Merging and Processing Heterogeneous Models

P. Dissaux[1] ,B. Hall[2]

1: Ellidiss Technologies, 24, quai de la douane, 29200 Brest, France
2: Advanced Technology, Honeywell, 1985, Douglas Drive, Golden Valley, MN,USA

Introduction

Model Driven Engineering is now recognized as a way to significantly improve the development process of industrial systems and software. This approach leads to the production of various kinds of models associated to each modelling and verification step of the life cycle. All these concrete models may differ in their abstract definition (meta-model) and in their syntactic expression. Such diversity cannot be easily avoided as each modelling language brings its own specific benefit or is fundamentally associated with a particular tool or technique. However, merging and processing heterogeneous models to support all the required development activities can become a real engineering issue in the context of industrial projects.

This paper presents a solution to this problem. The proposed approach is based on the LMP[1] (Logic Model Processing) technology to provide a unique, standardized and easy to process representation of each model that is involved in a given project. Using this solution leads to the realization of a global homogeneous repository from syntactical conversion of each input model, without altering their semantic diversity. It then dramatically facilitates the development of model processing tools, such as model explorations, model verifications, model transformations and architectural reasoning.

1. The model jungle

1.1. Model roles

Models can be used at the various stages of the system development life-cycle. In the descending branch of the life-cycle, we can easily find for a single project at least requirements models, design models and a set of models associated with early verification techniques. If we make the assumption that each individual model is well defined in the context of its own purpose, the key issue is then to ensure a proper inter-operability between these various steps of the life-cycle. This implies that we can manage various kinds of model dependencies, such as traceability (e.g. between design model entities and requirement model entities), transformation (e.g. between a design model and a verification model) and consistency (e.g. between the various subsets of a design model)

1.2. Model syntax

Models can be defined with different meta-modelling languages or approaches, including BNF, XML DTDs or schemas, MOF or ECore. This choice has of course an impact on the way the concrete models can be handled in a tool or a tool-chain. Most of the time, the internal representation of a model in memory is tool dependent. On the contrary, the result of a file serialization must comply with a standard representation which depends on the corresponding meta-model syntax. For instance, BNF leads to token based human readable models, XML DTD or schema leads to XML tag based models and MOF or ECore leads to XMI tag based models. In a given project, these three kinds of concrete model syntax may have to collaborate within a same tool chain.

1.3. Model correctness

Each modelling language carries semantic concepts that are defined more or less formally by the various layers of the language definition. For token based languages, syntax compliancy brings a first level of correctness that must usually be completed by the verification of additional legality rules. With other modelling languages, more rigorous structural rules, such as relation cardinalities, can be guaranteed by construct. However, the actual correctness of a model depends on its foreseen usage, and two models described with the same modelling language may have different contents according to their applicative domain. For instance, the model of a real-time system may look correct in the scope of the verification of its static architecture but not correct in terms of its timing behaviour. Another source of discrepancy may come from the compliancy with the corporate or project engineering rules. For instance the way to build a system in SysML[2] may significantly vary between the various users or tool vendors.

1.4. Model inter-operability

All these differences may be fully justified but can become a blocking issue for the definition of complete tool-chains or for system wide model integration. Several approaches may be considered to solve these model inter-operability issues. The first natural solution is to express all the models with the same meta-modelling language and implement all the tools within the same framework. The typical example of this solution is the use of the Eclipse platform as it has been done with the Topcased and Polarsys[3] initiatives. Such an “all-in-one” approach does minimize the inter-operability problem but raises a certain number of other issues like a lack of modularity and an increase of the effort required to include specialized legacy models and tools. Another solution consists in offering a standardized communication layer by the mean of a logical bus that can be used to ensure model transformations and tool interaction in a transparent way. An example of this solution is ModelBus[4] which allows for heterogeneous tools interaction, but is dedicated to ECore based models. The alternate approach that we are describing below does not have these restrictions. This approach is called Logic Model Processing (LMP).

2. Logic Model Processing

LMP is based on the use of the Prolog[5] language to formally specify rules to be applied to an appropriate representation of the applicative model. This representation of the model is composed of Prolog facts. Prolog (Programmation Logique) is a declarative language that is used to express rules applying on predicates. Rules can then be combined using Boolean Logic. Prolog syntax is very simple and most programs can be specified using AND, OR and NOT logical operators.

LMP consists of a methodology, a set of tools and prolog libraries.

Model Driven Engineering activities are supported by LMP as follows:
- The meta-model classes define prolog fact specifications whose parameters names correspond to the attributes names of the classes.
- An instantiated model consists in a populated prolog facts base, where facts parameters values correspond to classes attributes values.
- The model processing program is expressed as a set of prolog rules whose predicates are others rules or facts.
- To execute a LMP program, it is necessary to produce the facts base associated with the model to be processed, to merge it with the rules base associated with the processing to be performed and to run a query with the prolog interpreter.

With the prolog language being an ISO standard, any prolog environment can be used to support the LMP approach. The one that has been used until now is sbprolog[6]. Other environments such as SWI-Prolog[7] are also being considered. One of the particularities of sbprolog is that the facts and rules bases can be described in textual form or in a binary form (byte code). Sbprolog binary files can be concatenated which highly facilitates the realization of modular processing features and the merge of input models.

In addition to the prolog interpreter itself and its run-time environment, a set of additional tools and reusable libraries are also part of the LMP tool-box. These include in particular input model parsers and output model un-parsers (or printers). The currently available parsers are for XML/XMI models (xmlrev), AADL[8] models (aadlrev), and programming languages. Facts base generation can also be implemented for memory stored models handled by modeling tools.

The main benefits brought by the LMP approach are:
- A clear separation between the model to be processed (facts base) and the model processing program (rules base).
- A strong traceability between model processing requirements and their implementation (one rule per requirement).
- The declarative and logical programming style offered by the prolog language.
- The ability to define modular set of processing rules and to link them together at run time.
- The ability to use a same implementation language for all kinds of model processing, i.e. navigation within the model language constructs (query language), verification of model properties (constraint language), model to model, model to text and text to model transformations (transformation language).

3. Current LMP Applications

3.1. Stood

LMP principles have been applied in their early phase in the Stood[9] design tool. It has been used in this context for more than twenty years to implement various features such as static rules checkers, code and documentation generators as well as reverse engineering tools.

One of the most significant successes of the use of the LMP technology within the Stood tool has been the qualification of customized model verification by Airbus in support of the DO 178 certification process.

3.2. AADL Inspector

After the positive experience of the use of the LMP approach for increasing the capabilities of the Stood tool, it was decided to apply it in an extensive way for the development of the AADL Inspector[10] framework.

AADL Inspector is a model processing environment that can parse AADL models and connect them to a variety of verification and generation tools, such as Cheddar[11] for scheduling analysis, Marzhin[12] for event based simulation and Ocarina[13] for source code generation. AADL Inspector can also be used to convert SysML or UML-MARTE[14] models into a corresponding AADL specification to take advantage of the existing connections with the processing tools.

AADL Inspector can thus be seen as a generic model processing framework using AADL as pivot language and LMP as transformation technique.

3.3. The TASTE tool-chain

The TASTE[15] tool-set resulted from spin-off studies of the ASSERT project, which started in 2004 with the objective to propose innovative and pragmatic solutions to develop real-time software. One of the primary targets was satellite flight software, but it appeared quickly that their characteristics were shared among various embedded systems.

The solutions that have been developed now comprise a process and several tools. The development process is based on the idea that real-time, embedded systems are heterogeneous by nature and that a unique UML-like language was helping neither their construction, nor their validation. Rather than inventing yet another “ultimate” language, TASTE makes the link between existing and mature technologies such as Simulink, SDL, ASN.1, C, Ada, and generates complete, homogeneous software-based systems that one can straightforwardly download and execute on a physical target.

Within the TASTE tool-chain, LMP is used to insure a link between the Domain Specific Models (Interface View and Deployment View) and the corresponding AADL specification that is used during the model processing phases (real-time analysis and code generation).

3.4. The PMM editors

The Property Model Methodology[16] is a system engineering approach that involves several interconnected models: the Specification Model, the Property Based Requirements, four kinds of Design Models and the overall System Model.

A graphical editor is being developed to support this modelling approach and the use of LMP is foreseen to insure the various required model processing needs in this context.

4. Using LMP to process heterogeneous models

The examples of use of the LMP technology that have been presented in the previous section are all confined in the scope of a particular tool or tool-chain: processing HOOD[17] models in Stood, processing AADL models in AADL Inspector and in TASTE.

However, one of the most interesting benefits of this approach is that it can easily be generalized to address the models-interoperability issue that has been expressed in section 1 of this paper. In this section, we explain how LMP can be used to convert, merge and process heterogeneous models.

4.1. Converting heterogeneous models

The role of model parsing in the LMP context consists of performing a syntactic transformation from the original model stored in memory or serialized in a file into a normalized prolog facts base.

In the case of text based input models, the conversion are insured by a parser which output consists of a list of prolog predicates.

In the case of memory based input models, which are for instance produced by a graphical tool, the prolog predicates must be generated with a dedicated printing or serialization feature.

When the sbprolog environment is used, LMP provides a C library for the production of binary predicates. This library can be linked to the model parsers or editors.

4.2. Merging heterogeneous models

The conversion step that is described in the previous paragraph can be applied to all the input models, whatever meta-model they comply with and whatever they are serialized in a file or stored in memory.

The portable way of merging converted models is to concatenate the elementary textual prolog facts bases that have been produced by each conversion tool. I can be noted that additional information can be inserted at that stage under the direct form of dedicated prolog facts. This may be especially useful

to introduce processing instructions (pragmas) into the merged facts base.

The main constraint that applies at this stage is the management of facts redundancy and ordering. A good practice to avoid issues is to ensure that each conversion tool produces a different set of predicates. When this is not possible, the variation of the number of parameters (arity) can be used to avoid facts overwriting. Another solution consists of using a dedicated parameter to identify the model source.

In the case of sbprolog, it is not required to take care of the way textual facts are ordered. With other prolog environments it may be mandatory to group all the similar facts together. Moreover, sbprolog binary facts files can be also concatenated.

At this stage, the resulting facts base is a homogeneous data repository representing the merged heterogeneous input models that is ready for any kind of processing.

4.3. Processing heterogeneous models

Most of what has been explained for the input models facts base can also be applied to the model processing rules bases. The main difference is that the rules bases are usually statically defined and stored in a model processing library, whereas the facts bases are dynamically elaborated from the current state of the applicative model. Another difference comes from the facts/rules separation that has strictly been applied until now for all the LMP realizations. However, some future applications may require that rules are also specified within the input model. This may be the case for instance while adding model constraints insertions during the modelling phases.

All the variety of processing can now be applied to the merged facts base, such as static rules checkers, dedicated model generators to feed verification tools and source code generators.

4.4. Example of use

This section presents a practical example of use of the approach. This case study has been simplified as much as possible for illustrative purpose. We are addressing what could be the development process for a library of operations on complex numbers.

We consider a development workflow that would be composed of four separate steps. Each step is associated with an ISO 12207 standard activity and is likely to use different description languages and tools:

- step1: System Requirements Analysis (5.3.2).
- step2: Software Requirements Analysis (5.3.4).
- step3: Software Architectural Design (5.3.5).
- step4: Software Coding and Testing (5.3.7).

4.4.1 System requirements analysis

The table below gives a small subset of possible requirements for the realisation of a mathematical library on complex numbers.

Id	Name	Text
1	R_ComplexLib	The complex number library must define the complex number type and operations on complex numbers.
2	R_ComplexType	A complex number type must have a Real part and an Imaginary part.
3	R_ComplexAttributes	Real and Imaginary parts of a complex number must be real numbers.
4	R_ComplexAdd	The two operands and the return value of the add operation must be complex numbers.
5	R_ComplexSub	The two operands and the return value of the sub operation must be complex numbers.

When done with a requirements analysis tool, such as IBM Doors or SysML compliant editors, a requirements model can be serialised in various formats. For the purpose of the example, we have selected the Requirements Interchange Format (ReqIF), as it is an OMG standard. A small fragment of the corresponding file is given below:

```
<SPEC-OBJECTS>
  <SPEC-OBJECT LONG-NAME="R_ComplexLib" ...
  <SPEC-OBJECT LONG-NAME="R_ComplexType" ...
  <SPEC-OBJECT LONG-NAME="R_ComplexAttributes"
  <SPEC-OBJECT LONG-NAME="R_ComplexAdd" ...
  <SPEC-OBJECT LONG-NAME="R_ComplexSub" ...
</SPEC-OBJECTS>
```

Applying the appropriate LMP parser (xmlrev) to this file provides the corresponding list of prolog facts:

```
isXMLTag('#39','SPEC-OBJECTS','#13','39').
isXMLTag('#40','SPEC-OBJECT','#39','40').
isXMLAttribute('#40','SPEC-OBJECT',
       'LONG-NAME','R_ComplexLib','40').
...
```

The fact type **isXMLTag/4** keeps track of the XML tags hierarchy whereas **isXMLAttribute/5** provides the name and value of each attribute for each XML

tag. This low level description may be hard to use during the next steps. We can thus improve the access to relevant information by introducing a first level of processing that defines a new set of predicates of type **isSpecObject/1**:

```
getSpecObjects :-
  isXMLTag(X,'SPEC-OBJECTS',_,_),
  isXMLTag(Y,'SPEC-OBJECT',X,_),
  isXMLAttribute(Y,_,'LONG-NAME',R,_),
  assert(isSpecObject(R)).
```

This rule creates a new list of facts that hides the syntactic complexity of the original XML structure and filters the needed data for further use, such as performing requirements traceability.

```
isSpecObject('R_ComplexLib').
isSpecObject('R_ComplexType').
isSpecObject('R_ComplexAttributes').
isSpecObject('R_ComplexAdd').
isSpecObject('R_ComplexSub').
```

4.4.2 Software requirements analysis

We assume that the next modelling step consists in formalizing the data type requirements thanks to a UML class diagram, as shown below:

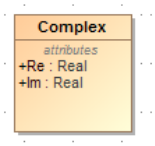

Figure 1 : a UML class

This UML class diagram can be serialized in standard XMI format. A little fragment of the resulting .uml file is given below:

```
<uml:Model xmi:type='uml:Model' ...>
<packagedElement xmi:type='uml:Class' ...
 name='Complex'>
  <ownedAttribute xmi:type='uml:Property' ...
  name='Re'/>
  <ownedAttribute xmi:type='uml:Property' ...
  name='Im' />
</packagedElement>
```

We can then use the same LMP parser as in the previous step to convert this UML model into an equivalent list of prolog facts.

```
isXMLTag('uml#3','uml:Model','#2','3').
isXMLAttribute('#3','uml:Model',
       'xmi:type','uml:Model','3').
isXMLTag('#5','packagedElement','#3','5').
isXMLAttribute('#5','packagedElement',
       'xmi:type','uml:Class','5').
isXMLAttribute('#5','packagedElement',
       'name','Complex','5').
isXMLTag('#6','ownedAttribute','#5','6').
isXMLAttribute('#6','ownedAttribute',
       'xmi:type','uml:Property','6').
isXMLAttribute('#6','ownedAttribute',
       'name','Re','6').
...
```

The XMI serialisation generated by UML tools may be huge even for a small model. To pre-select the useful information for a given processing purpose, we can create a more specialised facts base from the original one by specifying a filtering rule:

```
getUmlClasses :-
  isXMLTag(X,'uml:Model',_,_),
  isXMLTag(Y,'packagedElement',X,_),
  isXMLAttribute(Y,_,'xmi:type','uml:Class',_),
  isXMLAttribute(Y,_,'name',C,_),
  assert(isUmlClass(C)),
  isXMLTag(Z,'ownedAttribute',Y,_),
  isXMLAttribute(Z,_,'xmi:type',
                        'uml:Property',_),
  isXMLAttribute(Z,_,'name',P,_),
  assert(isUmlProperty(C,P)).
```

The new facts base would then looks like the following, and could be easily enriched to contain other details such as attribute types or requirement satisfy abstractions.

```
isUmlClass('Complex').
isUmlProperty('Complex','Re').
isUmlProperty('Complex','Im').
```

4.4.3 Software architectural design

We are now considering the software architecture and express the library as an AADL package, so that it can be used by the other parts of the application. Although the AADL standard has a graphical notation, its main usage is with the textual notation that is human readable and scalable.

```
PACKAGE ComplexLib
PUBLIC
WITH Base_Types;

  DATA Complex
  END Complex;

  DATA IMPLEMENTATION Complex.others
  SUBCOMPONENTS
    Re : DATA Base_Types::Float;
    Im : DATA Base_Types::Float;
  END Complex.others;

  SUBPROGRAM Add
  FEATURES
    C1 : IN PARAMETER ComplexLib::Complex;
    C2 : IN PARAMETER ComplexLib::Complex;
    R : OUT PARAMETER ComplexLib::Complex;
  END Add;

  SUBPROGRAM Sub
  FEATURES
    C1 : IN PARAMETER ComplexLib::Complex;
    C2 : IN PARAMETER ComplexLib::Complex;
    R : OUT PARAMETER ComplexLib::Complex;
  END Sub;

END ComplexLib;
```

In order to be able to process an AADL specification with LMP, we need to use the AADL parser (aadlrev). The result of the parsing is the facts base that is shown below:

```
isComponentType('ComplexLib','PUBLIC',
  'Complex','DATA','',6).
isComponentImplementation('ComplexLib','PUBLIC',
  'Complex','others','DATA','','',9).
isSubcomponent('ComplexLib','Complex','others',
  'Re','DATA','Base_Types::Float','','',10).
isSubcomponent('ComplexLib','Complex','others',
  'Im','DATA','Base_Types::Float','','',11).
isComponentType('ComplexLib','PUBLIC',
  'Add','SUBPROGRAM','',15).
isFeature('PARAMETER','ComplexLib','Add','C1',
  'IN','','ComplexLib::Complex','','',16).
isFeature('PARAMETER','ComplexLib','Add','C2',
  'IN','','ComplexLib::Complex','','',17).
isFeature('PARAMETER','ComplexLib','Add','R',
  'OUT','','ComplexLib::Complex','','',18).
...
```

Due to the token based nature of the AADL syntax as opposed to the XML/XMI based languages, the resulting facts base is much more compact and directly usable without having to define a new set of more precise predicates.

4.4.4 Coding

For the last step of our development process, we will process an implementation of the library in Ada language. A possible realisation in source code could be:

```
package ComplexLib is
  type Complex is record
    Re : Float;
    Im : Float;
  end record;
  function add (
    C1 : IN Complex;
    C2 : IN Complex )
    return Complex;
  function sub (
    C1 : IN Complex;
    C2 : IN Complex )
    return Complex;
end ComplexLib;
```

We can then use another parser of the LMP toolbox (adarev) to build a facts base from the Ada code.

```
packageSpec('ComplexLib','_root_').
typeComponent('ComplexLib','Complex',
  'Re','Float','').
typeComponent('ComplexLib','Complex',
  'Im','Float','').
typeSpec('ComplexLib','Complex','...').
operationSpec('ComplexLib','add','...').
param('ComplexLib','add','...',
  'C1','Complex','in','').
param('ComplexLib','add','...',
  'C2','Complex','in','').
param('ComplexLib','add','...',
  'return','Complex','out','').
...
```

For the same reason as for AADL, there is no need to create a new facts base to select the useful information.

4.4.5 All together

Although they are coming from different modelling languages, the syntactic transformation into prolog predicates allows for global processing of the merged model.

The facts sub-bases can be concatenated to provide all the required data to perform cross-activity processing. In particular, this can be used to verify the consistency of the workflow, such as requirements coverage or code compatibility with the architecture.

All these verification rules can be expressed in standard prolog language, as shown in the two simple examples given below.

Rule1. the data types specified in the software specification must be defined in the software architecture:

```
checkR1 :-
 isUmlClass(T),                          /*UML*/
 not(isComponentType(_,_,T,'DATA',_,_)),/*AADL*/
 write('Error R1 for: '), write(T).
```

Rule2: the data types specified in the software architecture must be defined in the source code:

```
checkR2 :-
  isComponentType(_,_,T,'DATA',_,_),    /*AADL*/
  not(typeSpec(_,T,_)),                 /*Ada*/
  write('Error R2 for: '), write(T).
```

Similar rules could be defined to check that all the system requirements are properly covered by design entities.

After having shown how LMP could ease static processing of merged heterogeneous models, we will now consider dynamic architectural reasoning.

5. Architectural Reasoning Using LMP

The prolog fact-based representation of the LMP form presents a good foundation for the logical reasoning and processing of the architectural models. In the next sections we present some simple examples to illustrate the flexibility of the LMP approach.

5.1.Physical Separation and Independence Analysis

To illustrate this potential, we present a simple LMP-based extension to illustrate how physical zonal independence can be assessed from the LMP model.

In modern aircraft, there is often a need to ensure that the independence assumed within a fault-tree is sufficient to mitigate the potential of physical

damage that may arise from adverse system events such as fire, or explosions. Many manufactures require a minimum physical separation among redundant elements. In Integrated Modular Avionic (IMA) architectures, assuring that this separation is achieved for all of the hosted functions can be non-trivial, as multiple sub-function elements are distributed across the IMA processing and input/output hardware elements. As architectures become increasing networked and distributed, the complexity of such analysis may only increase, hence the ability to address it systematically is attractive.

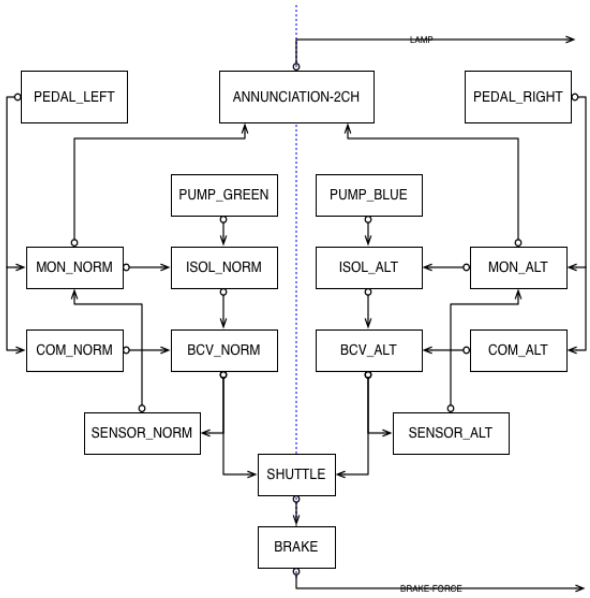

Figure 2 : Example Wheel Brake System

To illustrate the technique we present the wheel brake system case study, shown in Figure 2. This system comprises two independent hydraulic circuits that are each controlled by a dual lane command /monitor computation system. For each circuit, the command processor modulates the expected pressure to achieve the required braking force; the monitor processor supervises the commanded braking operation, monitoring commanded output pressure sensor feedback. If the monitored pressure is not in agreement with the monitor's expected limits, the monitor closes the isolation valve and removes all hydraulic pressure, rendering the channel in active. It then yields control to the other channel.

For brevity of the presentation, our example analysis focuses on the placement of the command and monitoring processing hardware of each lane.

To introduce the physical location of components onto the model a new AADL property is defined. In our simple example, we utilize a single point, but in practice this may be a series of points, which represent the physical boundaries of the components. Inherently extensible, via property sets adding such notions to an AADL model is very straightforward.

```
property set Location  is

Location: list of record
( x_pos : aadlreal;
  y_pos :aadlreal;
  z_pos : aadlreal; )
applies to
( processor, system, abstract, device );

end Location;
```

To represent both good and bad configurations, two system implementations are defined, by extending the base implementation. In the first configuration the command and monitor components within each lane are placed to be adjacent, and the computation elements of the Alt and Norm lanes are separated by 10 meters, as illustrated below.

```
System implementation
wbs_com_mon_dual_lane.good_impl
  extends wbs_com_mon_dual_lane.impl

properties
Location::Location =>
  ([x_pos => 1.0; y_pos => 1.0; z_pos=> 1.0;])
  applies to monAlt;
Location::Location =>
  ([x_pos => 1.1; y_pos => 1.0; z_pos=> 1.0;])
  applies to comAlt;
Location::Location=>
  ([x_pos => 10.0; y_pos => 1.0; z_pos=> 1.0;])
  applies to comNorm;
Location::Location =>
  ([x_pos => 10.1; y_pos => 1.0; z_pos=> 1.0;])
  applies to monNorm;

end wbs_com_mon_dual_lane.good_impl;
```

In the second configuration the monitor of the Alt channel is swapped with the monitor of the Norm channel. It should be noted that, this configuration is insufficient with respect to the required physical channel separation, since failure of a single physical zone will destroy critical components of both lanes of redundancy. Hence, it is our intention is to illustrate this using model analysis.

The AADL composite error model, records the assumptions of the structure of the redundancy, hence the first stage of analysis is to convert this into LMP form. This is done using the ADDL parser (aadlrev) that has been extended under this work to capture the structure of the Error Annex logical expressions.

```
composite error behaviour
states
--Unannunciated braking loss
[
( ( pumpGreen.Failed or isolNorm.Failed or
    bcvNorm.Failed or monNorm.Failed or
    comNorm.Failed ) and
  ( pumpBlue.Failed or isolAlt.Failed or
    bcvAlt.Failed or monAlt.Failed or
    comAlt.Failed ) ) or
( pedalLeft.Failed and PedalRight.Failed ) or
brake.Failed
]-> UnannunciatedBrakingLoss;
```

The LMP representation for the above model comprises two facts; **isEMV2CompositeStateElem** is used to store failure conditions, and **isEMV2CompositeStateExpr** facts are then used to map the logical relationships of the elements. Two example facts are shown below.

```
isEMV2CompositeStateExpr('wbs','wbs_com_mon_dual
_lane','impl','unnamed_A1','unnamed_K1','$1','pu
mpGreen.Failed','OR','isolNorm.Failed',235).

isEMV2CompositeStateElem('wbs','wbs_com_mon_dual
_lane','impl','unnamed_A1','unnamed_K1','$2','bc
vNorm.Failed','NIL',235).
```

In a similar manner to other LMP parsers, element and expression identifies and indexes '$1' are maintained to support the mapping and relating of the facts. Once in LMP form, the next stage of the physical separation, is to convert the logical structure of the fault-tree into a form that can be executed with prolog, This conversion comprises a simple mapping. Each expression is converted to a dynamic prolog fact as illustrated below.

```
init :-
        dynamic('pumpGreen.Failed'/0),
        dynamic('isolNorm.Failed'/0),
        dynamic('bcvNorm.Failed'/0),
        dynamic('monNorm.Failed'/0),
        dynamic('comNorm.Failed'/0),
        dynamic('pumpBlue.Failed'/0),
        dynamic('isolAlt.Failed'/0),
        dynamic('bcvAlt.Failed'/0),
        dynamic('monAlt.Failed'/0),
        dynamic('comAlt.Failed'/0),
        dynamic('pedalLeft.Failed'/0),
        dynamic('PedalRight.Failed'/0),
        dynamic('brake.Failed'/0).
```

The **init** predicate introduces all of the potential faults as dynamic facts in the prolog database. It is complimented by a clear, predicate (not shown) that retracts all facts related to failures. Clear is then used to establish a clean baseline for iterative analysis and queries. This is also useful to support working in the interactive prolog shell, when manually exploring failure combinations.

The **isEMV2CompositeStateExpr** are similarly reduced and mapped to a simplified executable form as illustrated below.

```
exp11 :- exp5,exp10.
exp12 :- 'pedalLeft.Failed','PedalRight.Failed'.
exp1 :- 'pumpGreen.Failed';'isolNorm.Failed'.
exp2 :- exp1;'bcvNorm.Failed'.
exp3 :- exp2;'monNorm.Failed'.
exp4 :- exp3;'comNorm.Failed'.
exp6 :- 'pumpBlue.Failed';'isolAlt.Failed'.
exp7 :- exp6;'bcvAlt.Failed'.
exp8 :- exp7;'monAlt.Failed'.
exp9 :- exp8;'comAlt.Failed'.
exp14 :- exp11; exp13.
exp15 :- exp14;'brake.Failed'.
exp5 :- exp4.
exp10 :- exp9.
exp13 :- exp12.
```

Logical structure of these is generated from the structure of the composite error model state-annotations. It should be noted that the code and effort required to generate executable form from the LMP representation is very small comprising less than 100 lines of prolog in total.

We remind that the logical operator AND (resp. OR) is expressed in prolog by a comma (resp. a semicolon).

Once in this reduced form, prolog is able to compute how the combination of failures impacts the top-level event. In our simple the top-level event is represented by the expression with the highest index.

The next stage of the analysis is to generate the set of failures that can correspond to the different zones. Once processed by LMP, the location properties introduced previously yield a set of facts for each component as shown below.

```
isRecordField('wbs','wbs_com_mon_dual_lane',
'good_impl','monAlt','LOCATION::LOCATION',1,'x_p
os','1.0',275).
isRecordField('wbs','wbs_com_mon_dual_lane',
'good_impl','monAlt','LOCATION::LOCATION',1,'y_p
os','1.0',275).
isRecordField('wbs','wbs_com_mon_dual_lane',
'good_impl','monAlt','LOCATION::LOCATION',1,'z_p
os','1.0',275).
```

The local facts are then grouped by component using a **ftacomponent** predicate as shown below, that simply maps the component name and X,Y,Z location.

```
ftacomponent(P,T,I,Name,X,Y,Z) :-
  isRecordField(P,T,I,Name,'LOCATION::LOCATION',
    1,'x_pos',XA,_),
  isRecordField(P,T,I,Name,'LOCATION::LOCATION',
    1,'y_pos',YA,_),
  isRecordField(P,T,I,Name,'LOCATION::LOCATION',
    1,'z_pos',ZA,_),
  atom_number(XA,X),
  atom_number(YA,Y),
  atom_number(ZA,Z).
```

The system then examines all instances of the **ftacomponent** and generates zones of collocated component using the a simple separation auxiliary predicate, that returns true if the two components are located within a defined separation limit (**Distance**), which in our case was 6 meters.

```
separation((X1,Y1,Z1),(X2,Y2,Z2),Distance):-
  Xd = X2 - X1,
  Yd = Y2 - Y1,
  Zd = Z2 - Z1,
  D is sqrt((Xd * Xd) + (Yd * Yd) + (Zd * Zd)),
  D < Distance.
```

As illustrated below the code to **build_zones** is relatively terse comprising only a few lines of code. This is one of the attractions of the declarative nature of the prolog processing.

```
build_zones([],PP,TT,II,Acc,Acc).
build_zones([(_,X,Y,Z)|T],PP,TT,II,Acc,Final):-
findall((B,BX,BY,BZ),
   (ftacomponent(PP,TT,II,B,BX,BY,BZ),
   (seperation((X,Y,Z),(BX,BY,BZ),6))),Zone),
    (not(member(Zone,Acc)) ->
       append(Acc,[Zone],NewAcc);
       NewAcc = Acc ),
    build_zones(T,PP,TT,II,NewAcc,Final).
```

Once the zones are complete, we simply instantiate a zonal related failure set for each zone, but adding zone predicates to the output. The zone predicates for the good and bad wheel brake command monitor configurations are shown below.

```
zone_good_impl0:-
       assertz('comAlt.Failed'),
       assertz('monAlt.Failed').
zone_good_impl1:-
       assertz('monNorm.Failed'),
       assertz('comNorm.Failed').
```

```
zone_bad_impl0:-
       assertz('monNorm.Failed'),
       assertz('comAlt.Failed').
zone_bad_impl1:-
       assertz('comNorm.Failed'),
       assertz('monAlt.Failed').
```

To explore the system, it is only necessary to instantiate each of the generated zonal fault-sets with the fault tree expression logic discussed previously, and querying the state of the top-level hazard, in out case **exp15**. To package such analysis in smaller form large LMP database, a stand-alone file 'ftap.pro' is generated by the LMP processing logic.

This can then be loaded into the interactive SWI-Prolog environment. A trace from an interactive session following the loading of the file is shown below for the good component placement example.

```
- ['ftap.pro'].
true.
?- init.
true.
?- clear.
true.
?- exp15.
false.
?- zone_good_impl0.
true.
?- exp15.
false.
?- zone_good_impl1.
true.
?- exp15.
true .
```

Evaluating **exp15** following the single zone failure **zone_good_impl0** concludes false, indicating that the failure of **zone_good_impl0** is not sufficient to cause the top-level event. However, the subsequent additional failure of **zone_good_impl1** does cause the top-level event, and the subsequent query of **exp15**.

Repeating the procedure with the bad configuration, we see that failures of the single zone are sufficient to cause the failure of the top-level event as illustrated in the trace below.

```
?- ['ftap.pro'].
true.
?- init.
true.
?- clear.
true.
?- exp15.
false.
?- zone_bad_impl0.
true.
?- exp15.
true .
```

Note, that in our toy example, we are using a simplified representation of location. In real deployments our single point may be expanded to present the component boundary points. Similarly, other types of queries such as the use of a common CPU type, cooling zone, and or power-supply distribution etc., are easily implemented given AADL's extensible property provisions. In each case basic analysis technique would remain largely unchanged, only requiring adaption of the **build_zones** criteria.

5.2. Modelling Completeness Checks.

A second application of the LMP processing is the implementation of automated model completion and completeness checking. In large systems the level of detail and abstraction within the model needs to be managed and maintained. Organizations often maintain modelling standards that define the required model content. However, enforcing such standards can be cumbersome without the appropriate automation.

However, if the model is expressed in LMP, the automation of consistence and compliance checks becomes very simple. Given that, all aspects of the architecture are represented by facts, simple queries against the fact bases can be generated for each requirement. For example, as shown below, only a few lines of code are necessary to execute the query to check that all processing hardware components have a consistent error model associated with them.

```
isComponentType(P,_,X,'PROCESSOR',_,_),
not(isAnnex(P,X,_,_,'EMV2',_,_)),
writeErrorMessage(P,X).
```

By examining the architectural model, using additional system composition predicates and/or predicates derived from fault-tolerance theory predicates, the architectural correctness may be simply validated. For example, a simple predicate may check that all bus components have a specified error model, e.g. Bit Error Rate (BER). A more

advanced predicate may check that protocols that are bound to the bus utilize a suitable end-to-end data transport protocol to mitigate the expected error-rate. More elaborate queries, such as Byzantine vulnerability analysis, are also possible, by highlighting all instances where forked data paths of the logical model, have terminals that bind to different physical components.

Related Work

The fault-tree representation used by our illustrative case-study was inspired from the original work of Shuchi[18].

In the area of architecture model processing, several alternate solutions have been explored, such as REAL [19], RESOLUTE [20] and AGREE [21]. The definition of a “constraint annex” is also under discussion by the AADL standardisation committee.

On-going and Future Work

LMP is used for the development of the new processing plug-ins that will be integrated in the future distributions of the AADL Inspector tool. The current work is focused on extending the import capabilities for UML profiles models such as MARTE, SysML, SCADE System, or CAPELLA as well as new processing features, especially for safety analysis.

In order to facilitate the connection with Domain Specific Modelling Languages, an automatic generation of most the prolog rules that are required to parse, navigate and process ECore based models is being developed. A similar approach is also envisaged for XSD definitions.

Following the idea that LMP allows a tool agnostic infrastructure to be developed, integration with solutions like Modelbus is also considered.

Another area of active research is the generation/ derivation of the AADL Error-Annex composite model annotations from a case-based reasoning approach of the known component failure modes in conjunction with architectural topology.

Conclusion

This paper introduces the raising issue of heterogeneous models processing and proposes an original solution to address it. This solution merges the principles of logic programming and those of model driven engineering to define the Logic Model Processing (LMP) approach.

This approach and the supporting tools are described in the paper and several examples are provided to illustrate its benefits.

References

[1] “Model Verification: Return of Experience”, P. Dissaux and P. Farail, ERTS 2014.
[2] SysML: Systems Modeling Language, http://sysml.org
[3] Polarsys: https://www.polarsys.org/
[4] Modelbus: https://www.modelbus.org/
[5] prolog: ISO/IEC 13211-1, 1995
[6] sbprolog: Stony Brook Prolog, https://www.cs.cmu.edu/Groups/AI/lang/prolog/impl/prolog/sbprolog/0.html
[7] SWI-Prolog: http://www.swi-prolog.org/
[8] AADL: SAE AS-5506B, 2012: http://www.aadl.info/
[9] Stood: http://www.ellidiss.fr/public/wiki/wiki/stood
[10] AADL Inspector: http://www.ellidiss.fr/public/wiki/wiki/inspector
[11] Cheddar: http://beru.univ-brest.fr/~singhoff/cheddar/
[12] “The SMART Project: Multi-Agent Scheduling Simulation of Real-time Architectures”, P. Dissaux, O. Marc and all, ERTS 2014.
[13] Ocarina: http://www.openaadl.org/ocarina.html
[14] MARTE: Modeling and Analysis of Real-Time and Embedded Systems, http://omgmarte.org/
[15] TASTE: http://taste.tuxfamily.org/
[16] “Model Based System Engineering”, P. Micouin, ISTE and John Wiley & Sons editors, September 2014.
[17] HOOD: Hierarchical Object Oriented Design, http://www.esa.int/TEC/Software_engineering_and_standardisation/TECKLAUXBQE_0.html
[18] "Construction of a fault tree using prolog." Fukuda, Shuichi. ICF6, New Delhi (India) 1984. 2013.
[19] “Modeling and verification of memory architectures with AADL and REAL”, S. Rubini, F. Singhoff and J. Hugues, ICECSS 2011.
[20] "Resolute: an assurance case language for architecture models." Gacek, Andrew, et al. Proceedings of the 2014 ACM SIGAda annual conference on HILT. ACM, 2014.
[21] "Compositional verification of architectural models.", Cofer, Darren, et al. NASA Formal Methods. Springer Berlin Heidelberg, 2012. 126-140.