

Computational Verification of System Architectures*

Abbas K. Zaidi and Alexander H. Levis, *Life Fellow IEEE*
System Architectures Laboratory, ECE Dept., MS 1G5
George Mason University
Fairfax, VA 22030 USA

Abstract—The paper presents a computational approach for verifying system architectures that employs a modal logic, an architecture design process, and a computer-aided formal model checking technique. The approach is shown to address the *traceability* issue between the architectural views, developed in accordance to the DoD Architecture Framework (DoDAF), and the executable model derived from the framework products. It provides an analytical underpinning of the verification of systems architectures, especially when requirements and capabilities of the systems under consideration evolve over time. The approach is presented with the help of an illustrative example.

I. INTRODUCTION

Two fundamental approaches have been developed at the System Architectures Lab, George Mason University [2-4] to implement the DoD Architecture Framework (DoDAF): structured analysis and object orientation. The end product in both cases is an executable model, derived from information contained in the framework's artefacts. These artefacts (or *products*) describe the structure, data and rules that manipulate the data to accomplish tasks. An executable model, if derived from these products in a traceable way, can enable logical, behavioural, and performance analyses: it can help verify if the combination of rules, data, and structure *works*, e.g., the rules are consistent and complete; its simulation runs can be used to debug the architecture and validate if the architecture does what it is supposed to. The nature in which modern-day systems evolve by integrating available (possibly at run-time) services or parts of other systems to develop new and unprecedented capabilities calls for robust analytical tools that can validate, verify, and even correct a system's behaviour well before the unintended consequences are observed. The ability to trace a system's behaviour, especially the undesired system trajectories, to its operational requirements forms a preliminary step, albeit an important one, towards building such tools.

This paper presents the use of modal temporal logics and formal automated model checking techniques, first proposed in [5], for an analytic underpinning of the architectural design process and analyses that follow it. The paper illustrates the approach with the help of an example architecture developed in [2] that uses the structured analysis paradigm. A branching-time logic, called ASK-CTL [6], is shown to model the specifications for the modelled system derived directly from the DoDAF architecture products. The set of formal logic

statements describing the system properties is also shown to be refined throughout the design process with the help of developed architectural products. For brevity, the example presented in this paper only uses processes/(by)products from the first three stages of the design approach to illustrate the refinement of some of the system properties. The last product in the design process, presented in [2], is an executable Colored Petri Net (CPN) of the system. The approach in this paper shows how a state-space based formal model checking technique can be employed to verify if the designed system satisfy the properties given as ASK-CTL statements. The CPN model is implemented using the software application *CPN Tools*. The tool is developed and maintained by the CPN Group, University of Aarhus, Denmark. The ASK-CTL toolkit provided with *CPN Tools* is used to run the verification step. This last step of the approach is fairly automated provided a designer correctly derived the properties to be verified; however, a lot of information that goes into an executable model (i.e., CPN) might have been provided by the designer (as modelling artefacts) without a proper mapping to/from the other architectural products, which creates a traceability problem between the products and the executable model. The approach presented in this paper can be used to address this issue since it interprets the logic statements, representing specifications in the architecture products, on the state-space of the executable model, thus linking behaviour of the executable model to the elements of architecture products.

The remainder of this paper is organized into six sections. Section II presents a very brief outline of the design approach presented in [2]. This is followed by another brief introduction to the temporal logic ASK-CTL [6] in Section III. Section IV presents the algorithm for the computational verification that combines the model checking technique with the architecture design process. The verification is then illustrated with the help of an example and the formal logic statements that capture the system properties in Section V. The paper concludes in Section VI with a summary of the verification process and a discussion on possible future directions for further inquiry.

II. ARCHITECTURE DESIGN APPROACH

The details of the architecture design approach based on structured analysis were first presented in [2]. The article also presented procedures for deriving, from the information

* The work was carried out with support provided by the Air Force Office of Scientific Research under contract number FA9550-05-1-0388.

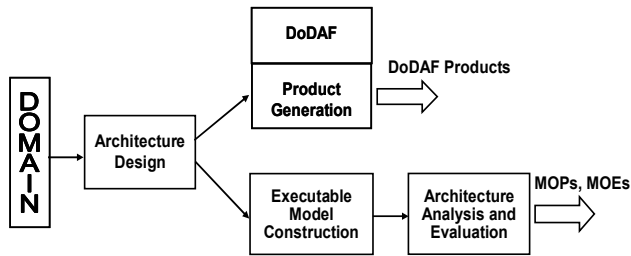


Fig. 1. Template of the Architecture and Evaluation Process [1].

contained in the architectural products, an executable Colored Petri Net (CPN) model. The CPN model is shown to exhibit behaviour and performance characteristics of the architecture. An abstract, top-level view of the architecture design and evaluation process is shown in Fig. 1, adapted from [1]. The detailed description of the process contains six stages. Each stage generates one or more of the DoDAF products for the *Operational* and *Systems* views. The first step (Stage 0), in this six-state process, involves the collection of domain information. Table 1 suggests some of the documents and information that can be useful inputs to the design process. As was described in [1], the architecting process must start with a clear purpose and viewpoint. An operational concept, a concise statement that describes how the goal will be met, must be provided. In this paper, we propose to use some of these source documents, namely D1, D2, D5, D6, D7, and D8, to extract formal logic statements about the system properties described in these documents. More specifically, the information provided in D1, D5 and/or D8 can be used to construct logical formulas comprising of propositions, states, events, and/or functions of the modelled system depicting their operational relationships and/or (un)desired behavioural properties. The other documents, D2, D6, and/or D7, can be used to construct the initial glossary (or dictionary) of terms used to represent the system propositions, states, events and/or functions. The construction of formal logic statements representing system properties can be carried out by either the users or the architects of the system. The resulting document listing these statements may be used as a formal requirements

TABLE 1
INPUT TO THE PROCESS (STAGE 0)

AV1	Purpose, Viewpoint (Problem Definition)
D1	Operational Concept Narrative
D2	Universal Joint Task List
D3	Current DoD Organization Charts
D4	Description of Organizational Relationships
D5	Textual Description of Doctrine, Tactics and Operational Procedures
D6	List of Operational Information Elements
D7	Definition of States and Events
D8	Description of System Functions
D9	Communication System Description
D10	Performance Attributes of Systems
D11	Migration Plans for Systems
D12	Description of Systems

document shared between the two groups for the system to be developed. In the following discussion, the document will be referred to as the 'Specification Document.' It can be further refined or updated by the architects throughout the design process.

The next three design stages, 1-3, of the six-stage design approach are shown in the Data Flow diagrams of Figs. 2 to 4. The diagrams show the needed inputs on the left (from Stage 0 or from previous stages) and the framework products on the right. The diagrams are included in this paper for the constituent processes that are shown to be helpful for refining the system correctness statements. Fig. 4 is partially drawn and does not show a number of processes that contribute to subsequent stages in the design approach. More specifically, the processes that contribute to the Specification Document are shown shaded in these diagrams. The detailed and comprehensive account of the approach is in [2].

The process of constructing and refining the Specification Document is illustrated with the help of an example system. The example uses only the first four stages to illustrate the verification approach. All the design stages used in this paper

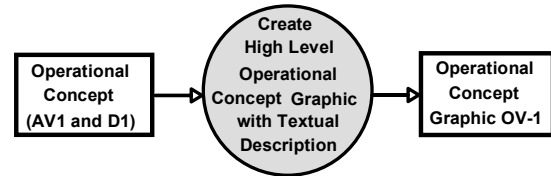


Fig. 2. Process Model of Stage 1.

yield DoDAF products for the *Operational View* of the architecture. The approach can be easily extended to include design stages resulting in the *Systems View* of the architecture.

Synthesis of Executable Model

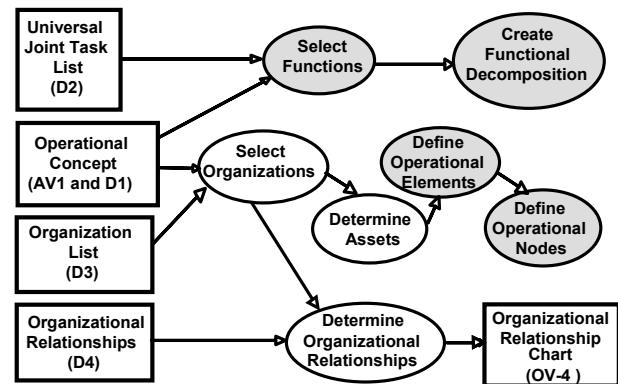


Fig. 3. Process Model of Stage 2.

The derivation of executable model, as described in [2], is carried out with structure and data/rule models extracted from Activity Model (OV-5), Operational State Transition Description (OV-6b), Operation Rules Model (OV-6a), and Logical Data Model (OV-7) of the architecture framework. Reference [2] provides details of the procedures for deriving, from the information contained in the architectural products, an executable Colored Petri Net (CPN) model. A number of commercially available software tools that support the design

of DoDAF compliant architectures have also picked up on the idea of developing executable models, in the form of Finite-State machines, to facilitate the validation and performance analysis of the designed system.

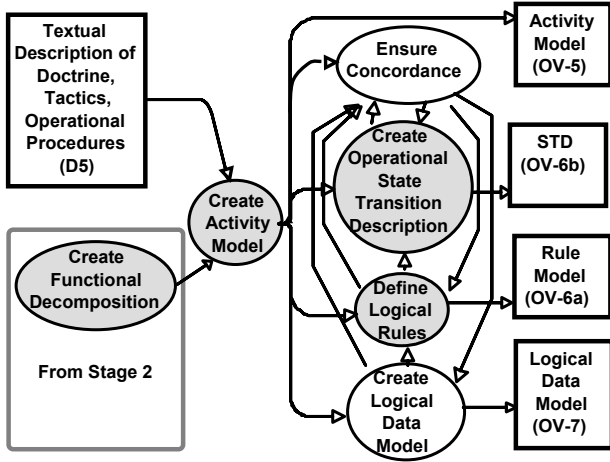


Fig. 4. Process Model of Stage 3a.

III. MODAL LOGICS AND FORMAL MODEL CHECKING

In this section, we present a very brief description of ASK-CTL taken from the detailed presentation in [6]. ASK-CTL is a *branching-time* modal logic and an extension of Computational Tree Logic (CTL), which is interpreted over the state-space of Colored Petri Nets (CPN), i.e., Occurrence Graphs (OG). Since an OG of a CPN is a labelled transition system, carrying information on both nodes and edges, the CTL extension in ASK-CTL allows the construction of logic formulas with both state and transition information depicting properties of the system under analysis. For brevity, we refrain from presenting details on CPNs, OGs and CTL. For more information on these topics, see [7] for material on CPNs and OGs, and [8] for CTL.

An ASK-CTL statement is defined to be a *state* or a *transition* formula. The two categories are mutually recursive and are defined, in [6], as follows:

Definition 1. ASK-CTL State Formula

An ASK-CTL state formula A is defined by the following production system:

$$A := tt \mid \neg A \mid A \vee A \mid EU(A, A) \mid AU(A, A) \mid \alpha \langle B \rangle$$

where tt is interpreted as true, α is a function from CPN *markings* to Boolean values and B is a transition formula, defined below.

Definition 2. ASK-CTL Transition Formula

An ASK-CTL transition formula B is defined by the following production system:

$$B := tt \mid \neg B \mid B \vee B \mid EU(B, B) \mid AU(B, B) \mid \beta \langle A \rangle$$

where β is a function from CPN transition *bindings elements* to Boolean values and A is a state formula.

A given ASK-CTL expression is by default a state formula and transition formulas appear as nested sub-formulas in a state expression. The following notation is used to define the

semantics of the operators introduced in Definitions 1 and 2 above.

A. Notation

Let the state-space, OG, of a CPN model be denoted by (V, D) , where V is the set of states and D is set of edges in the OG. A state in V is denoted by M , with M_0 being the initial state. An edge e_{ij} in D is denoted as $(M_i, \langle t, b \rangle, M_j)$, where M_i and M_j are the input and output states of the edge and $\langle t, b \rangle$ shows the transition label with t being the CPN transition and b the binding on t . A path in the OG is represented by the symbol θ with θ_i denoting the i^{th} state (θ_1 being the state the path originates from) and π_i the i^{th} edge on the path. In order to define the semantics, we will also need the functions α and β . The two functions return a Boolean value after checking some properties about the marking(s) in a state and the binding on a transition, respectively. The ASK-CTL toolkit of *CPN Tools* requires a user (or an architect) to implement these functions using CPN ML and Standard ML (SML) programming languages. (An example of such a function is provided in the following section.)

$$\alpha : V \rightarrow \{\text{True}, \text{False}\} \quad \text{and} \quad \beta : D \rightarrow \{\text{True}, \text{False}\}$$

The semantics of state formulas can now be defined as: Given a state M and a state formula A , the expression ' $M \models A$ ' denotes ' M satisfies A ' or ' A holds in M '.

$$M \models tt \text{ always holds}$$

$$M \models \alpha \quad \text{iff} \quad \alpha(M) = \text{True}$$

$$M \models \neg A \quad \text{iff} \quad \text{not } M \models A$$

$$M \models A_1 \vee A_2 \text{ iff } M \models A_1 \text{ or } M \models A_2$$

$$M \models A_1 \wedge A_2 \text{ iff } M \models A_1 \text{ and } M \models A_2$$

$$M \models EU(A_1, A_2) \text{ iff there exists a path } \theta \text{ in the OG starting at state } M \text{ such that } \exists i, \theta_i \models A_2 \text{ and } \forall j, 0 < j < i, \theta_j \models A_1 \text{ (The temporal operator } EU \text{ is read 'Exist Until'.)}$$

$$M \models AU(A_1, A_2) \text{ iff for all paths } \theta \text{ in the OG starting at state } M \text{ such that } \exists i, \theta_i \models A_2 \text{ and } \forall j, 0 < j < i, \theta_j \models A_1 \text{ (The temporal operator } AU \text{ is read 'For All Until'.)}$$

The semantics of transition formulas are defined similarly as follows. In an expression of the type ' $e \models B$ ', where e or $e_{ij} = (M_i, \langle t, b \rangle, M_j)$ represents an edge in the OG under investigation and B represents the transition formula.

$$e \models tt \text{ always holds}$$

$$e \models \beta \quad \text{iff} \quad \beta(t, b) = \text{True}$$

$$e \models \neg B \quad \text{iff} \quad \text{not } e \models B$$

$$e \models B_1 \vee B_2 \quad \text{iff } e \models B_1 \text{ or } e \models B_2$$

$$e \models B_1 \wedge B_2 \text{ iff } e \models B_1 \text{ and } e \models B_2$$

$$e_{ij} \models EU(B_1, B_2) \text{ iff there exists a path } \theta \text{ in the OG starting at state } M_i \text{ with } e_{ij} \text{ being the first edge on the path, } \exists k, \pi_k \models B_2 \text{ and } \forall m, 0 < m < k, \pi_m \models B_1$$

$$e_{ij} \models AU(B_1, B_2) \text{ iff for all paths } \theta \text{ in the OG starting at state } M_i \text{ with } e_{ij} \text{ being the first edge on all of these paths, } \exists k, \pi_k \models B_2 \text{ and } \forall m, 0 < m < k, \pi_m \models B_1$$

TABLE 2
DERIVED OPERATORS

Operator	Description
$Pos A \equiv EU(tt, A)$	It is <i>possible</i> to reach a state where A holds.
$Inv A \equiv \neg Pos \neg A$	A holds in every reachable state; A is <i>invariant</i> .
$Ev A \equiv AU(tt, A)$	For all paths, A holds within a finite number of steps; A is <i>eventually</i> true.
$Along A \equiv \neg Ev \neg A$	There exist a path which is either infinite or ends in a dead state, along which A holds in every state.
$\langle B \rangle A \equiv \langle B \wedge \langle A \rangle \rangle$	There exist an immediate successor state M satisfying A and B holds on the transition between the current state and M .
$EX(A) \equiv \langle tt \rangle A$	There exists an immediate successor state in which A holds. Read 'Exist Next'.
$AX(A) \equiv \neg EX(\neg A)$	A holds in all immediate successor states, if any. Read 'For All Next'.

The $\langle . \rangle$ operator in the definitions is used to switch from a state to a transition formula and vice versa. This operator allows a user to express a property about a transition originating from a state, if expressed as a state formula, or about the destination state of a transition, if expressed as a transition formula. Therefore, $M_i \models \langle B \rangle$ iff $\exists e_{ij}$, s.t. $e_{ij} \models B$ and $e_{ij} \models \langle A \rangle$ iff for $M_j \models A$.

Definitions 1 and 2 give the minimal syntax for the two types of formulas. The ASK-CTL library of *CPN Tools* offers a number of other derived, high-level operators for constructing complex expressions for the system properties. Some of these, for state formulas, are listed in Table 2 with their semantics expressed in terms of basic operators. A similar set of operators exists for transition formulas.

B. Model Checking and Computer-aided Verification Using ASK-CTL

The model checking problem with ASK-CTL now can be formally described as:

Given an OG of a CPN *system* and an ASK-CTL property (formula) p , determine if OG is a model for p (i.e., if $OG \models p$).

The following definition describes the interpretation of an ASK-CTL property over an OG.

Definition 3.

Given an OG (V, D) of a CPN system and an ASK-CTL property p ,

$OG \models p$ iff for initial state $M_0 \in V, M_0 \models p$

According to Definition 3, a CPN system is said to satisfy a property (or design requirement) if the ASK-CTL formula describing this property can be shown to hold in the initial state of the OG drawn for the CPN *system* under investigation.

The model checking algorithm of ASK-CTL, as implemented in *CPN Tools*, is presented in [6]. The complexity of the algorithm is shown to be linear in the product of the size of the formula and the size of the state space— $O(N(|V| + |D|))$, where N is the length of the formula, $|V|$ is the number of nodes and $|D|$ is the number of edges in the OG. This time complexity does not take into account the cost of evaluating the predicate functions α and β . For most cases, the two functions can be evaluated very efficiently without any significant effect on the overall running time. The two functions are implemented using CPN ML and SML programming languages, a requirement that offers both a

disadvantage and an advantage to a system architect: the disadvantage comes from the fact that the user or architect might need to learn yet another programming language to perform the verification; and the advantage is due to the flexibility of a programming language to construct a wide variety of predicate functions. For example, ML functions can be written for *Timed* CPNs that evaluate the timing information on the markings/bindings in the *Timed* OG for checking timeliness of certain properties in the system. This makes the CPN Tools implementation of ASK-CTL toolkit an especially powerful verification tool for real-time systems with specific timeliness and/or deadline requirements for processes and/or system states.

The model checking algorithm in [6] is a modification of the standard CTL algorithm presented in [8]. The ASK-CTL algorithm in *CPN Tools* employs some reduction rules that break an input formula into basic primitives and eliminate the redundant parts. The algorithm is optimized for most of the primitive *patterns* in a formula. It finally employs a search of the state-space (OG) to check the validity of these primitives and their combinations in the states of the OG. A modified, compact representation of a state-space, called Strongly Connected Component graph (SCC-graph), is employed by the implementation in *CPN Tools* to increase the efficiency of the algorithm. The implementation is said to have handled state spaces with millions of nodes. An overview of the computer-aided verification approach presented in this paper, employing *CPN Tools* based ASK-CTL model checker, is presented in Table 3.

IV. VERIFICATION OF ARCHITECTURES

In this section, we present a verification approach for architectures by combining the model checking technique with the architecture design process. The revised template for the architecture design and evaluation process of Fig. 1 is shown in Fig. 5. We have already identified the input source documents and sub-processes in design stages that potentially contribute to the development and refinement of Specification Document, with ASK-CTL formulas describing requirements for the system to be modelled. A synthesis of an executable CPN model directly from the DoDAF products at the end of the architecture design process has already been proposed in [2,4] for both structured and object-oriented approaches. In this paper, we propose to employ *CPN Tools*' ASK-CTL

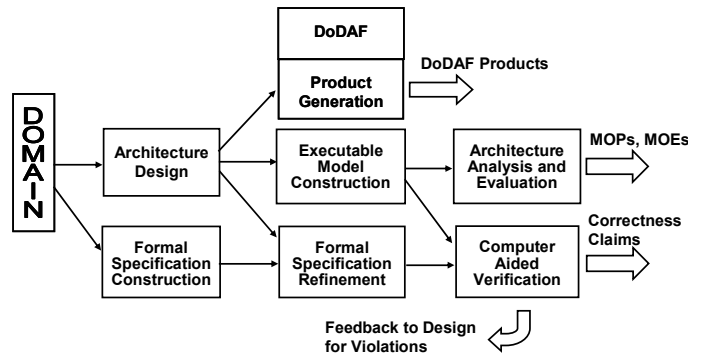


Fig. 5. Revised Template of the Architecture and Evaluation Process

TABLE 3
ALGORITHM FOR COMPUTER AIDED VERIFICATION USING ASK-CTL

<p>Given: (1) A CPN System (i.e., a CPN model with Initial Marking) implemented in CPN Tools. (2) A list of ASK-CTL formulas derived from architecture products</p> <p>Step 1. Run State Space Analysis Tool Step 2. Generate State Space and SCC graphs Step 3. Select a new ASK-CTL formula from the input list Step 4. Construct the predicate function(s) in the formula using CPN ML and SML syntax and embed the function(s) in the formula Step 5. Invoke ASK-CTL library Step 6. Evaluate the formula. If it returns true go to Step 3, else report the violation of property and stop</p>

toolkit to model check the synthesized CPN model against entries in the Specification Document.

Reference [2] illustrated the architecture design approach with the help of a fictitious commercial system, called FastPass system, that is inspired by the SpeedPass™ system used in the US by Exxon Mobil. We employ the same example system to illustrate our verification approach. The example has been extensively used in courses on the design approach.

V. THE EXAMPLE

Stages 0–2. Table 4 presents some of the input source documents available for the design process. Table 5 shows our first pass at the Specification Document with a sample of key statements taken from D1 and rewritten with the help of

functions and sub-functions listed in D2 and the Operational Elements from D6. These statements are also shown translated to corresponding ASK-CTL statements in the same table. The predicate terms representing a transition (sub)formula, in these statements, are constructed by identifying (wherever possible) the operational elements from D6 (e.g., Pump) and by selecting a task from the UJTL in D2, e.g., *Pump.Sense_FastPass*. The specific function/task names used in these predicates are the result of the processes (shaded in Fig. 3) in Stage 1. A closer look at the translated logic statements reveals the fact that the choice of temporal operators (e.g., *AU*, *EU*, *Inv*, etc.) may impose certain design restrictions that are not very obvious in the descriptive English statements. For example, in the statement *Sj* (Table 5), the use of operator *AX* mandates that after receiving the receipt the driver should be immediately out of the system. As an alternate, if the operator *Ev* is used instead of *AX*, the driver can go out of the system any time after the receipt. The formal, unambiguous nature of these statements, while desirable, may in some cases restrict (or impose) the design choices available to an architect. One possible solution could be to construct statements with operators that offer flexibility of the type illustrated by the use of *Ev* in *Sj*. Another solution could be to prepare a list of several ASK-CTL statements with each reflecting a possible translation of the requirement. For a meaningful verification of the architecture at the end, the set of formal statements should be made as comprehensive and accurate as possible. A missed requirement, or an incorrectly

TABLE 4
STAGE 0 DOCUMENTS FOR FASTPASS SYSTEM

AV1	Purpose: To increase convenience and reduce the time it takes for users buying gas with a credit card and in the self-service mode		Point of View: A key thread is considered. A key thread is a sequence of activities that take place when an individual driver pulls up at the pump and use the FastPass to get gas for his/her car.	
	Assumptions: The operation of a single pump equipped with the FastPass system is modelled. The architect need not worry about the details of the electronic accounting system used by the gas station. It has already been implemented and available for integration.			
D1: Operational Concept Narrative	A driver pulls up at one of the Self-Serve fuel pumps with the FastPass sign on it. If his/her car is equipped with the FastPass tag, then the sensor on the pump senses its presence and reads the information on the tag. If the driver has a key-chain tag, s/he waves the tag in front of the sensor (1 to 2 inches away) and the sensor reads the information. The sensor lights up. The information read from the FastPass tag is then sent from the pump, through the LAN at the gas station, to the FastPass central data base where the relevant credit card information is retrieved. The request for authorization along with the credit card data is then transmitted to the financial institution issuing the credit card. If the request is approved, the fuel pump is enabled and the driver can pump gas. If the request is denied, the pump is not enabled and the process terminates with the FastPass light going off. If enabled, the driver <i>selects the grade of gas</i> s/he desires and pumps gas until s/he turns off the pump by throwing a switch. Then the cost of the sale is computed at the pump and the amount is transmitted back to the financial institution where it is entered as a charge in the driver’s corresponding credit card account. A receipt is issued at the pump. The data about the sale are entered in the electronic ledger of the gas station. The pump returns to the idle state.			
D2: Universal Joint Task List (UJTL)	1. Validate Payments 1.1 Sense FastPass 1.2 Retrieve Driver Info 1.3 Validate Credit	2. Operate Pump 2.1 Maintain Status 2.2 Control Operation Mode 2.3 Dispense Gas	3. Manage Sales 3.1 Compute Cost of Sale 3.2 Request Charges 3.3 Print Receipt 3.4 Update Account	
D6: Operational Nodes and Elements	Operational Node		Operational Elements	
	Driver		Driver	
	Gas Station		Pump, Gas Station Office	
	OilCo		OilCo	
Financial Institution		Financial Institution		

TABLE 5
SPECIFICATION DOCUMENT

ID	Statement	ASK-CTL Formula
S1	When a driver pulls up with a FastPass tag and the pump senses FastPass, then the pump's light goes on	$(Driver.In \wedge \langle Pump.Sense_FastPass \rangle) \wedge AX(Pump.Light_On)$
S2	If the FastPass tag is sensed and the information on the tag is read, then the request of authorization and credit validation is sent to the financial institution.	$EV(Pump.FastPass_Sensed \wedge Pump.Driver_Info_Retrieved \wedge \langle Pump.Send_Authorization_Request \rangle \wedge AX\langle FinancialInst.Validate_Credit \rangle)$
S3	If request for authorization is approved, then Fuel Pump is Enabled.	$EV\langle Authorization_Approved \rangle \wedge AX(Pump.Enabled)$
S4	If request for authorization is denied, then pump's light goes off.	$EV\langle Authorization_Declined \rangle \wedge AX(Pump.Light_Off)$
S5	If the Pump is enabled and gas grade is selected, then gas is dispensed.	$EV(Pump.Enabled \wedge Driver.Gas_Grade_Selected \wedge EV\langle Pump.Dispense_Gas \rangle)$
:
Sj	Driver should get a receipt and go out of the system in the end	$EV\langle Driver.ReceiveReceipt \rangle \wedge AX(Driver.Out)$
Sk	The pump in the end returns to the initial idle state.	$Inv(Pos(Pump.IdleState))$

TABLE 6
OV-6a RULE IN ASK-CTL

Rule R22	ASK-CTL Formula
If (Authorization_Transaction.approval AND (Selection.On, Selection.Quantity, Selection.Grade)) Then (Dispensed_Gas_Data.Grade = Selection.Grade AND Dispensed_Gas_Data.Quantity = Selection.Quantity AND Display.Content = Message 3 Else Display.Content = Message 9	$EV((Authorization_Transaction.approval \wedge Selection.On \wedge Selection.Quantity \wedge Selection.Grade \wedge \langle Dispense_Gas \rangle \wedge AX(Dispensed_Gas_Data.Grade = Selection.Grade \wedge Dispensed_Gas_Data.Quantity = Selection.Quantity \wedge Display.Content = Message\ 3)) \vee Display.Content = Message\ 9)$

constructed statement, results in a system design with a low level of fidelity or unnecessary errors reported by the verification algorithm.

Stage 3. Stage 3 of the design process begins with creating the Activity Model (IDEF0), the Data Model (IDEF1X), and the rule model that correspond to the Activity Model (OV-5), Logical Data Model (OV-7), and Operational Rule Model (OV-6a), respectively. The construction of these products helps identify the exact labels for the predicates used to construct *atomic* state and transition formulas, shown in Table 5. An architect may also add rules from OV-6 to this document. In fact, most of the rules in OV-6a can be easily traced back to ASK-CTL formulas in the Specification Document. A software tool implementing the architecture design process may incorporate this mapping (or traceability) to/from the Specification Documents and the framework products. Table 6 presents a sample rule from OV-6a of the FastPass architecture and its corresponding ASK-CTL representation. In contrast to the rule in OV-6a, this representation of the rule can actually be used by the verification algorithm to check if it is used by the executable model to generate the right outputs.

Computer Aided Verification. Fig. 6 shows the *top-level* CPN of the FastPass system developed in [2]. Each transition

in the CPN represents a *substitution* transition with a *subpage* containing a detailed net. For lack of space, we only show the detailed CPN for the substitution transition labelled 'OperateFastPassSystem' in Fig. 6. The detailed CPN is shown in Fig. 7. Note that there are several substitution transitions in Fig. 7 depicting the fact that there exist subpages for each of them with even further detailed networks. The CPN shown is the object of the verification step. The use of *CPN Tools* and its built-in version of the ASK-CTL verifier require an architect to construct ML functions for each of the predicates in the Specification Document. For brevity, we illustrate the process for only one of the statements in Table 5. Table 7 presents the ML code constructed for checking the **Sj** property. In the code, Lines 1 and 2 implement *Driver.ReceiveReceipt* predicate; Lines 3 and 4 show the function for *Driver.Out* predicate. The two functions use CPN ML constructs for *transitions binding elements* and *node markings*. Lines 6 and 7 show the CPN ML implementation of the ASK-CTL formula.

TABLE 7
CPN CODE FOR ASK-CTL FORMULA Sj

```

1. fun ReceiveReceipt a = st_TI (ArcToTI (a))
2. = "Driver'ReceiveReceipt 1";
3. fun IsDriverOut n =
4. Mark.Environment'DriveOut 1 n <>[];
5. val myASKCTLformula =
6. EV (AND(MODAL(AF ("_", ReceiveReceipt)),
7. FORALL_NEXT (NF ("_", IsDriverOut))));

```

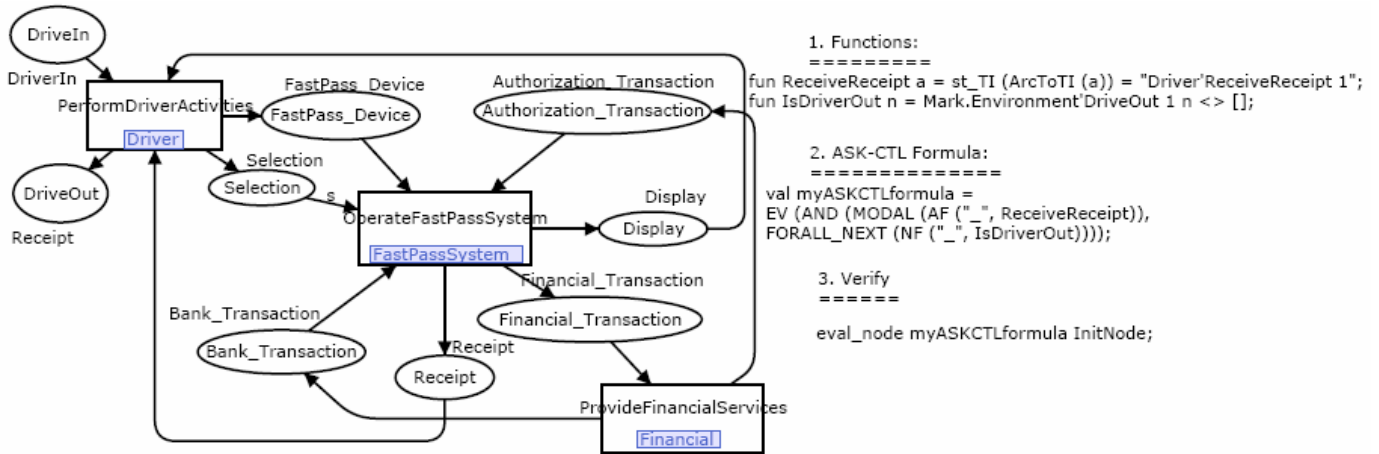


Fig. 6. CPN Top-Page for FastPass with Example Verification

The term MODAL implements the $\langle \cdot \rangle$ operator and the identifiers NF and AF represent α and β predicates, respectively. Line 5 assigns the formula to a variable that is then evaluated by invoking the built-in CPN ML compiler, as shown by '3.Verify' step in Fig. 6. For the example formula (Sj), the verifier returns a 'true' that confirms the property for the model. The other properties can be verified in a similar manner.

A high-level representation of the derivation of executable CPN and the Specification Document, as applied to the illustrative example, from a selected set of framework products and a subsequent verification process is shown in Fig. 8.

VI. CONCLUSIONS

An application of formal model checking techniques for developing analysis and assessment mechanisms for system architectures developed in accordance with the DoD Architecture Framework (DoDAF) is presented.

The presented approach offers a potential solution to the traceability problem between the DoDAF products and the executable model of the system: The construction of a fully executable model (e.g., Colored Petri net) requires an architect to incorporate information from several architectural products. The mapping of information from the products to artifacts in the executable model is not straight forward (or algorithmic), resulting in the loss of traceability. Any errors found in the executable model must be corrected in the appropriate views/products to have a meaningful evaluation. The approach uses ASK-CTL formulas, directly derived from the architecture products, to model check the architecture. It requires a user/architect to construct a Specification Document comprising these ASK-CTL formulas as part of the architecture design process. The strict semantics of the language, in turn, requires a user/architect to remove a number of ambiguities that might be present in the input description(s). CPN Tools is used for constructing the executable model and its built-in ASK-CTL library is used for running model checking algorithms. Errors, if found, can be

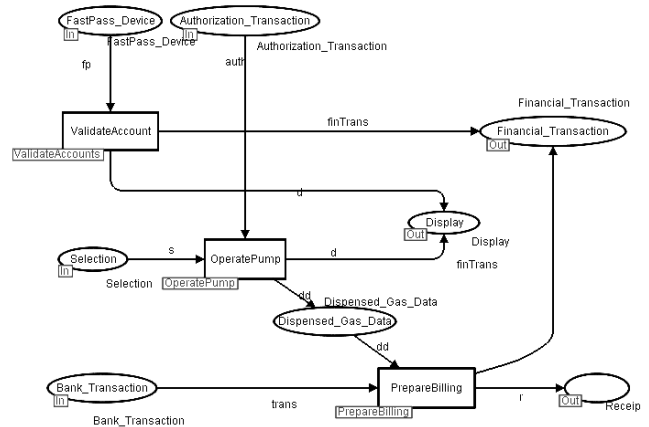


Fig. 7. Detailed View of the Substitution Transition 'OperateFastPastSystem' in Fig. 6.

traced back to the product(s) the corresponding formula was derived from and fixed. The approach can be employed using other available model checking, computer-aided verification tools. The syntax of a formal logic may not be very intuitive to a user and may require formal understanding of both the syntax and semantics of the logic and its language to develop a comprehensive and accurate, yet flexible, set of statements. An understanding of CPN Tools and SML programming language/environment is also required for someone interested in using the approach. Further research is required to fully automate the construction of Specification Document.

The approach was applied to an example architecture of a fictitious commercial system that has been extensively used in courses on system architectures. To facilitate the construction of the Specification Document, its refinement and a mapping between the logic statements in the document to architectural products and/or source documents, a number of software support tools can be developed as extensions to the existing software applications for developing DoDAF compliant architectures.

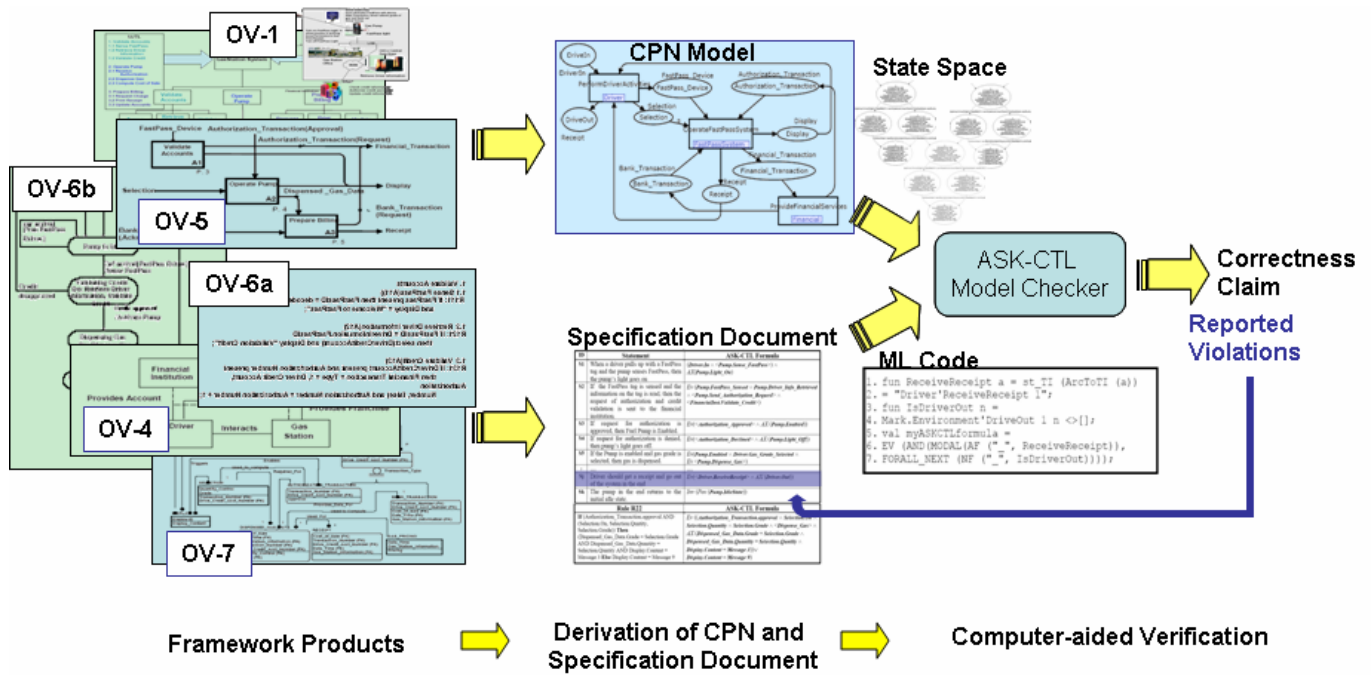


Fig. 8. The 3-Stage Representation of the Proposed Approach

REFERENCES

- [1] Levis, A.H. and Wagenhals, L.W., "C4ISR Architectures I : Developing a Process for C4ISR Architecture Design." *Systems Engineering*, Vol 3, No. 4, December 2000, pp. 225-247.
- [2] Wagenhals L.W., Shin, I, Kim, D and Levis, A.H., "C4ISR Architectures II : Structured Analysis Approach for Architecture Design." *Systems Engineering*, Vol 3, No. 4, December 2000, pp. 248-287.
- [3] Wagenhals, L.W., Haider S. and Levis, A. H., "Synthesizing Executable Models of Object Oriented Architectures." *Systems Engineering*, Vol. 6 No. 4. December 2003, pp. 266-300.
- [4] Bienvenu M.P., Shin, I and Levis, A.H., "C4ISR Architectures III : An Object Oriented Approach for Architecture Design." *Systems Engineering*, Vol 3, No. 4, December 2000, pp. 288-312.
- [5] Zaidi, A. K., and Levis, A. H., "Verification of System Architectures Using Modal Logics and Formal Model Checking Techniques." *Conference on Systems Engineering Research (CSER)*, Los Angeles, CA, April 2006.
- [6] Cheng, A., Christensen, S, and Mortensen, K.H., "Model Checking Coloured Petri Nets Exploiting Strongly Connected Components." in *Proc. of the International Workshop on Discrete Event Systems, Institution of Electrical Engineers, University of Edinburgh, UK, August 1996*, pp. 169-177.
- [7] Jensen, K., *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volumes 1-3, Monographs in Theoretical Computer Science, Springer-Verlag, 1997.
- [8] Clarke, E.M., Emerson, E.A. and Sistla, A.P., "Automatic Verification of Finite State Concurrent System Using Temporal Logic." *ACM Transactions on Programming Languages and Systems*, Vol. 8(2), 1986, pp. 244-263.