A DESIGN LANGUAGE AND TOOL FOR X-MACHINES SPECIFICATION

PARASKEVI KAPETI AND PETROS KEFALAS

Department of Computer Science, City Liberal Studies, Affiliated Institution of the University of Sheffield, 13 Tsimiski Str., 546 24 Thessaloniki, Greece E-mail: {kapeti, kefalas}@city.academic.gr

One of the most essential phases of software development is the system specification. X-machines is an intuitive formal method which can be easily applied at this phase and also facilitate testing in order to prove the correctness of an implementation with respect to the specification. Although the X-Machine theory is already defined in detail mathematically, there is a lack of automated tools which can facilitate its use. In this paper, a brief introduction to the X-Machine theory is presented and a language for specifying X-Machines is defined, which could act as a communication protocol among various X-Machine tools. In addition, a proof tool is presented which checks special properties of X-Machines that are defined as prerequisites for the X-Machine testing algorithm.

1 Introduction

Almost in any variation of a software development model, the process starts with *specification* and finishes with *testing*. The specification phase is a very important one, during which the directives for the rest of the development process are set. The specification models used are able to, on one hand, supply a concrete, concise and coherent specification at different levels of abstraction in order to avoid any misconceptions among developers, and on the other hand to establish the system's properties, so that the system can be proved to handle a certain set of situations.

Various methods have appeared, such as the *Z specification language* [1], *the state-charts* [2], etc. which can be roughly classified according to their degree of formalism [3]. Formal techniques are based on mathematics whereas completely informal techniques are based on natural languages. The latter, together with intermediate ones, known as semi-formal, have the major disadvantage of being incomplete and inconsistent. On the other hand, formal techniques are difficult to be learned or used, especially when the people who apply them do not have a mathematical background. Despite of their disadvantages, formal models are the only ones that can guarantee the *completeness* and consequently the *correctness* of a model, which justifies the current research community's concern for them.

An area of formal models is based on the *finite state machines* (FSM) theory enriched with some extensions. FSM are popular, not only because of their formal nature, but also because of their expressiveness and intuitiveness. They can represent in a very compact way information about states, transitions, inputs, outputs, conditions that should hold for a transition be followed etc. Further more, the notation used is simple and requires only a basic mathematical background. The

X-Machine theory is based on FSM theory and strongly integrates the problems of data and control processing specification in a single model as well as the testing of implementation through the specification [4]. Although the X-Machine theory is well established, there have not yet appeared automated tools that would support X-Machine definition and checking of special X-Machine properties. The current work contributes to solving the above problem by a X-Machine design language and tool for checking X-Machine properties.

2 A brief introduction to X-Machines

Specification methods describe either the *dynamic* or the *static* aspect of a system, or both, and they are respectively distinguished to *operational*, *descriptive* or *dual*. The dynamic part represents the system's control and is usually defined in terms of states, transitions, event triggering etc., whereas the static part represents the system's data and is usually defined in terms of data structures, tables and data processing functions for retrieving, updating and checking data. *Petri-nets*, for example, could be classified as an operational specification method, while *entity-relationship diagrams* as a descriptive one. The diagrammatic nature as well as the generality of FSM makes them an intuitive tool for specifying the dynamics of a computational system. However, FSM are inadequate to specify systems which include complex data structures and data manipulation. Therefore, FSM appear to be deficient to handle the corresponding static part.

X-machines overcome this deficiency by allowing both control (operational) and data (descriptive) processing of a system to be described separately [5], since the control processing is described by inputs, outputs, states and transitions, similarly to simple FSM, whereas the data processing is described by a memory and a set of functions. The memory acts as a global set of values, which can be retrieved and updated at any time and represents the main data that can be processed by the machine. The set of functions is the main data processing unit of the X-Machine. In order to handle the data, the functions can either perform alterations to the memory on their own or can call external data processing functions to do so. The latter are essential for an X-Machine, since they establish a communication between the X-Machine's control processing and the data-processing unit.

Except from the specification phase of a system, the X-Machine modeling strongly addresses the testing phase, too. According to the X-Machine testing theory, the specification and the implementation can both be expressed as X-Machines and it is proved that a test set which finds all the faults of the implementation can be generated. Thus, X-Machines act, not only as a specification tool, but also guarantee the correctness of the implementation with respect to the specification. Additionally, in a more practical perspective, it brings the first and final phases of a system's development very close, preserves homogeneity among all

the intermediate phases, and saves time and resources, since the testing process can be fully automated.

Formally a X-Machine is defined by [6] as:

$$\mathbf{X = (I, O, M, S, F, T, IS, IMS)}$$

where, **I** is a sequence of inputs (a finite set of constants describes the possible input values), **O** is a sequence of outputs (a finite set of constants describes the possible output values), **M** is the memory, **S** is a set of states, **F** is set of functions ($\mathbf{F: I \times M \rightarrow O \times M}$), **T** is a set of transitions ($\mathbf{T: S \times F \rightarrow S}$), **IS** is the initial State, and **IMS** is the initial memory state.

2.1 A X-Machine example

The usual procedure of a vending machine is that customer enters some coins (50, 100), selects a soft-drink (coke, lemonade, sprite) and finally presses a button to execute the order (enter button). The prices of the drinks are stored in some database that is accessed by a set of external data processing functions related to the model. In figure 1, the ovals denote the system states, the arcs the transitions and the arcs' labels the functions triggered.

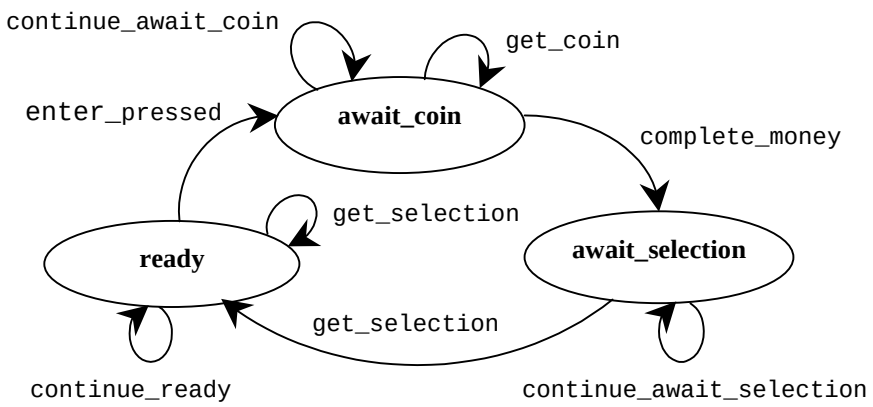

Figure 1. X-machine state diagram for the automatic soft drinks vending machine

The machine's input refer either to the coins (`CoinType`), or to the buttons pressed (`SelectionType`), or finally to a button (`ProcessType`) which executes the order.

```
Input = CoinType ∪ SelectionType ∪ ProcessType
 where CoinType = {50drc, 100drc}
         SelectionType = {coke_b, lemonade_b, sprite_b}
         ProcessType = {enter}
```

The machine's output consists of three basic elements: a) The current state of the machine (`State`), b) The amount of money inserted so far (`AmountType`), and c) a message (`AvailableDrinkType`) informing the user whether a drink is available or not.

```
Output = State × AmountType × AvailableDrinkType
where AmountType = {50, 100, 150, 200, 250, 300}
        AvailableDrinkType = {drink, no_drink}
        State = {await_coin, ready, await_selection}
```

The machine's memory includes information for the amount of money inserted so far (`AmountType`), and the drink that the user has selected (`SelectionType`).

```
Memory = AmountType × SelectionType
```

The initial state of the machine is the `await_coin` state, and the initial memory of the machine defines that no coins are inserted (the amount of money inserted is 0) as well as the default soft drink selection is coke (`coke_b`).

```
Initial State = await_coin
Initial Memory = (0, coke_b)
```

The process of the machine is based on the functions defined, which take as parameters a token representing the input and a n-tuple representing the current memory contents, and produce two n-tuples, one concerning the output and one the new memory contents. For example, the `get_coin` function takes as parameters some input (`x`) and some current memory state (`amount, selection`). If the input `x` is a coin (`x ∈ coin_type`) and the newly inserted coin `x` with the amount of money that is stored at the current memory state is not sufficient (`¬sufficient(x, amount)`) for the user to get a drink, then the machine should continue waiting for a coin. The output of this function should inform the user that the machine is still waiting for more coins (`await_coin`), that the new amount of money in the machine is the old amount of money plus the coin inserted (`add(x, amount)`), and finally that there is no available drink yet (`no_drink`). The function

should also update the machine's memory which should contain the new amount of money as well as some drink selection. As the user has not given any input for this, the drink selection of the new memory state is the same with the drink selection of the previous memory state.

```
get_coin(x, (amount, selection)) =
   if (x ∈ coin_type and ¬sufficient(x, amount) then
   ((await_coin, add(x, amount), no_drink),
   (add(x, amount), selection))
```

All the elements of the above function are variables or n-tuples of variables, except `sufficient` and `add`. These are external functions that enable the communication between the X-Machine and a data-processing machine. For example, `sufficient` returns true when a coin and an amount of money are enough to get a drink, and false in any other case:

```
sufficient: (coin_type × amount_type) = B
```

The rest of the X-Machine's functions are shown in the Appendix.

3 The X-Machine Design Language (XMDL)

It is a common ground that if software tools are to be built for a model, a detailed but general representation of such model should be specified. The main reasons for which already existing languages have been rejected for such representation are:

- The syntax as well the semantics of either functional or logic programming languages, are much more general and loose than the ones desired.
- The computational description of a X-Machine should be close enough to the corresponding mathematical description and notation.
- The model's representation using a specific programming language would require for the user to have some knowledge of the programming language.

An important aim of such representation is the reusability of the same X-Machines by different tools, therefore the representation should be independent of the programming language that would be used by any of these tools. Therefore, we have decided to define a new description language for representing a X-Machine, namely XMDL, which supports the following:

- *Constants* required in data type, state, and function definitions, for example: `drc100.`
- *Sets*, for example, the states of a X-Machine can be defined as: `#state {await_coin,await_selection,ready}.`
- *Data types* defined as finite sets of constant values, for example: `#datatype coin = {drc50, drc100}`

- *Variables* defined as any sequence of characters starting with a '?', for example:
  `if ?x belongs selection or ?x is enter then …`
- *The membership operator*, for example: `?x belongs coin.`
- *The equality operator*, for `example: sufficient(?x,?current_amount) is true.`
- *N-Tuples*, for example: `(200,coke_b)`
- *External functions*, for example:
  `#x_fun boolean = sufficient(coin, amount)`

A partial XMDL code of the X-Machine defined above is given in the figure 2.

Data types

```
#datatype coin = {drc50, drc100}
#datatype selection = {coke_b, lemonade_b, sprite_b}
#datatype process = {enter}
#datatype drink_ready = {yes, no}
#datatype amount = {0, 50, 100, 150, 200, 250, 300}
#datatype boolean {true, false}
```

Set of states

```
#state {await_coin, await_selection, ready}
```

Input type

```
#input (coin) union (selection) union (process)
```

Output type

```
#output (state, amount, drink_ready)
```

Memory type

```
#memory (amount, selection)
```

Initial State value

```
#init_state await_coin
```

Initial Memory value

```
#init_memory (0, coke_b)
```

Function

```
#fun get_coin((?x), (?current_amount, ?current_selection)):
if ?x belongs coin and ¬sufficient(?x, ?current_amount) is
true then
((await_coin, add(?x, ?current_amount), no_drink),
 (add(?x, ?current_amount), ?current_selection))
```

External functions

```
#x_fun boolean = sufficient(coin, amount)
#x_fun amount = add(coin, amount)
```

Transitions

```
#transition (await_coin, get_coin) = await_coin
```

Figure 2. Partial code for the automatic soft drink vending machine in XMDL

3.1 Correctness of description of a X-Machine

Based on the above design language, there are three basic levels of correctness for a X-Machine description, namely the *syntactic* level, the *semantic* level and the *logical* level.

3.1.1 Syntactic level

At this phase only the syntax of the X-Machine is checked, thus identifying all syntax errors in the XMDL code, as for example a missing bracket, a misspelled reserved word etc. However, errors such as `in #state {idle, record, idle, play}` where the set contains duplicate elements, are not intended to be identified at this level.

3.1.2 Semantic level

At this level mistakes concerning data types, functions and sets are identified. For example, when the function `((record), c) = ((recording), c)` is defined, it is assumed, that `record` is a legal input (is included in the input grammar) and `recording` is a legal output respectively. This level guarantees that a model is at least mathematically correct before any tool attempts to prove other properties for it. Therefore, it is concluded that at least three basic conditions should be checked:

- *Data types*: the types of parameters and returning values should be legal. Data type checking also applies to the definition of external operators. They can be used as part of the input, output and memory state definitions in functions. So the system has to guarantee that the values returned by the operators are of the expected data type. Somebody could argue that this kind of type checking could be applied at the previous phase of syntactic analysis. This would require the language to be *positional*, meaning that the user should define all states before any of the functions, all the functions before any of the transitions, etc. As positioning could be too restrictive for the user, it was preferred to check data types at this second level.
- *Sets*: Anything that is defined as a set (it is included in `{}` and/or it is expected by the language's syntax to be a set) should not contain duplicate elements.
- *Functions*: Each function is applied to a tuple containing an input and a memory state, and produces another tuple containing an output and a memory state. The system should guarantee that the same function should not be able to produce more than one output – memory state tuples for one input – memory state tuple.

3.1.3 Logical level

This phase refers to the model's general consistency. Questions referring to the model's *completeness*, its output *distinguishability*, its *minimality* etc. should be addressed. As they are beyond the levels of grammar or mathematical notation

checking, other tools especially built for such purposes should do so. Such proof tools and their possible extensions are discussed in the following section.

4 The X-Machine Checking Tool

The modeling of a system aims to its comprehension that is accomplished at the X-Machine specification phase and its testing. The X-Machine testing method can be applied given that the system specification and implementation can both be viewed as stream X-Machines with the same set of functions, and the set of functions satisfies some "*design for test conditions*", namely completeness and output-distinguishability. The aim of the X-Machine proof tool presented in this section is to determine whether the testing method can be applied to a specific X-Machine, i.e. the "design for test conditions" hold, and as a consequence to inform the user for any conflicting functions.

A function f of a X-Machine is called *test-complete* with respect to the memory, if:

$$\forall m \in \textbf{Memory}\ \exists\ i \in \textbf{Input} \mid (m, i) \in \textbf{domain f.}$$

The set of all functions Φ of a X-Machine is called complete with respect to the memory, if all its functions are complete with respect to the memory. An algorithm for proving the completeness is listed in figure 3. If the `nonCompleteSet` is empty, function f is complete with respect to the memory. If this holds for all the functions of a X-Machine then Φ can be said to be complete with respect to the memory. The X-Machine proof tool applies the algorithm to all the X-Machine's functions and returns the `nonCompleteSet`, thus informing the user for the model's completeness.

Another algorithm can be devised to prove *output-distinguishability*. The set of functions Φ of a X-Machine is called *output-distinguishable*, if the following holds:

$$\forall\ f_1, f_2 \in \textbf{Functions, if}\ \exists\ m \in \textbf{Memory},\ i \in \textbf{Input} \mid$$
$$f_1(i, m) = (o, m_1') \textbf{ and } f_2(i,m) = (o, m_2')$$
$$\textbf{for some } m_1', m_2' \in \textbf{Memory}, o \in \textbf{Outputs, then f1 = f2.}$$

```
procedure completeness(XMDL code);
possibleMemories ← all combinations of all memory values;
possibleInputs ← all combinations of all possible inputs;
nonCompleteSet ← {};
initialize index for possibleMemories;
while possibleMemories ≠ {}
pm ← get element from possibleMemories;
applicable ← false;
initialize index of possibleInputs;
while possibleInputs ≠ {} and applicable = false
      pi ← get element from possibleInputs;
      if f(pi,pm) ∈ domain f then applicable ← true
endwhile;
if applicable = false then assert pm to nonCompleteSet
endwhile;
return nonCompleteSet.
```

Figure 3. An algorithm for proving completeness of functions of an X-Machine

So, any two different processing functions should produce different outputs for any memory-input pair. The X-Machine proof tool applies the algorithm to the X-Machine specified by XMDL code and returns the conflicting functions. All algorithms used as well as their implementation and other issues under consideration are more extensively described in [7].

The complete tool aims at being the primary design tool for X-Machine modeling and XMLD the common representation of all other tools build for X-Machines independently programming language used to implement them (Fig. 4).

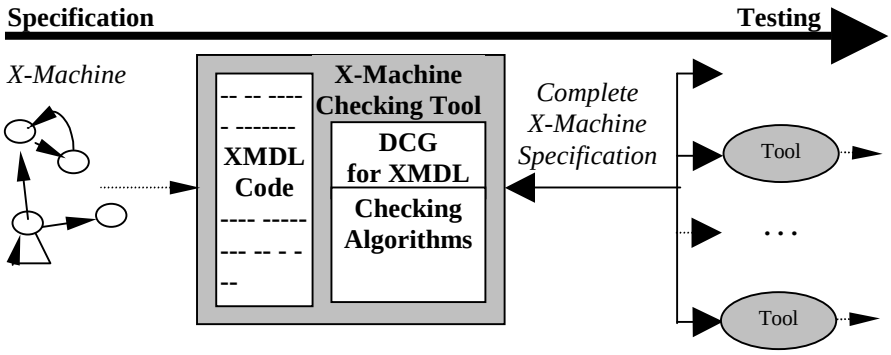

Figure 4. The role of the proposed tool and language in the process of software development based on X-Machine theory

5 Related work

The popularity and intuitiveness of finite state automata was the development of languages and tools for them which refer either to the FSMs' initial definition or to their extensions. Most of these languages are related with some interface that convert a graphical representation to an equivalent textual one and vise versa. The graphical representation serves mainly user-interface purposes, whereas the textual representation is the one actually manipulated by the tool, as the input of an animator, a model checker, a test case generator etc. Such tools are roughly classified in:

- *Educational*, such as ASSIST [8] and JFLAP [9]. Such tools usually do not support data types except of an input, output and stack alphabet. They could not also handle constraints related to the states and/or transitions as well as communication between different state machines. The language they use is restrictive, positional and not easily readable.
- *Software engineering oriented*, such as AUTOFOCUS [10], HYTECH [11] and UPPAAL [12]. Although these are more advanced tools with the ability to handle data types, functions, constraints, assignment statements etc., the language they use is a mixture of mark-up and positional (procedural) language, and often compact and less clear.

None of the above is easily extensible to handle X-Machine modeling. A more detailed review of existing tools is made elsewhere [7].

6 Conclusion, evaluation and further work

The aim of this paper was to set a framework for any computerised tool for X-Machines, which was accomplished by the X-Machine design language. XMDL is very close to the initial mathematical notation of X-Machines. XMDL is characterised as:

- *orthogonal*, in the sense that there are rather few primitive structures that can be combined in rather few ways to build the whole description of a X-Machine,
- *mark-up*, so it supplies the users with the freedom of a non-positioning language,
- *strongly typed*, as it performs type checking, set checking and function checking, as well as exception handling, and

The implementation was based in describing the XMDL syntax and semantic rules by using Prolog's DCG notation. The algorithms described above were also implemented in Prolog. The algorithms implemented are very close to the corresponding mathematical definition of completeness and output-distinguishability. The X-Machine checking tool is the basic prerequisite for any X-

Machine testing tool, because the design-for-test conditions are proved to be met and therefore the test sequences generated by any other tool is complete.

The tool has currently been tested to apply two exhaustive algorithms to quite a limited state space. It is expected that the algorithms will be transformed to make their application to very large state-spaces more efficient. Our intention is to apply those into the parts of X-Machines, and not to the whole model. On the other hand, we are currently working in the implementation of a graphical X-Machine specification tool that would automatically produce the corresponding output in XMDL, and the implementation of a testing tool, more alike [13] but specialised for X-Machines, in order to generate and apply test sequences according to the characterization set method [14].

References

1. Spivey J.M., The Z notation: Reference Manual (2nd Edition), (Prentice Hall Int., 1992)
2. Harel D., Pnueli A., Schmidt J.P., Sherman R., On the Formal Semantics of Statecharts, Proc. 2nd IEEE Symp. Logic in Computer Science, IEEE Press, New York, (1987) pp. 54 – 64
3. Bucci G., Cambanai M., Nesi P., Tools for specifing Real-Time Systems, *Real-Time Systems* **8** (1995) pp. 117-172
4. Holcombe M., Ipate F., Correct Systems: Building a Business Process Solution, (Springer-Verlag, 1998)
5. Fairtlough M., Holcombe M., Ipate F., Jordan C., Laycock G., Duan Z., Using an X-machine to model a video cassette recorder, *Current issues in Electronic modelling* **3** (1995) pp. 141-151
6. Ipate F., Holcombe M., Specification and testing using generalised machines: a presentation and a case study, Software Testing, *Verification and Reliability* **8** (1998) pp. 61-81
7. Kapeti E., Kefalas P., The X-Machine Design Language XMDL and a Tool for Proving Completeness and Output-Distinguishability, (Technical Report CS-06/99, Department of Computer Science, City Liberal Studies, 1999)
8. Head E., ASSIST: A Simple Simulator for State Transitions, http://watson2.cs.binghumpton.edu/~tools/ASSIST.html, (MSc thesis, University of New York, 1998).
9. Gramond E., Rodger S.H., Using JFLAP to Interact with Theorems in Automata Theory, In *13th SIGCSE Technical Symposium on Computer Science Education* (1999) pp. 336-340
10. Huber F., Schaetz B., Einert G., Consistent Graphical Specification of Distributed Systems, *Lecture Notes in Computer Science* **1313** (1997) pp.122-141

11. Henzinger T.H., Ho P-H., Wong-Toi H., A user guide to HyTech, *Lecture Notes in Computer Science* **1019** (1995) pp.41-71
12. Bengtsson J., Larsen K.G., Larsson F., Pettersson P., Yi W., UPPAAL: a tool suite for automatic verification of real-time systems, *Lecture Notes in Computer Science* **1066** (1996) pp.232-243
13. Nguyen T., FSM-based test Sequence Generator, http://www.site.ottawa.ca/~ural/tsg/into.html, (University of Ottawa, Ottawa, Ontario, Canada, 1994)
14. Chow T., Testing Software Design Modeled by Finite State Machines, *IEEE Transactions on Software Engineering* **4** (3) (1978) pp. 178-187

Appendix

The X-Machine functions of Figure 1 (vending machine) written in XMDL.

```
complete_money(x, (money, selection)) =
  if (x belongs coin_type and sufficient(x, money) then
  ((await_selection, add(x, money), no_drink),
   (add(x, money), selection))

get_selection(x, (money, selection)) =
  if x belongs selection_type then
  ((ready, money, no_drink), (money, x))

enter_pressed(enter, (money, selection)) =
  (await_coin, 0, drink), (0, selection))

continue_await_coin(x, (money, selection)) =
  if x belongs selection_type or x = enter then
  ((await_coin, money, no_drink), (money, selection))

continue_await_selection(x, (money, selection)) =
  if x belongs coin_type or x = enter then
  ((await_selection, money, no_drink), (money, selection))

continue_ready(x, (money, selection)) =
  if x belongs coin_type then
  ((ready, money, no_drink), (money, selection))
```