Toward Executable Architectures to Support Evaluation

Lee W. Wagenhals\textsuperscript{1,}a, Stewart W. Liles\textsuperscript{b}, and Alexander H. Levis\textsuperscript{a}
\textsuperscript{a}: Systems Architecture Laboratory, ECE Department, MS 1G5, George Mason University, Fairfax, VA 22030-4444, USA
\textsuperscript{b}: Knowledge Management Office, I Corps
Fort Lewis, WA 98433
Stewart.liles@us.army.mil

\{lwagenha, alevi\}@gmu.edu

\textsuperscript{1} Primary and Responsible Author

ABSTRACT

It has been 10 years since the Department of Defense (DoD) first issued its Architecture Framework (DODAF). In the past we have shown that it is possible to take an architecture description that conforms to DODAF and convert it to an executable model to support rigorous evaluation. We now have a proof of concept that demonstrates the feasibility of making the conversion to the executable model automatically. Implementing this automatic translation will enable architects to apply more rigorous analysis and evaluation approaches to complex architecture evaluation problems such as determining the agility in particular architecture designs that are based on information sharing. This paper describes the issues in automatic translation of architectures that are based on either the Structured Analysis or the Object Oriented approaches into the rigorous executable modeling language of Colored Petri Nets. The paper includes a proof of concept case study and briefly discusses the advantages of using the executable model in evaluating logical, behavioral, and performance aspects of the architecture.

1. INTRODUCTION

The move toward loosely coupled System of Systems (SOS), including Service Oriented Architectures to increase flexibility and information sharing, is creating the need to understand not only the static description of the architecture, but also the expected dynamic behavior and performance of the resulting designs. Architectures have been chosen because their level of abstraction enables relatively rapid analysis of alternative design concepts. One problem facing the community of system architecture developers is that current frameworks, such as the Department of Defense Architecture Framework (DODAF), prescribe only static representations of the architecture description. As a result, this is what the architecting community provides. Furthermore, many practitioners and managers of architecture projects do not have the time to spend developing full architecture descriptions nor do they have the expertise to convert the architecture description into an executable model that can be used for evaluation. But the need to evaluate the characteristics of architecture descriptions to determine their suitability to meet user needs requires evaluations that go beyond those possible with just a static representation. Comparison of alternatives requires understanding of the requirements and an assessment of the ability of candidate designs to meet those requirements. Evaluation requires careful selection and definition of measures that quantify in terms of formulae the effects of changing parameter values. It also requires the collection of parameter values generated by a particular instance of a candidate system architecture solution so that comparisons can be made to perform the evaluation. Obtaining the data needed for evaluating many of these measures is not possible with just static representations of the architecture. However, an executable model of an architecture description can generate data that enable the evaluation of the architecture’s ability to meet stakeholder needs.

Past efforts have described mappings from architecture descriptions to executable models. The architecture creation and description process relies on one of two prevalent methodologies, Object Orientation or Structured Analysis. Either methodology can produce all the information needed for conversion to the executable model [4], [5], and [6], but care must be taken to follow procedures that ensure all the needed data are captured. Past efforts required a manual transformation, i.e., an architecture was created in one tool and then the architecture artifacts were used to manually create the executable model in a different tool. This requires the expertise of someone familiar with both the architecture...
description language and the executable model. Furthermore it cannot be accomplished unless the architecture description contains all of the constructs that are needed for the executable model. If one or more constructs is missing, then the complete executable model cannot be created. In many cases, the architecture descriptions that satisfy the requirements for architecture delivery are lacking in one or more aspects needed for the executable.

To mitigate these problems, the creation of processes and tools that will automatically generate executable models of the architecture description seems to be a potential solution. Over the years, tools have been developed to support architects, system engineers, and software developers in designing their products. The tools tend to favor one of the two prevalent methodologies, Structured Analysis or Object Orientation. Most tools used by practitioners support the creation of the static models; a few of these tools include executable models, but include them either as separate applications, or they automatically create the models using a set of default constructs.

An automated process that creates an executable model that has complete bi-directional traceability between the static modeling language representations and the executable model is now feasible. Indeed, Liles [7] has created the code that will generate the complete executable model within an existing tool (Rational System Developer) from an architecture expressed using the Unified Modeling Language (UML). The output of the code is a file that can be opened in CPNTools [1] that is ready to support analysis.

In this paper we describe the issues involved in developing an automated process for creating an executable model from either an Object Oriented or Structured Analysis based architecture. Section 2 discusses the issues of meta modeling and multi modeling that are at the heart of the proper transformation process formulation. The section provides both an informal and a formal description of the elements of the Structured Analysis as a set of architecture modeling languages and Colored Petri Nets as a target executable language. Section 3 illustrates that process and implementation that Liles developed for creating an architecture description using UML that is automatically converted to the Colored Petri Net (CPN) target for analysis. Section 4 discusses architecture evaluation using the executable model, and Section 5 provides observations and conclusions.

2. META MODELS AND MULTI MODELING

To address the proposed solution of creating an automatic conversion from an architecture description to an executable model, it is important to understand that we are dealing with the translation of concepts in different modeling languages. Structured Analysis uses a set of modeling languages to describe a design. It is the predominate approach used by system engineers, and it is a process oriented approach based on functional decomposition. Structured Analysis designs are based on a combination of (1) an activity or process modeling language (e.g. IDEF0 or Data Flow Diagram) which provides a description of functions and relationships, (2) a Data Modeling language (e.g., IDEFIX or Entity Relationship Diagram) that provides entities with attributes and domain values that describe the exchanges between the process, and (3) Rule Modeling that is used to provide activation rules for activities using Structured English, Decision Tables, or Decision Trees.  Each modeling language describes important aspects of the design although no single language provides a complete description. Each language contains some concepts that are common to one or more of the other languages.

Transformation or translation between modeling languages requires an understanding of the symbols, syntax, and semantics of the different languages and the relationships between those elements in the different languages. We call this meta modeling and multi modeling. A meta model is a model about a modeling language. Since we are attempting to integrate two or more modeling languages, we must create and compare the meta models of the different languages in a method we all multi modeling. This means it is important to be sure that entities that appear to be the same in two modeling languages do not in fact contradict in concept when used in the translation between models, that is they have the same semantics. Mappings between the meta models of the languages must be consistent and complete. They must be consistent in the concepts each language conveys and complete meaning that all the concepts in the “target” language have a mapping from a source language. Then instances of the languages must also be complete (all rules must be followed). One technique to do this is to create formal models of the languages that will be involved in the transformation (e.g., both the source languages and the target language).

Fig. 1 shows the basic concept behind automatic translation. There are two types of meta models shown. The one in the upper left hand side represents the set of meta models of the architecture description languages. The one in the upper right is the meta model of the target executable modeling language. In order to create the automatic translation from the architecture to the executable model, the translation or mapping rules must
be developed. These mapping rules are then expressed in code. The code will take an instance of the architecture description and translate it into an instance of the executable model.

We illustrate these concepts in Fig. 2. It shows three elements of a Structured Analysis based architecture design, the activity model (language IDEF0), the corresponding Data Model (language IDEF1X), and the Rule model (as Activation Rules in the IDEF0 Model). Note that each of these models is expressed in its own modeling language. Since these models are used together to describe the complete architecture, they must be concordant. This means that the elements of each model must be consistent with one another. Notice that each ICOM flow in the IDEF0 model has a representation as an entity in the IDEF1X model. The same name is used for the same concept in both models. Furthermore, the concepts are consistent within the two models. The arcs of the IDEF0 model are labeled with the name of the type of entity that flows between activities and the data model further describes those entities. Each entity has attributes for which domains (allowable values) are defined. Each leaf level activity in the IDEF0 model has activation rules that consist of clauses. Each clause is a statement about a condition or a consequence of a condition. The clauses are based on the description of the entities and their attributes in the data model. The condition represents the possible combination of values that the inputs and controls to the activity can have. The consequence provides the description of the output if the condition is true, that is what entity instances are produced when the activity receives instances of the input and control entities. The output is described in terms of the output entities as described in the data model. Clearly there is a need to maintain concordance across the set of models used in Structured Analysis. If the three models are not in concordance then contradiction or ambiguity exists and must be corrected.

Fig. 2 also shows the target executable model, which in this case is a Colored Petri Net (CPN). This type of model is appropriate for the executable model of many architecture descriptions, and the Colored Petri Net (CPN) is a very general discrete event dynamical system model [2]. CPNs are mathematically rigorous, executable, and enable both simulation and analysis of properties. Included graph theoretic constructs support the visualization of process, communications, and control patterns of a system. They capture the precedence relations and structural interaction of concurrent and asynchronous events. Their mathematical foundation enables the analysis of important properties of discrete event systems including reachability and boundedness. Because they are executable, they can support simulation. Once created, the executable model can be used to support logical, behavioral, and performance evaluations as described in Section 4. Petri nets have four elements: Places, Transitions, Arcs, and Tokens. They are executable based on firing rules (see Fig. 3). Petri nets have been extensively used to model manufacturing processes, and banking, transportation, and information
systems. CPNs are a generalization of the basic Petri Net. Tokens can be typed representing entity instances of arbitrary complexity. They are typed in what are called color sets. Firing rules are expressed in rule modeling format as arc inscriptions, guard conditions, or code segment (Fig. 4 is an example.) In Fig. 2 one can visually see the relationships between the elements of the three models used to create the Structured Analysis based architecture description and the elements of the CPN model. These relationships must be formally established in order to develop the translation rules and then the translation implementation code to enable automatic translation of an architecture instance to a CPN executable model.

Fig. 5 shows a formal data model for an architecture description based on the Structured Analysis concepts described above. This model shows the individual meta models for the activity, data, and rule models integrated into one meta model. Fig. 6 shows a similar data model (meta model) for a CPN. Again the mapping between the two formal meta models is evident. Basically, the model pages of the activity diagram map to the CPN model pages. Activities map to transitions, including substitution transitions. Activity model (ICOM) arcs are mapped to places in CPN. Activation rules map to CPN arc inscriptions, guard conditions, or code segment. Entity descriptions in the data model map to the Global Declaration Node and the Color Declaration and the Variable Declarations that are used in those arc inscriptions, guard conditions, or code segments.

There are several challenges associated with the automated translation of Structured Analysis based architecture descriptions. Any tool that would support this translation must have concordance rules implemented along with the algorithms to check that the individual models are complete and in concordance. Most modeling tools that are used by architects do not implement the concordance compliance checking across the set of Structured Analysis architecture languages. Furthermore, the tools must implement the translation algorithm in order to be fully capable of supporting the design of an architecture and its translation into an executable model. Again, most tools used by practitioners do not have this capability. This means that practitioners using these tools who wish to create an executable model of their architecture design must do so manually.

When UML is used to describe an architecture, the problem is less challenging because the UML includes a meta model that is the basis for the concordance rules. Many UML tools have concordance compliance (consistency) checking built in; therefore such tools enable an architect to create a completed architecture instance that is consistent across the multiple models. UML has become the de facto standard modeling language for the Object Oriented Methodology which is
In UML the architecture description uses Classes and of the constructs described in the architecture description. There have been several efforts to enable the automatic of the architecture description. Links (Association instances) for a particular instantiation diagrams that illustrate the Objects (Class instances) and Associations to describe the constructs system under physical architecture that describes the specific instances system under analysis. An architecture instance is a relationships between the constructs for the particular Systems engineers are starting to use SysML which is a the dominant methodology for software engineering. One type of effort (e.g. Executable UML) has been the transformation to the executable model. The state machine diagram of UML as the basis for the automatic generation of an executable model of an architecture description. The approach is based on a specific process for designing an architecture. This means the architect will use UML in a precise manner to develop the architecture description. Liles has created the code that enables the automated translation from the architecture description to the executable model. In this section we illustrate Liles’ transformation process.

The process is based on specifying the architecture in a manner that is compliant with the DoD Architecture Framework. The application could be adapted to other frameworks. The DODAF provides multiple views of an architecture. The operational view (OV) is analogous to a business process view, and it does not describe the material solution. The OV has nine products, each showing a different aspect of the view. In particular the OV shows the “operational activities” (OV-5) which can be thought of as the business functions, descriptions of the operational information (OV-7) and flows that occur between the operational activities and operation nodes (OV-2 and 3), rules for the operational activities (OV-6a), descriptions of the organizations that carry out the operational activities (OV-4), and examples of behavior (i.e. business processes) using sequence diagrams (OV-6c). The Systems View (SV), now called the System and Services View in DODAF Version 1.5, describes the systems, services, and communications systems that will support the business process of the OV. Like the OV, the SV has multiple products that focus on different aspects of the SV. These include descriptions of systems, services, and interfaces (SV-1), system or service functions and system data and flows (SV-4 and SV-11), rules for those system and service functions (SV-10a), descriptions of behavior using sequence and state machine diagrams (SV-10c and b), and the communications infrastructure that supports the exchange of system or service data (SV-2). Notice that both the OV and SV have the basic modeling concepts needed to produce a CPN including the functional description (activity model), the data description, and the rules. This means that it is possible and may be desirable to create executable models of both views. In this way architects can perform dynamic evaluation of the OV and adjust the
OV design to meet operational (business) requirements. Then the architect can perform a more detailed evaluation of the SV to refine the performance evaluation. We discuss this further in Section 4.

The transformation process requires the architect to create a complete static behavioral description captured in a UML activity diagram that uses swim lanes. This activity diagram must be consistent with the other UML diagrams and tabular products of the DODAF products. The process requires specification of nodes (OV-2 or SV-1), activities (or functions) (OV-5 or SV-4), interrelationships between those activities (OV-5 and OV-3 or SV-4 and SV-6), a data model that formally describes the data and messages exchanged between functions (OV-7 or SV-11), and rules for transforming input data into output data (OV-6a or SV-10a). The UML behavioral representation of these products is reflected in a single UML activity diagram with swim lanes. Once this activity diagram has been completely specified, the transformation implementation executes a two step process to automatically create the executable model. In the first step, the code produces a CPN data model, which is an instance of the data model shown in Fig. 6. Then this data model is transformed into an XML file that is specific to the CPN tool (in this case CPNTools). The CPN tool is able to load the file for execution and analysis. The DODAF views have been implemented in UML using Rational System developer. The transformation code was implemented in Java. This enabled the creation of the UML architecture description and the automatic conversion to CPN to occur in one tool.

The transformation process translates the activity diagram components to the CPN. The basic components of the activity diagram are: Action, Object Node, Call Behavior Action, Fork/Join Node, Decision Node, Initial Node, Final Node, Stop Node, and Activity Parameter Node. An Action describes a fundamental unit of executable functionality, e.g., some processing. These are analogous to activities in the activity model of Structured Analysis. Actions translate to transitions in the CPN. In the resultant CPN model, transitions represent actions that take specific input from places and produce specific output to places. CallBehaviorAction is an action in the activity model used by the methodology to create a hierarchy of behavior diagrams. The CallBehaviorAction invokes lower level behavior diagrams. In this process they are the activity diagrams described for each Element.

Fig. 7. Activity Diagram to CPN Translation

It translates to a Substitution Transition in the CPN which is a transition that is associated with another page in the CPN. Object Nodes assist in describing the data that passes from Action to Action [3]. Liles extended Storrle’s [9] activity diagram transformation to include the CallBehaviorAction in order to implement hierarchical models. Four types of Object Nodes are used: Activity Parameter Node, Input/Output Ports, Data Stores, and Buffers. All of these Activity Nodes are transformed into CPN Places. Four types of Control Nodes are used: Fork, Join, Decision, and Merge. All the Control Nodes translate into CPN transitions. Decision Nodes require accompanying arc inscriptions that control the passing of tokens based on the value of variables represented in the token. These inscriptions are translated from the associated guards inscribed on the Decision Node output arcs represented in the activity diagram. Finally, the terminal Nodes (Initial and Final), represent the beginning and end of each Activity. They are transformed into places in the CPN. Fig. 7 summarizes the translation from the activity diagram to the CPN components.

The CPN can be generated from the activity diagram of either the DODAF OV-5 product (the Operational Activity Model) or the SV-4 (the System Functionality Description). The activity diagram must be complete in that it must be in concordance with the other DODAF products and their UML representation. For example, if
the SV-4 is used, it contains system functions represented as action nodes, system data as object nodes, and the rules based on the four types of control nodes (fork, join, decision, and merge) along with inscriptions on the flow arcs. Sets of Action Nodes are allocated to swim lanes that represent either the operational node or the system that performs those actions. (This must be consistent with the nodal diagrams of the architecture description, e.g. the OV-2 (Operational Node Connectivity Description) or the SV-1 (System Interface Description). The Object Nodes must be in concordance with the data model.

Figures 8 through 11 illustrate the various DODAF SV products that are created and show the concordance that must be maintained between them. Fig. 8 is a small SV-1. It shows two system nodes, a Land Node and a Sea Node. One system is shown in each node. The upper part shows interfaces between System 1 and System 2 that represent one capability and the lower part shows two systems in the Sea Node and interfaces between them that represent another capability. Fig. 9 is a fragment of the SV-6 (System Data Exchange Matrix). It describes how the system data messages are exchanged and is consistent with the SV-1. Fig. 10 shows the SV-11 (Physical Schema or Data model), and Fig. 11 shows the SV-10a (Rule Model). Both are consistent with the SV-6.

The architecture data captured in these products is assembled into the activity diagram, SV-4 shown in Fig. 12. The top part of Fig 12 maps to the first SV-1 capability of Figures 8 through 11 and the bottom part maps to the second capability of those figures. Note that Fig. 12 shows CallBehaviorAction nodes (indicated by the small pitch fork). Furthermore three systems are shown via the CallBehaviorAction nodes. These three nodes require decomposition. Each will be represented as substitution transitions in the CPN and will have a decomposed CPN page. The decomposition of each system is shown in Fig. 13 shows the separate behavior of each system. Note how this part of the activity diagram implements the rules that were described in the Rule Model of Fig. 11.

Once the UML description is completed (and has passed the concordance compliance check) it is ready to be transformed into the CPN. Fig. 14 illustrates the concrete syntax transformation using a fragment of an activity diagram and the transformation rules of Fig. 7. The input activity diagram is shown in the upper left hand side. In Step 1, the algorithm identifies the ends of connection arcs as transitions. Step 2 shows the transformation of the actions into transitions and then connects them with places to hold the token that are generated by the action. Step 3 creates the CPN data representation with the actions named in the transitions and the places used to hold the tokens generated by the actions.
Once the CPN data representation has been created, the CPN instance data model is generated (Fig. 15). This data model is then translated into the specific XML file used by the target CPN modeling tool, CPNTools, which can open the file. Fig. 16 shows the actual CPN model (the top page) that was generated from the data model in the example and opened in CPNTools.

The final step is to instrument the CPNTools model with monitors so that appropriate data can be collected to support the needed architecture evaluation and to create specific scenarios that will support that data collection.

4. EVALUATION USING EXECUTABLE

Once the executable model has been created, it can be used to address, in part, the following layered questions: (1) Is the Architecture logically correct? (2) Does the architecture exhibit the desired behavior? (3) Do instantiations of this architecture exhibit the desired performance characteristics? (4) Do systems built in conformance to the architecture provide the desired capability? (5) Can we analyze alternatives?

The construction of the executable model, especially one based on the operational architecture, provides the basis for checking the logical consistency and correctness. The first step is to validate the logic of the model. The static views describe the structure, data, and rules that manipulate that data to accomplish tasks. We need to verify that the combination of rules, data, and structure “works”, e.g. the rules are consistent and complete. This can be accomplished by executing the model to be sure that it runs properly. In a sense we are “debugging” the architecture. Any errors found must be corrected in the appropriate static views to preserve traceability.

We can execute the model using notional inputs to determine whether activities do indeed use data specified by the information exchange requirements. “Flaws” can result in either an incorrect response or a deadlock. We can test the sequence of events; i.e., does the executable model produce the sequences specified by the sequence diagrams? And we can evaluate whether the execution of activities is a correct implementation of the operational concept.

Once we know that the executable model runs, we can examine the behavior of the architecture; this is an examination of the functionality of the architecture. The behavior of the executable model and the behavioral diagrams should correlate. This behavior evaluation has several facets: Does the architecture produce all the correct output for a given stimulus? Does the information arrive at the right functions in the right sequence, i.e., are the inputs processed in the required way? The behavior of the architecture can be compared to the user’s requirements.

The behavioral correctness can be approached at two levels: at the operational level and the systems and services level. At the operational level, scenarios are developed and executed to determine whether the desired behaviors (as reflected in the state charts or state machine diagrams) are obtained. What is of particular interest here
is the identification of undesired behaviors or the possibility of undesired states. Note that state transition descriptions (OV-6b and SV-10b) and the event trace descriptions (OV-6c and SV-10c) capture only a few of the desired behaviors. A real system may exhibit many more behaviors and the use of an executable model is one way of determining them. If Petri Nets are used to represent the executable model, then algorithms such as those based on invariants can be used to relate the structure of the model to its behavior [8] or other techniques such as state space analysis [2] can be used to complement evaluation using simulation.

Once Behavioral Correctness has been established performance can be examined. This requires the incorporation of performance parameters of the systems in the executable model. Scenarios need to be developed that are consistent with the use cases and from the simulation results relevant Measures of Performance (MOPs) are obtained.

CPNs provide the capability of interacting with other applications. This means that it is possible to extend the evaluation by connecting the CPN executable model to other models to enhance the evaluation. A good example is interconnecting a CPN model with a network simulator such as OmNet++. The network simulator can provide higher fidelity behavior of the communications infrastructure that supports the system data exchanges. Other models (using other modeling languages) that model specific systems can be connected to the CPN. Properly interconnect these models entails similar meta model and multi model analysis concerns to those described in this paper.

5. COMMENTS AND CONCLUSION

It has been a long term goal to enable automatic executable model generation from the standard architecture description languages that are in common use. In this paper we have described the issues involved in correctly making this automated translation a reality. We also have illustrated a proof of concept for automated translation. The proof of concept described has some limitations in that it is based on the semantics of the UML activity diagram; these limitations could be reduced with further effort. Certainly the approach can be generalized to support a large number of architecture descriptions. There is a need for improved tools that can generate the executable automatically from the architecture data which will make this step even easier. This will enable architects who are well qualified to create the static representations to conduct the types of dynamic evaluation described in this paper. Of course evaluation techniques need to be included in the tools. Education and Training of practitioners is also needed.

Architectures, if done properly in a layered way, are a great tool for innovative design. They allow the exploration of radical alternatives in a short amount of time thus expanding the number of alternatives to be considered. But to handle the questions of SOAs and
System of Systems being addressed today we need to be able to rapidly and easily create the executable version of the architecture description. If we do not do this, the evaluations will be lacking. We need to refocus our efforts and develop applications that will take us to the desired end: an efficient, architecture-based systems engineering process that enables us to integrate legacy systems with new systems and exploit technology advances to provide desired capabilities to the users.

REFERENCES


ACKNOWLEDGEMENT

This work was supported by the Office of Naval Research under grant N00014-06-1-0081.

BIOGRAPHIES

Dr. Lee W. Wagenhals is a Research Associate Professor with the System Architectures Laboratory of the Electrical and Computer Engineering Department of the Volgenau School of Information Technology and Engineering at George Mason University. His education includes the BS in electrical engineering from Lehigh University (1965), the MS from the Air Force Institute of Technology (1971), and the Ph.D. in Information Technology from George Mason University (2000). He has over 20 years of experience with the United States Air Force prior to joining GMU in 1992.

Dr. Stewart W. Liles is a Lieutenant Colonel in the US Army. He is a 2008 graduate of George Mason University with a Ph.D in Information Technology where he was closely associated with the System Architectures Laboratory of the Electrical and Computer Engineering Department of the Volgenau School of Information Technology and Engineering His education includes a BS in Management Sciences and Computer Systems from Oklahoma State University (1988), a MS from the Naval Postgraduate School (1998), and the Ph.D. in Information Technology from George Mason University (2008).

Dr. Alexander H. Levis is University Professor of Electrical, Computer, and Systems Engineering and heads the System Architectures Laboratory of the Electrical and Computer Engineering department, George Mason University, Fairfax, VA. From 2001 to 2004 he served as the Chief Scientist of the U.S. Air Force, on leave from GMU. He was educated at MIT where he received the BS (1965), MS (1965), ME (1967), and Sc.D. (1968) degrees in Mechanical Engineering with control systems as his area of specialization. He also attended Ripon College where he received the AB degree (1963) in Mathematics and Physics. Dr. Levis is a Fellow of the Institute of Electrical and Electronic Engineers (IEEE) and past president of the IEEE Control Systems Society; a Fellow of the American Association for the Advancement of Science (AAAS) and a Fellow of the International Council on Systems Engineering (INCOSE); and an Associate Fellow of the American Institute of Aeronautics and Astronautics (AIAA).