LAMP: A new model processing language for AADL

P. Dissaux[1]

1: Ellidiss Technologies, 24, quai de la douane, 29200 Brest, France

Introduction

Increasing use of Model Driven Engineering in industry implies the need to elaborate efficient solutions for model processing. Typical model processing activities address early verification (model exploration, constraints enforcement, static analysis, interfacing with verification languages and tools) as well as production (automatic generation of documentation, code or test cases).

Most of the time, model processing languages are closely associated with the modelling language they are working on, such as OCL [9] for UML. Sometimes, they can be dedicated to a specific development environment like Eclipse Acceleo [10]. Finally, they have generally been developed to meet a particular need, such as ATL [11] for model transformations.

The AADL [1] community has also identified the need to define processing languages that can leverage the intrinsic strong semantics of this modelling approach. Dedicated languages like REAL [12], LUTE [13] or RESOLUTE [8] have been developed in that purpose.

This paper presents an original and powerful model processing language that can be directly embedded within AADL models. Based on the use of the Prolog language [14] and the LMP framework [2,3], this new model processing language is called LAMP standing for Logic AADL Model Processing.

1. LAMP Foundations

1.1. The Prolog language

Prolog, whose name is a shortcut of Programmation Logique in French, is a declarative language that can be 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 only AND, OR and NOT logical operators.

Executing a Prolog program consists in specifying a query on one of the rules and letting the interpreter find all the solutions for which this query is logically true.

The Prolog language is an ISO standard (ISO/IEC 13211-1, 1995) and many development and training resources are available, either with free access or commercial support.

1.2. The LMP framework

LMP (Logic Model Processing) is an adaptation of logic programming to Model Driven Engineering using standard prolog language. The LMP framework consists of a methodology, a set of tools and Prolog libraries.

Assuming that the modeling language to be processed is defined by a meta-model, the LMP methodology can be summarized as follows:

- Each class of the meta-model defines a Prolog fact specification whose parameters correspond to the attributes of the metaclass.
- 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 by a set of Prolog rules whose predicates are other 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.

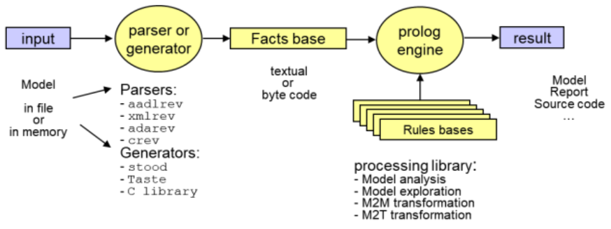


Figure 1: Logic Model Processing

LMP has been used for the implementation of many tool features and successfully applied in the context of industrial projects [2]. This significant return of experience has contributed to the definition of a set of libraries that facilitates the development of new LMP applications.

LMP currently uses the sbprolog open source prolog interpreter [15] but could be adapted to any other Prolog environment if needed.

When applied to AADL model processing, LMP gives access to a low-level API to all model elements corresponding to a node in the parser's abstract tree. For instance, the corresponding AADL declarations will be translated into a Prolog facts base by the parser as shown below:

```
package P public
system S
features
  I : in data port D;
  O : out data port D;
end S;
data D end D;
end P;
```

```
isPackage('P','PUBLIC',1).
isComponentType('P','PUBLIC','S','SYSTEM',…).
isFeature('PORT','P','S','I','IN','DATA','D',…).
isFeature('PORT','P','S','O','OUT','DATA','D',…)
isComponentType('P','PUBLIC','D','DATA',…).
```

It is thus possible to use standard Prolog queries or rules on top of such a facts base to get any processing result. For instance, the following query prints all the component types and their category that are found in the AADL statements defined above.

```
isPackage(P,'PUBLIC',_),
isComponentType(P,'PUBLIC',N,C,_,_),
write(T), sp, write(C), nl.
```

This can be interpreted as follows: "*look inside the public section of all the packages (P), AND get the name (N) and category (C) of all found component types AND print them*". The result of such query will be:

```
SYSTEM S
DATA D
```

The benefits of the LMP approach are multiple:

- It uses an existing ISO standard language: There is no language specification and maintenance cost. Moreover, its semantics is formally defined and many tool implementations and learning material are available.
- The declarative style of the Prolog language is very appropriate to specify queries and processing rules; implicit loops make Prolog programs more readable.
- The clean separation between the facts bases (input data) and the rules bases (program) brings robust and secure model processing implementations.
- The approach can be applied to any kind of data source, in memory or in a file, given that there is a way to convert it into a facts-base in its textual or binary form. Usual parsing technologies can be used for this purpose.
- Facts bases from different data sources can be merged to perform cross-models processing.

1.3. The AADL standard

AADL has been defined to describe software intensive real-time systems and to embed a sufficient level of semantics to enable the use of early analysis or production tools.

AADL is an international standard of the SAE, Aerospace Division, under reference AS-5506C, January 2017.

An AADL model can be fully described by its textual representation, which makes it very scalable and appropriate for version and configuration management.

The language standard is composed of a core and several optional annexes. The core language addresses the description of multi-threaded, distributed software architectures. Currently standardized annexes cover in particular Time and Space Partitioned architectures (ARINC 653 annex), Real-Time behaviour (Behavior annex) and Error modelling (EMV2 annex). Some time ago, the AADL standardization committee started the specification of a constraint language for AADL. However, this work was stopped, and the currently available solutions take the form of specific annexes embedding foreign sub-languages such as LAMP that is further described in the next sections.

Interfacing AADL models with verification solutions requires having access to all the modelling entities through an appropriate API. This API may be implicitly available with the modelling framework, such as with Eclipse based tools, or specifically designed as a separate tool, like the LMP AADL parser (aadlrev) that produces the complete Prolog facts base corresponding to the input AADL model.

Except for the simplest static model verification activities, the AADL declarative model requires to be instantiated prior to be processed. This leads to the definition of an AADL instance model that can be automatically derived from the declarative model as soon as the top-level component has been identified.

2. LAMP Principles and Implementation

It is thus possible to use LMP principles with AADL models and this approach has been applied intensively to develop most of the AADL processing features that are embedded inside the AADL Inspector tool. A summary of these LMP programs is listed below:

LMP feature	category
AADL semantic rules	model checker
AADL instance builder	model exploration
AADL ARINC 653 rules	model checker
UML MARTE to AADL	model transformation
SysML to AADL	model transformation
Capella to AADL	model transformation
AADL to Cheddar	model transformation
AADL to Marzhin	model transformation
AADL to OpenPSA	model transformation
AADL printer	model unparser
LAMP checker	model checker

Table 1: LMP plugins in AADL Inspector

However, all these features are statically defined at tool design time and cannot be customized by the user. The need to be able to develop LMP features at modelling time has been identified. This is useful for prototyping new LMP features, to perform dynamic model explorations or when the set of rules must remain closely attached to the AADL input model, for technical or confidential reasons. This is the purpose of LAMP.

2.1. Principles

Like LMP, LAMP allows for writing standard Prolog programs operating on standard AADL models. Low-level access to AADL model elements is still given by the Prolog facts that are automatically generated by the LMP framework. However, as opposed to LMP, LAMP rules can be embedded within the AADL model under the form of Annex subclauses.

LAMP is currently implemented in the AADL Inspector tool [4] and is composed of three main components:
- LAMP Annexes in the end user source text, where processing goals and rules can be defined.
- The LAMP Standard Library providing a list of predefined utility rules and access to the AADL declarative and instance models.
- A LAMP Checker plugin that assembles the Prolog facts and rules bases together, runs the Prolog interpreter and displays the results in a console.

Figure 2 shows that LAMP lies on top of the LMP framework (parsers and libraries), which itself makes use of the Prolog environment (interpreter and libraries). For the components in orange colour, the Prolog source code is available to the end user.

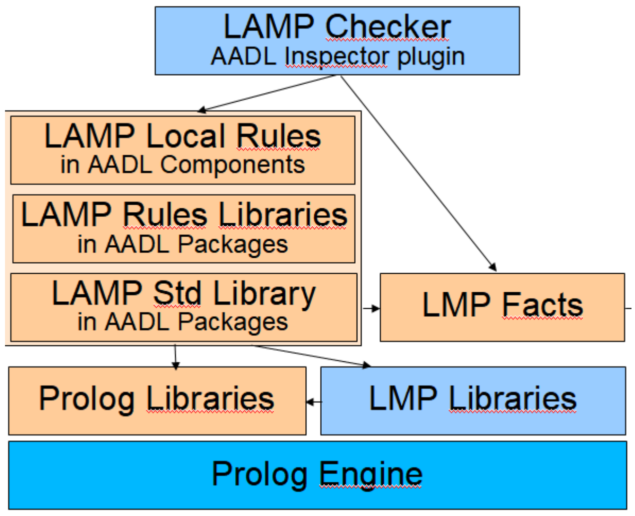


Figure 2: The LAMP stack

2.2. LAMP Annexes

LAMP annex subclauses can be located either within AADL Components or inside AADL Packages and can thus be directly interpreted while editing the AADL source text.

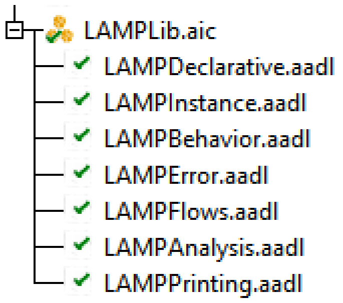
```
package Ellidiss::ERTS2020::paper26::e1
public

abstract A
  /* a LAMP annex at component level */
  annex LAMP {**
    /* standard prolog syntax */
  **};
end A;

/* a LAMP annex at package level */
annex LAMP {**
  /* standard prolog syntax */
**};

end Ellidiss::ERTS2020::paper26::e1;
```

It is recommended to restrict the use of LAMP annexes at component level to insert short processing goals or queries and to group longer or reusable rules into LAMP annexes at package level.

2.3. LAMP Standard Library

The LAMP Std Lib provides easy access to all the AADL model elements, including the Behavior and Error standard annexes. It supports not only the AADL declarative model, but also the AADL Instance model. It may also include more specific rules to gather information that comes from other sources, such as non-AADL models (requirements, source code…) or analysis tools outputs (simulation, …).

Practically, the LAMP Std Lib is implemented by a set of dedicated AADL Packages containing one or several LAMP annexes.



LAMPLib.aic
- LAMPDeclarative.aadl
- LAMPInstance.aadl
- LAMPBehavior.aadl
- LAMPError.aadl
- LAMPFlows.aadl
- LAMPAnalysis.aadl
- LAMPPrinting.aadl

Figure 3: The LAMP Standard Library

2.3.1. Access to core declarative model

Textual AADL specifications defines a declarative model. It is composed of a list of component definitions inside packages representing namespaces. The AADL declarative model describes a library of individual components with a single level of hierarchy (subcomponents).

The AADL declarative model is formally defined by a BNF grammar and can thus be parsed with usual techniques. As explained above, the LMP AADL parser generates a Prolog facts base representing all the elements of the AADL declarative model. For the LAMP user, this facts base represents the LMP facts or the AADL low-level API. This API is documented in wiki pages that can be accessed at the following URL:
http://www.ellidiss.fr/public/wiki/wiki/aadlparser.
A fragment of this low-level API is given below:

isPackage/3
isComponentType/6
isFeature/10
isComponentImplementation/8
isSubcomponent/9
isConnection/10
isFlowSpec/9
isFlowImplementation/8
isMode/7
isModeTransition/8
isAnnex/7
isPropertySet/2
isProperty/9

Table 2: AADL core low-level API

However, this low-level API may be too detailed for writing readable processing rules. That's why higher-level access to the AADL declarative model and its annexes has been defined in specialized libraries that are provided and documented in the AADL Inspector project browser. This set of LAMP rules is located in the `Declarative` sub-package.

```
package Ellidiss::LAMP::Declarative
public
annex LAMP {**
 getClassFeatures(…) :- …
 getClassSubcomponents(…) :- …
 getLocalProperties(…) :- …
… **};
end Ellidiss::LAMP::Declarative;
```

This high-level API consists of a set of Prolog rules using the low-level API and a set of general purpose utility functions. An example is shown below:

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*\
| getClassFeatures(Cla,Name,Cat,FCla):
| succeeds for each feature found in
| given component type and ancestors.
| - Cla (+): component classifier
|          qualified name
| - Name(-): feature identifier
| - Cat (?): feature kind in upper
|          case (e.g. 'IN DATA PORT')
| - FCla(-): feature classifier
|          reference
\*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

getClassFeatures(Cla,Name,Cat,FCla) :-
  getClassifier(Cla,P,T,_),
  getAncestorRec(P,T,'NIL',Q,U,_),
  isFeature(K1,Q,U,Name,K2,K3,FCla,…),
  concat(K2,' ',K3,' ',K1,Categ).
```

The AADL declarative model may also include optional annexes representing specialized sub-languages. The currently standardized sub-languages are the Behavior Specification annex and the Error Model annex. Both can be recognized by the LMP AADL parser which produces additional Prolog predicates for the low-level API.

2.3.2. Access to behavior model

The AADL Behavior Specification annex allows for adding behavioural details, typically to enrich the description of Threads and Subprograms. The behavior annex specifies state-transition automata that can act as pseudo-code for advanced timing analysis and simulation. This annex is formally defined in document SAE AS-5506/3.

As for the core language, the LMP AADL parser translates textual Behavior Specification statements into Prolog predicates. The most significant ones are shown below:

isBAVariable/9
isBAProperty/10
isBAState/7
isBATransition/9
isBACondition/9

isBADispatchExpression/10
isBADispatchTrigger/8
isBAAction/11

Table 3: AADL Behavior Annex API

Within the LAMP library, higher-level accessors have also been defined for the AADL Behavior Specification annexes. They can be found inside the BA2 sub-package.

```
package Ellidiss::LAMP::BA2
public
annex LAMP {**
 getBAVariables(…) :- …
 getBAStates(…):- …
 getBADispatch(…) :- …
 getBAComputation(…) :- …
…  **};
end Ellidiss::LAMP::BA2;
```

2.3.3. Access to error model

A similar approach has been followed for the other standardized AADL sub-language, the Error Model annex that is defined in document SAE AS-5506/1A. This annex, that is often called EMV2, covers many aspects required to follow a complete safety analysis process and support analysis techniques such as Failure Mode and Effects Analysis (FMEA) or Fault Tree Analysis (FTA).

The output of the LMP AADL parser for Error Model statements consists of another list of Prolog predicates:

isEMV2ErrorBehavior/4
isEMV2UseTypes/7
isEMV2UseBehavior/6
isEMV2Event/10
isEMV2State/9
isEMV2Transition/12
isEMV2Condition/11
isEMV2ConditionTrigger/10
isEMV2TransitionBranch/10
isEMV2ErrorPropagation/8
isEMV2ErrorSource/11
isEMV2ErrorSink/8
isEMV2ErrorPath/11
isEMV2OutgoingPropagation/11
isEMV2ErrorDetection/11
isEMV2CompositeState/9
isEMV2CompositeStateExpr/10
isEMV2CompositeStateElem/9
isEMV2ConnectionError/10

Table 4: AADL Error Model API

For the same reasons as the AADL core language and its Behavior Specification sub-language, the Error Model is more easily managed via the higher-level API that is implemented by Prolog rules in a EMV2 sub-package of the LAMP library.

```
package Ellidiss::LAMP::EMV2
public
annex LAMP {**
 getErrorTypes(…) :- …
 getErrorStates(…) :- …
… **};
end Ellidiss::LAMP::EMV2;
```

2.3.4. Access to instance model

As explained above, the AADL declarative model describes each component individually with a single level of hierarchy. It can be seen as a library of reusable component classifiers that may be instantiated several times in a given project and reused across different projects. In most model processing situations, there is a need to access each individual component instance independently. This alternate view of the same set of AADL components is called the AADL instance model.

The AADL instance model can be automatically derived from an AADL declarative model, provided that the top-level component is properly identified. This specific component is the root of the component hierarchy. Note that as opposed to the declarative model, the instance model format is not standardized by the AADL specification. Its implementation remains tool dependant. It cannot be used for model interchange between AADL tools, the declarative model must be used instead for that purpose.

The LMP framework and the LAMP standard library provide a set of rules to automatically detect the most likely root component, compute the instance model and give access to each individual instance through an appropriate API. The automatic selection of the root component can be overridden by setting a dedicated property.

At low-level, the main Prolog predicates that are produced by the LMP framework are shown in the table below:

isAADLRoot/6
isAADLInstance/12
isAADLConnection/12
isAADLBinding/7

Table 5: AADL Instance Model API

Note that each AADL component instance is identified by a unique identifier representing its position inside the instance tree with a dotted notation. The first element of each identifier is always *root*, which represents the top-level classifier of the declarative model from which the instance model is computed.

```
package Ellidiss::ERTS2020::paper26::e2
public
-- …
system implementation instance.i
subcomponents
  hw : processor hw.i;
  sw1 : process sw.i;
  sw2 : process sw.i;
properties
  actual_processor_binding =>
   (reference(hw)) applies to sw1, sw2;
end instance.i;

process implementation sw.i
subcomponents
  th1 : thread t;
  th2 : thread t;
  th3 : thread t;
end sw.i;

thread t end t;
-- …
end Ellidiss::ERTS2020::paper26::e2;
```

For example, from the AADL declarative model fragment presented above and considering that the root component is *instance_pkg::instance.i*, all the instantiated components will be:

category	identifier
system	root
processor	root.hw
process	root.sw1
thread	root.sw1.th1
thread	root.sw1.th2
thread	root.sw1.th3
process	root.sw2
thread	root.sw2.th1
thread	root.sw2.th2
thread	root.sw2.th3

Table 6: AADL Instance Model Identifiers

As before, the LAMP layer introduces a more compact set of accessors to facilitate the processing rules work for the developer.

```
package Ellidiss::LAMP::Instance
public
annex LAMP {**
 getRoot(…) :-
 getInstances(Id,…) :- …
 getProperties(Id,…) :- …
… **};
end Ellidiss::LAMP::Instance;
```

2.3.5. Access to foreign data

In addition to the information that can be directly (declarative model) or indirectly (instance model) extracted from the AADL specification, it is possible to complete the database by lists of Prolog predicates generated from foreign models or tools.

For example, the Marzhin simulator that is embedded in AADL Inspector produces Prolog predicates giving various details about the last simulation run. These raw data can then be processed to provide higher-level information for further analysis. The following fragment shows how information generated by simulation is made available to LAMP processing rule, such as computed response time for threads or observed port values at a given time.

```
package Ellidiss::LAMP::Analysis
public
annex LAMP {**
 getResponseTime(Id,Duration) :- …
 getPortValue(Id,Tick,Value) :- …
… **};
end Ellidiss::LAMP::Analysis;
```

Other possibilities consist in adding predicates coming from remote tools to perform cross-model processing. Examples of such foreign data could be as a list of requirements expressed in a ReqIF [16] file or system engineering components described by a SysML [17] model.

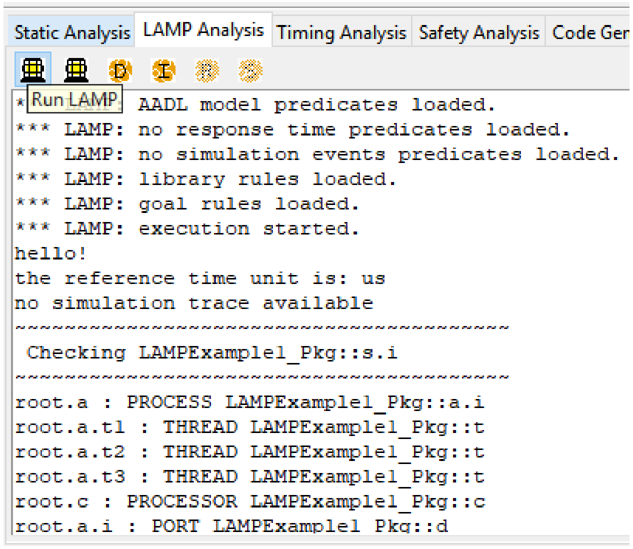
These foreign data are often serialized in XML or XMI files that can be parsed by the appropriate LMP parser to produce the corresponding list of Prolog predicates. When possible, direct generation of Prolog predicates can be implemented into the foreign tool to avoid the serialization and parsing steps. This is typically what is done for the modelling tools developed by Ellidiss, such as Stood.

2.4. LAMP Checker

The LAMP checker is a dedicated AADL Inspector plugin that is used to run LAMP programs. When the user pushed the *Run LAMP* button, the following sequence of actions is performed:

- The selected AADL files are parsed by the LMP framework and the corresponding predicates are loaded in the Prolog engine.
- Optionally, additional foreign data predicates are also loaded if they exist.
- The contents of all the LAMP annexes that are found inside the selected AADL files are concatenated together to build a single Prolog program that is loaded into the Prolog interpreter program area.

- The Prolog interpreter is launched and executes the Prolog queries found in the LAMP annexes declared at component level.
- The results can be printed in the LAMP checker console or stored in output files.



```
*** LAMP: AADL model predicates loaded.
*** LAMP: no response time predicates loaded.
*** LAMP: no simulation events predicates loaded.
*** LAMP: library rules loaded.
*** LAMP: goal rules loaded.
*** LAMP: execution started.
hello!
the reference time unit is: us
no simulation trace available
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Checking LAMPExample1_Pkg::s.i
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
root.a : PROCESS LAMPExample1_Pkg::a.i
root.a.t1 : THREAD LAMPExample1_Pkg::t
root.a.t2 : THREAD LAMPExample1_Pkg::t
root.a.t3 : THREAD LAMPExample1_Pkg::t
root.c : PROCESSOR LAMPExample1_Pkg::c
root.a.i : PORT LAMPExample1_Pkg::d
```

Figure 4: The LAMP Checker Tool

Note that like all the other AADL Inspector plugins, the LAMP checker itself is implemented in Prolog on top of the LMP framework.

3. Examples of use

This section provides an overview of the processing capabilities of LAMP that are illustrated by simple examples of use.

3.1. Model exploration

The simplest usage of LAMP consists in doing advanced introspection inside the overall AADL model that has been loaded. Using default Prolog syntax makes it easy to perform *SQL Select* like queries based either on the low-level LMP API or the higher-level LAMP Libraries.

```
package Ellidiss::ERTS2020::paper26::e3
public
-- …
abstract LAMP_query
annex LAMP {**
 /* query 1 */
 ( getInstances(Id,_,_),
   write(Id), nl );
 /* query 2 */
 ( getInstances(Id,'THREAD',_),
   write(Id), nl )
**};
end LAMP_query;
-- …
end Ellidiss::ERTS2020::paper26::e3;
```

This example uses one of the predicates of the LAMP standard library that gives all the component entities of the AADL instance model. Its first parameter is the unique instance identifier of the entity; the second parameter is the corresponding AADL category and the third one gives the qualified classifier name in the declarative model.

Running the first rule (query 1) will list all the component identifiers, whereas running the second one (query 2) will only list the identifiers of thread components. Similar filtering could use the third parameter to list all the instances of a given classifier. In practice, both queries will be executed in sequence.

3.2. Static analysis

LAMP programs can be used to enforce semantic rules within an AADL model. It is likely that many of these rules will already be hardcoded within the AADL tool, however it may be interesting to add custom rules that reflect specific methodological good practices or language restrictions (AADL subsets).

In the next example, the LAMP annex implements a methodological rule that will issue a warning message if it exits a thread that is not either periodic or sporadic. This could be for instance part of a Ravenscar profile compliancy verification process.

```
package Ellidiss::ERTS2020::paper26::e4
public
-- …
abstract LAMP_check
annex LAMP {**
 getInstances(Id,'THREAD',Cla),
 Prop = 'DISPATCH_PROTOCOL'
 getProperties(Id,Cla,Prop,Val),
 Val \= 'PERIODIC', Val \= 'SPORADIC',
 write('Warning'), nl
**};
end LAMP_check;
-- …
end Ellidiss::ERTS2020::paper26::e4;
```

This simple Prolog program can be understood as follows: “*get the unique identifier (Id) and classifier (Cla) of each thread in the instance hierarchy, AND get their dispatch protocol (Val), AND test if it is not periodic, AND test that it is not sporadic, AND write a warning message*”.

3.3. End to end flow latency analysis

More sophisticated verification programs can be elaborated thanks to both the intrinsic processing power of the Prolog language and the exhaustiveness of the LMP and LAMP model accessors.

Computation of end to end flow latency is one of these non-trivial analysis functions that are frequently requested by AADL users. AADL end to end flows are used to specify semantic links across a hierarchy of AADL components, from an ultimate data source to an ultimate data sink. Source and sink are usually a device or a thread.

The LAMP standard library includes a few facility rules to manage these complex objects. For instance, the rule `getEndToEndFlow(Id, Flow, Elems)` gives a Prolog list (`Elems`) containing all the model elements contributing to a given flow.

To compute the global latency of an end to end flow, it is necessary to sum up the response time of all the time-consuming model elements contributing to this flow. These are typically threads and connections. A fragment of the LAMP program that performs this computation is shown below:

```
package Ellidiss::ERTS2020::paper26::e5
public
-- …
abstract LAMP_flow
annex LAMP {**
 /* threads */
 addFlowLatencyContributor(Elem) :-
  splitName(Elem,G,X),
  isAADLInstance('THREAD',…,P,T,…,X,…),
  isFlowSpec(_,P,T,'NIL',H,_,_,_,_),
  memezTra(G,H),
  getMaxResponseTime(X,D,3),
  strToNum(D,N), cpt1Add(N), !.
 /* connections */
 addFlowLatencyContributor(Elem) :-
  isAADLBinding('CONNECTION',Elem,_),
  splitName(Elem,C,S),
  concat(S,'.VirtualLink.',C,X),
  getMaxResponseTime(X,D,3),
  strToNum(D,N), cpt1Add(N), !.
**};
end LAMP_flow;
-- …
end Ellidiss::ERTS2020::paper26::e5;
```

The `addFlowLatencyContributor/1` rule has an input parameter that represents one of the elements of an end to end flow. It is implemented by a disjunction of two cases that can be explained as follows: "*'test if the input parameter (Elem) corresponds to a flow (G,H) crossing a thread (X), AND get its response time computed by simulation (D,N), AND add it to a global variable), OR (test if the input parameter corresponds to a connection bound to a bus, AND gets the response time(D,N) of the corresponding message (X) on the bus, AND add it to the global variable)*".

This rule uses various LMP utility functions to manipulate strings and numbers, as well as the `getMaxResponseTime/3` predicate that provides the measured response time of each thread and bus message obtained by simulation. All these rules are also written in prolog within the LMP framework and the LAMP libraries.

3.4. Fault tree generation

Another useful utilization of LAMP is the implementation of model transformations to generate input data for a remote processing tool. An example of such a feature is the generation of a file complying with the Open PSA standard [5] that can be processed by Fault Tree Analysis (FTA) tools like Arbre Analyste [6].

Open PSA uses the XML syntax, but its contents are defined by a BNF grammar. The role of the LAMP program is to convert the appropriate AADL Error Model statements into Open PSA XML entities and attributes. The following example shows a fragment of this process, and more precisely how Fault Tree nodes are generated from AADL Error Model states.

```
package Ellidiss::ERTS2020::paper26::e6
public

system implementation  ControlSystem.i
subcomponents
 Sensors: system Sensors.i;
 Controlunit: system Controlunit.i;
 Actuators: system Actuators.i;
 Dashboard: system Dashboard.i;
 Network: bus Network;
annex EMV2 {**
 use behavior errorlibrary::failstop;
 composite error behavior
 states
  [ Dashboard.FailStop or
    Sensors.FailStop or
    ControlUnit.FailStop or
    Actuators.FailStop or
    Network.FailStop ]-> FailStop;
 end composite;
**};
annex LAMP {**
 /* inherited error behavior */
 getFTState(Class,Name,K,TS) :-
  getTypeAndImpl(Class,P,T,I),
  isEMV2UseBehavior(P,T,I,_,R,_),
  splitReference(R,L,B,_),
  isEMV2State(M,…,C,Name,TS,K,_),
  memezTra(L,M), memezTra(B,C).
 /* local error behavior */
 getFTState(Class,Name,K,TS) :-
  getTypeAndImpl(Class,P,T,I),
  isEMV2State(Q,U,J,_,_,Name,TS,K,_),
  memezTra(P,Q), memezTra(T,U),
  memezTra(I,J).
**};
end ControlSystem.i;
-- …
end Ellidiss::ERTS2020::paper26::e6;
```

The `getFTState/4` rule has one input parameter to specify the component classifier (Class) that must be explored, and three output parameters giving an error state name (Name), its kind (K) and the corresponding error type set (TS). As there are two separate sources of error states, within an imported library or within the component itself, the rule is implemented by a disjunction. It can be commented as follows: "*(check if the component inherits from a behavior described in a shared error model library, AND returns all the error states that are defined there), OR (return all the error states that are defined inside the local error annex library)".*

3.5. Security rules

In some situations, there a great interest that the verification rules remain closely attached with the input model. This is often the case with security rules for which specific security policies must be applied for a given project. LAMP can be used to implement these rules and include them inside the project packages.

An example of project specific security rules policy could be:

- *Sec_R1*: All components involved in a same end to end Flow must be at the same security level.
- *Sec_R2*: The security level of a component is the higher security level value associated with its Data ports.
- *Sec_R3*: When two components are connected via a shared Bus, they must comply with the No-Read-Up and No-Write-Down rules [7].

A fragment of the corresponding LAMP implementation is given below:

```
package ControlSystemAnalysis
public

annex LAMP {**
/* rule Sec_R1 */
checkFlowSecurity :-
 getRoot(R), getClassifier(R,P,T,I),
 getAncestorRec(P,T,I,Q,U,J),
 F = 'END TO END',
 isFlowImplementation(F,Q,U,J,E),
 concat('root.',E,F),
 getEndToEndFlow('root',E,M),
 getFlowSecurityLevels(M,[],L,0,N),
 N > 1,
 printMessageSec_R1(F,L).
checkFlowSecurity :- nl.

/* rule Sec_R2 */
checkMaxSecurityLevel :-
 getMaxSecurityLevel(X,L),
 printMessageSec_R2(X,L).
checkMaxSecurityLevel :- nl.

/* rule Sec_R3 */
checkNoWriteDown :-
 isAADLBusBinding(_,C,_),
 isAADLConnection(_,P,T,I,_,_,_,C,…),
 getConnectionEnds(P,T,I,C,Xs,Xd),
 getMaxSecurityLevel(Xs,Ls),
 getMaxSecurityLevel(Xd,Ld),
 Ls > Ld,
 printMessageSec_R3(C,Ls,Ld).
checkNoWriteDown :- nl.
**};
-- …
end ControlSystemAnalysis;
```

These rules use a set of utility rules that are not shown here because of the limited space. One of them is `getMaxSecurityLevel(X,L)` that returns the security level (L) of a given component identifier (X). An informal description of the implementation of rule Sec_R3 is: "*search all the connections (C) that are bound to a bus, AND get their source (Xs) and destination (Xd) ends , AND find the security level of each ends (Ls,Ld), AND test if the security level of the source is higher than the one of the destination, AND write an error message*".

Conclusion

Improving readiness of model verification techniques is a key concern to promote Model Driven Engineering and to guarantee successful development of software intensive critical systems. Making modelling and verification better integrated together contributes to the enhancement of the global development process.

Within the scope of an AADL project, the LAMP model processing solution brings all the benefits of formal logic programming with Prolog, the industrial return of experience of the LMP framework and a close and exhaustive interface with the AADL standard. Embedding verification or processing rules inside the input model is made easy in this context thanks to the ability to add customized annexes that remain transparent for the AADL environments that cannot interpret them.

All the features and examples that have been presented in this paper are implemented in the current AADL Inspector distribution at the time or redaction, i.e. version 1.7.1.

References

[1] AADL: Architecture Analysis and Design Language:
https://www.sae.org/standards/content/as5506c/
[2] “Model Verification: Return of Experience”, P. Dissaux and P. Farail, ERTS 2014.
[3] “Merging and Processing Heterogeneous Models”, P. Dissaux and B. Hall, ERTS 2016.
[4] AADL Inspector:
http://www.ellidiss.fr/public/wiki/wiki/inspector
[5] The Open PSA initiative:
http://www.open-psa.org/
[6] Arbre Analyste:
https://www.arbre-analyste.fr/en.html
[7] Combined security and schedulability analysis for MILS real-time critical architectures, I. Atchadam, F. Singhoff, H. N. Tran, N. Bouzid and L. Lemarchand, in 4th international workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS), Stuttgart, Germany, 2019.
[8] “Resolute: an assurance case language for architecture models”. A. Gacek, J. Backes, D. Cofer, K. Slind, M. Whalen. HILT 2014.
[9] OCL: Object Constraint language
http://www.omg.org/spec/OCL/
[10] Acceleo:
https://www.eclipse.org/acceleo/
[11] ATL: Atlas Transformation Language
http://www.eclipse.org/atl/
[12] “Expressing and Enforcing User-Defined Constraints of AADL Models”,Gilles O., Hugues J. ICECCS 2010, 337-342.
http://www.openaadl.org/ocarina.html#about-ocarina
[13] "Compositional verification of architectural models.", Cofer, D., et al. NASA Formal Methods. Springer Berlin Heidelberg, 2012. 126-140.
[14] Prolog language: ISO/IEC 13211-1, 1995.
[15] sbprolog: Stony Brook Prolog,
https://www.cs.cmu.edu/Groups/AI/lang/prolog/impl/prolog/sbprolog/0.html
[16] ReqIF: Requirements Interchange Format,
https://www.omg.org/reqif/
[17] SysML: Systems Modeling Language,
http://sysml.org