Model Checking X-Machines: Towards Integrated Formal Development of Safety Critical Systems

George Eleftherakis and Petros Kefalas
Computer Science Department
City Liberal Studies
Affiliated College of the University of Sheffield
13 Tsimiski Str., 54624 Thessaloniki, Greece

{eleftherakis, kefalas}@city.academic.gr

Abstract. *Computer systems used in most safety critical applications, such as in medical, nuclear, space domains etc. are often responsible for major damages and injuries due to unpredicted malfunction. Misleading user requirements, errors in the specification and in the implementation are the usual reasons responsible for the creation of such non-safe systems. This paper advocates the use of X-Machines in the development of safety and business critical systems and proposes its extension by a model checking technique, thus leading towards an integrated formal method. The resulting development methodology, adopting the X-machines as a specification formalism and the model checking as a verification technique, gives the ability to the software engineer to intuitively as well as formally specify a system, then automatically check if this model has all the desired properties, and finally test if the implementation is equivalent to the specification by applying a complete set of test cases. Therefore, the use of this method in the development of systems in safety critical domains can assure that the final product is valid with respect to the user requirements by revealing errors during the development life cycle and subsequently adding to the confidence of their use. The proposed methodology in this paper is accompanied by an example, which demonstrates the use of X-Machines in specification, testing and validation.*

1 Introduction

In *safety critical* domains, such as medical, space, nuclear domains etc., the traditional development life cycle failed several times to produce a reliable and correct system. There are a number of reasons for this. Among others, the most prominent are:

- Misleading user requirements may lead to a potential "correct" specification and implementation, which however do not meet the actual requirements.
- Errors in the specification of the system produce a model different from the one needed.
- Errors in the implementation lead to a different product from the one specified.

Errors in such systems not only cause an increase in the cost of a project, but are also responsible for accidents, even provoking deaths in some rare cases [1]. The application of *formal methods* in safety and business critical systems could add to the confidence in using the system by revealing errors during the development process, in both the system's specification and its implementation.

Formal specification is the procedure of describing a system and its desired properties precisely, by using a language with rigorously defined syntax and semantics. Logic and sets often form the basic theory behind such mathematical languages, in order to avoid ambiguity and allow automated verification techniques on the specification [2]. Specifying a system means to create the appropriate model

that describes it. The most usual general definition of a *model* is as a triple <W, R, π> where:

- W is a non-empty set of states,
- R is a binary relation on W, i.e. $R \subseteq W \times W$, that shows which states are possibly related to other states
- π is a truth assignment function that shows which propositions are true in each state, so $\pi: W \times P \rightarrow \{true, false\}$, where P is the set of all the properties in this model.

Considering the above definition, *model checking* can be defined as the process of proving if a given property is valid in any, some or all states of the system. This can be achieved by searching the state space of the model (W, R), checking in which states the property is valid by applying the π truth assignment function. In more detail, there is a need to distinguish one or a combination of the following: a property p_i can be valid for all paths, for some paths, for all states of a path, or for some states of a path. *Temporal logic formulae* [3] through temporal operators give the ability to express combinations of all the previous cases. Using a temporal logic, it is possible to write a specification as a formula and check it against this model, in order to prove that this specification is valid.

On the other hand, assuming a correct specification, *testing* has an essential role to play in system development. Although testing may identify errors in the implementation without being able to prove its correctness, it is considered as a major activity in the project's development life cycle. Since testing can be regarded as the process of determining whether a system matches its specification, the two activities, i.e. specification and testing are directly linked. Testing based only in the implementation is imperfect [4]. In contrast, testing based on formal specifications provides a structured and rigorous approach to the development of the test set. However, formal specifications used as the basis for proofs of correctness should be able to provide a well-defined testing strategy.

Assuming that the user requirements reflect the desired system, there is a need for an integrated formal technique that will allow us to:

- formally specify the system,
- check that the specification has the desired properties,
- check the implementation against the specification.

The specification language should be:

- formally based, thus making it suitable for mathematical analysis,
- rigorous,
- expressive,
- unambiguous,
- capable of capturing both static and dynamic system information,
- based on a fully general and formalised computational model that could form the basis of a universal approach to the design of systems,
- user friendly,
- supported by appropriate tools.

The technique which verifies that the specification model has the desired properties should be able to:

- prove whether a desired property is satisfied by the model of the system,
- provide valuable debugging information to the software engineer in order to modify the model aiming towards a correct system,
- be completely automated, and
- adopt techniques that tackle the state explosion problem.

The technique that tests the implementation should:

- be able to prove that the implementation matches the formal specification, and
- derive all the appropriate test cases from the specification.

In this paper, we propose the use of a formal method, namely X-Machines, which can accommodate all three above-mentioned activities. A X-machine is a general computational machine that is like a Finite State Machine (FSM) but with a significant difference; transitions are not labelled with simple inputs but with functions that operate on inputs and a memory, allowing the machine to be more expressive and flexible than the FSM. X-Machines are able to model both the control and the data part of a system. This integrated method uses X-machines as a specification language, a testing strategy to check the implementation against the X-machine specification and a model checking technique to prove the validity of the specification. We argue that, by applying X-Machines, it is possible to assure that a system is correct. In safety critical systems such a formal methodology can be used to prove that several "safety" properties hold in the final product.

The methodology suggested to be followed using X-Machines as a formal method is depicted in Figure 1. The grey shaded areas are tasks in which X-Machines are used as the primary methodology. Step 1 results in the formal specification described as an X-Machine. Step 2 verifies that certain safety properties hold in the model and feedback is used to refine the system specification through step 3. Step 4 is the actual implementation task. Finally, steps 5 and 6 facilitate the testing of the implementation for correctness with respect to the specification and the refinement of the implementation, through the use of a complete test set derived from the X-Machine model.

In the next sections, the integrated methodology of X-Machines will be introduced. First, in section 2, X-Machines will be formally defined, presenting the use of X-Machines as a specification formalism and as a methodology for testing a system respectively. The main contribution of this work is analytically discussed in section 3, where a methodology for model checking X-Machine specifications is presented. A concrete example of a small safety critical system is given in order to accompany the theory and demonstrate the applicability of the approach.

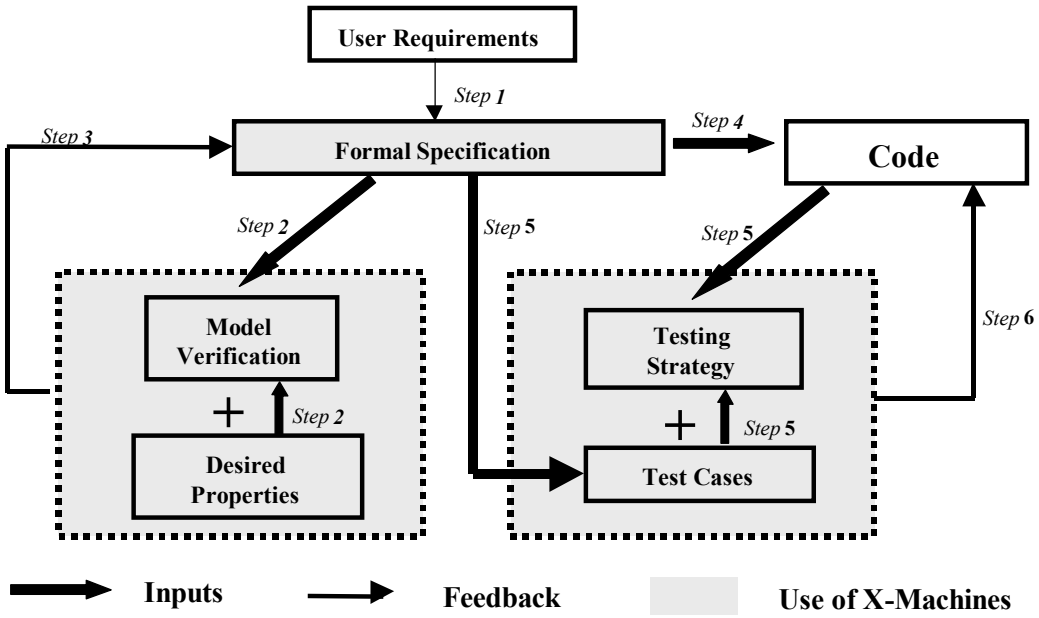

Fig. 1. The methodology of building safety critical systems with X-Machines

2 X-Machines

2.1 Basic definitions

X-machine is a specification formalism introduced by Eilenberg [5], which is capable of modelling both the data and the control by integrating specification methods, which describe each of these aspects in the most appropriate way. X-machines employ a diagrammatic approach of modelling the control by extending the expressive power of the FSM. Transitions between states are no longer performed through simple input symbols but through the application of functions. These functions are written in a formal notation and model the processing of the data. Data is held in memory, which is attached to the X-machine. Functions receive input symbols and memory values, and produce output while modifying the memory values (Figure 2).

For fourteen years X-machines did not receive special attention until in 1988 Holcombe proposed this model as a basis for a possible specification language [6]. In 1992, stream X-machines were defined as X-machines with input and output sets of streams of symbols. In short, the idea was that the machine has infinite internal memory and depending on the current state of control and the current state of the memory, an input symbol from the input stream determines the next state, the new memory state and the output symbol, which will be part of the output stream. The formal definition of a deterministic stream X-machine [7] is the following:

A stream X-machine is an 8-tuple $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$

where:

- Σ, Γ is the input and output finite alphabet respectively,
- Q is the finite set of states,
- M is the (possibly) infinite set called memory,
- Φ is the type of the machine $\mathcal{M}$, a finite set of partial functions φ that map an input and a memory state to an output and a new memory state,

$$\varphi: \Sigma \times M \rightarrow \Gamma \times M$$

- F is the next state partial function that given a state and a function from the type Φ, denotes the next state. F is often described as a transition state diagram.

$$F: Q \times \Phi \rightarrow Q$$

- q_0 and m_0 are the initial state and memory respectively.

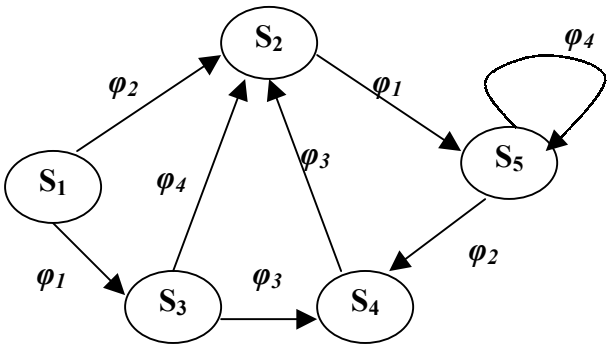

Fig. 2. An abstract example of a X-Machine; φ_i: functions operating on inputs and memory, S_i: states

2.2 X-Machines for Specification

There are several formal techniques (Z, VDM, FSM etc.) that may be used for the specification of computer software, each one possessing advantages in describing either the static or the dynamic part of a system [2]. Thus, the majority of formal specification languages facilitate the modelling of either the data processing, or the control of the system. X-machine is a general computational machine [8] that, being a blend of diagrams and simple formalisms, it is capable to model both the static and the dynamic part of a system. It also possesses highly expressive power being at the same time user friendly by providing a convenient and intuitive way to specify systems.

A simple example of an X-machine will be used as a vehicle of study: "A medical x-ray beaming system is controlled using three buttons: i) one for charging the machine (a single button press increases the voltage by a 10 mV step), ii) one for the beam activation and iii) one for resetting the machine at any time. The system will only beam if the charge in mV has reached a preset maximum, e.g. 30mV. Any attempts to increase the charge of the machine should be rejected, since there is a danger to seriously injure the patient".

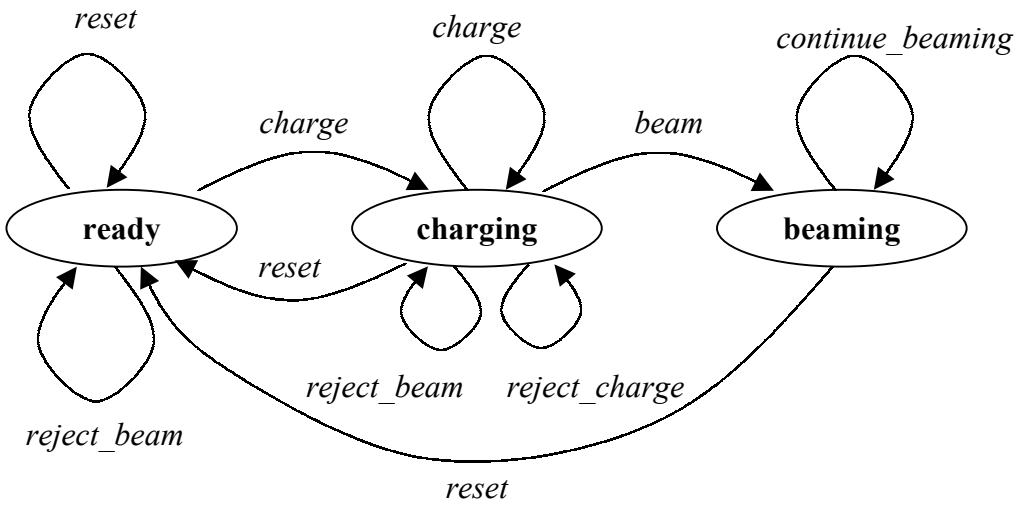

Fig. 3. The state transition diagram of the stream X-machine specification.

The state transition diagram of the stream X-machine $\mathcal{M}$ = (Σ, Γ, Q, M, Φ, F, q_0, m_0) corresponding to the system described above is shown in figure 3. The X-machine's input set is:

Σ = {`charge_button, beam_button, reset_button`}

The output of the system consists of a set of messages that are displayed on a screen together with the current charge of the machine:

Γ = { `MachineCharging, ChargeRejected, BeamRejected, MachineBeaming, ContinueBeaming, MachineReseting` } $\times$ N_0

The set of states is:

Q = { `ready, charging, beaming` }

The X-machine's memory M is:

M = (`MaxCharge , CurrentCharge`)

where:

- `MaxCharge` is a variable holding the maximum accumulating voltage accepted by the machine and takes values from a set, e.g. `MaxCharge` $\in$ {`30`}.

- `CurrentCharge` is a variable holding the current voltage.

Initially, the machine is in the state ready, the current voltage is zero, while its maximum value is 30.

q_0 = `ready`, m_0 = `(30, 0)`

The next state function F: $Q \times \Phi \rightarrow Q$ is shown diagrammatically in figure 3. Finally, the functions of the X-machine need to be defined. The X-machine functions get as input an event and the current state of the memory, and they produce an output and a new memory:

$\varphi : \Sigma \times M \rightarrow \Gamma \times M$

The functions are defined below. The notation used is:

$\varphi(\sigma,m) = (\gamma,m')$ `if condition`

and it is very close to the X-Machine Description Language [9], which is intended to be an interchange language between X-Machine tools.

```
charge(charge_button, (MaxCharge, CurrentCharge))=
((MachineCharging, CurrentCharge+10), (MaxCharge, CurrentCharge+10))
      if CurrentCharge+10 < MaxCharge

charge(charge_button, (MaxCharge, CurrentCharge))=
((MachineCharging, MaxCharge), (MaxCharge, MaxCharge))
      if CurrentCharge+10  ≥ MaxCharge ∧ MaxCharge ≠ CurrentCharge

reject_charge(charge_button, (MaxCharge, MaxCharge))=
 ((ChargeRejected,MaxCharge),(MaxCharge,MaxCharge)).

reject_beam(beam_button, (MaxCharge, CurrentCharge))=
((BeamRejected, MaxCharge),(MaxCharge, CurrentCharge))
      if CurrentCharge < MaxCharge

beam(beam_button, (MaxCharge, MaxCharge))=
  ((MachineBeaming,0),(MaxCharge,0))

reset(reset_button, (MaxCharge, CurrentCharge))=
  ((MachineReseting,0), (MaxCharge,0)).

continue_beaming(button, (MaxCharge,0))=
  ((ContinueBeaming,0), (MaxCharge,0))
      if button ∈ {beam_button, charge_button}
```

2.3 X-Machines for Testing

Ipate and Holcombe [7] presented a testing method, which is a generalisation of Chow's W-method [10] for finite state machine testing. It is proved that this testing method finds all faults in an implementation [11]. The method works based on the following assumptions:

- the specification and the implementation of the system can be represented as X-machines,
- the machine corresponding to the specification and the machine corresponding to the implementation have the same type Φ.

The associated automaton $\mathcal{A}$=(Φ, Q, F, q_0) of an X-machine $\mathcal{M}$ = (Σ, Γ, Q, M, Φ, F, q_0, m_0) is defined as the conversion of the X-machine $\mathcal{M}$ to a FSA by treating the elements of Φ as abstract input symbols. Assuming the above, the method also requires that:

- the specification satisfies the design for test conditions, and

- its associated automaton is minimal.

The design for test conditions state that the type Φ of the two machines is both complete with respect to M and output distinguishable [8].

A processing function $\varphi \in \Phi$ is called *complete* with respect to M if:

$$\forall\, m \in M, \exists\, \sigma \in \Sigma \text{ such that } (\sigma, m) \in \text{dom } \varphi$$

A type Φ is called complete with respect to M if any basic function will be able to process all memory values, that is if:

$$\forall\, \varphi \in \Phi, \varphi \text{ is complete with respect to M}$$

A type Φ is called *output distinguishable* if any two different processing functions will produce different outputs on each input / memory pair, that is if

$$\forall\, \varphi_1, \varphi_2 \in \Phi \text{ if } \exists\, m \in M, \sigma \in \Sigma \text{ such that for some } m_1', m_2' \in M, \gamma \in \Gamma$$
$$\varphi_1(\sigma, m) = (\gamma, m_1') \text{ and } \varphi_2(\sigma, m) = (\gamma, m_2'), \text{ then } \varphi_1 = \varphi_2.$$

When these requirements are met, the W-method may be employed to produce the k-test set X of the associated automaton, where k is the difference in the number of states of the associated automata corresponding to the specification and the implementation. The test-set X consists of sequences of inputs for the associated automaton. Let S be a *state cover* and W a *characterisation set* of the associated automaton, then a k-test set is:

$$X = S(\Sigma^{k+1} \cup \Sigma^k \cup \ldots \cup \Sigma \cup \{\varepsilon\})W$$

The fundamental test function, which is defined below, is recursive and converts these sequences into sequences of inputs for the X-machine. The produced test-set is proved to find all faults in the implementation [8].

Let $\mathcal{M}$ = $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be a deterministic stream X-machine with Φ complete with respect to M and let $q \in Q$, $m \in M$. A function $t_{q,m}: \Phi^* \rightarrow \Sigma^*$ is defined as:

$t_{q,m}(\varepsilon) = \varepsilon$ (the empty input symbol)

or

$$t_{q,m}(\varphi_1 \ldots \varphi_n \varphi_{n+1}) = \begin{cases} t_{q,m}(\varphi_1 \varphi_2 \ldots \varphi_n)\, \sigma_{n+1}, \\ \quad \textbf{if } \exists \text{ a path } q, q_1, \ldots, q_{n-1}, q_n \text{ in M starting from q, where } \sigma_{n+1} \text{ is such} \\ \quad \text{that } (m_n, \sigma_{n+1}) \in \text{dom } \varphi_{n+1} \text{ and } m_n \text{ is the final value computed by} \\ \quad \text{the machine along the above path on the input sequence } t_{q,m}(\varphi_1 \varphi_2 \ldots \varphi_n) \\ t_{q,m}(\varphi_1 \varphi_2 \ldots \varphi_n), \text{ otherwise} \end{cases}$$

and this function $t_{q,m}$ will be called a *test function of $\mathcal{M}$ with respect to q and m.*

The testing process can therefore be performed automatically by checking whether the output sequences produced by the implementation are identical with the ones expected from the specification. Using the above described method for the example presented in the previous section and following the W-method, a characterisation set and a state cover of the associated automaton are `W={reject_charge, continue_beaming}` and `S={ε, charge, charge beam}` respectively. Using them and assuming that k=0, the 0-test set for the associated automaton is produced and using the fundamental test function the test sets as inputs for the X-machine are derived.

For example, one test for the associated automaton is the sequence <charge, reset, charge> that is transformed to a test for the X-machine <charge_button, reset_button, charge_button>. If the implementation fails to produce the same output sequence to that of the specification, then the system would probably have a fault in resetting the x-ray at any given time, as the specification imposes. Thus using the whole test set it is possible to test the implementation and prove that it is correct with respect to the specification.

Concluding, X-machines not only provide a model to specify a system but also offer a strategy to test the implementation against the specification [12]. With the addition of a method to verify whether several desired properties hold in this specification or not, X-machines will provide a complete methodology to develop a safety critical system.

3 Model Checking X-Machine Specifications

Model Checking is a formal verification technique, which is based on the exhaustive exploration of a given state space trying to determine whether a given property is satisfied by the system. A model checker takes a model and a property as inputs and outputs either a claim that the property is true or a counterexample falsifying the property. In order to use model checking, the most efficient way to express a model is any kind of state machine, a CCS agent [13], a Petri Net [14], a CSP agent [15], etc. The property is usually expressed as a Modal-Mu property or a Computational Tree Logic (CTL) property. The most common properties to check are that something will *never* occur or something that will *eventually* occur.

In *Temporal Logic Model Checking* [16] a property is expressed as a formula in a certain temporal logic. The verification can be accomplished using an efficient breadth first search procedure, which views the transition system as a model for the logic and determines if the specifications are satisfied by that model. This approach is simple, completely automated but has one major problem, namely the state explosion.

Symbolic Model Checking [17] is a model checking variant that instead of visiting individual states, visits a set of states at a time. Thus, symbolic model checking avoids the state explosion problem [18], through the use of Binary Decision Diagrams [19]. This method has also the advantage that it is completely automated. A property is specified as a temporal logic formula [17].

3.1 A Methodology for Model Checking X-Machines

In X-Machines, the search of some properties P of the model being true or false cannot be applied in a straightforward manner, since these properties are implicitly expressed in the X-machine memory values. Thus, checking whether a property p_i is valid in some states of the X-Machine means whether there are some states in which some memory values satisfy the property p_i. In essence, search should be performed through all instances of the memory of a X-Machine model. For example, in order to verify that the current charge of the x-ray beaming machine will never exceed the maximum charge in any of the states, a model checker needs to search through all possible states as well as all possible instances of memory. Therefore, the appropriate model that facilitates model checking $<W,R,\pi>$ should include:

- W is the set of all possible states of the X-Machines combined with all possible instances of memory in each state,
- R is the set of transitions between states in W,

- π is the truth assignment function, i.e. given a member in W (a state with a specific memory instance) which properties are true depending on the values of this memory instance.

For example, some members of W are: `charging (10,30)`, `charging (20,30)`, `charging (30,30)` etc., i.e. the x-ray beamer is at state charging, the current charge is 10mV and the maximum charge is 30mV, and so on respectively. Then, the property `CurrentCharge < MaxCharge` is true in `charging (10,30)` and `charging (20,30)` but false in `charging (30,30)`.

Bearing the above, model checking of a X-Machine model for specific properties can be achieved through the transformation of the X-Machine into the form <W,R,π>. More generally, the resulting state space (W,R) resembles a FSM (Σ, Γ, Q_{fsm}, T, q_{0fsm}) where:

- Σ is a finite set that is called the input alphabet.
- Γ is a finite set that is called the output alphabet.
- Q_{fsm} is the finite set of states, $Q_{fsm} \subseteq W$.
- T is the (partial) transition function, T: $Q_{fsm} \times \Sigma \rightarrow Q_{fsm} \times \Gamma$, (T is a labelled R).
- q_{ofsm} is an initial state,

that encapsulates memory values which correspond to properties in each of its states.

The proposed model checking provides a way to prove whether a property expressed within a temporal logic formula is satisfied by the X-Machine or not. The methodology can be completely automated and used by various X-Machine tools and avoid the state explosion through the selective expansion of the properties in question. The next section shows how it is possible to derive a state space for a X-Machine.

3.2 Derivation of the Equivalent FSM

The sets Γ and Σ are the same in both models. Below, we present a way to derive the set of states Q_{fsm} and the transition function T.

Let Q be the set of states of the X-machine and the memory tuple $M_1 \times M_2 \times \ldots \times M_n$ with n memory variables, where M_i is the set of all the possible values. Then, let the set of states S be the candidate set of FSM states:

$$S = Q \times M_1 \times M_2 \times \ldots \times M_n$$

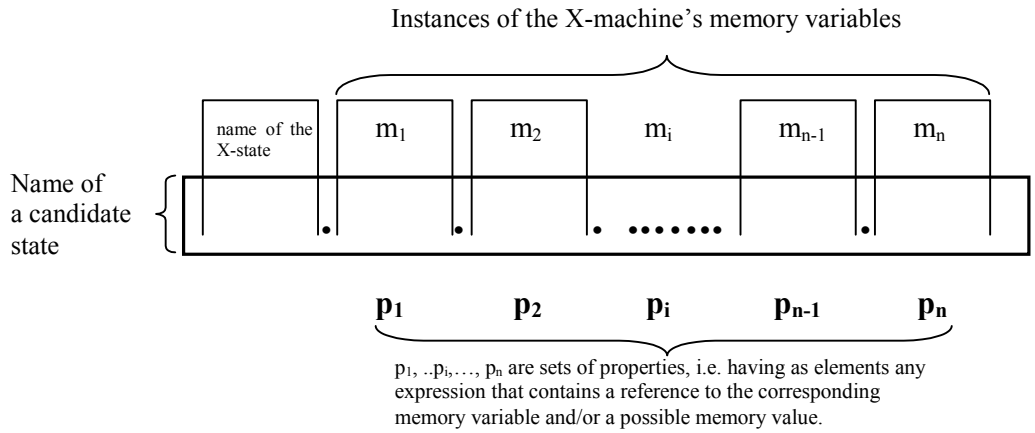

Fig. 4. The construction of compound names of the equivalent FSM states

The initial state of the equivalent FSM q_{0fsm} is:

$$q_{0fsm} = (q_0, m_{01}, m_{02}, \ldots, m_{0n})$$

where q_0 is the initial state of the X-Machine and $m_0 = (m_{01}, m_{02}, \ldots, m_{0n})$ is the initial memory.

Practically, one could imagine that a state $s \in S$ is constructed as a compound name of the name of the states $q \in Q$ and a combination of all possible memory values M_i defined in the X-Machine specification (Figure 4). All candidate transitions of the equivalent FSM are defined as:

$$
\begin{aligned}
T1 = \{ & ((q, m_1, m_2, \ldots, m_n), \sigma, (q', m_1', m_2', \ldots, m_n') , \gamma) \mid \\
& \sigma \in \Sigma, \gamma \in \Gamma \wedge \\
& q, q' \in Q \wedge \\
& (m_1, m_2, \ldots, m_n), (m_1', m_2', \ldots, m_n') \in M \wedge \\
& \exists \varphi \in \Phi \bullet \varphi(\sigma, (m_1, m_2, \ldots, m_n)) = (\gamma, (m_1', m_2', \ldots, m_n')) \wedge \\
& \exists f \in F \bullet f(q, \varphi) = q' \}
\end{aligned}
$$

Some of the states in these candidate transitions are unreachable from the initial state q_{ofsm}, i.e. there is no path from leading to those states. Therefore, the transitions, which should be excluded from the set of transitions, are:

$$
\begin{aligned}
T2 = \{ & (s_1, \sigma_1, s_1', \gamma_1) \mid \\
& (s_1, \sigma_1, s_1', \gamma_1) \in T1 \wedge \\
& (s, \sigma_2, s_1, \gamma_2) \notin T1 \wedge \\
& \sigma_1, \sigma_2 \in \Sigma \wedge \gamma_1, \gamma_2 \in \Gamma \wedge \\
& s, s_1, s_2 \in S \wedge \\
& s_1 \neq q_{0fsm} \}
\end{aligned}
$$

From the above set, the initial state should be excluded. The set of transitions of the equivalent FSM becomes: T = T1 - T2. Finally, since the set S contains some redundant states, i.e. states without any transition or unreachable states from the initial state, the set Q_{fsm} becomes:

$$Q_{fsm} = \{ s \mid s, s' \in S \wedge ((s, \sigma, s', \gamma) \in T \vee (s', \sigma, s, \gamma) \in T) \wedge \sigma \in \Sigma, \gamma \in \Gamma \}$$

In the example presented in the previous section, the equivalent FSM derived from the X-Machine specification is illustrated in Figure 5.

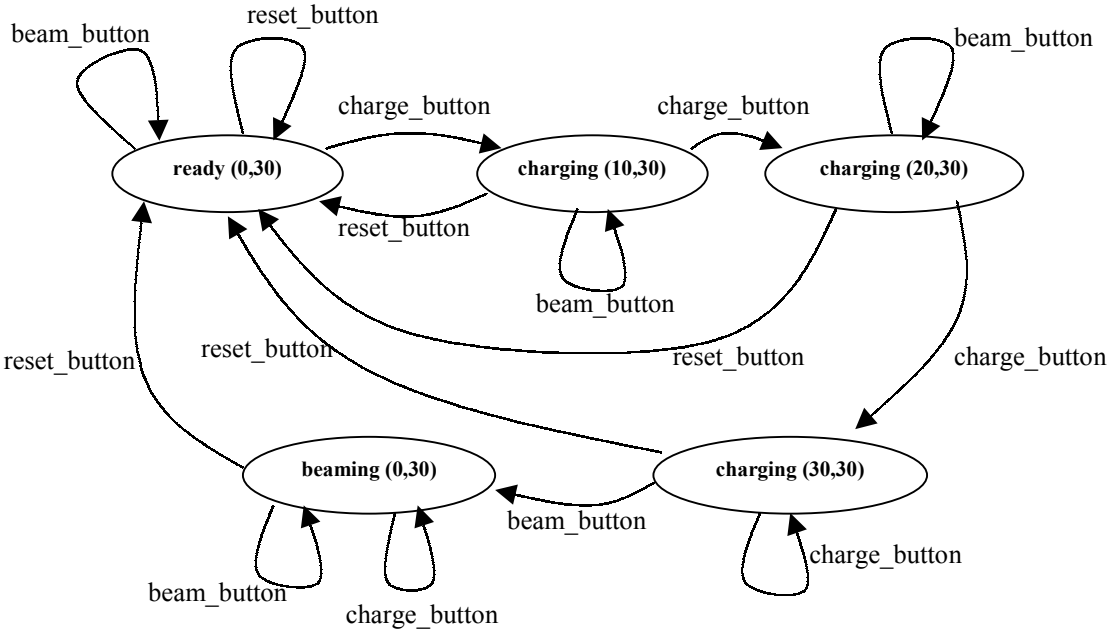

Fig. 5. The equivalent FSM of the x-ray beamer X-Machine specification

3.3 Checking for Properties

It is now possible to query the model if it has the desired properties by searching the state space defined by the equivalent FSM in order to verify that the properties are satisfied. In order to express such queries, there is a need for a mathematical language with built-in notion of time. Temporal logic seems to be the most appropriate. With the use of modal operators, i.e.:

- necessarily ($\Box$),
- possibly ($\Diamond$) and
- next ($\bigcirc$),

it is possible to express if a desired property is valid in the whole model or in part of it, starting from the initial state. In the example used previously in this paper the logic proposition: $\Box$(`CurrentCharge ≤ 30`), expresses that "it is impossible the voltage to become greater than the maximum value, which in this case is 30", meaning that the machine will never be charged with more than the permitted value, avoiding the chance to injure the patient. The translation of this temporal formula is that the requirement expressed with that formula holds in the model if in every state of the state space the following is true; the voltage is less or equal to 30 (the property p ⇔ `CurrentCharge ≤ 30`, is true in every state).

Since there is a need for more expressive and flexible operators, a more proper variation of temporal logic is the CTL [16]. A *CTL formula* contains:

- a boolean expression,
- an existential (E) path formula,
- a universal (A) path formula, or } path quantifiers
- the application of standard Boolean operators to CTL formulae.

A *path formula* contains:

- the application of the temporal operators:
  - next (X),
  - eventually (F),
  - or globally (G), to a CTL formula; or } state quantifiers
- the application of until (U) to a pair of CTL formulae.

In a quantified CTL formula the temporal operators always come in pairs of a path quantifier and a state quantifier. CTL formulae are interpreted with respect to an infinite computation tree derived from finite state transition machines. Each path in the tree is a sequence of states. So, for example, if p is a Boolean expression then CTL formulae:

- AG p translates into "in all paths, in all states p holds", i.e. p is invariant;
- EF p translates into "in some path, there is some state in which p is true" or more informally, i.e. p is potentially true.

There are four different cases and all are explained with the aid of the example in the following table.

Example of Property p	Temporal Operators in CTL	Explanation
CurrentCharge ≤ 30	AG p	For every path and for every state in the path, the property p is be valid
CurrentCharge < 30	AF p	For every path, there exists at least one state where p is valid
CurrentCharge = 0	EG p	There are some paths (at least one), where in every state of these paths p is valid
MaxCharge=CurrentCharge	EF p	There are some paths (at least one), where in some states of these paths p is valid

3.4 Implementation Issues

Efficiency in the model checking a X-Machine specification is still an issue under consideration due to the state explosion. Efficiency also depends on query, i.e.:

- the CTL operators involved, and
- the number of properties involved.

For the first, there exist appropriate search algorithms, which can be applied in the model checking paradigm, according to the given query. For instance, in AG p, since an exhaustive search is required, Depth-First Search seems rather appropriate because of its memory efficiency in finite search spaces. In contrast, in EF p, Iterative Deepening can be perform better in large search spaces in order to find at least one state where the property holds.

For the latter, optimisation in deriving the equivalent FSM can be performed. If the properties in the query refer to the whole memory tuple, then nothing can be done and the X-Machine should be fully expanded into a FSM. If, however, some of the elements in memory tuple do not correspond to a given property in the query, then the memory values of this element should not participate in the construction of the equivalent FSM, thus reducing the number of states in it. Let $P= \{p_a, p_b, .., p_m\}$ the set of properties in the CTL formula, and $M_1\times M_2\times \ldots\times M_n$ the memory of the X-Machine. The properties may correspond to the memory elements i, j ,…, k. Then, let the set of states S, which is the candidate set of FSM states can be reduced to $S = Q\times M_i\times M_j\times\ldots\times M_k$. The derivation of the equivalent FSM is based on S, and therefore the resulting FSM contains exactly the necessary states for model checking.

The actual derivation of the equivalent FSM is possible under certain assumptions concerning infinite domains. The strictest conditions, which can be imposed on the implementation of a model checker are:

- the domain of every element in the memory tuple is both finite and discrete, and
- the input set is finite and discrete.

There are, however, ways to relax those requirements. Infinite memory values and infinite input set refer either to:

- infinite values within a finite range
- discrete values within infinite range

It is possible to apply the proposed algorithm to X-machine models with infinite variables in the memory, with the restriction to check only properties relevant to the finite variables. This is feasible by "forgetting" all the infinite variables and using in the algorithm only the discrete and finite ones.

In the case of infinity between a range, the values should be changed to discrete with an accepted step (something acceptable in computer systems). For discrete but infinite values an assumption should be made about the maximum value this variable could take.

4 Conclusions

X-Machines is a formal method, which facilitates the development of correct systems [8]. We have presented a methodology for model checking X-Machine specifications. The methodology satisfies all the criteria of a technique that can verify the specification of a model, i.e. it can be fully automated, it can tackle in some cases the state explosion problem and it can act as the complementary part in the system development. The effectiveness of X-Machines theory in specification and testing has been already demonstrated elsewhere [7, 12]. Together with the proposed model

checking method, X-Machines consists an integrated formal method, which can be used in all stages of safety critical system development. Future work will include the implementation of tools for model checking X-Machine specifications, which will be added to the existing tools already built around X-Machines [20, 21], thus providing an integrated framework for system development.

References

1 Leveson N.G., Safeware: System Safety and Computers, Addison Wesley Longman, 1995.

2 Clarke E., Wing J. M., "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, Vol.28, No.4, December 1996, pp. 626-643.

3 Pnueli A., "The temporal logic of programs", In Proceedings of the 18th Annual Symposium on Foundations of Computer Science. IEEE Computer Society Press, New York, 1997, pp. 46–57.

4 Goodenough J.B., Gerhart S.L., "Toward a Theory of Test Data Selection", IEEE Trans. Software Eng., Vol. 1, No. 2, June 1975, pp.156-173.

5 Eilenberg S., Automata Machines and Languages, Vol. A, Academic Press, 1974.

6 Holcombe M., "X-machines as a basis for dynamic system specification", Software Engineering Journal, Vol.3, No.2, 1988, pp. 69-76.

7 Ipate F. and Holcombe M., "Specification and testing using generalised machines: a presentation and a case study", Software Testing, Verification and Reliability, Vol.8, 1998, pp. 61-81.

8 Holcombe M. and Ipate F., Correct Systems: Building a Business Process Solution, Springer Verlag, London, 1998.

9 Kefalas P. and Kapeti E., "A Design Language and Tool for X-Machines Specification", In Advances in Informatics, D.I. Fotiadis, S.D. Nikolopoulos (eds.), World Scientific Publishing Company, April 2000, pp. 134-145.

10 Chow T.S., "Testing Software Design Modeled by Finite-State Machines," IEEE Transactions on Software Engineering, Vol.SE-4, No.3, 1978, pp.178-187.

11 Ipate F. and Holcombe M., "An integration testing method that is proved to find all faults", International Journal of Computer Mathematics, Vol.63, No.3, 1997, pp. 159-178.

12 Kehris E., Eleftherakis G., and Kefalas P., "Using X-Machines to Model and Test Discrete Event Simulation Programs", In Systems and Control: Theory and Applications, N. Mastorakis (ed.),, World Scientific and Engineering Society Press, July 2000, pp. 163-168.

13 Milner A., "A Calculus of Communicating Systems", Lecture Notes in Computer Science LNCS 92, Springer-Verlag, 1980.

14 Petri, C.A., "Communication with Automata", PhD Dissertation, University of Bonn, Bonn, West Germany, 1962.

15 Hoare C.A.R., Communicating Sequential Processes, Prentice Hall International, 1985.

16 Clarke E.M., Emerson E.A., and Sistla A.P., "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", ACM Trans. Programming Languages and Systems, vol. 8, no. 2, Apr. 1986, pp. 244–263.

17 McMillan K.L., "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

18 Burch J.R., Clarke E.M., McMillan K.L., Dill D.L. and Hwang J., "Symbolic model checking:10^{20} states and beyond", in Symposium on Logic in Computer Science, 1990

19 Bryant R.E., “Graph-Based Algorithms for Boolean Function Manipulation”, IEEE Transactions on Computers, Vol. C-35, No. 8 (August, 1986), pp. 677-691. Reprinted in M. Yoeli, Formal Verification of Hardware Design, IEEE Computer Society Press, 1990, pp. 253-267.
20 Kefalas P., “Automatic Translation from X-Machines to Prolog”, Technical Report TR-CS01/2000, Dept. of Computer Science, CITY Liberal Studies, January 2000
21 Kefalas P., Sotiriadou A., “Transforming X-Machines to Z Specification”, Technical Report TR-CS06/2000, Dept. of Computer Science, CITY Liberal Studies, January 2000