (e.g. communication dependencies and resource allocation). In both cases, the mapping from analytical properties to architecture design is well structured through the language. Analyses belonging to this category include resource utilisation and cost estimation for the allocation of functions, timing budgeting, qualitative fault tree analysis, harness related calculations, change impact assessment, etc.
- **Class II – Analysis of behaviour-centric properties:** the objective of this type of analysis is related to the behaviour assessment, where the variables are given by the precise behaviour constraints annotated in EAST-ADL, such as parameter quantifications, states and state transitions. The analysis provides the support for requirements engineering, precise design specification, and the assessment of behavioural compositionality and composability. With EAST-ADL, the composition, refinement, and other relations of behavioural specifications are maintained in an architecture based way, i.e. based on the architecture specific topological relations. The analytical leverage is enabled by aligning the EAST-ADL native behaviour description with well-known behaviour formalisms. Analyses belonging to this category include dynamic behaviour analysis, cross-level behaviour conformance check (e.g., behaviour refinement), model validation in regard to use cases.

## 3.2 Enhanced language support for behaviour-centric analysis

Recent advances of EAST-ADL address the needs for a precise behaviour specification in order to enable a more precise specification of FEV system properties and to facilitate the transformations from EAST-ADL to external formalisms for the analytical leverage. We introduce the concepts of this EAST-ADL enhancement and current solutions in the following parts of this section.

### 3.2.1 The usages of EAST-ADL native behaviour constraint specification

Native support in EAST-ADL for precise behaviour description would give the following benefits:

1. The modelling support for behaviour constraint descriptions would allow the system developers to refine textual requirements by formalising the statements of use case and operational scenarios and thereby to elicit, validate, and derive related concerns on the basis of particular architectural design assumptions.
2. As a part of the overall language support for managing the traceability of requirement satisfactions, the descriptions of behaviour constraints can also be used to specify the required conformity of IP-protected black-box functions or components. This constitutes a basis for having a more precise reasoning about the compositional effect of system functions or components (i.e. the compositionality and composability).
3. By managing all behavioural specification in a common architectural context, EAST-ADL provides fundamental support for assessing the correctness and completeness of refinements of system artefacts across multiple levels of abstraction for final code generation.

4. By having all behaviour specifications in the same context of architecture design, the language also provides a powerful basis for the design of advanced mode logics, e.g. in regard to fault-tolerance and quality-of-services. For such advanced features, the ability to capture, predict and manage the system wide effects of modes is of critical importance. This means that one should be able to relate system modes control with system operational situations, application behaviours, the schemes of execution and resource deployment, etc.
5. When targeting error specifications, the behaviour descriptions can be used to refine the definitions of estimated failure modes by providing a precise specification of faulty conditions in value and time and capturing the transitions between nominal states and errors.

In modelling practice with EAST-ADL, a behaviour constraint has both type and prototype(s) based on a type-prototype pattern for composition specification. See Figure 24 for an example model. While a type definition establishes a template for a range of behavioural concerns that share some common declarations and semantics, a prototype definition stipulates the instantiation of a behaviour constraint type in a particular context given by another type. In other words, the prototypes constitute the compositional parts of the types. The parameters for the instantiations of behaviour constraint types include *Attribute* and *EventOccurrence.* During an instantiation, such parameters (which are declared *a priori* in the type definition) are bound to some corresponding parameters given in the context. Through such binding declarations, the prototypes of behaviour constraint types (i.e. their instantiations) are connected. When such behaviour instantiation bindings go across different system functions or components, there should be at least one corresponding structural communication connector through which such bindings can take place.

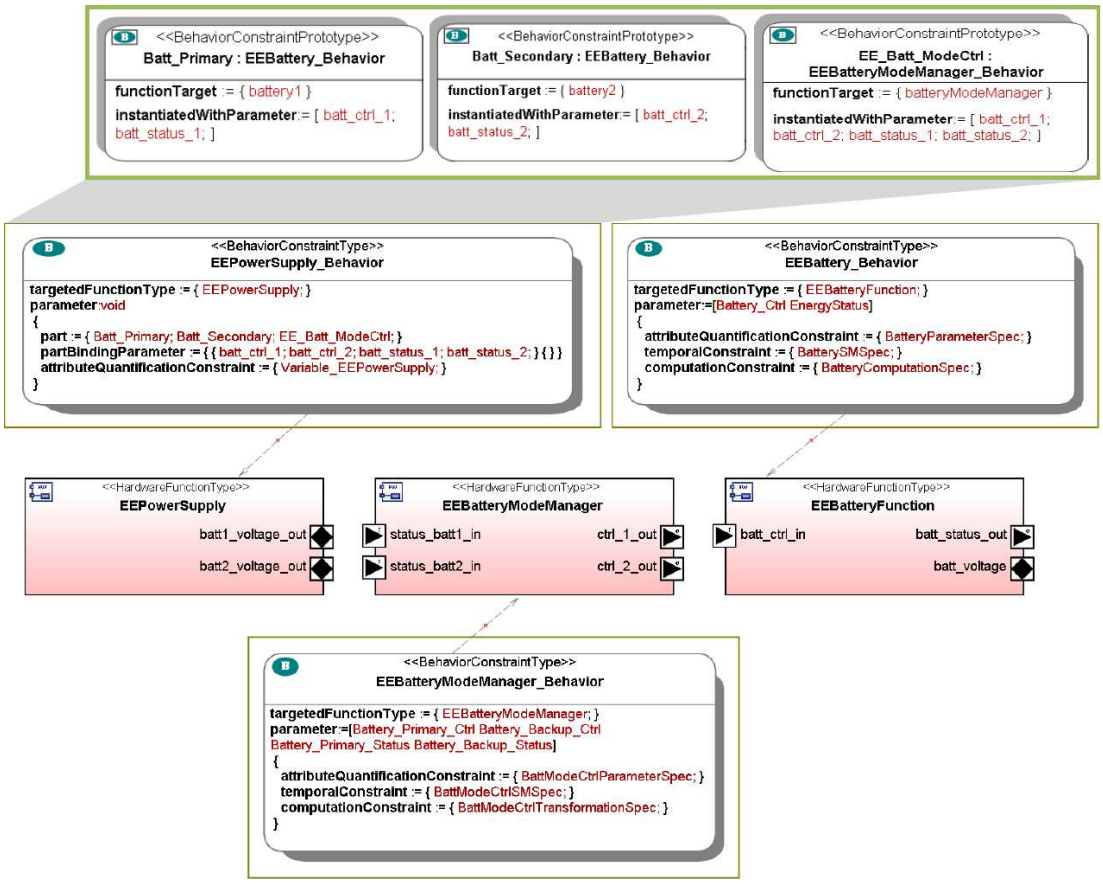



**Figure 24. An example specification of behaviour constraints for a power supply system. EEPowerSupply_Behavior and EEBattery_Behavior are the behaviour constraint types targeting the types of system components. Batt_Primary and Batt_Secondary are two**

**behaviour constraint prototypes instantiated with the same behaviour constraint type for the two replicated batteries.**

### 3.2.2 Key content of EAST-ADL native behaviour constraint specification

According to the fundamental needs of system design and analysis, an EAST-ADL behaviour constraint specification is subdivided into three categories (see Definition 1). The language design is introduced in D3.1.1. It is up to the users of EAST-ADL language, according to their particular design and analysis contexts, to decide the exact types and degree of constraints to be applied.

> **Definition 1 (Behavior constraint specification)** The EAST-ADL behavior constraint $\mathcal{BEHCONS}$ is an aggregation of three constraint specifications: $\mathcal{BEHCONS} = \mathcal{ATTQ} \cup \mathcal{TEMP} \cup \mathcal{COMP}$, where
>
> - $\mathcal{ATTQ}$ is a set of attribute quantification constraints, which are represented by the *AttributeQuantificationConstraint* class in the EAST-ADL meta-model. See Definition 5.
> - $\mathcal{TEMP}$ is a set of temporal constraints, which are represented by the *TemporalConstraint* class in the EAST-ADL meta-model. See Definition 9.
> - $\mathcal{COMP}$ is a set of computation constraints, which are represented by the *ComputationConstraint* class in the EAST-ADL meta-model. See Definition 13.

The attribute quantification constraints (given by the *AttributeQuantificationConstraint* class in the DM) are concerned with the value conditions of attributes underlying a behaviour on a timeline. They are useful for declaring the variables (e.g. the input-, output- and internal variables of a function), their expected values and logical relations. For example, an attribute quantification constraint specification can be used to annotate behaviour constraints in terms of continuous-time and discrete-time dynamics models.

The temporal constraints (given by the *TemporalConstraint* class in the DM) provide support for capturing the dependencies that a behaviour has in regard to its own history and other behaviours on a timeline. They are useful for defining discrete-event behaviours in requirements or design. All logical time conditions are also defined through temporal constraints. The language definition is aligned with timed-automaton.

The computation constraints (given by the *ComputationConstraint* class in the DM) provide support for specifying the restrictions on data processing, especially when the details of design are not available (e.g. for reasons of software component IP-protection). The descriptions can be related both to the expected logical transformations of data and to the expected cause-effect flows of events.

#### 3.2.2.1 Logical time condition - $\mathcal{LTC}$

For the descriptions of behaviour constraints, an abstract notion of time, referred to as the "logical time condition", is introduced. Declarations of such time conditions can be used to denote the time basis of continuous- and discrete-time dynamics or the timing concerns in state-machine or data-processing related behaviours.

> **Definition 2 (Logical time condition)** The EAST-ADL logical time condition $\mathcal{LTC}$ is the union of an infinite sequence of time intervals
>
> $$\mathcal{LTC} = \bigcup \{I_i = [t_i, t_i'] | 0 \leq t_i \leq t_i' \text{ and } t_i' \leq t_{i+1}, i \in \mathbb{N}\}$$

An illustrative example of $\mathcal{LTC}$ is that each time interval $I_i$ corresponds to the possible time range of an event occurrence. If $\forall i \in \mathbb{N}(t_i = t'_i)$, then $\mathcal{LTC}$ reduces to an ascending sequence of discrete time instances. The declarations of logical time conditions are supported by the meta-class (*LogicalTimeCondition*) and contained in the declarations of temporal constraints (*TemporalConstraint*).

The semantics of logical time conditions can be further refined by associating such conditions with the occurrences of execution events (*EventOccurrance*), such as due to the change of an environmental condition or the triggering of a function. This makes it possible to precisely define the reference points of a time interval (i.e. *startPointReference* and *endPointReference*). Two consecutive time conditions on the same time line can be declared using the *consecutiveTimeCondition* property. For example, two time intervals $t_k$ and $t_{k+1}$, shown in Figure 25, are consecutive if they represent the time durations $[t_k, t_{k+1}]$ and $[t_{k+1}, t_{k+2}]$ respectively.

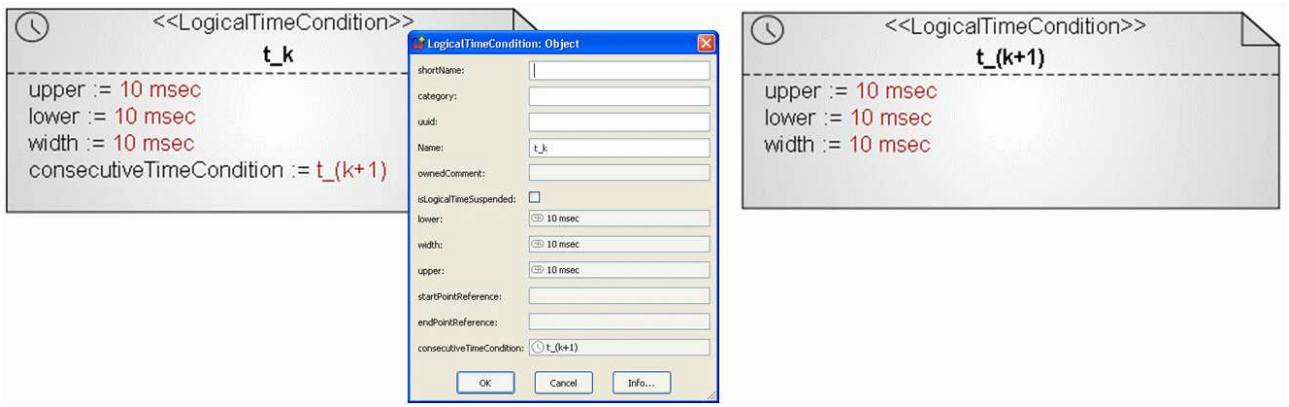



**Figure 25. An example of EAST-ADL specification of two consecutive time conditions.**

### 3.2.2.2 Attribute quantification constraints - $\mathcal{ATQ}$

Attribute quantification constraints (*AttributeQuantificationConstraint*) are concerned with the value conditions of attributes underlying a behaviour on a timeline. They are useful for declaring the variables (e.g. the input-, output- and internal variables of a function), their expected values and logical relations. An attribute quantification constraint can be expressed by either simple equations like $F = m^* a$, $V < 90$, or dynamics models. When necessary, the corresponding constraints on computational operations for data transformations and value assignment can be declared through the computation constraints (*ComputationConstraint*).

**Definition 3 (Attribute)** The EAST-ADL attribute set contains parameters or arguments of the behavior constraint specification $\mathcal{ATT} := \{x_1, \cdots, x_n\}$, where every element $x_i$ $(i = 1, \cdots, n)$ is a name declared in the EAST-ADL model and $n$ the total number of attributes. Each element is called an attribute. Let

$$\eta : \mathcal{ATT} \rightarrow \mathbb{R}$$

be an evaluation function of all attributes. The feasible range of attribute $x_i$ is denoted as set $X_i \subseteq \mathbb{R}$, i.e., $\forall \eta(\eta(x_i) \in X_i)$.

Here, we use the convention of representing an attribute by a lower-case letter and its scope by the corresponding capital letter. The descriptions of attributes are supported by the *Attribute* class in the EAST-ADL. An attribute can be a constant, simple, or complex data, given by the EAST-ADL data types (*EADataType*) for the related meta-information like unit, valid range, required accuracy, etc. If an attribute is externally visible (i.e. *isExternVisible* = *true*), it denotes an

input or output variable and has associated ports for the external accesses. Attributes are also used as instantiation parameters, *BehaviorInstantiationParameter*, to which certain values can be assigned when a behaviour constraint type is instantiated as behaviour constraint instances in certain modelling contexts.

**Definition 4 (Quantification)** The EAST-ADL quantification $\mathcal{QUANT}$ is the set of statements over the attributes about their value conditions or relations $\mathcal{QUANT} = \left(\mathcal{ATT}, \mathcal{LTC}, \mathcal{EXP}\right)$, where

- $\mathcal{ATT}$ denotes the set of attributes. By Definition 3, an attribute evaluation is equivalent to a vector in $\mathbb{R}^n$ and the set of all feasible attribute evaluations is $AE := \prod_{i=1}^{n} X_i$.
- $\mathcal{LTC}$ denotes the logical time condition as defined in Definition 2.
- $\mathcal{EXP}$ denotes the set of valid expressions of the timed attribute-vectors. A timed attribute-vector is a history of the attribute evaluations over time formalized as a function $h : \mathcal{LTC} \rightarrow AE$. Then the set of all feasible timed attribute-vectors is $AE^{\mathcal{LTC}}$. Any valid expression $exp \in \mathcal{EXP}$ converts a timed attribute-vector to other, i.e., a function $exp : AE^{\mathcal{LTC}} \rightarrow AE^{\mathcal{LTC}}$.

The descriptions of quantifications are supported by the *Quantification* class in the EAST-ADL meta-model. A quantification description can be decomposed into sub-quantification for complexity control. The specification can be used to annotate behaviour constraints in terms of continuous-time and discrete-time dynamics models. As mentioned in Definition 2, $\mathcal{LTC}$ can describe either continuous-time dynamics or discrete-time dynamics.

- Because continuous-time dynamics models are based on the rate of change on the state variables, if a state attribute $x \in \mathcal{ATT}$ exists in the model, then its derivative must also exist, i.e., $\dot{x} \in \mathcal{ATT}$.
- Discrete dynamics models specify behaviours by defining the state at next time instant given the current state and inputs. For any attribute $x \in \mathcal{ATT}$, the expression necessarily involves at least two different timed evaluation sequences, i.e., $x(t_i)$ and $x(t_{i+1})$ for any $i \in \mathbb{N}$.

**Definition 5 (Attribute quantification constraint specification)** The EAST-ADL attribute quantification constraint specification $\mathcal{ATTQ}$ is a pair $\mathcal{ATTQ} = \left(\mathcal{ATT}, \mathcal{QUANT}\right)$, where

- $\mathcal{ATT}$ is the set of attributes for the behavior being specified.
- $\mathcal{QUANT}$ is the set of quantification statements over the attributes.

For example, to specify a continuous-time dynamics behaviour, the EAST-ADL annotations will be applied as shown in Table 6. In the example, the attribute $x$ represents the system state, the attribute $u$ the input, and $v$ the state disturbance. See also Fig. 8 for a modelling example of defining a discrete-time dynamics behaviour based on the declarations of attributes and quantifications with explicit time conditions in terms of $t_k$ and $t_{k+1}$.

**Table 6. EAST-ADL annotations for an example dynamics model $\dot{x}(t) = f(x(t), u(t), v(t))$.**

| Annotation | Definition |
|---|---|
| $\mathcal{LTC}$ | $\mathcal{LTC} = \{t \mid t_i \leq t \leq t_i', t \in \mathbb{R}, t_i \in \mathbb{R}, t_i' \in \mathbb{R}, i \in \mathbb{N}\}$ |
| $\mathcal{ATT}_D$ | $\mathcal{ATT}_D = \{att \mid att \in \mathbb{R}^n, n = 1\} = \{x, u, v\}, \lvert\mathcal{ATT}_D\rvert = 3$ |
| $\mathcal{EXP}_D$ | With $\mathcal{U}^{\mathcal{LTC}} = \{u(t)\}$, $\mathcal{V}^{\mathcal{LTC}} = \{v(t)\}$, $\mathcal{X}^{\mathcal{LTC}} = \{x(t)\}$, and $\mathcal{DX}^{\mathcal{LTC}} = \{\dot{x}(t)\}$, we have $\mathcal{EXP}_D = \left\{exp_D : \{x(t), u(t), v(t)\} \rightarrow \{\dot{x}(t)\}\right\}$, $\lvert\mathcal{EXP}_D\rvert = 1$ |

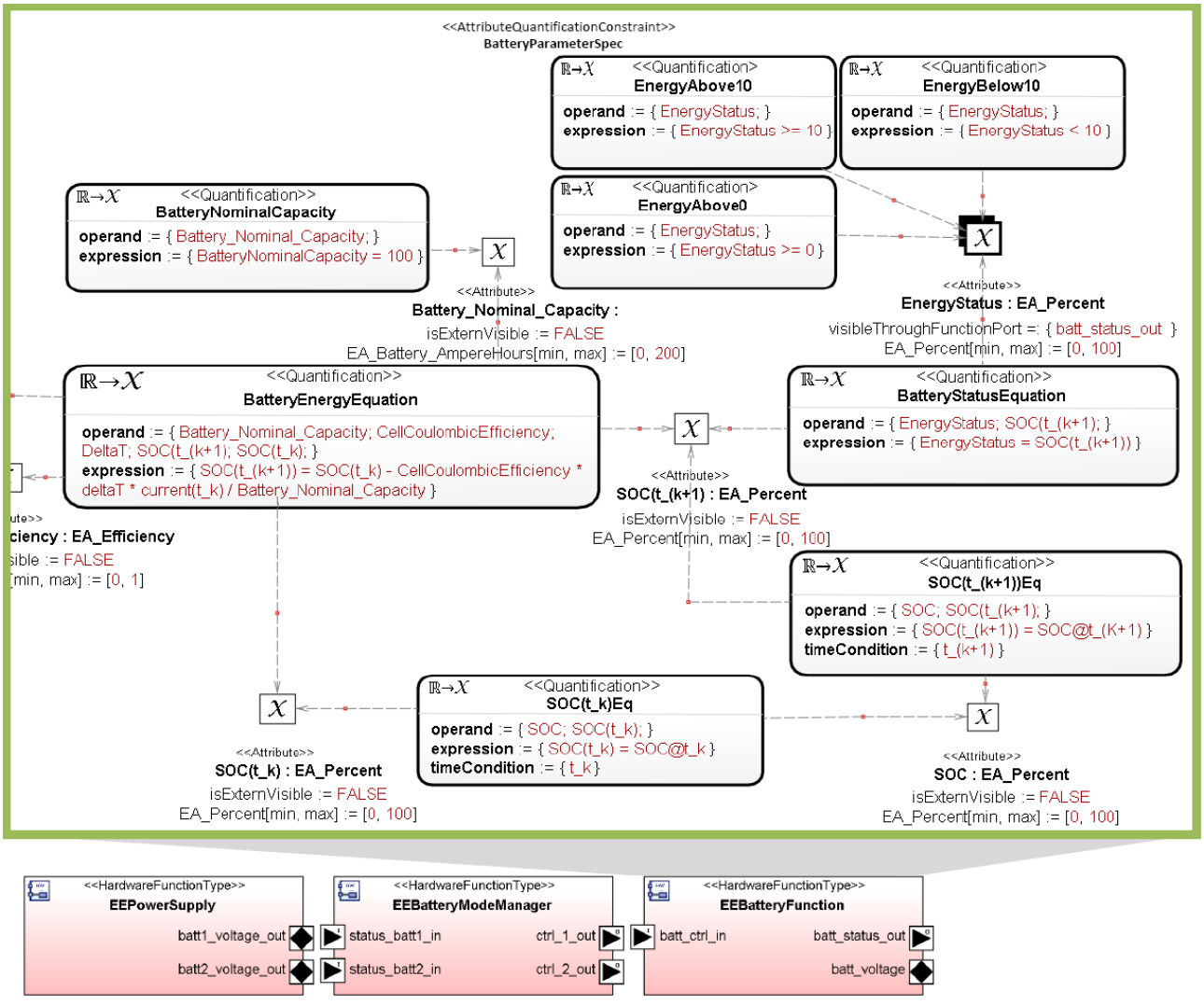



**Figure 26. The EAST-ADL specification of attribute quantification constraints for a battery function.**

### 3.2.2.3 Temporal constraints - $\mathcal{Temp}$

Temporal constraints (*TemporalConstraint*) constitute the language support for capturing the dependency that a behaviour has in regard to its own history and other behaviours on a timeline.

They are useful for defining the discrete-behaviours in requirements or design solutions. All logical time conditions are also defined through temporal constraints.

A system or component has a finite set of discrete states $\mathcal{STATE} = \{s_0, \ldots, s_k\},\ k \in \mathbb{N}$. Each state defines a situation where certain value- and time-conditions must hold.

**Definition 6 (State Invariant)** The state invariants at every state is given by the function

$$\mathcal{SINV} : \mathcal{STATE} \rightarrow \mathcal{SQI} \times \mathcal{STI}$$

where

- $\mathcal{SINV}$ denotes the state invariants,
- $\mathcal{SQI}$ refers to the value invariants of states, i.e. the value conditions that must hold in individual states, $\mathcal{SQI} \subseteq 2^{AE}$. Recall that $AE$ is the set of attribute evaluations defined in Definition 4.
- $\mathcal{STI}$ refers to the time invariants of states, i.e. the time conditions that must hold in individual states, $\mathcal{STI} \subseteq 2^{\mathcal{LTC}}$.

The descriptions of discrete states are supported by the *State* class in the EAST-ADL meta-model with the state invariants given by the associations to *LogicaTimeCondition* and *Quantification*. A state $s$ is an initial state $s_0$ when *isInitState* = *true*. In the context of system design, a state $s$ can represent one or multiple operation modes when *isMode=true*; or one or multiple errors in the system when *isErrorState=true*, or hazards when *isHazard=true*.

**Definition 7 (Event occurrence)** An event occurrence denotes the actual happening of certain logical, execution, and erroneous conditions, i.e.,

$$\mathcal{EVTOC} = \mathcal{EVT}_{\mathcal{L}} \dot{\cup} \mathcal{EVT}_{\mathcal{E}} \dot{\cup} \mathcal{EVT}_{\mathcal{F}}$$

where

- $\mathcal{EVT}_{\mathcal{L}}$ denotes the logical events triggered by the change of predicates expressed by the attributes and time.
- $\mathcal{EVT}_{\mathcal{E}}$ denotes the execution events with $\mathcal{EVT}_{\mathcal{E}} = \mathcal{EVT}_{\mathcal{E}\_\mathcal{F}} \dot{\cup} \mathcal{EVT}_{\mathcal{E}\_\mathcal{P}}$. That is, the execution events are given by the function events $\mathcal{EVT}_{\mathcal{E}\_\mathcal{F}}$ or port events $\mathcal{EVT}_{\mathcal{E}\_\mathcal{P}}$. Each execution event specifies a distinct form of state change in a running system, taking place at distinct points in time. While a function event refers to the triggering of a function block, a port event refers to the data sending or receiving at a port of function block.
- $\mathcal{EVT}_{\mathcal{F}}$ denotes the erroneous events with

$$\mathcal{EVT}_{\mathcal{F}} = \mathcal{EVT}_{\mathcal{F}\_\mathcal{F}} \dot{\cup} \mathcal{EVT}_{\mathcal{F}\_\mathcal{H}} \dot{\cup} \mathcal{EVT}_{\mathcal{F}\_\mathcal{A}}$$

That is, an erroneous event belongs to exactly one of three disjoint event sets, namely feature flaw events $\mathcal{EVT}_{\mathcal{F}\_\mathcal{F}}$, system hazards $\mathcal{EVT}_{\mathcal{F}\_\mathcal{H}}$, and system anomalies (faults and failures) $\mathcal{EVT}_{\mathcal{F}\_\mathcal{A}}$.

The descriptions of discrete event occurrences EVT OC are supported by the *EventOccurrence* class in EAST-ADL.

**Definition 8 (Discrete transition)** Discrete transitions describe the possible switches between discrete states due to the occurrences of discrete events or due to the violations of a state invariant in time or in value quantification

$$\mathcal{TRANS} \subseteq \mathcal{STATE} \times \mathcal{RWEVTOC} \times \mathcal{QG} \times \mathcal{TG} \times \mathcal{EFF} \times \mathcal{STATE}$$

where

- $\mathcal{STATE}$ denotes the *from* and *to* discrete states,
- $\mathcal{QG}$ denotes the possible value guards, $\mathcal{QG} \subseteq 2^{AE}$
- $\mathcal{TG}$ denotes the possible time guards,$\mathcal{TG} \subseteq 2^{\mathcal{LTC}}$
- $\mathcal{RWEVTOC}$ denotes the event occurrence to read or write, $\mathcal{RWEVTOC} \subseteq \mathcal{EVT}_{\mathcal{E}}$.
- $\mathcal{EFF}$ denotes the effects of the occurrence of the logical transformations associated to the transition, $\mathcal{EFF} \subseteq 2^{\mathcal{LFNOC}}$, (see Definition 11 for the definition of $\mathcal{LFNOC}$).

The descriptions of transitions are supported by the meta-class *Transition* in EAST-ADL. The *from* and *to* roles can be applied to two distinct states or a single state. When all the given guard conditions are met, a transition will be fired to respond to the occurrence of an event (which is indicated by the role *readEventOccurrences?*) or to signal the occurrence of an event (which is indicated by the role *writeEventOccurrance!*). A transition, when fired, will lead to entering the associated to state, while invoking one or more logical transformations (*TransformationOccurrance!*) as the effects of the transition.

**Definition 9 (Temporal constraint specification)** The EAST-ADL temporal constraint $\mathcal{TEMP}$ is a tuple

$$\mathcal{TEMP} = \left(\mathcal{STATE}, \mathcal{EVTOC}, \mathcal{TRANS}, s_0, \mathcal{LTC}\right)$$

where

- $\mathcal{STATE}$ denotes the states of a discrete behavior to be satisfied by a system or its environment, $\mathcal{STATE} = \{s_0, \ldots, s_k\}, k \in \mathbb{N}$. Each state is associated to an invariant function as defined by Definition 6.
- $\mathcal{EVTOC}$ denotes the discrete events whose occurrences are taken into consideration, as in Definition 7.
- $\mathcal{TRANS}$ denotes the discrete transitions of the discrete behavior, as in Definition 8.
- $s_0 \in \mathcal{STATE}$ is the initial state.
- $\mathcal{LTC}$ denotes a set of time intervals that constitute the logical time basis.

The descriptions of temporal constraint are supported by the *TemporalConstraint* class in the EAST-ADL language. See Fig. 15 for a modelling example.

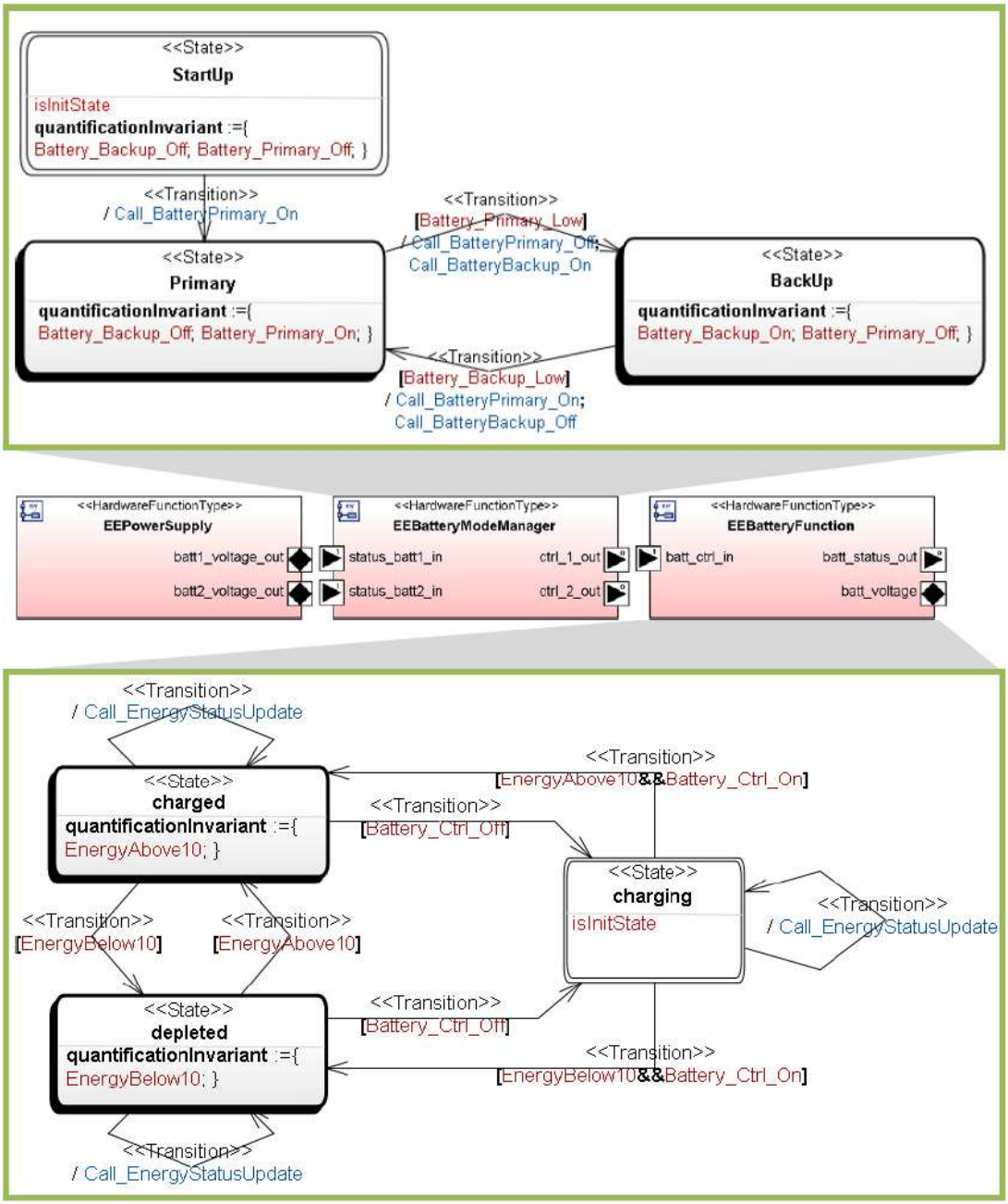



**Figure 27. The EAST-ADL specification of temporal constraints for a battery controller and battery functions. The specification is based on the declarations of two state machines with both textual and graphical views.**

### 3.2.2.4 Computation constraints - $COMP$

Computation constraints in EAST-ADL (*ComputationConstraint*) provide the language support for specifying the restrictions on data processing, especially when the details of design are not available (e.g. for the reasons of software component IP-protection). The descriptions can be related both to the expected logical transformations of data and to the expected cause-effect paths of events.

**Definition 10 (Logical transformation)** The EAST-ADL logical transformation $\mathcal{LFN}$ is a set of conditional executions of function components to compute the output attributes.

$$\mathcal{LFN} \subseteq \mathcal{PRE} \times \mathcal{EXP} \times \mathcal{PST} \times \mathcal{TINV}$$

where

- $\mathcal{PRE}$ and $\mathcal{PST}$ are the sets of preconditions and postconditions, respectively, over attributes and logical time, namely $\mathcal{PRE}, \mathcal{PST} \subseteq 2^{AE \times \mathcal{LTC}}$;
- $\mathcal{EXP}$ is the set of valid expressions of the timed attribute vectors, as defined in Definition 4;
- $\mathcal{TINV}$ is the set of time invariants, namely $\mathcal{TINV} \subseteq 2^{\mathcal{LTC}}$, to specify the time constraints of the function execution, such as the worst-case execution time and deadline.

Given some in- and local-data that meet certain preconditions, a logical transformation always maps such data to some out-data that meet the related post-conditions if the time- and value-invariants are not violated during the data processing. In system development, the specification of a logical transformation captures one particular contract that the implementations of a system function by software and hardware means have to satisfy. Such a specification can be used for many purposes, such as for component design and integration, test case generation, run-time monitoring and quality-of-service adaptation, diagnostics, etc. The EAST-ADL meta-class for the declarations of logical transformations is *LogicalTransformation*.

**Definition 11 (Occurrence of Logical Transformation)**
Given a logical transformation $lfn = (pre, exp, pst, tinv) \in \mathcal{LFN}$, its occurrence is characterized by a tuple $oc\_lfn := (execution, time)$, where

- $execution$ is a function

$$execution : AE \to \{0,1\} \times AE : a \mapsto \begin{cases} (0, a), & a \notin pre \\ (0, exp(a)), & a \in pre \wedge exp(a) \notin pst \\ (1, exp(a)), & a \in pre \wedge exp(a) \in pst \end{cases}$$

- $time \in \mathcal{LTC}$ denotes the actual time duration for the occurrence of the logical transformation.

Let $\mathcal{LFNOC} \subseteq \{oc\_lfn | lfn \in \mathcal{LFN}\} \times \mathcal{LTC}$ be the set of all possible occurrences of all logical transformations.

The EAST-ADL metaclass for the occurrences of logical transformations is the *LogicalTransformationOccurrence*. Activated by state transitions or logical paths, each occurrence of logical transformation can have particular values of in-data (*inQuantification*) and out-data (*outQuantification*) due to the particular invocation contexts.

**Definition 12 (Logical path)**
An logical path $lph$ in EAST-ADL is defined recursively as follows.

- $p_0 := (e_s, lfn\_oc, e_r)$ is a logical path if $e_s, e_r \in \mathcal{EVT}_{\mathcal{L}}$ and $lfn\_oc$ is a transformation occurrence as defined in Definition 11. The semantics is that the occurrence of the logical event $e_s$ (stimulus) results in the occurrence of the logical event $e_r$ (response) through the occurrence $lfn\_oc$ of a logical transformation. Denote $s(p_0) := e_s$ and $r(p_0) := e_r$ to retrieve the stimulus and response events of the event path $p_0$.
- Let $p_1, p_2$ be two logical paths.
  1. If $r(p_1) = s(p_2)$ (or $r(p_2) = s(p_1)$), then the serial combination of adjoining $r(p_1)$ and $s(p_2)$ (or $r(p_2)$ and $s(p_1)$) is a logical path $p_3 := p_1 + p_2$ (or $p_2 + p_1$). Moreover $s(p_3) = s(p_1)$ (or $s(p_2)$) and $r(p_3) = r(p_2)$ (or $r(p_1)$).
  2. If $s(p_1) = s(p_2)$ and $r(p_1) = r(p_2)$, then the parallel combination of adjoining $s(p_1)$ and $s(p_2)$ and adjoining $r(p_1)$ and $r(p_2)$ is a logical path $p_3 := p_1 \| p_2$. Moreover $s(p_3) = s(p_1)$ and $r(p_3) = r(p_1)$.

The EAST-ADL logical path, based on the meta-class *LogicalPath*, provides the modelling support for annotating the expected cause-effect traces across a system or a component.

**Definition 13 (Computation constraint specification)** The EAST-ADL computation constraint $\mathcal{COMP}$ is a pair $\mathcal{COMP} = \left(\mathcal{LFN}, \mathcal{LPH}\right)$, with

- $\mathcal{LFN}$ denotes the restrictions on the logical transformations of data, specified by Definition 10.
- $\mathcal{LPH}$ denotes the restrictions on the cause-effect paths of events, specified by Definitions 12.

The EAST-ADL meta-class for computation constraints is the *ComputationConstraint*. See Figure 28 for a modelling example.

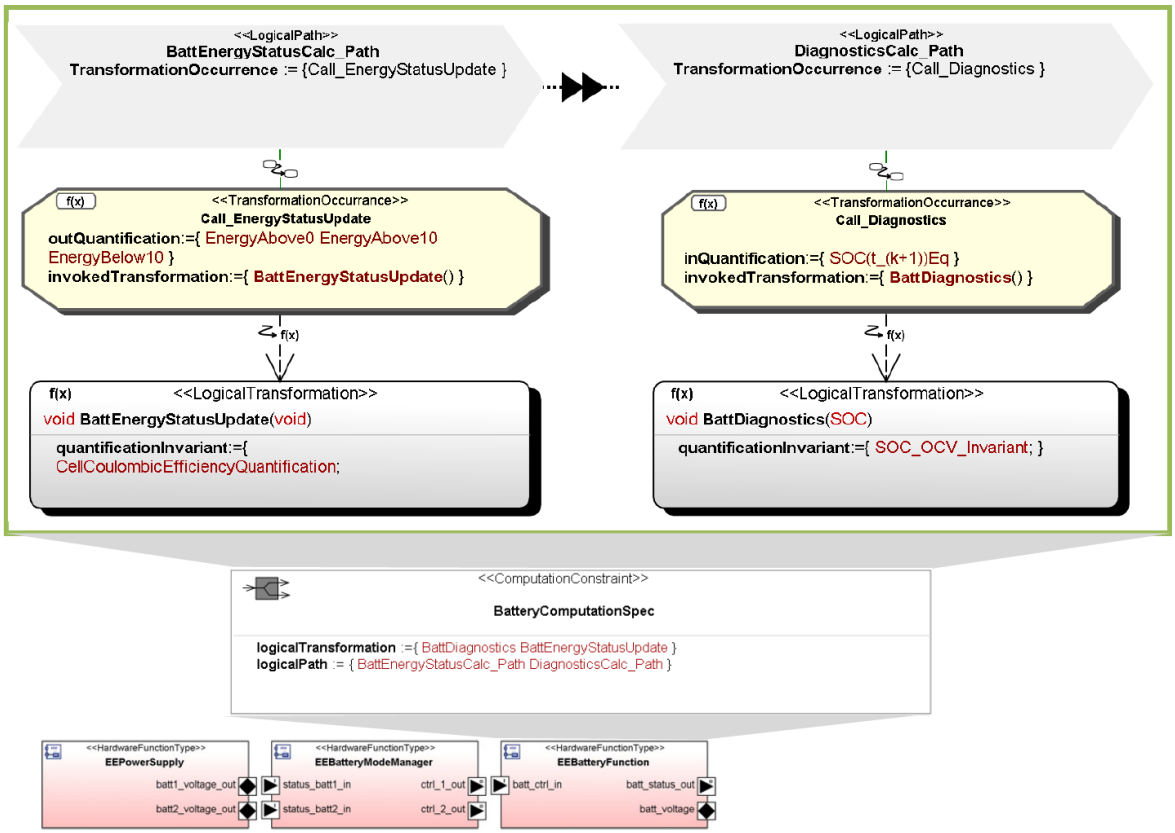



**Figure 28. The EAST-ADL specification of computation constraints for the battery function. The specification defines two ordered logical paths and two transformations that must be invoked according to the precedence given by the path definition.**

#### 3.2.2.5 Execution Semantics

In modelling practice with EAST-ADL, a behaviour constraint ($\mathcal{BEHCONS}$) is always instantiated with prototypes. The execution of each atomic prototype has its own thread of control without any internal concurrency, also following the run-to-completion semantics: when triggered, it reads all input ports, executes the computation, and then writes the output ports. If new data arrives at the input ports during the execution or writing phase, it cannot be processed in the current executing cycle. Moreover, each incoming port represents a message buffer with specific semantics.

**Definition 14 (Semantics of Message Buffer)** The message buffer in EAST-ADL has length one, does not block the sender when it is full or the receiver when it is empty, allows the stored message to be replaced by new messages, and always retains the message when it is read.

### 3.2.3 Analysis of EAST-ADL native behaviour constraint specification through to external tools

The EAST-ADL support for native behaviour constraint description is fundamentally a hybrid-system model as defined in [28]. The approach emphasises the provision of a practical modelling language for precisely annotating various behavioural concerns in system development as well as for managing the descriptions of such concerns together with the specifications of requirements,

system design and constraints, and verification and validation cases on a common information basis. To ensure the analytical leverage, the support for model transformations from EAST-ADL to some external techniques has currently been investigated. The external techniques currently under investigation include SPIN, UPPAAL, Matlab/Simulink, and Modelica. Here, we introduce the key concepts of the SPIN transformation below.

#### 3.2.3.1 Support for SPIN Transformation

For the analysis based on the behaviour constraint specification, one interesting tool is the SPIN model checker, which is a powerful open-source software tool for detecting defects in distributed and concurrent system designs [29]. It is a logic model checker that efficiently characterises all possible executions generated by the design model, rather than a small subset reachable by the traditional methods based on human inspection, testing, or random simulation. SPIN can therefore exhaustively verify the design model against logical requirements, including assertion, freedom of deadlock, reachability of the desired state, avoidance of or compliance with given execution patterns, and other complex user-defined correctness criteria.

SPIN model is specified by PROMELA (Process Meta-Language), which emphasises the modelling of process synchronisation and coordination as opposed to computation. The basic building blocks of SPIN models are asynchronous processes, buffered and rendezvous message channels, synchronising statements, and structured data. Inside SPIN, the asynchronous processes and their interactions are represented as finite state automata. SPIN offers several ways for specifying the correctness requirements, including basic assertions, invalid end-states, linear temporal logic (LTL) constraints, etc.

As the EAST-ADL behaviour constraint description has wider scope than the native SPIN, some modelling limitations are necessary for the transformation. For example, SPIN does not naturally support the counting of time, nor does it support complex calculation or floating data type. In our investigation, the following restrictions on the behaviour constraint specification are necessary:

- The specifications of logical time conditions ($\mathcal{LTC}$) depending on precise time values should be avoided or replaced with qualitative specifications in temporal logic.
- The specifications of EAST-ADL attributes and quantifications based on floating-point data type and expressions like $\sin(x)$ or $\sqrt{x}$ should be avoided for the time being, as the direct support of SPIN covers only basic arithmetic calculations.

By the EAST-ADL definition, a behaviour constraint prototype consists of a set of attributes, a set of computation constraints, and a temporal constraint formalised as an automaton. Table 7 summarises the mapping rules from an EAST-ADL behaviour constraint prototype to a SPIN model. In essence, attributes in the behaviour model are mapped to data objects in the SPIN model, computations mapped to PROMELA expressions and assignments, the temporal constraint mapped to a process type in SPIN, and the read/write event occurrence mapped to message receiving/sending statements in SPIN.

To enable computer-aided modelling, formal analysis and even automatic code generation, the expression in EAST-ADL model follows the syntax and semantics of a subset of the C programming language, namely the MISRA-C. The syntactic resemblance between C and PROMELA allows most C expressions used by EAST-ADL to be directly reused in the SPIN model. Some exceptions and the corresponding mapping rules are listed in Table 8. Row 2 is not a complete list of advanced C assignment operators. Others like *=, /= can be converted to PROMELA statements in the same way.

**Table 7. The mapping of related concepts in EAST-ADL and PROMELA**

| EAST-ADL Concept | EAST-ADL Meta-Class | PROMELA | Comment |
|---|---|---|---|
| *BEHCONS* | *BehaviorConstraint* | *Proctype*, which is the key word to declare a process definition | The abstract definition of a process describes the execution of an atomic SW component |
| *ATTQ* | *AttributeQuantification Constraint* | PROMELA expression | The expression must comply with PROMELA language grammar |
| *ATT* | *Attribute* | PROMELA data, e.g., Boolean, byte, integer, communication channel | Floating point datatype is not naturally supported by SPIN. The communication channel is employed for asynchronous communication between processes |
| *TEMP* | *TemporalConstraint* | PROMELA statements within the process | Temoral constraints such as sequence, precedence, synchronization, and consequence are specified within the process definition |
| *COMP* | ComputationConstraint | PROMELA expression | Promela supports only basic arithmetic calculations |
| *QUANT* | *Quantification* | PROMELA expression | |
| *LTC* | | Positive integer | As a logic model checker, SPIN does not directly support time, but the clock ticks can be counted by an integer |
| *RWEVTOC* | *EventOccurrence* | Read or write the message channel | The operation is nonblocking |
| *STATE* | *State* | The separator ";" or "->" between two statements | |
| *TRANS* | *Transition* | The statement | In PROMELA, each statement corresponds to one transition |

**Table 8. Basic mapping between C and PROMELA**

| C | PROMELA |
|---|---|
| `++var`<br>`-var` | `var++`<br>`var-` |
| `var += expr`<br>`var -= expr` | `var = var + expr`<br>`var = var - expr` |
| `expr1 ? expr2 : expr3` | `(expr1 -> expr2 : expr3)` |
| `if(cond1)`<br>`  statement1;`<br>`else if(cond2)`<br>`  statement2;`<br>`else`<br>`  statement3;` | `if`<br>`:: cond1 -> statement1`<br>`:: cond2 -> statement2`<br>`:: else -> statement3`<br>`fi` |
| `while(cond)`<br>`{`<br>`  statement;`<br>`}` | `do`<br>`:: cond ->`<br>`  statement`<br>`:: else -> break`<br>`od` |

Note a subtle difference in selection between the two languages. There are two subtle differences between EAST-ADL semantics and PROMELA semantics. First, EAST-ADL is deterministic. At a state where multiple exit transitions are defined, EAST-ADL must deterministically choose one

when more than one transitions are executable, whereas PROMELA non-deterministically chooses a valid branch. Second, EAST-ADL is non-blocking. By the run-to-completion semantics, if no transition is executable at a state, the EAST-ADL behaviour model will terminate the current running instance, whereas the SPIN model will be blocked.

To reconcile the conflict, we enforce that (1) the guards of all outgoing transitions of a state must be mutually exclusive and (2) in case where no guard is valid, the default transition to the final state is executable.

For automatic PROMELA conversion, we formalize a behaviour constraint type as a 7-tuple:

$\mathcal{BEHCONS} = (\mathcal{STATE}, s_0, s_f, \mathcal{TRANS}, \mathcal{ATT}, \mathcal{ATT}_0, \mathcal{RWEVTOC})$, where

- $\mathcal{STATE}$ is the finite set of discrete states as defined by Definition 6, and $s_0, s_f \in \mathcal{STATE}$ are the initial and the final states, respectively. Each state $s \in \mathcal{STATE}$ has an invariant condition defined by the function $\mathcal{INV}(s) = (QI, TI)$, where $QI \subseteq \mathcal{QUANT}, TI \subseteq \mathcal{LTC}$. As usual, the two sets can be implicitly represented by two predicates $\mathbf{P}_{QI}, \mathbf{P}_{TI}$. The invariant at state $s$ is hence $\mathbf{P}_{QI} \,\&\&\, \mathbf{P}_{TI}$. If $\mathcal{INV}(s) = true$, then the invariant condition at this state is omitted from the model.
- $\mathcal{TRANS}$ is the finite set of discrete transitions as defined by Definition 8. For convenience we define the functions $guard : \mathcal{TRANS} \to \mathcal{QG} \times \mathcal{TG}$, $effect : \mathcal{TRANS} \to \mathcal{EFF}$, $to : \mathcal{TRANS} \to \mathcal{STATE}$ to retrieve respectively the guard condition, the effect function, and the target state of the transition. Similar to the state invariant condition, a guard condition is also represented as the conjunction of a state invariant and a time invariant.
- $\mathcal{ATT}$ as defined by Definition 3 is the set of data objects used in the behavior constraint prototype and $\mathcal{ATT}_0 \subseteq \mathcal{ATT}$ the subset of the externally visible attributes so as to initialize the prototype.
- $\mathcal{RWEVTOC}$ is the set of event occurrences reading or writing the message buffers. Because EAST-ADL prescribes precise semantics of the message buffer, as in Definition 14, and the semantics is different from the message channel in SPIN, we have written dedicated PROMELA scripts to realize the semantics.

For automatic PROMELA conversion, we also order the states in $\mathcal{STATE}$ so that the initial state $s_0$ has the smallest index and the final state $s_f$ the largest. Define the index function:

$$index : \mathcal{STATE} \to \{0, 1, \cdots, n\},$$

where $n = \|\mathcal{STATE}\| - 1$, $index(s_0) = 0$, and $index(s_f) = n$. Here $\|S\|$ denotes the cardinality of the set $S$. We also define the function

$$exit : \mathcal{STATE} \to 2^{\mathcal{TRANS}} : s \mapsto \{(s, e, s') \in \mathcal{TRANS}\}$$

to determine all exit transitions at state $s$.

The execution trigger of a behaviour constraint prototype is modelled as a synchronisation channel $trigger \notin \mathcal{ATT} \cup \mathcal{RWEVTOC}$

between the process and a dispatcher process. To dispatch a process, the dispatcher simply sends a triggering signal to the channel $trigger$.

Finally we sketch the algorithm to convert the behaviour constraint prototype $\mathcal{BEHCONS}$ to a SPIN process process_P. In Algorithm 1, the notation "[x]" means that the whole string should be replaced by the actual value of x in the text context, where x is an expression that may return an integer, a text string, or a set of data objects. For instance, if x = 1, then "N_[x]" represents "N_1". The notation "\n" switches the text to a new line.

**Algorithm 1:** Model Transformation

**Input**: A behavior constraint prototype
$\mathcal{BEHCONS} = (\mathcal{STATE}, s_0, s_f, \mathcal{TRANS}, \mathcal{ATT}, \mathcal{ATT}_0, \mathcal{RWEVTOC})$
**Output**: A SPIN process `process_P`

1 Let $n := ||\mathcal{STATE}|| - 1$;
2 Write the process declaration:
"`proctype process_P(chan [`$\mathcal{RWEVTOC}$`]; [`$\mathcal{ATT}_0$`])\n {\n`";
3 Declare local data objects: "`[`$\mathcal{ATT} - \mathcal{ATT}_0$`];\n`";
4 Specify the trigger: "`end: do\n ::` $trigger_P$ `? 1->\n`";
5 **for** $k$=*1* **to** $n$ **do** `/* Do nothing for` $s_0$ `*/`
6 Let $a \in \mathcal{STATE}$ with $index(a) = k$;
7 **if** $\|exit(a)\| > 1$ **or** $guard(tran) \neq$ "" **then** `/*` $a$ `has multiple exit transitions or the guard of the transition is not empty */`
8 Start the selection: "`S_[`$k$`]: if\n`";
9 **foreach** $tran \in exit(a)$ **do**
10 Write the line: "`::` $[guard(tran)]$ `->\n`";
11 Write the line: "$[effect(tran)]$`;\n goto S_[`$to(tran)$`]\n`";
12 **end**
13 Write the line: "::else -> goto S_[$n$]";
14 Terminate the selection: "`fi;\n`";
15 **else if** $a == s_f$ **then** `/*` $a$ `is the final state */`
16 Do nothing: "`S_[`$n$`]: skip;\n`";
17 Complete and rewind the process: "`od\n`";
18 **else** `/*` $a$ `has one exit transition whose guard is always true */`
19 Commit the action: "`S_[`$k$`] :` $[effect(tran)]$ `;\n`";
20 Let $x = to(tran)$;
21 **if** $index(x) \neq k + 1$ **then**
22 Write the line: "`goto S_[`$index(x)$`];\n`";
23 **end**
24 **end**
25 **end**
26 Terminate the process declaration: "`}\n`";

## 3.3 Modelling support for dependability analysis

### 3.3.1 Dependability and error modelling in EAST-ADL

As part of its support for dependability engineering, EAST-ADL allows the modelling of system error behaviours for safety analysis (e.g. FTA) and allows safety constraints to reference the faults/failures under consideration. Figure 29 shows the related key language concepts, which are explained below.

An *ErrorModel* is a constrained model providing the analytical information about the error logic of a target component/system, i.e., which faults may occur and how they propagate. Associations to the nominal architecture identify the target components/systems and the modelling contexts. While the fault ports (*FaultInPort)* declare which incoming faults the target can receive from its environment, the failure ports (*FailureOutPort)* declare which local failures may occur and propagate to the environment. A *PropagationLink* connects an output port with an input port for error propagation across different target components/systems. Such analytical ports and links can have references to the corresponding nominal ports and connectors (e.g. the data flow ports and connectors) when a fine-grained traceability is needed.

An *InternalFault* represents an internal condition of the target component/system, which can cause errors when become activated. A *ProcessFault* refers to a systematic fault introduced due to design or implementation flaws (e.g. incorrect requirements or buffer size configuration). Because any of the output failures of an error model may be caused by process flaws, a process fault affects all failures that the component may exhibit.

By means of the *FaultFailure* construct, the actual value of an anomaly can be declared (i.e. one of the possible values of an incoming fault, an internal fault, a process fault, or an outgoing failure) based on an enumeration type, such as {*omission, commission*} or {*brake-loss, brake-fade, brake-too-much*}. Within an error model, the syntax and/or semantics of existing external formalisms can be adopted for a precise description of the *FailureLogic*, capturing which output failures are caused by which input and internal faults within the target component/system. This, together with the error propagation links, makes it possible to perform safety analyses and derive the related safety constraints.

A *safety constraint* defines the qualitative bounds on the fault and failure values in terms of the safety integrity level (ASIL) for avoiding its occurrence (as defined in ISO 26262). For an incoming fault or an internal fault, a safety constraint means that the error model assumes this integrity level on the fault. For process faults and output failures, it means that the error model propagates failures at this integrity level.

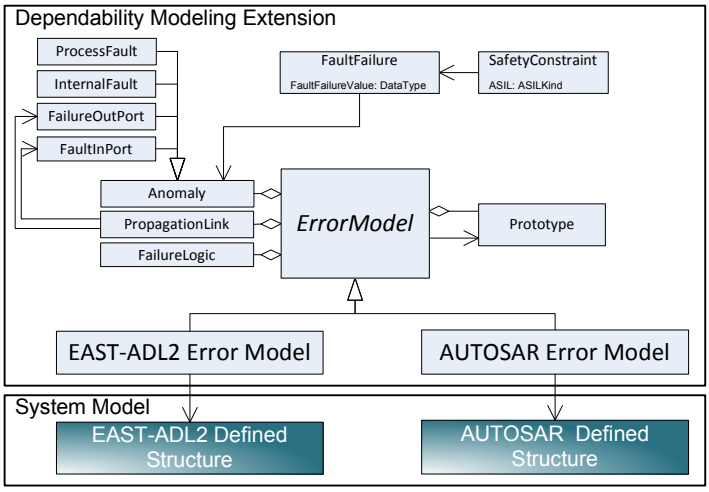



**Figure 29 - Error modelling in EAST-ADL**

### 3.3.2 Safety Analysis with HiP-HOPS Technology

Hierarchically Performed Hazard Origin & Propagation Studies, or "HiP-HOPS", is a comprehensive safety analysis methodology with a great degree of automation. In this technique, fault trees and FMEA are automatically constructed from topological models of the system that have been augmented with appropriate component failure data [4].

3.3.2.1 Overview of the methodology

A HiP-HOPS study of a system under design has four phases:

- System modelling and failure annotation
- Fault Tree synthesis
- Fault Tree analysis & FMEA synthesis
- Optimisation

The first phase is executed manually while the later three phases are fully automated.

The first phase consists of developing a model of the system (hydraulic, electrical or electronic, mechanical systems, and conceptual block and data flow diagrams) and then annotating components in that model with failure data. In this phase, modelling can be carried out in a modelling tool like Matlab Simulink as it would normally have been carried out for the purposes of simulation. Failure annotations are then added to components of the model.

The second phase is the fault tree synthesis process. In this phase, an automated algorithm is applied to the annotated system model to create a set of fault trees that define system failures and their causes in the architecture. The algorithm works by taking failures of system outputs and progressively working backwards through the model to determine which components caused those failures. System failures and component failures are then joined together using the appropriate logical operators to construct fault trees with the failures at the system outputs as the top events and the root causes as basic events. The concept is illustrated in Figure 30.

In the third phase, fault trees are analysed and an FMEA is constructed which combines all the information stored in individual fault trees. The FMEA is presented in the form of a table that lists, for each component, the effects of each component failure on the system. As part of this process, both qualitative (logical) and quantitative (numerical-probabilistic) analyses are carried out on the fault trees. These analyses provide both the minimal cut sets of each fault tree and the unavailability (i.e. failure probability) of top events.

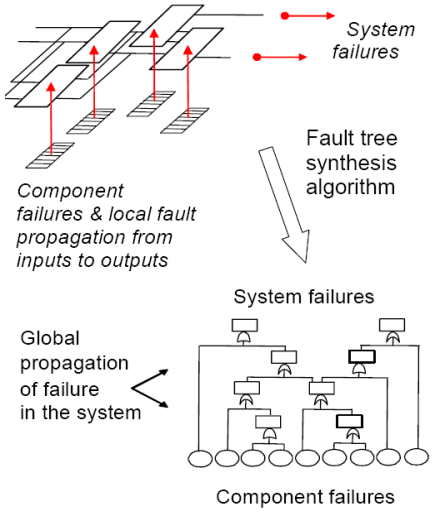



**Figure 30 - The synthesis of fault trees from the system model**

If the system is to be optimised, then the fourth phase is applied. This is an iterative optimisation of the system model, using genetic algorithms to continually evolve the model to find the best configuration of components on the basis of safety and cost.

#### 3.3.2.2 Modelling Phase

HiP-HOPs studies can be performed on any model of a system that identifies components and the material, energy or data transactions among components. Such models can be hierarchically arranged, to manage complexity, if necessary. The basic idea of HiP-HOPs is that an output failure of a component can either be caused by an input failure, an internal failure, or some combination of both. The local component output deviations and topology information are used to determine the relation between local deviations and top events.

For the purpose of the analysis, each component in the model must have its own local failure data, which describes how the component itself fails and how it responds to failures propagated by other components in the vicinity. Essentially, this information specifies the local effects that internally generated or propagated failures have on the component's outputs. This is achieved by annotating the model with a set of failure expressions showing how deviations in the component outputs (output deviations) can be caused either by internal failures of that component or corresponding deviations in the component's inputs. Such deviations include unexpected omission of output or unintended commission of output, or more subtle failures such as incorrect output values or the output being too early or late. This logical information explains all possible deviations of all outputs of a component, and so provides a description of how that component fails and reacts to failures elsewhere. At the same time, numerical data can be entered for the component, detailing the probability of internal failures occurring and the severity of output deviations. This data will then be used during the analysis phase to arrive at a figure for the unavailability of each top event. Once done, the component can then be stored together with the failure data in a library, so that other components of the same type can use the same failure data or this type of component can be re-used in other models with the same failure data. This avoids the designer having to enter the same information many times.

For the specification of the components' failure modes (which are the effects by which the component failures are observed), a generic and abstract language was developed. There are different ways of classifying failure modes, e.g. by relating them to the function of the component, or by classifying according to the degree of failure – complete, partial, intermittent etc. In general, however, the failure of a component will have adverse local effects on the outputs of the component, which, in turn, may cause further effects travelling through the system on material, energy or data exchanged with other components. Therefore in HiP-HOPS, effects are generally classified into one of three main classes of failure, all equally applicable to material, energy or data outputs. These are: the omission of an output, i.e. the failure to provide the output; a commission of an output, i.e. a condition in which the output is provided inadvertently and in the wrong context of operation; and an output malfunction, a general condition in which the output is provided but at a level which deviates from the design intention. Since this classification adopts a functional viewpoint which is independent of technology, it could provide a common basis for describing component failures and their local effects. However, HiP-HOPS can work with any classification of failure modes as long as it is consistent from one component to the next.

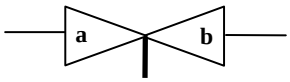
Components do not only cause failures; they also detect and respond to failures caused by other components or transfer failures of other components. In addition to generating, mitigating, or propagating failures, they may also transform input failures to different types of output failure. For instance, a controller may be designed to respond to detected sensor failures by omitting any further output to ensure that hazardous control action is avoided. In this case, malfunctions at the input are intentionally transformed into omission failures. To capture those general patterns of behaviour of a component in the failure domain, manual analysis is performed at component level and focuses on the output ports through which a component provides services to other components in the system. In the course of the analysis, each output port is systematically examined for potential deviations of parameters of the port from the intended normal behaviour, which generally fall into the following three classes of failure:

(a) Omission: failure to provide the intended output at the given port

(b) Commission: unintended provision of output

(c) Malfunction: output provided, but not according to design intention.

Within the general class of malfunctions, analysts may decide to examine more specific deviations of the given output which, in most applications, will include conditions such as the output being delivered at a higher or lower level, or earlier or later than expected. As an example, Figure 31 shows the analysis of a two-way computer controlled valve. The figure shows the valve as it would typically be illustrated in a plant diagram and records the results of the local safety analysis of the component in two tables that define valve malfunctions and output deviations respectively.




***Valve Malfunctions***

| *Failure mode* | *Description* | *Failure rate* |
|---|---|---|
| blocked | e.g. by debris | 1e-6 |
| partiallyBlocked | e.g. by debris | 5e-5 |
| stuckClosed | Mechanically stuck | 1.5e-6 |
| stuckOpen | Mechanically stuck | 1.5e-5 |

***Deviations of Flow at Valve Output***

| *Output Deviation* | *Description* | *Causes* |
|---|---|---|
| Omission-b | Omission of flow | blocked OR stuckClosed OR Omission-a OR Low-control |
| Commission-b | Commission of flow | stuckOpen OR Commission-a OR Hi-control |
| Low-b | Low flow | partiallyBlocked OR Low-a |

**Figure 31 - Failure annotations of a computer-operated two-way valve**

In normal operation, the valve is normally closed and opens only when the computer control signal has a continuously maintained value of a logical one. Valve malfunctions include mechanical failures such as the valve being stuckOpen or stuckClosed, and blockages caused by debris such as blocked and partiallyBlocked. For each malfunction, the analysis records an estimated failure rate while the effects of those malfunctions on the output of the valve can be seen in a second table that lists output deviations.

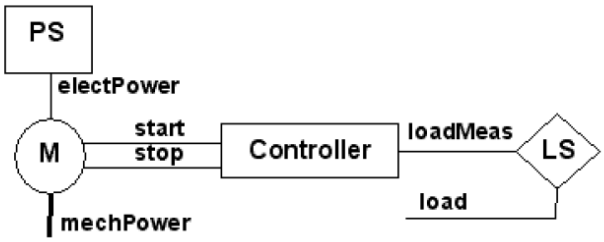
This specification of failure modes is generic in the sense that it does not contain references to the context within which the valve operates. Failure expressions make references only to component malfunctions and input/output ports of the component. The failure behaviour described in these expressions has been derived assuming a simple operation that the component is expected to perform in every application (valve is normally closed unless the value of control signal is 1). For these reasons, the specification of Figure 31 provides a template that could be re-used in different models and contexts of operation, perhaps with some modifications, e.g. on failure rates, to reflect a different environment.

#### 3.3.2.3 Synthesis Phase

As seen in Figure 31, component failure data relate output deviations to logical expressions that describe the causes of those deviations as component malfunctions and deviations of the component inputs. Each such expression is effectively a mini fault tree that links a top event (the output deviation) to leaf nodes, some of which may represent input deviations. When a component is examined out of system context, input and output deviations represent only potential conditions

of failure. However, when the component is placed in a model of a system, the input deviations specified in the analysis can actually be triggered by other components further upstream in the model and the specified output deviations can similarly cause more failures further downstream.

This mechanism by which output failures of a particular class at one end of a connection trigger input failures of the same class at the other end results in a global propagation of failures through the system which may ultimately cause significant hazardous failures at the outputs of the system. Given a model of the system and the local safety analyses of its components, it is possible to capture this global propagation of failure in a set of fault trees. These fault trees are mechanically constructed by traversing the model and by following the propagation of failure backwards from the final elements of the design (e.g. electromechanical actuators) towards the system inputs (e.g. material/energy resources, operators and data sensors). The fault tree is generated incrementally, as the local safety analyses of the components are encountered during the traversal, by progressively substituting the input deviations for each component with the corresponding output failures propagated by other components. Figure 32 illustrates the principle that underpins this process of fault tree synthesis. The figure shows a hypothetical motor and its starter circuit as a unit (M) that transforms electrical power provided by a power supply (PS) to mechanical power on a rotating axis.




| *Component* | *Output Deviation* | *Description* | *Causes* |
|---|---|---|---|
| **M** | OE-mechPower | Omission of mechanical power | motorFailed OR OE-electPower OR OS-start OR CS-stop |
| **PS** | OE-electPower | Omission of electrical power | powerSupplyFailed |
| **Controller** | OS-start | Omission of start signal | controllerFailed |
| | CS-stop | Commission of stop signal | elecMagnInterference OR HV-loadMeas |
| **LS** | HV-loadMeas | Hi value in load measurement | sensorBiased |

**Figure 32 - Example system and fragments of local safety analyses**

The motor starter receives normal start and stop commands from a control computer (Controller). As a safety feature the controller is also connected to a sensor (LS) that monitors the load connected to the axis of the motor and when the measurement exceeds a specified load it issues a stop command to protect the motor. To illustrate the fault tree synthesis, the figure also provides a table that contains, for simplicity, only fragments from the local analyses of the Motor, the Power Supply, the Controller and the Load Sensor. For simplicity, deviations in this table refer to names of connections in the model rather than local names of component ports. Collectively, these deviations and their causes define the propagation of failures that result in an omission of mechanical power at the output of the motor.

The local analysis of the motor, for example, defines that this event can be caused by a failure of the motor, an omission of electrical power at the input, an omission of the start signal or, interestingly, a commission of the stop signal. The causes of some of those events can in turn be explored further in the local analyses of the components connected to the motor. For example, the analysis of the power supply defines that a failure of this component will cause an omission of electrical power. In turn, the analysis of the controller defines that an omission of the start signal will be caused by a controller failure while a commission of the stop signal can be caused by electromagnetic interference or in response to an abnormally high measurement of motor load.

The analysis of the load sensor defines that the latter is indeed a plausible failure mode that can be caused in normal conditions of loading if the sensor is positively biased.

An overall view of the global propagation of failure in the system can be automatically captured by traversing the model and by following the causal links specified in the local safety analyses of the components that are progressively encountered during this traversal. The result of this process for the above example is the fault tree that is illustrated in Figure 33. Note that this mechanically synthesised fault tree records the propagation of failure in a very strict and methodical way. It starts from an output failure, the omission of mechanical power, and following dependencies between components in the model it systematically records other component failures that progressively contribute to this event. The logical structure of the tree is determined only by interconnections between the components and the local analyses of those components. This logical structure is straightforward and can be easily understood, unlike the structure of many manually constructed fault trees, which is often defined by implicit assumptions made by analysts. Note that although in this example the tree only incorporates OR gates, other logical symbols such as AND and priority AND gates would appear in the tree structure if such relationships were originally specified in some of the local analyses.

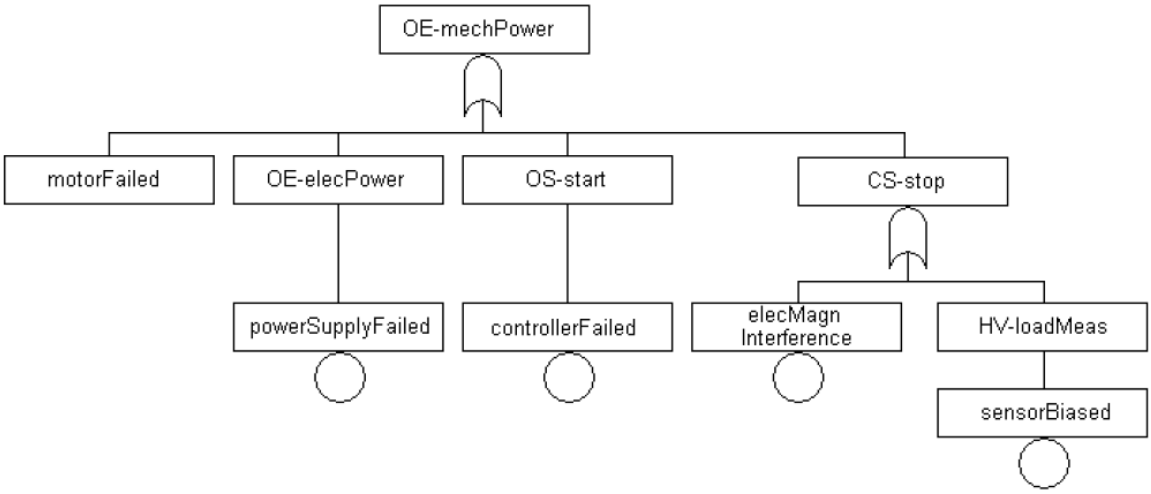
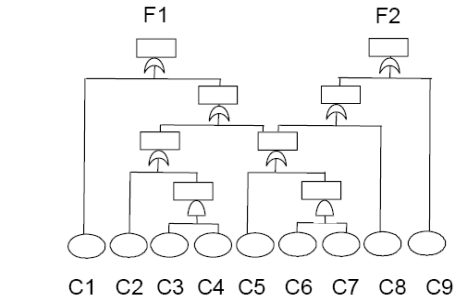



**Figure 33 - Mechanically constructed fault tree for the example system**

### 3.3.2.4 Analysis Phase

In the final phase, the synthesised system fault trees are analysed, both qualitatively and quantitatively, and from these results the FMEA is created. Firstly, the fault trees undergo qualitative analysis to obtain their minimal cut sets, which reduces them in size and complexity. This is typically done using a mixture of classical logical reduction techniques, which usually means applying logical rules to reduce complex expressions, and more modern techniques developed specifically for HiP-HOPS. Once the minimal cut sets have been obtained, they are analysed quantitatively, which produces unavailability values for the top events of each fault tree.

The last step is to combine all of the data produced into an FMEA, which is a table that concisely illustrates the results. The FMEA shows the direct relationships between component failures and system failures, and so it is possible to see both how a failure for a given component affects everything else in the system and also how likely that failure is. However, a classical FMEA only shows the *direct effects* of single failure modes on the system, but because of the way this FMEA is generated from a series of fault trees, the HiP-HOPS is not restricted in the same way, and the FMEAs produced also show what the *further effects* of a failure mode are; these are the effects that the failure has on the system when it occurs in conjunction with other failure modes. Figure 34 shows this concept.




Multiple Failure System FMEA

| Component failure | Direct effects on the system | Effects caused in conjunction with (other events) |
|---|---|---|
| C1 | F1 | - |
| C2 | F1 | - |
| C3 | - | F1 (C4) |
| C4 | - | F1 (C3) |
| C5 | F1,F2 | - |
| C6 | - | F1,F2 (C7) |
| C7 | - | F1,F2 (C6) |
| C8 | F2 | - |
| C9 | F2 | - |

**Figure 34 - The conversion of fault trees to FMEA**

In Figure 34, F1 and F2 are system failures, and C1 – C9 are component failures. For C3, C4, C6 and C7, there are no direct effects on the system – that is, if only one of these components fail, nothing happens. However, they do have further effects; for example, C3 and C4 both occurring in conjunction will cause F1 to occur. The FMEAs produced, then, show all of the effects on the system, either singly or in combination, of a particular component failure mode. This is especially useful because it allows the designer to identify failure modes that contribute to multiple system failures (e.g. C5 in the example). These common cause failures represent especially vulnerable points in the system, and are prime candidates for redundancy or extra reliable components.

### 3.3.3 Multi-perspective analysis of EAST-ADL models

EAST-ADL does not limit the modeller to a single architecture, as HiP-HOPS does. Not only are the Error Model and nominal model separate, but EAST-ADL provides multiple layers or levels of potential modelling as well as different views or *perspectives* of a model. HiP-HOPS is expected to be applied to EAST-ADL models at the analysis/artefact and design levels, and at these levels EAST-ADL offers a number of different perspectives. At the analysis level, the primary perspective is the Functional Analysis Architecture (FAA), which is a relatively abstract view of the functions of the system. At the design level, the model is more concrete and is separated into the Functional Design Architecture (FDA) or software perspective and the Hardware Design Architecture (HDA) or hardware perspective.

Although HiP-HOPS is designed to be able to support iterative analyses at different levels of abstraction, and thus can cope equally well with an initial qualitative analysis of an abstract functional model and a later quantitative analysis of a more detailed component model, it was only designed to support a single perspective. The concept of separating software and hardware perspectives of the same system is without analogue in HiP-HOPS. This is because HiP-HOPS assumes a compositional model that combines hardware and software, with individual software functions (or subsystems of functions) contained within the hardware processors that run them. Therefore, although HiP-HOPS is capable of analysing both software and hardware, it requires them to be in the same model, with the hardware the primary means of failure propagation: it is not possible for failures in one software function to propagate directly to another remote, physically separated software function; instead software failures must propagate up to their parent hardware component and then propagate along hardware connections to other hardware components that carry the affected functions.

This can be emulated in EAST-ADL by combining everything into a single perspective, but the advantages of having multiple perspectives are then lost. Instead, HiP-HOPS has been extended to provide native support for *multi-perspective analysis*: the analysis of failure behaviour across two or more separate perspectives of the same system (e.g. H/W and S/W). This required the introduction of the concept of a 'perspective' to the HiP-HOPS model hierarchy.

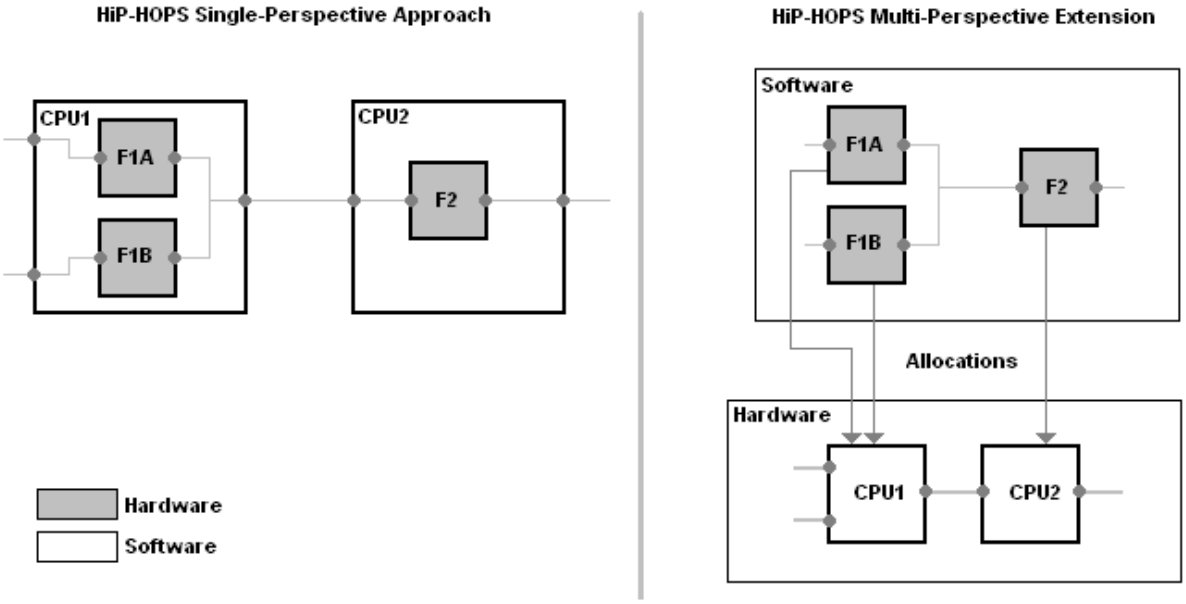



**Figure 35 - Multi-Perspective Modelling in HiP-HOPS**

In EAST-ADL, hardware and software entities are in separate architectures (HDA and FDA) and communication from one perspective to the other is accomplished by means of *allocation* relationships: software functions are allocated to hardware components that execute them. Failures of the hardware components will propagate to the software functions allocated to them. Failures can also propagate from H/W to H/W and also from S/W to S/W, but this form of connection is restricted to only H/W → S/W propagations. This avoids problems inherent in the multi-perspective modelling paradigm where failures can initiate intractable circular logic loops when propagating through the system (e.g. a S/W fault could cause a H/W failure that in turn is caused by the same S/W fault occurring).

HiP-HOPS has been extended with a similar approach, as shown in Figure 35. Although the original single-perspective approach is still possible, when analysing FDA/HDA models from

EAST-ADL, the new multi-perspective representation can be used. To accomplish this, a new 'Perspective' layer in the model hierarchy was added:

- *Model* (top-level entity representing an entire system under analysis)
- *Perspective* (contains a different perspective of the entire system, e.g. S/W, H/W)
- *Subsystem* (subsystem containing one or more components)
- *Component* (component representing some H/W or S/W entity)
- *Implementation* (the failure behaviour of a component; may also contain a subsystem)

Any number of perspectives can be used, but one perspective must be the 'primary' perspective. Since EAST-ADL assumes that the FDA is the primary perspective, and restricts failures to propagating from H/W to S/W but not from S/W to H/W, it is assumed that an exported EAST-ADL model will use the software perspective as the primary perspective in HiP-HOPS too.

As in EAST-ADL, HiP-HOPS also provides means for failures to propagate from one perspective to another. As in EAST-ADL, it is possible to define 'allocations' of components in one perspective to components in another perspective. Failures can propagate along these allocations by means of 'External Propagation' constructs. In fact, in both EAST-ADL and in HiP-HOPS, it is possible to define *more* than one possible allocation, e.g. using variability constructs in EAST-ADL. This allows scope for potential optimisation of the model on the basis of changing the allocation of software functions to different hardware components, and indeed the optimisation capabilities of HiP-HOPS are also being extended in that direction.

HiP-HOPS provides for certain types of 'local failure data'. Traditionally this has included definitions for Basic Events (representing internal failure modes), Output Deviations (representing the propagation of failures at component outputs), and Common Cause Failures (linked to common model-level failures). Each, once declared, is visible to other definitions in that component, e.g. a basic event can be used in an output deviation of the same component.

Due to multi-perspective analysis, HiP-HOPS now also supports the definition of 'External Propagation' failure data. These are equivalent to Output Deviations, but rather than representing the propagation of failure from the component outputs, they represent the propagation of failure across allocation relationships. Also, unlike other failure data types, External Propagations are visible to allocated components as well as the local component. This means that they can be used in the definition Output Deviations of allocated components.

For example, consider CPU2 and F2 in Figure 35 above. We may define the local failure data of CPU2 to include:

- Basic event "CPU Failure", optionally with appropriate failure rates etc.
- Output Deviation "Omission-output", which is caused by either "Omission-input" or "CPU Failure" (defined above) or "EMI" (defined below).
- Potential Common Cause "EMI", which is caused by electromagnetic interference affecting the entire system.

In addition, because we know that CPU2 can have software functions allocated to it, we define an External Propagation like so:

- External Propagation "Omission", which is caused by "Omission-input", "CPU Failure", or "EMI".

Then when we define the local failure data for F2, which is a software function allocated to CPU2, we can reference this external propagation in F2's output deviation:

- Basic event "Undetected bug"
- Output Deviation "Omission-fout", caused by "Omission-fin" or "Omission". The external propagation is referred to more explicitly as "FromAllocation(Omission)".

Because any component can only be allocated to one component at a time (it has at most one 'current allocation'), we know that "Omission" must refer to an external propagation found in CPU2. HiP-HOPS can therefore link this "Omission" in F2 to the failure logic for "Omission" in CPU2 during its fault tree synthesis process, connecting the perspectives together and allowing for propagation of failure from hardware to software.

Note also that the semantics of common cause failures have been changed too; whereas before common cause failures (CCFs) were globally visible throughout the model, now CCFs are defined per perspective, thus a CCF defined for the hardware perspective (e.g. flooding, fire) would not be accessible from a component in the software perspective, for instance.

HiP-HOPS also allows a more flexible form of connection by means of the 'LocalGoto' and 'GlobalGoto' declarations. These support a more abstract type of connection for situations where there is no form of allocation relationship between two components in different perspectives, but propagation is nevertheless necessary. In these cases, the Gotos act like special forms of input deviations that can connect directly to another output deviation elsewhere in the model. Local gotos only connect to output deviations in the same subsystem (hence 'local') whereas Global gotos can connect to any fully-qualified output deviation in any perspective. For example:

- "O-F2.out = LocalGoto(O-F1A.out)" is valid and states that O-F2.out is caused by O-F1A.out.
- "O-F2.out = LocalGoto(O-CPU1.out)" is invalid as the output deviations are not in the same subsystem.
- "O-F2.out = GlobalGoto(O-CPU1.out)" is valid and states that O-F2.out is caused by O-CPU1.out.

However, these gotos are considered potentially harmful to a coherent analysis and should only be used sparingly, because they break the connection between the propagation of failure and the architecture of the system.

Because these new connections (allocations and gotos) connect existing HiP-HOPS constructs (namely, output deviations), they fit relatively seamlessly into the HiP-HOPS fault tree synthesis process. Once the fault trees are generated, then they can be analysed as normal without any further distinction between one perspective and another.

#### 3.3.3.1 Quantitative analysis and ISO26262

According to ISO 26262, only hardware is subject to quantitative failure requirements. Furthermore, it is the entire hardware platform supporting a specific ASIL X safety requirement that must conform to the ISO 26262-specified failure probability limit specified for that ASIL. This is

because the requirement failure rate would increase as more components were added. For example, 10 components (in series configuration) each meeting the ASIL D failure rate requirement would be 10 times less reliable than a single ASIL D component.

This also increases the importance of multi-perspective analysis, as an accurate representation of the overall failure behaviour can only be obtained using a combined error model for both functional and hardware architectures. This allows us to assess which hardware failures contribute to the top event corresponding to a particular ASIL X safety requirement (and how), and so the combined failure rate must then be lower than what is required for ASIL X.

Thus, unlike with the functional components, the ASIL fulfilment cannot be assessed purely on individual components.

### 3.3.4 Dynamic SM-based Safety Analysis

State Machines (SMs) have become a prevalent paradigm for the description of dynamic systems. Such models are well-suited to representing the behaviour of complex systems, including in conditions of failure, and where the order in which failures and fault events occur can affect the overall outcome (e.g. total failure of the system). For certain types of safety assessments though, the SM failure behavioural models need to be converted to analysis models like Generalised Stochastic Petri Nets (GSPNs) & Markov Chains (MCs) — often for quantitative analysis — or to Fault Trees (FTs) for deductive, qualitative analysis.

#### 3.3.4.1 Motivation

One of the existing SM-based analysis approaches consists of the conversion of SMs to GSPNs. This was proposed for use with AADL in [23] — AADL error models are effectively state automata showing e.g. transitions from normal to degraded and failed states [24]. Yet this approach is mainly focused on the evaluation of quantitative measures, thereby requiring the definition of failure distribution for the basic events; it can be problematic to perform early qualitative analysis in a deductive process, i.e. establishment of full causal relationships between effects and causes of failure, as in traditional analysis methods like FTA. Qualitative analysis is particularly important when probabilistic data are not available, e.g. at early stages of design; decisions made at these early stages can be critical for determining the future shape of the system, and so it is important that safety can be taken into account at all stages of the design process.

An alternative approach involves conversion of SMs to fault trees, e.g. as applied to AADL models [25] and to Altarica descriptions of systems [26]. Fault trees are logical networks of events that show how combinations of failures can cause a given system failure and are ideally suited for qualitative analysis. However, there are difficulties with this type of conversion; in particular, the temporal semantics of SMs (which are dynamic models) are lost in the translation to combinatorial fault trees (which are static models), and this can potentially cause serious errors/inaccuracies (e.g. when the sequencing of faults affects the outcome). There have been some efforts made to work around this issue; for example, in [26], NOT gates were incorporated into the conversion to fault trees to indicate that some events did not occur. Although this prevents a conjunction of two mutually exclusive SM paths occurring as an analysis result, it still cannot distinguish between paths that differ only in sequence – e.g. two faults which lead the system into two mutually exclusive states depending on which fault occurred first.

#### 3.3.4.2 Potential solution

To remedy the problem of converting dynamic models to static fault trees, an approach which consists of generating Temporal Fault Trees (TFTs, which are also dynamic) from SMs has been described in [27]. The technique mainly uses a Priority-OR (POR) gate to differentiate sequences of faults during the conversion. The gate can represent situations where one event takes priority

over others and must occur first, but without specifying that the other events must also occur – e.g. A POR B is true if A occurs and B does not, or if A occurs and B occurs but A occurs first. One of the advantages of the conversion algorithm in [27] is that the temporal constraints are imposed only when necessary during the conversion. This will positively impact the efforts needed for the minimisation of the generated fault trees — minimisation of TFTs is known for being a complex process. For example, a SM of a static system where two events e.g. A then B lead to a 'Degraded' state and two other different events e.g. C then D lead to a 'Failed' state can be adequately represented by standard Boolean logic, and thus a correct analysis can be performed afterwards. In such a situation, the two paths have no events in common and each event contributes only to a single final state. Here, a change in the sequence of the events (e.g. D before C instead of C before D) will not lead to a different failure. In the case that D happens first, the system simply stays in the initial state and when C occurs it performs two instantaneous transitions to reach the final 'Failed' state; thus the failure behaviour of this example is not sequence-dependent.

Hence, in general a SM will have as many fault trees as there are final states. Each final state represents a system failure (i.e., one fault tree top-event) and each transition between states represents an event in a fault tree. Every full path between the initial state and a given final state becomes a new branch of its corresponding FT — i.e., represented by the conjunction of the events that label that path. In particular, if some system failures are sequence-dependent then the corresponding FTs should be temporal. Imposing temporal constraints during the conversion depends on whether or not the SM has at least one event appearing in more than one path. Typically, if there is at least one event that contributes to the occurrence of more than one system failure, then conversion to temporal fault trees may be needed. It may also be true even if there is an event that is a contributory factor to the occurrence of only one system failure, but as a result of more than one sequence of events. In such cases, an accurate analysis depends upon the correct preservation of the temporal semantics, as different sequences of those shared events – or other events relative to those shared events – may lead to different final states (and thus different system failures). The application of the approach in [27] allows complex dynamic analysis to be applied only when necessary, depending on whether each part of the system is static or dynamic.

To this end, a state machine path can be traversed forwards or backwards. The (temporal) fault tree expression $\phi_s$ which corresponds to a final state s is generated by performing backward traversals of all paths starting from that final state to the initial state. Forward traversals are performed starting from every 'join' state (i.e., a state at which paths diverge) to, at worst, all reachable final states, or until the condition for imposing a temporal constraint is satisfied. Therefore, $\phi_s$ is the disjunction over the paths $\pi$ (from the initial state to s) of the conjunction of events that label $\pi$. Moreover, any one of the events e that label $\pi$, which is incident from a state u with an out-degree of two or more (i.e., a 'join' state), is the input event, which must occur first or alone, of a POR gate that associates it with the disjunction of other events e' incident from u to the states u' if, and only if, the subpath of $\pi$ from u to s shares an event with the paths from u through u' to any state.

This approach of converting SMs to (temporal) FTs also deals with the issue of scalability by proposing a recent method, termed SAFORA [27] — State Automata to Fault-trees extended, if necessary, with temporal information (ORA or time, 'ώρα' in Greek). This method uses a process of composition; it helps to improve scalability by generating large-scale system (temporal) FTs that represent the failure conditions of a system from the SMs that describe the behaviour of its components, without building the full-scale system's SM.

#### 3.3.4.3 Overview of the methodology

A central issue on which Safora relies is the compositional modelling of the system behaviour. This consists of a highly abstract SM which describes the monolithic failure behaviour of the system at the top-level of the hierarchy and the component & sub-component SMs at the lower levels (see Figure 36). Typically, each event of the abstract SM of the system is the effect of one (or more)

final state(s) being reached in some component SM(s). If it is the effect of more than one component SM each reaching a final state, then the order in which the final states were entered may affect the overall outcome (and in which case the conversion algorithm will enforce a temporal order).

For example, if two of the SMs of the sub-components level (those which are downstream in the event-causal chain in bottom of Figure 36) each enters a final state, then the remaining SM in the same level enters its final state too. Thereafter, the conversion algorithm (when applied to the latter SM) will determine if the temporal order in which the first two final states were reached is relevant and thus needs to be preserved. Moreover, several final states in one SM can each when entered cause the same event to occur in any other SM (upstream in the event-causal chain) — one example can be such that an effect which is caused by a disjunction of final states being reached in a given downstream SM. Another example can be a common cause failure like in the SM of the sub-components level of Figure 36, and whose final state also causes a safety issue at system level (in the top of the hierarchy).

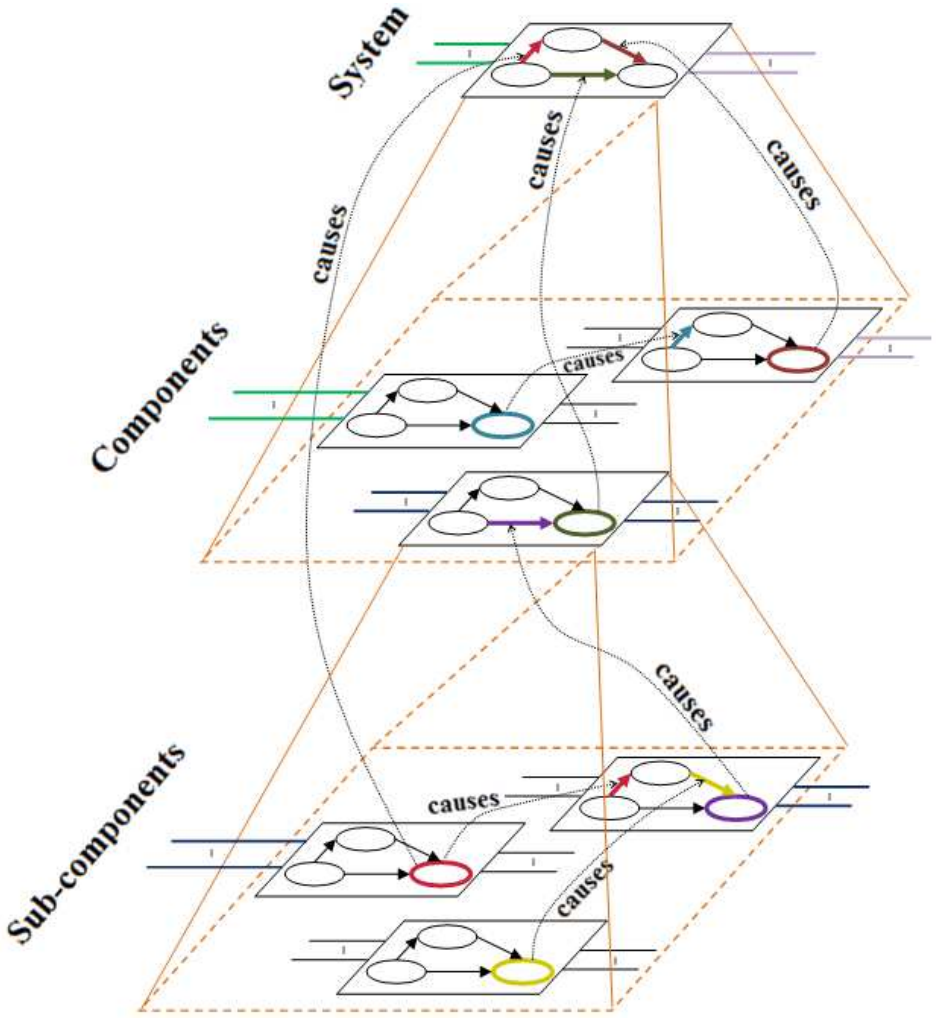



**Figure 36 - Compositional modelling of the system failure behaviour, from [27]**

With this type of description only small-sized SMs, with a limited number of paths, are to be converted and the temporal expressions of the failure conditions of each component are hence computed and then combined by symbolic calculus. Therefore, the global TFTs are obtained from algebraic expressions, without composing SMs. This has the benefit of simplifying the task of the

system designers, allowing them to focus on modelling local failures in each component and also helping to reduce the efforts which are required to perform temporal analysis by producing smaller, more manageable TFTs via the compositionality.

The work in [27] formally describes the algorithms which generate and synthesise fault trees from a state automata description of the failure behaviour using the below definitions:

- A state machine has a finite number of states. It may change state when an event occurs, but at each instant it is in only one state. Definition 1 formally describes a state machine.

**Definition 1:** ***State Machine (SM)***
*A state machine is a quadruple* $(S, \Sigma, \delta, s_0)$ *where:*

- $S$ *is a finite set of states.*
- $\Sigma$ *is a finite set of events, such that* $S \cap \Sigma = \emptyset$.
- $\delta$ *is a partial function:* $S \times \Sigma \longrightarrow S$ *s.t. for* $(u, u') \in S^2$ *and* $e \in \Sigma$, $u' = \delta(u, e)$ *iff* $e$ *is incident from* $u$ *to* $u'$, *and we write it as:* $u \xrightarrow{e} u'$.
- $s_0$ *is the initial state.*

- A state machine is assumed to be acyclic. A cyclic SM may imply that failures are repairable or repeatable, which is incompatible with the semantics of the (temporal) FTs. There is, therefore, a finite set of possible paths in the state machine. If π is a path from u to u', where (u, u') $\in S^2$, we write it as $u \overset{\pi}{\rightsquigarrow} u'$.

**Definition 2:** ***Paths set***
*Let* $P$ *be the set of all paths in the SM,*
$P = \{\pi \mid u \overset{\pi}{\rightsquigarrow} u',\ (u, u') \in S^2\}$

- We write $u \rightsquigarrow u'$ iff $\exists \pi \in P$ s.t. $u \overset{\pi}{\rightsquigarrow} u'$. In such a case, state u' is said to be reachable from state u. However, if there exists a one-event path from u to u', then u' is said to be immediately reachable from u and we write it as $u \rightarrow u'$ (i.e., $u \rightarrow u'$ iff $\exists e \in \Sigma$ s.t. $u \xrightarrow{e} u'$).
- We assume that from $s_0$ we can reach any other state, i.e., $\forall t \neq s_0 \in S\ s_0 \rightsquigarrow t$.
- Also, $\forall \pi \in P\ \exists (u, u') \in S^2$ s.t. $u \overset{\pi}{\rightsquigarrow} u'$ and $Seq_\pi = \langle u_0, u_1, \ldots, u_n \rangle$ is the sequence of the states of the path π, where n = length(π) is the number of events that label π and the sequence is ordered for backward traversal, i.e., $u_0$=u' and $u_n$=u.
- A state machine path can be traversed forwards or backwards. The TFT formula which corresponds to a final state is generated by performing backward traversals of all paths starting from that final state to the initial state. Forward traversals are performed starting from every 'join' state (i.e., with an out-degree strictly greater than 1) to, at worst, all reachable final states, or until the condition for imposing a temporal constraint is satisfied.

**Definition 3:** ***Forward and backward incidence sets***
*For any state* $u \in S$, *let* $\Sigma_{u\blacktriangleright}$ *(resp.* $\Sigma_{u\blacktriangleleft}$*) be the set of events incident from* $u$ *(resp. incident to* $u$*),*
$\Sigma_{u\blacktriangleright} = \{e \in \Sigma \mid \exists u' \in S \text{ s.t. } u \xrightarrow{e} u'\}$
$\Sigma_{u\blacktriangleleft} = \{(e, u') \in \Sigma \times S \mid u' \xrightarrow{e} u\}$

The choice of an event in $\Sigma_{u\blacktriangleright}$ uniquely determines the state that the event is incident to. As for $\Sigma_{u\blacktriangleleft}$, an event incident to u needs to be distinguished from every possible

identical event also incident to u, but from a different state. This is done by associating each event with the state from which it is incident, and hence the definition 3 above.

- All final states are permanent states — there are no events that lead from a final state to any other state.

**Definition 4:** *Set of final states*
*Let F be the set of the final states,*
$F = \{f \in S | \Sigma_{f\blacktriangleright} = \emptyset\}$

- Let $\phi_s$ s∈F be the (temporal) FT formula which we generate for a final state s.

Now assuming we have an system with several complex interrelated components, each modelled by its own SM, Safora makes it possible to construct the system TFTs for a final analysis in a compositional manner. The first step consists of transforming the hierarchy top level SM — i.e., a highly abstract description of the monolithic behaviour of the system — to produce a set of preliminary TFTs (see top of Figure 37).

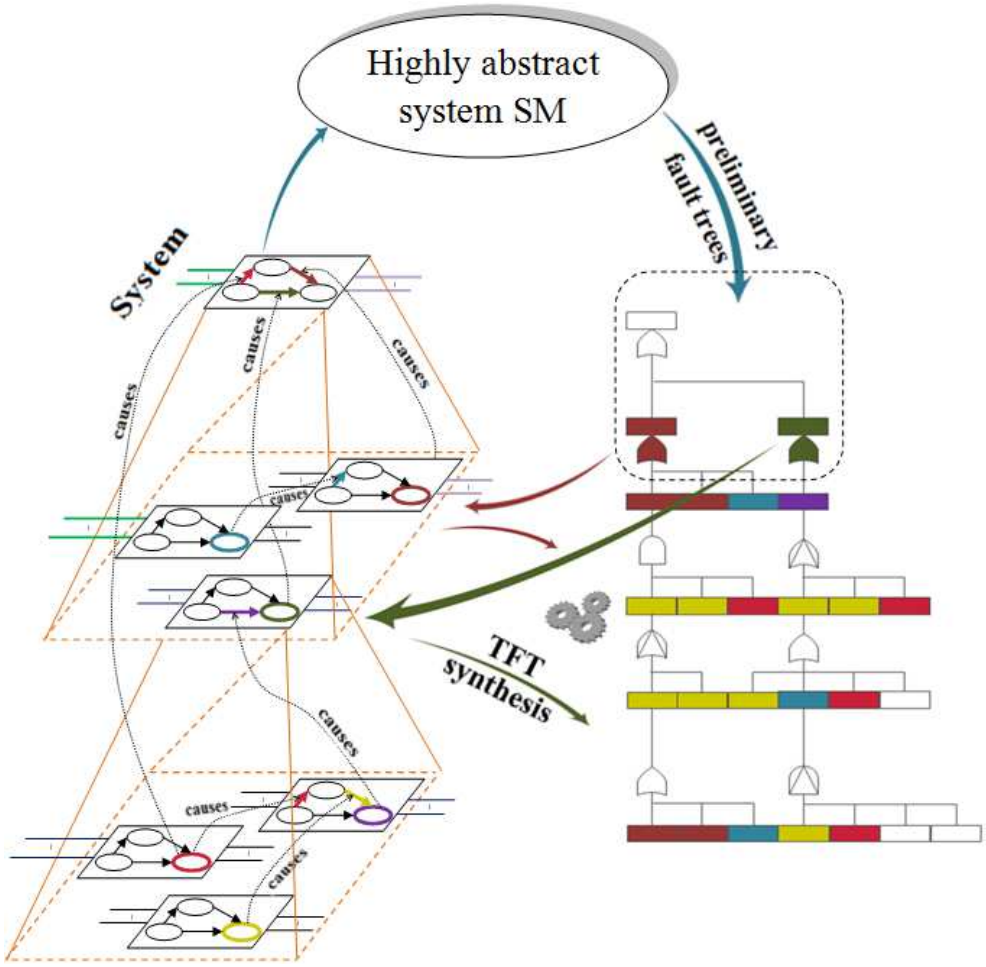



**Figure 37 - Overview of the Safora method, from [27]**

Algorithm 1 takes the top level SM as parameter. Then, for each final state (line 3), it generates the corresponding failure expression (line 4) of a preliminary system TFT. Ultimately, the algorithm

produces a set of system failure expressions (line 7); each corresponds to the first level of a system TFT.

**Algorithm 1** : $\Phi_{syst}$ GenerateSystFailureTFTs($SM_{syst}$)

**Input:** A highly abstract $SM_{syst}=(S, \Sigma, \delta, s_0)$ describing the monolithic behaviour of the system.

**Output:** A set $\Phi_{syst}$ of preliminary system failure TFTs.

```
1: let Φ_syst = ∅
2: get F from S
3: for each s∈F do
4:     let φ_s = GenerateFailureExpression(SM_syst, s)
5:     let Φ_syst = Φ_syst ∪ {φ_s}
6: end for
7: return Φ_syst
```

The 'GenerateFailureExpression' method in line 4 of the algorithm takes the system abstract SM as first parameter and a final state s (s□F) for which it generates a failure expression as second parameter — Algorithm 2 generates that failure expression denoted □s. Then □s can be logically reduced using, e.g., the set of temporal laws of Pandora.

To briefly describe Algorithm 2, let $\pi_i$ $1 \le i \le n$ be the paths in the SM (line 4) and let $s_{ij}$ be the $j^{th}$ visited state of the path $\pi_i$ during the backward traversal of that path (line 6). Then, for each join state $s_{ij}$ during the traversal (line 8), a temporal constraint is imposed (line 13) if, and only if, the set of the events that label the sub-path h of $\pi_i$ which has been already traversed backwards from s to $s_{ij}$ (i.e., $s_{ij} \overset{h}{\rightsquigarrow} s$) shares an event with the remaining, i.e., the other paths from the join state $s_{ij}$ to all possible reachable final states. However, these remaining paths need to be traversed forwards until a common event with the events of h is found.

At this stage, since these □s failure expressions are generated from a highly abstract SM, each typically contains mainly non-atomic (i.e., expandable) events. For example, Figure 37 shows the first level of the system fault tree composed of two non-atomic events, each is a fault tree top-event which corresponds to a SM final state of a lower level component — entering one of these final states causes a transition to the final state in the SM upstream, i.e. a complete failure which corresponds to the top-event of the preliminary system FT.

**Algorithm 2** : $\phi_s$ GenerateFailureExpression(SM, s)

**Inputs:** - A state machine SM = (S, Σ, $\delta$, $s_0$).
- A final state s of SM, i.e. s∈F (F⊂S).

**Output:** The failure expression $\phi_s$ for s.

1: get P from SM
2: get Ps from P where Ps = {$\pi$∈P| $s_0 \overset{\pi}{\rightsquigarrow}$ s}
3: let n = |Ps|
4: let $\pi_i$∈Ps $1 \le i \le n$ /* $s_0 \overset{\pi_i}{\rightsquigarrow}$ s */
5: let $len_i$=length($\pi_i$) $1 \le i \le n$ /* the number of events of $\pi_i$ */
6: let $Seq_{\pi_i}$=⟨$s_{i_0}$, $s_{i_1}$, …, $s_{i_{len_i}}$⟩ $s_{i_j}$∈S $e_{i_j}$∈Σ $1 \le i \le n$ $1 \le j \le len_i$ s.t. $s_{i_j} \overset{e_{i_j}}{\rightarrow} s_{i_{(j-1)}}$ where $Seq_{\pi_i}$ is ordered from $s_{i_0}$=s to $s_{i_{len_i}}$=$s_0$ /* $Seq_{\pi_i}$ $1 \le i \le n$ is the sequence of states of $\pi_i$ ordered for backward traversal*/
7: let $\phi_s = \bigvee_{1 \le i \le n}(\bigwedge_{1 \le j \le len_i} e_{i_j})$/*$\phi_s$ is the disjunct over paths $\pi$ of the conjunct of events of $\pi$*/
8: **for each** $s_{i_j}$ in $Seq_{\pi_i}$ $1 \le i \le n$ $1 \le j \le len_i$ s.t. $|\Sigma_{s_{i_j}\blacktriangleright}|>1$ **do**
9: let e = $e_{i_j}$ /* $\Sigma_{s_{i_j}\blacktriangleright}$ is the set of events incident from $s_{i_j}$ */
10: **for each** e'≠$e_{i_j}$ in $\Sigma_{s_{i_j}\blacktriangleright}$ **do**
11: let t∈S s.t. $s_{i_j} \overset{e'}{\rightarrow}$t
12: **if** ((e'∈{$e_{i_{(j-1)}}$, $e_{i_{(j-2)}}$, …, $e_{i_1}$}) **or** (∃(y∈Σ, $\pi'$∈P, f∈S, v∈S in $Seq_{\pi'}$, w∈S in $Seq_{\pi'}$) s.t. $t \overset{\pi'}{\rightsquigarrow}$f, $v \overset{y}{\rightarrow}$w and y∈{$e_{i_j}$, $e_{i_{(j-1)}}$, …, $e_{i_1}$})) **then**
13: let e = e|e' /* e' is one of the events that label subpath h of $\pi_i$ s.t. $s_{i_j} \overset{h}{\rightsquigarrow}$s, or h and $\pi'$ share an event */
14: **end if**
15: **end for**
16: replace $e_{i_j}$ with e in $\phi_s$
17: **end for**
18: **return** $\phi_s$

Algorithm 3 is developed to synthesise the lower levels of such FTs by expanding every non-atomic event (lines 2 and 3). We would like to emphasise, though, that a non-atomic event is handled indistinguishably in the Safora approach. In that sense, such event can be the effect of a source component SM as well as a sub-component SM being in a final state — both cases are commonly referred to as downstream SMs in the state automata hierarchy of Figure 37. In the case of a source connected component SM that is in an affecting final state (i.e., it impacts a component upstream); this typically represents an output deviation from the source component which corresponds to a matching input deviation into the impacted component. In the second case (i.e., a sub-component's SM being in an affecting final state), this typically represents an output deviation from the sub-component which causes the composed component's SM to change its state.

**Algorithm 3** : FaultTreeSynthesiser($\Phi_{syst}$)

**Input:** A set $\Phi_{syst}$ of preliminary system failure TFTs.

**Output:** Each system failure fault tree is synthesised. A fault derived from failures in a component which is not modelled (i.e., incomplete state automata) remains undeveloped.

```
1: for each φ∈Φ_syst do
2:    for each derived fault η in φ do
3:       SynthesiseFaultTree(φ, η)
4:    end for
5: end for
```

The synthesis is done by working backwards, starting with the expressions of the system failures. The input of algorithm 3 is the set of expressions which are generated from a highly abstract system SM (like the top-level SM of Figure 37). The outcome consists of synthesised system fault trees, reached by recursively merging the component fault tree expression which corresponds to a non-atomic event into the higher level expression (line 3) — the *'SynthesiseFaultTree'* method is first invoked to expand the non-atomic events of the preliminary failure expressions (like the events which are immediately below the top event at the right hand side of Figure 37), then it recursively synthesises them into, ultimately, full-scale system-wide TFTs (as described by algorithm 4).

**Algorithm 4** : SynthesiseFaultTree($\varphi$, $\eta$)

**Inputs:** - A failure expression $\varphi$ corresponding to a fault tree.
- A fault $\eta$ which is a non-atomic event derived from failures in some constituent parts or from input deviations.

**Output:** The fault trees that cause $\eta$ are merged into a bigger $\varphi$.

```
1: if (∃i 1≤i≤n s.t. SM_i generates the output failure η) /* otherwise η remains un-
   developed — n is the totality of the models which compose the state automata,
   some components may not be modelled */ then
2:    let MergedTree = false
3:    for each final state s of SM_i s.t. η is the effect of entering s do
4:       let MergedTree = MergedTree ∨ GenerateFailureExpres-
         sion(SM_i, s) /* merge the trees into a disjunction since each one can lead
         to η */
5:    end for
6:    replace η with MergedTree in φ
7:    for each derived fault γ in MergedTree do
8:       SynthesiseFaultTree(φ, γ) /* should other expandable events γ exist in
         the newly merged tree */
9:    end for
10: end if
```

If distinct final states of a component's SM represent the same failure — i.e., an identical output deviation (e.g. omission of output) from that particular component — then the TFTs which correspond to these final states need to be merged, such that each TFT becomes a new branch of

a common bigger tree in a disjunctive form; this is because either branch can lead to that same failure.

Therefore, given a failure expression $\varphi$ and a non-atomic event $\eta$ of $\varphi$, if the *'SynthesiseFaultTree'* method finds the state machine which produces the output failure $\eta$ (line 1 of algorithm 4), then it generates a disjunction of local TFT expressions — one expression for each final state that represents $\eta$ (lines 3 through 5). The newly generated TFTs for each component can then be minimised if appropriate to obtain a simplified intermediate form; this helps to remove any redundancies or complexities as early in the process as possible. Next, these TFTs (combined in a disjunctive form) will be merged into $\varphi$ by substituting $\eta$ (line 6). The newly merged disjunctive tree can contain both symbols which represent basic events (e.g. internal failures of the component from which it was generated) as well as non-atomic events. In the case of newly introduced non-atomic events $\gamma$, the algorithm (lines 7 through 9) recursively synthesises each corresponding TFT into the fault tree represented by $\varphi$ (assuming the state automata contains the SM which produces $\gamma$).

When no more non-atomic events remain in the expression, the system fault tree has been thoroughly synthesised for that particular system failure as all possible substitutions have taken place. At this point, a final analysis takes place to further reduce the cut-sequences (or the cut sets, if it contains no temporal logic) and thus determines the minimal sequences or combinations of events that lead to that system failure.

#### 3.3.4.4 A simple example

To illustrate the presented approach to dynamic analysis, we use the example of a primary-standby (PS) system used in [36]. Figure 38 describes such a system, which is not an uncommon pattern in the safety-critical domain. This example is generic in the sense that components A (i.e. the primary) and B (i.e. the backup) can be any sensing, control or actuating device. S is a monitoring sensor whose role is to activate B upon detection of an output deviation from A (e.g. omission of output). I represents the input to the system – input to each of the two redundant components, and Out is the output of the system. Out must receive input from either A or B for the PS system to function.

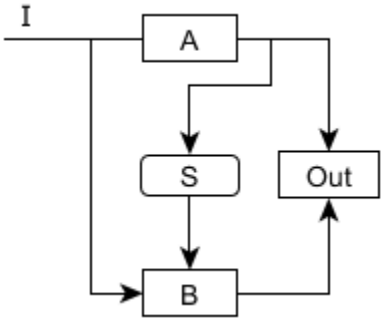



**Figure 38 - a primary standby system**

This example system exhibits dynamic behaviour; for instance it can switch from primary component A to backup B during its operation — i.e., the state of the system moves from state1 = [A: active, B: OFF] to state2 = [A: failed, B: ON]. This behaviour means the system is fault tolerant since a failure of A can be tolerated without loss of system function.

Looking at scenarios of system failure in more detail, we can observe that the supposedly fault tolerant system is not always fault tolerant. For instance, if the system is operating with its primary component but with a failed monitoring sensor, so it is in a state such as state3 = [A: active, S: failed, B: OFF], then a failure of A will immediately cause a total failure of the PS system (B will not be activated since S has already failed). It is clear that the order of failure of A and S is important for the system, since the outcomes are so different when A fails before or after S. This causes

implications for fault tree analysis, a technique unable to capture the ordering of events, and thereby introducing inaccuracies in the analysis results which correspond to dynamics systems in general.

For the Safora method, the input for its TFT generation and synthesis engine is a highly abstract description of the monolithic behaviour of the system. Thus, the PS can be abstracted such that the system fails as a whole if there is an omission of input ('O-I') or if the backup component is unable to take over a failed primary component (and thus omission of output 'O-B' from 'B') as in Figure 39 — assuming that the system is initially working (state ON) and that the 'Out' component in Figure 38 does not fail on its own as it only abstracts the output of the system.

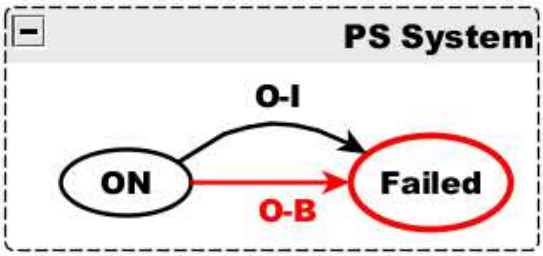



**Figure 39 - Highly abstract SM depicting the monolithic behaviour of the PS, from [27]**

Thus, the generated preliminary fault tree for the example system is 'PS Failed' = 'O-B' OR 'O-I'. Thereafter, for each event that is derived from the component failures, the synthesis method expands the preliminary system TFT with the fault trees of the components. Such non-atomic events are typically represented each by the top event of a component's fault tree.

For example, 'O-B' (which is an output deviation of the component B) is the top event of the fault tree 'O-S' OR ('O-A' AND 'B fails'), as depicted from the SM of B in Figure 40. O-S (resp. O-A) means omission of output from the sensor (resp. component A). B is initially OFF as the PS starts working in primary mode. Then, either omission of output from A is detected which causes a switch to stand-by mode (state ON) or something went seriously wrong downstream in the event causal chain which cannot be detected (state Permanently OFF), e.g., S is unable to activate B upon failure of A (i.e., O-S). B can also fail dormant, but this becomes severe upon a subsequent failure of the primary (or the inverse, i.e. the primary fails first then the backup fails second).

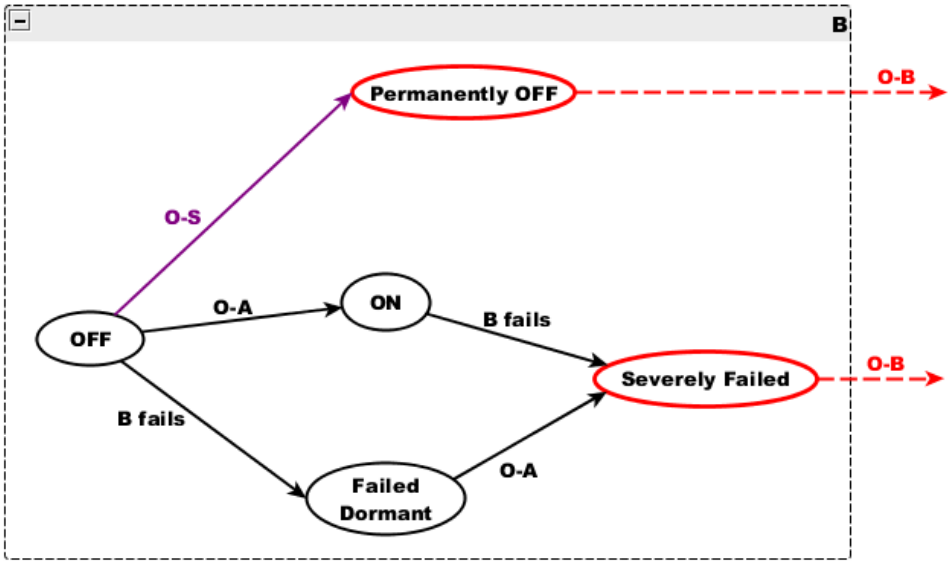



**Figure 40 - SM of the backup B of the PS system, from [27]**

At this stage of the synthesis, expanding ‘PS Failed’ gives ‘O-S’ OR (‘O-A’ AND ‘B fails’) OR ‘O-I’. This will be repeated at each level of the system TFT until no more non-atomic events remain — i.e., the expression of the system failure cannot be expanded any further as all possible substitutions have taken place, and the complete fault tree has been synthesised for that particular system failure (i.e., PS Failed).

Next, ‘O-S’ will be substituted with ‘S fails’ PAND ‘O-A’ — i.e., the TFT expression which is generated from the SM of the sensor S, see Figure 41. S is initially monitoring the primary component. A premature failure of S causes the redundant component B to be irrevocably disabled; as a consequence, the composed system (the PS) relies exclusively on a single component (the primary). In such a situation, the failure state of S may only deteriorate to a severe level should an omission of output from A occur. This wouldn’t happen if the failure of S was not premature, as B can take over A’s job (which is still a safe level). Therefore, the effect O-S from the sensor is restricted to a severe failure of S.

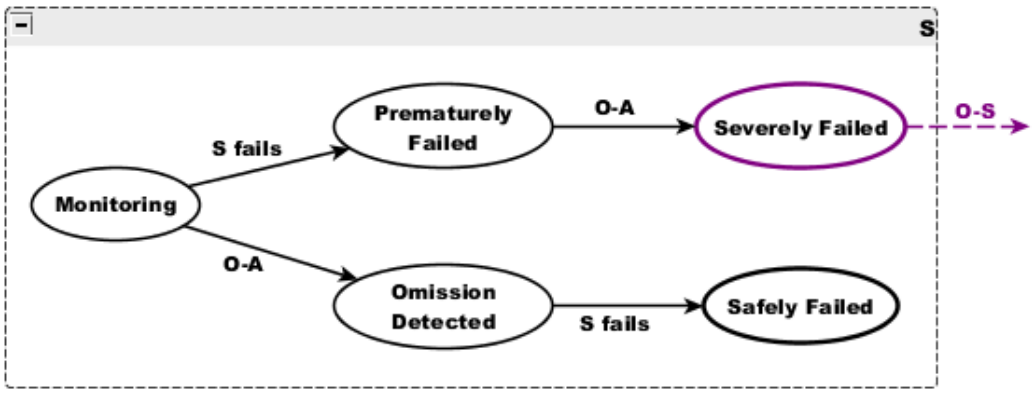



**Figure 41 - SM of the sensor S, from [27]**

Thereafter, ‘O-A’ will be substituted with ‘A fails’ — i.e., the FT expression which is generated from the SM of component A, see Figure 42 — omission of input O-I is considered at system level (in the highly abstract SM) since it is a single point of failure. Thus, upon completion of the synthesis, the extensive system-wide TFT expression of ‘PS Failed’ becomes (‘S fails’ PAND ‘A fails’) OR (‘A fails’ AND ‘B fails’) OR ‘O-I’. The corresponding synthesised system TFT is represented by Figure 43.

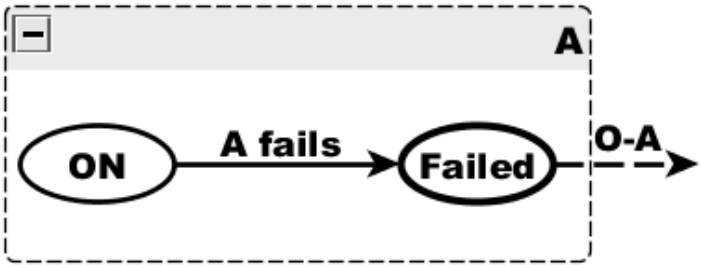



**Figure 42 - SM of A, from [27]**

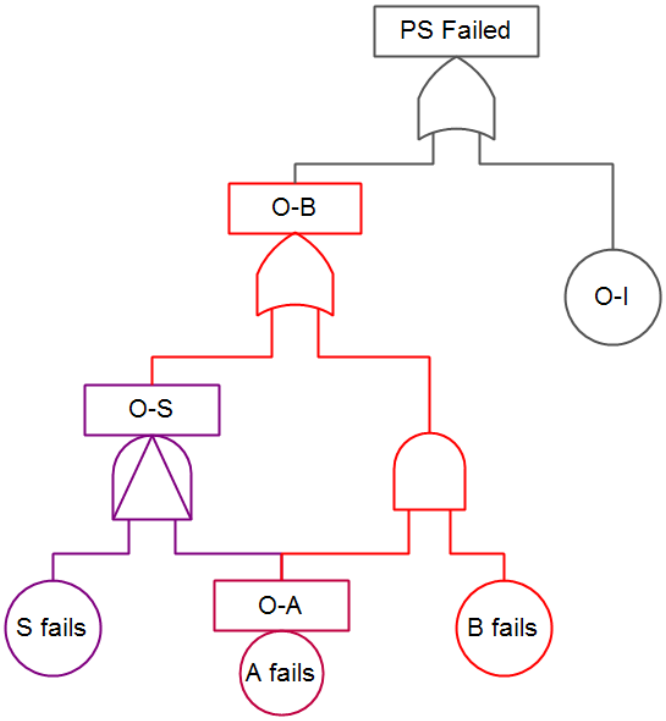



**Figure 43 - Synthesised fault trees, from [27]**

At the end of the synthesis, a final analysis takes place to obtain more reduced cut sequences wherever possible (or minimal cut sets, if the fault tree contains no temporal logic).

#### 3.3.4.5 Summary and remaining work

In this approach, a qualitative analysis which is adequate to dynamic systems can be performed early in the life cycle of such systems as well as late. In that sense, the approach can be applied at stages when probabilistic data are not available as well as at stages when stochastic models with transitions fully labelled with failure rates for the components can be made [27]. The latter means that sequences/combinations of events can also be deduced from the hazards which are represented by the final states of stochastic state machines (e.g. continuous time MCs). These can then be logically reduced to significantly optimise such models such that the quantitative results which correspond to the final states are preserved. Overall, the presented solution can potentially address limitations of earlier work and thus help to improve the safety analysis of increasingly complex modern safety-critical systems.

In the third year of the project, we want to make the Safora method compatible with EAST-ADL. In that sense, the language descriptions of dynamic behaviour (i.e., the EAST-ADL SM error models) can be input to Safora in order to generate and synthesise system TFTs. These TFTs can be analysed during and/or after synthesis using, for instance, Pandora — a recent technique for introducing temporal logic to FTs equipped with a set of temporal laws which allow the significance of the SM temporal semantics to be preserved along the logical analysis, and thereby enabling a true qualitative analysis of a dynamic system. We also aim to potentially make the Safora approach automated as part of the HiP-HOPS synthesis and analysis tool framework.

### 3.3.5 Tool specification and exchange formats

The EAST-ADL Safety Analysis Plugin is a tool providing the means to perform safety analysis on systems modelled using the EAST-ADL language. In EAST-ADL, the user explicitly models the

propagation of errors in an 'error model'. The error model can be created easily because it is built from local error information. For system safety, however, the global effects of the error propagations are of interest. Tools such as HiP-HOPS can calculate the global effects through safety analyses like FMEAs and FTAs by combining the local error information. To provide a view of the global failure effects to EAST-ADL users without additional effort on the users' side, we provide an automated link between EAST-ADL and HiP-HOPS.

### 3.3.6 Tool Integration

To establish the link between EAST-ADL design tools such as Papyrus and safety analysis tools such as HiP-HOPS, they have to be integrated. Integration is described by Wassermann [5] to have five different aspects:

- *control integration*: programs can interoperate;
- *data integration*: programs can use each other's data;
- *presentation integration*: programs have a unified GUI;
- *platform integration*: services provided by platform;
- *process integration*: the software development processes are integrated.

The plugin mainly performs three forms of integration: data, control and presentation integration. All of these forms are realised in the form of a plugin for the Eclipse framework.

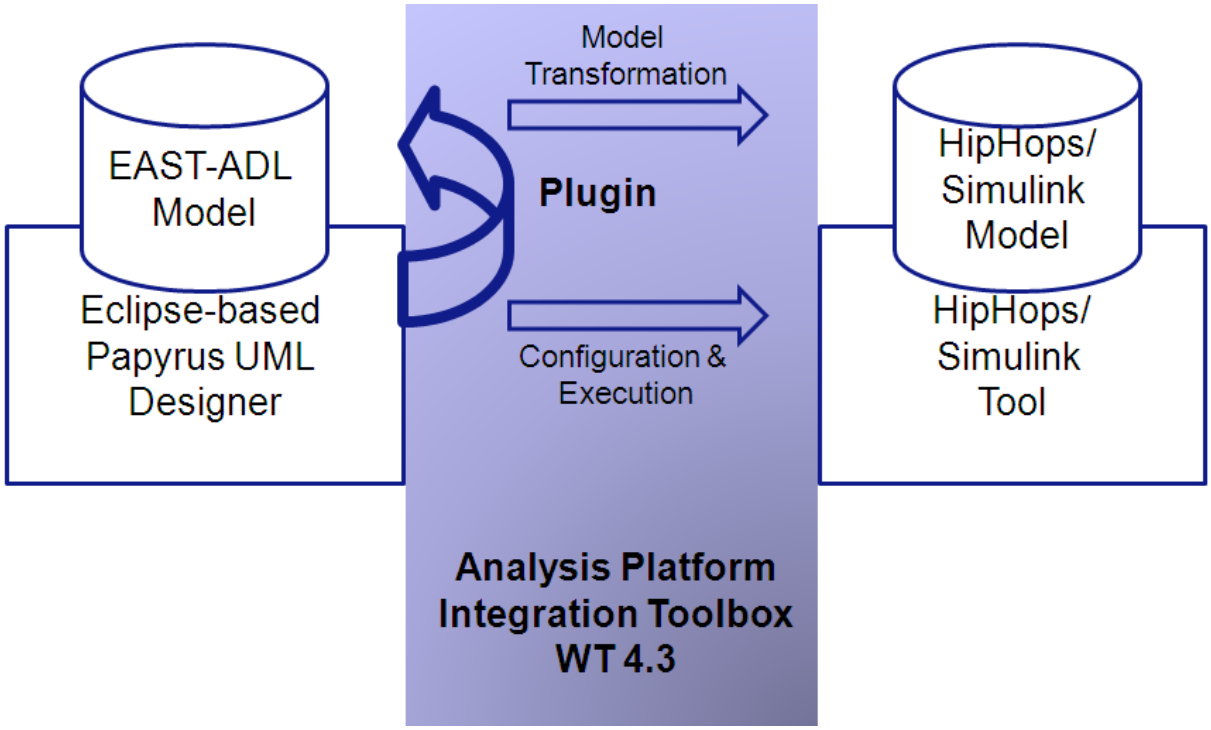



**Figure 44 - Integration between EAST-ADL and HiP-HOPS**

#### 3.3.6.1 Control and Presentation Integration

This link is provided in the form of a plugin. This ensures seamless integration in the modelling environment (Papyrus/Eclipse) and keeps the safety analysis overhead experienced by the user as low as possible and thus allows for an iterative development process.

Without tool support, safety analysis requires tedious, manual work that is frequently seen as an obstacle by engineers, often reducing the scope of this task or limiting it to a single safety analysis. Automation of safety analysis has several advantages: it makes safety analysis easier, it is readily available, and it allows the engineers to obtain a thorough and quick analysis of their design. This rapid feedback based on analysis results allows engineers to perform more micro iterations in the development process, where each iteration refines and improves the previously built model.

#### 3.3.6.2 Data Integration

Data integration in this context is concerned with the transformation of error modelling data. We transform from an EAST-ADL representation to a HiP-HOPS representation, while preserving the semantics. State of the art data integration for model-based development is supported by powerful model transformation engines and languages. Different transformation languages and engines are available, each of them solving a particular problem especially well. Identifying the right model transformation language/engine for the task at hand is a fundamental part of the solution.

#### 3.3.6.3 Transformation Design

The model transformation process from EAST-ADL to HiP-HOPS is partitioned into two steps. Each step has a separate purpose and concern.

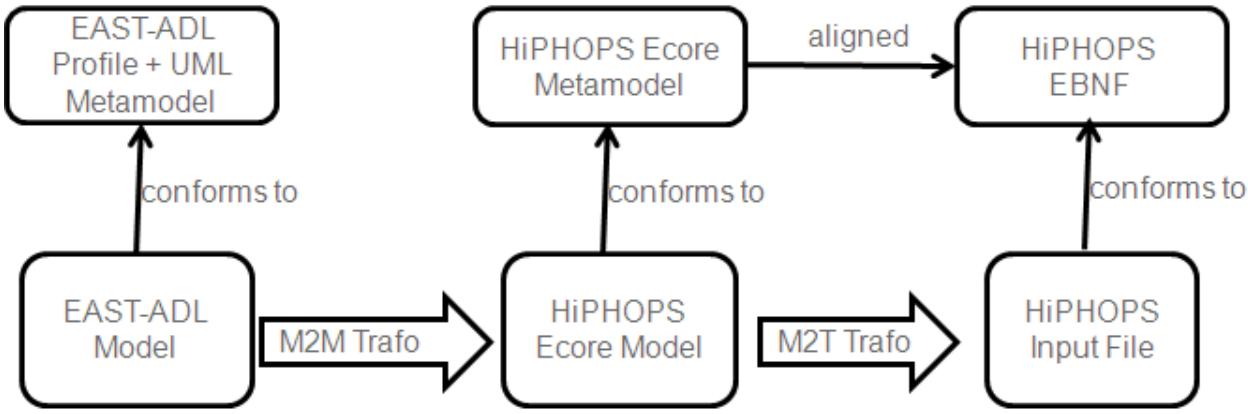



**Figure 45 - Model transformation process**

(1) *Semantic Mapping Transformation*: The first transformation step transforms an EAST-ADL model that is created in the Papyrus UML modelling tool and creates an intermediate model. The structure of the intermediate model resembles the HiP-HOPS grammar, so it is close to the structure of the desired output. This stage performs the semantic mapping between the domains of EAST-ADL and that of HiP-HOPS. However, this stage is not concerned with the actual representation of the data.

(2) *Representation Transformation*: The second transformation step takes the intermediate model and creates the input file for the HiP-HOPS program. This step is mainly concerned with the representation of the information according to the concrete syntax required by HiP-HOPS.

In the following section we discuss the benefits of this solution.

- Our solution separates two different concerns of the transformation from EAST-ADL to HiP-HOPS: (1) the semantic mapping between the domains of EAST-ADL and that of HiP-HOPS and the (2) details of the concrete syntax of the HiP-HOPS input file.
- Each transformation is a separate, self-contained module, which can be developed, changed and tested independently. This decomposition into two separate transformations allows us to parallelise the work on the two transformations and reduce development time. It also allows the two transformations to evolve independently without affecting each other, e.g. a change in the HiP-HOPS grammar will only affect the representation transformation.

As discussed in the section on data integration, different transformation engines have different strengths that can be played out for different concerns. The solution allows us to select the best tool for each concern.

### 3.3.7 Semantic Mapping Transformation

The purpose of the Semantic Mapping Transformation is to map concepts from EAST-ADL to HiP-HOPS in a way that preserves the semantics of the original model. The mapping between EAST concepts and HiP-HOPS concepts is explained in the table below.

| EAST-ADL | HiPHOPS |
|---|---|
| ErrorModelType | System |
| ErrorModelType.errorConnector of type ErrorPropagationLink | System.Lines |
| ErrorModelType.parts of type ErrorModelPrototype | System.Component |
| ErrorModelPrototype.type.errorPort of type ErrorPort | System.Component.Ports |
| ErrorModelPrototype | System.Component.Implementation |
| ErrorModelPrototype.type.errorBehaviorDescription.internalErrorEvent of type ErrorEvent | System.Component.Implementation.FData. basicEvent |
| ErrorModelPrototype.type.genericDescription of type String | System.Component.Implementation.FData. outputDeviation |
| ErrorModelPrototype.type of type ErrorModelType | System.Component.Implementation.System **(recursion)** |

**Figure 46 - Mapping of concepts from EAST-ADL to HiP-HOPS**

Model to model transformations are well suited for a semantic mapping transformation. Both input and output of a model to model transformation are models themselves. Mapping patterns can be described by relational and declarative transformation languages in a concise manner. We chose the ATLAS Transformation Language (ATL), a language that allows a choice of relational and imperative constructs. It furthermore allows processing of models that have a profiled metamodel, i.e., a metamodel that consists of a metamodel and a profile description. In the case of EAST-ADL the metamodel consists of the UML metamodel and the EAST-ADL profile.

### 3.3.8 Representation Transformation

The purpose of the Representation Transformation is the generation of a textual description based on the intermediate model. The intermediate model is designed to have structure which is aligned to HiP-HOPS. No structural changes are required in this transformation. The focus is on serialising the model as text.

Textual representations can be generated particularly well with model to text transformation languages. We chose the Xpand language from OpenArchitectureWare. Xpand is a template-based model transformation language, which incorporates the output in the form of templates into the control structure. The intermediate model is explored using a depth-first strategy.

EAST-ADL models created in Papyrus have a metamodel that is a composition of several separate metamodels. In the case of EAST-ADL, the metamodel consists of the UML metamodel and the EAST-ADL profile. These artefacts are composed by the Eclipse Framework to an EAST-ADL metamodel at runtime. The EAST-ADL metamodel corresponding to a Papyrus EAST-ADL model is not a separate artefact, and this complicates the model transformation and limits the choice of model transformation engines.

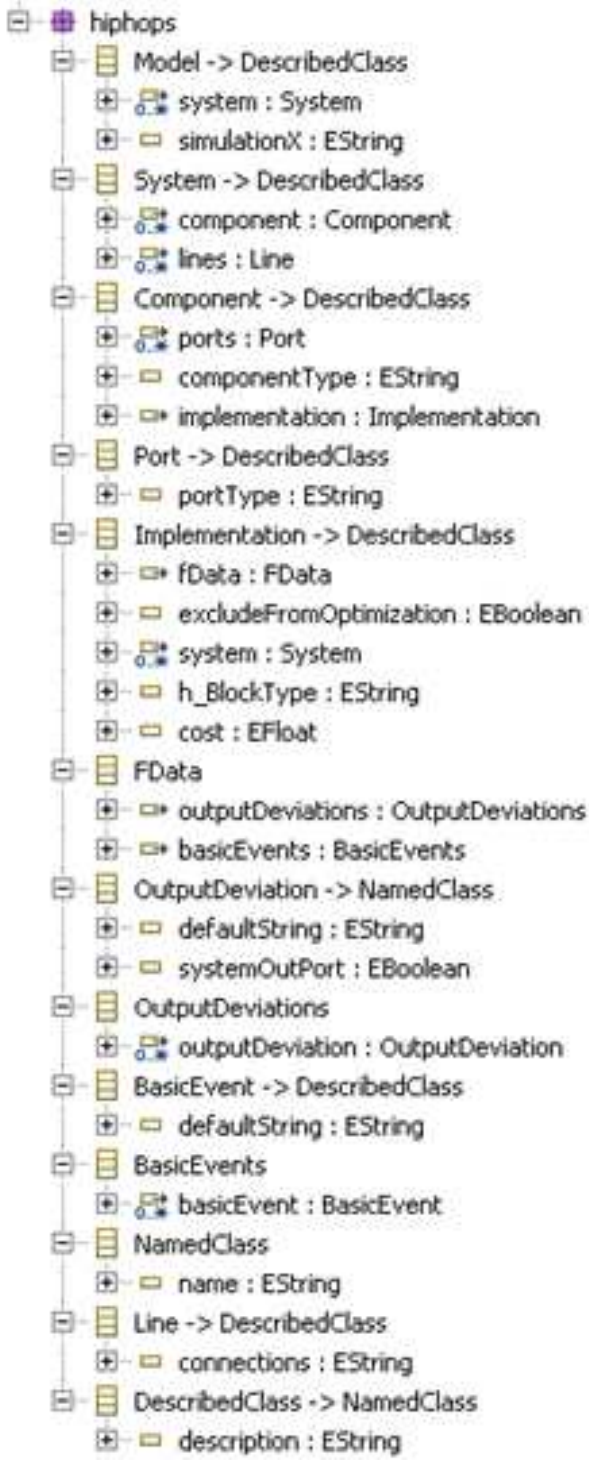



**Figure 47 - Intermediate HiP-HOPS model**

The intermediate model conforms to the HiP-HOPS Ecore metamodel. It is aligned to the HiP-HOPS grammar. It also conforms to Ecore, and is thus processable with the Eclipse Modelling Framework (EMF).

## 3.4 Timing Analysis

EAST-ADL provides a rich set of concepts to support timing analysis (coming also from TADL). The term timing analysis, however, encompasses a wide range of analyses applicable to different models (and levels of abstraction) produced during the development process.

High-level analyses, such as the verification of timing properties of the functional behavioural models via model-checking, can be usually employed from the very beginning, as soon as functional behaviours are defined. Conversely, low-level analyses, e.g. schedulability analysis, compute timing properties of functions when conceptually executed on detailed software/hardware resource models. These resource models include information about processors/buses' schedulers and tasks/message configurations and are in general produced at later stages of the development process.

In this chapter, among the wide range of timing analyses, we focus on schedulability analysis, as schedulability analysis is considered a good candidate for analyzing timing properties at the design stage [31].

Timing analysis at this stage aims at evaluating the impact that hardware resources may have on the execution of functions. To give a simple example on the importance of evaluating the resource impact on function execution, let us consider the architecture in Figure 48. Let us suppose that Function F1A and F1B do not have precedence dependencies, so that they can be logically executed in parallel. On the other hand, the two functions are allocated on the same CPU that will execute F1A and F1B sequentially. Now let us consider that Function F2 must produce a value for each cycle of duration T (deadline). In each cycle F2 must consume at least one value produced by F1A and by F1B in the same cycle. It is clear that if the two functions could be executed in parallel and start at the beginning of the cycle, input values for F2 will be available after *max(C1,C2)*, where *C1* and *C2* are estimated worst case execution times. On the other hand, in case of sequential execution, the input values for F2 will be available only after *sum(C1,C2)> max(C1,C2).* These situations must be carefully analysed as the effect of accessing hardware resources may lead to response times violating deadlines.

Schedulability analysis computes response times and checks if deadlines are respected. Nevertheless, traditional schedulability analysis takes as input model a 'task model', i.e. does not handle functions. To apply schedulability analysis, the set of functions must be previously partitioned into a set of tasks and task priorities. Moreover, platform resources must be characterised by the scheduling algorithm arbitrating the access to the resources.

However, EAST-ADL lacks implementation-level concepts such as tasks, task priority and schedulers. Thus the analysis that one can perform at this level is restricted to feasibility assessment regarding, for instance, resource utilisation, weight of remote communications, and synchronisation. This provides an interesting insight into how the software implementation could later be defined. To go beyond this, one needs to go to the implementation level (i.e. Autosar architecture) where the allocation of execution to tasks is defined. For this, MARTE provides a good basis. In the experiments done on timing analysis in ATESST2, the extra information needed in EAST-ADL models was added using MARTE constructs.

The following section provides a summary of the timing modelling features from EAST-ADL (TADL) and MARTE enabling schedulability analysis. Then the gap between EAST-ADL and MARTE for schedulability is discussed, and design-level schedulability analysis is introduced. The last section discusses tooling support for (design- and implementation-level) schedulability analysis.

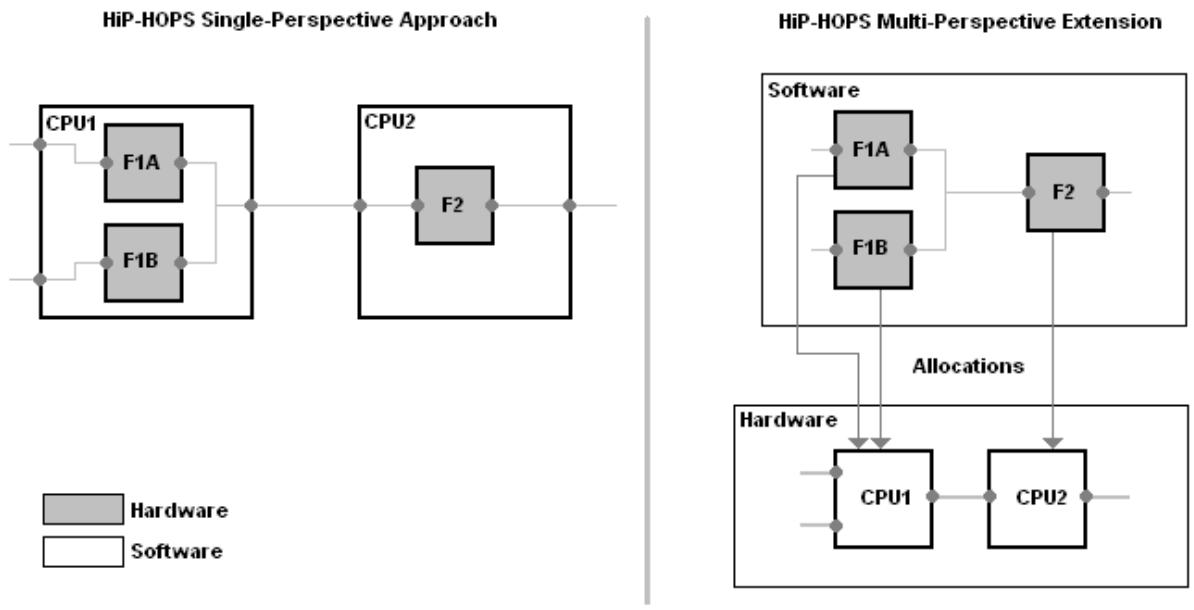



**Figure 48 - Design-level architecture example**

### 3.4.1 EAST-ADL/TADL for schedulability analysis

Timing modelling in EAST-ADL results from the work done in TIMMO project, which produced a dedicated language called TADL (e.g. see TIMMO deliverable D6, available from http://www.timmo.org/) and from Timmo2Use, which produced a second version of the language called TADL2. TADL concepts were integrated in the course of the ATESST2 project in the EAST-ADL language. TADL2 concepts will be integrated in EAST-ADL during the third year of the MAENAD project, but in the current language version (2.1.10) TADL2 has not been integrated yet. For this reason we will refer to TADL in the reminder of the section.

EAST-ADL divides timing information into timing requirements and timing properties, where the actual timing properties of a solution must satisfy the specified timing requirements. EAST-ADL currently focuses on modelling of timing requirements on the functional abstraction levels of the architecture description language. The implementation level, i.e. AUTOSAR, is currently not explicitly considered, but it is expected that the information can be modelled in a similar way. The same holds for timing properties on both the functional abstraction levels and the implementation level.

Timing information on the functional abstraction levels is perceived as follows: timing requirements for a function can be captured on logical abstraction levels where no concrete hardware is yet available. This allows the specification of general timing requirements such as end-to-end delays from sensors to actuators regardless of how the final solution is built. This reflects the notion that a purely logical functional specification is not concerned with its technical realisation, i.e. how many ECUs or bus systems are ultimately involved. What matters from the functional perspective are the recurring end-to-end delays of a control application, which need to keep pace with the real plant. Specifying timing requirements on the implementation level might be both too late in the development process and rather difficult because of language complexity (e.g. AUTOSAR) and the number of details on this level.

Timing properties are characteristics of a solution, e.g. actual response times, and should be reflected in the functional abstraction levels.

In EAST-ADL, timing requirements are divided into various kinds of delays (or latencies) for single time-consuming modelling entities as well as specific requirements for temporal synchronisation of input or output data. The delays are either end-to-end delays, which are subject to segmentation

along the functional decomposition track (e.g. end-to-end delay for a top-level function), or the delays form part of an end-to-end timing chain, and thus constitute segments of such an end-to-end timing chain. Furthermore, delays can be sub-classified as being induced either by data transformation performed by a "FunctionPrototype" or data transmission via a "FunctionConnector".

More precisely, EAST-ADL Timing concepts are based on

- Events: relate to EAST-ADL and AUTOSAR structural entities and depict observables : e.g. data arriving on a port, triggering of function execution, etc.
- Event Chains: bind together events to establish sequences/relations between events e.g. to capture a complete end-to-end flow requirement between data sent by a sensor to output by an actuator.
- Constraints: put temporal constraints on sets of events or on event chains, e.g. deadlines to be met, expected delays, patterns of data arrival, synchronisation of outputs on a set of ports, etc.

One example is featured in Figure 49.

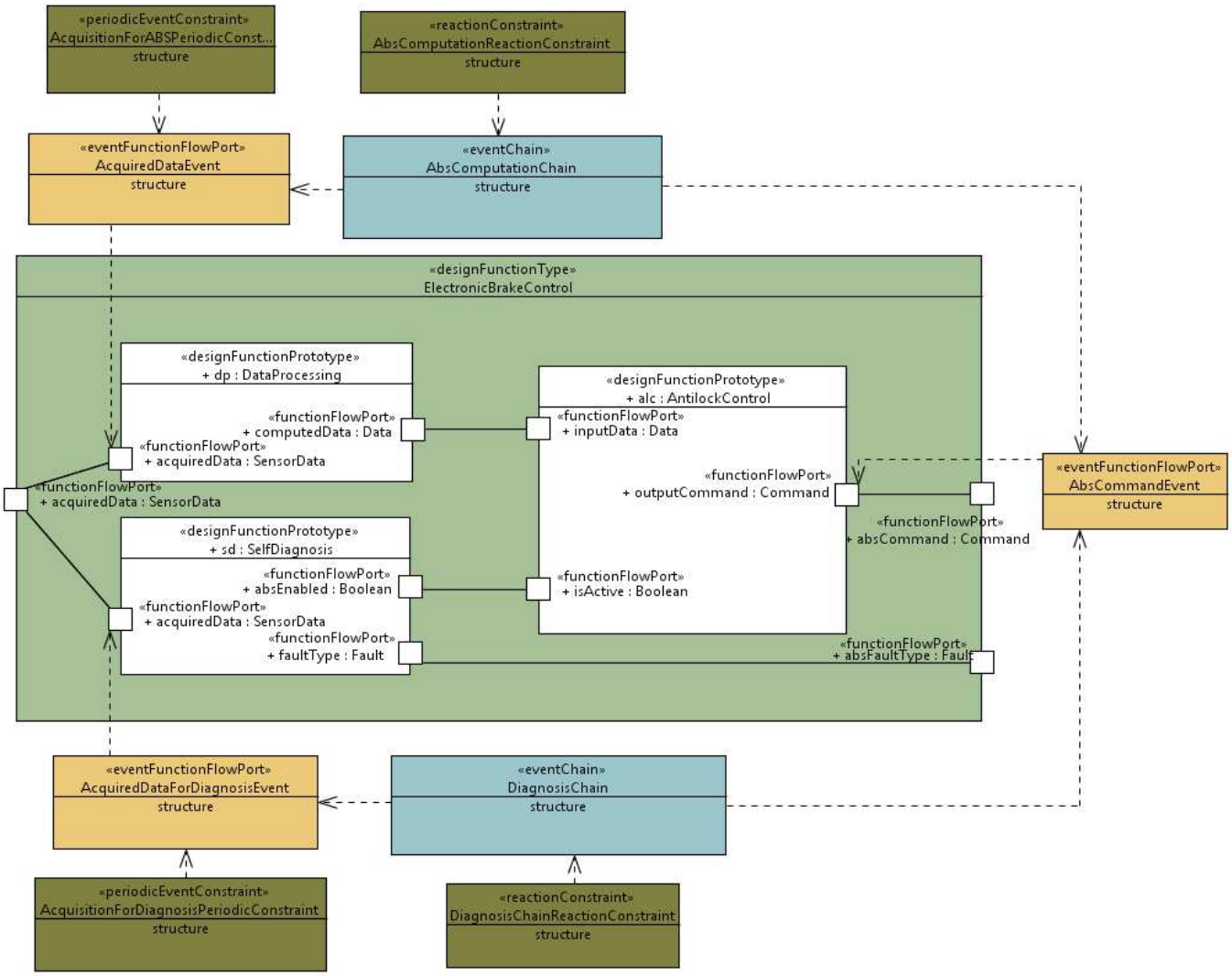



Figure 49 - Timing concepts

Analysis on the basis of analysis-level timing concepts concerns consistency among timing constraints, for example ensuring that the response constraints of parts do not exceed the response constraint of their composition, or that synchronisation constraints on a set of events are

not tighter than the variation among the response constraints that result in those events. Control performance based on specified timing constraints is also a typical concern on this level.

Analysis on the basis of design-level timing concepts can indicate feasibility of a solution where functions of functional design architecture are allocated to nodes of hardware design architecture. Nodes have a processing speed and functions have an execution time and information about triggering. It is also possible to assign link speeds to take into account communication delays based on the data size of information exchange. In cases where communication buses are not unambiguous, explicit allocation of function connectors to communication buses is possible.

### 3.4.2 MARTE for Schedulability Analysis

The domain model for non-functional analysis in MARTE is organized around the notion of Analysis Context (see Figure 50). An analysis context is the root concept used to collect relevant quantitative information for performing a specific analysis scenario. Starting with the analysis context and its elements, a tool can follow the links of the model to extract the information that it needs to perform the model analysis.

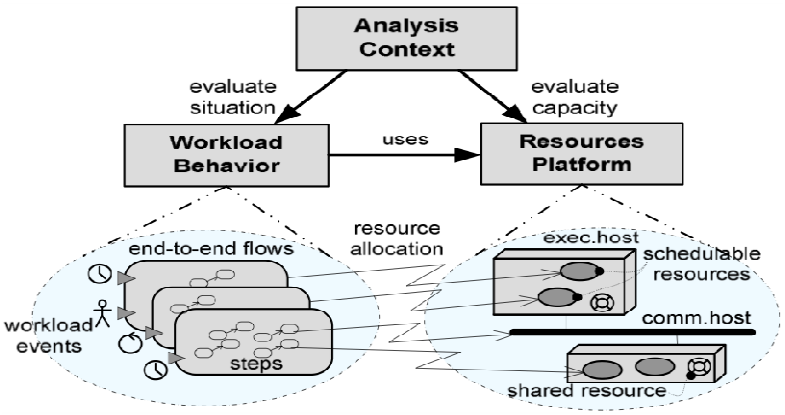



**Figure 50 - Organisation of analysis-specific model elements in MARTE**

Within the application model, “workload behaviour” describes how often events will arrive and how much data are provided as inputs to the system, and how response events are generated in return by the system. In addition, workload behaviour models provide information about the processing behaviour that is used to execute the various tasks.

On the other hand, “resource platform” models provide information about the properties of the computing and communication resources that are available within a system, such as processor speed and communication bus bandwidth. The system model thereby captures information about the applications and the available resource platform of the system, and it also defines the mapping of application tasks to computation or communication resources.

The main analysis technique in the scope of MARTE is scheduling-aware timing analysis. This kind of technique offers a mathematically-sound way to determine whether a system will meet its timing requirements and how it could be balanced (e.g., task allocation, priority assignment, component deployment) while still respecting timeliness. To this purpose, scheduling analysis tools are a key component in this chain. Implementation of scheduling analysis tools depends a lot on the kind of scheduling analysis supported. In particular, the domain of scheduling analysis for automotive applications has received special attention in recent real-time systems literature. A comprehensive summary of analysis techniques for automotive applications is provided in [17]. Among relevant advances in this field, the holistic approach [18] extends the classical single-processor scheduling theory and applies it for specific combinations of input event models, resource sharing and communication policies. The global view on the system allows taking global

dependencies into account (offsets between the activation instants of tasks), thus providing tightly calculated analysis bounds. A different approach for distributed architectures is the compositional scheduling analysis [19,20]. The basic idea of this approach is to break down the analysis calculation into separate local component analyses (e.g., mono-processor analysis with RMA) and to integrate them for system-level analysis by evaluating the propagation of event stream models.

In our experiment in ATESST2 we worked with the MAST tool from the University of Cantabria, which supports classical RMA analysis and holistic analysis [21,22].

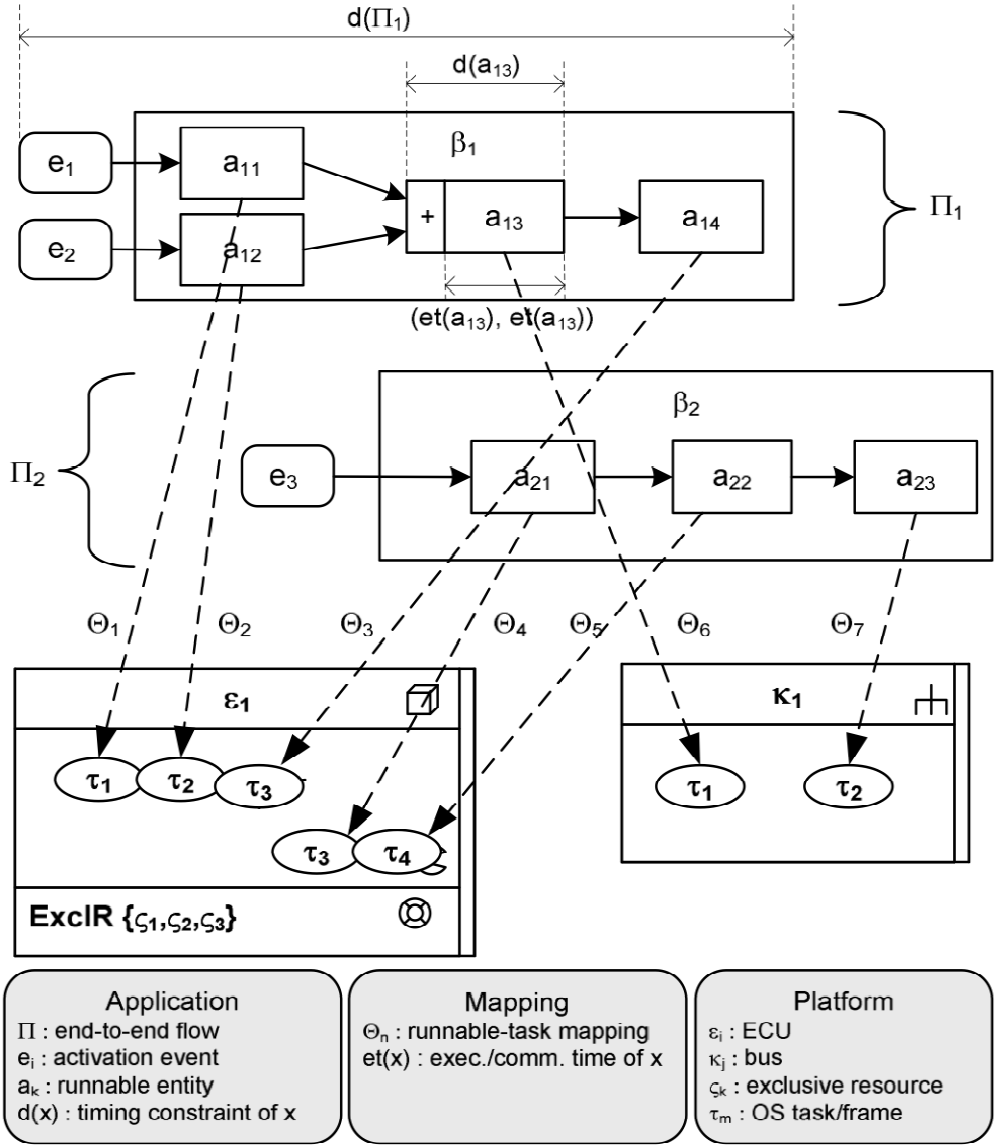



**Figure 51: Simplified canonical model for scheduling analysis**

Figure 51 shows a simplified canonical model of the modelling features required for scheduling analysis, which is discussed in this section.

*Timing constraints*

We can distinguish at least three abstraction layers of time constructs:

- Time-related abstraction layer, level 1: time is modelled as a set of fundamental notions such as time instants, durations, time bases, or clocks. They can be specified by relative/absolute durations (maximum time intervals) or instants (occurrence of a timeout event).

- Time-related abstraction layer, level 2: MARTE provides mechanisms to annotate timing requirements and constraints in models. Basic timing constraints include deadlines and maximum jitters. One key-modelling feature is the concept of observation. Observations provide marking points in models to specify real-time assertions. Some typical assertions have been predefined in ready-to-use patterns, such as jitters or conditional time constraints.

- Time-related abstraction layer, level 3: time concepts are defined as part of the behaviour, not mere annotations. This set of constructs cover both physical and logical time. Most of the current scheduling analyses use the notion of physical time as measured by a unique time base. However, distributed applications often experience problems for agreement on consistent time reading due to clock synchronisation. This means that scheduling them depends on different time bases, and therefore constraints must refer to specific clocks.

*End-to-end flows*

In MARTE, end-to-end flows describe logical units of processing work in the system, which contend for the use of processing resources (e.g., processors and buses). It should be noted that firstly data and control can be part of the processing, and secondly, different kind of timing constraints can be attached to end-to- end flows (e.g., deadlines or output jitters).

One important feature in MARTE is that end-to-end flows can be represented in behavioural views (e.g., Sequence or Activity diagrams) complementing component models. This approach allows modellers to specify multiple end-to-end flow configurations that could be likely related to (a) specific operational modes, (b) alternative execution chains, or (c) different quantitative scenarios of activation parameters or other non-functional annotations.

*Activation events*

Both event-triggered and time-triggered paradigms are often involved in automotive applications. Event-triggered means that tasks are started, or messages are transmitted, following the occurrence of one (or a conjunction of) significant event (e.g., "a door has been opened"). Time-triggered consists of tasks started, or messages transmitted, at predetermined points in time, usually periodically.

In MARTE, activation models are denoted by means of workload events. Workload events can be modelled under different forms: by known patterns (e.g., periodic, aperiodic, sporadic or burst), by irregular time intervals, by trace files, or by workload generator models (e.g., state machine models). Workload events also enable to specify additional parameters for periodic and aperiodic patterns such as jitters, burst parameters, and distribution probabilities.

*SW and HW resources*

What is needed for scheduling analyses is to take into account the impact of the OS and hardware resources on applications (e.g., overheads due to the OS and the stack of communication layers or throughputs and bandwidths of underlying networks). Among these aspects, access protocols to mutual exclusive resources are of paramount importance in scheduling analysis of modern multiprocessor architectures.

The MARTE analysis model distinguishes two kinds of processing resources: *execution hosts*, which include for example processors, coprocessors and controllers, and *communication hosts*, which include networks and buses.

Processing resources can be characterized by throughput properties as for example processing rate, efficiency properties such as utilisation, and overhead properties such as blocking times and clock overheads.

### 3.4.3 The 'analysis gap' between EAST-ADL/TADL and MARTE for schedulability

Both languages are complementary. In general MARTE SAM is more focused on a behavioural description of a system in which timing information is a first-class input, whereas EAST-ADL assumes a fixed execution semantics for the functions, resulting in a more structured and static nature of models. Both approaches rely on end-to-end flow descriptions. The main difference is the level of detail in which both languages regard allocation. On this point MARTE SAM requires more: an initial mapping from functions to tasks is needed, which is explained by the fact that MARTE considers the whole design life-cycle, where EAST-ADL delegates to AUTOSAR the details of the implementation levels.

Let us remark that standard schedulability algorithms can compute response times, only if assumptions regarding function-to-task, data-to-frame, priorities, etc, exist. These assumptions can be made in the case of mono-processor systems. In mono-processor systems, it is well-known what strategies apply to function-to-task mapping and to priorities [32]. Unfortunately, for distributed systems, this is not the case. The problem of finding a good mapping and scheduling is NP-hard and no satisfactory solutions can be found in the literature. Some attempts can be found in [33,34], but these papers do not deal with the function-to-task mapping in distributed architectures. A first attempt to solve at the same time the mapping of functions to tasks, tasks placement and tasks scheduling can be found in [35]. In this paper, an optimisation technique is proposed to find a solution for placement, mapping and scheduling. The technique is based on two consecutive steps. In the first step, an allocation of functions/signals on execution nodes/communication buses is found. In the second step, the allocation found in the first step is taken as input, and then priorities are assigned to functions/signals in order to group them in tasks/messages according to the assigned priorities. As in [33], the proposed technique is based on Mixed Integer Linear Programming (MILP). The proposal considers timing and platform resources constraints. As for metrics, we consider a multi-objective optimisation which includes the minimisation of latency (response time along an end-to-end flow made of tasks) and maximisation of tasks extensibility (maximum possible increase of execution time without exceeding node capacity). For such technique, we introduce specific intermediate metrics for the first step. These intermediate objectives are detailed below:

- Function extensibility: it represents the maximum possible increase in execution time of functions while utilisation capacity of the execution nodes is not exceeded. The maximisation of this objective guides the solution to task extensibility maximisation.
- Inter-node communication: it consists of minimising communication over the buses. The minimisation of this objective favours a model optimised for latency, as remote communications are more costly than local ones.
- Interference: interference of a function represents the waiting time to access to the CPU. This delay is caused by concurrent functions that belong to the same execution node. Two functions are concurrent if their executions are independent. The minimisation of interference means increasing the parallelism, thus minimizing latency.

It should be noted that these intermediate metrics apply directly to the EAST-ADL functional design level without any further refinements. In the meantime, they represent precious metrics that have a direct impact to task-level response time. These metrics can be used in the case of distributed systems to evaluate different allocations and are implemented in the Timing Analysis Tool (see next section).

### 3.4.4 Timing analysis tooling

A first version of a timing analysis gateway was developed in the context of the EDONA project http://www.edona.fr by CEA LIST. It consisted of a set of plugins which took a MARTE-annotated

model of a system and enable the modeller to run various schedulability analyses provided by a timing analysis tool. The plugins made a bridge to two specific tools:

MAST http://mast.unican.es/

RT-Druid http://www.evidence.eu.com/content/view/28/51/.

The model taken as input was a MARTE model, corresponding to an EAST-ADL design architecture model, enriched with schedulability analysis parameters provided by MARTE annotations; these combine to produce a view of the system suitable for task scheduling and allocation to hardware. Based on this, the plugin transformed the model into the specific analysis tool format, provided an interface to the tool and returned analysis results, store in the MARTE models. A typical scenario allowed for the testing of whether a task allocation scheme allows for a system to be schedulable given particular hardware performance, or provide parameter values for task priority assignment.

The first version of the timing analysis gateway had several limitations listed below:

1. No automatic transformation from EAST-ADL models was provided, the corresponding MARTE counter-part needed manual construction.
2. The MARTE annotations and function-task mapping needed manual construction. This was a quite long and error-prone activity, requiring MARTE knowledge
3. Function-to-task mapping was required, but no support to guide the designer in the mapping was provided. As already highlighted in the previous section this mapping is far from being trivial and determines response times from schedulability analysis.
4. External engines made internal transformations transparent to the users that modified the function-to-task mapping, without any apparent reason. This made difficult to interpret results.

To overcome these limitations, a new Timing Analysis plug-in, called Qompass, has been developed since. To address point 4, this plug-in now features its own schedulability analysis algorithm, which no longer needs a connection to an external engine.

To address point 1, a transformation has been developed in order to import EAST-ADL profiled models in Qompass. The transformation allows translating EAST-ADL event chains in MARTE end-to-end flows. Hardware topology information and allocation is also translated in corresponding MARTE models.

To address point 2, a number of graphical interfaces have been added. Thanks to GUIs, the user can easily type schedulability-related information. This information is automatically stored in MARTE annotations.

For point 3, Qompass timing analysis provides now design-level and implementation-level analysis. In the special case of mono-processor systems, an automatic function-task mapping is provided and RMA analysis is performed to get end-to-end latencies.

For design-level, so-called "early stage timing analysis" has been added as well. This analysis allows evaluating the functional architecture and more in particular its allocation to hardware. The metrics presented in the previous sections have been selected for evaluation, namely:

- Function extensibility: it represents the maximum possible increase in execution of functions while maximal utilisation capacity of the execution nodes/buses is not exceeded. Large extensibility is desirable.
- Inter-node communication: amount of communication over the buses. A small amount of communication is desirable.

- Interference: waiting time to access to the CPU of the execution node when the CPU is busy with a concurrent function (functions belonging to another end-to-end flow). Low interference is desirable.

The possibility of comparing different allocations along these three metrics has been added as well. It should be noted that early stage timing analysis is the main contribution delivered in P2. Early stage timing analysis allows the designer evaluating allocation, from a schedulability point of view, without the necessity of adding implementation-level concepts.

For further information, a first version of the user guide is available and can be found here:

www-maenad.cea.fr/svn/WP5/WT5.2/qompass0.8.2-eastadl2.1.10/QompassUserGuide.doc

### 3.4.5 Timing analysis limitations

Schedulability analysis implemented in Qompass works only under the assumption of linear event chains (as MAST). This limitation makes impossible at the moment to analyze the complex graph of end-to-end flows that is generated after the import of the BBW EAST-ADL model. In future versions the gap between the BBW end-to-end flow graph and a model of linear end-to-end flows (canonical graph) will be evaluated. Algorithms that aim at transforming any graph in a canonical graph will be studied.

## 3.5 Analysis and Synthesis concepts to support Electrical Vehicles

This section describes analysis and synthesis concepts relevant for electrical vehicles and how we intend to support them in MAENAD.

### 3.5.1 Identified Needs

The engineering of an electrical vehicle is similar to that of a conventional vehicle in many ways. The particular needs we want to address in this section concern the analysis and synthesis of electrical power consumption, current analysis, power architecture analysis, cable length analysis, component cost analysis and component count analysis. This needs to be supported both statically and dynamically. Other possible FEV-related analyses include:

- Insulation analysis, to examine overall resistance and voltage compliance
- RESS short circuit analysis
- Charging inlet voltage analysis
- Transient voltage analysis, e.g. for accidental discharge of capacitors into accessible parts of the EV
- Range analysis, to determine maximum range of the EV given the battery capacity and electricity consumption of auxiliary components
- Power analysis in key-off mode, to prevent excessive discharge due to vehicle equipment operating in standby mode
- Powered braking analysis, to ensure correct braking operation in depleted-power scenarios

Constraints can be linked to EAST-ADL and AUTOSAR elements. The *mode* element defines where the constraint is applicable, and it is possible to analyse the total set of constraint in each mode. To find an average or total measure, the result can be considered in steady state where the

modes change according to the behavioural definition. Below we will discuss what this means for each type of constraint studied.

### 3.5.2 Component count and cost analysis

The number of components produced and their combined cost is a critical parameter for the assessment of a solution, e.g. during optimisation or manual exploration of alternatives. The total cost is defined as the development cost for each component type plus the sum of piece cost multiplied by piece count summed over all component types:

*Total Cost = Sum of Development Costs + (Piece Cost x Piece Count) for all components*

To know the piece count, it is necessary to define the absolute or relative number of elements in the feature tree or in the artefact model. A constraint solver can compute the number of elements of any given component based on such constraints, and it can also warn if the model is underspecified or inconsistent.

In the example in Figure 53, 20000 vehicles are produced. Since 16% of all vehicles go to the US market, and 50% of those have trim level Prime, it is possible to deduce that 1600 such vehicles with will be produced. 10% of all vehicles will be Plus, so these represent another 2000 vehicles. Both trim levels have ABSPlus which means 3800 PlusECUs will be produced and thus 16200 Standard ECUs.

Having established the number of components produced, it is possible to take this into account during optimisation: solutions with good price and performance for high volume products are favoured before solutions that only benefit low volume products.

For example, if a high-volume vehicle needs an advanced component, while the low volume vehicle can do with a simpler component, it may well be better to equip all vehicles with the advanced component. This may also open up for after-sales revenue by selling upgraded functionality that relies on the better ECU. Figure 52 shows that the eliminated fixed cost for the low-end ECU compensates for the higher piece cost. This is consistent with Figure 53, as there is no configuration decision that states whether standard or plusECU shall be used. In the takerate model, it is undefined whether standardECU or PlusECU is used, unless the ABSPlus is selected (in which case, PlusECU is mandatory).

50*3800+300000 + 45*16200+300000=1519000

50*3800+300000 + 50*16200+0=1300000

**Figure 52 - Total Product Line cost for same and different ECU**

The assumption is that this is the stated rate; the constraints are part of the same GenericConstraintSet and thus intended to be consistent and simultaneously applicable. Further, the semantic assumption is that the root element of each product feature tree represents all vehicles.

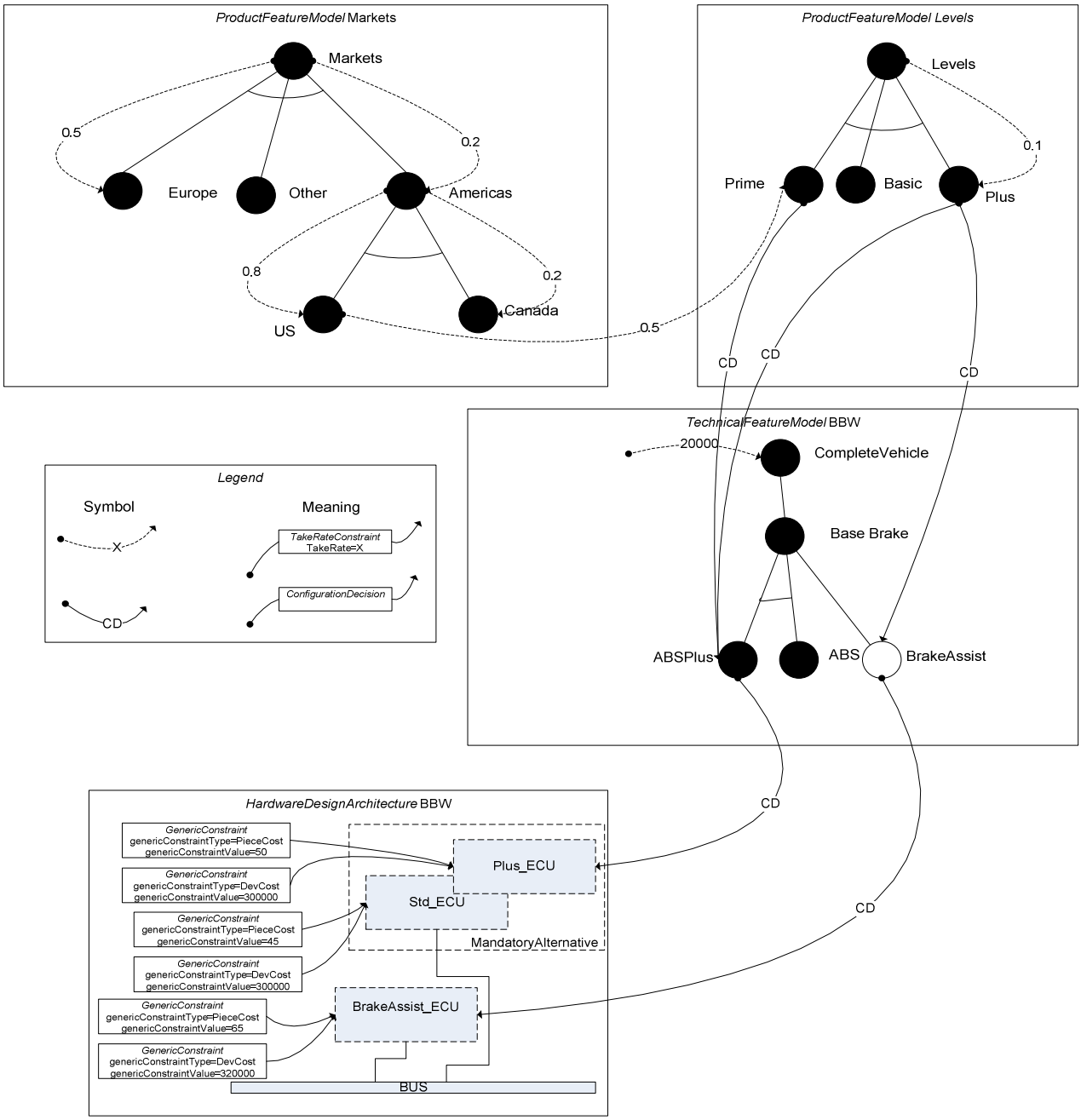



**Figure 53 - Feature models and constraints to consider in optimisation**

One way to resolve the take rate constraints is to translate it to a constraint programming problem and use a suitable solver. Figure 54 shows how a constraint programme in Prolog may look. It also contains a minimisation criteria for cost.

```
% Prolog CLP(FD) pseudo-code

:- use_module(library(clpfd)).

% The feature tree as constraints

feature_tree(TotVehicles,Cost,Variables):-
        % All variables should be greater than zero
        TotVehicles#>=0,
        VehiclesOther#>=0,
        VehiclesEurope#>=0,
        VehiclesAmerica#>=0,
        VehiclesUS#>=0,
        VehiclesCanada#>=0,
        Prime#>=0,
        Basic#>=0,
        Plus#>=0,
        BaseBrake#>=0,
        ABSPlus#>=0,
        ABS#>=0,
        BrakeAssist#>=0,
        PlusECU#>=0,
        StdECU#>=0,
        BrakeAssistECU#>=0,
        PlusECUYesNo in 0..1,
        StdECUYesNo in 0..1,
        BrakeAssistECUYesNo in 0..1,
        Cost#>=0,

        % Constraints from Markets
        VehiclesEurope#=TotVehicles*0.5,
        VehiclesAmericas#=TotVehicles*0.2,
        TotVehicles#=VehiclesEurope+VehiclesAmericas+VehiclesOther,

        VehiclesUS#=VehiclesAmerica*0.8,
        VehiclesCanada#=VehiclesAmerica*0.2,
        VehiclesAmerica#=VehiclesUS+VehiclesCanada, % redundant constraint

        % Constraints from Levels
        TotVehicles#=Prime+Basic+Plus,
        Prime#=0.5*VehiclesUS,
        Plus#=0.1*TotVehicles,

        % Constraints from BBW feature tree
        BaseBrake#=TotVehicles,
        BaseBrake#=ABSPlus+ABS,
        ABSPlus#=Prime+Plus,
        BrakeAssist#=Plus,

        %Constraints from BBW architecture design
        StdECU#<=ABS
        PlusECU#>=ABSPlus
        PlusECU#<=ABSPlus+ABS
        StdECU#+PlusECU#=ABS+ABSPLus
        BrakeAssisECU#=BrakeAssist,

        % Objective function (single objective)
        PlusECU#>0 #<=> PlusECUYesNo,
        StdECU#>0 #<=> StdECUYesNo,
        BrakeAssistECU#>0 #<=> BrakeAssistECUYesNo,

        Cost#=PlusECUYesNo*300000+PlusECU*50+StdECUYesNo*300000+StdECU*45+BrakeAssistECUYesNo*320000+Brak
eAssistECU*65,

        Variables=[VehiclesEurope,VehiclesAmerica,VehiclesUS,VehiclesCanada,Prime,Basic,Plus,BaseBrake,ABSPlus,ABS,Bra
keAssist,PlusECU,StdECU,BrakeAssistECU].

% Find minimum cost solution for given number of vehicles

find_cost(Cost,Variables):-
        feature_tree(20000,Cost,Variables),
        labeling([minimise(Cost)],Variables).

% Find maximum number of vehicles for maximum given cost

find_vehicles(TotVehicles,Variables):-
        Cost#=<1600000,
        feature_tree(TotVehicles,Cost,Variables),
        labeling([maximise(TotVehicles)],Variables).
```

**Figure 54 - Prolog code corresponding to the TakeRate constraints**

### 3.5.3 Cable Length

Cable Length is a static measure and is established by considering all cableLength constraints allocated to elements of the HardwareDesignArchitecture, typically the HWconnectors. In case such fine granularity is not needed it is also possible to define cable lengths for components or component compositions that use a set of cables. In such case the cable length is specified without defining which cables it refers to.

To find the total cable length, the *genericConstraintValue* over all *GenericConstraint*s with the *genericConstraintType* cableLength is added up.

If there is variability, all variation groups of the architecture need to be resolved first.

If an entire product line is considered, it is necessary to either sum the cable lengths over each fully resolved architecture and iterate over all fully resolved vehicles while summing up the total cable lengths, or calculate component counts over the entire product line (see TakeRate constraint) so that the component's cable length is multiplied with the component count and summed.

### 3.5.4 Power Consumption

The fidelity in modelling power consumption may vary. The average power regardless of mode could be represented (1), or the different power levels in each mode (2), or a varying power within each mode (3). Establishing the total power or energy consumption over an interval of time would then be treated differently in each case.

Alternative 1) would mean that the *genericConstraintValue* over all *GenericConstraints* with *genericConstraintType* powerConsumption is added up.

Alternative 2) would mean that each unique mode assignment for the integrated architecture is traversed, and the *genericConstraintValue* over all *GenericConstraint*s with *genericConstraintType* powerConsumption are multiplied by the time spent in this mode and added.

Alternative 3) would mean that simulation or analytical means are used to establish the instantaneous power in each component. Because power is allowed to vary, the component power must be defined by a function. Typically, power is related to the nominal functional behaviour of the component. The *genericConstraintValue* for a powerConstraint could thus be an expression with nominal signals as operands. This is however not supported by EAST-ADL.

The instant view of the power consumption can be used for integrating to energy consumption, averaging power levels for certain intervals, finding peak values, etc.

### 3.5.5 Current analysis

Current analysis is useful to establish cable and connector dimensions. It can be assumed to be the same procedure as for power, only that the results are divided by voltage.

### 3.5.6 Power Distribution Analysis

The power distribution is a critical service in an all-electrical vehicle. An analysis of how each hardware component and each SW component, function or feature relies on the power distribution network is thus important. This analysis can be done through the error modelling and error propagation analysis described elsewhere.

Analysis can also be conducted by considering the nominal architecture: each feature is realised by functions allocated to nodes in the HardwareDesignArchitecture. Nodes have HardwarePins that may be used for power. There is a PowerSupply component.

Power distribution is typically organised as independent power networks. Using the elements above, it is possible to analyse and visualise:

- which nodes, sensors and actuators depend on which PowerSupply. In case multiple power supplies are connected to a component, it is assumed that one is sufficient for operation (OR-logic);
- which functions depend on which PowerSupply (based on allocation to nodes, sensors and actuators);
- which features depend on which PowerSupply (based on realisation by functions) In case realising functions relies on different power supplies, it is assumed that all of these are required (AND logic).

The distinction between OR and AND dependency on the power networks can thus be established and visualised.

## 3.6 Application of analysis techniques to the E/E lifecycle

The various analysis algorithms and techniques described in this section are expected to be used in concert, although they can also be used in isolation as required. To provide a context, the EAST-ADL development lifecycle for E/E systems is presented in classic V form below. The left side represents the collection of requirements through to realisation of concrete system design, whereas validation & verification activities are on the right side of the V.

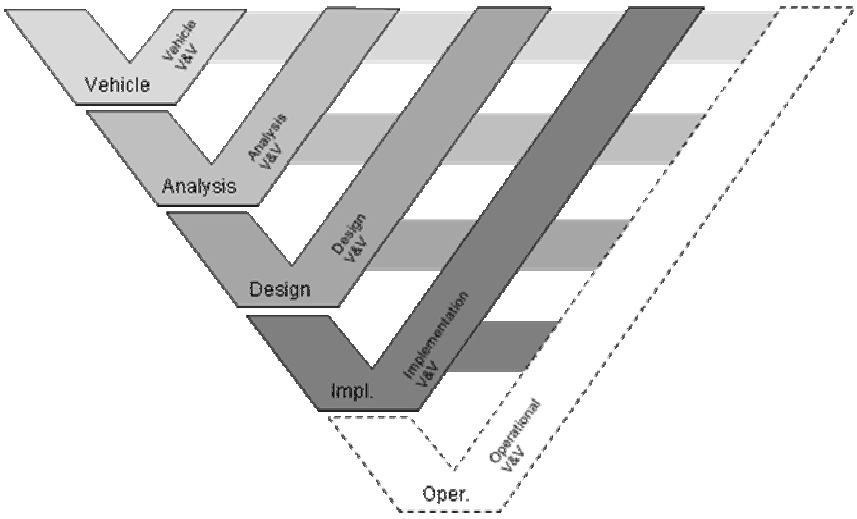



**Figure 55 - EAST-ADL development lifecycle**

The main analysis algorithms discussed are as follows:

- Hazard analysis & risk assessment
- ASIL decomposition & allocation
- Behavioural analysis and simulation
- Dependability analysis (e.g. FTA, FMEA, including temporal FTA from state machines)
- Timing & schedulability analysis
- Non-functional analyses for electric vehicles, including component count, cost analysis, power consumption, current analysis, cable length, and power distribution analysis

The subsections below describe at which stage of the design process each analysis technique is meant to be applied, although in some cases analysis is applied as part of cross-cutting 'swimlanes' that apply to multiple levels.

### 3.6.1 Vehicle Level

The Vehicle Level is the initial stage of the process, involving collection of external requirements for the vehicle and definition of the top-level functions/features the system is intended to possess. This takes the form of a vehicle feature model, which organizes the vehicle features and allows system requirements, use cases, etc. to be linked.

In the context of analysis, it is at this stage that the preliminary hazard analysis and risk assessment process takes place, as described in section 2, in line with the safety lifecycle of ISO 26262. This considers possible scenarios for system failure and helps to identify possible malfunctions and flaws in system features. As per ISO 26262, the hazardous events that these malfunctions may lead to are then used to define the safety goals for the system, together with corresponding safety requirements (including ASILs). This forms the general basis of the functional safety concept.

### 3.6.2 Analysis Level

The Analysis Level is the second stage and involves the definition of a logical definition of abstract system functionality to fulfil the features and requirements described at Vehicle Level. The result of this is the functional analysis architecture or FAA, which describes the functional logic of the system and how they are connected, but avoids the specific issues of software/hardware. It also links to V&V cases and the appropriate requirements defined earlier.

At this stage, the safety requirements and the ASILs defined earlier can be decomposed across the logical system functions. As described in sections 2.4-2.6, this can be supported through the use of techniques to automatically determine possible allocations of ASILs that meet the overall safety requirements without unnecessary expense.

However, before this can take place, a more thorough understanding of how the logical functions contribute to system failure must be obtained. This can be achieved through the use of dependability analysis tools and techniques such as fault tree analysis (FTA), failure modes and effects analysis (FMEA), common cause analysis (CCA), and abstract behavioural simulation techniques; at this stage, analysis is primarily qualitative in nature. A system hazard analysis will also be performed to provide a more concrete definition of the top-level hazards. This is all supported in EAST-ADL by its error model capabilities and compatible tools such as HiP-HOPS.

The outcome of this overall dependability assessment is a clearer understanding of potential function failures and how their effects propagate throughout the system, and therefore of whether or not the system is capable of meeting its safety requirements. If not, the design can be modified appropriately.

Initial behavioural simulation may also take place at this stage via the new behavioural constraint description representation capabilities of EAST-ADL and their transformation to external analysis tools like SPIN, UPPAAL, Modelica, and Matlab Simulink etc, as described in section 3.2. This allows verification of functional requirements, e.g. against temporal or computational constraints or other attribute quantification. Definition of system behaviour in the form of state machines also allows additional dependability assessment via transformation to temporal fault trees, as described in section 3.3.4.

### 3.6.3 Design Level

The Design Level is the third stage of the process and focuses on the realisation of the logical function definitions in the FAA at Analysis Level with the high-level specification of software functions and hardware components, together with I/O connections and any required system resources. The result are design architecture models, e.g. the functional design architecture (FDA) for software functionality and hardware design architecture (HDA) for hardware architecture.

At this point, a more concrete realisation of the system is available, particularly in terms of quantitative data (e.g. in terms of FEV attributes such as power demand and availability, potential component cost, available computational resources, and further dependability data, e.g. hardware failure rates etc). This allows a series of more detailed analyses to take place, such as quantitative safety analyses (e.g. FTA), further behavioural simulation to verify against more concrete constraints, and initial FEV-specific analyses into power distribution/consumption and even estimates of total cable length etc (e.g. via transformation to Matlab Simulink and subsequent behavioural analysis).

Furthermore, with a more concrete definition of available computational resources at this stage, it is possible to perform some performance/timing analyses, e.g. schedulability analysis. This is done on the basis of the timing information from TADL concepts and compatible tools/plugins like Qompass, as described in section 3.4.

### 3.6.4 Implementation Level

The Implementation Level is the fourth stage and deals with the configuration and implementation of the system design in terms of its software and hardware elements. This is typically based on an AUTOSAR model but continues to link to the requirements and constraints defined earlier in the process, providing traceability to show how the original system requirements and features have been realised. With concrete information available, system characteristics can be verified against the results of analyses from previous stages and any additional required analyses (e.g. in terms of the actual power consumption etc) to ensure the system continues to meet the original functional and non-functional requirements defined at the vehicle level and refined through the analysis & design levels.

### 3.6.5 Optimisation

While not a specific analysis technique, the optimisation process described in the next section provides a powerful toolkit that can be applied at various stages of the design process (but particularly the design level) to help refine the design automatically based on the combined results of a number of different analyses, e.g. dependability, cost, power, and timing analyses. The optimisation architecture defines an attractive way of performing a multi-faceted analysis by virtue of its optimisation objectives and the automatic consideration of both system and product line variability present in the design.

# 4 Automatic multi-objective optimisation of system models

This section describes the algorithms necessary to support automatic optimisation of EAST-ADL models. For a fuller introduction to the concepts involved, see D3.1.1, chapter 4.

## 4.1 Brief description of optimisation definitions and concepts

As systems grow more complex, they require increasingly complex analyses and designs to ensure that they meet all their requirements and do so efficiently, with a minimum of waste and excess cost. In many cases, the complexity and time involved means that this cannot be done manually, at least for more than a handful of possible alternative designs. However, automatic optimisation - particularly the type of model-based, multi-objective optimisation discussed in this document - presents one possible solution.

The goal is to rapidly evaluate many different potential designs to find the optimal (or at least more optimal) candidate. However, even though computers can do this automatically many times quicker than it could ever be done manually, the number of possibilities means that it is still prohibitively expensive - if not impossible - to check *all* of the different solutions. This means that we need some kind of algorithm to guide the process and select which solutions to investigate and which to ignore.

In MAENAD, one of the goals of the project is to allow EAST-ADL models to be optimised according to a variety of objectives, such as safety and timing, to enable better designs to be rapidly evaluated and evolved.

To aid in understanding during the following section, here is a brief summary of the concepts involved:

- **Design/Search Space** - The search space is the name given to the set of all possible solutions that can be derived from modifying the original starting point. In our case, this can also be called the design space, because it is the set of all possible designs that can be obtained via the variability inherent in the original model.
- **Optimisation Objective** - An objective is the goal of the optimisation process and typically takes the form of an evaluation criteria and a direction, e.g. to maximise safety, or minimise cost. Some optimisation approaches combine multiple objectives into a single objective via some kind of weighted formula, but the goal is still to minimise or maximise the result of the formula.
- **Multi-objective Optimisation** is the optimisation of more than one objective simultaneously. This means that multiple evaluation methods are required (to evaluate each objective) and it also requires some kind of heuristic for resolving the inevitable trade-offs between multiple, possibly conflicting objectives.
- **Genetic Algorithms** are one of the more popular types of optimisation algorithms. Some types of genetic algorithms also support multiple objectives. They are so called because they echo the evolutionary process found in nature, where organisms with more adaptable genes tend to survive to pass those genes on to later generations where other less adaptable and less successful genetic lines die out.
- **Evaluation & Fitness Functions** - To be able to judge whether one potential solution is better than another, it has to be evaluated against one (single-objective) or more criteria (multiple-objective). In genetic algorithms, these evaluation functions are often called "fitness functions", because they determine the fitness of an individual and therefore its likelihood of survival into the next generation.

- **Populations & Individuals** - In a genetic algorithm, a population is a collection of individuals, where an individual represents a particular solution (or in this case, design model). Each individual has a different genetic encoding.
- **Encoding** - In a genetic algorithm, this is the "DNA" of the individual, representing its genes. There are different types of encodings, ranging in complexity depending on the complexity of the model variability they are required to capture. In this case an encoding represents a particular configuration of the variability in the original model.
- **Dominance** - In a multi-objective optimisation, there is no global "best" solution, because everything is a trade-off between the various objectives. However, it is possible for one solution to *dominate* another if it is evaluated to be at least as good in all objectives. Thus the dominating solution would always be chosen over the dominated solution (it is the optimal design for those particular objective values).
- **Pareto Frontier** - The Pareto frontier is the set of all dominating solutions that have been found (none of the Pareto solutions dominate each other). Essentially, it is the set of current "optimal" solutions, representing the best trade-offs between the various objectives that have been found so far.

## 4.2 Comparison of optimisation algorithms

Automatic optimisation is still a relatively new field, but various approaches have been put forward - often inspired by natural processes - to serve as optimisation heuristics and algorithms. Some of the more prominent examples will be discussed below, with their advantages and disadvantages.

### 4.2.1 Genetic Algorithms (GAs)

Genetic algorithms, as briefly explained above, are optimisation algorithms inspired by the evolutionary processes found in nature. They have been identified as a particularly effective method for solving combinatorial optimisation problems (such as those faced in MAENAD) as they are capable of navigating large, complex search spaces [6]. The general process is as follows:

- A population of candidate solutions is randomly generated. Each individual is represented by a different encoding, which encapsulates the way they can vary from each other.
- Genetic operators - crossover and mutation - are applied to the population to obtain a new set of child solutions, which are added to the population (or, in some genetic algorithms, become the sole new population).
- This process of reproduction continues until the population grows too large, at which point those with the least successful genomes are discarded. This is established by evaluating the solutions in the population according to some criteria and retaining those with the best results - in other words, survival of the fittest.
- After a set number of generations, the optimisation ceases, and the current population should contain the best (i.e., optimal) solutions discovered so far.

The nature of the domain and the context involved can heavily influence the nature of the genetic algorithm and there are many types and variants of GA, and each can be tweaked and modified further according to various parameters. However, they all have the same general features in common: a population of individuals represented by some form of encoding, crossover and mutation operators that can be applied to the encodings to produce new individuals, and some kind of fitness function to evaluate the individuals.

Encodings represent the DNA of an individual. One of the most popular forms of encoding is to represent the individual as a string of integers - the value of each position in the string represents

a different variability point. Take for example a simple model consisting of three components connected in series. Each component can be replicated (i.e., duplicated and connected in parallel to achieve a measure of redundancy). The encoding string for such a system could consist of three digits, each of which refers to one of the three components and indicates how many times it is replicated. Thus the default system would be 111 - indicating no replication - and 121 would indicate that the second component is replicated once, and so on.

This form of encoding is easy to manipulate, but is quite limited in flexibility (for example, it cannot represent hierarchical systems). Other types of encoding are also possible, including tree-based methods, which are more suitable for modelling hierarchical and compositional models. In tree-based approaches, encodings typically consist of a tree of integer strings; some integers open up a new branch (indicating a subsystem) whereas others do not. So for example, the top level encoding of the default system may still be 111, but 121 implies that the second component is replaced by a subsystem, which will in turn have its own sub-encoding, and so forth.

Encodings are important because they make it possible for new individuals to be created in a mechanistic way from the existing population. This is done via two mechanisms, or genetic operators: crossover and mutation. Crossover is essentially reproduction - it takes two individuals and produces a third individual whose encoding is a random mix of the encodings of its parents. So for example, if individual A has an encoding 111, and individual B has an encoding 222, then the child individual C may have an encoding 122. Crossover is designed to persist good gene lines by allowing successful individuals to pass on their DNA to the next generation - the resulting child has characteristics of both parents. Mutation, by contrast, takes an existing individual and randomly alters some part of the encoding to produce a new one, e.g. individual D may be created by taking individual C and mutating it to have an encoding 123 instead. Mutation is meant to ensure that there is an element of randomness to the process, to allow entirely new areas of the design space to be explored and to help avoid the algorithm becoming stuck in a local optimum (a situation where it appears as though an optimum has been found, but in reality there are better solutions elsewhere in the design space, just apparently inaccessible from the current position; mutation enables the algorithm to 'jump' to other areas).

Genetic algorithms can be applied to multiple objectives. Many "multiple objective" algorithms deal with the multiple constraints imposed by the various objectives by combining them into a single objective/fitness function, typically in the form of a weighted formula; in effect, converting a multiple objective problem into a single objective problem. So for example, if the objectives were safety, performance, and cost, each would be normalised and weighted - e.g. 0.4 x safety, 0.3 x performance, 0.3 x (inverse of cost) - and the result of the formula used as the fitness for that individual. However, this is not always desirable and the fitness formula can be very difficult to derive, so better ways of dealing with multiple-objective problems were sought.

One such approach is the penalty-based GA approach. In this approach, one objective is treated as the main objective and maximised - e.g. safety - and other objectives are treated as scaled, weighted constraints and applied as a penalty to the main objective. So for example, a maximum cost limit might be set, and the more the design candidate exceeds the limit, the more harshly the fitness is penalised. Thus even if two solutions both achieve the same reliability, the one that does not violate any constraints - or which violates them by the smallest degree - is preferred by the algorithm. However, successful exploration of the design space often requires infeasible solutions (i.e. those that break the constraints) to be explored too, so the constraints are not hard constraints; instead, a dynamic penalty function is used that factors in not just the degree of constraint violation but also how long the algorithm has been running. By doing this, it allows the algorithm to explore more infeasible solutions early on, when the penalties are still small, and then increasingly focus on more feasible solutions (that do not violate the constraints) as the algorithm progresses. This form of dynamic penalty-based GA has been found to have superior performance compared to either static versions or versions that only allow feasible solutions to be part of the population [7].

Another type of GA is the multi-objective evolutionary algorithm. These are true multi-objective approaches that evaluates against multiple objectives at once and maintains a Pareto front of optimal, non-dominated solutions. This allows a broader set of solutions and generally results in a better coverage of the search space [8], albeit at the cost of higher complexity. These types of algorithms are generally better suited to multi-objective problems because they allow the various trade-offs between the objectives to be better represented, rather than being biased towards one objective and using the others as constraints or weighted penalties etc.

There are many different multi-objective GAs, including:

- PESA-II (Pareto Envelope-based Selection Algorithm 2) [9], which seeks to maximise evenly distributed spread in the Pareto set by including crowding in the selection critera; solutions that are found in less crowded regions of the search space are preferred for selection to encourage the algorithm to look at under-explored areas of the search space.
- SPEA-II (Strength Pareto Evolutionary Algorithm 2) [10]. Unlike PESA-II, which adopts a "pure elitist" strategy that allows no dominated solutions to be retained, SPEA-II does keep some dominated solutions. The selection criteria are altered to include a dominance strength (how many solutions it dominates) and a density value (how close an individual is to other solutions). Thus although dominated, crowded solutions are possible, non-dominated, non-crowded solutions are preferred.
- NSGA-II (Non-Dominated Sorting Genetic Algorithm 2) [11]. In NSGA-II, both non-dominated and dominated solutions are kept in the same population, ranked according to dominance (based on how many other solutions they dominate). More dominating solutions are preferred during selection, and crowding is used as a tie-breaker if two solutions have equal dominance.

Although a full comparison of all the various approaches is difficult, [12] found that PESA was easy to implement and was efficient, NSGA-II was also efficient and widely used, while SPEA-II was criticised for being computationally expensive.

### 4.2.2 Tabu Search

Tabu Search is another type of heuristic algorithm. On the contrary of population-based algorithms like GA, it only considers one solution per iteration. It works as follows:

- An initial solution is randomly generated and evaluated.
- Next, all neighbouring solutions - i.e., those that can be obtained by making a single modification to the initial solution - are generated and evaluated.
- The neighbour with the best fitness is chosen as the next solution, and the process repeats.

In this way, Tabu search deterministically moves through the design space. To prevent it from looping or getting stuck, it also maintains a list of forbidden or 'taboo' moves, which is populated by recently tried solutions. This helps force the algorithm to search in new, unexplored regions, although the list is finite and so it is possible to loop back some time later. The whole process is repeated until either a set number of moves have been made or a solution is found for which no better solutions are available. Modifications to the algorithm are also possible, e.g. by using an adaptive penalty function similar to that found in penalty-based GAs.

Tabu search can be more efficient than GAs in some circumstances [13], requiring less evaluation, but they can also be more prone to becoming stuck in local optima. Variations on the algorithm exist, e.g. to offer better support for multiple objectives; one such approach randomly selects one of multiple objectives to be active at each stage [12], giving a more equal balance to the multiple objectives and exploring more of the search space. In this approach, a set of good solutions is remembered; at each stage, the best newly discovered solutions are added, and any dominated solutions in the resultant set are discarded. This produces a Pareto front. To ensure an even spread of the Pareto front over the search space, a diversification feature can be added wherein

the algorithm can randomly select another Pareto solution to be used as the starting point if no new solutions have been discovered for a set number of iterations.

### 4.2.3 Ant Colony

The Ant Colony optimisation algorithm is another nature-inspired approach, this time taking its source from the way ants navigate. In nature, ant colonies use pheromone trails to establish and navigate efficient routes between their nests and food sites. Although the pheromones evaporate over time, good routes are reinforced and retained as more ants pass through it, while bad routes are left to decay and eventually disappear [14].

The algorithm generally works as follows:

- An initial ant colony is generated.
- The ants conduct a local search and establish initial pheromone trails.
- The better trails (e.g. those leading to better solutions) are reinforced by more ants over a series of iterations.
- The strongest trail is followed and a new ant colony established; the process then repeats.

The Ant Colony approach can have similar performance to Tabu search [14] but can also be more computationally expensive than other approaches, e.g. penalty-based GA. As with the other algorithms, modifications and variants exist, e.g. ones which allow the use of *a priori* knowledge or which store and rank a number of best individuals rather than only a single individual. Using multiple individuals allows a larger portion of the design space to be explored (essentially, by multiple ant colonies) at the expense of performance.

### 4.2.4 Simulated Annealing

Simulated Annealing is inspired by the process of metallurgical annealing: when a metal is heated, the atoms are free to change their position randomly in the high-energy environment. By controlling how quickly the metal cools, it is possible to encourage the atoms in the metal to form in stronger, more beneficial arrangements, thus strengthening the metal. Simulated annealing works in a similar way:

- An initial solution is generated and a high 'global temperature' is set.
- The solution is randomly mutated to another solution by making a single change.
- Acceptance of the solution depends on the temperature; if it is high, then highly random changes are possible and acceptable; if it is low, then only minor changes are possible.
- The process repeats but the global temperature is steadily reduced.

The result of the process is that large areas of the design space are explored early on via highly random changes, and later on only minor improvements are made. Simulated annealing follows a greedy method - it seeks to achieve a global optimum by making decisions that are locally optimum at each stage [15].

Simulated annealing's primary advantage is that it is very fast, but as a result it is not always as thorough as other algorithms.

## 4.3 Tool support for automatic optimisation

The main tool support for automatic optimisation in MAENAD is the HiP-HOPS optimisation engine, developed by the University of Hull as part of the HiP-HOPS analysis tool. HiP-HOPS implements a multi-objective genetic algorithm to allow for the automatic optimisation of system models (in HiP-HOPS format). Currently, only three objectives are natively supported - unavailability (as determined by the HiP-HOPS safety analysis engine, which is based on FTA) cost, and weight. The latter two are calculated as a simple sum of component costs/weights.

### 4.3.1 Variability in HiP-HOPS

Variability is the means by which the design space is defined. In a HiP-HOPS model, variability is represented by **implementations.** Each component in the model can have one or more possible implementations, each of which will have different characteristics (cost, safety etc.). It is also possible for implementations to have their own subsystems, meaning that the variability is compositional. This allows several different types of variability to be supported, including:

- **In-place Substitution** - one implementation can be substituted for another that performs the same purpose and has the same interface, but with different cost and reliability attributes.
- **Compositional Substitution** - an implementation can also be swapped for a subsystem that performs the same purpose and has the same interface, but consists of multiple subcomponents (and possibly more subsystems). This is also known as *architectural substitution*.
- **Replication** - by means of compositional substitution, an implementation can be replaced by a subsystem which contains the same implementation(s), but replicated to achieve redundancy.

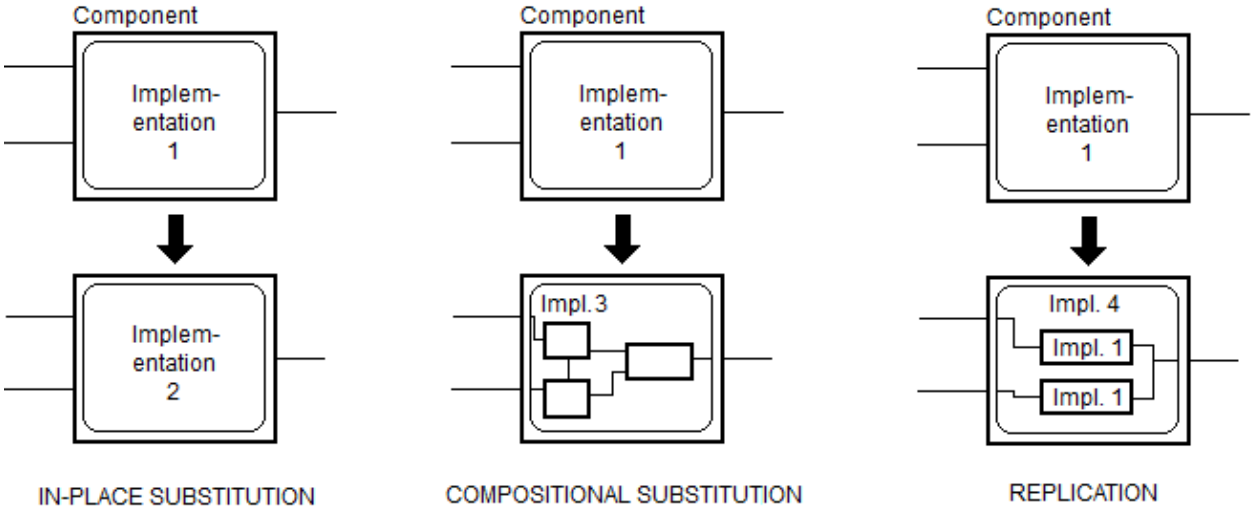



**Figure 56 - HiP-HOPS Variability**

All of these approaches require that the original model is annotated with the additional implementations by the designer. The designer can also add disable or enable certain implementations for a given optimisation.

Currently, each HiP-HOPS Component defines its own interface in terms of Ports and connections to Lines, but the failure behaviour and characteristics of that Component - including any subsystems - is defined solely by the Implementations. This means that the Component interface

is fixed, but the behaviour (and sub-architecture) can be varied by means of the Implementations. Each Implementation has:

- a name
- an optional description
- a flag to exclude it from the current optimisation (e.g. to turn off an particular implementation)
- an optional cost figure
- an optional weight figure
- a setting to indicate whether failure logic is hierarchical or flat
- a set of failure data, including basic events, input & output deviations and their failure logic, any propagations to other perspectives, and any common cause failures
- an optional subsystem

Currently, HiP-HOPS variability is not harmonised with the more extensive variability support found in EAST-ADL. One of the goals in MAENAD is to look at how these can be brought closer together and to identify any EAST-ADL variability features that HiP-HOPS should be extended to support.

### 4.3.2 HiP-HOPS Optimisation Algorithm

As stated above, HiP-HOPS employs a multi-objective optimisation algorithm. In particular, it implements a heavily modified version of the NSGA-II algorithm [16].

#### 4.3.2.1 Encoding

HiP-HOPS originally used a traditional fixed-length integer string encoding, but to support proper compositionality in the model, it was upgraded to use a tree-based encoding that echoes the hierarchical structure of the model itself:

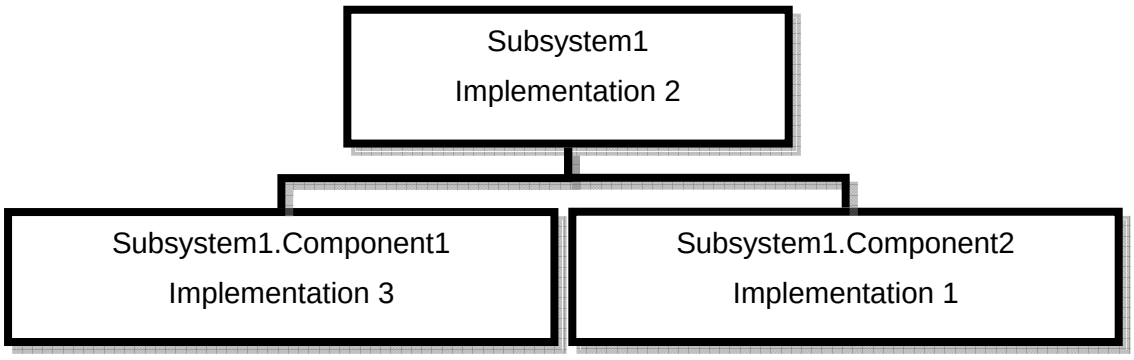



**Figure 57 - Tree Encoding in HiP-HOPS**

In the tree encoding, a certain implementation may have a subsystem of its own, the components of which may in turn have their own subsystems too. A 'default' implementation is always set as this defines the structure of the original design model and facilitates normal safety analysis.

Mutation is handled by randomly changing the implementation of one or more of the nodes in the tree. In some cases this can result in a subsystem being removed, in others it results in a subsystem being added; in the latter case, subsystem's implementations are randomly selected.

Crossover is achieved by performing a simultaneous depth-first traversal of the tree encodings of both parents and the child (the child being constructed in the process). There is a 50% chance of a node from either parent being inherited. Where a subsystem node is inherited from one parent that the other parent does not possess, the entire subsystem is inherited from that parent; where the subsystem is present in both parents, the traversal continues and the sub-nodes are randomly inherited from one or the other parent.

#### 4.3.2.2 Algorithm

In pseudo-code, the HiP-HOPS algorithm is as follows:

```
Randomly initialise population
While (currentGeneration < maxGeneration)
      Generate child population
      Calculate total population crowding
      Crop population that exceeds siseLimit
End While
```

The sub-functions are defined below:

```
Generate child population
While (childPopulationSise < maxChildPopulationSise)
      Select parent1 using Binary Tournament Selection
      Select parent2 using Binary Tournament Selection
      Create child using Crossover of parent1 and parent2
      Mutate child
      Add child to childPopulation
End While
```

```
Crossover
For Each Node in Tree Encoding
     If parent1 node not equal to parent2 node
           Flip a coin to choose parent1 or parent2
           Child node is set to selected parent node
     Else
           Child node is set to (equal) parent node
     End If
End For Each
```

```
Mutation
For Each Node in Tree Encoding
     If within mutationRate
           Randomise node and subNodes
     End If
End For Each
```

### 4.3.3 Achieving full multi-objective optimisation in MAENAD

To achieve the optimisation objectives in the MAENAD project, there are three main steps that must be fulfilled:

- **It must be possible to define the design space**. This is primarily a language issue and means ensuring that EAST-ADL and its variability mechanisms can be used to define optimisation-based variability as well as normal product line variability. Although the language definitions are largely beyond the scope of this document, in practical terms these mechanisms must also be harmonised with HiP-HOPS and any plugins, so that tool support exists for them.
- **It must be possible to evaluate the design candidates**. This will require a close link with both existing and developing analysis tools. HiP-HOPS already has a direct connection with its safety analysis elements but to allow other forms of analysis, e.g. timing, it will likely require some substantial modifications to the optimisation engine, particularly if the engine is to use a modelling tool such as Papyrus as its base.
- **The multi-objective optimisation heuristics must allow for an efficient exploration of the design space**. In practice, the demands of accomplishing the MAENAD objectives are likely to require extensions, modifications, and tweaks to the optimisation algorithm implemented in HiP-HOPS.

The first step is the variability support. If it is not possible to define the design space, then no optimisation can take place. The central principle here is **substitutability**: the language has to allow the designer to specify alternative versions of model elements while ensuring that the alternatives are sufficiently constrained such that they can readily be substituted for one another by the optimisation algorithm. If swapping one alternative for another results in an invalid model, then they are not substitutable. Of particular difficulty here is ensuring a consistent interface; some variants may have more ports or connections than other variants, and so it has to be possible to ensure that those connections remain valid regardless of which variant is chosen. For example, if a variant with more connections to other components is chosen, then failure propagation should be possible along those connections.

This leads on to the second step - that of evaluation of design variants. The notion of substitutability is encapsulated in practice by the process of **variability resolution** wherein a model containing variability is *resolved* to produce a new model in which all the variability has been fixed to choose a particular configuration with a concrete set of objective attributes. Until this problem is solved, and it becomes possible to produce models with resolved variability that can be subsequently evaluated, true multi-objective optimisation of EAST-ADL models will not be possible due to the heterogeneity of analysis tools involved.

Although the tool architecture is still to be fully defined, the overall work process can be divided into four main elements (as shown by the rectangles in Figure 58):

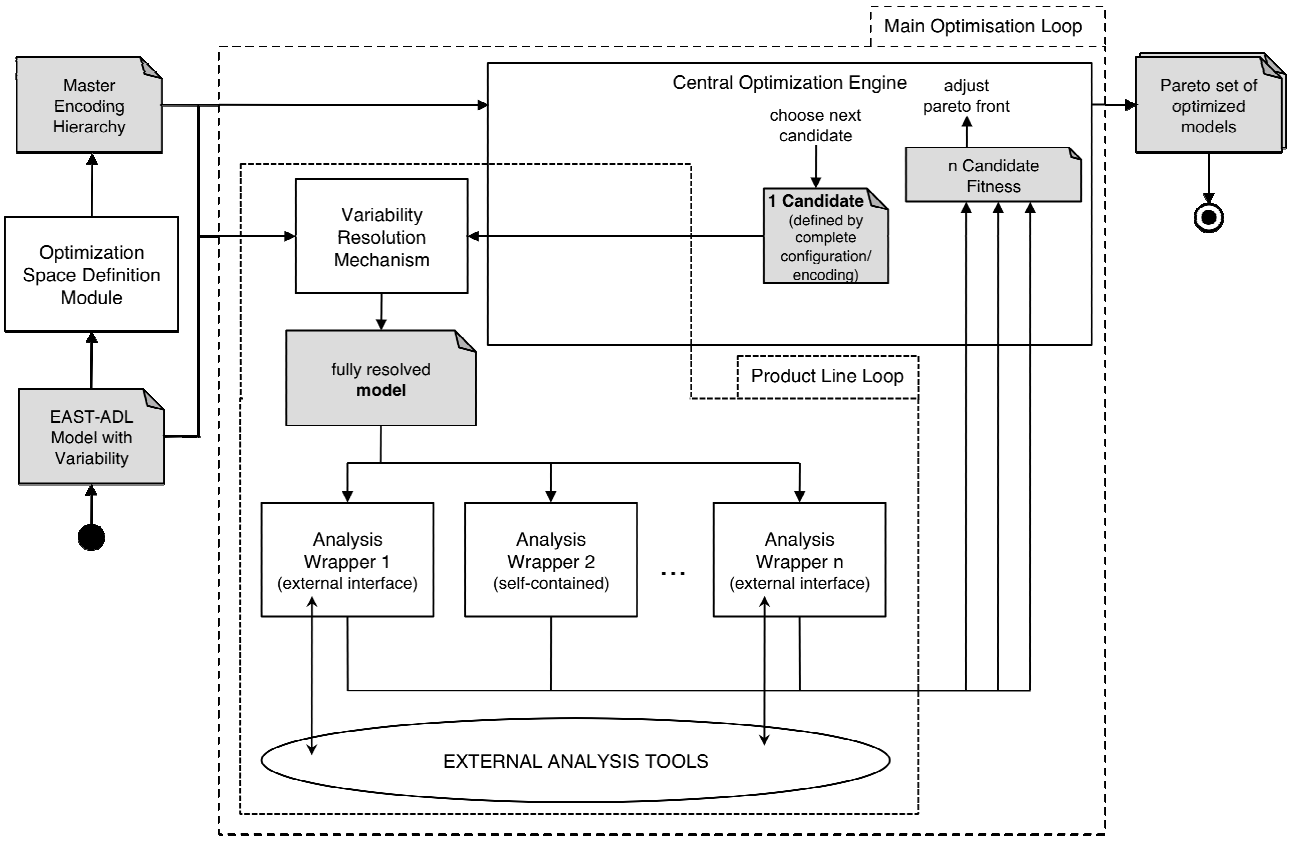



**Figure 58 - Optimisation process**

This diagram shows the flow of operations in the optimisation process. The initial model (with variability) is passed to the OSDM, which generates a Master Encoding Hierarchy. This defines the search space for the Central Optimisation Engine, which generates different model encodings that are passed to the VRM to be resolved into analysable models. The analysis wrappers then evaluate them according to different objectives and return the results. Eventually the optimisation settles on a set of Pareto optimal solutions, which constitute the final results of the process.

The architecture itself will be further explained below.

### 4.3.4 Optimisation Architecture Elements

#### 4.3.4.1 Optimisation Space Definition Module (OSDM)

**Function**

The OSDM provides the input for the overall optimisation process by taking the original model, which contains variability elements to define the variation points of the design, and derives from this input the set of possible encodings of the optimisation search space – the *Master Encoding Hierarchy*, which is a structure that can be manipulated to obtain the set of all valid design candidates (also called population). This allows the Central Optimisation Engine (below) to explore the design space automatically by using an abstract structure rather than needing semantic knowledge of the design model itself. This is "abstract" in the sense that it does not include any information on the structure, behaviour or other properties of the individual candidates; it just provides a means to unambiguously denote each candidate. Unlike the other elements, which are used in each optimisation iteration, the OSDM is only required once, at the beginning of the optimisation, not iteratively in each optimisation cycle as the other elements.

**Inputs**

- A variant-rich model, i.e. one containing variability in the form of variation points. Provided as an external input. May contain both optimisation variability and product line variability (the precise distinction between the two types is still to be determined; it might be binding time or it might be something more explicit).

**Outputs**

- Master Encoding Hierarchy (MEH) of the optimisation space. This will be used by the optimisation engine to select and denote candidates to be analysed in each optimisation cycle. This notion of optimisation space corresponds exactly to what is called a configuration space in feature modelling terminology. Therefore, a feature model would be a natural fit for representing the optimisation space. However, for pragmatic implementation reasons a different form of encoding might be required to simplify the implementation of the OSDM and/or the optimisation engine or to enable the reuse of existing implementations (to be investigated). The format of the MEH is yet to be finalised but is likely to be based on XML somehow.

**Discussion**

The optimisation space definition module (OSDM) takes a model with variability and derives from that an abstract definition of the optimisation space, i.e. a structure which can be manipulated to obtain the set of all valid design candidates – the Master Encoding Hierarchy (MEH). This definition is "abstract" in the sense that it does not include any information on the structure, behaviour or other properties of the individual candidates; it just provides a means to unambiguously denote each candidate.

It takes the form of a tree structure which shows each point of variability in the model; therefore it is analogous (and very similar to) an ordinary feature model. The precise mechanics are yet to be determined, but the principle is that each node represents a variability point and the nature of the node describes the type of variability. For example, assume we have a simple model with one optional component (C1) and another component (C2) that can have one of three implementations (A, B, or C). Implementation C itself is a subsystem that can be replicated up to 3 times for redundancy purposes. This model may produce a MEH like this:

```
MASTER ENCODING
          +
          |
          +---> C1: 0..1  +---> Implementation 1
          |
          +---> C2: 1     +---> Implementation A
                          |
                          +---> Implementation B
                          |
                          +---> Implementation C: 1..3
```

Note that the above is purely illustrative and not meant to be a definitive example of what the MEH may ultimately look like. In particular, the 'types' of variability we wish to include are not yet finalised. The example above uses optionality, multiple implementations, and replication, but we may decide to use other types (like k-out-of-n selections) or fewer types. Also to be determined is whether or not it is necessary (or possible) to 'link' variability points so that e.g. selecting one point automatically enables another variability point too, so we can model dependencies between them.

Regarding product line variability, the exact handling of product line variability vs. optimisation variability is yet to be determined, both in terms of how they are distinguished in the EAST-ADL model and also in terms of how they appear in the Master Encoding Hierarchy. It is likely that only 'true' optimisation variability will be present in the MEH, and the product line variability remains unresolved in the model; in this scenario, the VRM would then be responsible for generating all possible product line examples (resolving both optimisation and product line variability) for analysis purposes. The OSDM could also have a mode in which it strips out or hides any irrelevant variability, e.g. allowing optimisation to take place on only one product line variant. In its initial incarnation, this is likely to be the case, with future extensions to allow full product line functionality (thus we plan for the full functionality but only implement a subset initially).

- Optimisation Space Definition Module (OSDM): this element provides the input for the overall optimisation process by taking the original model, which contains variability elements to define the variation points of the design, and derives from this input the possible encodings of the optimisation search space. This allows the optimisation engine (below) to explore the design space automatically by using an abstract structure rather than needing semantic knowledge of the design model itself. Unlike the other elements, which are used in each optimisation iteration, the OSDM is only required once, at the beginning of the optimisation.
- Optimisation engine: this is the driver of the process, responsible for exploring the design space (by choosing which encodings to evaluate) and for collecting the best solutions so far. The likely algorithm to use for the optimisation is a genetic algorithm, but others could be used (including options to use a simple random selection or enumeration of a small design space, for example).
- Variability Resolution Mechanism (VRM): this element is responsible for taking the original model (with variability) together with an encoding and then producing a new model from them, with all variability resolved. This model can then be subjected to analysis for the purposes of evaluation.
- Analysis: this element is responsible for analysing the model according to the objectives given to the optimisation engine. Typically there would be a separate analysis for each objective, although in practical terms these analyses could be carried out by the same tool or in the same

process. The results of the analysis/analyses are then fed back to the optimisation engine, which uses them to decide whether or not to retain that candidate model.

Apart from the OSDM, the optimisation process therefore consists of a loop - from engine to VRM, to analysis tools, and back to engine - which is iterated continuously until the optimisation is complete.

The OSDM is necessary to ensure that the optimisation engine itself does not need to know anything about the semantics of the models it deals with other than what variability is present in the original model and how it can be encoded. It is then job of the VRM to convert a particular encoding - i.e., a particular configuration of the variability in the model - to produce a new model in which all the variability is resolved. This can then be analysed by the tools which feed their results back to the optimisation engine, which determines whether to keep or discard that particular design candidate on the basis of those results.

#### 4.3.4.2 Central Optimisation Engine (COE)

**Function**

The Central Optimisation Engine is the driver of the optimisation process, responsible for exploring the design space (by choosing which encodings to evaluate) and for collecting the best solutions so far. The COE will use genetic algorithms based on HiP-HOPS technology, but HiP-HOPS itself will only be used as an analysis tool. Instead the intention is to create a new implementation that can more tightly integrate with the other elements of the optimisation architecture (e.g. in the form of a Papyrus plugin).

When optimisation is required, the optimisation engine receives from the OSDM a tree containing the different variation points in the model together with a set of optimisation objectives. This is the *master encoding*, the base DNA of the model to be optimised. To perform optimisation, the optimisation engine chooses a particular encoding (by selecting which variation points to use and which not to use according to its internal heuristics) and passes this on to the VRM. The VRM returns a version of the original model with the variability resolved (although product line variability may still be unresolved). This resolved model is then passed to the plugin interfaces to the various analysis engines (timing, safety, cost etc.), according to the optimisation objectives. Once complete, the analysis results are returned so the optimisation algorithm can evaluate them against its objectives and then repeat the cycle.

**Inputs**

For starting optimisation:

- A set of optimisation parameters. These should specify which analyses are to take place (e.g. safety, timing) and with which tools, which particular attributes are to be optimised (e.g. cost, reliability, schedulability etc.), and whether to maximise or minimise those objectives. Any constraints may also be included here (e.g. a minimum level of safety, a maximum cost etc.). The set of optimisation parameters comes from the user (requiring some kind of user interface).
- The Master Encoding Hierarchy, which describes all the variability points in the model. The master encoding tree should be provided by the OSDM. It is likely to be in XML format and provided in-memory (but there are other alternatives).

During optimisation, in each iteration the engine receives:

- A resolved model; after passing the current encoding to the VRM, the optimisation engine receives a model with the variability resolved accordingly. It can then pass this to the analysis plugins.

- The set of analysis results for each objective; this is essentially a single floating point number for each of the objectives (e.g. total cost, system reliability etc.). It receives this information from each of the analysis plugins, which are interfaces to the relevant analysis tools. It is possible for multiple objectives to be analysed by the same tool, as this is transparent to the optimisation engine (which only needs to know which plugin to call for which objective).

**Outputs**

During optimisation, in each iteration the engine provides:

- A configuration encoding, which is passed to the VRM. The VRM can then resolve the variability in the model according to that encoding (equivalent to configuring the feature model). This is likely to be in the same format as the Master Encoding Hierarchy.
- Execution instructions to the analysis plugins, together with the resolved model to be analysed. Note that the model may still contain *product line* variability; this will be discussed later.

In terms of optimisation results:

- When the optimisation process finishes (e.g. because it meets its time limit, or because no progress has been made for X number of iterations, or because the user manually stops the process etc.), it will output the Pareto set of all optimal encodings it has found so far, together with the evaluation results for each objective. Ultimately this would probably be presented to the user in the form of a selection dialogue; when the user selects a particular encoding (and set of evaluation results), the VRM can take that encoding and provide the user with the resolved model in question. This avoids the need to store all of the optimal models themselves. Thus the Pareto set will probably be stored as a set of XML-based model encodings, together with their analysis results.

**Discussion**

The Central Optimisation Engine is the driver of the whole process, but paradoxically is also the element that needs to know the least about the model being optimised. The engine will likely be implemented using genetic algorithms as a Papyrus plugin (i.e. Java), to allow a closer integration of the different optimisation modules.

Because the COE would (probably) be responsible for dispatching the models to be analysed by the various analysis tools, it will also need to know when those analyses are complete. There is therefore the need for some kind of event or callback mechanism so that the COE knows to carry on with the next iteration of the optimisation. Another question is to decide on the parameters for finishing the optimisation: should it run for a set number of iterations, for a given time period, can it be interrupted etc.? However, these are technical details and if we choose the simplest option now (e.g. number of iterations), we can expand the solution with other methods over time.

#### 4.3.4.3 Variability Resolution Mechanism (VRM)

**Function**

The Variability Resolution Mechanism (VRM) is responsible for taking the original model (with variability) together with an encoding and then producing a new model from them, with all variability resolved. This model can then be subjected to analysis for the purposes of evaluation.

**Inputs**

- A variant-rich model, i.e. one containing variability in the form of variation points. Provided as an external input.
- A configuration encoding (from the optimisation engine). The precise format of this encoding has to be clarified. One option is to use feature configurations but it is likely to be in a similar format as the Master Encoding Hierarchy. It will likely be XML-based and passed in-memory from the COE.

**Outputs**

- A fully resolved model, i.e. one that does no longer contain any (optimisation) variability. No special representation has to be devised for this, because it will just be a normal model with variation points removed. Will be forwarded by the optimisation engine to the analysis tool(s), together with the analysis execution instructions.

**Discussion**

As the name suggests, the main responsibility of the variability resolution mechanism (VRM) is to take a model with variability (provided as an external input to the overall optimisation process) plus the encoding of a particular configuration (provided by the optimisation engine) and produce from that a fully resolved model in which all variation points have been replaced by the correct variant according to the configuration defined by the encoding. In principle, it is also conceivable to accept an encoding that represents an only partial configuration and then produce a partially resolved model, i.e. one with some variation points being resolved but other remaining in the model unchanged. In the context of optimisation, however, this is probably not useful because most analyses will require a fully resolved model.

One complication is with product line optimisation. The mostly likely solution is that the VRM will first resolve all optimisation variability according to the received encoding (resulting in a partially resolved model), and then generate multiple fully resolved models, one for each piece of the product line. It could then pass this set of models (and the production numbers of each one) to the COE so that analyses can take place across the whole product line. However, the precise details are yet to be fully established.

Additional care may be necessary to ensure that optimisation variability, once resolved, still produces a viable, meaningful model, in which the component/function interfaces etc. align correctly. We may need to establish limits or constraints on what is possible to ensure this is the case, but this may require testing of many examples before potential problems can be identified.

#### 4.3.4.4 Analysis Wrappers

The analysis tools are responsible for analysing the model according to the objectives given to the optimisation engine. Typically there would be a separate analysis for each objective, although in practical terms these analyses could be carried out by the same tool or in the same process. The results of the analysis/analyses are then fed back to the optimisation engine, which uses them to decide whether or not to retain that candidate model.

The interface to each analysis tool is provided by a particular analysis wrapper / plugin. Although they should (ideally) present a common interface to the COE, the interfaces to the analysis tools can vary, as each tool has different requirements. In some cases the wrapper may be the analysis tool, e.g. in the case of a custom cost analysis plugin. In other cases, it performs model transformations and passes it to an external tool (e.g. the HiP-HOPS plugin). It should not matter to the rest of the architecture whether an analysis plugin is self-contained or only a gateway to an external tool.

**Inputs**

- Each analysis takes place on a model (or set of models) with all variability resolved, i.e., each analysis looks at only one design candidate and product line variant at a time. It may also take extra parameters as needed per each analysis tool (e.g. to control the type or goal of the analysis).

**Outputs**

- Each analysis tool outputs its own results; ideally we would need some way of standardising the output to make it more easily read in by the optimisation engine, which uses the results to evaluate the design candidate. Ultimately we only need a single floating point value back.

**Discussion**

The integration of the analysis tools is perhaps the most challenging part of the whole process. Clear candidates for optimisation are safety/dependability analysis (with HiP-HOPS), timing analysis, and cost analysis (either simple summation with HiP-HOPS or more advanced cost analysis with a dedicated plug in). Other possible analyses include things like EMI, weight, temperature, and perhaps some kind of behavioural analysis.

Each of these presents its own particular set of problems. For example, safety analysis requires an error model be present in the EAST-ADL model, but it is not fully clear yet how the error model may be annotated with variability and stay synchronised with the nominal model. Cost analysis may work as a simple summation for a single model, but will not work across a product line as large volumes of components are cheaper than small volumes, and so the cost of a given component cannot necessarily be determined until after the whole product line has been analysed and the number of needed components of that type calculated. Thus for product line optimisation, a dedicated cost analysis plugin may be necessary. With timing analysis, performance may be an issue and so a subset of the full timing analysis may be necessary, e.g. only schedulability analysis, concentrating on information present in the EAST-ADL model (without need for tasks, runnable allocations etc.).

In general, analysis should take place on a fully resolved model, i.e. one with no optimisation or product line variability left unconfigured. Thus to analyse a product line, we need to iteratively analysis the complete set of product line models for a given optimisation encoding. Then a loop takes place, cycling through all of the models in the set and analysing them. The VRM is responsible for generating this set, but the analysis wrappers/interfaces are responsible for handling the analysis and calculating the final result (e.g. a sum, product, minimum, maximum etc.). In some cases this could be done sequentially, e.g. if the final result was just a total or a product; in other cases, as with cost analysis, it may be necessary to analyse them all as a set and then calculate the result at the end. Possible performance enhancements may include running multiple analyses simultaneously or perhaps finding some way of ‘sharing’ analysis results (e.g. if two models are substantially the same, perhaps some results could be cached), though this latter idea is likely to present substantial technical challenges.

### 4.3.5 Product Line Optimisation

One of the additional things we included is the notion of product line optimisation. Variability in EAST-ADL was originally included to help cater for different product lines that may have different sets of features but share an overall commonality of architecture. Therefore, the optimisation process had to be extended to work with not just a single model of a product line, but all models of a product line simultaneously.

The inner dashed area in Figure 58 represents this 'product line loop'. The goal is that the optimisation variability will be separate to the product line variability, enabling the former to be resolved by the VRM but not the latter. This results in a model being sent to the analysis wrappers that has had all of the optimisation variability resolved, to settle on a particular architecture to evaluate, but retains the product line variability showing which products have which features. The inner 'product line loop' then iterates (exhaustively or according to a particular selection algorithm) through the different variations in the product line, evaluating each of them separately, and then combining them in an analysis-specific manner to return an overall heuristic evaluation value for that particular optimisation candidate. This process therefore allows for different product line characteristics to be included in the consideration and evaluation of different architectural design candidates.

The following figure shows how the optimization architecture from Figure 58 has been refined to support such product line optimization:

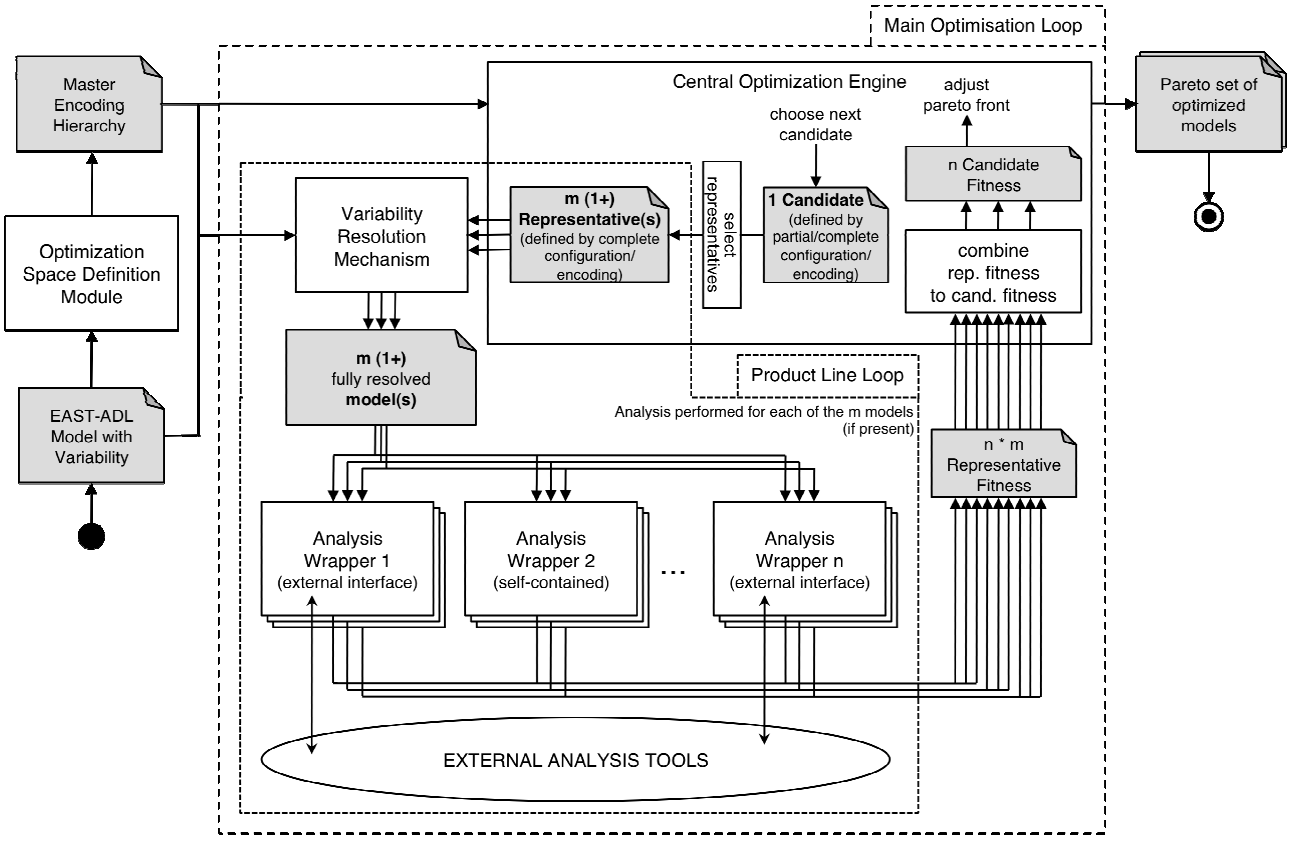



**Figure 59 - Optimisation process with product line support**

### Candidates

A candidate is a certain design variant to be evaluated during optimization. In the non-product line case when all variability in the system model is design space variability, each candidate is defined by a complete configuration and corresponds to exactly one fully-resolved system model. On the other hand, when some variability in the system model is product line variability, then each candidate is defined by a partial configuration and corresponds to a partially-resolved model (or, in other terms, a set(!) of fully-resolved models).

### Representatives

As analysis must always take place on a fully-resolved system model, in the product line case the candidate itself is not sufficient for evaluation of its fitness; instead, one or more so-called representatives have to be selected for the candidate. Each such representative defines how the

candidate's unresolved variability will be resolved and therefore provides a complete configuration and corresponds to a fully-resolved model.

As an example, let us investigate a brake-by-wire (BBW) system. The engineers are considering two potential algorithms (A1 and A2) and whether they should have one ECU per wheel or a single, central ECU (called EpW and ~~EpW~~, respectively). Furthermore, the BBW system will be used in the context of a product line that offers vehicles with two and three axles and an optional stability control. So we have the following variability:

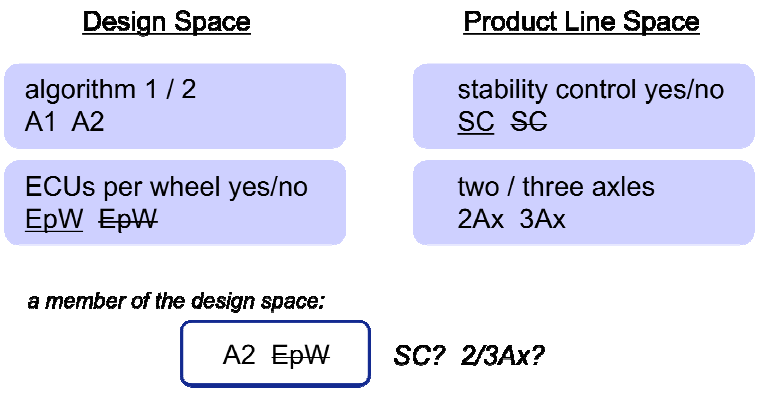



**Figure 60 - Variability of the BBW example**

By performing an optimization, the engineers want to find out the best combination of algorithm A1 or A2 and central or per-wheel ECUs; they are not interested in whether two or three axles is "better" with respect to a certain design objective, because the decision of having 2 or 3 axles will be governed by other factors (probably customer choice), which is typical for product line variability. During optimization, we therefore have to evaluate the following four candidates that make up our design space:

1. A1 EpW (i.e. algorithm 1 with ECUs per wheel)
2. A2 EpW (i.e. algorithm 2 with ECUs per wheel)
3. A1 ~~EpW~~ (i.e. algorithm 1 with central ECU)
4. A2 ~~EpW~~ (i.e. algorithm 2 with central ECU)

Without product line variability, the optimization would be straightforward: each candidate would correspond to a fully resolved model (without variability) and we could send this model to an external analysis engine for evaluation (per objective) and build the pareto-front according to the fitness values returned by the analysis engines. In the product line case, however, each candidate corresponds to several models, because the candidates only decide on the design space variability but make no statement about how to configure the product line variability. The following figure shows this situation for candidate A2 ~~EpW~~:

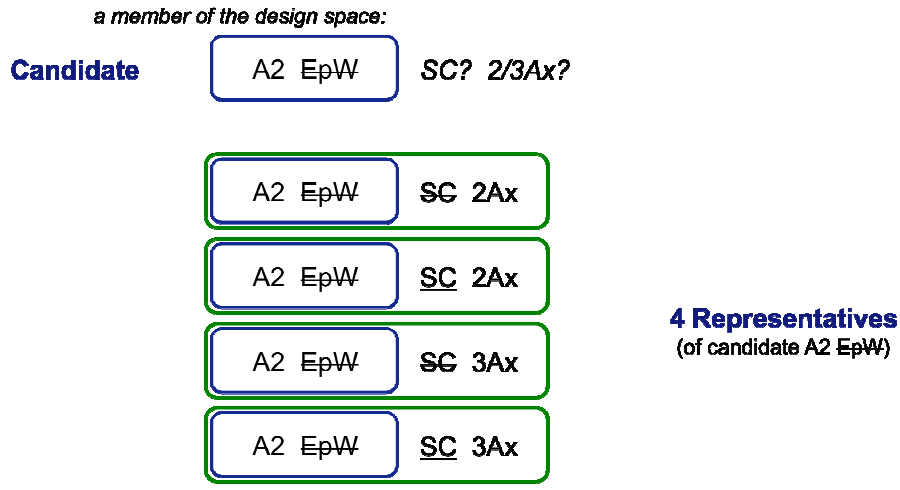



**Figure 61 - A candidate and its 4 representatives in the BBW example**

At this point we are facing a dilemma: for optimization, we are interested in the candidates, not the representatives (because we want to know the best design across the entire product line) but we cannot evaluate the candidates (because they only correspond to partially-resolved models with remaining variability); on the other hand, we could easily evaluate individual representatives, but this is not what we are interested in, because each representative corresponds only to a single member of our product line. The solution, then, is to not evaluate the candidate directly but instead evaluate its representatives with our external analysis engines and then derive a combined fitness value for the candidate from the individual fitness values of the representatives (e.g. by computing a weighted arithmetic mean). Clearly, this introduces two new challenges: (1) selecting which representatives to evaluate for each candidate and (2) finding a reasonable function for computing the combined candidate fitness from the obtained individual fitness values of the selected representatives.

Before going into further detail on these two aspects, let us summarize the notion of candidates and representatives: in the product line case we have two kinds of variability in the model: design space and product line variability; each candidate provides a certain configuration of the design space variability while leaving the product line variability unresolved; for each candidate there are a number of representatives, each providing a certain configuration of the product line variability; therefore, representatives correspond to fully-resolved models.

### Selection of Representatives

The representatives for a candidate are obtained by completing the partial configuration that defines this candidate. “Completing” a partial configuration means configuring all those variation points in the configuration that are still undecided. There are different strategies for selecting the representatives to be evaluated for a given candidate, for example full evaluation (all representatives are selected; only feasible in case of a product line with a very small scope), random selection of a certain number of representatives or selection of the same representatives for all candidates. This selection of representatives is done by the COE, because it is very similar to the selection of a candidate and therefore algorithms and implementations of candidate selection can be reused for this purpose within the COE.

### Combining Representative Fitness into Candidate Fitness

Assuming the COE has selected m representatives R1 … Rm for a given candidate C, then m fully-resolved models (one per representative) will be provided by the VRM, these will each be sent to the n analysis wrappers (assuming we have n objectives / analysis wrappers) leading to m*n

individual fitness values for each representative and for each objective. In the end, the optimization is not about representatives but about candidates, and therefore for each of the n objectives the m representative fitness values have to be combined into a single candidate fitness value, leading to n candidate fitness values (one per objective / analysis wrapper). Again, there are different strategies for combining the fitness of m representatives into the overall fitness of the candidate; a typical example might be a weighted arithmetic mean.

In the non-product line case, m equals 1 and the two versions of the optimization architecture without and with product line support become almost identical (i.e. there is exactly 1 representative per candidate which is defined by the same complete configuration as the candidate). Therefore, the first version without product line support from Figure 58 is contained in the second as a special case and is thus no longer required.

We feel that with this terminology and reference architecture, we have provided a solid framework for complex, product-line-aware multi-objective optimization based on a rich architecture description language such as EAST-ADL. Interesting opportunities for further research lie, for example, in the area of representative selection and candidate fitness computation. It might be interesting to come up with a comprehensive catalogue of possible selection strategies and computation functions and investigate in each case the strength and weaknesses and identify categories of use cases from industrial practice for which they are particularly appropriate or not well suited.

# 5 Implementation support for variability management

This section provides an overview of variability-related plugins used in MAENAD and their relevance for the analysis and synthesis concepts described above.

The tool support for variability is provided in four major parts:

1. **CVM Variability Management Framework:** core CVM with Model Editor, textual VSL Editor, etc.
2. **EAST-ADL Bridge for CVM Organiser:** EAST-ADL related functionality
3. **EPM Function Editor**
4. **Artifact-level variability resolution mechanism**

No. 1 is a stand-alone version of CVM covering the fundamental variability management concepts of EAST-ADL but without relying on the rest of EAST-ADL. No. 2 then integrates CVM into the EAST-ADL language and framework, using the EAST-ADL UML2 profile from CEA as a basis. This also makes it possible to use CVM together with Papyrus. No. 3 is an experimental domain specific editor for editing the core of EAST-ADL elements, especially FunctionTypes and their internal structure. It was initially implemented to experiment with various editing related aspects of variability management in EAST-ADL; for example, to see how the editing of a FunctionType's public feature model and internal binding can best be supported with editing and visualisation functionality in a DSL editor.

The main intention of the above tools is to define system variability from a product line management perspective. This means the defined variations represent differences between distinct products being offered to the end-customer, for example a medium-class versus a luxury car or a passenger vehicle versus a truck. We refer to this kind of variations as *product line variability*.

When using EAST-ADL variability concepts and the corresponding tooling in the context of optimisation, however, there are several important differences to note. In this case, the variations of interest do not represent different variants of the complete system that are actually being built and offered to the end-customer but they represent different solutions for realising and implementing the system that seem worth considering during development. This kind of variation is referred to as *design space variability*.

The key defining difference between product line and design space variability can be summarised as follows: in the case of design space variability, ultimately only a single – the optimal – variant is of interest and will be built and delivered; in the case of product line variability, all variants are intended to be built and shipped.

To illustrate the relevance of this distinction let us consider a brief example. If we were to conduct a cost optimisation across system variants that represent product line variability, e.g. defining the differences between our model range from compact cars via medium-class models up to high-end luxury vehicles, we would always end up with the minimum configuration of the cheapest low-end model. Such a cost optimisation isn't of much use. This clearly shows that not all forms of variability are suited for optimisation. (Note that this is not to say that product line variability does not play any role in design space optimisation; for example, take rates in product line variability may provide relevant information for design space optimisations.)

The distinction between the above two forms of variability is extremely important for the future work in MAENAD, esp. WT 3.2, mainly because:

1. The difference and relation between product line and design space variability is not yet well understood in science and most often only one of these two forms of variation is investigated at a time. For example, it is not yet clear if these two classes of variations can always be clearly separated, or if there are cases where a particular variation in a system

model is partly product line as well as design space oriented (and if so, how to deal with these cases).

2. The EAST-ADL language concepts for variability management were mainly targeted at product line variability. The same applies to the related tooling. Therefore, amendments to the concepts and tools might be required in order to provide a comprehensive solution for defining both product line and design space optimisation within a single system model.

For the MAENAD objective of utilising EAST-ADL's variability handling mechanisms in the context of optimisation, the above two items constitute several demanding research challenges. If solved, however, this will allow dealing with product line and design space variability in a concise and consistent form within a single framework, thus enabling a product-line aware optimisation of system models.

As a first step towards tool support for this kind of combination of design space and product line variability, an exhaustive and reliable artifact-level variability resolution mechanism has to be created in order to facilitate the necessary design space exploration. This very objective is currently being established in No. 4 ("Artifact-level variability resolution mechanism"), which aims to constitute a plug-in with broad support for the complete set of EAST-ADL artifact-level variability concepts. At present, work on this plug-in is still an ongoing effort, although the actual core functionality of VariationGroup-based variability resolution has already been implemented and successfully tested. The plug-in is therefore already being used as the design space exploration basis for small and exemplary optimization projects. Once the plug-in is complete and its overall functionality has been conclusively tested and validated, it is to be ported to the EAST-ADL Tool Platform EATOP, so that in collaboration with CVM-based product line variability, product-line aware optimization of system models becomes a concrete possibility. In its current state, the plug-in already features a number of advanced algorithmic processing functionalities on top of its mere variability resolution capacity.

Firstly, it features a filter algorithm that can detect redundant system configurations and remove them from the set of valid configurations. This kind of processing is necessary in the case of optional elements nested within other optional elements. For each valid configuration where an outer optional element is deselected, the states of the related inner optional elements is effectively irrelevant to the system configuration's behavioural characteristics and can therefore be disregarded. The applied algorithm traverses the system model's component tree recursively and identifies such cases, in order to discard them from the result set later on.

Secondly, the plug-in also features a functionality for resolving variant-rich system models per instance, as opposed to per type, for permitting additional valid configurations that could not be generated the conventional way. This issue is induced by the EAST-ADL's underlying type-prototype-concept, which in effect enforces a specific, selected configuration of a type's inner variable content for each of the prototypes (i.e. instances) that are based on that type, within a specific system model configuration. It is therefore not possible to have differing valid configurations of the variability content of different prototypes, in the case that these prototypes are based on one and the same type — thus: resolution per type. The applied algorithm uses an iterative search-and-rectify approach for identifying patterns in the system model where this issue is relevant, and then resolves each identified occurrence by introducing new types, which are essentially copies of the outlined crucial type, into the system model — one for each of the deviated prototypes. In doing so, the variable contents of each of those prototypes can then be configured independently from each other — thus: variability per instance.

# 6 Summary and conclusions

This document describes some of the techniques, approaches, and algorithms being looked at and developed as part of the MAENAD project. They range from techniques to support safety analysis under the ISO 26262 standard, such as automated ASIL decomposition, to various analysis approaches to take advantage of the capabilities of the EAST-ADL language, to optimisation algorithms that can be used - with the aid of variability mechanisms defining the design space - to automatically optimise design models.

In terms of the project objectives:

- O1-1: Support in EAST-ADL for the safety process of ISO 26262 and representation of safety requirements.

Being able to support ISO 26262 is important for any automotive language or toolset. The initial safety process is briefly defined in Section 2. ISO 26262's approach to safety revolves to a significant degree around the use of ASILs — automotive safety integrity levels — which are used to define required levels of safety and which are defined as part of a hazard analysis of the system. EAST-ADL provides support for this process and for the definition of ASILs.

- O1-2: Automatic allocation of safety requirements (ASILs).

The standard also describes how ASILs can be decomposed across a system; this can be a difficult task if performed manually, and as a result, in MAENAD we have developed a concept for the automatic decomposition and allocation of safety requirements, as described in Section 2.4. It involves the use of the HiP-HOPS analysis tool and its FTA capabilities, and functions by assigning ASIL constraints to the results of the FTA (minimal cut sets, which are sets of basic events that can cause system failures). Once these constraints (known as Derived Safety Requirements) have been applied, the algorithm looks at how they interact to determine what possible assignments of ASILs to basic events will meet the overall requirements. This set of possible ASILs — known as Safety Requirement Allocations — can then be used to determine ASIL assignments for other system elements, such as input/output failures and whole components.

One disadvantage is that the algorithm is subject to combinatorial explosion, and thus as part of MAENAD, we have also investigated the idea of reformulating the ASIL decomposition process as an optimisation problem. Currently, we have developed an approach that uses Tabu search to achieve an efficient and scalable allocation of ASILs.

- O2-1: Dependability analysis of EAST-ADL models (with capabilities for multi-mode and temporal analysis of failures, together with integrated assessment of HW-SW design perspectives).

To support ISO 26262, general safety analysis of system models — such as FTA and FMEA — has to be possible. In MAENAD, these capabilities are primarily provided by the HiP-HOPS analysis tool, which is described in Section 3. As well as providing automatic analysis capabilities, it has also been extended to support the assessment of multiple perspectives (such as separated hardware and software concerns, as found e.g. in EAST-ADL). Section 3 also describes other tools and approaches that can be used for the purposes of dependability analysis. We have also more recently been investigating the possibility of using state-based error models and how these can be translated into more readily analysable forms such as fault trees.

- O2-2: Behavioural Simulation of EAST-ADL models.

Section 3 also describes other analysis approaches, focusing behavioural simulation approaches. With the definition of the EAST-ADL Behaviour Description Annex (BDA) now complete, there is scope for introducing new algorithms and techniques for behavioural analysis. New analytical models for FEV development scenarios (based on EAST-ADL artefacts, properties, and relationships) are still in progress, as is work on additional behavioural analysis algorithms. The BDA has enabled tool development work to link behavioural analysis tools like SPIN and UPPAAL with MetaEdit+. These tools and techniques provide valuable capabilities when applied to EAST-ADL models, enabling the modeller to form a dynamic view of the system design in addition to the primarily static view typically given by most dependability analyses.

- O2-3: Timing Analysis of EAST-ADL models.

The third type of model-based analysis included MAENAD is timing and performance analysis and this is also briefly described in Section 3. The main line of work focused on defining so-called "early-stage schedulability analysis". This analysis should directly work on EAST-ADL models at design level, without needing any additional implementation-level concepts (tasks, priorities, etc) that are currently absent in EAST-ADL. Other work focused on the definition of a transformation between EAST-ADL timing information and MARTE models for schedulability analysis. This transformation along with analysis algorithms have been implemented in the Timing analysis tool called Qompass. Analysis refinements, however, are needed to fully cover complex cases (or at least a subset of them) as the ones arising in the Brake-By-Wire model. In particular a method to handle non-linear event chains is needed.

- O3-1: Extension of EAST-ADL with semantics to support multi-objective optimisation for product lines.

Section 4 describes the concepts towards the implementation of a multi-objective optimisation capability for EAST-ADL models. A new tool and process architecture has been defined, and is in the process of being implemented in prototype form in the "OptiPAL tool". The approach relies upon the use of variability mechanisms to define the design space and then the resolution of this variability — to achieve substitution of one element of a design for another — to enable evaluation of the design candidates. Variability implementation support is summarised in Section 5. Evaluation of the different options is achieved automatically by means of wrappers and bridges to EAST-ADL-compatible analysis tools and plugins; the goal is to allow many choices of objectives (e.g. dependability, timing, cost, FEV-specific options like energy consumption or cable length etc) to be accessible from the optimisation process.

- O3-2: Definition of a library of standard architectural patterns that can be automatically applied on an un-optimised EAST-ADL model in order to improve dependability and performance.

Appendix A at the end of this document contains a library of example architectural patterns for this purpose.

# 7 References

[1] INTERNATIONAL ORGANISATION FOR STANDARDISATION: *Final Draft International Standard ISO/FDIS 26262*, 2010.

[2] Bengtsson, J. and Wang, Y. 2004. *Timed Automata: Semantics, Algorithms and Tools*, Lecture Notes in Computer Science, Volume 3098/2004, 87-124

[3] Zeigler, B.P., Praehofer, H., and Kim T.G. 2000. *Theory of Modelling and Simulation*, 2nd Edition, Academic Press, 2000.

[4] Y.I. Papadopoulos and J.A. McDermid. (1999) "Hierarchically performed hazard origin and propagation studies," in Computer Safety, Reliability and Security, 18th International Conference, SAFECOMP'99, Toulouse, France, September, Proceedings, ser. *LNCS*, M. Felici, K. Kanoun, and A. Pasquini, Eds., vol. 1698. Springer, 1999, pp.139–152

[5] Wasserman, A.I. 1990. *Tool integration in software engineering environments*. Proceedings of the International Workshop on Software Engineering Environments, 1990, Chinon, France. pp 137-149.

[6] Coit, D. W. and Smith A.E. 1996a. Penalty guided genetic search for reliability design optimisation. *Computers and Industrial Engineering*. **30**(4), pp.895-904.

[7] Coit, D W and Smith A E. 1996b. Reliability optimisation of series-parallel systems using a genetic algorithm. *IEEE Transactions on Reliability*. **45**(2), pp.254-260.

[8] Salazar, D, Rocco C M, and Galvan B J. 2006. Optimisation of constrained multiple-objective reliability problems using evolutionary algorithms. *Reliability Engineering and System Safety*. **91**, pp.1057-1070.

[9] Corne, D W, Jerram N R, Knowles J D, and Oates M J. 2001. PESA-II: Region-based Selection in Evolutionary Multiobjective Optimisation. *In*: *Proceedings of the Genetic and Evolutionary Computation Conference GECCO*. Morgan Kaufmann Publishers, pp.283-290.

[10] Zitzler, E, Laumanns M, and Thiele L. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm. *In*: *Proceedings EUROGEN 2001 Evolutionary methods for Design, Optimisation and Control with Applications to Industrial Problems*. Athens, Greece.

[11] Deb, K, Pratap A, Agarwal S, and Meyarivan T. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*. **6**(2), pp.182-197.

[12] Konak, A, Coit D.,W, and Smith A E. 2006. Multi-objective optimisation using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*. **91**, pp.992-1007.

[13] Kulturel-Konak S., Smith A.E, Coit D.W. 2003. Efficiently solving the redundancy allocation problem using tabu search. *IIE Transactions*. **35**, pp.515-526.

[14] Liang, Y C and Smith A E. 2004. An ant colony optimisation algorithm for the redundancy allocation problem (RAP). *IEEE Transactions on Reliability*. **53**(3), pp.417-423.

[15] Kim, H, Bae C., and Park S. 2004. Simulated annealing algorithm for redundancy optimisation with multiple component choices. In: Advanced Reliability Modelling, Proceedings of the 2004 Asian Internationals Workshop. World Scientific, pp.237-244.

[16] Parker, D.J. 2010. Multi-Objective Optimisation of Safety-Critical Hierarchical Systems. PhD Thesis, University of Hull, UK.

[17] E. Frank, Reinhard Wilhelm, Rolf Ernst, Alberto L. Sangiovanni-Vincentelli, Marco Di Natale: Methods, Tools and Standards for the Analysis, Evaluation and Design of Modern Automotive Architectures. DATE 2008: 659-663

[18] Palencia, J. C. and González Harbour, M. 1998. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In Proceedings of the IEEE Real-Time Systems Symposium (December 02 - 04, 1998). RTSS. IEEE Computer Society, Washing- ton, DC, 26.

[19] Richter, K., Ziegenbein, D., Jersak, M., and Ernst, R. 2002. Model composition for scheduling analysis in platform design. In Proceedings of the 39th Annual Design Automation Conference (New Orleans, Louisiana, USA, June 10 - 14, 2002). DAC '02. ACM, New York, NY, 287-292

[20] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. IEE Proceedings Computers and Digital Techniques, 152(2):148–166, March 2005.

[21] UML-MAST project web site. http://mast.unican.es/umlmast/

[22] J. Drake, M. Harbour, J. Gutierrez, P. Lopez, J. Medina, J. Palencia. Modelling and Analysis suite for Real-time Applications (MAST 1.3.7). http://mast.unican.es/umlmast/

[23] A. Rugina, “Dependability Modelling and Evaluation – From AADL to Stochastic Petri Nets. Ph.D. Thesis,” Institut National Polytechnique de Toulouse - INPT, France, 2007.

[24] P. H. Feiler, and A. E. Rugina, “Dependability Modelling with the Architecture Analysis and Design Language (AADL),” Carnegie Mellon Software Engineering Institute, N°CMU/SEI-2007-TN-043, 2007.

[25] A. Joshi, S. Vestal, and P. Binns, “Automatic Generation of Static Fault Trees from AADL Models,” In DSN Workshop on Architecting Dependable Systems, Edinburgh, Scotland – UK, 2007.

[26] A. Rauzy, “Mode automata and their compilation into fault trees,” Reliability Engineering and System Safety, 78(1) , 2002, pp. 1-21.

[27] Mahmud, N. 2012. “Dynamic Model-based Safety Analysis: From State Machines to Temporal Fault Trees”. PhD Thesis, The University of Hull, UK.

[28] J. Lygeros, C. Tomlin, and S. Sastry, Controllers for reachability specifications for hybrid systems, Automatica, Special Issue on Hybrid Systems, 1999.

[29] G.J. Holzmann, The SPIN Model Checker:Primer and Reference Manual, Addison Wesley (2003)

[30] DeJiu Chen, Lei Feng, Tahir Naseer Qureshi, Henrik Lönn, Frank Hagl. An Architectural Approach to the Analysis, Verification and Validation of Software Intensive Embedded Systems. Computing Journal, Springer. To appear.

[31] B. Selic: “A Generic Framework for Modelling Resources with UML”, IEEE Computer vol. 33 no.6, pp.64-69 2000.

[32] Liu, C. L. & Layland, J. W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment." Journal of the Association for Computing Machinery 20, 1 (January 1973): 40-61.

[33] Q. Zhu, Y. Yang, E. Scholte, M. Di Natale, and A. Sangiovanni-Vincentelli, “Optimising extensibility in hard real-time distributed systems,” in Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. pp. 275–284.

[34] C. Bartolini, G. Lipari, and M. Di Natale, “From functional blocks to the synthesis of the architectural model in embedded real-time applications,” in Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. pp. 458–467.

[35] A. Mehiaoui, Sara Tucci-Piergiovanni, Jean-Philippe Babau. Optimising the Deployment of Distributed Real-time Embedded Applications. The 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012).

[36] Mahmud, N., Papadopoulos, Y., & Walker, M. (2010). A Translation of State Machines to Temporal Fault Trees. In: Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-W '2010), pp. 45-51, ISBN: 978-1-4244-7729-6.

[37] F. Glover, "Tabu search-part I," ORSA Journal of Computing, vol. 1, no. 3, pp. 190-206, 1989.

[38] F. Glover, "Tabu search-part II," ORSA Journal of Computing, vol. 2, pp. 4-32, 1990.

[39] M.-H. Lin, J.-F. Tsai and C.-S. Yu, "A Review of Deterministic Optimization Methods in Engineering and Management," Mathematical Problems in Engineering, vol. 2012, 2012.

[40] P. Hansen and B. Jaumard, "Algorithms for the maximum satisfiability problem," Computing, vol. 44, no. 4, pp. 279- 303, 1990.

[41] P. Hansen and K.-W. Lih, "Heuristic reliability optimization by tabu search," Annals of Operations Research, vol. 63, pp. 321-336, 1996.

[42] T.-I. Kuo, S.-H. Chen and J.-E. Chiu, "Tabu Search In The Redundancy Allocation Optimization For Multi-State Series-Parallel Systems," Journal of the Chinese Institute of Industrial Engineers, vol. 24, no. 3, pp. 210-218, 2007.

[43] L. Silva Azevedo, D. Parker, M. Walker, Y. Papadopoulos and R. E. Araújo, "Automatic Decomposition of Safety Integrity Levels: Optimization by Tabu Search," in Workshop CARS (2nd Workshop on Critical Automotive applications : Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security - SAFECOMP 2013, Toulouse, France, 2013.

# 8 Appendix A — Library of Architectural Patterns

This section contains a library of architectural patterns describing common scenarios for improving dependability characteristics. It may be employed as a guide when introducing additional variability to EAST-ADL models for the purposes of optimisation.

The models are also available as EAST-ADL models in EAXML files.

## 8.1 Base function type definition

This represents the most basic definition of an un-modified FDA Function Type. It should fulfil the functional requirements (in terms of carrying out its intended function) but may not do so according to the non-functional requirements (e.g. in terms of carrying out that function with a minimum level of dependability or performance, etc). Therefore, one of the subsequent redundancy patterns may need to be applied.


**Figure 62. Base function type definition**

## 8.2 Duplicate with Combination Block

This pattern represents the replication of the basic function block (see previous pattern) with a second function and a separate element that combines the outputs of the two blocks. The intention is to provide additional redundancy such that a failure of the subsystem occurs only if both the original function and its duplicate fail.

The failure logic of such a system can vary; it may consist of simple AND-based logic (failure of the Voter2 function only occurs if both the function type and the duplicate function type fail), or it may consist of more sophisticated behaviour, such as fail-silent logic (where the Voter2 function acts as a comparator and outputs nothing if it detects differences in the output from the two duplicate function blocks, meaning that potential value failures are transformed to an omission failure).

Furthermore, the two duplicated elements may be homogeneous or heterogeneous; for maximum reliability, the two elements should be heterogeneous to eliminate common cause design failures.

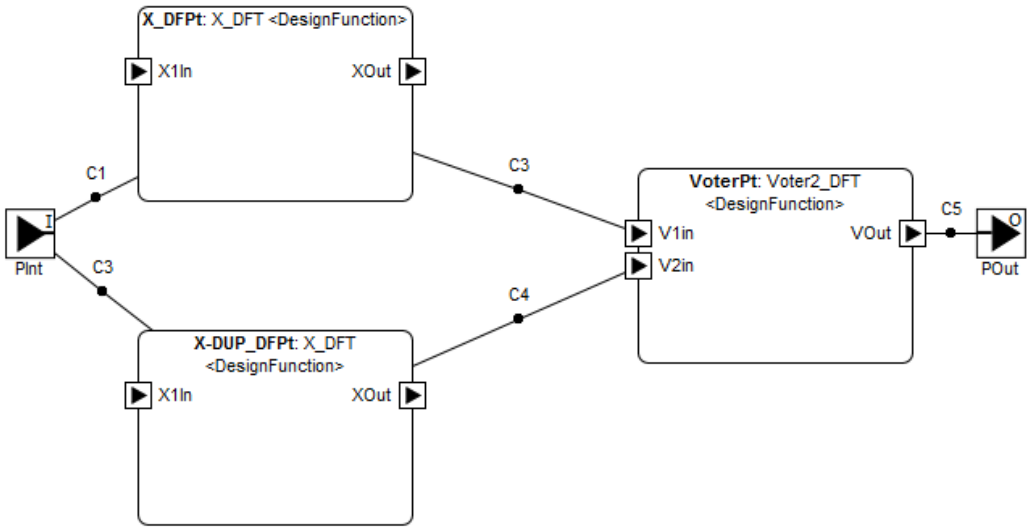



**Figure 63. Duplicate with Combination Block**

## 8.3 Primary/standby recovery block

This pattern represents a type of dual-redundant function. Compared to the relatively simple behaviour of the duplicate pattern (see previous pattern), the primary/standby block offers two separate modes of operation. Under normal operation, the output of the block is provided by the primary function type; this function is also monitored for failure, either by a separate monitor/watchdog function or by the standby function. If the monitor detects failure of the primary function, the standby function is activated to replace it.

The standby function may be a 'cold' spare, which is dormant until required, or a 'hot' spare, which means it is continuously operating even when not required.

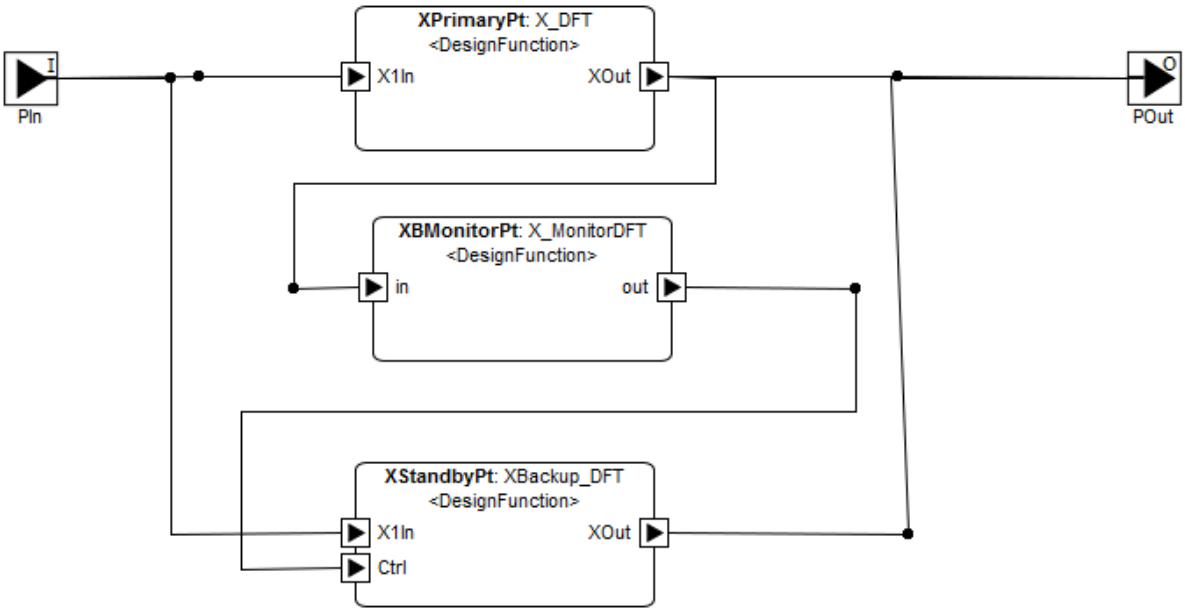



**Figure 64. Primary Standby Pattern**

## 8.4 K-out-of-N Voter

For additional redundancy and reliability, more than 2 duplicates may be employed. These are then connected to a comparator or voter which takes the majority output as the correct output. The minimum needed for a k-out-of-n system to operate is 3 elements to ensure a majority.

These n functions form a 'quorum' and may consist of homogeneous or heterogeneous implementations. For greater reliability, heterogeneous versions of the functions should be employed.

Furthermore, the voter itself could be subject to a redundancy pattern, e.g. replication or primary/standby, to ensure that the voter is not a single point of failure.

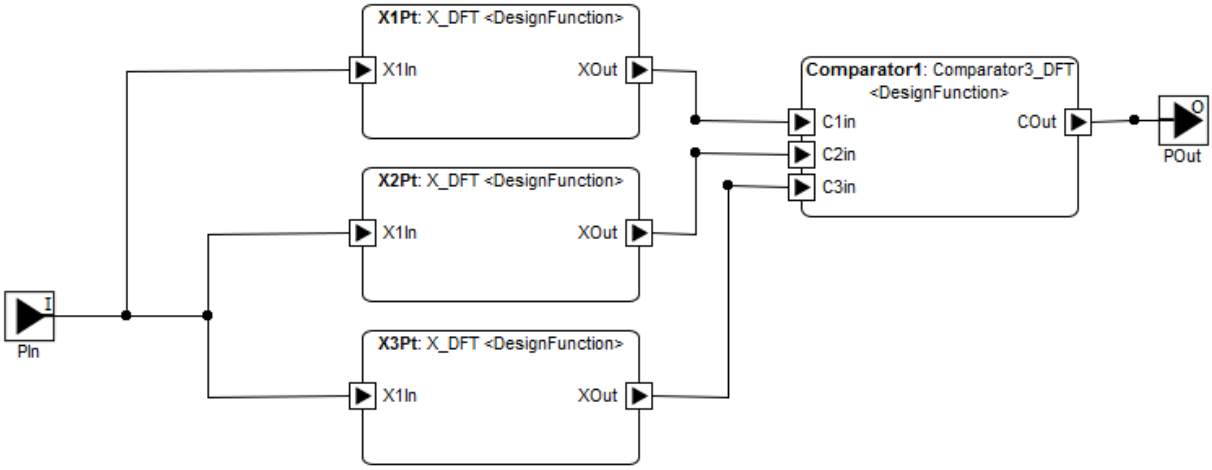



**Figure 65. K-out-of-N Voter**

## 8.5 Triple redundant standby-recovery block

This pattern represents a triply-redundant function. The original base function serves as the primary function, and is connected to a monitor which will activate a secondary function if it detects failure. This secondary function is also monitored (by a second monitor) and will activate a tertiary function on failure. All three redundant functions are connected to a single arbiter function which determines which component is active and forwards the output appropriately.

Again, the failure logic of the arbiter function may vary; it could also perform more sophisticated voting or error detection/mitigation behaviour, depending on whether the duplicate functions are cold standby functions (they remain dormant until the primary fails) or hot standby functions (they are always active and delivering output). At the most basic level, however, the arbiter acts essentially as a propagator, propagating whichever output is active; the resultant failure logic is thus AND logic, so that failure of the subsystem only occurs if all three of the triple redundant functions fail.

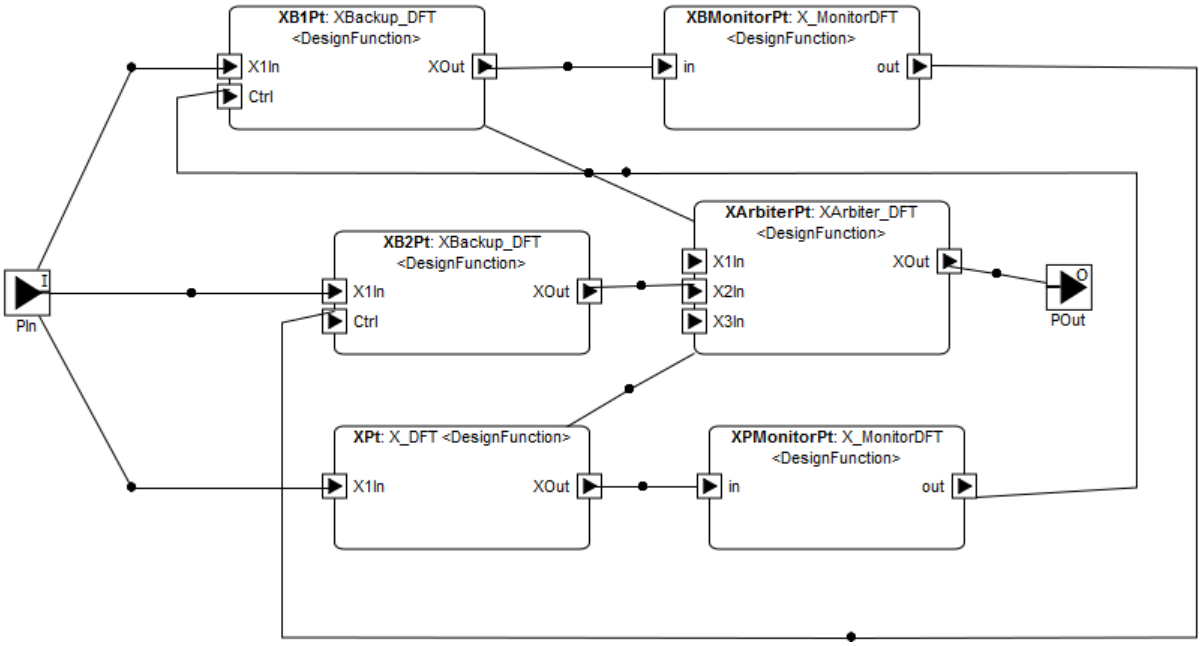



**Figure 63. Triple redundant standby-recovery block**

## 8.6 Pair & Spare

The pair-and-spare system consists of four functional elements arranged in two pairs. Each pair is connected to a comparator which outputs an error signal if the outputs of the two elements of the pair are not in agreement. The two separate pairs are then connected to a third comparator which decides what output to use; if either pair is generating an error signal, then the output from that pair is discarded. Consequently, failure of the overall subsystem only occurs when both pairs experience errors.

As with previous patterns, the pairs may be homogeneous or heterogeneous implementations.

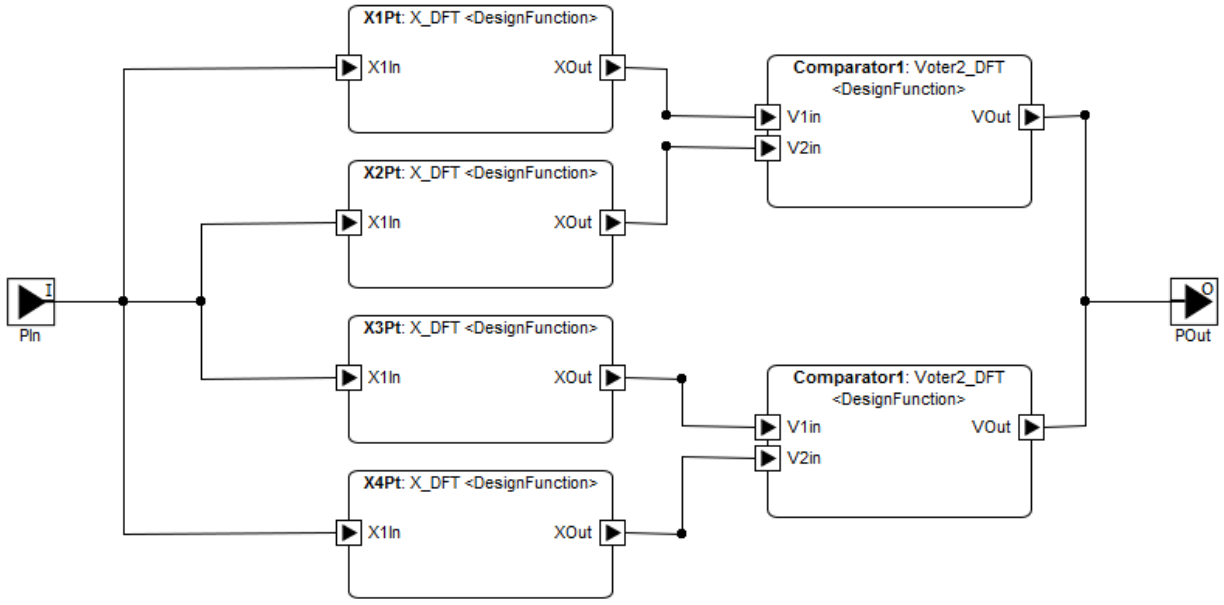



**Figure 66. Pair & Spare**

## 8.7 Base hardware component type definition

Similar to the base function type definition, this is the basic definition of a simple hardware component.

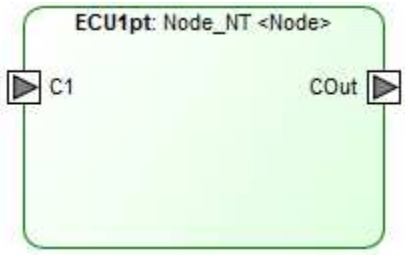



**Figure 67. Base hardware component type definition**

## 8.8 Duplicated hardware component

As with the duplicate function type definition, this pattern replicates the hardware components to achieve additional redundancy. These replicated components should offer similar functionality but may be heterogeneous, e.g. in terms of implementation, vendor, and failure characteristics, thereby helping to avoid systematic or common cause failures that may occur if components of the same design were used.

The logic for handling the combination of the output is achieved on the functional layer (see previous patterns).

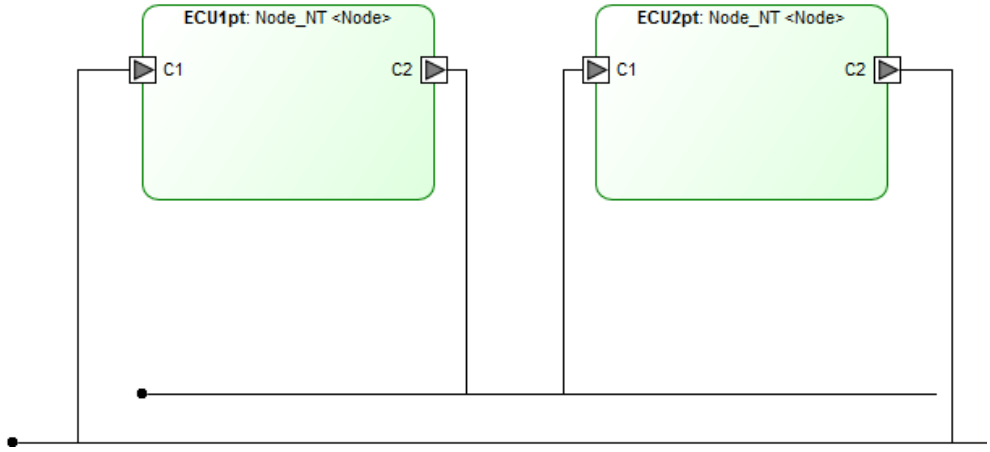



**Figure 68. Duplicated hardware component**

## 8.9 System-level allocation

This pattern illustrates how "multi-perspective" allocation can be achieved. An "allocation matrix" provides a mapping that explains how function types from the functional perspective (e.g. FDA) can be allocated to relevant hardware components — such as ECUs etc — on the hardware perspective (e.g. HDA). Alternative allocations may also be specified, facilitating further optimisation variability by allowing the optimisation algorithms to explore the effects of choosing different allocations.

In the example below, a standard duplication pattern with 1 comparator/voter block is allocated across two ECUs, so that if either ECU fails, there is at least one function type capable of operating on the remaining ECU.

All of the above patterns may be distributed across two or more hardware components in this manner.

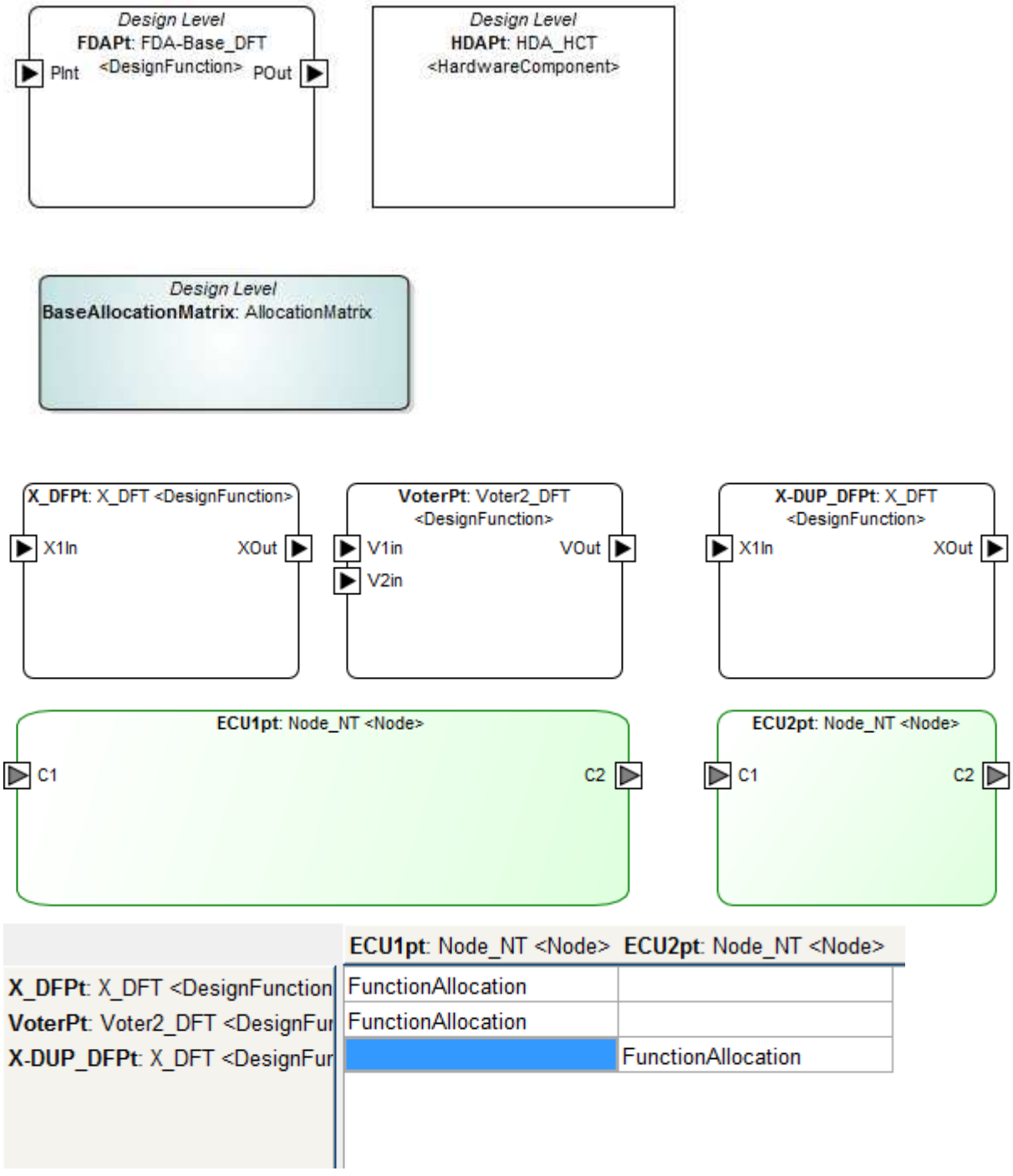



**Figure 69. System-level allocation**