Architecture-Driven Development of Embedded Systems with ACOL

Dries Langsweirdt, Nelis Boucké, Yolande Berbers
Department of Computer Science
K.U.Leuven
Leuven, Belgium
{dries.langsweirdt, nelis.boucke, yolande.berbers}@cs.kuleuven.be

Abstract—Architecture-Driven Development of embedded systems involves finding the right trade-off between multiple non-functional properties at the model level. In this paper we define ACOL: a model annotation language with at its core the combination of analysis, constraint and optimization expressions. This combination results in a powerful framework for efficient architectural design space exploration, usable from the earliest phases of embedded system design. We use AADL as an example to demonstrate how ACOL can be embedded in component-based Architecture Description Languages. The functionality of ACOL is illustrated with several use cases.

I. INTRODUCTION

The continuous growth of complexity and reliability needs faced in the embedded domain made it an early adopter of Architecture-Driven Development (ADD). Several examples are available from the literature ([1], [2]). Although no uniform definition is present, we informally define ADD as the process in which formal architectural models are used as the primary information carrier during system design and realization. The formal nature of these models renders them suitable for analysis, system integration and code-generation. Architectural models are typically defined using Architecture Description Languages (ADL) such as EAST-ADL[3], Wright[4] and AADL[5]. ADD can be thought of as a special case of Model-Driven Engineering (MDE)[6], where the same modeling formalism is retained as an overlay throughout requirements analysis, design, implementation and integration.

Creation of architectural models preferably starts as early as possible during development: at the boundary between the requirements analysis and design phases, where design space exploration takes place. Finding the right trade-offs during design between the different functional and non-functional properties is key. It determines the success of the system architecture and often the project as a whole, as it avoids expensive development iterations.

It is noted however, that current ADD techniques and tools primarily focus on the development phases logically following design: implementation and integration, where verification takes place. This claim is associated with the following observations:

- Most analysis and verification techniques require sufficiently complete architectural models and detailed information, unavailable during the early phases of development.
- Quantification of non-functional properties frequently relies on project-specific calculations and context, for which appropriate tool-support is nonexistent. Implementation of custom, in-house analysis tools during the design phase consumes time from the critical development path, and is therefore highly undesirable.
- Most analysis and verification tools operate by extracting specific information from the architectural model, related to one or a set of closely related non-functional aspects ([7], [8], [9], [10]). Non-functional properties are essentially analyzed in isolation, making it hard to optimize along different axes simultaneously. This results in an inefficient trade-off process.

Our solution to these problems lies in the definition of the language ACOL. ACOL allows the annotation of architectural model elements with specifications that combine analysis, constraint and optimization expressions. These three components together define a framework in support of the early ADD design phases, and ease the transitions between different levels of architecture refinement. ACOL can be used in concert with architecture evaluation methods, such as ATAM[11], providing a more formal basis for early discovery and resolution of architectural trade-offs.

We first elaborate on the design principles that underpin ACOL. The Architecture Analysis and Design Language (AADL)[5] is subsequently used to illustrate how these principles can be transformed in a concrete ADL-embeddable language.

II. ACOL DESIGN PRINCIPLES

We focus on components and connections as they constitute the fundamental modeling aspects in the majority of the ADL population [12]. Components model the elements from which a system is built, and define the context for specification of connections, configurations and non-functional properties. ACOL allows annotation of individual components and connections of the architectural model, with a combination of expressions in three categories:

- Analysis expressions allow to quickly define new analyses for custom non-functional properties. Specification of analyses on the model allows an incremental approach in
parallel with the construction of the model itself. It also provides transparency on the functionality of the analysis, as the user is not shielded from its implementation. Analysis expressions serve furthermore a role in keeping the model consistent, by updating derived properties. The execution time of a software component, based on its execution cycles and the frequency of the processor it is bound to, is an example of a custom and derived non-functional property.

- **Constraint expressions** guard the allowed values of the element’s non-functional properties. Constraint expressions logically follow the analysis expressions, as the results of analysis calculations are used as input. Constraints typically result from requirements analysis. An example is the maximum allowed power consumption of a component.

- **Optimization expressions** formulate optimization directives for one or several non-functional properties of the model element. A property can be optimized through minimization or maximization. For example, we could express the desire to minimize the total power consumption of a component.

A sequence of related ACOL expressions is grouped in an ACOL implementation, as shown on the high level metamodel in Fig. 1. Each ACOL implementation conforms to exactly one ACOL interface, which defines its view on the model through ports. When an ACOL implementation is associated with a model element, each port of its interface must be mapped to a model element conforming to the type of the port. The mapping is restricted by the obligation to map ports only to elements visible from the annotated element, on the same or lower layers in the system hierarchy. This measure favors correct element annotation in terms of layering and information locality. The separation of interface and implementation, combined with annotation by mapping, allows for the construction of reusable ACOL libraries.

ACOL finds its effectiveness in the combination of local annotation of model elements with global resolution of the constraints and optimizations. To illustrate this, consider an architectural model specification with a processor component and a set of thread components bound to it. The power consumption of the processor is at least related to its frequency. This dependency can be quantified through one or more ACOL analysis expressions, grouped in an ACOL implementation and mapped to the processor component through the appropriate ACOL interface. As such, the power consumption of the processor becomes a derived property, appropriately updated when the frequency changes. Based on requirements and physical limitations, we add several constraints to this ACOL implementation on the allowed range of operating frequencies and maximum power consumption, together with an optimization expression on minimizing this power consumption. The complete implementation could be specified as follows:

```
analysis
Scaled := Freq/5;
Power := (3*Scaled)^2;

constraints
Fail when {Freq < 100 MHz};
Fail when {Freq > 500 MHz};
Fail when {Power > 20 W};

optimizations
Minimize => Power;
```

Evaluation of this implementation in isolation would reveal the dependency between frequency and power, resulting in minimization of the frequency until the lower bound of the allowed range is reached. That is, until one of the constraints fails. This optimum can shift if new ACOL specifications are added to the model.

Suppose we annotate the thread components with a calculation of their execution time, as stated above, and constrain this value to be lower than their period, avoiding reentrant behavior:

```
analysis
F := Processor.Freq;
ExecTime := WCET(Cy)/F(Hz);

constraints
Fail when {ExecTime > Period};
```
Because ACOL specifications are solved globally, the optimum frequency value now becomes a function of both the constraints on the processor component, as well as the frequency related constraints on its threads. The frequency can be minimized until none of the constraints in this group fail. Note however, that the model components were annotated independently. It is possible that the minimum frequency needed to satisfy all execution time constraints is in conflict with the constraints on maximum power consumption or allowed frequency range of the processor. At that point, the optimization directive is unable to find a solution, and it is said that the model fails in solving the trade-off between the different properties. Architectural changes are needed, such as changing the processor type or merging threads.

ACOL is a set of principles, founded on the metamodel shown in Fig. 1. These principles are translated to a specification language with a concrete syntax aiming for tight integration with a specific ADL. This significantly lowers the acceptance barrier, as it avoids constructs alien to the designer and the modeling context. Conformance to the host ADL is valued higher than the one time effort needed to map ACOL to a different ADL. We refrain here from expanding the given metamodel to the complete abstract syntax of ACOL, but illustrate the alignment with the AADL metamodel and concrete syntax by example.

III. BACKGROUND ON AADL

In this section, we briefly introduce relevant AADL concepts in the interest of the remainder of this paper. Fig. 2 illustrates the graphical syntax of AADL and depicts an example model of a DSP system with a receiver, a transmitter, two redundant processing units and a common bus. The processing units contain a processor, a process and two threads. Each of these system elements maps to one of the predefined AADL component categories for modeling software (thread, subprogram, process, data), hardware (processor, bus, device, memory) and hybrid (system) components. An AADL component is specified by its type and zero or more implementations. For example, component PrimeSystem shown in Fig. 2 could be modeled as follows:

```plaintext
-- type
system PrimeSystem
features
  Ping : out data port;
end PrimeSystem;

-- implementation
system implementation PrimeSystem.Impl
subcomponents
  P1 : process P1.Impl;
P1U : processor P1U.Impl;
properties
  Clock_Jitter => 3 ms;
annex Error_Model {**
  Model => Error::Ex1.Basic;
  Occurrence => fixed 0.9
  applies to error Data_D;
end PrimeSystem.Impl;
```

Fig. 2. Example DSP system modeled with AADL.

Type features define the interface of a component, realized in its implementations. Component interactions are modeled using connections between their features. Subcomponents support hierarchical decomposition of the modeled system. Types and implementations are extensible: an extending type inherits its parent features, refines them or adds new ones. Similar rules apply to extending implementations. Model components can be annotated with properties, expressing relevant non-functional aspects such as Period, Latency and Weight. Property definitions (Period) have a type (Time) defined in function of the fundamental AADL property types (Aadlinteger). Properties optionally carry a unit, defined as a set of unit literals with explicit scaling relations (Ms, Sec, Hour). Extending the core AADL specification is possible using two predefined mechanisms: by specification of new properties in property sets, and by definition of annexes. Annexes enable the controlled extension of AADL with domain specific languages, having different syntax and semantics. Annex clauses can be specified in an annex library, or added to a component type or implementation. The listing above illustrates the annotation of PrimeSystem.Impl with an example Error Model Annex clause. For a more complete discussion on AADL, the reader is referred to [15].

IV. ACOL TO AADL MAPPING

As the mapping of ACOL intends to integrate the ACOL principles and metamodel with the host ADL, significant parts of the core AADL specification were re-used, such as the type system, annex extension mechanism and notion of libraries. Introduction of alien concepts was avoided as far as possible. ACOL is defined as an AADL annex.

A. Type System

ACOL is a statically and strongly typed language with partial type inference. The fundamental property types of AADL are inherited with minor adjustments. See Table I. Of
TABLE I
ACOL-AADL TYPES

<table>
<thead>
<tr>
<th>Primitive Types</th>
<th>Aadlinteger, Aadlreal, Aadestring, Aadlboolean, Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite Types</td>
<td>Range, Enumeration, List</td>
</tr>
<tr>
<td>Object Types</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Super-Type</td>
</tr>
<tr>
<td></td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>Connection</td>
</tr>
<tr>
<td></td>
<td>Flow</td>
</tr>
<tr>
<td></td>
<td>Feature</td>
</tr>
</tbody>
</table>

the primitive types, Classifier is excluded as it conflicts with the ACOL annotation mechanism (see section IV-E). Some of the AADL meta-model objects [16], conveniently called object types, are explicitly available as type in ACOL. Units are seen as attributes of the types Aadlinteger and Aadlreal.

B. ACOL Libraries
ACOL specifications are defined in AADL annex libraries. These libraries contain the ACOL interfaces, called types, and their implementations, similar to the AADL specification of components. The ACOL notion of ports is aligned with AADL features, while ACOL implementations define the analysis, constraint and optimization expressions. These ACOL library implementations can be instantiated and assigned to an AADL model component by mapping the abstract features to concrete model elements, as stated before.

```
package Library_Name
  public
    annex acol {**
    [acol types]
    [acol implementations]
    **};
  end Library_Name;
```

C. ACOL Types
ACOL types contain zero or more feature expressions. Similar to a function signature, features control the flow of information in and out of the ACOL specification. Features define both the direction and the type of the information available in its implementations. As an example type definition, consider:

```
acol Type_A extends Type_B
  features
    F1 : in list of component;
    F2 : in out property Frequency;
    F3 : out property Time;
  end Type_A;
```

The direction of a feature is specified with in, out or in out, respectively indicating read, write or read/write access to model properties. The feature type can be an AADL property type, a property definition or the name of an ACOL object type. Similar to the AADL core specification, ACOL types are extensible. To an extending type, new features can be added and inherited features refined. ACOL object super-types are more general than their sub-types, and a property definition is equally or more specific than a property type. It is thus allowed to refine the previously considered features to 'F1 : in list of thread;' and 'F3 : in out property Period;' respectively.

D. ACOL Implementations
ACOL implementations contain analysis, constraint and optimization expressions grouped in three distinct sections. The order of definition is strict, but each section is optional. Feature names are available throughout the implementation and represent the only visibility the expressions have on the AADL model. Property sets are an exception, as they are visible throughout the ACOL specifications. Implementations are extensible, but refinement of expressions is not possible. As an example, consider the hypothetical implementation Impl_A of Type_A as defined above:

```
acol implementation Type_A.Impl_A
  analysis
    P_List : Time(ms) := F1.Period;
    P_Sum := P_List'Sum;
    Pow : Power(W) := F2(Hz) * 2.21;
    F3 := 5 ms;
  constraints
    Fail when {P_Sum < 10 ms};
    Warn when
      {Pow >= Prop::Max_Power - 3};
  optimizations
    Minimize =>
      F2 delta Prop::Freq_Delta;
  end Type_A.Impl_A;
```

1) Analysis: The analysis section is a listing of variable assignments. The order of expressions is important, as a variable is only visible after it is assigned a value. Analysis expressions use an Ada inspired syntax:

```
<variable_name> :
  [:: <variable_type>]
  [([<unit_literal>])] :=
  (<value_expression> | <select_expression> | <function_expression>);
```

Explicit typing is unnecessary when the type is inferable from the expression. Examples are V := 3; and P_Sum := P_List'Sum, where V evaluates to Aadlinteger and P_Sum to property type Time. When more rigorous type control is needed, or when units are in use, variables can be typed by referring to the property type or -definition they comply to. Value scaling is possible by providing a unit literal. For example, Pow : Power(W) := F2(Hz) * 2.21; scales feature F2 of property type Frequency to unit literal
and multiplies this value with 2.21 to obtain a value with property type Power scaled to unit literal W. The name of a feature with direction out or in out must appear exactly once on the left-hand side of an analysis expression, as it indicates the updating of a property. Analysis expressions are categorized as follows:

- **Value Expressions** use a combination of feature names, previously defined variables, arithmetic-, logic- and unitary operators to calculate a value. All analysis expressions of the example implementation Type_A.Impl_A listed above are value expressions. The unitary operators are: property calls (F1.Period) and operation calls (P_List’Sum). A property call on a list of object types results in a new list with the individual property values, given that each object type defines the property. An important operation call is the Iterator, enabling easy iteration over discrete range types and lists:

\[ I := \text{Processor\_List’Iterator}; \]
\[ I.\text{Period} := 3 \text{ ms}; \]

- **Select Expressions** act exclusively on lists. A new list is constructed by selecting elements from an input list by means of boolean expressions. As an example:

\[ \text{Reentrant\_Threads} := t \text{ in Thread\_List} \]
\[ \text{where} \{ t.\text{Deadline} > t.\text{Period} \}; \]

- **Function Expressions** allow to delegate advanced calculations to external methods or functions, defined in a classic programming language such as Java or C. For example:

\[ \text{Schedulable} : \text{aadlboolean} := \]
\[ \text{function} \]
\[ \"\text{be.distrinet.Analysis.Scheduling}\" \]
\[ \text{with} \{ \text{Priority\_List, Utilization\_List} \}; \]

2) **Constraints:** The constraints section lists constraint expressions in an arbitrary order, with the following syntax:

\[ \text{\langle Fail|Warn\rangle when\ \langle not\rangle\ \{\text{boolean\_expression}\}}\]
\[\langle boolean\_operator\rangle\]
\[\langle not\rangle\ \{\text{boolean\_expression}\}\}]*;

Fail and Warn indicate which action is executed when the logic combination of boolean expressions evaluates to false. Fail indicates that a critical system requirement is violated. The intention of Warn is to inform the user about a value coming close to the border of its allowed range. Warnings are useful to indicate properties that need special attention during the model to realized system transition.

3) **Optimizations:** The optimizations section lists optimization expressions in an arbitrary order, with the following syntax:

\[ \text{\langle Minimize|Maximize\rangle => \langle feature\_name\rangle} \]
\[\text{[delta <value\_expression}>]; \]

Only features with a property type based on the fundamental AADL types Aadlinteger and Aadlreal are eligible for optimization, as it makes semantically no sense to optimize other types. Optimization expressions optionally define a delta, when it is not possible to optimize the property value continuously. An example is a processor with a minimum increment between two adjacent clock speeds.

E. Model Annotation

All AADL model components can be annotated with one or several ACOL library specifications. As an example, consider the association of ACOL implementation Type_A.Impl_A from section IV-D with system implementation PrimeSystem.Impl from section III as follows:

\[ \text{system\ implementation\ PrimeSystem.Impl} \]
\[ \text{\-- other declarations} \]
\[ \text{-- ACOL\ annex\ clause} \]
\[ \text{annex\ ACOL\ \{**} \]
\[ \\text{ACOL => MyLib::Type\_A.Impl\_A} \]
\[ \\text{\{ F1 <= Self’Subcomponents;} \]
\[ \\text{F2 <= PU1.Clock\_Speed;} \]
\[ \\text{F3 => Self.Clock\_Jitter\};} \]
\[ \text{**}}\}; \]
\[ \text{end\ PrimeSystem.Impl;} \]

The annotation instantiates the ACOL specification by assigning its abstract features to concrete model elements. These elements can be properties of the component itself, or object types and their properties reachable from the component. The scheme effectively allows information harvesting throughout the model hierarchy, but updating of properties is restricted to the component or its direct subcomponents. Fig. 3 shows the system hierarchy of the example model from section III, starting from system component PrimeSystem. An ACOL annotation to PrimeSystem has potential visibility on itself, together with Processor1, Process1, Thread1 and Thread2. The annotation can update the properties of PrimeSystem, Process1 and Processor1. This scheme favors ACOL annotation on the right level in the system hierarchy. It is also the reason why the AADL Classifier type cannot be used with ACOL, as it would allow arbitrary references to model components.

V. Use Cases

Section II discussed how ACOL can assist in trading-off several non-functional properties: timeliness and power
consumption in particular. This section highlights two other ACOL aspects, automated updating and distributed calculation of properties, aided by some short modeling fragments.

A. Automated Updating

One of the assumptions for RMA-scheduling [17] on threads is that they are assigned a unique priority number inversely proportional to their period. Consider an ACOL type with the following features:

\[ S : \text{in list of thread} ; \]
\[ P : \text{out list of property} \text{ Priority} ; \]

The AADL processor component on which RMA-scheduling is performed can map these features as follows:

\[ S \leq \text{Self’Subcomponents * thread} ; \]
\[ P \Rightarrow (\text{Self’Subcomponents * thread}).\text{Priority} ; \]

The RMA priority assumption is then guarded with ACOL implementation:

\[ \text{acol implementation RMS.P} \]
\[ \text{analysis} \]
\[ S_{\text{Sorted}} := S’\text{SortUp(Period)} ; \]
\[ \text{Priorities} := 1 \ldots S’\text{Count} ; \]
\[ I := \text{Priorities’Iterator} ; \]
\[ S_{\text{Sorted}}(I).P := I ; \]
\[ \text{end RMS.P} ; \]

The constraints and optimizations sections are empty in this example. It is noted that the ACOL specification, nor the model annotation need change when thread subcomponents are added or removed from the processor. The model is kept consistent automatically. The annotation will furthermore implicitly guard that every thread defines the \textit{Priority} and \textit{Period} property.

B. Distributed Property Calculation

Calculating the availability of a system involves combining the availabilities of its components. Figure 4 depicts the availability graph of the example DSP system introduced in section III. It shows how the components combine availability-wise in series and parallel.

![Fig. 4. Example DSP system availability graph.](image)

The availability of the top level system component \textit{DSP} can be calculated with ACOL in a distributed fashion. Components on the bottom level of the system hierarchy (such as threads) calculate their availability based on other, measurable, properties such as \textit{Mean Time Between Failure} and \textit{Mean Time To Repair}. Each component on an intermediate level up to the top of the system hierarchy calculates its own availability by combining the availabilities of its subcomponents in series, parallel or a combination of both according to the availability graph. The following ACOL specification could be used to combine subcomponents in series, and to ensure that the minimum required availability is reached:

\[ \text{acol Series} \]
\[ \text{features} \]
\[ S : \text{in list of component} ; \]
\[ A : \text{out property} \text{ Prop::Availability} ; \]
\[ \text{end Series} ; \]

\[ \text{acol implementation A.Series} \]
\[ \text{analysis} \]
\[ A_{\text{List}} := S.\text{Prop::Availability} ; \]
\[ A := A_{\text{List’Product}} ; \]
\[ \text{constraints} \]
\[ \text{Fail when} \{A < \text{Prop::Two_Nines} \} ; \]
\[ \text{end A.Series} ; \]

When ACOL is invoked, it will create a dependency graph between the components based on the direction of the ACOL features. As such, it is possible to determine that ACOL specifications associated with components lower in the system hierarchy need processing first. The total availability will be calculated by visiting the components bottom-up, ensuring model consistency.

VI. RELATED WORK

ACOL combines three distinct aspects of architectural modeling: analysis, constraints and optimizations. Most modeling formalisms include an annotation language primarily focusing on constraints or invariants in isolation, keeping analysis and optimization external to the modeling context. A prime example is the Object Constraint Language (OCL) [18]: a declarative language intended to resolve ambiguities in UML models through specification of formal rules.

Rooted in OCL is the Value Specification Language (VSL), defined as part of the UML profile MARTE [19]. VSL allows users to textually define variables, properties, expressions and relations on UML model elements. Due to the expressive power of VSL, we believe a mapping of the ACOL principles to MARTE should be straightforward.

REAL [20] is an approach to enforce invariants on AADL models. REAL is a DSL based on set-theory, where sets are predefined collections of model elements or derivatives thereof. REAL makes it natural to check non-functional properties globally on the model, and as such avoids model inconsistencies. The introduction of sets during the ACOL to AADL mapping was avoided, because the concept is alien and AADL provides lists. It is however possible to define REAL as a subset of ACOL, when set types and predefined sets are introduced.

ACME [21] is an architecture interchange language, grasping commonalities between different ADLs. The specification of ACME includes the Armani Predicate Language [22], a constraints language based on first-order predicate logic. Armani, REAL and OCL all share the possibility to construct collections by querying the model globally, and to validate the properties of their elements with boolean expressions. ACOL
includes this functionality as well, but applies it on a local scale making it potentially more verbose.

VII. CONCLUSION AND FUTURE WORK

Architecture Driven Development is a proven methodology in the embedded system domain. Finding the right trade-off between multiple non-functional properties early during design is key, as it avoids expensive development iterations. Appropriate tools and techniques are needed in support of an efficient, model based design process.

In this paper we propose ACOL: a model annotation language based on the combination of analysis, constraint and optimization expressions. We show how local annotation of model components, combined with global resolution, leads to a powerful framework usable during architectural design space exploration. The ACOL rationale was illustrated through several examples, and a mapping of the ACOL principles to AADL was provided.

We are currently working on the implementation of ACOL as an extension to the open source AADL Tool Environment (OSATE)[23]. Future work will include the mapping of ACOL to other ADLs, and validation of the framework on an industrial case-study. We will furthermore investigating the combination of ACOL and model based refactoring, as refactoring could streamline the necessary model changes when the architecture fails to find the appropriate trade-offs.

VIII. ACKNOWLEDGMENT

This work has been carried out as a part of the Condor project (http://www.esi.nl Projects->Condor) at FEI company under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

REFERENCES


