Model Verification:
Return of experience

P. Dissaux[1], P. Farail[2]

1: Ellidiss Technologies, 24, quai de la douane, 29200 Brest, France
2: Airbus Operations SAS, 316 route de Bayonne, 31060 Toulouse, France

Abstract

This paper introduces an original, although not new, technology that was initially developed to implement Model Verification tools.

This technology, called LMP, has been integrated in critical software design and analysis tools over the last twenty years and used in particular for the development of DO 178 certified embedded applications at Airbus.

Use of LMP has now been extended to support the various aspects of Model Processing, such as performing model queries, model verifications and model transformations. It is now used in particular for the implementation of the AADL Inspector product.

Introduction

Model Driven Engineering is now recognized as being a way to improve the development process of industrial critical software. This concern is especially strong for embedded and/or real-time software that are designed for avionics, space, military, railways, automotive and medical devices applications.

The goal of these technological changes is to enhance at the same time the quality and the productivity of the software development activities. However, applying models in the context of the development of critical systems implies that it is possible to verify that these models are correct and usable. These correctness and usability criteria are not absolute but obviously depend on the actual role of the model in the development process which defines their level of abstraction and level of completeness.

In the first section, the role of early verification to reach this goal is highlighted. In the second section, a specific model verification approach is presented. In the third section, concrete industrial return of experience in using this technology in a civilian avionics certification context is described. Then, generalized use of this verification technique for various model processing functions that are implemented in a commercial tool is also introduced. Finally, a few elements about the assessment of the LMP technology are provided in section five.

1. Model Verification

When Model Driven Engineering principles are applied to the traditional V cycle that is still in use in the context of large scale industrial projects, it mostly consists in a better formalization of the architectural design phases of the software development process.

Proper use of Models for these activities is well recognized as beneficial by enforcing engineers to well describe the architecture of the software before entering actual programming activities. The role of a Model is thus to offer a set of higher level abstractions, associated with corresponding textual or graphical notations that can improve many aspects of the software life-cycle:

- Requirements traceability
- Reuse of subsystems and libraries
- Collaborative development and subcontracting
- Testability
- Maintainability

However, all these benefits are quite often theoretical and depend on the actual integration of the modelling solution and tools within the industrial process and their adoption by the development teams.

Nevertheless the most promising improvements brought by Model Driven Engineering approaches is the ability to perform early verifications of the software application at a model level. Model Verification can indeed bring a high value added to the development process by increasing the confidence in architectural design choices and contribute to the certification process.

Moreover, efficient Model Verification activities are the first step to achieve before putting in place automatic code generation techniques. They provide a way to ensure the semantic compliance between the representation of the applicative software at a model level and its corresponding source code.

The semantic definition of a model is given by a set of more or less formally expressed structural, naming, legality and consistency rules, that can be enriched by more specialized constraints such as, for instance, those enforced by the scheduling theory for real time systems.

Model Verification activities in the development process thus consists in defining and implementing these rules with an appropriate technology, and applying them to the application being developed. The quality of the implementation of the Model Verification tools obviously becomes a key concern by itself.

The ideal approach would consist in implementing implicit Model Verification by using formal techniques for the specification of the model. Unfortunately, most current model implementation languages restrict this verification "by construct" to a very limited range of semantic rules. In addition some categories of rules, such as completeness, cannot be verified at any stage of the model construction. It thus becomes necessary to add dedicated Model Verification tools that work on an appropriate representation of the model to produce a compliance report for the complete set of semantic rules.

Another important characteristic of Model Verification activities is its needs to be perfectly adapted to the industrial development process. In order to avoid rejection by development teams and improve the benefit of the approach, the precise list of verification rules to be applied may need to be tuned with different levels of variability:

- Compliance rules with the core definition of the modelling language.
- Compliance rules with specific analysis facets (i.e. scheduling analysis, safety analysis, power consumption, …).
- Compliance with corporate or project specific methodological rules (quality insurance, certification process, …).

All these constraints express Model Verification requirements that are fully part of the critical software development process definition. It is thus mandatory that the corresponding Model Verification implementation technology ensures proper traceability towards these requirements.

Model Verification can also be ensured by creating a dedicated model transformation to a input language of an existing verification tool. The Model Verification implementation technology must thus ideally not only support constraints checking, but also be used to perform other Model Processing actions such as model queries and model transformations.

In the next section, an original approach to achieve this goal is presented.

2. Logic Model Processing (LMP)

2.1. LMP Overview

With Model Driven Engineering, a model is an instance of a meta-model. The meta-model expresses a first set of structural rules that will be verified "by construct". Such a meta-model is very often expressed with MOF [1] or Ecore [2] languages for UML [3] based models. However, traditional BNF [4] descriptions can play exactly the same role for text based models. In the area of Model Verification, several technologies such as OCL [5], ATL [6], QVT [7], Kermeta [8], TOM [9], REAL [10] have been developed to support model constraints checking or model transformations. Other solutions like EXPRESS [11] can encompass both the meta-model specification and its exploitations.

In this paper, we focus on an alternate approach that can be used to implement any model processing feature, including constraints checking and model transformations. This solution can also be used to create consistent automatic code and documentation generators.

This technology, known as LMP (Logic Model Processing), is an extension of a solution that was firstly developed twenty years ago to implement Hierarchical Object Oriented Design (HOOD) [12] rules checkers. Although the term "Model Driven Engineering" had not been invented at that time yet, HOOD is a graphical and textual language to formalize software architectural and detailed design. The HOOD Reference Manual also contains a list of design rules with which a correct model must comply. Initially promoted by the European Space Agency, HOOD still plays a important role in the software design activities of major European projects such as the Eurofighter and the Airbus family.

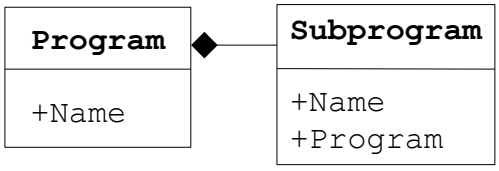
This approach is based on the use of the prolog language [13] to formally specify rules to be applied to an appropriate representation of the applicative model. This representation of the model is composed of Prolog facts (rules that are always true).

Prolog (Programmation Logique) 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 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.

In the context of Model Driven Engineering, these principles are applied as follow:

- The meta-model classes generate prolog fact definitions whose parameters names correspond to classes attributes name.
- An instantiated model consists in a populated prolog fact 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.

The following very basic example shows an illustration of LMP. Given the following meta-model expressing that a Program is Composed of Subprograms:



The corresponding LMP facts definition would be as follow:

```
isProgram(Name).
isSubprogram(Name,Program).
```

A set of models, instances of this meta-model could be represented by the following populated prolog facts base:

```
isProgram('P1').
isProgram('P2').
isProgram('P3').
isSubprogram('S1','P1').
isSubprogram('S2','P2').
isSubprogram('S3','P2').
```

An example of a Model Verification rule applying on such models could be:

R1: A Program must contain at least one Subprogram.

The LMP implementation of such a rule would simply be the query searching for all the cases that make this rule fail. Note that a comma between two prolog rules represents a logical AND operator:

```
ruleR1 :- isProgram(P),
          not(isSubprogram(_,P)).
```

Which would give us the following result:

```
P = 'P3'
```

This example is an illustration of the use of LMP for the implementation of constraints checkers. A similar approach can be followed to realize model transformations, code and documentation generators, and source code reverse engineering.

The most important drawback of this technology comes from possible bad performance issues that can however be minimized by using appropriate prolog coding rules.

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 or model to text transformations (transformation language).

These benefits are illustrated in sections 3 and 4 that describe industrial use of this technology, and then analysed in more details in section 5.

2.2. LMP Implementation

The LMP toolbox is composed of a set of executable files and libraries of predefined prolog rules. The prolog environment that is currently used is sbprolog [14]. With this environment, it is possible to operate on prolog source code (.pro files) or on pre-compiled byte code (.sbp files or data in memory) of the fact base and the rules base.

The LMP methodology implies that all the model processing rules are statically pre-compiled and loaded at run-time when the corresponding operation is required. One interesting property of the sbprolog byte code files is that they can be concatenated without needing any further compiling or linkage

action. It is thus easy to build libraries of modular model processing functions that can be assembled together in order to provide a composite behaviour.

As far the elaboration of the facts base is concern, it relies on the modelling tool that can be a graphical editor or a source text parser. In both cases, the facts base must be generated dynamically to reflect the actual contents of the model to be processed. The facts base generator can either produce prolog source code or sbprolog byte code. The latter option significantly improves the performance of the initialization phase of the model processing, as a simple concatenation of the facts and rules bases replace the dynamic complilation of the Prolog facts.

Another advantage of using the byte code format for the facts base is that it is not necessary to embed the sbprolog development environment into the end-user model processing product.

3. Industrial use at Airbus

For 20 years Airbus, has been using the HOOD method and the LMP technology, as part of the Stood [15] tool, to support the detailed design activity for all the embedded software developed for A330/340, A380, A400M and now A350 programs. The method and the technology is the same for all dependable level software (in the sense of DO178B [16]) : A, B, C or D.

Airbus processes all these design models with LMP in order to automatise following actions:

Documentation generation with respect to design documentation standard. This generated documentation is used at the same level as the model to control the design. This document becomes the reference for the customer and for the design activity. It contains all the Low level requirements with respect to the DO178B recommendations. It is an input of the overall traceability analysis made with the Reqtify tool.

Code skeleton generation with respect to coding standards. The LMP technology helps Airbus to customise the HOOD code generation for Ada, C or assembly languages. This automatic generation insures the traceability between Code and design which is required by the DO178B standard.

Source files generation used by source code verification tools. For example, skeleton of unit test file has been produced to initialise test procedures, but also design flows files as input of source code flow controller, and then function properties for formal code verification activities.

Design metric generation to follow-up design activity and design rules verification with respect to project design standards. The flexibility of the LMP technology helps Airbus to match the automatic verification tools with the projects specific rules : syntactic, semantic, consistency, completeness but also on data and control flows. Due to the LMP technology, the traceability between the rules and the prolog implementation is easy to do.

In order to have the complete benefits of this automation, some of the functionalities are qualified in the sense of the DO178B standard. The goal of this qualification is to avoid the manual verification of all outputs every time they are generated by the tool. So with respect to the DO178B standard and because these outputs are involved in verification activities, Airbus qualified such functions as verification tools. This is the case for the design rules checker for example.

4. AADL Model Processing

Another return of experience for LMP is its intensive use for the implementation of the AADL Inspector [17] commercial product. AADL Inspector is a software program that aims at importing a set of files containing textual AADL [18] models, and giving access to a variety of Model Processing tools that can be applied to such models.

AADL (Architectural Analysis and Design Language) is an international standard of the SAE (AS-5506). It defines a modelling language for the architectural description of software intensive real-time systems. The standard definition of AADL consists in a textual Bakus-Naur Form (BNF) syntax and a set of semantic rules expressed in natural language.

In the next sub-sections, we will show how the LMP technology is used to implement various Model Processing features in AADL Inspector.

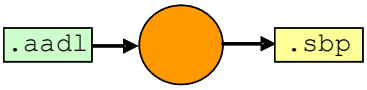
4.1. Model Query Language

The first step for all these Model Processing functions consists in transforming a textual AADL specification obtained by files concatenation into a binary representation of the corresponding prolog facts base in memory. This result is obtained thanks to a dedicated tool (aadlrev) that parses AADL specifications to generate a list of facts representing the AADL model:

```
isComponentType(...).
isComponentImplementation(...).
...
```

This initial phase can be seen as a purely syntactic transformation. Its goal is to make all the model elements reachable as prolog terminal predicates.

These predicates play two roles. The first one is to provide an appropriate organization of all the model entities (facts base) and the second one consists in offering an easy way to perform queries on the model.



The work-flow is very simple here: an AADL source text is transformed into sbprolog byte code thanks to the aadlrev tool represented by the orange circle. Note that aadlrev can also generate a textual prolog facts base.

As an example of use, let us consider the following fragment of a fact base produced after parsing an AADL textual model:

```
isComponentType('Pkg','PUBLIC','S','SYS
TEM','NIL','4').
isComponentType('Pkg','PUBLIC','X','PRO
CESSOR','NIL','51').
isComponentType('Pkg','PUBLIC','P','PRO
CESS','NIL','240').
isComponentType('Pkg','PUBLIC','T1','TH
READ','NIL','365').
isComponentType('Pkg','PUBLIC','T2','TH
READ','NIL','410').
```

the following statement is a query to get all the AADL components of the specified “Thread” category, and gives a value to each unbound parameter denoted by an upper case character. Note that the '_' character means that the corresponding parameter may take any value.

```
isComponentType(P,_,C,'THREAD',_,_)
```

According to the facts base described above, the result of this query will be as follows:

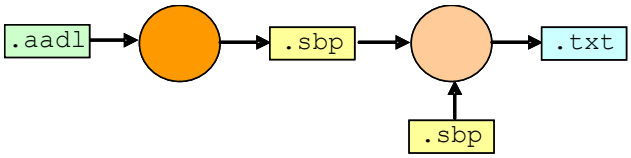
```
P='Pkg', T='T1'.
P='Pkg', T='T2'.
```

Such queries can be specified for each terminal predicates that describe the entire model. Moreover, a similar approach is also applied for the various sub-languages defined by some of the AADL Annexes, such as the Behaviour Annex and the Error Model Annex.

4.2. Model Constraints Language

Constraint rules on the model can then be defined by specifying logical combination of queries on the model. This approach enables easy implementations of semantic rules checkers, as illustrated below when applied to the AADL language.

The AADL standard defines a number of semantic rules that have to be verified in order to check the compliance of the model. These rules are organized in three categories: naming rules, legality rules and consistency rules.



The work-flow is now enriched with a second processing segment where the pink circle represents the sbprolog engine that takes the byte code facts base and the byte code rules base as input and produces a textual report.

For example, the implementation one of the naming rules that are defined in chapter 4.3 of the SAE AS-5506B document (current version of the AADL standard) is specified as follows:

(N1) The defining identifier for a component type must be unique in the namespace of the package within which it is declared.

A basic implementation of this rule in Prolog would be:

```
isComponentType(P,_,C1,_,_,L1),
isComponentType(P,_,C2,_,_,L2),
L1 \= L2, C1 = C2,
write('Error N1').
```

The first parameter (P) indicates that we only consider the components that are in a same package. The third one (C1 and C2) contains the name of the components which identity would be erroneous. The last parameter (L1 and L2) represents the unique identifier of the two components that would be different in case of an error.

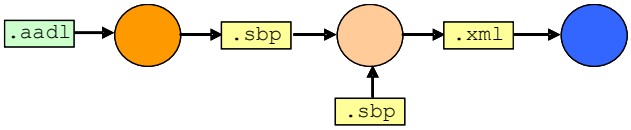
This implementation can thus be interpreted as follows: Find all the pairs of components in package P and whose identifiers differ and whose names are the same.

This approach is well appropriate to implement static rules checkers aiming at verifying the structural semantic properties of a modelling language. However, for more sophisticated analysis such as real-time scheduling analysis, simulation or safety analysis, it is required to use specialized tools. The next paragraph shows how the LMP technology can

also be used for interfacing with existing model analysis tools.

4.3. Interfacing with remote verification tools

Cheddar [19] is a real-time performance analysis tool that is developed by the University of Brest (UBO). Cheddar embeds its own internal language (Cheddar-ADL) that specifies the various entities and relationships that are required for applying the scheduling theory. In order to be able to perform scheduling analysis on an AADL model with Cheddar, it is thus necessary to implement a model transformation to generate an instance of its internal meta-model. The Cheddar ADL uses an XML format.



The work-flow is again enriched by a third tool (Cheddar) that is represented by the blue circle.

The initial and mandatory step in the implementation of such model transformations is the definition of the semantic mapping between the two sets of modelling entities.

A view of the mapping between AADL and the Cheddar ADL is shown in the table below:

Cheddar ADL	AADL
Processor	Processor subcomponent
Address Space	Process subcomponent
Task	Thread subcomponent
Resource	Data subcomponent
Buffer	Event Data port
Dependency	Data port

The following simplified implementation of the model transformation with LMP can be split in two parts. A first set of rules consists in producing the XML tags that describe the Cheddar entities:

```
insertProcessors :-
    openTag(1,'processors'),
    getProcessor(N),
    insertProcessor(N),
    closeTag(1,'processors').
```

And a second set of rules actually implement the mapping between the two sets of modelling entities:

```
getProcessor(Name) :-
    isComponentType(_,_,Name,'PROCESSOR
      ',_,_).
```

The terminal rules must be predicates belonging to the facts base produced by the AADL parser.

A similar transformation has been developed to build an interface with the Marzhin [20] real-time simulator and several others are under development like the interfaces to the Fiacre [21] verification tool-chain, to the Compass [22] safety analysis tools and Polychrony [23].

4.4. Homogeneous transformations

A particular case of model transformation is when the source and target models are both instances of the same meta-model. Such homogeneous transformations can be realized with any language, but they are especially easy to implement with LMP.

Such transformations are composed of the following elements that must be plugged together:
- The AADL parser producing the facts base (aadlrev)
- AADL to AADL mapping rules.
- The AADL unparser generating textual AADL again from this facts base.

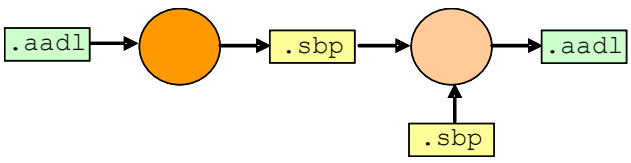
The simplest AADL to AADL mapping is the “identity” relation where the output predicate is identical to the input one that is produced by the AADL parser.

```
isComponentType(Package,Scope,Name,Cate
gory,Ancestor) :-
  isComponentType(Package,Scope,Name,
    Category,Ancestor,_).
```

Note that these two sets of predicates can be distinguished by their number of parameters. The unparser is a code generator whose output syntax complies with the AADL BNF:

```
insertComponentType(P,S) :-
  isComponentType(P,S,Name,Category,_),
  write(Category), sp, write(Name), nl,
  ...
  write('END'), sp, write(Name), sc, nl
```

The AADL to AADL transformation rules and the AADL unparser rules can then be concatenated together to provide a single rules base file.



The work-flow associated with such homogeneous transformations is as shown above, where the orange circle represents the aadlrev parser and the pink circle the sbprolog engine.

The result is a similar AADL textual specification that differs from the source one by a few text formatting styles. It can thus be used to implement a “pretty printer”.

A most interesting application consists injecting additional prolog facts to locally amend the original model. An example of use of this technique is given by the real-time properties editor of AADL Inspector.

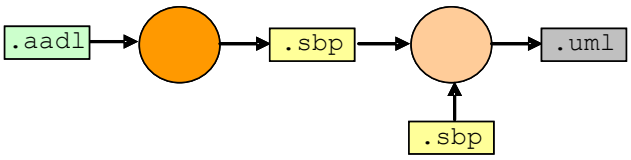
The AADL to AADL mapping rules are then lightly modified in order to take into accounts a new set of facts that enrich the original fact base. The effect is the same as the “identity” transformation except for the few changed properties:

```
isProperty(...) :-
  isProperty(...,_);
  isUpdatedProperty(...).
```

Note that a semicolon between two prolog rules represents a logical OR operator.

4.5. Heterogeneous transformations

The general case for model transformations is of course when the input and output meta-models differ. If we consider for instance a transformation from AADL to its equivalent representation in UML MARTE [24], the implementation process that is presented in section 4.3 can be applied.



The work-flow consists in parsing the textual AADL specification with aadlrev (orange circle), then to apply the resulting facts base and the transformation rules base to the sbprolog engine (pink circle). The transformation rules must include in that case the AADL – MARTE mapping as well as an XML generator complying with the UML meta-model.

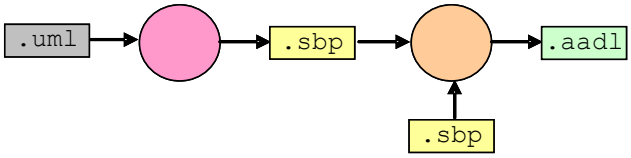
The AADL - MARTE mapping is specified in the OMG standard itself (annex A2). For illustration purpose, a small fragment of this mapping is shown in the table below:

UML MARTE	AADL
Package	Package
Block (SysML)	System
memoryPartition	Process
swSchedulableResource	Thread
hwProcessor	Processor
HwMemory	Memory

The mapping implementation in prolog can then look like as follows:

```
insertMemoryPartition(P) :-
  isComponentType(P,_,N,'PROCESS',_),
  write(Category), sp, write(Name), nl,
  ...
  write('END'), sp, write(Name), sc, nl
```

Interestingly, it is also possible to apply the reverse mapping, i.e. to build a UML MARTE to AADL transformation, using a similar implementation.



The work-flow now consists in parsing the XML file representing the UML MARTE model. This parsing is achieved by xmlrev, another tool from the LMP toolbox that works in a similar way as aadlrev excepts that it produces a fact base describing the structure of an input XML file:

```
isXMLTag(...).
isXMLAttribute(...).
```

It is then necessary to implement the reverse mapping and merge it with the AADL unparser that has been introduced in section 4.4.

```
isComponentType(Pkg,'PUBLIC',Name,'PROC
ESS','NIL') :-
  isXMLTag(X,'xmi:XMI','NIL',_),
  isXMLTag(M,'uml:Model',X,_),
  isXMLTag(P,'packagedElement',M,_),
  isXMLAttribute(P,'packagedElement',
    'xmi:type','uml:Package',_),
  isXMLAttribute(P,'packagedElement',
    'name',Pkg,_),
  isXMLTag(S,'packagedElement',P,_),
  isXMLAttribute(S,'packagedElement',
    'xmi:type','uml:Component',_),
  isXMLAttribute(S,'packagedElement',
    'name',Name,_),
  isXMLAttribute(S,'packagedElement',
    'xmi:id',I,_),
  isXMLAttribute(_,
    'SW_Concurrency:MemoryPartition',
    'base_Classifier',I,_).
```

5. LMP assessment

In this section, we attempt to provide a few elements about the assessment of the LMP technology. We firstly give a more exhaustive overview of the various model processing rules sets that have been realized using the LMP technology, an then we provide a few evaluation criteria for each ISO 9126 standard quality assurance characteristic, which are:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

5.1. Overview of the main LMP realizations

We presented in sections 3 and 4 various kinds of model processing transformations that have already been developed using the LMP technology. This approach has also been followed for the purpose of other products, such as Adele [25] and TASTE [26].

The table below gives a non exhaustive overview of the current contents of the LMP model processing rules library that has been developed by Ellidiss and the name of the product they are embedded in.

Rules	Product
AADL Consistency	AADL Inspector
AADL Legality	AADL Inspector
AADL Naming	AADL Inspector
MARTE to AADL	AADL Inspector
AADL Metrics	AADL Inspector,Taste
AADL to Cheddar	AADL Inspector,Taste
AADL to Marzhin	AADL Inspector,Taste
AADL to AADL	AADL Inspector,Taste
AADL to TasteIV	Taste
AADL to TasteDV	Taste
TasteIV to SMP2	Taste
SMP2 to TasteIV	Taste
AADL to HOOD	Stood
Ada to HOOD	Stood
C to HOOD	Stood
HOOD to AADL	Stood
HOOD to Ada	Stood
HOOD to C	Stood
HOOD to C++	Stood
HOOD Checker	Stood
AADL to Adele	Adele
AADL to Compass	Under development
AADL to Fiacre	Under development
AADL to Signal	Under development

This list should be extended by the specific sets of rules that have been developed by the end users of these products, like Airbus.

5.2 Functionality

One of the base principle of the LMP approach is “one rule per requirement”. This principle cannot always be achieved but together with the declarative style of the prolog language, it enforces a good traceability between the implementation and the desired functionalities.

Although the main products that are encompassing the LMP technology are not distributed under an Open Source license, the prolog source code of the model processing rules can be made available to the end user, so that he can actively contribute to their realization, which increases the relevance of the developments.

5.3. Reliability

The clear and rigorous separation between the facts and rules bases ensures that the execution of the program will not impact the input model.

Moreover, the exclusive use of Boolean logic for the implementation of the model processing rules increases the ability to verify its correctness.

5.4. Usability

As far the end user is concerned, the LMP based features are fully hidden by the Graphical User Interface of the product in which they are integrated (e.g. Stood, Adele, TASTE, AADL Inspector).

For the tool administrator, configuration of LMP based features is made easy thanks to the modularity of the implementation.

The job is harder for the LMP features development teams. Although the prolog language is a standard, the realization of relevant and efficient sets of rules require a very good expertise in this technology.

5.5. Efficiency

Prolog programs are often criticized for their supposed bad performances. It is true that minor changes in a prolog program may have a major impact on its efficiency. However, with an appropriate skill, it is most of the times possible to solve these issues.

The memory footprint of the LMP programs is very low when compared to their equivalent realizations in the Eclipse/Java world. The size of a complete rules base is rarely more than 250 kilo bytes to which must be added the size of the facts base. The size of the facts base is about the same as the size of the serialized form of the model to be processed. This

sum represents the memory size that is required for the program area. An equivalent memory allocation is required for the stack area. At the end, most of the times, a few mega bytes of memory are sufficient.

Timing efficiency is more difficult to control, but the potential problems can be solved during the development phase with proper program profiling analysis. The pure declarative programming style sometimes imply that some time consuming rules are called several times. This can be avoided for instance by dynamically asserting new predicates that store intermediate results. An improvement of the timing performance has also been obtained by the direct generation of prolog byte code which reduces the duration of the initialization phase.

For instance, building an basic instance hierarchy of 5 processors, 22 processes and 123 threads from a 12 000 lines AADL source file takes around 20 seconds on a two year old low cost laptop.

5.6. Maintainability

The first realizations based on a preliminary version of the LMP technology have been elaborated in the mid nineties for the implementation of HOOD rules checker and code generators in the Stood product.

These features have been maintained continuously during nearly twenty years, and used by many large scale industrial projects.

For the most recent LMP realizations, Ellidiss has set up an integrated development environment that facilitates the graphical design of new rules bases and the reuse of modular libraries. This environment also includes a sbprolog byte code generator and design documentation facilities.

5.7 Portability

The use of a standard programming language minimizes the effort to ensure the portability of the model processing functions to another run-time environment.

In our case, we currently use the sbprolog run-time environment, whose C source code is available to ensure its portability to a non already supported platform, if required. Currently, LMP is supported by Ellidiss for Windows, Linux, Solaris and MacOs environments.

Finally, the sbprolog byte code has specifically been specified to ensure a complete portability of the facts and rules bases across the various supported platforms.

Conclusion

This paper highlights the importance of Model Verification activities in industrial critical software development processes.

It also introduces an original technology developed by Ellidiss and called LMP, for the implementation of such Model Verifications that has been extended to cover the more general need of Model Processing.

This paper finally provides feedback on practical use of this approach in major industrial avionics programs and in the realization of a commercialized software tool. Presented return of experience mostly concerns HOOD and AADL modelling languages and the LMP technology is implemented by a lightweight standalone toolbox. LMP can however be applied to any kind of meta-model and adapted to any modelling framework.

References

[1] MOF: Meta Object Facility
http://www.omg.org/mof/
[2] Ecore: Eclise Modeling Framework Core
http://www.eclipse.org/modeling/emf/
[3] UML: Unified Modeling Language
http://uml.org/
[4] BNF: Backus Naur Form
[5]OCL: Object Constraint language
http://www.omg.org/spec/OCL/
[6] ATL: Atlas Transformation Language
http://www.eclipse.org/atl/
[7] QVT: Query View Transformation
http://www.omg.org/spec/QVT/1.0/
[8] Kermeta:
http://www.kermeta.org/
[9] TOM:
http://tom.loria.fr/wiki/index.php5/Main_Page
[10] REAL:
http://www.openaadl.org/ocarina.html
[11] EXPRESS: ISO 10303-11. STEP Part 11 : EXPRESS Language Reference Manual, 1994.
[12] HOOD: Hierarchical Object Oriented Design
http://www.esa.int/TEC/Software_engineering_and_standardisation/TECKLAUXBQE_0.html
[13] prolog: ISO/IEC 13211-1, 1995
[14] sbprolog: Stony Brook Prolog
http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/prolog/sbprolog/0.html
[15] Stood:
http://www.ellidiss.com/products/stood/
[16] DO178:
http://www.rtca.org/
[17] AADL Inspector
http://www.ellidiss.com/products/aadl-inspector/
[18] AADL: Architecture Analysis and Design language

http://www.aadl.info/
[19] Cheddar:
http://beru.univ-brest.fr/~singhoff/cheddar/
[20] Marzhin
http://www.ellidiss.fr/public/wiki/wiki/inspector
[21] Fiacre:
http://projects.laas.fr/fiacre/home.php
[22] Compass:
http://compass.informatik.rwth-aachen.de/
[23] Polychrony:
http://www.irisa.fr/espresso/Polychrony/
[24] MARTE:
http://omgmarte.org/
[25] Adele:
http://gforge.enseeiht.fr/projects/adele/
[26] TASTE:
http://taste.tuxfamily.org/