International Journal of General Systems
Vol. 38, No. 7, October 2009, 719–746

Synergistic verification and validation of systems and software engineering models

Yosr Jarraya[a]*, Andrei Soeanu[a1], Luay Alawneh[a2], Mourad Debbabi[a3] and Fawzi Hassaïne[b4]

[a]*Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, 1515, Ste Catherine West, EV-7-642, Montreal, Quebec, H3G 2W1 Canada;* [b]*Capabilities for Asymmetric and Radiological Defence and Simulation, Defence Research and Development Canada – Ottawa, 3701, Carling Avenue, Ottawa, Ontario, K1A-0Z4 Canada*

(*Received 12 November 2008; final version received 26 January 2009*)

In this paper, we present a unified approach for the verification and validation of software and systems engineering design models expressed in UML 2.0 and SysML 1.0. The approach is based on three well-established techniques, namely formal analysis, programme analysis and software engineering (SwE) techniques. More precisely, our contribution consists of the synergistic combination of model checking, static analysis and SwE metrics that enables the automatic and efficient assessment of design models from static and dynamic perspectives. Additionally, we present the design and implementation of an automated computer-aided assessing framework integrating the proposed approach. Moreover, we discuss the related technical details and the underlying synergism. Finally, we illustrate the proposed approach by assessing a design case study that is composed of state machine and sequence diagrams.

Keywords: software and systems engineering; UML; SysML; design models; verification and validation; model checking; software metrics

1. Motivations

Modern modelling languages for software and systems, including the most prominent ones, namely UML 2.0 (Object Management Group 2003) and SysML 1.0 (Object Management Group 2006) emerged in order to cope with the continuous advancement in software and systems design. The aforementioned modelling languages are playing an increasingly important role in software and systems engineering (SE) processes. Software engineering can be defined as the application of a systematic approach to the development, operation and maintenance of software (IEEE 1990), while SE represents an interdisciplinary approach that enables the realisation of successful systems focusing on the system as a whole (INCOSE 2004). Ubiquitous systems such as hi-tech portable electronics, mobile devices, automated teller machines (ATMs) as well as many other advanced technologies present in aerospace, automotive or telecommunication platforms represent important application fields of SE. However, nowadays the critical aspect of software and systems design is not represented by conceptual difficulties or technical shortcomings, but it is rather related to the increased difficulty of assuring specification-compliant designs.

*Corresponding author. Email: y_jarray@encs.concordia.ca

ISSN 0308-1079 print/ISSN 1563-5104 online
© 2009 Taylor & Francis
DOI: 10.1080/03081070903029253
http://www.informaworld.com

To that effect, the process of design and development mandates a strong, sound and cost-effective Verification and Validation (V&V) process.

Verification is the process of evaluating a system to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase (IEEE 1990). Conversely, validation is defined as the process of evaluating a system to determine whether it satisfies the specified requirements (IEEE 1990). The V&V process can be a major bottleneck in the life cycle of any complex software or SE product since it may take up to 80% of the total development effort (Averant 2001). Additionally, a lot of software and systems are required to meet a very high level of reliability, security and performance especially in safety-critical areas. Therefore, ensuring that their predefined requirements are met and that they behave as expected often represent some very challenging issues. However, in many modern engineering disciplines, conventional V&V methods such as those involving testing and simulation are not always the most adequate. Conversely, using automatic and more exhaustive techniques (e.g. model checking) complementary to simulation provides a higher level of confidence since they have rigorous and well-defined bases.

Our main objective is to derive a unified approach for the V&V of design models in software and SE. The underlying models need to be subjected to an efficient V&V process, involving formal techniques complementary to simulation. Our approach for V&V in software and SE, briefly presented in Alawneh *et al.* (2006), is based on a proposed synergy between three major techniques: formal verification, programme analysis and SwE techniques. By formal verification, we mean model checking. By programme analysis, we mean data and control flow analysis among other techniques. By SwE techniques, we mainly intend to use software metrics. Specifically, for assessing the quality of a design from the structural point of view, we advocate the use of empirical methods such as metrics, which are used extensively to measure quality attributes of object-oriented software design. Conversely, with respect to the dynamic behaviour, model checking turns out to be an appropriate choice to assess behavioural aspects since it is automatic, exhaustive and has a solid mathematical basis. Finally, in order to address model checking scalability issues, we propose the use of program analysis techniques, such as control and data flow analysis, which may tackle the state space explosion problem while restricting the verification scope to specific properties in the model. More details with respect to this issue are provided in Section 3.

The benefits of the proposed approach are manifold. First, our approach inherits rigour from the use of formal techniques. Moreover, it is cost-effective since it is applied in the early stages of the development process, namely during the design phase. This is because early and efficient identification of flaws in the design can have significant economical advantages if compared to the same task done during the maintenance phase (Boehm and Basili 2001). Furthermore, it is entirely automatic, thus requiring no related background for systems engineers. In addition, different qualitative and quantitative attributes can be measured using SwE metrics in order to assess the overall design quality. Also, to the best of the knowledge of the authors, this is a heretofore unattempted endeavour in using these three techniques synergistically combined in a comprehensive and automated V&V framework.

The rest of the paper is structured as follows. We survey the related work in Section 2. Section 3 presents our approach with details related to our methodology and our V&V framework. Section 4 is dedicated to the assessment of an ATM design case study that is composed of state machine and sequence diagrams. The case study is meant to depict the structure of a hypothetical real-life system. Finally, we conclude by discussing our contributions and future work in Section 5.

2. Related work

In this section, we survey the state of the art in terms of V&V of software and SE design models. Particularly, we focus on UML 2.0 and SysML 1.0 design models. We present hereafter the research initiatives concerning three of the most important and widely used UML diagrams (Ambler 2004), namely state machine, sequence and class diagrams.

Concerning the V&V of SysML-based design models, to the best of our knowledge, there are only a few initiatives (Viehl *et al.* 2006; Jarraya *et al.* 2007). In Jarraya *et al.* (2007), a proposal is advanced for performance evaluation of systems given their design models expressed as SysML activity diagrams endowed with time annotation. The discrete-time Markov chains model is considered as a semantic interpretation of such diagrams. The approach is based on probabilistic model checking of the underlaying probabilistic model using PRISM.[5] Viehl *et al.* (2006) explore formal and simulation based performance analysis of systems specified with UML2/SysML. The detection of potential conflicts on shared communication resources in a system based on its target architecture is addressed. This is based on defining the global timing behaviour of the system and the related violations. Time-annotated UML 2.0 sequence diagrams are considered together with UML 2.0 structured classes/SysML assemblies for describing the system architecture.

Few approaches studying the assessment of the UML 2.0 state machine diagrams have been advanced. Fecher *et al.* (2005) present an attempt to define a structured operational semantics for UML 2.0 state machine in terms of sets and relations. This work deals with major issues in UML 2.0 features including shallow and deep history, join and fork pseudostates together with entry/exit actions and do-activity. However, junction and choice pseudostates, completion event/transition and communicating state machines are not considered. Similarly, Zhan and Miao (2004) propose a formalisation using the general purpose Z language. The semantic model is then used to transform state machine diagram into a flattened regular expression state model. The latter is helpful in identifying issues related to inconsistency and incompleteness and also in automatically generating test cases. The paper takes into account well-formed state machine diagrams including simple and composite states, concurrent and non-concurrent substates, simple and compound transitions as well as firing priority among conflicting transitions. Nevertheless, pseudostates such as fork/join and history are not considered.

Though we are interested in UML 2.0, it is worth mentioning other related work concerning statecharts (renamed state machines in UML 2.0) described in earlier version of UML. These approaches can be classified according to the used model checker. Some authors use SPIN (Holzmann 1997), while others prefer the well-known symbolic model verifier (SMV) (Hsin-Hung 2003). Latella *et al.* (1999a) as well as Mikk *et al.* (1998) propose a translation of a subset of UML statecharts into SPIN/PROMELA using an operational semantics as described in Latella *et al.* (1999b). The translation process is done in two phases. First, the statechart is converted into an extended hierarchical automaton. Then, the latter is modelled in PROMELA and subjected to model checking. Knapp *et al.* (2002) present a prototype tool, HUGO/RT, for the automatic verification of a subset of timed state machines and time-annotated collaborations UML 1.x diagrams. The model checker UPPAAL is used to verify state machine diagrams (compiled to timed automata) against the requirements described in the collaboration diagrams (compiled to observer timed automaton).

Concerning sequence diagrams, Grosu and Smolka (2005) adopt nondeterministic finite automata as their semantic model. A given diagram is translated into a hierarchical automaton and both safety and liveness Büchi automata are derived from it. These automata are subsequently used to define a compositional notion of refinement of UML 2.0 sequence diagrams. Li *et al.* (2004) define a static semantics for UML interaction diagrams to support verifying the well-formedness of interaction diagrams. The dynamic semantics is interpreted as a trace-based terminated communicating sequential process that is used to capture the finite sequence of message calls. Cengarle and Knapp (2004) propose a rich trace-based semantics for UML 2.0 interactions. Störrle (2003) presents a partial order semantics for time constrained interaction diagrams.

Class and package diagrams have been investigated in several research initiatives. In a NASA (1995) technical report, metrics have been used to measure the quality attributes of such diagrams. These metrics could be classified in 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. Genero *et al.* (2000) illustrate the use of several object-oriented metrics to assess the complexity of a class diagram at the initial phases of the development life cycle. More recently, topics like performing V&V by applying audits and metrics to UML models are addressed in Gronback (2004). Audits refer to the compliance 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 of the design.

Other initiatives prefer the use of model checking coupled with simulation (Ober *et al.* 2003, 2006). These emerged in the context of the Information Society Technologies *Omega* project.[6] Ober *et al.* (2003) describe the implementation of the defined semantics, the definition of a property specification formalism, and the application of model checking and simulation techniques in order to validate the design models expressed in the *Omega* UML profile. This is achieved by mapping the design to a model of communicating extended timed automata in IF (Bozga *et al.* 1999) format (an intermediate representation for asynchronous timed systems developed at *Verimag*). Properties to be verified are expressed in a formalism called UML observers, defined in the same paper. In another work, Ober *et al.* (2006) present a case study of a complex system validation, namely the control software of the Ariane-5 launcher. The experiment is done on a representative subset of the system, in which both functional and architectural aspects are modelled using *Omega* UML 1.x profile. The IFx, a toolset built on top of the IF environment, is used for the V&V of both functional and scheduling-related requirements using simulation and model checking functionalities.

Metrics for class diagrams were first initiated by Chidamber and Kemerer (1994), where six metrics are proposed to measure the diagram's complexity with respect to different quality attributes such as maintainability, reusability, etc. Li and Henry (1993) propose a metrics suite to measure several class diagram internal quality attributes such as coupling, complexity and size. Metrics for an object-oriented design (MOOD) suite was proposed by Abreu and Carapua (1994). These metrics are defined at different levels of granularity instead of the class diagram level only. They are used to measure the use of object-oriented concepts such as inheritance, data hiding and polymorphism and to have an appraisal of their impact on the quality of the products. Lorenz and Kidd's (1994) metrics suite measures the static characteristics of software design such as size, inheritance and internal attributes of the class. Last but not least, Briand *et al.*'s (1997) metrics suite is defined at the class level and measures interactions between classes. This set of metrics aims at measuring coupling between classes.

3. Approach

As previously stated, the foundation of our approach lies in the harmonious synergy between three well-established techniques that are: automatic verification (model checking), SwE techniques (metrics) and programme analysis (static analysis). The synoptic overview of the approach is depicted in Figure 1. Our proposal is targeting UML 2.0 or SysML 1.0 SE design models to be assessed along with the related requirements and specifications. However, since these design models are graphically represented, they have to be first encoded into their corresponding semantic model. The latter captures the information and the meaning of the system design and allows us to proceed with an automatic analysis. With respect to the requirements and specifications, they have to be mapped to a set of formal properties in order to be checked on the design model.

The case for the proposed approach can be made based on the following reasoning. A design model can be fully characterised by its structural and its behavioural perspectives. Thus, both perspectives have to be assessed with appropriate techniques. Empirical methodologies, such as those involving software metrics, can help assess the quality of the structural architecture of the design. Complementary automatic verification techniques such as model checking, can achieve a thorough assessment of the model behaviour. However, such exhaustive means of verification require some strategies to cope with the scalability issues (e.g. state explosion). In this context, techniques like programme analysis, mainly data and control flow analysis, have the potential to address these kind of problems. An important remark relates to the fact that the aforementioned techniques are not simply combined, but rather synergistically leveraging each other.

First, model checking, which is a model-based verification technique, has been successfully used for the verification of real applications, both software and hardware systems such as digital circuits, controllers and communication protocols. It is characterised with a higher degree of automation compared to other formal verification techniques (e.g. theorem proving). Examples of model checkers include SPIN (Holzmann 1997), SMV (McMillan 1992) and NuSMV (Cimatti *et al.* 1999). This technique is used to check whether the dynamic aspects of a model satisfy the specified properties (e.g. safety

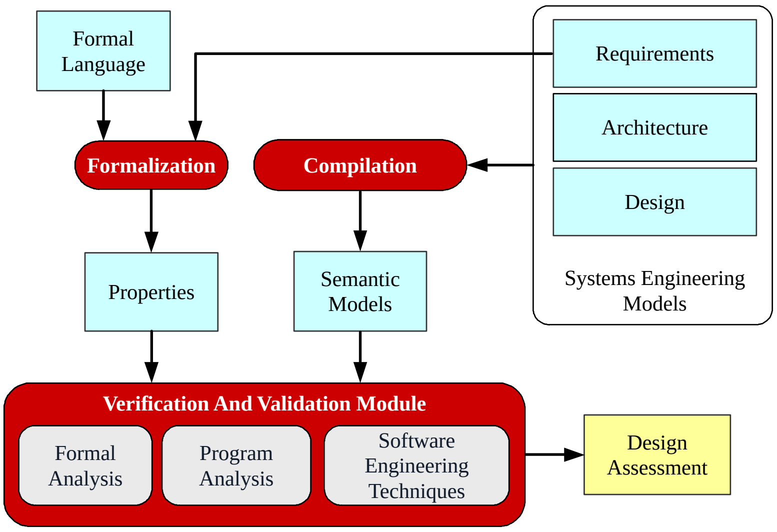


Figure 1. Synoptic overview of the V&V approach.

or liveness). Accordingly, we extract the semantic model from the behavioural diagram we plan to check (e.g. state machine). Model checking has been successfully used in medium-sized complex designs. Even though this technique was generally coupled with severe scalability issues, numerous efforts tackled this problem in various ways, such as on-the-fly model checking (Peled 1994), symbolic model checking (Burch *et al.* 1990) and distributed on-the-fly symbolic model checking (Ben-David *et al.* 2000). In contrast to these techniques, we propose the use of model slicing based on flow analysis (data and control) as a complementary technique. Indeed, programme analysis techniques have been applied successfully in compilation and programme verification and optimisation. Particularly, model slicing can be used in the context of automatic formal verification to cope with scalability issues. The objective is to narrow the scope of the model checking on the part of the model that exhibits the dynamic subject of checking. This idea, which represents the second layer of our approach, will be detailed in Section 4.3.

Finally, the third layer consists in a set of fifteen metrics that we have adopted from SwE (Li and Henry 1993; Chidamber and Kemerer 1994; NASA 1995; Alawneh *et al.* 2006). We advocate their use to assess quality attributes of various models. More precisely, this feature enables us to assess the structural aspects of the systems model. We found in the literature some initiatives about the use of metrics in SE. For instance, Tugwell *et al.* (1999) outline the importance of metrics in SE, especially those related to complexity measurement. In addition to applying metrics on the structural diagrams such as class diagram, we propose their application on the semantic model that is derived from different behavioural diagrams. For example, cyclomatic complexity and length of critical path could be applied on the semantic model. Thus, the quality assessment of a given design can combine both the static and dynamic perspectives. The aim is to be able to compare the structural and the behavioural views qualitatively and quantitatively. For example, comparing the complexity of a state machine diagram and the one of its corresponding semantic model can contrast how close is the diagram structure reflecting the behaviour. If the complexity of the semantic model is less than the one for the corresponding state machine diagram, this can imply that some parts of the structure might be redundant or meaningless.

In the next section, we present our V&V framework and its underlaying components. Moreover, we explain in details the related steps required to assess SE design models, focusing on state and sequence diagrams as the behavioural diagrams and class diagram as the structural ones.

3.1 *V&V framework*

Our V&V framework is intended to be used in conjunction with an SE modelling tool wherefrom various design models can be fetched and subjected to the V&V task. Our current choice is the ARTiSAN Real-time Studio,[7] which is a modelling tool that supports UML and SysML designs. 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 modelling tool and retrieves the needed information about the diagrams. Finally, the assessment results component is devoted to the presentation of the parsed results. Should a specified property fail, the trace provided by the model checker is analysed and the relevant information is provided as a feedback to the designer.

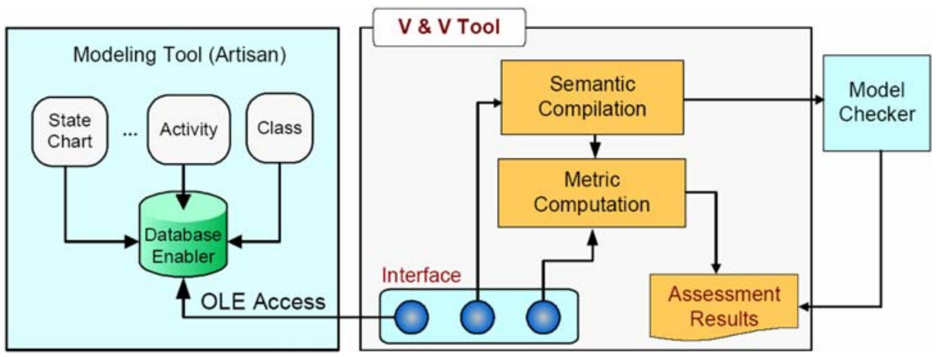


Figure 2. Architecture of the framework.

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 behavioural ones. In the literature, many metrics were developed to measure the quality of software systems, especially for structural diagrams namely class and package. However, until now, they were not considered in the V&V of SE designs. With respect to behavioural diagrams, we are currently in early stages of experimenting with metrics like the cyclomatic complexity and the length of critical path. However, nominal ranges are not absolute but tailored to the intended system's characteristics such as size, specialisation and redundancy.

Concerning the assessment of the behavioural aspects, we address the V&V of design models consisting of sequence, activity and state machine diagrams. In this paper, we elaborate more on our methodology with respect to the assessment of behavioural diagrams, focusing on sequence and state machine diagrams. The main idea is to extract from the studied behavioural diagrams the exhibited dynamics in order to derive the corresponding semantic model. Once the latter is obtained, properties expressed in a temporal logic, computation temporal logic (CTL) (Queiroz 2003) in our case, can be verified on the semantic model using our V&V framework. The latter automatically specifies and verifies CTL properties related to the absence of deadlock and to states reachability. Furthermore, manual specification of customised properties is also possible using intuitive macros, which are automatically expanded to their corresponding CTL format. Thus, the designers can easily express properties without being required to know formal logics or temporal formulas. Once the required properties are specified, the model checker is invoked in order to assess the design.

In the next section, we present the semantic model that we defined and adapted for each of the studied behavioural diagrams. Thereafter, we briefly explain the steps to extract the semantic model from the state machine and sequence diagrams and then we describe the general model checking procedure.

3.2 *Semantic model for behavioural diagrams*

It is generally accepted that any system that exhibits a dynamic behaviour 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 global state wherein the system abides at any particular moment. Hence, all possible configurations summed up by the dynamics of the system and the transitions thereof can be coalesced into what we will henceforth denote as a configuration transition system (CTS). The latter represents the underlying semantic model of the system behaviour. In essence, CTS is basically a form of automaton and it is characterised by a set of configurations that include a set (usually a singleton) of initial ones and a transition relation that encodes the evolution of the CTS from one configuration to another. A configuration depends on the dynamic elements in the system while evolving in its state space. Thus, the CTS definition may be parametrised as to be adapted to the behavioural diagram for which it would represent the semantic model. Hence, the CTS can be used to systematically generate the model checker input in order to assess the dynamics of a given behavioural diagram.

Accordingly, given an instance of a behavioural diagram, one can define the corresponding CTS provided that the following are defined: the set of dynamic elements and the step relation that enables the systematic computation of the next possible configurations from any given configuration. The dynamic elements characterising a behavioural diagram can be abstracted to boolean or bounded range variables, where for instance the true boolean value corresponds to the active status of some elements and the false to the inactive status of some other elements. The set of dynamic elements for which an order relation is established, form the definition of a configuration. For instance, a configuration enclosed within a CTS can be represented by a set of variables that are active simultaneously (bound to the boolean true value), while the transitions are labelled with those elements that are required to trigger the change from the current configuration to the next one. For instance, the dynamic elements of a state machine diagram can be represented by the list of states and guards, which may characterise the corresponding CTS of the diagram. For the special case of the state machine, which is a reactive model, there is also a pre-established set of triggering events that are used to label the transitions. Thus, the list of all states and guards may define all possible configurations (depending on the boolean value bound to the diagram states and guards) whereas, events are used to label the transitions between pairs of configurations.

Definition 3.1. A *configuration* c is a particular binding of boolean or bounded range values to the set of variables included in the dynamic elements of a behavioural diagram arranged according to an established ordering and that characterise the global state of the system at a particular step in its evolution.

Note that the number of configurations summed up by any CTS must be bounded in order to achieve tractability of the semantic model. That is, we have to assume a finite number of dynamic elements in the diagram. Moreover, assuming that each variable v_i needs a finite amount n_i of bits to be represented, then a configuration c belonging to the CTS needs at most $I = \sum n_i$, while the number of configurations is at most 2^I. Notwithstanding, the actual number of configurations is usually much smaller and is restricted to the number of configurations reachable from the set of initial configurations. Furthermore, the dynamic elements of the diagram are in the most of the cases confined to boolean values.

Definition 3.2. A *CTS* corresponding to a given behavioural diagram is a tuple $(C, \Lambda, \rightarrow)$, where C is a set of configurations of the diagram, Λ is a set of labels and

$\rightarrow \subseteq C \times \Lambda \times C$ is a ternary relation, called a transition relation. If $c_1, c_2 \in C$ and $l \in \Lambda$, the common representation of the transition relation is: $c_1 \xrightarrow{l} c_2$.

The CTS structure can also provide useful feedback to the designer. Thus, it can be visualised in a suitable graph editor such as uDraw(Graph) (known also as daVinci).[8] After generating the CTS, it has to be encoded in the input language of the model checker for further processing. The selected model checker for our V&V framework is the NuSMV (Cimatti *et al.* 1999), which is an improved version of SMV (McMillan 1992).

3.3 *Generation of the CTS*

In what follows, we briefly present the necessary steps for generating the CTS corresponding to each type of diagram. For the sake of illustration, where clarity should prevail over completeness, we concentrate only on the state machine and sequence diagrams.

3.3.1 *Derivation of the state machine diagram's semantics*

A state machine diagram is a specification that thoroughly describes all the possible behaviours of some dynamic model. Briefly, a state machine diagram is composed of hierarchically organised states that are related with transitions labelled with events and guards. Each state is either basic or composite. Composite states represent a further aggregation of either sequential or concurrent substates. Each transition can be basically categorised as either simple or inter-level. The former relates states of the same parent, whereas, the latter relates states belonging to different parents. The state machine evolves in response to events that trigger the corresponding transitions provided that the source state is active, the transition has the highest priority, and the guard on the transition is true. If transitions have conflict, priorities are assigned to decide which transition will fire. Higher priority is assigned to those transitions whose source states are nested deeper in the containment hierarchy. Moreover, the hierarchical structure of the state machine diagram can be represented as a tree, where the root is the top state, the basic states are the leaves, and all the other nodes are composite states. The tree structure can be used to identify the least common ancestor of a source and a target state of a transition. This is useful in identifying the states that will be deactivated and those that will be activated after firing a transition.

In the following, we explain the procedure used for the generation of the CTS, presented by Algorithm 1. Basically, we derive the CTS by proceeding iteratively with a breadth-first construction procedure for all possible configurations reachable from a current configuration picked at each iteration from the newly discovered ones. To avoid redundancy, we consider only the active states in the CTS configurations, as the rest of the states are implicitly inactives.

Algorithm 1 aims at constructing the list of all the reachable configurations identified in the state machine, denoted by *CTSConfList* and the list of all the transitions linking these configurations, denoted by *CTSTransList*. We denote by *CurrentConf* the current configuration and *nextConf* its successor relatively to a specific event. As the state machine diagram may contain decision pseudostates, where a guard has to be evaluated to either true or false, it might be possible to reach a list of configurations from the current configuration where we consider both evaluations of the guard. Thus, the configurations of the CTS need to be augmented with guard values corresponding to each configuration. We denote by *NextTransList* the list of the outgoing transitions from the current configuration that

Algorithm 1 Generation of CTS

```
CTSConfList = { } //Contains all the configurations of the resulting CTS
CTSTransList = { }// Contains all the transitions of the resulting CTS
FoundConfList = {initialConf} //List of the configurations that have to be explored
NextTransList = { } //List of transitions originating from the CurrentConf for all the events
EventList //All sensitive events for the state machine
while FoundConfList is not empty do
  CurrentConf = pop(FoundConfList)
  if CurrentConf not in CTSConfList then
    CTSConfList = CTSConfList ∪ CurrentConf
  else
    continue
  end if
  NextTransList = getNext(CurrentConf,EventList)
  for all nextTrans in NextTransList do
    nextConf = getDestination(nextTrans)
    FoundConfList = FoundConfList ∪ nextConf
  end for
  CTSTransList = CTSTransList ∪ NextTransList
end while
```

contains a list of transitions with the corresponding event and the target configuration. We have three functions defined as follows. First, a function *pop*(*configurationlist*) is applied on a list and pops an element of it. Second, a function *getNext*(*configuration*,*event-list*) is an abstraction of the procedure responsible for generating all possible transitions outgoing from the current configuration having the event list. Thus, it returns a possibly singleton list of transitions. Moreover, this function deals with the decision pseudostates by considering all possible values of the guards. Whenever we have an event triggering a transition that reaches a decision pseudostate, additional configurations will be generated for the corresponding possible selection paths. Third, a function *getDestination*(*transition*) that computes the next configuration given a specific transition.

The iterative procedure starts with *FoundConfList* containing only the initial configuration of the state machine, denoted by *initialConf*. At each iteration, a configuration is popped from *FoundConfList* and assigned to *CurrentConf*. If it is not already a configuration in *CTSConfList*, it has to be added to it. Based on a list of possible incoming events referred to as *EventList* and the current configuration, the function *getNext* provides all the transitions sourcing from the current configuration along with its corresponding event. The destinations of all the transitions have to be recorded. Thus, for each transition in *NextTransList* we extract the destination configuration and add it to *FoundConfList* if not present in *CTSConfList*. Then, *NextTransList* is added to *CTSTransList* and the next iteration starts. The procedure stops when no element can be found in *FoundConfList*.

3.3.2 *Derivation of the sequence diagram's semantics*

A sequence diagram, as defined by UML 2.0, is composed of a set of lifelines, which correspond to objects interacting in a temporal order. The abstraction of the most general interaction unit is denoted by *InteractionFragment* (Object Management Group 2003). A combined fragment, denoted by *CombinedFragment* (Object Management Group 2003), is a specialisation of *InteractionFragment* that has an interaction operator with at least one operand. *Seq*, *Alt*, *Opt*, *Par* and *Loop* are examples of *CombinedFragment*. The interaction between lifelines is represented by message exchange. More specifically, it represents a communication (e.g. raising a signal, invoking an operation, creating or destroying an instance of an object).

Generally, the sequence diagram can be used to capture attributes such as latency and precedence. By extracting all possible execution paths of a given sequence diagram, we can construct its corresponding CTS. To proceed to the generation of the corresponding CTS, first, we have to encode the messages in a particular syntax. Each exchanged message *Msg* is written in the following format: *S_Msg_R*, where the sender of *Msg* is denoted by *S* and the receiver by *R*. In this case, a configuration is composed of the set of messages sent in parallel (separated by a comma). Messages enclosed in a *CombinedFragment* of type `Alt` represent multiple branching successor configurations. Messages enclosed in a *CombinedFragment* of type `Loop` represent a cycle of configurations in the CTS. Messages that are not enclosed in any *CombinedFragment*, but in *Seq*, represent, respectively, a singleton configuration. The transitions are derived from the sequencing events between messages in a sequence diagram.

In order to generate the CTS corresponding to the sequence diagram, an algorithm similar to Algorithm 1 is applied. Briefly, the modified algorithm consists of discarding the *EventList*, and changing both auxiliary functions *getNext* and *getDestination*. The latter are used to explore the dependent subsequent configurations in a sequence diagram according to the sequentially identified *CombinedFragment*.

3.4 *CTL-based property specification*

The verification process by means of model checking requires the precise specification of the properties in order to unfold the potential benefits of this technique. The NuSMV model checker uses the CTL (Dasgupta *et al.* 2002; Queiroz 2003) temporal logic for this purpose. This logic has interesting features and a great expressiveness. It can be used to express general safety and liveness as well as more advanced properties like conditional reachability, deadlock freedom, sequencing, precedence, etc. In the following paragraphs, we briefly introduce the CTL logic and its operators.

CTL is used to reason about the computation tree that is unfolded from a given state transition graph, where the various paths in a computation tree represent all possible computations of the corresponding model. CTL is classified as a branching-time logic since it has operators that allow the description of properties on the branching structure of the computation tree. The related properties are built using atomic propositions, propositional logic boolean connectives and temporal operators. The atomic propositions correspond to the variables in the model while each temporal operator consists of two components: a path quantifier and an adjacent temporal modality. The temporal operators are interpreted in the context of an implicit current state. Since in general it is possible to have many execution paths starting at the current state, the path quantifier indicates whether the modality defines a property that should hold for all the possible paths (universal path quantifier `A`) or only on some of them (existential path quantifier `E`). Figure 3 presents the syntax of CTL formulas, while Table 1 explains the underlaying meaning of the temporal modalities.

In order to relieve the load of knowledge of formal logics from the designer's side, we elaborated a pragmatics on top of the CTL logic, which enables the manual specification of properties using our V&V framework. This pragmatics is based on intuitive macros

$$
\begin{array}{lll}
\phi ::= & p & \text{(Atomic propositions)} \\
\quad | & !\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi & \text{(Boolean Connectives)} \\
\quad | & AG\ \phi \mid EG\ \phi \mid AF\ \phi \mid EF\ \phi & \text{(Temporal Operators)} \\
\quad | & AX\ \phi \mid EX\ \phi \mid A[\ \phi\ U\ \phi] \mid E[\ \phi\ U\ \phi] & \text{(Temporal Operators)}
\end{array}
$$

Figure 3. CTL syntax.

(e.g. `ALWAYS`, `MAYREACH`, etc.) that are systematically expanded to their corresponding CTL-based formulas.

3.5 *Model checking*

The back-end processing for CTS model checking requires its encoding in the NuSMV input language. It basically involves a grouping in three main syntactic declarative divisions[9] as follows. First we need a syntactic block wherein the state variables are defined along with their type and range. Secondly, we have to specify an initialisation block, wherein the state variables are given their corresponding initial values or a range of possible initial values. Third, we have to describe the dynamics of the transition system in a so called next clause block, wherein the logic governing the evolution of the state variables is specified. Consequently, the state variables are updated in every next step based on the logical valuation done at the current step.

Since the CTS dynamics is given in the form of pairwise configuration transition relations, any given CTS transition links a source configuration to a destination one. It is conceivable that we could encode each configuration as a distinct entity in the NuSMV model. However, one can note that the number of CTS configurations may be significantly higher than the number of states that are members of different configurations. Also, the properties to be verified ought to be expressed on states and not on configurations. Thus, in order to encode the CTS representation into the model checker language in a compact and meaningful way, we need to use as dynamic entities, the configuration states rather than the configurations themselves. This will be reflected accordingly in all three declarative blocks. The first one consists of enumerating the labels that are associated with each dynamic element that appears in at least one configuration along with its type and range. The second declarative block is compiled by using the initial configuration in order to specify the initial values. The third one is more laborious in nature and consists of analysing the CTS transitions in order to determine its state based evolution.

For every state `s` contained in any given configuration, which might be a destination for at least one or more transitions, the required conditions for the activation of `s` need to be specified for each incoming transition. Moreover, in the absence of these conditions, `s` should be deactivated. The aforementioned activation conditions can be expressed as boolean predicates in the form of conjunctions over the active status belonging to each state in the corresponding source configuration along with the test term for the transition trigger if it is the case. In the more general case where the source configuration elements

Table 1. CTL modalities.

G_p	Globally, p is satisfied for the entire subsequent path
F_p	Future (eventually), p is satisfied somewhere on the subsequent path
X_p	neXt, p is satisfied at the next state
$p\ U\ q$	Until, p has to hold until the point that q holds and q must eventually hold

might contain multiple value state variables, the activation condition predicates would also include value test terms for the corresponding multivalued variables. Consequently, for each state variable in the configurations of the CTS, we have to specify what we denote as transition candidates. Specifically, a candidate for each state s, represents the disjunctive combination over the activation conditions of all the destination configurations that have s as a member. After running the NuSMV model checker, the corresponding output is analysed by our V&V framework in order to identify which properties are satisfied in the model. In the case where a specified property fails, the model checker provides a trace-based counterexample that falsifies the property.

Figure 4 depicts a small fragment of the NuSMV code that is generated for a subset of the state machine diagram case study illustrated in Figure 8. In Sections 1 and 2, we show the application of our approach with respect to verifying behavioural diagrams. In the next section, we present the metrics computation related to our approach.

3.6 *Metrics computation*

In order to asses the quality of an object-oriented system, we need to measure various attributes such as complexity, understandability, maintainability, stability and others. The obtained values are compared against corresponding nominal ranges that represent empirically established intervals denoting a good design. Figure 5 depicts a snapshot of the metric computation component of our V&V tool.

In the following paragraphs, we detail some of the most relevant metrics along with some comments about their usefulness.

The set of six MOODs that Chidamber and Kemerer (1994) proposed, aims to assess the complexity by measuring different quality attributes such as maintainability, reusability, etc. However, only a subset, as depicted in Table 2, of three metrics can be applied on UML class diagrams:

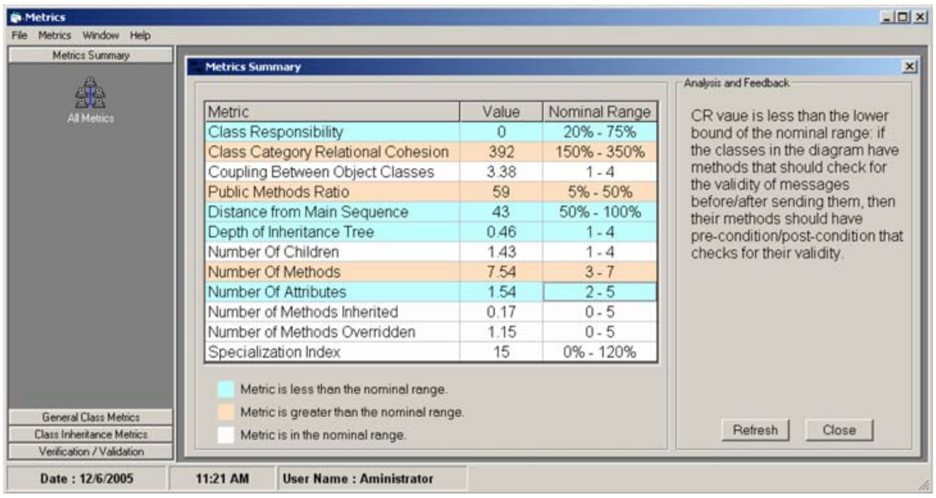
- Coupling between object (CBO) classes measures the level of coupling among the classes in the diagram. Excessive coupling hinders modularity and prohibits reuse and maintainability.
- Depth of inheritance tree (DIT) represents the length of inheritance tree from a class to its root class. Deep inheritance results in a relatively high number of methods for the specialised classes and increases the complexity.
- Weighted methods per class (WMC) is the summation of the complexity of all methods in the class. A high WMC value indicates increased complexity and less reusability. If unity is taken as the complexity of each method, then WMC is considered as the number of methods in the class.

The MOOD (Table 3) proposed by Abreu and Melo (1996) targets the structural mechanisms of the object-oriented paradigm such as encapsulation and inheritance:

- Method hiding factor (MHF) is a measure of the encapsulation in the class. It is the ratio of the sum of hidden methods (private and protected) to the total number of methods defined in each class. A high MHF value indicates potential accessibility and reusability issues whereas a zero value indicates encapsulation issues.
- Attribute hiding factor (AHF) is the average of the data hiding in the class diagram. It is the ratio of the sum of hidden attributes (private and protected) for all the classes to the sum of all defined attributes. A high AHF value indicates appropriate data hiding.

```
MODULE main
DEFINE
insuf_2:=1;
balOk_3:=2;
empty_1:=3;

Cand_modify0 := (evt=insuf_2) & chkbal;
Cand_all_modify := Cand_modify0;
Cand_chkbal0 := (evt=empty_1) & modify;
Cand_all_chkbal := Cand_chkbal0;
Cand_debit0 := (evt=balOk_3) & chkbal;
Cand_all_debit := Cand_debit0;
Cand_any := Cand_all_modify|Cand_all_chkbal|Cand_all_debit;

VAR
modify:boolean;
chkbal:boolean;
debit:boolean;
evt:1..3;

ASSIGN
init(modify):=1;
init(chkbal):=0;
init(debit):=0;

next(evt):=case
chkbal & !(modify | debit) : {insuf_2, balOk_3};
modify & !(chkbal | debit) : {empty_1};
1: {1, 2, 3};
esac;
next(modify):=case
Cand_all_modify: 1;
Cand_any: 0;
1: modify;
esac;
next(chkbal):=case
Cand_all_chkbal: 1;
Cand_any: 0;
1: chkbal;
esac;
next(debit):=case
Cand_all_debit: 1;
Cand_any: 0;
1: debit;
esac;
FAIRNESS Cand_any
SPEC EF modify
SPEC AG (modify -> EF !modify)
SPEC EF chkbal
SPEC AG (chkbal -> EF !chkbal)
SPEC EF debit
SPEC AG (debit -> EF !debit)
```

Figure 4. NuSMV code fragment.



Metric	Value	Nominal Range
Class Responsibility	0	20% - 75%
Class Category Relational Cohesion	392	150% - 350%
Coupling Between Object Classes	3.38	1 - 4
Public Methods Ratio	59	5% - 50%
Distance from Main Sequence	43	50% - 100%
Depth of Inheritance Tree	0.46	1 - 4
Number Of Children	1.43	1 - 4
Number Of Methods	7.54	3 - 7
Number Of Attributes	1.54	2 - 5
Number of Methods Inherited	0.17	0 - 5
Number of Methods Overridden	1.15	0 - 5
Specialization Index	15	0% - 120%

🟦 Metric is less than the nominal range.

🟧 Metric is greater than the nominal range.

⬜ Metric is in the nominal range.

CR vaue is less than the lower bound of the nominal range: if the classes in the diagram have methods that should check for the validity of messages before/after sending them, then their methods should have pre-condition/post-condition that checks for their validity.

Date : 12/6/2005 | 11:21 AM | User Name : Aministrator

Figure 5. Tool snapshot: metric summary.

- Method inheritance factor (MIF) and attribute inheritance factor (AIF) are two metrics that measure the class inheritance degree. MIF is the ratio of all inherited methods in the class diagram to total number of methods in the diagram. Likewise, AIF is the ratio of all inherited attributes in the class diagram to the total number attributes in the diagram. For both metrics, a zero value indicates no inheritance usage, which is undesired unless the class is a utility or a base class in the hierarchy.
- Polymorphism factor (POF) is a measure of method overriding. It is the ratio between the number of overridden methods in a class and the maximum number of methods that can be overridden in that class.
- Coupling factor (COF) measures the coupling level. It is the ratio between the actual couplings among all classes and the maximum number of possible couplings among all the classes. Coupling is present whenever a class is accessing the methods or members of another class. High values of COF indicate tight coupling and potential maintainability and reusability issues.

The set of metrics proposed by Lorenz and Kidd (1994) can be used to asses the static characteristics of software design such as size and inheritance. A subset of these metrics targeted the class size. The first size metric is the public instance methods and it represents the count of public methods in a class. The second metric is the number of instance methods and is the count of all methods in a class. The last metric, number of instance variables counts the total number of variables in a class. Another set of metrics addresses the class inheritance usage degree. The first is called the Number of Methods Overridden (NMO) metric and it gives a measure of the number of methods overridden by a subclass. The second is the NMI and it corresponds to the total number of methods inherited by a subclass. Additionally, the Number of Methods Added metric counts the methods added in

Table 2. Chidamber and Kemerer (1994) object-oriented metric suite.

CBO	The level of coupling among the classes.
DIT	The length of inheritance links from a class to its root.
WMC	The sum of the complexity of all methods in the class.

Table 3. Abreu *et al.* object-oriented metric suite.

MHF	Measure of the encapsulation in the class.
AHF	The average of the data hiding.
MIF	The class inheritance degree.
AIF	The class inheritance degree.
POF	The method overriding.
COF	The coupling level.

a subclass. Finally, the NMO and DIT (Chidamber and Kemerer 1994) metrics are used to calculate the specialisation index of a class, which gives an indication of the class inheritance utilisation.

The three metrics for UML package diagram proposed Martin (1994), namely the instability (I), abstractness (A) and distance from main sequence (DMS), measure the interdependencies among packages. Highly interdependent packages tend to be inflexible and thus hardly reusable and maintainable. The Instability metric measures if a package depends more on other packages than they depend on it. The Abstractness metric is a measure of the package's abstraction level. The DMS metric measures the balance between the abstraction and instability.

4. Case study

In the following paragraphs, we present a case study related to a UML 2.0-based design describing an ATM. We perform V&V of the design with respect to predefined properties and requirements. We present hereafter the behavioural view of the design captured in sequence and a state machine diagrams.

The ATM interacts with a potential customer (user) via a specific interface and communicates with the bank over an appropriate communication link. A user that requests a service from the ATM has to insert his ATM card and to enter his personal identification number (PIN). Both pieces of information are needed in order to be sent to the bank for validation. If the credentials of the customer are not valid, the card will be ejected. Otherwise, the customer will be able to perform one or more bank transactions (e.g. cash advance or bill payment). The card will be retained in the machine during the customer interaction until the customer wishes no further service.

4.1 *Sequence diagram*

The sequence diagram presented hereafter depicts one possible execution scenario of the interaction between three actors: the User, the ATM and the Bank. Though there are other possible execution scenarios, we will focus only on the one shown in Figure 6. The diagram comprises three main *CombinedFragment* (illustrated by labelled boxes): two are related to the authentication process and one to a banking operation. The label on each box denotes the corresponding *interactionOperator*, which specifies the semantics of the *CombinedFragment*. For instance, the label `par` indicates that the CombinedFragment represents a parallel execution between the behaviours of the operands. For the case of the label `alt`, it designates a choice of behaviour among the specified operands. The first *CombinedFragment* captures the validation of the card and a request of the PIN. The second one is an alternative choice that captures the validation of the PIN. The subsequent *CombinedFragment* depicts a possible interaction of using a cash advance service in the case where the credentials are valid.

In order to assess this diagram, we derive its corresponding semantic model, which is the CTS depicted in Figure 7. Since deadlock and reachability properties are generic specification, we only present some relevant properties such as service availability and safety. We describe each of them in two different notations: macro and CTL.

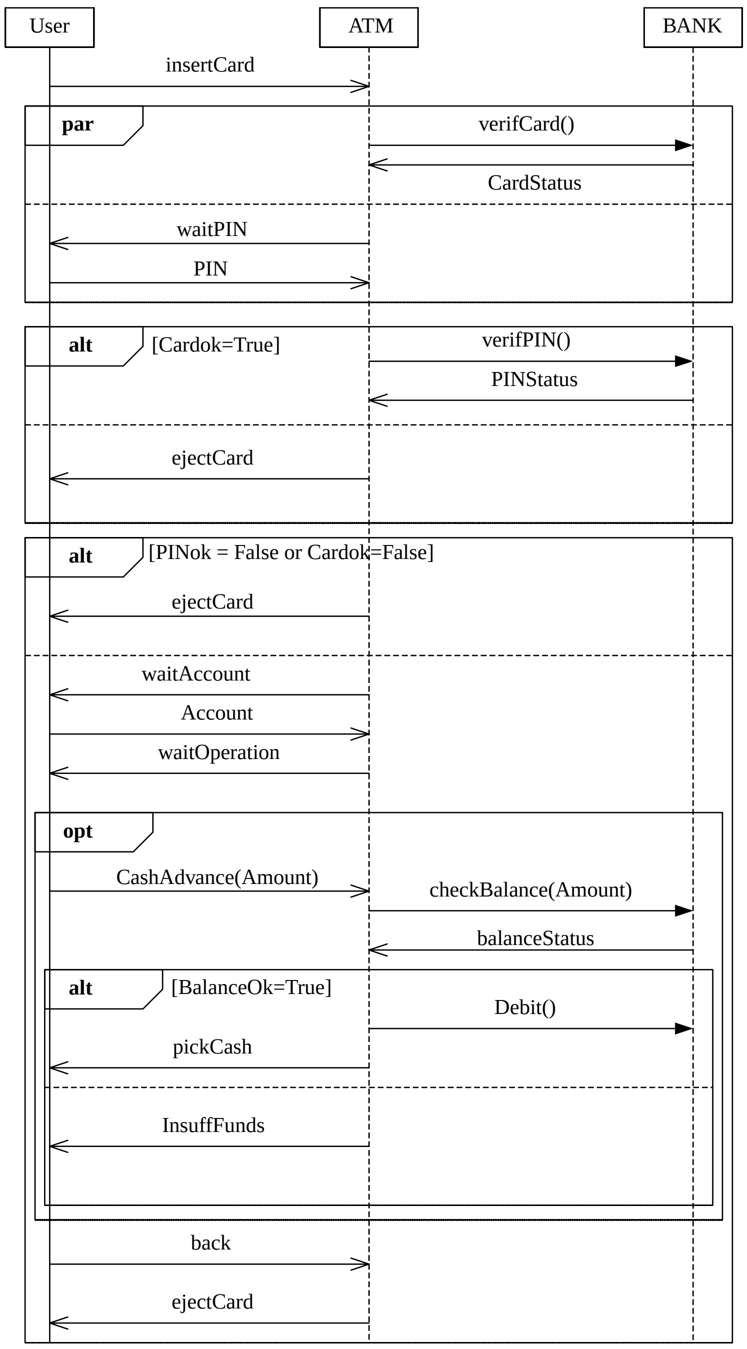


Figure 6. ATM sequence diagram example.

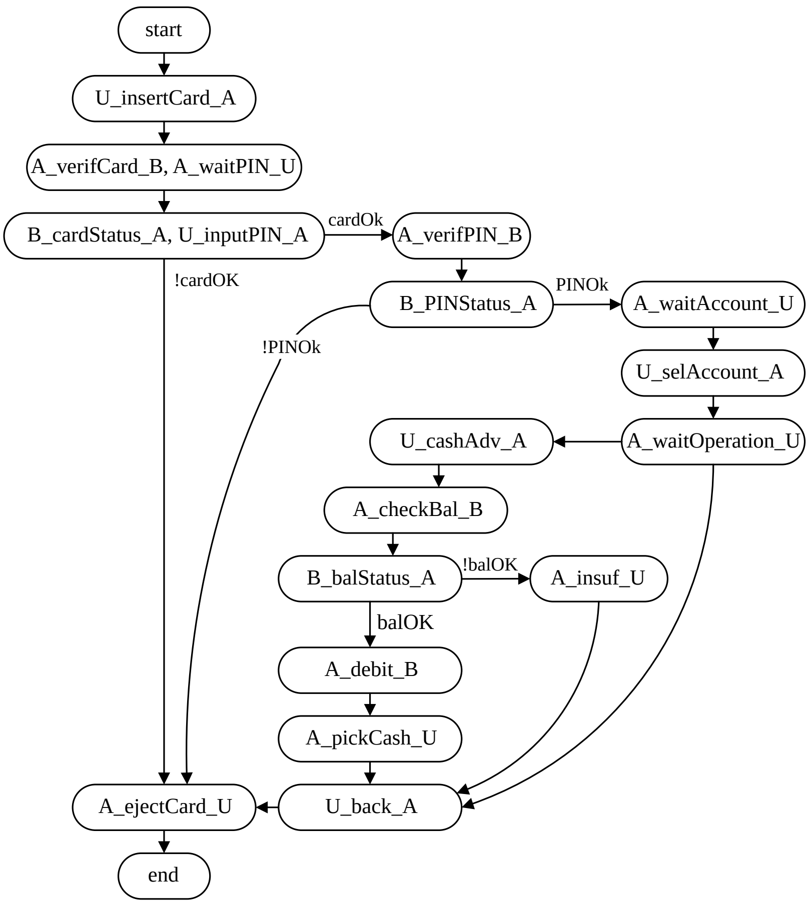


Figure 7. The CTS of the ATM sequence diagram example.

The first property (1) is a service availability specification. It asserts that always, if the user inserts his card, then a situation where the ATM advances cash should be reachable

$$\begin{aligned}&\textbf{Macro}: \texttt{ALWAYS}\ \mathit{U\_insertCard\_A} \longrightarrow \texttt{MAYREACH}\ \mathit{A\_pickCash\_U}\\ &\textbf{CTL}: \texttt{AG}(\mathit{U\_insertCard\_A} \longrightarrow \texttt{E}[\,!\,(\mathit{end})\,\texttt{U}\,\mathit{A\_pickCash\_U}]).\end{aligned}\tag{1}$$

The second one (2) is a safety property. It asserts that whenever the credentials are not valid, there should be no possibility for the user to request a banking operation:

$$\begin{aligned}&\textbf{Macro}: \texttt{ALWAYS}\ (!\mathit{CardOk}\ \mathit{or}\ !\mathit{PINOk}) \longrightarrow\ !\texttt{MAYREACH}\ (\mathit{A\_waitAccount\_U})\\ &\textbf{CTL}: \texttt{AG}((!\mathit{Cardok}\ \texttt{or}\ !\mathit{PINOk}) \longrightarrow\ !(\texttt{E}[\,!\,(\mathrm{end})\,\texttt{U}\,\mathrm{A\_waitAccount\_U}])).\end{aligned}\tag{2}$$

The third one (3) is related to an ergonomic specification stating that whenever the specified amount exceeds the available funds, it should be possible for the user to request a new cash advance operation (the user might want to correct the amount):

$$\begin{aligned}\textbf{Macro} &: \texttt{ALWAYS}\, A\text{-}insufFunds\text{-}U \rightarrow \texttt{POSSIB}\, U\text{-}CashAdvance\text{-}A \\ \textbf{CTL} &: \texttt{AG}(A\text{-}insufFunds\text{-}U \rightarrow \texttt{EX}(U\text{-}CashAdvance\text{-}A)).\end{aligned} \tag{3}$$

When subjecting the sequence diagram to V&V task, we found that only the first two properties are satisfied. However, the model checker was able to produce a counterexample for the third property. The interpreted result of the trace provided by the model checker consists in the following path in the CTS:

start; U_insertCard_A; (A_verifCard_B, A_waitPIN_U);
(B_cardStatus_A, U_inputPIN_A); A_verifPIN_B; B_PINStatus_A;
A_waitAccount_U; U_selAccount_A; A_waitOperation_U; U_cashAdv_A;
A_checkBal_B; B_balStatus_A; A_insuf_U; U_back_A;

As a notation convention, the identified path contains a series of semicolon-separated messages that are exchanged between the actors. Hence, when analysing the counterexample, one can note that there is no possibility of reaching the state U_cashAdv_A from the state A_insuf_U. Thus, we can conclude that the sequence diagram does not comply with this requirement.

4.2 *State machine diagram*

The state machine diagram, as defined by UML 2.0 standard, is an object-oriented variant of Harel state charts (Object Management Group 2003). It is used to model discrete event-driven behaviour of reactive systems. In this section, we explain how our V&V approach is performed on a given state machine diagram.

Figure 8 shows an example of UML 2.0 state machine diagram of the ATM system. The model is based on a hypothetical behaviour and is meant only as an example. We intendedly modelled some flaws in the design in order to outline the usefulness of our approach in discovering problems in the behavioural model. The diagram has several states that are going to be presented in accordance to the diagram containment hierarchy. The top container state ATM encloses four substates: IDLE, VERIFY, EJECT and OPERATION. The IDLE state, wherein the system waits for a potential ATM user, is the default initial substate of the top state. The VERIFY state represents the verification of the card validity and authorisation. The EJECT state depicts the phase of termination of the user transaction. The OPERATION state is a composite state that includes the states that capture several functions related to banking operations, which are the SELACCOUNT, PAYMENT and TRANSAC.

The SELACCOUNT state is where an account, belonging to the proprietary of the card, has to be selected. When the state SELACCOUNT is active, and the user selects an account, the next transition is enabled and the state PAYMENT is entered. The latter has two substates for cash advancing and bill payment, respectively. It represents a two-item menu, controlled by the event next. Finally, the TRANSAC state captures the transaction phase and includes three substates for checking the balance (CHKBAL), modifying the amount if necessary (MODIFY) and debiting the account (DEBIT). Each one of the states PAYMENT and TRANSAC contains a shallow history pseudostate. If a transition targeting a shallow history pseudostate is fired, the activated state is the most recent active substate in the composite state containing the history connector.

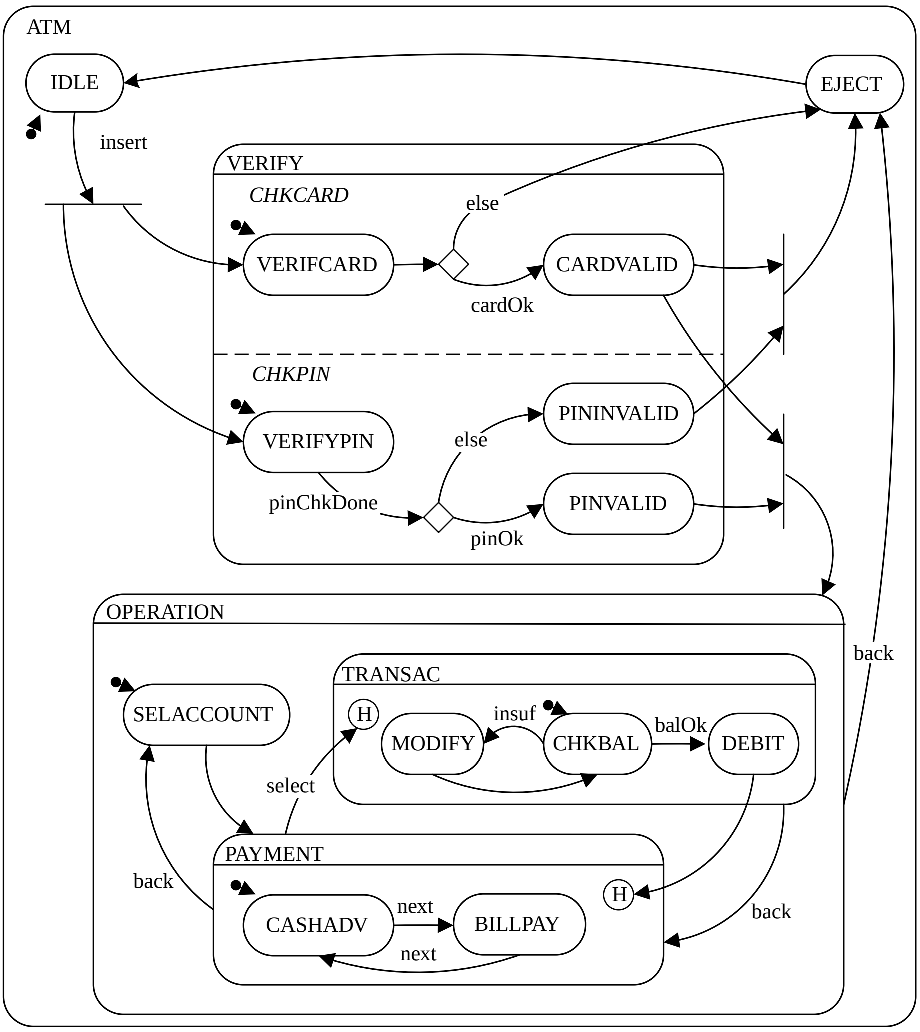


Figure 8. ATM state machine diagram example.

By applying our approach, we obtain as intermediate result the semantic model (CTS depicted in Figure 9) corresponding to the given state machine diagram. Each configuration is represented by a set (possibly singleton) of states and variable values of the state machine diagram. In the same time, deadlock and reachability properties are automatically generated for every state in the diagram. Thereafter, user-defined specification that are entered by the user in macro notation, are automatically expanded into CTL formulas and appended to the input of the model checker. Once the model checking procedure terminates, the assessment results 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 is due to the fact that, according to the semantic of UML state machine diagrams, the transitions with 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 stable configuration that is containing the corresponding source state. This is precisely the case with the transition from SELACCOUNT to PAYMENT. Thus, there is no configuration that allows the OPERATION state to be exited. This is clearly observable when looking at the corresponding CTS where we can notice that once a configuration containing the OPERATION state is reached, there is no transition to a configuration that does not contain it. We present hereafter, some relevant user-defined properties described in both macro and CTL notations and their corresponding model checking results.

The first property (1) asserts that it is always the case that if the VERIFY state is reached then from that point, the OPERATION state should be also reachable:

$$\begin{aligned}&\textbf{Macro}:\ \texttt{ALWAYS}\ \mathit{VERIFY}\longrightarrow\texttt{MAYREACH}\ \mathit{OPERATION}\\&\textbf{CTL}:\ \texttt{AG}(\mathit{VERIFY}\longrightarrow(\texttt{E}[\,!\,(\mathit{IDLE})\ \texttt{U}\ \mathit{OPERATION}]))\end{aligned}\tag{1}$$

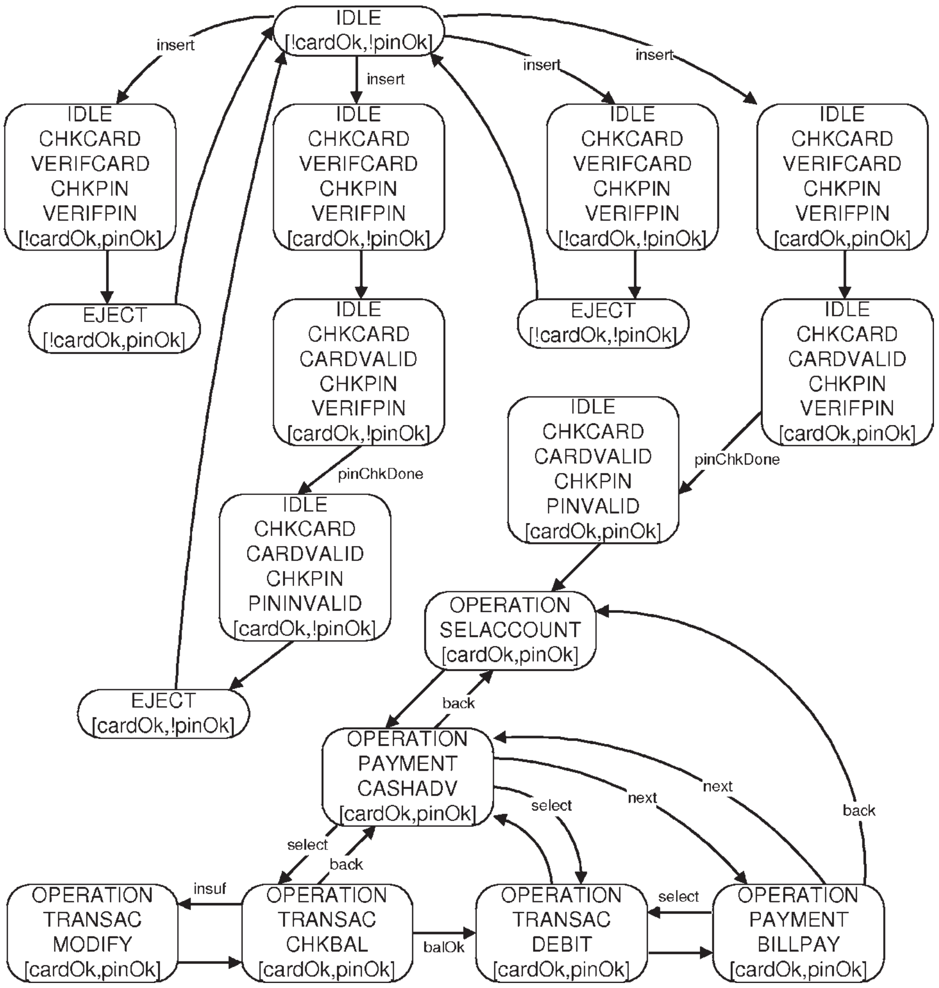

IDLE [!cardOk,!pinOk]
insert
insert
insert
insert
IDLE CHKCARD VERIFCARD CHKPIN VERIFPIN [!cardOk,pinOk]
IDLE CHKCARD VERIFCARD CHKPIN VERIFPIN [cardOk,!pinOk]
IDLE CHKCARD VERIFCARD CHKPIN VERIFPIN [!cardOk,!pinOk]
IDLE CHKCARD VERIFCARD CHKPIN VERIFPIN [cardOk,pinOk]
EJECT [!cardOk,pinOk]
IDLE CHKCARD CARDVALID CHKPIN VERIFPIN [cardOk,!pinOk]
EJECT [!cardOk,!pinOk]
IDLE CHKCARD CARDVALID CHKPIN VERIFPIN [cardOk,pinOk]
IDLE CHKCARD CARDVALID CHKPIN PINVALID [cardOk,pinOk]
pinChkDone
pinChkDone
IDLE CHKCARD CARDVALID CHKPIN PININVALID [cardOk,!pinOk]
OPERATION SELACCOUNT [cardOk,pinOk]
EJECT [cardOk,!pinOk]
back
OPERATION PAYMENT CASHADV [cardOk,pinOk]
next
next
select
back
select
back
OPERATION TRANSAC MODIFY [cardOk,pinOk]
insuf
OPERATION TRANSAC CHKBAL [cardOk,pinOk]
balOk
OPERATION TRANSAC DEBIT [cardOk,pinOk]
select
OPERATION PAYMENT BILLPAY [cardOk,pinOk]

Figure 9. CTS of the ATM state machine example.

The next property (2) asserts that whenever the state OPERATION is reached, it should be unavoidable to reach the state EJECT at a later point:

$$\begin{aligned}&\textbf{Macro}: \texttt{ALWAYS}\ \mathit{OPERATION} \rightarrow \texttt{INEVIT}\ \mathit{EJECT}\\&\textbf{CTL}: \texttt{AG}(\mathit{OPERATION} \rightarrow (\texttt{A[!}(\mathit{IDLE})\ \texttt{U}\ \mathit{EJECT}]))\end{aligned} \tag{2}$$

The last one (3) states that the CHKBAL state must precede the state DEBIT:

$$\begin{aligned}&\textbf{Macro}: \mathit{CHKBAL}\ \texttt{PRECEDE}\ \mathit{DEBIT}\\&\textbf{CTL}: \ \texttt{!E[!}(\mathit{CHKBAL})\ \texttt{U}\ \mathit{DEBIT}]\end{aligned} \tag{3}$$

The property (1) turned out to be satisfied when running the model checker. However, the last two properties (2) and (3) failed. The failure of the property (2) was not unexpected since, from the automatic specifications, we noticed that state OPERATION is never left once entered (it exhibits deadlock) and it does not have the state EJECT as a substate.

The failure of the last property (3) was demonstrated by a counterexample provided by the model checker. Though the model checker can provide a counterexample for any of the failed properties, we present this last one as it captures a critical unintended behaviour. The result was parsed by our tool into a trace, as follows:

```
IDLE [!cardOk,!pinOk];
(VERIFY,CHKCARD,VERIFCARD,CHKPIN,VERIFPIN [cardOk,pinOk]);
(VERIFY,CHKCARD,CARDVALID,CHKPIN,VERIFPIN [cardOk,pinOk]);
(VERIFY,CHKCARD,CARDVALID,CHKPIN,PINVALID [cardOk,pinOk]);
(OPERATION,SELACCOUNT [cardOk,pinOk]);
(OPERATION,PAYMENT,CASHADV [cardOk,pinOk]);
(OPERATION,TRANSAC,DEBIT [cardOk,pinOk]);
```

The foregoing counterexample is represented by a series of configurations separated by semicolons. Additionally, a comma is used to separate two or more states that are present simultaneously in a given configuration and the variable values are enclosed in square brackets. The failure of the last property is due to the presence of a transition from the state PAYMENT to the shallow history connector of the state TRANSAC. This allows for the immediate activation of the state DEBIT when reentering the TRANSAC state by its history connector.

The counterexample can help the designer to infer the necessary changes that will fix the identified problems. In order to do that, the first modification consists of adding a trigger event such as select to the transition from the state SELACCOUNT to the state PAYMENT. This will fix the deadlock problem and the second user-defined property (2). Another modification is required to correct the problem related to the last unsatisfied specification property (3). It consists of removing the history connector of the state TRANSAC and changing the incoming transition from this target directly to the state TRANSAC. After re-executing the V&V task for the fixed diagram, all the specifications that were defined automatically or by the user were satisfied.

4.3 *Programme analysis integration*

This section discusses the use of programme analysis techniques (data and control flow), on the semantic model, namely the CTS. Our goal is to identify and extract those parts

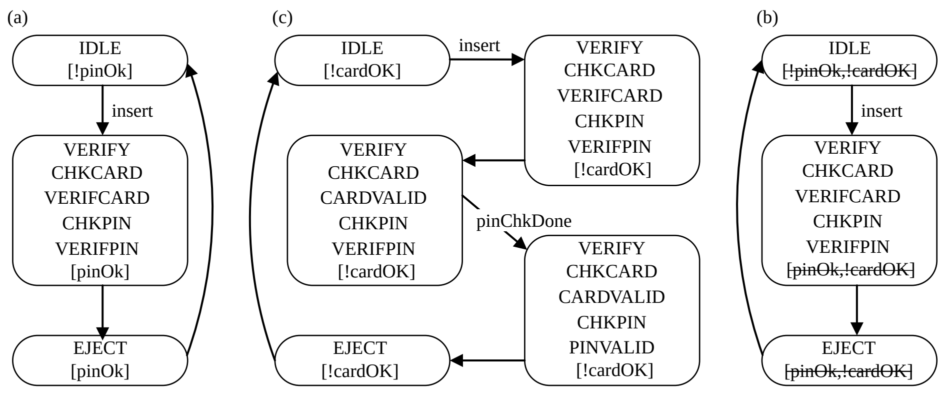

(a)
IDLE [!pinOk]
insert
VERIFY CHKCARD VERIFCARD CHKPIN VERIFPIN [pinOk]
EJECT [pinOk]
(c)
IDLE [!cardOK]
insert
VERIFY CHKCARD VERIFCARD CHKPIN VERIFPIN [!cardOK]
VERIFY CHKCARD CARDVALID CHKPIN VERIFPIN [!cardOK]
pinChkDone
VERIFY CHKCARD CARDVALID CHKPIN PINVALID [!cardOK]
EJECT [!cardOK]
(b)
IDLE ~~[!pinOk,!cardOK]~~
insert
VERIFY CHKCARD VERIFCARD CHKPIN VERIFPIN ~~[pinOk,!cardOK]~~
EJECT ~~[pinOk,!cardOK]~~

Figure 10. Data flow subgraphs. (a) Data flow subgraph with !cardOK invariant. (b) Data flow subgraph with !pinOK and !cardOK invariants. (c) Data flow subgraph with !pinOK invariant.

of the CTS that exhibit properties that can be used to simplify the transition system. This in turn has the potential to leverage the effectiveness of the model checking procedure. The aspects that we are interested in are the data and control flow. The former is applied by basically searching for the presence of invariants whereas the latter is used in order to detect the control flow dependency among various parts of the transition system.

In order to illustrate data and control flow analysis on the CTS, we will use the CTS corresponding to the state machine of Figure 9. In this example, in every configuration there are various values for the variables `cardOk` and `pinOk`. Whenever we have an exclamation mark preceding a variable in a particular configuration, it means that the variable has a false value in that configuration. There are several parts of the graph where some invariants hold. Figure 10 presents three subgraphs, each having invariants that can be abstracted. Thus, the subgraph of Figure 10(a) has the invariant `!cardOk`. Similarly, the subgraph of Figure 10(c) contains the invariant `!pinOk`. Another subgraph presented

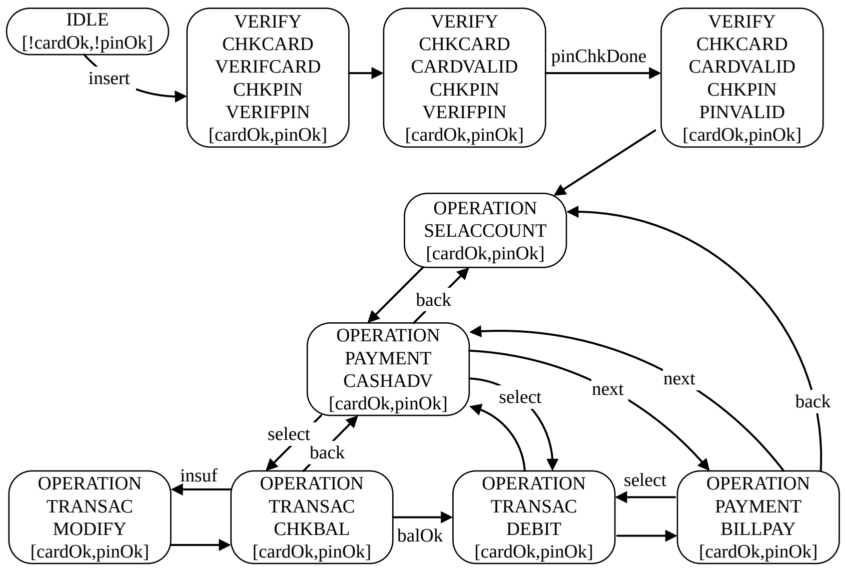

IDLE [!cardOk,!pinOk]
insert
VERIFY CHKCARD VERIFCARD CHKPIN VERIFPIN [cardOk,pinOk]
VERIFY CHKCARD CARDVALID CHKPIN VERIFPIN [cardOk,pinOk]
pinChkDone
VERIFY CHKCARD CARDVALID CHKPIN PINVALID [cardOk,pinOk]
OPERATION SELACCOUNT [cardOk,pinOk]
back
OPERATION PAYMENT CASHADV [cardOk,pinOk]
select
next
next
back
select
back
OPERATION TRANSAC MODIFY [cardOk,pinOk]
insuf
OPERATION TRANSAC CHKBAL [cardOk,pinOk]
balOk
OPERATION TRANSAC DEBIT [cardOk,pinOk]
select
OPERATION PAYMENT BILLPAY [cardOk,pinOk]

Figure 11. Control flow subgraph.

Table 4. Statistics related to memory consumption during model checking.

Graph	Memory footprint (BDD nodes)
Figure 9	70,000–80,000
Figure 10(a)	≈4000
Figure 10(b)	≈4000
Figure 10(c)	≈8000
Figure 11	28,000–33,000

in Figure 10(b) with the invariants `!cardOk` and `!pinOk`. Additionally, Figure 11 depicts a subgraph that is independent from the control flow perspective. To that effect, once the control is transferred to this subgraph, it never leaves it.

The subgraphs identified in the foregoing paragraph represent the basis that enable us to slice (decompose) the initial model into several independent parts that can be analysed separately. Obviously, the subgraphs have reduced complexity when compared to the original model. Accordingly, for each of them, the corresponding transition system that is going to be subjected to model checking requires fewer resources in terms of memory space and computation time. Though it might be possible to specify some properties that could span across more than one subgraph of the original CTS, the slicing can be safely done under the following assumptions:

- The properties to be verified fall into liveness or safety category;
- No property specification should involve sequences or execution traces that require the presence of the initial state more than once.

It must be noted that the second constraint does not represent a major hindrance for the verification potential. In this respect, the presence of invariants is assuring that revisiting the initial state or entering it for the first time is equivalent with respect to the dynamics of the transition system. In order to emphasise the benefits of the slicing procedure, we give some edifying statistics. While for the initial CTS graph, the model checker allocated between 70 and 80 thousand binary decision diagram (BDD) nodes (depending on the variable ordering), for the sliced subgraphs the allocated BDD nodes were significantly reduced as shown in Table 4.

It must be mentioned at this point that even though some of the configuration subgraphs are very simple, it is nevertheless required for the model checking procedure that one specifies all the elements of the original model. However, this must be done such that the underlying dynamics is captured by the particular configuration subgraph in question. Moreover, due to the fact that the dynamics may be severely restricted in some cases, one has to take this fact into account when interpreting the model checking results. Thus, even though it might be the case that a liveness property fails for a transition system corresponding to a particular subgraph, the property should not be declared as failed for the original model as long as there is at least one subgraph whose transition system satisfies the property in question. Conversely, whenever a safety property fails for a particular subgraph, then it is declared as failed for the original model as well. Notwithstanding, this task can be automated and virtually transparent to the front-end of the verification framework.

5. Conclusion and future work

In this paper, we reported a synergistic approach for the V&V of object-oriented design models in SwE and SE. The ingredients are three well-established techniques: model checking, static analysis and software metrics. Harmoniously combined, these techniques make the V&V task more comprehensive and cost-effective. In fact, we can assess a design model from both structural and behavioural perspectives using metrics and model checking. We illustrated the synergy by applying static analysis (control and data flow) on the semantic model, before performing the model checking procedure. Currently, we are in the process of investigating the use of metrics for the semantic model complexity measurement. As future work, we intend to explore other aspects of the synergy and to undertake more elaborated case studies. Furthermore, we plan to conduct other case studies for assessing design models involving SysML 1.0 diagrams.

Acknowledgements

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). The research is supported by the CapDEM (Collaborative Capability Definition, Engineering and Management) project.

Notes

1. Email: a_soeanu@encs.concordia.ca
2. Email: l_alawne@encs.concordia.ca
3. Email: debbabi@encs.concordia.ca
4. Email: Fawzi.Hassaine@drdc-rddc.gc.ca
5. http://www.cs.bham.ac.uk/~dxp/prism/
6. http://www-omega.imag.fr/
7. http://www.artisansw.com
8. http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html
9. One might use more convenient NuSMV constructs or various levels of hierarchy where a main module is referring several other sub-modules due to the modular aspect that some particular transition systems might exhibit. However, this has no semantic impact with respect to the considered declarative divisions.

Notes on contributors

Yosr Jarraya is a PhD candidate in Electrical and Computer Engineering at Concordia University, Montreal, Quebec, Canada. She is also a research assistant in the Computer Security Laboratory (CSL) at Concordia Institute for Information Systems Engineering. She holds MSc degree in Telecommunications from École Supérieure des Communications de Tunis, Tunisia. The main topic of her current research activities is the verification and validation of systems and software design models expressed using UML and SysML including timed and probabilistic behaviour verification.

Andrei Soeanu graduated from Concordia University where he obtained a Masters Degree from the faculty of Electrical and Computer Engineering in the area of Software Intensive Systems Engineering. He participated in research activities sponsored by the Defense Research and Development Canada (DRDC) on various topics including System Engineering, Modelling Languages as well as Verification and Validation of design models.

Luay Alawneh is a PhD student in Software Engineering at Concordia University in Montreal, Canada. His PhD research topic is specialised in Metamodeling and Dynamic Analysis of various runtime execution traces. He holds a Master's degree in Verification and Validation of Software and Systems engineering design models from Concordia University. Luay Alawneh has more than 7 years of professional experience as a software developer. Currently, he is working as an ERP and CRM Solutions Developer in TekSystems Inc – Montreal for Rockwell Automation, a global automation leader.

Dr Mourad Debbabi is a Professor and the Director of the Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds the Concordia Research Chair Tier I in Information Systems Security. He is the founder and one of the leaders of the Computer Security Laboratory (CSL) at Concordia University. He is the Specification Lead of four Standard JAIN (Java Intelligent Networks) Java Specification Requests (JSRs) dedicated to the elaboration of standard specifications for presence and instant messaging. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France. Dr Debbabi holds PhD and MSc degrees in computer science from Paris-XI Orsay, University, France. He published more than 150 research papers in journals and conferences on computer security, formal semantics, Java security and acceleration, cryptographic protocols, malicious code detection, programming languages, type theory and specification and verification of safety–critical systems. He supervised to completion more than50 graduate students at MSc and PhD levels. He can be reached at debbabi@ciise.concordia.ca. His webpage is at http://www.ciise.concordia.ca/~debbabi

Dr Fawzi Hassaïne holds a PhD and a Master degree in Computer Science from Paris VI University. He spent more than ten years in various industrial R&D centres working on distributed and parallel applications, real-time and embedded systems, data acquisition and control systems. Dr Hassaïne is presently a Defence Scientist within the Defence Research and Development Canada, the R&D establishment of Canada's Department of National Defence. His present research interests include: Synthetic Environments, Distributed Simulation, Computer Generated Forces, Command and Control and the application of formal and non-formal methods to the Verification and Validation of Systems Engineering design models.

References

Abreu, F. and Carapua, R., 1994. *Object-oriented software engineering: measuring and controlling the development process*. October. McLean, VA: American Society for Quality.

Abreu, F. and Melo, W., 1996. *Evaluating the impact of object-oriented design on software quality*. Washington, DC: IEEE Computer Society, 90–99.

Alawneh, L., *et al.*, 2006. *A unified approach for verification and validation of systems and software engineering models*. Washington, DC: IEEE Computer Society, 409–418.

Ambler, S.W., 2004. *The object primer 3rd edition: agile model-driven development with UML 2.0*. New York, NY: Cambridge University Press.

Averant, I., 2001. Static functional verification with solidify, a new low-risk methodology for faster debug of ASICs and programmable parts, Technical report, Averant, Inc.

Ben-David, S., *et al.*, 2000. *Scalable distributed on-the-fly symbolic model checking*. Berlin/Heidelberg: Springer, 390–404.

Boehm, B.W. and Basili, V.R., 2001. Software defect reduction top 10 list. *IEEE computer*, 34 (1), 135–137.

Bozga, M., *et al.*, 1999. *IF: an intermediate representation and validation environment for timed asynchronous systems*. Berlin/Heidelberg: Springer, 307–327.

Briand, L.C., Devanbu, P.T. and Melo, W.L., 1997. *An investigation into coupling measures for C++*. New York, NY: ACM, 412–421.

Burch, J., *et al.*, 1990. *Symbolic model checking: 1020 states and beyond*. Washington, DC: IEEE Computer Society Press, 1–33.

Cengarle, M.V. and Knapp, A., 2004. *UML 2.0 interactions: semantics and refinement*. München: Technische Universität München, 85–99.

Chidamber, S.R. and Kemerer, C.F., 1994. A metrics suite for object-oriented design. *IEEE transaction on software engineering*, 20 (6), 476–493.

Cimatti, A., *et al.*, 1999. *NUSMV: a new symbolic model verifier*. London: Springer, 495–499.

Dasgupta, P., Chakrabarti, A. and Chakrabarti, P.P., 2002. *Open computation tree logic for formal verification of modules*. Washington, DC: IEEE Computer Society, 735.

Fecher, H., Kyas, M. and Schönborn, J., 2005. Semantic issues in UML 2.0 state machines, Technical report 0507, Christian-Albrechts-Universität zu Kiel.

Genero, M., Piattini, M. and Calero, C., 2000. Early measures for UML class diagrams. *L'Objet*, 6 (4), 489–515.

Gronback, R., 2004. *Model validation: applying audits and metrics to UML models*, Borland Developer Conference, Borland Software Corporation.

Grosu, R. and Smolka, S.A., 2005. *Safety-liveness semantics for UML 2.0 sequence diagrams*. June. Washington, DC: IEEE Computer Society, 6–14.

Holzmann, G., 1997. The model checker SPIN. *IEEE transaction on software engineering*, 23 (5), 279–295. Special issue on formal methods in software practice.

Hsin-Hung, L., 2003. *A research of model checking UML statechart diagrams*. Master's thesis. Japan Advanced Institute of Science and Technology.

IEEE, 1990. IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology, Technical report, IEEE.

INCOSE, 2004. The international council on systems engineering (INCOSE). Available from: http://www.incose.org/practice/whatissystemseng.aspx [Accessed October 2008].

Jarraya, Y., *et al.*, 2007. *Automatic verification and performance analysis of time-constrained SysML activity diagrams*. Los Alamitos, CA: IEEE Computer Society, 515–522.

Knapp, A., Merz, S. and Rauh, C., 2002. *Model checking – timed UML state machines and collaborations*. Berlin/Heidelberg: Springer, 395–414.

Latella, D., Majzik, I. and Massink, M., 1999a. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model checker. *Formal aspects of computing*, 11 (6), 637–664.

Latella, D., Majzik, I. and Massink, M., 1999b. *Towards a formal operational semantics of UML statechart diagrams*. Deventer: Kluwer, B.V., 465.

Li, W. and Henry, S., 1993. *Maintenance metrics for the object-oriented paradigm*. Washington, DC: IEEE Computer Society Press, 52–60.

Li, X., Liu, Z. and He, J., 2004. *A formal semantics of UML sequence diagrams*. April. Melbourne: IEEE Computer Society, 13–16.

Lorenz, M. and Kidd, J., 1994. *Object-oriented software metrics: a practical guide*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Martin, R.C., 1994. OO design quality metrics. Available from: http://www.objectmentor.com/resources/articles/oodmetrc.pdf.

McMillan, K.L., 1992. The SMV system, Technical report CMU-CS-92-131, Carnegie Mellon University.

Mikk, E., *et al.*, 1998. *Implementing statecharts in PROMELA/SPIN*. Washington, DC: IEEE Computer Society, 90.

NASA, 1995. *Software quality metrics for object-oriented system environments*. Technical report SATC-TR-95-1001. Greenbelt, MA: National Aeronautics and Space Administration, Goddard Space Flight Center.

Ober, I., Graf, S. and Lesens, D., 2006. *Modeling and validation of a software architecture for the Ariane-5 launcher*. Vol. 4037. Lecture Notes in Computer Science. Berlin Heidelberg: Springer, 48–62.

Ober, I., Graf, S. and Ober, I., 2003. *Validating timed UML models by simulation and verification*. Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems (SVERTS'03), a satellite event of UML 2003, San Francisco, CA, October 2003.

OMG, 2003. *UML 2.0 superstructure specification*. Technical Report, Object Management Group. http://www.omg.org/docs/ptc/03-08-02.pdf.

OMG, 2006. *Systems modeling language (OMG SysML) specification*. Technical Report, Object Management Group, Final Adopted Specification of Systems Modeling Language (SysML).

Peled, D., 1994. *Combining partial order reductions with on-the-fly model-checking*. London: Springer-Verlag, 377–390.

Queiroz, R.D., ed., 2003. *Logic for concurrency and synchronisation*. Norwell, MA: Kluwer Academic Publishers.

Störrle, H., 2003. *Semantics of Interactions in UML 2.0*. IEEE Computer Society, 129–136.

Tugwell, G., *et al*., 1999. *Metrics for full systems engineering lifecycle activities (MeFuSELA)*. Brighton, UK.

Viehl, A., *et al*., 2006. *Formal performance analysis and simulation of UML/SysML models for ESL design*. Munich: European Design and Automation Association, 242–247.

Zhan, X. and Miao, H., 2004. *An approach to formalizing the semantics of UML statecharts*. Berlin/Heidelberg: Springer, 753–765.