

# A Unified Approach for Verification and Validation of Systems and Software Engineering Models

Luay Alawneh<sup>†</sup>

Mourad Debbabi<sup>†</sup>

Fawzi Hassaine<sup>‡</sup>

Yosr Jarraya<sup>†</sup>

Andrei Soeanu<sup>†</sup>

<sup>†</sup>Computer Security Laboratory, Concordia Institute for Information Systems Engineering,  
Concordia University, Montreal, Canada.

<sup>‡</sup>Future Forces Synthetic Environments Section,  
Defence Research and Development Canada, Ottawa, Ontario, Canada.

## Abstract

*We present in this paper a unified paradigm for the verification and validation of software and systems engineering design models expressed in UML 2.0 or SysML. This paradigm relies on an established synergy between three salient approaches, which are model-checking, program analysis, and software engineering techniques. To illustrate the accomplishment of our results, we have designed and implemented an integrated and automated computer-aided assessment tool. We provide three case studies for sequence, state machine, and class and package diagrams to demonstrate the benefits of our methodology.*

## 1. Motivations

The continuous advancement in software and systems design led to the emergence of modern modeling languages, including the most prominent ones, namely UML [16] and SysML [17]. Both modeling languages are playing an important role in software and systems engineering. While software engineering (IEEE) [12] is defined as the application of a systematic approach to the development, operation, and maintenance of software, systems engineering (INCOSE) [7] is an interdisciplinary approach that enables the realization of successful systems focusing on the system as a whole. Ubiquitous systems such as ATMs or portable electronics as well as advanced technologies like aero-space, radars, defence or telecommunications are important application fields of systems engineering. In this context, the process of design and development mandates a strong and sound Verification and Validation (V&V) phase.

Verification is defined as the confirmation by examination and provision of objective evidence that the specified requirements have been fulfilled, whereas validation is defined as proving that the particular requirements for a specific intended use are fulfilled [19].

The V&V phase can be a major bottleneck in the development life cycle of any complex software or systems engineering product since it may range from 50 to 80% of the total design effort [2]. In addition, designed products are required to meet a very high-level of reliability, security, and performance especially in safety-critical areas. Therefore, ensuring that such systems meet their predefined requirements and that they perform as expected is a challenging issue. However, in many modern engineering disciplines, conventional V&V methods such as testing and simulation have become less useful and are not always applicable. Conversely, using formal techniques complementary to simulation provides a certain level of confidence since they are rigorous and complete.

The research presented hereby, aims at providing a unified paradigm for V&V in software and systems engineering fields. This effort is supported by the Collaborative Capability Definition, Engineering and Management (CapDEM) project which is an R&D initiative within the Canadian Department of National Defence<sup>1</sup>. The latter aims to the development of a Systems-of-Systems engineering process and relies heavily on modeling and simulation. The underlying models need to be subjected to formal V&V as a complementary technique. In this context, our research

<sup>1</sup>This research is the result of a fruitful collaboration between the Computer Security Laboratory (CSL) at Concordia University and Defence Research and Development Canada (DRDC) at Ottawa. The research is supported by the CapDEM (Collaborative Capability Definition, Engineering and Management) project.

provides the means to achieve such an objective.

The main contribution of this paper is to introduce a unified paradigm for V&V in software and systems engineering. It is based on an established synergy between three major techniques: Formal verification, program analysis, and software engineering techniques. By formal verification, we mean mainly model-checking. By program analysis, we mean flow analysis (data and control), type-based analysis, and abstract interpretation. By software engineering, we point out those techniques that are used to measure quality attributes in software such as metrics. Our key idea is to harmoniously combine these techniques in a synergetic way to achieve V&V of software and systems engineering models. Furthermore, to the best of our knowledge, this is a pioneering endeavor in using these three techniques together. In addition, the suggested approach is rigorous, formal and cost-effective since it is entirely automatic.

The remainder of the paper is structured as follows. We start by briefly surveying some related work. Then, Section 3 introduces our approach and shows how it can contribute to high-quality, practical, and formal V&V of software and systems engineering design models. Thereafter, we present the assessment of three case studies. Section 4 is dedicated to the assessment of a sample sequence diagram (interaction view). Section 5 contains the assessment of state machine diagram (behavioral view). In Section 6, we give some insights into metrics and the usage thereof with a relevant example of class and package diagrams. Since the case studies are only meant to depict the structure of some real-life systems, we will not be concerned about the underlying engineering discipline. Finally, we conclude with some remarks and discuss future work in Section 7.

## 2 Related Work

In this section, we survey the state of the art in terms of V&V of software and systems engineering design models. Particularly, we focus on UML 2.0 and SysML design models. It is worthy to mention that only subsets of the syntax and semantics specified by the considered standards are covered in the related work. In the sequel, we present V&V of the most prominent UML diagrams namely state machine, activity, sequence, class and package diagrams.

Several approaches to the verification of UML state machine diagrams have been advanced. D. Latella et al. [13] as well as E. Mikk et al. [15] address the formal verification of a subset of UML state machine using SPIN, based on an operational semantics as described in [14]. The translation process is done in two phases. First, the state machine is converted into an Extended Hierarchical Automaton (EHA). Then, the latter is modeled in PROMELA and subjected to model-checking.

For activity diagrams, several verification approaches

have been proposed. Van der Aalst in [23] used interval timed colored Petri nets [22] as an ascribed semantics to activity diagrams. E. Börger et al. in [3] provided an Abstract State Machine (ASM) semantics for activity diagrams. R. Eshuis et al. [8] mapped activity diagrams to equivalent activity hypergraphs by flattening their structure. Then, activity hypergraphs are mapped to a Clocked Labeled Kripke Structure (CLKS).

In [5], the authors propose a rich trace-based semantics for UML 2.0 interactions (sequence diagram). Harald Störrle presents in [20] a partial order semantics for time constrained interaction diagrams.

Class and package diagrams have been investigated in several research initiatives. In [1], metrics have been used to measure the quality attributes of class and package diagrams. These metrics could be classified into two categories. The first one deals with traditional metrics such as cyclomatic complexity while the second, which is specifically related to object oriented systems, involves metrics such as coupling, depth of inheritance, and the number of children. In [9], the authors illustrated the use of several object oriented metrics to assess the complexity of class diagrams at the initial phases of the development life cycle. More recently, topics like validation by applying audits and metrics to UML models are addressed in [10]. Audits refer to conformance to standards while metrics are viewed as numerical measurements that allow the analysis of a model with respect to an already established scale indicating the quality attributes of the design.

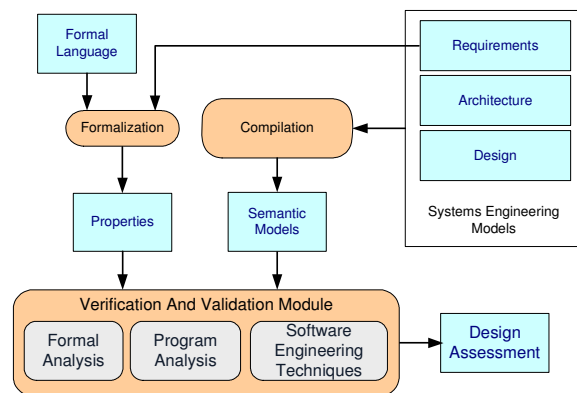
## 3 Approach

As previously stated, the main trait of our V&V paradigm lies in the harmonious synergy between three well-established techniques: Formal verification, software engineering techniques, and program analysis. The experiments that we conducted demonstrate the viability of this approach as it will be exemplified later in this paper. In Figure 1, we depict the synoptic of our approach.

The foundation of our paradigm is sustained by three distinctive layers. First, model-checking, as a formal technique, can be fully automated and has been successfully used in the verification of real applications (e.g. digital circuits, communication protocols, etc). Hence, for each behavioral diagram, we derive a formal semantic model reflecting its characteristics and we express the properties that the design must satisfy as temporal logic formulas. A widely used model-checker within the scientific community is SMV. It has interesting features like branching time logic for expressing properties (CTL). We are currently using NuSMV [6] model-checker (a modified version of the original SMV) that has been used in the verification of hardware and other systems modeled as FSMs (e.g. software).

The second layer consists of fifteen metrics adopted from software engineering field and applied directly on UML and SysML models. Thus, they can be used to assess quality attributes of various models independently of their underlying discipline (software or systems engineering). For instance, G.C. Tugwell et al. [21] outline the importance of metrics in systems engineering especially those related to complexity measurement. Furthermore, the metric concept can also be applied on the semantic model derived from different behavioral diagrams. Some examples include cyclomatic complexity, length of critical path, and others. Thus, the quality assessment of a given design can combine both the static and dynamic perspectives.

The third layer is represented by program analysis techniques such as flow analysis, type systems, and abstract interpretation. We advocate their use in verifying important model properties such as data dependencies, control dependencies, invariants, anomalous behavior, reliability, and compliance to certain specifications.



**Figure 1. Synoptic of the V&V Process.**

In the following, we briefly present some considerations with respect to the behavioral diagrams and their corresponding semantic models. Any system that exhibits a dynamic of some kind can be abstracted to one that evolves within a discrete state space. Such a system is able to evolve through its state space assuming different configurations where a configuration is understood as the set of states wherein the system abides at any particular moment. All the possible configurations summed up by the dynamics of the system and the transition thereof can be coalesced into a configuration transition system (CTS) or more simply put a configuration system. The latter is characterized by a set of configurations, a transition relation and a set of events.

Moreover, any CTS can be processed such that it can be graphically visualized using an external graph drawing tool (e.g. daVinci [4]). The latter can be used to follow the trace

provided as a counterexample for the unsatisfied properties. Additionally, the graph provides a visual appraisal of the diagram complexity with respect to the number of nodes and edges of the corresponding CTS. Furthermore, it can also be used as a quick feedback when applying corrective measures giving some insights about the resulting increase or decrease in the diagram complexity.

The NuSMV code generated for a given CTS will slightly vary depending on the type of the original diagram. This is necessary in order to trace back the unsatisfied properties to the original diagram. In order to grasp the idea, we give the following edifying example. Though one might initially think that it is pretty sound to consider that each configuration can represent a distinct entity in the NuSMV model, one would quickly stumble on some shortcomings when considering the case of a CTS derived from a state machine diagram. The fact that the states within a state machine diagram share a hierarchical containment relation can lead one to portray a scenario wherein a compound state is never exited while allowing for transitions among its sub-states. It follows that we have a state that when reached, is never left, though the system is continuously progressing. In such a case, one would be unable to specify a property that would detect such a problem without considering an independent entity for every state that might belong to different configurations of the corresponding CTS. Equivalently, one can specify only the individual states as entities in the transition system and not the configurations themselves. This amounts to expressing the evolution of each state by specifying all the conditions required for its activation or deactivation. Thus, even models exhibiting a wild dynamics and having complex CTS graphs, can be efficiently subjected to model-checking.

Accordingly, the core component responsible with the semantic model generation procedure represents an important aspect of our framework. However, the main intent of this paper is to present our results when applying the proposed paradigm for V&V on different UML diagrams. Thus, we will concentrate in the remainder of the paper in presenting some relevant case studies without detailing the theoretical aspects underlaying the procedure that is used for obtaining the corresponding configuration systems of the behavioral diagrams.

### 3.1 System Aspects and System Properties

There are many systems engineering aspects that we target: Requirements, time, concurrency, structure, interface, control, and performance. In the sequel, we briefly present the main ones:

- *Requirements*: They are a description of what a system should do and are captured by requirement diagrams in SysML or using sequence and use case diagrams.

- **Concurrency:** It identifies how activities, events, and processes are composed (sequence, branching, alternative, parallel, etc.). It could be specified using sequence, activity, and timing diagrams.
- **Performance:** It is the total effectiveness of the system. It makes reference to the timeliness aspects of how systems behave including different types of quality of service characteristics like latency and throughput. Timing and sequence diagrams depict performance aspects.
- **Structure:** It is shown in class and composite structure diagrams. The class diagram shows the relationships between different classes of the system. The composite structure diagram shows the internal structure of the building blocks of the system and how these blocks are interfacing with other components of the system.

Verification and validation contribute to the design assessment by detecting the unsatisfied properties. Hence, system developers will know if the design is flawed and apply corrective measures. We consider the following properties in our V&V framework: Latency, liveness, safety, deadlock, livelock, precedence, complexity, maintainability, reusability, coupling, and cohesion. Due to space limitations, we present the main ones, as follows:

- **Latency:** It is the measure of the temporal delay between the request for the execution of an operation and the reply to this request. Detecting latency contributes to V&V by analyzing the efficiency of the system.
- **Liveness:** It asserts that under certain conditions, a given event will occur. It is known as “something good will always happen”. Liveness analysis consists of checking whether some important or crucial events may or may not eventually happen in the system.
- **Safety:** It means that nothing bad can occur with respect to the design of the system. In other words, it is a judgment of the acceptability of risk, which implies that no harm will occur under the specified conditions.
- **Deadlock:** It describes a state wherein a process is waiting for some event that will never happen. It could be waiting for a resource that another process is holding indefinitely. Thus, the system would not progress.
- **Reachability:** It consists of checking whether a particular state is reachable in a design, starting from an entry point. Unreachable states negatively impact the quality design since they denote dead entities.
- **Complexity:** It is the quality of being intricate and compounded, measuring the degree to which a design is difficult to be understood and/or to be implemented.
- **Reusability:** It measures the easiness and rapidity with which a part (or more) of a system design and/or implementation can be reused.
- **Coupling:** It measures how strongly system parts depend on each other. Generally, a loose coupling is sought in a high-quality design. Moreover, there is a strong correlation between coupling and other quality attributes (e.g. complexity, reusability, etc).
- **Cohesion:** It refers to the degree to which system components are functionally related. Generally, a strong cohesion is sought in a high-quality system design.

### 3.2 V&V Framework

Our V&V framework requires an underlying modeling tool wherefrom various models can be fetched and assessed. We chose ARTiSAN Real-time Studio [18] which is a modeling tool that supports UML and SysML designs. Additionally, it provides component-based development specifically for real-time systems. The current version of our framework is composed of three core components, as shown in Figure 2. First, we have the semantic compilation component responsible for deriving the semantic model of a specific diagram. It communicates with the model-checker by providing the semantic model along with the properties to be verified. Second, we have the metric computation component that is used for applying metric algorithms. We have provided an interface that accesses the object repository of the modeling tool and retrieves the needed information about the diagrams. Finally, the assessment results component is devoted to the presentation of interpreted results. Should a specified property fail, the trace provided by the model-checker is analyzed and the relevant information is provided as feedback.

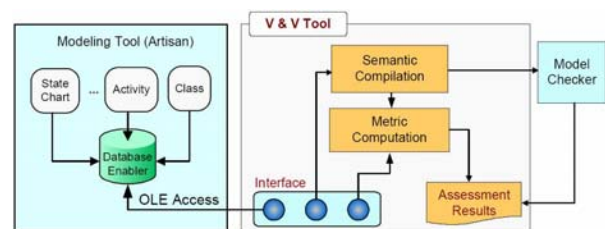


Figure 2. Architecture of the Framework.

In the following paragraphs, we give a glimpse about V&V with respect to class, package, and state machine diagrams. Subsequently, we detail the employed methodology for the V&V of the behavioral diagrams.

The quality of an object oriented system depends on different attributes such as complexity, understandability,

maintainability, stability, and others. According to the type of diagram, we have a class of metrics for structural diagrams and another for behavioral ones. In the literature, many metrics were developed to measure the quality of software systems, especially for structural diagrams namely class and package ones. However, until now, they were neglected in the verification and validation of systems engineering designs. We adopted a set of fifteen metrics including *Coupling Between Object classes* (CBO), *Depth of Inheritance Tree* (DIT), and *Instability* (I). CBO measures the interrelationship between objects. DIT measures the level of a class in the class inheritance hierarchy. The instability metric (I) measures the rate of instability of a package. A package is unstable if it depends more on other packages than they depend on it. With the respect to behavioral diagrams, we are currently in early stages of experimenting with metrics like the cyclomatic complexity and the length of critical path. However, nominal ranges can not be absolute but are tailored to the intended system's characteristics such as size, specialization, and redundancy.

Concerning the behavioral diagrams, we addressed the V&V of sequence, activity, and state machine diagrams. We will illustrate our methodology with respect to verifying behavioral diagrams in the case study sections for sequence and state machine diagrams. After converting each diagram into its corresponding semantic model, we can automatically specify CTL properties for deadlock and reachability. Manual specification of properties can also be specified by using macros, with intuitive names (e.g. ALWAYS, MAYREACH etc.), that are systematically expanded to corresponding CTL properties. Thus, the designers can easily express properties without being required to know formal logics or temporal formulas.

The sequence diagram depicts object interactions arranged in a time sequence. Furthermore, it can be used to capture attributes such as latency and precedence. Using sequence diagrams, one can check whether a system meets some specified properties by techniques such as reachability analysis and model-checking. It follows that by extracting all possible execution paths of a given sequence diagram, we can construct a corresponding transition system that can be analyzed by a model-checker.

State machines can be used to specify the behavior of the various elements that are going to be modeled. A state machine is a specification that thoroughly describes all possible behaviors of some dynamic model. We were initially inspired by Latella et al. [13], however, we preferred the NuSMV model-checker (since it supports branching time). We also recently succeeded in efficiently mapping the state machine diagram to a kind of Labeled Transition System in the form of CTS. This allowed us to unify the semantic model of the behavioral diagrams. In the sequel, we give some insights with respect to our methodology and present

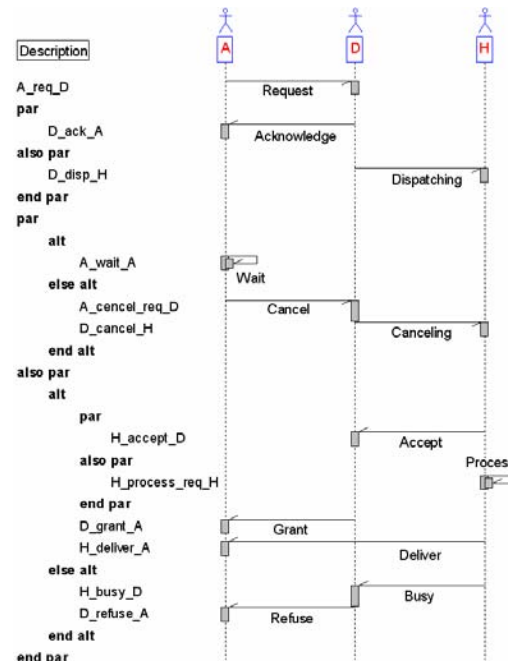


Figure 3. Sequence Diagram Example.

the analysis of the state machine case study.

## 4 Sequence Diagram Case Study

Figure 3 shows a sample sequence diagram that we use as case study. The diagram encompasses three actors interacting in order to accomplish an arbitrated request/delivery service. The actors are the requestor, the dispatcher, and the handler. The envisioned interaction is as follows: the requestor (A) is sending a request to the dispatcher (D). After receiving the request, D is acknowledging A while dispatching at the same time (in parallel) the request to the handler (H). Subsequently, we have the parallel composition of two groups of messages. The first group is composed of an alternative between two cases: in the first case, A is sending itself a wait message; in the second case, A is sending to D a cancel message, followed by a canceling message of the dispatched request from D to H. The second group of messages is composed of an alternative between two other cases: in the first case, H is sending to D an accept message in parallel with a self processing message, followed by a grant message that D is sending to A and a subsequent delivery message from H to A. In the second case, H is sending a busy message to D, followed by the refuse message from D to A.

In order to assess the diagram, we convert it to a cor-

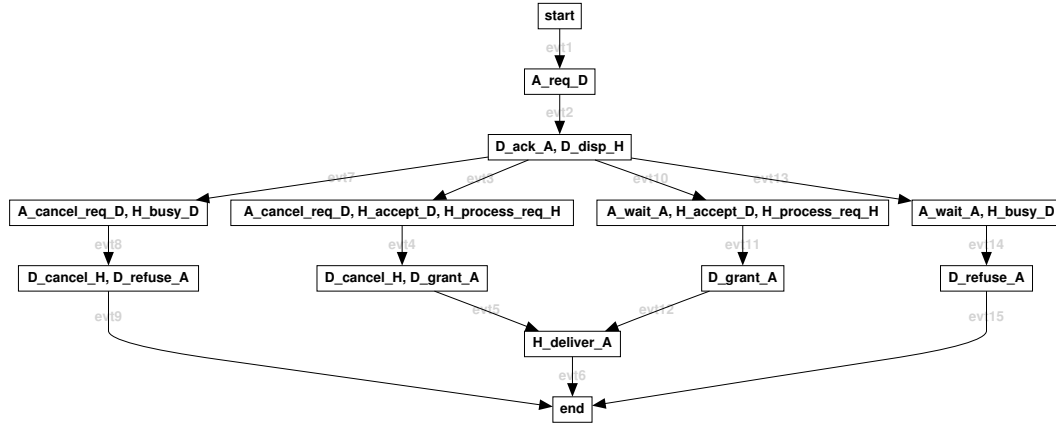


Figure 4. CTS of the Sequence Diagram.

responding semantic model which is a kind of a transition system (configuration system) wherein the elements represent sets (possibly singleton) of interaction messages. In order to construct it, we first explore all possible execution paths of the diagram. The corresponding CTS is depicted in Figure 3.

For the sequence diagram, we are mainly interested in the manual property specification as aside from deadlock and reachability, one can not infer automatically the required properties to be verified. Thus, we will exemplify the usage of manual specification of properties. For the presented sequence diagram, we present some relevant properties that reflect the intended interactions. We have the following properties presented in both simplified macro notation along with their corresponding CTL notation.

The first one asserts that it is always the case that if A is sending a request to D, then there should be case where a corresponding grant will be sent from D to A:

ALWAYS A\_req\_D  $\rightarrow$  MAYREACH D\_grant\_A  
 CTL: SPEC AG((A\_req\_D  $\rightarrow$  E[!(end) U D\_grant\_A]))

The second one asserts that it is always the case that if D is sending a grant message to A, then it should be possible for H to deliver to A at the next step:

ALWAYS D\_grant\_A  $\rightarrow$  POSSIB H\_deliver\_A  
 CTL: SPEC AG((D\_grant\_A  $\rightarrow$  EX(H\_deliver\_A)))

The third one is stating that it is always the case that if A is canceling the request to D, then the system may not reach the point where the handler H is sending to A:

ALWAYS A\_cancel\_req\_D  $\rightarrow$  !MAYREACH (H\_deliver\_A)  
 CTL: SPEC AG((A\_cancel\_req\_D  $\rightarrow$  !(E[!(end) U H\_deliver\_A])))

When subjecting the sequence diagram to our V&V tool, we found that only the first two properties are satisfied. However, the model-checker was able to produce a coun-

terexample for the third user specified property. The interpreted result of the trace provided by the model-checker consisted in the following path in the CTS:

A\_req\_D; D\_ack\_A, D\_disp\_H;  
 A\_cancel\_req\_D, H\_accept\_D, H\_process\_req\_H;  
 D\_cancel\_H, D\_grant\_A; H\_deliver\_A

The identified path contains a series of messages (semicolon separated) that are exchanged between the actors. For readability reasons, we have by convention each message label starts with the sender actor and terminates with the receiving one. Also, whenever two or more messages are being sent in parallel (thus part of the same configuration in the transition system), a comma is separating them.

## 5 State Machine Diagram Case Study

In this section, we present an interesting example that represents an abstracted model for an Automated Teller Machine (ATM) system. As depicted in Figure 5, the model is based on a hypothetical behavior and is meant only as an example that outlines the usefulness of our proposed V&V paradigm. The diagram has several states that are going to be presented in accordance to the diagram containment hierarchy. The top state ATM is composed of four substates: idle, verif, eject, and operation. Initially, the system waits for a potential user to use the ATM in the idle state. The verif state represents the verification of the card validness and authorization. The eject state depicts the phase of termination of the user transaction. The operation state is also a composite state that includes the states that capture several functions related to banking operations. These are the account, payment and transac. The account state is where the account corresponding to

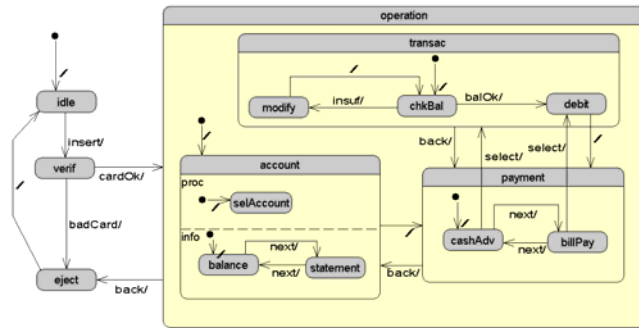


Figure 5. ATM State Machine Diagram.

the card is selected and the user can receive information about balance and statement. The `payment` state has two substates for cash advancing and bill payment respectively. Finally, the `transac` state captures the transaction phase and includes three substates for checking the balance, modifying the amount (if necessary) and debiting the account.

When applying our paradigm to assess the presented state machine diagram, the steps that are being followed are similar to those involved in assessing the sequence diagram. That is, we first converted the diagram to its corresponding semantic model which in this case is a CTS, as depicted in Figure 6, wherein each element is represented by a set (possibly singleton) of states of the state machine diagram. Thereafter, we automatically specify deadlock and reachability properties for every state. Furthermore, we also provide some manual property specification in both macro and CTL notations. After finishing the model-checking procedure for the automatically generated properties, the results that have been obtained pinpointed some interesting problems in the ATM state machine design.

The model-checker determined that the `operation` state exhibits deadlock, meaning that once entered, it is never left. This was found to be caused by the fact that in UML state machine diagrams, the transitions that have the same trigger are given higher priority when the source state is deeper in the containment hierarchy. Moreover, the transitions that have no event are fired as soon as the state machine reaches a configuration that is containing the corresponding source state. This is precisely the case with the transition from `account` to `payment`. Thus, there is no case that allows the `operation` state to be exited. This can also be seen by looking at the corresponding CTS where we can notice that once a configuration that contains the `operation` state is reached, there is no transition to a configuration that does not contain it.

Another property that failed was with respect to the reachability of state `statement`. The problem is caused by the fact that the transition from `account` to `payment`

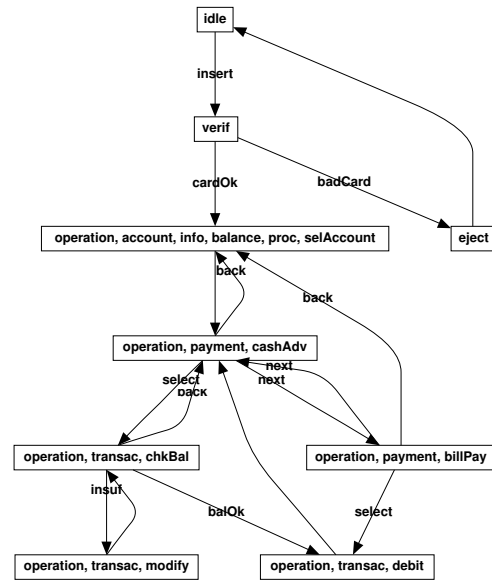


Figure 6. CTS of the ATM.

is taken as soon as there is a configuration containing the `account` state (there is no trigger labeling the transition).

In addition to the automatically generated properties, we have the following manual specifications that represent an intended behavior. We subsequently present both the macro and the CTL notations.

The first property states that it is always the case that if `verif` is reached then it should be possible to reach `operation` at the next step:

ALWAYS `verif`  $\rightarrow$  POSSIB `operation`  
 CTL: SPEC AG((`verif`  $\rightarrow$  EX(`operation`)))

The second one asserts that it is always the case that if `operation` state is reached then from that point `payment` state should be reachable:

ALWAYS `operation`  $\rightarrow$  MAYREACH `payment`  
 CTL: SPEC AG((`operation`  $\rightarrow$  (E[!(`idle`) U `payment`])))

The next property asserts that it is always the case that after reaching state `operation` it should be unavoidable to reach state `eject` at a later point:

ALWAYS `operation`  $\rightarrow$  INEVIT `eject`  
 CTL: SPEC AG((`operation`  $\rightarrow$  (A[!(`idle`) U `eject`])))

The last one states that `chkBal` must precede `debit`:

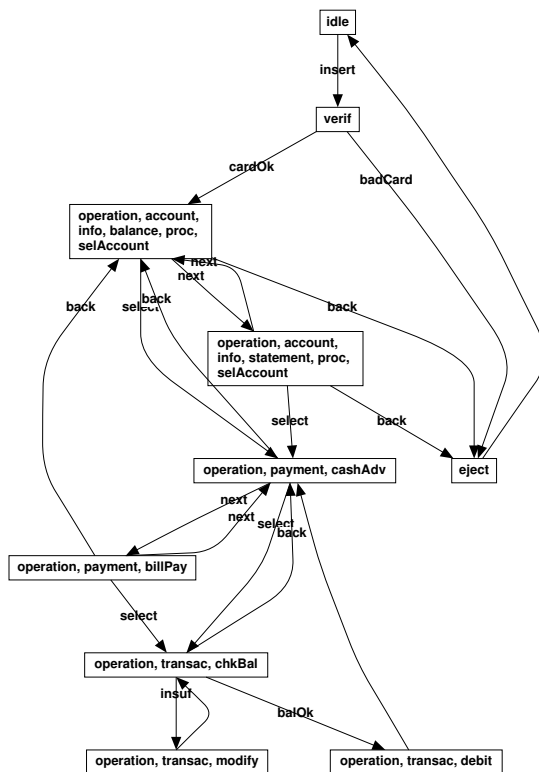
`chkBal` PRECEDE `debit`  
 CTL: SPEC (!E[!(`chkBal`) U `debit`]))

The first two manually input specifications turned out to be satisfied when running the model-checker. However,

the last two properties failed. The failure of the third one was not unexpected as from the automatic specifications we noticed that state `operation` has deadlock and does not have state `eject` as a substate. The failure of the last property was accompanied by the following trace provided by the model-checker. Though the model-checker provided other counterexamples for previous failed properties, we present this last one as it captures a critical unintended behavior:

```
idle;
verif;
operation, account, info, balance,
proc, selAccount;
operation, payment, cashAdv;
operation, payment, billPay;
operation, transac, debit
```

The presented counterexample follows exactly the same notation as the sequence diagram counterexample. For the



**Figure 7. CTS of the Fixed ATM.**

presented state machine diagram, we also provide the necessary changes that will fix the problems identified by the model-checker. The first modification consists in adding

a trigger such as `select` to the transition from state `account` to state `payment`. This will fix both the deadlock and the reachability problems. The second modification targets the last manual specification and consists in changing the target of the transition from state `billPay` to state `debit` to a new target state namely `transac`. After re-executing the verification phase for the fixed diagram, a modified CTS was obtained, depicted in Figure 7, and all the specifications, both automatic and the manually specified properties were satisfied.

## 6 Class and Package Diagrams Case Study

The proposed example of class and package diagrams, depicted in Figure 8, illustrates a real-time heart monitoring system consisting of three packages. One contains the windows components that display the monitoring results. Another package contains the platform specific heart monitoring tools, whereas the last one contains the heart monitoring components. This is a good example to be tested due to the different types of relationships among the classes. Our tool implements a set of fifteen metrics [11] for class and package diagrams. In the following, we briefly present our results when applying these metrics on this diagram. The analysis results indicate that some classes are rather complex thus having a weak reusability potential.

Package Name	A	I	DMS
Platform Specific HM	0	49	51
HM	0	49	51
Windows Components	0	50	50
Average	0	50	50
Nominal Range	-	-	50 - 100%

The table above shows the metrics related to package diagrams. The distance from the main sequence metric (DMS) measures the balance between the abstraction and instability levels in the package. As shown in the table, the three packages in the diagram fall within the nominal range of DMS. However the *Abstraction* and *Instability* metrics do not have nominal ranges due to the difference in the design perspectives. Thus, DMS is a compromise between their values. In the same table, the zero abstractness value for the three packages shows a weak potential for extendibility and modifiability. Also, the elevated value of the instability metric in the second column indicates that the three packages are subject to change if other packages do change.

Table 1 presents the analysis results of the class diagram inheritance related metrics. The Depth of Inheritance Tree (DIT) metric shows a proper use of inheritance as it has no negative impact on the complexity level of the diagram. Furthermore, our results show that the diagram has a shallow inheritance tree indicating a good level of understandability and testability. With respect to the Number Of Children (NOC), the analysis shows that only four classes in the

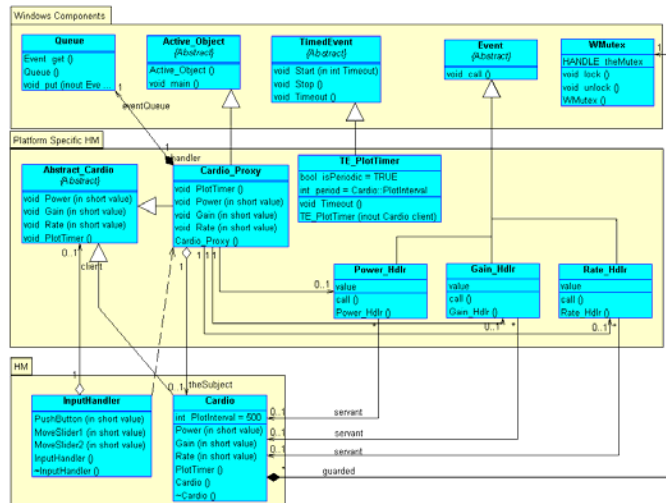


Figure 8. Class and Package Diagrams Example.

Class Name	DTI	NOC	NOM	NOA	NMA	NMO
Cardio	1	0	10	1	6	4
Rate_Hdlr	1	0	3	1	2	1
Gain_Hdlr	1	0	3	1	2	1
Power_Hdlr	1	0	3	1	2	1
TE_PlotTimer	1	0	5	2	2	3
Cardio_Proxy	1	0	11	0	5	4
Abstract_Cardio	0	2	4	0	4	0
WMutex	0	0	3	1	3	0
Event	0	3	1	0	1	0
TimedEvent	0	1	3	0	3	0
Active_Object	0	1	2	0	2	0
Queue	0	0	3	0	3	0
Input_Handler	0	0	5	0	5	0
Average	0.46	0.88	4.31	0.54	3.08	1.08
Nominal Range	1 - 4	1 - 4	3 - 7	2 - 5	0 - 4	0 - 5 %

Table 1. Class Diagram Inheritance Metrics

Class Name	CR	CCRC	CBO	PMR
Cardio	0	200	1	100
Rate_Hdlr	0	200	1	100
Gain_Hdlr	0	200	1	100
Power_Hdlr	0	200	1	100
TE_PlotTimer	0	100	0	100
Cardio_Proxy	0	700	5	100
Abstract_Cardio	0	100	1	100
WMutex	0	0	0	100
Event	0	0	0	100
TimedEvent	0	0	0	100
Active_Object	0	0	0	100
Queue	0	0	0	100
Input_Handler	0	200	2	100
Average	0	146	0.92	100
Nominal Range	20 - 75%	150 - 350%	1 - 4	5 - 50%

Table 2. Class Diagram General Metrics

diagram have a good NOC value. In a class diagram, the number of children is an indication of the class reusability.

The analysis results also show that five classes have weak Number Of Methods (NOM) value. On the other hand, the class diagram has an overall NOM that lies in the nominal range. The problem of unsuitable NOM values may be solved by modifying the class diagram by decomposing the existing classes to smaller new classes to share the number of methods that exceed the nominal range. Consequently, classes in the class diagram will be more reusable. Table 1 shows only one class in the Number Of Attributes (NOA) nominal range. This invites to further enhancement by adding new attributes to the non-abstract classes, though the size of the class increases with its number of attributes.

The Number of Methods Added (NMA) measures the inheritance usefulness degree. Three classes have a high NMA value indicating a misuse of inheritance. Classes with

high NMA may be difficult to reuse, whereas, classes with no specialization and having large number of methods may impede other classes from reusing their functionality requiring the decomposition into smaller specialized classes in order to improve the design.

The Coupling Between Object (CBO) classes metric measures the level of coupling between classes, denoting an increase in the complexity for high coupling. Table 2 shows seven classes outside the CBO nominal range while six classes are falling within it. This shows an increased complexity and suggests further modification by reducing the number of relationships between the classes.

The Class Category Relational Cohesion (CCRC) measures the cohesion of classes within the diagram. This metric reflects the diagram's architecture strength. Table 2 shows a good CCRC level for only five classes whereas the remaining eight classes have a weak CCRC level. We

can also see that the average CCRC is outside the nominal range indicating a lack of cohesion between the classes. The Class Responsibility (CR) results in Table 2 show that none of the classes in the diagram is implementing pre-conditions and/or post-conditions. CR is measured in the cases when a class method should be responsible to check whether a message is correct before taking any action. In the current example, the CR value can be enhanced by adding pre/post-conditions to the methods that need to check the validity of messages prior or after the execution of an action. In the design of a class diagram, the use of pre/post-conditions should be carefully considered. Therefore, this metric is useful to check systems with real-time messaging.

Finally, Table 2 shows that all methods in the class diagram are accessible, which inhibits encapsulation in the diagram. This requires some adjustments for the access control level for all the classes in the diagram.

## 7 Conclusion and Future Work

In this paper, we proposed a unified paradigm for V&V in software and systems engineering with three distinctive features: model-checking, program analysis, and software engineering techniques. With respect to the latter, we showed a real life example of class and package diagrams assessment. Also, we showed two case studies for sequence and state machine diagrams respectively accompanied by the analysis based on formal verification and the benefits of our approach in term of the assessment results.

As future work, we intend to cover the complete set of UML and SysML behavioral diagrams using the presented methodology. Moreover, we intend to fully pursue the integration of the metrics related to the semantic models that are derived from various behavioral diagrams.

## References

- [1] N. Aeronautics and S. Administration. Software Quality Metrics for Object Oriented System Environments. Technical Report SATC-TR-95-1001, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt Maryland 20771, JUNE 1995.
- [2] I. Averant. Static Functional Verification with Solidify, a New Low-Risk Methodology for Faster Debug of ASICs and Programmable Parts. Technical report, Averant, Inc.s, 2001.
- [3] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In *AMAST*, pages 293–308, 2000.
- [4] U. Bremen. `udraw(graph)tool`. <http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html>.
- [5] M. V. Cengarle and A. Knapp. UML 2.0 Interactions: Semantics and Refinement. In *3rd Intl. Workshop on Critical Systems Development with UML (CSDUML '04, Proceedings)*, pages 85–99. Technische Universität München, 2004.
- [6] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A New Symbolic Model Verifier. In *cav*, 1999.
- [7] C. Committee. The International Council on Systems Engineering (incose). <http://www.incose.org/practice/whatissystemseng.aspx>.
- [8] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.
- [9] M. Genero, M. Piattini, and C. Calero. Early Measures for UML Class Diagrams. *L'OBJET*, 6(4), 2000.
- [10] R. Gronback. Model Validation: Applying Audits and Metrics to UML Models. In *BorCon 2004 Proceedings*, 2004.
- [11] U. R. Group. Object Oriented Model Metrics. Technical report, The United States Air Force Space and Warning Product-Line Systems, <http://www.cin.ufpe.br/inspector/relacionados/Object-orientado%20Model%20Metrics%20Document.htm>, 1996.
- [12] IEEE. Standard 610.12.
- [13] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart diagrams using the spin model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.
- [14] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465. Kluwer, B.V., 1999.
- [15] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 90. IEEE Computer Society, 1998.
- [16] O. M. G. (OMG). UML 2.0 Superstructure Specification, 2003.
- [17] S. Partners. System Modeling Language: SysML, 2004.
- [18] A. Software. ARTiSAN Real-time Studio. [http://www.artisansw.com/pdflibrary/Rts\\_5.0\\_datasheet.pdf](http://www.artisansw.com/pdflibrary/Rts_5.0_datasheet.pdf). Datasheet.
- [19] D. E. Stevenson. Verification and Validation of Complex Systems. In *Proceedings of ANNIE 2002, Smart Engineering System Design*. ASME, November 2002.
- [20] H. Störrle. Semantics of Interactions in UML 2.0. In *HCC*, pages 129–136, 2003.
- [21] G. Tugwell, J. Holt, C. Neill, and C. Jobling. Metrics for Full Systems Engineering Lifecycle Activities (MeFuSELA). In *Proceedings of the Ninth International Symposium of the International Council on Systems Engineering (INCOSE 99)*, Brighton, U.K., 1999.
- [22] W. M. P. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In *Application and Theory of Petri Nets*, pages 453–472, 1993.
- [23] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.