Ontology-based Model Driven Engineering for Safety Verification

Konstantinos Mokos, George Meditskos, Panagiotis Katsaros, Nick Bassiliades
Aristotle University of Thessaloniki
Department of Informatics
54124 Thessaloniki, GREECE
{ mokosko@otenet.gr } { gmeditsk, katsaros, nbassili@csd.auth.gr }

Vangelis Vasiliades
Gnomon Informatics S.A.
Thessaloniki, Greece
{ v.vassiliadis@gnomon.com.gr }

Abstract—**Safety assessment of dependable systems is a complex verification task that is desirable to be explicitly incorporated into the development cycle during the very early stages of a project. The main reason is that the cost to correct a safety error at the late stages of system development is excessively high. Towards this aim, we introduce an ontology-based model-driven engineering process for automating transformations of models that are utilized as reusable artifacts. The logical and syntactical structures of the design and safety models have to conform to a number of metamodel constraints. These constraints are semantically represented by mapping them onto an OWL domain ontology, allowing the incorporation of a Description Logic OWL reasoner and inference rules, in order to detect lacks of model elements and semantically inconsistent parts. Model validation throughout the ontology-based transformation assures that the generated formal safety model fulfils a series of requirements that render it analyzable. Our approach has been implemented as a response to an industrial problem[1], where the architecture design is expressed in Architecture Analysis and Design Language (AADL) and safety models are specified in the AltaRica formal language.**

Keywords- model driven engineering, safety, verification and validation, ontology reasoning, transformation.

I. Introduction

Dependable systems are developed based on complex safety requirements that need to be verified by rigorous methods, according to well established engineering processes and standards [1]. In industrial projects, safety assessment is one of the fundamental risk reduction and control processes that is performed during the very early stages of system development. In requirements analysis, engineers identify critical system items and all potential failure modes and technical risks that can lead to nonconformance of the anticipated safety standards. During design, the dependability characteristics (e.g. availability, reliability, etc.) are traded with other system attributes, such as performance, in order to discover an optimal design that fulfills the safety requirements.

In [2], we proposed an ontology-based representation of component failure behavior. The motivation behind our proposal is to cope with:

- heterogeneity in textual representation, syntax, semantics and scope of the used modeling languages for architecture design and safety assessment;
- possible inconsistencies between the design and safety models, as a consequence of the use of different tools;
- the need for a repository of reusable knowledge artifacts;
- the need to support and enhance collaboration between multiple project contributors, as well as information sharing intra and inter organizationally.

In the same line of work, we elaborate on ontology-based model transformation for safety assessment. The primal reusable artifacts are executable design specifications of system components with underspecified failure behavior (nominal behavior models). The nominal models are combined with failure modes, i.e. reusable artifacts of error behavior that are also stored in the ontology. The obtained extended system models are then transformed into formal safety models, which have to fulfill a series of constraints, in order to become analyzable by tools that provide model checking, fault tree, simulation and other analyses.

These constraints are semantically represented by mapping them onto an OWL domain ontology. The ontology and the inference rules checked by the incorporated OWL Description Logic reasoner allow detecting lack of model elements and semantically inconsistent parts. Our model-transformation isolates design flaws that invalidate safety analysis and provides support towards attributing the problem(s) to specific component(s) of the design model. Consequently, we avoid the cost of interpreting errors that are encountered in the formal representation of the extended system model, which is inherently difficult to conceive.

[1] This work is partly funded by the European Space Agency (ESA) ESTEC Contract Ref: 22262/09/NL/CBI

The outlined work was implemented as a response to an industrial problem, where the architecture design is expressed in the Architecture Analysis and Design Language (AADL) [3, 4] that provides precise execution semantics for modeling software systems and their target platform. Safety models are generated in the AltaRica formal language [5], which is processed by a series of analysis tools [6, 7, 8]. Our approach can potentially be applied in different verification and validation contexts, which will be based on other modeling and formal analysis languages.

Related work is reviewed in Section 2, before the introduction to necessary ontology background found in Section 3. We then present a short description of the AADL and AltaRica languages. Section 5 describes the domain ontology for the proposed model-driven engineering process, while in section 6 we summarize the rules behind the developed transformation. Section 7 introduces a case study that demonstrates model validation throughout the transformation process and provides the obtained analysis results. We conclude with a summary of the proposed process and we comment on its benefits and its potential impact.

II. RELATED WORK

To the best of our knowledge, related work on ontology-based model transformation is reported only in [12]. However, in that work the motivation of the authors is to improve cross-organisational modeling by supporting automatic generation and evolution of model transformations, a concern which is not addressed in our case.

In [7] the authors mention two related attempts on AADL model transformation to Altarica in the frame of the ASSERT European Integrated Project [9]. The first model transformation [8, 10] was based on extracting, the functional and hardware architecture of the system from the AADL model, as well as on the use of libraries of Altarica nodes, which can be reused from one project to the other. According to the authors, this approach worked well only for families of similar systems.

In the second case the transformation was based on AltaRica specifications that were enriched with failure propagations derived from AADL code written in AADL Error Annex [11]. The ASSERT reports inform us that the Altarica code was much more complex than before, but the transformation was feasible, as long as component relationships and various kinds of analysis are properly defined [7].

The fundamental difference of our work is that the ontology-based model transformation is built on a semantic bridge between AADL and Altarica that levels the differences in language syntax, scope and semantics. We believe that this semantic gap-filling along with the increased opportunities for model reuse and reasoning supported by the underlying ontology, contribute towards widening the applicability of the model transformation.

III. BACKGROUND INFORMATION ON ONTOLOGIES

The Semantic Web initiative [13] attempts to solve problems related to knowledge representation, by suggesting standards, tools and languages for information annotation, which use "data about data" called metadata. By uncovering implicit knowledge hidden into metadata, machines are able to reason over the represented data, draw conclusions and turn implicit knowledge into explicit.

Ontologies play a key role in the evolution of the Semantic Web and are widely used to represent knowledge, by describing data in a formal and explicit way. The Web Ontology Language (OWL) [14] is the W3C recommendation for creating and sharing ontologies in the Web and its theoretical background is based on the Description Logic (DL) [15] knowledge representation formalism, a subset of predicate logic. It has been emerged as the solution to the expressive limitations of RDF and RDF Schema that offer the possibility to define only simple hierarchical relationships among concepts and properties, domain and range property restrictions and concept instances. OWL is a richer vocabulary description language for representing properties and classes, such as relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry) and enumerated classes [16].

The formal semantics of the OWL language enable the application of reasoning techniques, in order to make logical derivations that involve class membership, equivalent classes, ontology consistency and instance classification. These derivations are performed by reasoners (e.g. Pellet [17]), which are systems able to handle and apply the semantics of the ontology language. Furthermore, already existing frameworks [18, 19] allow the combination of rules and ontologies, in order (a) to manipulate ontological knowledge, enhancing its expressiveness and (b) to allow ontology queries to be conducted in the form of rules.

IV. THE AADL AND ALTARICA LANGUAGES

The Architecture Analysis & Design Language (AADL) is a textual and graphical language for specifying the software and hardware architecture of safety critical real-time systems [3, 4]. AADL components encapsulate computations, and connectors represent communication between the components. Connectors connect components using ports, where each port plays a specific role in the context of the connector. Implementations are instances of the components and may also represent compositions of components and connectors with their ports and roles.

AADL is designed to be extensible and currently is accompanied with two basic annexes: the behavior annex [20] and the error model annex [11]. The Error Model Annex provides additional properties related to the reliability of the system components and allows defining state machines for specifying error behavior. The Behavior Annex completes the standard AADL language by additional syntax and semantic definitions to express component behavior. Several tools have been developed for editing and syntactically analyzing AADL specifications [9, 21].

AltaRica [5] is a dependability language for specifying constraint automata, i.e. formal models for the error behavior of systems. In AltaRica, a model consists of hierarchies of nodes, which gather flow variables that can be read and shared with other nodes. A model includes also states, events, transitions that describe how the initial states may evolve and assertions that are boolean formulae defining constraints, in order to link flows to internal states.

Several dependability tools process AltaRica models [5, 6, 7, 8], in order to perform various analyses (symbolic simulation, model-checking, fault tree analysis, sequence analysis etc.) with diverse requirements on the model.

V. DOMAIN ONTOLOGY AND THE MODEL-DRIVEN ENGINEERING PROCESS

The domain ontology has been designed for allowing storage, retrieval and correlation of constructs relevant to the AADL specification, the Error Annex and the Behavior Annex and for providing the desirable reasoning capabilities among them. In order to ensure decidability of the reasoning process, the ontology has been defined in the OWL DL fragment of OWL. We have used the Pellet DL reasoner, which was combined with the Jena framework for defining custom inference (consistency checking) rules. The ontology design has been focused on three main aspects:

- For the core AADL specification, the ontology provides the necessary constructs in order to model components (e.g. data, subprogram, system etc.), together with their corresponding properties (e.g. features). In this way, it is possible to add new components to the ontology or to extend existing ones, by defining appropriate ontology subclasses.
- For the AADL Error Annex, the ontology provides the necessary constructs to represent error models in terms of states and events, as well as to define implementations of error models in terms of transitions. The error models are classified in a hierarchy, according to the error type they model [22], e.g. computational problems, hardware errors, memory exceptions etc.
- For component implementation, the ontology provides the constructs (connections and subcomponents) to define the implementation of a component. Furthermore, the ontology allows the representation of the behavior specification that corresponds to a specific error model, thus bridging the semantic gap between the Error and Behavior annexes. This behavior can be either explicitly stated in the ontology or it can be semantically derived by reasoning during the procedure of the correlation of an error model with a component by the domain expert, without needing to have a complete picture of the actual knowledge that is required to achieve this task. This is feasible by exploiting the semantic capabilities that are provided by the ontology, such as hierarchical relationships or necessary and sufficient conditions, through the use of an OWL ontology reasoner.

Furthermore, the ontology-based representation of components allows the definition of custom inference rules, in order to check the model for potential component inconsistencies. Such inconsistencies cannot be detected using the default modeling capabilities of OWL and the use of rules is required. Currently, the system detects three types of inconsistencies that render the models invalid for certain types of formal analysis:

- Dynamic behavior of components is the situation where the final state of a component depends on a failure ordering scenario (fault tree analysis is impossible).
- Conflict transition fire, where there is more than one event that can be triggered simultaneously (fault tree and sequence analyses are impossible).
- Incompleteness transition errors, i.e. missing transitions in a component (ill-defined design specification).

The aforementioned inconsistencies are detected using custom SPARQL [23] rules that query the ontological knowledge and report potential errors. Furthermore, the rule-based representation of the inconsistency semantics allows new inconsistency rules to be easily added. We consider extending inconsistency detection capabilities with additional problems like unreachable states and unreachable transitions.

Figure 1 illustrates the described class hierarchy and Figure 2 presents a snapshot of the relations between the ontology representations of the AADL constructs. The ontology follows a meta-modeling design approach and defines classes for artifact definitions, implementations and instantiations. Each AADL component is represented as an ontology class, which should be a direct or indirect subclass of the upper-level class `AADLComponent`. Error models are also represented as ontology classes, descendents of the upper-level class `ErrorModel`. Furthermore, the ontology allows categorization of error models [22]. Each error model is a subclass of the provided error model hierarchy representing error types, where the upper-level class is `ErrorModelTaxonomy`.

Figure 3 depicts the proposed model-driven engineering process. The steps are:

1. Select/reuse failure modes from the error model hierarchy.
2. Correlate them with the nomimal component models at hand.
3. Check the ontology model for potential component inconsistencies.
4. Transform the extended architecture model to the formal safety model.
5. Analyze the safety model with tools that provide model checking, fault tree, simulation and other analyses.

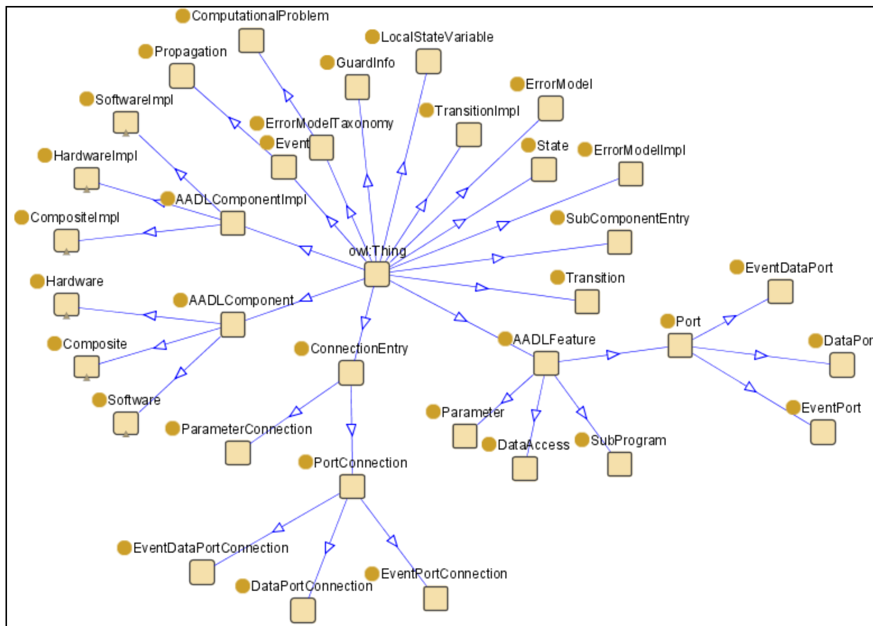


Fig. 1. Hierarchy of AADL Error/Behavior Annex

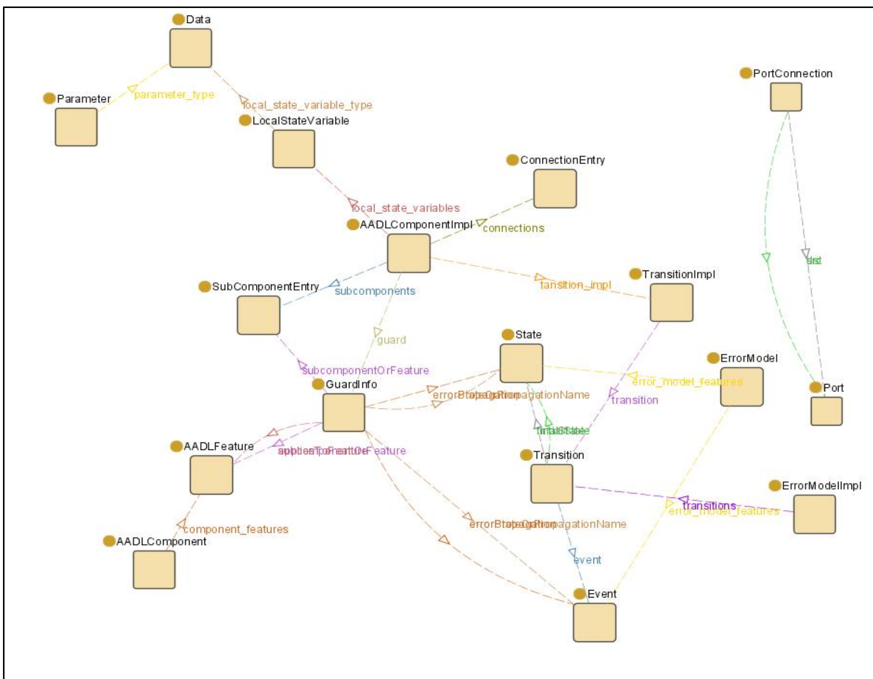


Fig. 2. Relations of AADL Error/Behavior Annex

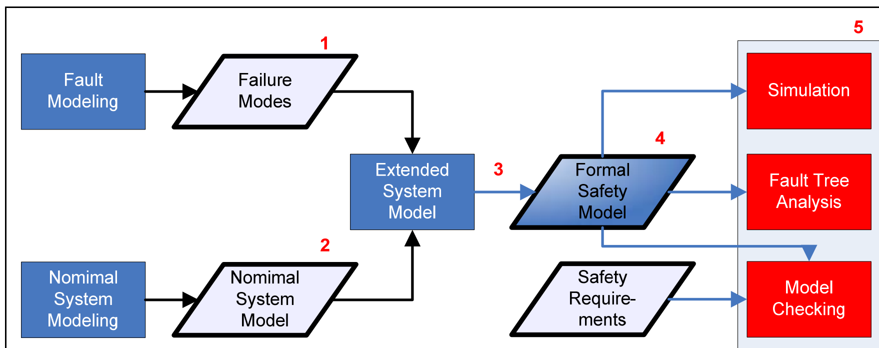


Fig. 3. Model-driven engineering process

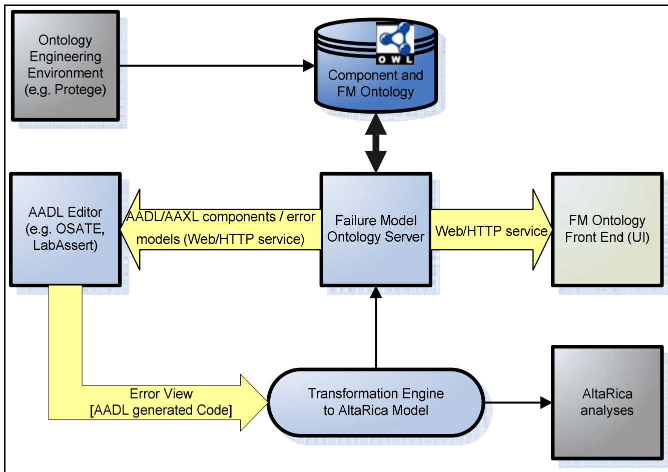


Fig. 4. Ontology-based model engineering architecture

The model-driven engineering architecture is shown in Figure 4. The Ontology organizes the components and the error models into functional and structurally interdependent hierarchies and enforces constraints and rules on the associations between components and error models.

A custom Web based front-end is used for viewing, editing, posting and updating components and error models within the ontology. The transformation engine generates the AltaRica formal safety model from the system design, which is developed in an appropriate AADL editor.

VI. MODEL TRANSFORMATION

Model transformation was based on existing modeling experience for discovering structurally equivalent constructs in the two languages. Transformation rules are driven by knowledge matching the used constructs through the underlying domain ontology.

The first set of rules is devoted to the transformation of AADL components to AltaRica nodes. All component types (i.e `system`, `process`, `thread`, `thread group`, `data`, `subprogram`, `processor`, `device`, `memory` and `bus`) are transformed to AltaRica `nodes` and their defined `features`, `flows` and `properties` to equivalent `flows`, `init` and `extern` AltaRica declarations. Furthermore, AltaRica `sub`, `state` and `assert` declarations are generated based on the AADL component implementations. AADL `property set` definitions are transformed to equivalent AltaRica flow type declarations.

The second set of rules concerns the components' error models and take into account their `states` and `transitions` triggered by `events`. AADL Error Annex `events`, `states` and `transitions` are transformed to equivalent AltaRica `events`, `states` and `transitions`. AltaRica `assert` statements are filled by assignments found in the matched AADL Behavior Annex code. In case of subprogram calls the transformation preserves their order.

The third set of rules focuses on the failure propagation, filtering and masking mechanisms (`Guard_In` and `Guard_Out` properties) and the mechanisms for connecting error `states` to operational `modes` (`Guard_Transition` property). AADL Error Annex `Guard_in` and `Guard_out` declarations are transformed into equivalent AltaRica synchronization assignment statements. Since AltaRica does not support failure masking capabilities, `Guard_transition` declarations create additional component variants, where in each `mode` the transformed component is mapped to its own error model and behavior.

Another set of transformation rules concerns the used architecture design process and the associated AADL editor. The LabAssert system design environment [9], which is used in the ASSERT architecture process, uses multiple views, as was first proposed by Kruchten et al [24]. The main problem encountered is that the LabAssert Interface View follows a control-flow based specification approach, thus resulting in a semantic gap compared to the data-flow oriented representation of the AltaRica specification. The transformation rules that are specific to LabAssert are:

- The Interface View provides the number of times each function is called. When a function is called multiple times our transformation generates as many nodes as the number of calls.
- In order to express the internal implementation (`subprogram`) of each function and the ordered sequence of calls we employ the constructs of the AADL Behavior Annex.

VII. CASE STUDY AND ANALYSIS RESULTS

An illustrative example, of our ontology-based model driven engineering approach, is shown in Figure 5. The example is about a double-adder function based on a simple add operation shown in the AADL standard AS5506 [20]. The nominal models were initially designed using AADL LabAssert Tool [9] and were combined with failure modes from the domain ontology in the OSATE toolset [21] (since at the moment there is no support for both annexes in LabAssert).

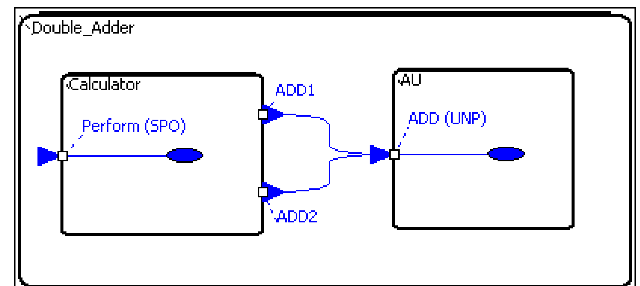


Fig. 5. AADL Specification of Double Add Example in Labassert Tool

In detail, the system's arithmetic unit (AU) performs a simple add operation, which is defined inside the ADD subprogram. Similarly, the Calculator system is responsible for performing a double add operation, which is defined inside the Perform subprogram. ADD1 and ADD2 are required interfaces of Calculator and ADD (UNP) is a provided interface of AU. One system calls a subprogram, when the required interface of the former is connected to the provided interface of the latter. Since Calculator is connected to the AU provided interface using two required interfaces, the ADD subprogram is called twice. The output of the first call is used as input to the second, resulting in a Double Add operation. We used behavior annex declarations to express sequential subprogram calls. All specification models were extended with error models representing the Overflow Event and State, shown in Table 1.

TABLE I. AADL SPECIFICATION OF ERROR MODEL OVERFLOW

Error Model	Implementation
error model OVERFLOW features Error_Free: initial error state; overflow: error event; Over_Flow: error state; end OVERFLOW;	error model implementation OVERFLOW.Notional transitions Error_Free -[overflow]-> Over_Flow; end OVERFLOW.Notional;

A. *Dynamic behavior of component errors*

The purpose of this kind of inconsistency detection refers to the implementation part of the defined error models, in order to check whether the final state of an error model transition depends on failures order scenario. Specifically, SPARQL rules search all transitions in Error Annex declaration and throw an exception, if there are at least two transitions with the same initial state, the same trigger event but different final state. If A is a state declared in Error Annex, then all transitions from state A to itself (ie. A -[e]-> A, where e is the event trigger) are ignored, since these kind of declarations are also used to express error propagation. The example, shown in Table 2, is a variation of the overflow error model defined previously, where a new state Ovf_Temp is added to express a new transition to a temporary overflow state.

TABLE II. AADL SPECIFICATION OF ERROR MODEL OVERFLOW

Error Model	Implementation
error model OVERFLOW_2 Features Error_Free: initial error state; Ovf_Temp: error state; Over_Flow: error state; overflow: error event; end OVERFLOW_2;	error model implementation OVERFLOW_2.Notionala transitions Error_Free -[overflow]-> Over_Flow; Error_Free -[overflow]-> Ovf_Temp; end OVERFLOW_2.Notionala;

The error model implementation of OVERFLOW_2 (Notionala) has two declared transitions from state Error_Free to Ovf_Temp and Error_Free to Over_Flow, both triggered by the same event overflow. Ontology exception handling will scan the error model implementation and will prompt the user that the first reference rule is violated. The two transitions defined in error model OVERFLOW_2.Notionala are responsible for the component's dynamic behavior, since the final state will either be Over_Flow or Ovf_Temp, depending on which transition will fire first.

B. *Conflict transition fire errors*

The purpose of this kind of inconsistency detection also refers to the implementation part of the defined error models, in order to check whether more than one event can be triggered simultaneously. Specifically, SPARQL rules search all transitions in Error Annex declaration and throw an exception, if there are two transitions with the same initial state, different trigger event and different final state. Similarly to dynamic behavior of component errors, if A is a state declared in Error Annex, then all transitions from state A to itself (ie. A -[e]-> A, where e is the event trigger) are ignored, since these kind of declarations are also used to express error propagation. The example, shown in Table 3, is a variation of the overflow error model defined previously, where a new event overflow_temp is added to lift the first rule violation. The error model implementation of OVERFLOW_2 (Notionalb) has two transitions from state Error_Free to Ovf_Temp and Error_Free to Over_Flow, triggered by different events overflow_temp and overflow additionally. Ontology exception handling will scan the error model implementation and will prompt the user that the second reference rule is violated. The two transitions defined in error model OVERFLOW_2.Notionalb are responsible for the component's conflict transition fire behavior, since both overflow and overflow_temp events can be triggered. Furthermore, the final state will also be different (Over_Flow or Ovf_Temp) depending on which event will fire first, violating the first reference rule as well.

TABLE III. CONFLICT TRANSITION FIRE EXAMPLE

Error Model	Implementation
error model OVERFLOW_2 Features Error_Free: initial error state; Ovf_Temp: error state; Over_Flow: error state; overflow_temp: error event; overflow: error event; end OVERFLOW_2;	error model implementation OVERFLOW_2.Notionalb transitions Error_Free -[overflow]-> Over_Flow; Error_Free-[overflow_temp]-> Ovf_Temp; End OVERFLOW_2.Notionalb;

C. *Incompleteness transition errors*

The purpose of this kind of inconsistency detection refers to the implementation part of the component in order to check whether transitions declared in a component's Behavior Annex are complete. Specifically, SPARQL rules search all states in Behavior Annex declarations and throw an exception if they find at least one final state, in all the existing transitions, that has no code attached to that state.

The example, shown in Table 4, is a variation of the overflow error model defined previously, where transition declarations are fixed to lift the two previous rule violations. The error model implementation of OVERFLOW_2

(Notionalc) has two transitions from state Error_Free to Ovf_Temp and Ovf_Temp to Over_Flow, triggered by different events overflow_temp and overflow additionally.

TABLE IV. ERROR FREE EXAMPLE

Error Model	Implementation
error model OVERFLOW_2 Features Error_Free: initial error state; Ovf_Temp: error state; Over_Flow: error state; overflow_temp: error event; overflow: error event; end OVERFLOW_2;	error model implementation OVERFLOW_2.Notionalc transitions Error_Free-[overflow_temp]-> Ovf_Temp; Ovf_Temp -[overflow]-> Over_Flow; end OVERFLOW_2.Notionalc;

In the example shown in Table 5, the subprogram implementation of ADD (ADD.impla) has three states declared (Error_Free, Overflow and Ovf_Temp), according to the attached error model from Table 4. Furthermore, state Ovf_Temp has no behavior declarations in any of the behavior annex transitions, where it appears as final state. Ontology exception handling will scan the subprogram implementation and will prompt the user that the third reference rule is violated. The third transition defined in the subprogram's behavior annex is responsible for the component's incompleteness transition behavior, since the code under final state Ovf_Temp in transition overflow_temp is not complete.

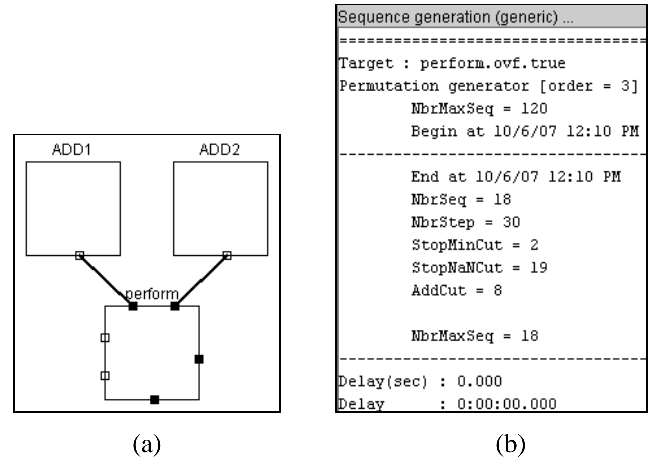
TABLE V. INCOMPLETENESS TRANSITION ERROR EXAMPLE

Error Model	Implementation

Subprogram ADD
Features
a: in parameter integer;
b: in parameter integer;
c: out parameter integer;
ovf: out parameter boolean;
end ADD;
	subprogram implementation ADD.impla
	annex behavior_specification {**
	states
	Error_Free : initial state;
	Semi : return state;
	Over_Flow : return state;
	transitions
	normal:Error_Free-[ ]->Error_Free {
	c:=(a+b);
	ovf := false;};
	overflow:Semi-[ ]->Over_Flow {
	c:=0;
	ovf := true;};
	overflow_temp:Error_Free-[ ]-> Semi {};
	**};
	annex error_model {**
	Model => OVERFLOW_2.Notionalc;
	**};
	end ADD.impla;

Dynamic behavior of components and Conflict transition fire rules inform us whether the transition declaration part of the AltaRica specification will be complete and error free, after the model transformation from AADL into AltaRica. The role of incompleteness transition errors detection rules is to make us sure that a behavior is specified for all declared states of the component. Thus, it insures us that, after the model transformation from AADL into AltaRica, all cases in the assert declaration part of the AltaRica specification have been taken care of and no code is missing from the AltaRica model.

Based on the prompts taken from checking the whole specification, the AADL specification design was converted into AltaRica specification code. Figure 6a shows the double adder AltaRica specification from Cecilia Ocas tool. Figures 6b and 6c show the results taken after executing Cecilia Ocas Sequence generator, and figure 7 displays the results taken after executing Cecilia Arbor Fault Tree viewer. The analysis made in Double Add example shows that the system will fail due to overflow error, if at least two of the three components fail due to overflow or overflow_temp events.



(a) (b)

```
products(Basic('perform.ovf.true')) =
('perform.overflow_temp', 'ADD2.overflow_temp', 'ADD1.overflow_temp')
('perform.overflow_temp', 'ADD1.overflow_temp', 'ADD2.overflow_temp')
('ADD2.overflow_temp', 'perform.overflow_temp', 'ADD1.overflow_temp')
('ADD2.overflow_temp', 'ADD2.overflow', 'ADD1.overflow_temp')
('ADD2.overflow_temp', 'ADD1.overflow_temp')
('ADD1.overflow_temp', 'perform.overflow_temp', 'ADD2.overflow_temp')
('ADD1.overflow_temp', 'ADD2.overflow_temp')
('ADD1.overflow_temp', 'ADD1.overflow', 'ADD2.overflow_temp')
end
```

(c)

Fig. 6. The AltaRica model structure (a), the Sequence generation (b) and results (c) for the Double Add Example in Cecilia Ocas

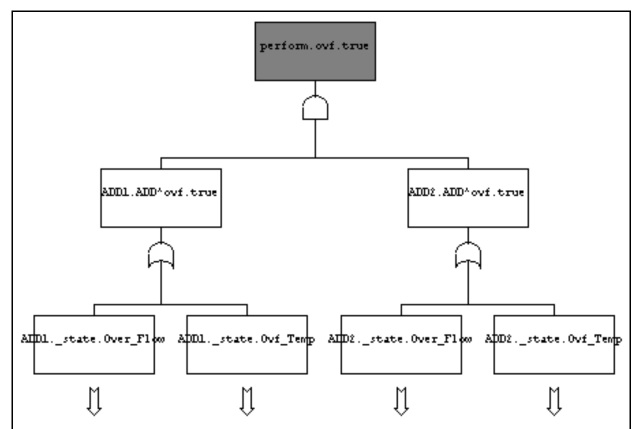


Fig. 7. Fault Tree generated by Cecilia Arbor for the Double Add Example

VIII. Conclusion

In this paper we introduced an ontology-based model-driven engineering process for compositional safety analysis. The proposed process enhances system architecture design by promoting model reuse, as well as model transformation with the following characteristics:

- Modular and extensible components representation.
- Interoperability between the design and formal analysis languages at a semantic level, as well as model transformation, which is not based solely on a purely syntactic analysis.
- Extended reasoning, taking the advantage of the ontology languages built-in reasoning capabilities.
- Web-based engineering environment accessible from anyone within or across organizations.

The underlying domain ontology and the checked inference rules allow detecting lack of model elements and semantically inconsistent parts. This allows system designers identifying flaws at the design level, without having to interpret errors that are encountered in the formal representation of the system model, which is overly difficult to comprehend. Model editing is also accelerated, since inconsistency errors can be found without having to perform complex analyses.

Although the shown process was developed in the frame of an industrial problem with specific technology constraints, we believe that the concept, as well as the architecture of the proposed engineering environment can boost the use of formal methods in the software/system engineering practice.

Our claim is based on the fact that model transformation eliminates the need of manually developing formal specifications of system models. Future research and development plans include the incorporation of additional types of inconsistency errors like the loop assert condition problem and the causality loop possibility, which render fault tree analysis impossible. Another research goal is the development of appropriate support for extending model transformation functionality and for validating the developed transformations [25]. The presented process will be fully beneficial upon the development of appropriate ontology semantics for a range of inheritance relations that will be able to promote component models retrieval and reuse.

References

[1] European Cooperation For Space Standardization – ECSS-E-40 Part 1B, Space engineering – Software – Part 1: Principles and requirements, ESA-ESTEC, Noordwijk, The Netherlands, 2003.

[2] Mokos, K., Katsaros, P., Bassiliades, N., Vassiliadis, V., Perrotin, M.: Towards Compositional Safety Analysis via Semantic Representation of Component Failure Behaviour. In: Knowledge-based Software Engineering/Proc. of the 8th JCKBSE - Piraeus, Greece, Front. in Artificial Intelligence & Applications, IOS Press, pp. 405-414, 2008

[3] SAE Architecture Analysis and Design Language (AADL) Annex Volume 1 http://www.sae.org/technical/standards/AS5506/1.

[4] Society of Automotive Engineers, SAE Architecture Analysis and Design Language, SAE standard AS5506. Available in: http://www.aadl.info.

[5] "Safety assessment with AltaRica – Lessons learnt Based on Two Aircraft System Studies". In 18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification, Toulouse France, 26 -26 August 2004, IFIP.

[6] Joshi, A., Miller, S. P., Whalen, M., Heimdahl, M. P. E. A proposal for model-based safety analysis, 24th Digital Avionics Systems Conference (DASC), 2005.

[7] Bieber, P., Blanquart, JP., Durrieu, G., Lesens, D., Lucotte, J., Tardy, F., Turin, M., Seguin, C., Conquet, E. Integration of formal fault analysis in ASSERT: Case studies and lessons learnt, 4th European Congress on Embedded Real Time Software (ERTS 2008), Toulouse, France, Jan 2008.

[8] Bieber, P., Castel, C., Seguin, C. Combination of Fault Tree Analysis and Model Checking for safety assessment of complex system, 4th European Dependable Computing Conference, LNCS 2845, Springer, pp. 19-31, 2002.

[9] The ASSERT Project: Automated proof-based System and Software Engineering for Real-Time Systems, http://www.assert-project.net/

[10] Xavier Dumas, Claire Pagetti, Laurent Sagaspe, Pierre Bieber, Philippe Dhaussy: Vers la génération de modèles de sûreté de fonctionnement. CAL 2008: 157-172

[11] Feiler, P., Rugina, A. Dependability Modeling with the Architecture Analysis & Design Language (AADL), Software Engineering Institute (SEI), 2007.

[12] Roser S., Bauer B.: Automatic Generation and Evolution of Model Transformations Using Ontology Engineering Space. J. Data Semantics (JODS) 11, Vol. 5383, pp 32-64 (2008).

[13] W3C, The Semantic Web Activity, http://www.w3.org/2001/sw/.

[14] McGuinness, D. L. and Harmelen, F. OWL Web Ontology Language Overview, W3C Recommendation, http://www.w3.org/TR/owl-features/.

[15] Baader, F. The Description Logic Handbook : Theory, Implementation and Applications. Cambridge University Press, 2003.

[16] Antoniou, G. and Harmelen, F. A Semantic Web Primer. Cooperative Information Systems. MIT Press, 2004.

[17] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. Pellet: A Practical OWL-DL Reasoner. Web Semantics: Science, Services and Agents on the World Wide Web, 5(2), pp. 51-53, 2007.

[18] McBride B.: Jena: Implementing the RDF Model and Syntax Specification. In Proceedings of the 2nd Int. Workshop on the Semantic Web, 2001.

[19] KAON2, http://kaon2.semanticweb.org/.

[20] AADL behavioral annex V2.0 http://gforge.enseeiht.fr/projects/osate-ba.

[21] OSATE website, http://www.aadl.info/aadl/currentsite/tool/osate.html

[22] Roy A. Maxion, Robert T. Olszewski, Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study, IEEE Transactions on Software Engineering, vol. 26, no. 9, pp. 888-906, 2000.

[23] SPARQL, http://www.w3.org/TR/rdf-sparql-query/.

[24] Kruchten, P. “Architectural blueprints – The 4+1 view model of software architecture”, IEEE Software, 12 (6), 1995, pp. 42-50

[25] J. M. Küster. Definition and Validation of Model Transformations, Software and Systems Modeling (SoSyM), Vol. 5, No 3, pp. 233-259, September 2006.