Developing Tools for Formal Methods

P.Kefalas, G.Eleftherakis and A.Sotiriadou

Computer Science Department, CITY Liberal Studies
Affiliated College of the University of Sheffield
13 Tsimiski Str., 54624 Thessaloniki, Greece
Email: {kefalas,eleftherakis,sotiriadou}@city.academic.gr

Abstract. Formal methods are based on rigorous mathematical notations, which aim to describe systems in the early stages of software development. Such formal descriptions are useful to precisely specify a system's data and/or control, and as a consequence to verify whether certain properties are true as well as to test whether the final product meets the initial description. All of these stages are crucial in the development process and researchers involved with formal methods claim that "correct" software can only be achieved through the use of formal methods. However, practitioners are skeptical against formal methods, arguing that, apart from the lack of training, the lack of tools to support formal development is the major drawback. In this paper, we present a framework for developing such tools for formal methods by describing the requirements and the steps that are necessary to meet this objective. The outline process will facilitate tool developers to identify the necessary steps to be taken, i.e. to define a practical core specification language, to compile such language to some form of executable code and use this code as means to aid further tool support for verification, model checking, testing etc. We record our experience on this proposed process by giving an example of a set of tools developed around a specific formal method, which we present as a case study.

1 Introduction

The extensive use of computer systems, as integrated parts of almost any engineered product has brought up two important issues, safety and reliability. The need for more robust, reliable and safe software and hardware and the fact that errors in various different stages of the production of a computer system can be responsible for the creation of non-reliable systems, has started to seriously concern the community 35 years ago [1]. Traditional computer system development methods proved to be inadequate to develop reliable and safe software. In the early days of this software crisis, it was a general belief that formal methods will become mainstream software development techniques in the future.

Over the last years, it is widely admitted that use of formal methods in software engineering is essential [2], while there are several cases proving the applicability of formal methods in industrial applications [3] showing very good results. However, many practitioners are still reluctant to adopt formal methods. The main reasons are: (a) other software engineering techniques successfully increased system quality by

improving processes and methodologies, and by using friendly tools, thus allowing rapid development of systems [4]; (b) the complex notations formal techniques use; (c) the lack of education in formal methods; (d) the lack of knowledge of the formal methods community about the needs of real systems; and (e) the lack of tools.

The claim that the lack of tools is one of the major reasons for the difficulties of incorporating formal methods in industry is a misconception and a myth [5]. During the past two decades a large number of software tools, which aim to support formal methods have been developed and are widely available [6, 7]. The actual reason is the lack of adequate, powerful, user friendly industrial strength tools that will aid the application of formal methods to industry and also allow them to be fully integrated with existing methods [2, 8, 9].

A formal method, to be accepted by the industry, should provide an intuitive formalism and be supported by easy-to-use tools that will enable the average practitioner to become familiar with it. Such tools should: (a) provide an intuitive way to describe the model of the system using the formal method with rigorous syntax and semantics; (b) incorporate techniques to facilitate the use of the formal model in as many stages of the development process as possible adding to the confidence for the creation of more reliable and "correct" systems; and (c) support an open architecture so that other tools that implement future additions can be integrated.

In this paper, we propose a framework for developing such tools for formal methods by describing the requirements and the steps that are necessary to meet this objective. The outline process, which is the main contribution of this work, will facilitate tool developers to identify the necessary steps to be taken, i.e. to define a practical core specification language, to compile such language to some form of executable code and use this code as means to aid further tool support for verification, model checking, testing etc. We also record our experience on this proposed process by giving an example of a set of tools developed around a specific formal method, namely X-Machines, which we present as a case study.

2 Tools for Formal Methods

There are many tools available that support formal methods, such as Z, OBJ, VDM, Finite State Machines (FSM) etc., which intend to increase the productivity and accuracy in all the phases of the formal development of systems. However, these tools vary in their capabilities, the extend to which they are used in industry and the extend to which they are able to support most of the stages of formal development.

CafeOBJ [10], a direct successor of OBJ, is a new generation algebraic specification and programming language. Cafe is an environment for systematic development of formal specifications based on algebraic specification techniques, which supports reasoning methods and semantic representations of problems. B-Toolkit [11], which supports the development of software systems using the B-Method, and the associated B-Tool, is a suite of fully integrated software tools. The Concurrency Workbench toolkit [12] is an integrated toolset for specification, simulation, verification, and implementation of concurrent systems such as communication protocols and process control systems. It aims to provide practical support to developers who are not famil-

iar with formal verification. FDR [13], based on the theory of CSP, allows the checking of many properties of finite state systems and the investigation of systems, which fail these checks. FDR has been applied successfully in a number of industrial applications. PVS [14] is a verification system that consists of the PVS specification language, which is based on classical, typed higher-order logic, integrated with support tools and a theorem prover. It has been used for significant applications in NASA and several aerospace companies. SMV [15] is a model checker for finite state systems, using the specification language CTL. SPIN [16] is a model checker, developed at Bell Labs, that can be used for the formal verification of distributed software and it uses a high level language to specify systems descriptions. SPIN can be used in three basic modes: as a simulator, as an exhaustive verifier, as proof approximation system that can validate very large protocol systems with maximal coverage of the state space. VDMTools [17] is available in two versions: one that supports the ISO VDM-SL notation and the other VDM++. The Toolbox includes a syntax checker, static semantic checker and a C++ code generator and a Java code generator. Z/EVES [18] consists of a graphical user interface that allows Z specifications to be edited. It supports the analysis of Z specifications in terms of syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. Finally, CADiZ [19], is a suite of tools designed for syntax and type checking, typesetting and proving properties of Z specifications.

3 A Framework for Developing Tool Support

3.1 Specification Language

A well-defined syntax and semantics is the first step towards the development of tools for formal methods. The specification language that most formal methods use is a mixture of diagrammatical and mathematical notation. Both possess the inherent problem that there exists a degree of freedom with which diagrams can be drawn or mathematical expressions can be written. As a consequence, whoever uses them feels rather free to express the specification of the system in any graphical editor or specific text editors, like LaTEX. The diversity and variants of notation for system specification may serve the actual purpose, which is a well-defined, unambiguous statement about the system specification to be further used by the designer but does not allow machine interpretation of the specification itself.

On the other hand, the development of tools requires a special notation, which can be machine-interpretable. This necessitates the definition of rules in which specifications are written, and may possibly introduce syntax constraints, which in turn may reduce the degree of freedom in writing mathematical expressions. This core language should be well-defined in terms of syntax and semantics and should act as a standard on which tool development should be based. The advantage of having such a standard interlingua for a formal method is that specifications written in this language can be used by other system developers without modification. However, the definition of the core language requires certain properties to be taken into account.

Firstly, the core language should have the same expressivity as the mathematical or diagrammatical notation. Although, it may be rather difficult to establish a one-to-one mapping of the constructs of both sides, the specifier should nevertheless be able to encode the same classes of system specifications as in the mathematical or diagrammatical notation. Secondly, the core language should meet all criteria of language evaluation [20], i.e. it should be readable, writable, reliable and cost effective. Properties like overall simplicity, orthogonality, and inclusion of control statements, data types, type checking etc. should be preserved. Finally, the language needs to be rather declarative (as opposed to procedural), since it matches with the philosophy of formal methods, which specify "what" a system is and not "how" the system works. Although it may be difficult to preserve all the above mentioned characteristics, every effort should be made to create a formal notation which will accommodate as less tradeoffs as possible.

3.2 Parsing and Completeness Checking

Following the complete definition of the core language grammar, the implementation of a syntax analyser facilitates any further tool development. The parser is an integral part of any tool that is built for the formal method in question and acts as an interface component of the actual core language specification and the processing component, which gives the tool its desired functionality. A parser may be constructed using already existing tools like LEX and its variants, JavaCC etc. Apart from being able to check whether the specification is syntactically correct, the parser may be augmented with a functionality to check whether the specification is valid. This means that errors in type definitions and variable usage or errors arising from incomplete description can be identified. Such functionality is extremely important since the specifier realizes that it is often the case that although the specification created by hand on paper looked fine, it actually contains several errors and omissions, which could not have been identified unless machine processing of the specification was possible.

3.3 Compiling Core Language to Executable Code

Assuming that the specification encoded in the core language is correct and complete, the tool development process should proceed with the compilation of the core language specification to some kind of executable code. The tool developer is presented with several options about the target language. The greater the distance of the original specification language to the target language, the more difficult this task becomes. By distance, we mean the lack of these two languages to share common types or constructs, which will prevent a close matching between the two. Whichever the case is, this step is a rather complex and needs careful consideration.

One of the important aspects that special care should be taken is the equivalence of the two languages. One needs to prove in advance that the transformation of the formal specification language to executable language is correct, i.e. every step taken to transform a specification statement to a number of declarations or executable commands is sound. This can be achieved by a number of methods, e.g. bisimulation.

It is often the case that a completely automatic compilation is not possible. Intervention of the user may be required to resolve ambiguities, which result from the distance of the source and target language. This is not necessarily bad, but it may result to a slightly altered meaning of the one intended.

It is implied that in case of fully automatic transformation, the target code may not be optimized, because the executable code may include commands or statements that would not appear if someone has written the code manually from scratch.

3.4 Runable Specifications

Apart from the role that the executable code produced in the previous step can play in the development of other tools, it can also act as a prototype implementation. Although, the executable code may be far from being optimal, it can be used to check whether the intended functionality described in the specification is preserved. The specifier of the system can quickly identify what might have gone wrong with the specification of the system by experimenting with the prototype. It is often the case, that a specification, although syntactically and semantically correct, does not meet the specification of the system. This is hard to reveal from a paper specification, which most of the times intuitively seems to describe whatever was intended to be described.

The choices presented in this phase are many, varying from direct execution of the compiled code, to interpretation or animation of this code. The performance of the latter may be again not optimal but this is not a requirement of the prototype implementation anyway.

3.5 Model Checking

Formal verification is the process of using formal methods to prove the existence of user required properties in the proposed model of the system, i.e. to prove that the model is “correct”. Model Checking [21] is a formal verification technique, which is based on the exhaustive exploration of a given state space trying to determine whether a given property, expressed as a temporal logic formula, is satisfied by a system. It is a widely used technique over the last years mainly because it has recently achieved significant results both for hardware and software systems. Efficient algorithms have been devised and tools, model checkers, have been developed that are able to verify properties of realistic complex systems. 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.

This is by labeling each state of the model with a set of subformulas of the temporal logic formula that expresses the property, which are true in that state. Initially, this set is composed only by the atomic propositions that are true in each state. The algorithm proceeds in stages; during *i*-th stage the subformulas with *i-1* nested temporal operators are processed and added to the states, which are true. Finally, the property holds in the model iff the formula expressing the property belongs to the set of properties that are true in the initial state. Using the runable specification of the tool it is feasible to implement the model checking algorithm.

3.6 Testing

The principle purpose of testing is to detect and remove errors in the implementation of a system under development [22]. Testing will always be an integral part of the development cycle because it improves confidence on using the final product. Testing and verification should complement each other with testing providing confidence in the correctness of the assumptions made in verification. That is why a tool should be able to support both.

It is possible to extract the test cases from the formal description by implementing an algorithm as the formal testing strategy describes. Then, it is possible to animate the model through the prototype with all the test cases and derive the expected by the model sequences of output. By feeding these test cases to the actual implementation, it is possible to compare the sequences of outputs with the sequences of outputs from the model in order to prove that the implementation is equivalent with the model.

4 Case Study: X-machines Tool Support

In this section we record our experience with tool development around a formal method, namely X-machines. An X-machine is a general computational machine [23], that resembles a Finite State Machine (FSM) but with two significant differences: (a) there is an underlying data set attached to the machine, and (b) the transitions are not labeled with simple inputs but with functions that operate on inputs and data set values. An interesting class of X-machines is the stream X-machines that can model non-trivial data structures as a typed memory tuple. Stream X-machines employ a diagrammatic approach of modeling the control by extending the expressive power of the FSM [24]. They are capable of modeling both the data and the control by integrating methods, which describe each of these aspects in the most appropriate way [25, 26, 27, 28]. Transitions between states are performed through the application of functions, which model the processing of the data that are held in the memory. Functions receive input symbols and memory values, and produce output while modifying the memory values. The machine, depending on the current state and the current values of the memory, consumes an input symbol from the input stream and 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 [29] is the construct $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, where:

- Σ, Γ is the input and output finite alphabet respectively, i.e. two sets of symbols,
- Q is the finite set of states,
- M is the (possibly) infinite set called memory, i.e. an n-tuple of typed symbols,
- 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, $F: Q\times\Phi\rightarrow Q$, and
- q_0 and m_0 are the initial state and memory respectively.

Consider the X-Machine, which models a stack with limited capacity (Fig.1).

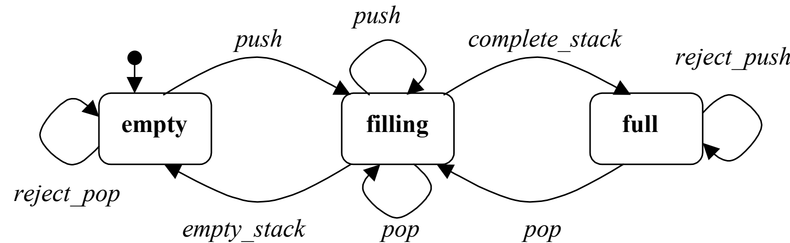

Fig. 1. The X-Machine transition diagram of a stack specification

The basic type defined in the stack is `[ELEMENT]` which implies that the stack can accept any type of elements. The set `Q={empty, filling, full}`, represent the states in which the stack is empty, neither empty nor full, and full respectively. The state q_0`=empty` is the initial state. Other types need to be defined for the input and output sequences are: `Action ::= ↓ | ↑`, indicating that an element should be pushed or poped, and `Messages ::= "Element Pushed" | "Element Popped" | "Stack Full"` etc. Consequently, the input set is then defined as: `Σ =Action×(ELEMENT∪{ε})` and the output set as: `Γ =Message×(ELEMENT∪{ε})` where ε denotes no element. The memory should contain the sequence of elements in the stack as well as the capacity of the stack: `M=seq ELEMENT × N`, while the initial memory is, if capacity is 10 elements: m_0 `= (nil, 10)`.

Functions `push` and `pop` perform the usual operations, and are complemented by `complete_stack` and `empty_stack` when the last element is pushed or popped respectively. Functions `reject_push` and `reject_pop` are applicable when the stack is full or empty and a ↓ or ↑ action is attempted respectively.

4.1 XMDL

X-machine modelling is based on a mathematical notation, which, however, implies a certain degree of freedom, especially as far as the definition of functions are concerned. In order to make the approach practical and suitable for the development of tools around X-machines, a standard notation is devised and its semantics fully defined [30]. Our aim was to use this notation, namely X-machine Definition Language (XMDL), as an interchange language [31] between developers who could share models written in XMDL for different purposes. To avoid complex mathematical notation, the language symbols are completely defined in ASCII.

Briefly, a XMDL based model is a list of definitions corresponding to the construct tuple of the X-machine definition. The language also provides syntax for (a) use of built-in types such as integers, Booleans, sets, sequences, bags, etc., (b) use of operations on these types, such as arithmetic, Boolean, set operations etc., (c) definition of new types, and (d) definition of functions and the conditions under which they are applicable.

In XMDL, the functions take two parameter tuples, i.e. an input symbol and a memory value, and return two new parameter tuples, i.e. an output and a new memory value. A function may be applicable under conditions (if-then) or unconditionally.

Variables are denoted by a preceding "?". The informative "where" in combination with the operator "<-" is used to describe operations on memory values. For example the X-Machine for the stack described above can be written in XMDL as follows:

```
#model stack.
#basic_types = [ELEMENT].
#type capacity = Natural.
#type action = {down_arrow, up_arrow}.
#type messages = {ElementPushed, ElementPoped, StackFull, StackEmpty,
RejectStackFull, RejectStackEmpty}.
#type stack = sequence_of ELEMENT.
#type allelements = ELEMENT union {no_element}.
#states = {empty, filling, full}.
#memory (stack, capacity).
#init_state {empty}.
#init_memory (nil, 10).
#input (action, allelements).
#output (messages, allelements).
#fun push ((down_arrow,?element), (?stack, ?c))=
      if ?number < ?c then
      ((ElementPushed,?element),(<?element::?stack>, ?c))
where  ?number <- cardinality <?element :: ?stack>.
#fun pop ((up_arrow, no_element), (<?element :: ?stack>, ?c))=
      if ?stack \= nil then ((ElementPoped, ?element),(?stack, ?c)).
. . .
#transition (empty, reject_pop) = empty.
#transition (empty, push) = filling.
#transition (filling, push) = filling.
. . .
```

A number specifiers who used XMDL have evaluated it as readable and writable. The XMDL language is also: (a) orthogonal, since there are rather few primitive structures that can be combined in rather few ways; (b) non-positional, declarative and mark-up; (c) strongly typed, since it inludes type checking, set and function checking, as well as exception handling, and (d) close to the mathematical notation.

4.2 Parsing with Definite Clause Grammars

The parser of XMDL was built using Definite Clause Grammars (DCG) notation, which is integrated in Prolog language (Fig.2). Having defined the grammar of XMDL, DCG clauses were straightforward to code. For example, the following Prolog DCG code represents the parser segment that deals with the definition of states of the X-machine:

```
states(StateList) --> ['#states'], ['='], set(StateList).
set(Empty)-->  emptyset(Empty).
set([]) --> ['{'], ['}'].
set(L) --> ['{'], set_elements(L), ['}'].
set_elements([Elem])--> one_set_element(Elem).
set_elements([E|R])-->one_set_element(E), [','], set_elements(R).
one_set_element(I) --> identifier(I).
```

where `identifier` is defined accordingly. In DCG, the symbol "`-->`" reads as "consists of" and the terminal symbols are enclosed in square brackets "`[ ]`".

Apart from the syntax errors, which may occur in a XMDL code, there might be logical errors or omissions which are output as warnings, e.g. "State defined is not used in transitions", "The X-Machine is non-deterministic", "User types are not defined" etc., or errors, such as "The initial memory tupple arity is different from memory type", "Function in transition is not defined", "Cannot infer the type of variable", "Memory parameter inconsistent with memory" etc.

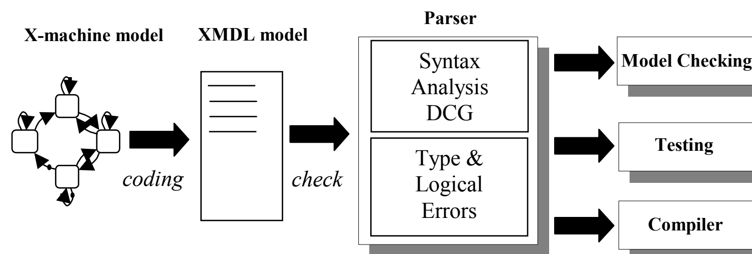

Fig. 2. The Parser of XMDL code

4.3 XMDL to Prolog

Given a specification in XMDL code, we have developed the rules under which the specification is translated into the equivalent Prolog code (Fig.3). The issue under consideration is devising the equivalent representation of:

- types, including basic types and built-in types;
- various types of sets, like ordinary sets, sequences, bags, as well as tuples;
- type checking of input and output parameters, as well as memory;
- functions, including condition expressions and function bodies;
- external functions, type checking of their arguments and external file reference.

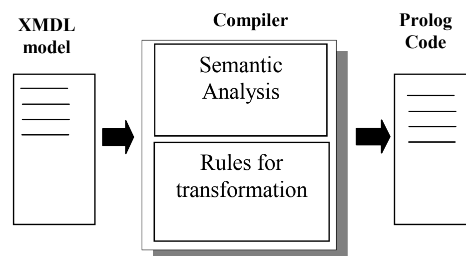

Fig. 3. The XMDL to Prolog Compiler

Due to the inability of Prolog to perform type checking, the types should be declared explicitly as Prolog code. Types can be built-in, user defined (a set of values), a tuple of types (Cartesian product), etc. For each different form, there should be a different translation. Some examples are given in Table 1.

Prolog Code	Explanation
`type(X, element):-!.`	Since `[ELEMENT]` is a basic type, anything can be of type `ELEMENT`.
`type(X, capacity):-` `integer(X), X>0,!.`	Since `Capacity` is of built-in type N, something is of type `Capacity` only if it is an integer greater than zero.
`type(X, action):-` `member(X,[down_arrow,up_arrow]),!.`	Since type `Action` is explicitly defined, something can be of type `Action` if it belongs to the set of given values.

`type(S, stack):-`
`forall(member(X,S),`
`type(X,element)),!.`
	Since `Stack` is a sequence of `ELEMENT`, any sequence is of type stack, only if all its elements are of type `ELEMENT`.

`type(X, allelements):-`
`type(X, element);`
`type(X, [no_element]),!.`
`type(X, [no_element]):-`
`member(X, [no_element]),!.`
	Since `Allelements` is declared as a union of `ELEMENT` and the set `{no_element}`, something is of type `Allelements` if it is of type `ELEMENT` or it is a member of the set `{no_element}`.

Table 1. The equivalent Prolog code for type definitions.

Prolog lacks the ability to represent different types of sets. The only structure provided is the list, which corresponds to a sequence. Sets, sequences and bags can be represented as lists, as long it is taken special care about the following invariants:

- a set does not have duplicate elements while a bag may have some;
- set operations and bag operations take into account the above characteristic, e.g. they do not insert an element in a set if it exists;
- Boolean operations take into account the fact that sets and bags do not have order, e.g. the bags `[a,b,b,a]` and `[a,a,b,b]`, as well as the sets `[a,b,c]` and `[b,a,c]` are equal, while the corresponding sequences are not equal.

Tuples can be either represented in the usual way, i.e. a list of elements enclosed in round brackets `( x1, x2, …, x3)`, or as a list of elements, e.g. `[x1,x2,…x3]`, as long as the list length is not modified. For implementation simplicity the latter is chosen in this approach. Thus for example the memory, input and output tuples are implemented as shown in Table 2.

The type Φ of X-Machine functions φ: $\Sigma \times M \rightarrow \Gamma \times M$ should be mapped to equivalent Prolog predicates. According to the XMDL definition, any $f \in \Phi$, is of the following format:

```
#fun f ((I1,…, In), (M1,…, Mm) )=
    if FUNCTION CONDITION then ((O1,…, Ok), (M1',…, Mm') )
where FUNCTION BODY.
```

The equivalent Prolog predicate should have the following format:

```
f([I1,…, In], [M1,…, Mm], [O1,…, Ok], [NM1,…, NMm]):-
    <PREDICATE TYPE CHECK OF INPUT>,<PREDICATE TYPE CHECK FOR MEMORY>,
    <PREDICATE EQUIVALENT FUNCTION BODY>,
    <PREDICATE EQUIVALENT FUNCTION CONDITION>,
    <PREDICATE TYPE CHECK OF OUTPUT>,<PREDICATE TYPE CHECK OF MEMORY>.
```

<table>
<tr><th>Prolog Code</th><th>Explanation</th></tr>
<tr><td><pre>memory([M1, M2]):-
     type(M1, stack),
     type(M2, capacity),!.
memory(_) :-
write('WARNING:Invalid Memory!'),nl,!.</pre></td><td>The memory consists of a tuple; the first element should be of type <code>stack</code> and the second element of type <code>capacity</code>. All other memory tuples are considered as invalid.</td></tr>
<tr><td><pre>input([I1, I2]):-
     type(I1, action),
     type(I2, allelements),!.
input(_) :-
write('WARNING:Invalid Input!'),nl,!.</pre></td><td>The input consists of a tuple; the first element should be of type <code>Action</code> and the second element of type <code>Allelements</code>. All other input tuples are considered as invalid.</td></tr>
<tr><td><pre>output([O1, O2]):-
     type(O1, messages),
     type(O2, allelements),!.
output(_) :-
write('WARNING:Invalid Output!'),nl,!.</pre></td><td>The ouput consists of a tuple; the first element should be of type <code>Messages</code> and the second element of type <code>Allelements</code>. All other output tuples are considered as invalid.</td></tr>
</table>

Table 2. The equivalent Prolog code for memory, input, and output definitions.

The arguments of the predicate are clearly the input, memory, output, new memory tuple respectively. The type checks are calls to `memory/1, input/1` and `output/1` predicates shown above. The function conditions and function body are automatically translated into the equivalent Prolog code, possibly containing calls to built-in operations developed separately in a library. An example of a function translation is given in Table 3.

```
push( [down_arrow,Element],[Stack,C],
      [elementpushed,Element],[[Element|Stack], C]):-
         input([down_arrow, Element]), memory([Stack, C]),
         cardinality([Element|Stack], Number),
         Number<C,
         output([elementpushed,Element]), memory([[Element|Stack],C]).
```

Table 3. The equivalent, automatically produced, Prolog code of a function definition.

The rest declarations of XMDL code are translated in a quite straightforward way into Prolog. Examples are given in Table 4.

<table>
<tr><th>Prolog Code</th><th>Explanation</th></tr>
<tr><td><code>model(stack).</code></td><td>The model specified is called <code>stack</code>.</td></tr>
<tr><td><code>init_state([empty]).</code></td><td>The initial state is <code>Empty</code>.</td></tr>
<tr><td><code>init_memory([[], 10]).</code></td><td>The initial memory is <code>(nil, 10)</code></td></tr>
<tr><td><pre>transition(empty, reject_pop, empty).
transition(filling, push, filling).
. . .</pre></td><td>The transitions are represented as facts, with state, function, next state as ground arguments.</td></tr>
</table>

Table 4. The equivalent Prolog code for other XMDL declarations.

4.4 Animation of Prolog Code

The animation of the original specification can be achieved by putting together the resulting Prolog equivalent code, the file in which external functions are defined, the actual animator and the library of built-in operations (Fig. 4). The Prolog code resulted from the translation may not be the most efficient, but one has to bear in mind that this is not the real implementation and in addition it is automatically generated through high level transformations. The library of built-in operations contains operations on sets, bags and sequences that are assumed as built-in in XMDL. As previously mentioned, the predicates should be implemented in such a way that they do not affect the characteristics of individual set types, even though all sets are treated as lists. The animator is a Prolog program, which implements an algorithm that simulates the computation of an X-machine. Starting from the initial state and initial memory values, and for as long as the input stream is not empty, it reads an input, which triggers a transition function. As an effect an output is produced and the memory values are altered. The computation continues with the new state. A sample output from the animator is the following:

```
State : empty - Input ? [down_arrow,1].
- Applied Function : push
- Output: [elementpushed, 1] Memory : [[1], 10]
State : filling - Input ? [up_arrow,no_element].
- Applied Function : empty_stack
- Output: [stackempty, no_element] Memory : [[], 10]
State : empty - Input ? [up_arrow,no_element].
- Applied Function : reject_pop
- Output: [rejectstackempty, no_element] Memory : [[], 10]
State : empty - . . .
```

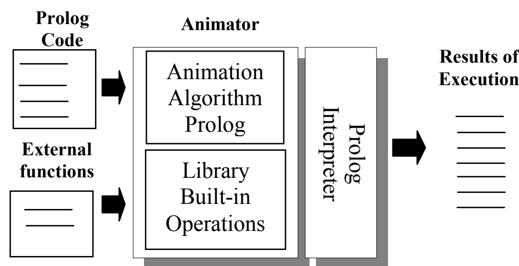

Fig. 4. The executable code animator

4.5 XmCTL Model Checking

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 apply model checking to X-machines, temporal CTL* formulae with operators like F (even-

tually), G (always), A (for all paths), E (there exists a path), etc. [32] are extended to include two memory quantifier operators:

- M_x, for all memory instances, and
- m_x, there exist memory instances

The new logic, XmCTL, is fully defined in [33]. Interesting properties about a system may be revealed by XmCTL formulas, for example `AG M_x [#mem1 ≤ mem2]`, meaning that for all paths and for all states in these paths the number of elements in the stack (residing in memory position `mem1`) does not exceed the capacity of the stack (`mem2`).

Again, the XmCTL language is defined using DCG. The resulting formula together with the executable specification described above and the implementation of model checking algorithms [33] can determine whether a property is true or false (Fig.5).

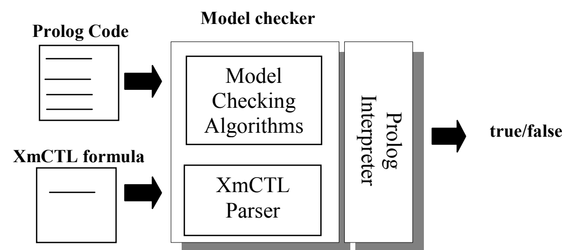

Fig. 5. The model checker

4.6 Automatic Generation of Test Cases

For X-machines, there exists a testing method, which is proved that it finds all faults in the implementation [34]. The method works based on certain assumptions, and design-for-test conditions, i.e. output distinguishability and completeness and can produce a complete test set of input sequences. In order to check whether the design-for-test conditions are met, the executable specification described above is used. It can then generate the test cases, which will be eventually used to test the actual implementation of the system [35].

4.7 Other Tools

There are also other tools that have been constructed for X-machines. Indicatively, we can mention the compiler of X-machines to Z language, which is performed by transforming XMDL code to Z schemata [36], and the graphical environment developed in order to allow a specifier to draw the transition diagram of an X-machine, and automatically output the XMDL code [37]. Finally, XMDL has been extended to accommodate the need of modeling communicating systems, through communicating X-machines [25].

5 Conclusions

We have presented a framework for developing tools for formal methods and we have used it for building various tools for X-machines. This process is based on a core specification language, which is close to the mathematical notation used for the definition of X-machines, and acted as an interlingua between tools. A parser has been constructed that also helped in identifying omission and logical errors in the formal paper-written specification. The animator developed assists the specifier to work on early stages at a prototype implementation, thus identifying inadequacies of the original specification. Finally, the model checker is used to prove whether certain desired properties of the system are true and the tester to generate a complete test set for the implementation. The tool has been successful in facilitating formal development of complex real world problems [38]. Future work will include the integration of more tools in the system, as the developing theory of X-machines imposes, and will also finalize the graphical interface of the integrated system.

References

[1] P. Naur, B. Randell (eds.), Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, Scientific Affairs Division, NATO, (1969)

[2] M.D. Fraser, K. Kumar, V.K. Vaishnavi, Strategies for Incorporating Formal Specifications in Software Development, *Communications of the ACM*, 37(10):74-86, (1994)

[3] D. Craigen, S. Gerhart, T.J. Ralston, Formal Methods Reality Check: Industrial Usage, *IEEE Tranactions on Software Engineering*, 21(2):90-98 (1995)

[4] I. Sommerville, *Software Engineering*, 6th edition, Addison-Wesley, (2001)

[5] J. Bowen, M. Hinchey, Seven More Myths of Formal Methods, *IEEE Software*, 12(4):34-41 (1995)

[6] Formal Methods Europe http://www.fmeurope.org

[7] Formal Methods Virtual Library http://www.afm.sbu.ac.uk/

[8] M. Hinchey, J. Bowen, To Formalize or not to Formalize?, *IEEE Computer*, 18:18-19 (1996)

[9] C.M. Holloway, R.W. Butler, Impediments to Industrial Use of Formal Methods, *IEEE Computer,* 24(9):25-26 (1996)

[10] R. Diaconescu, K. Futatsugi, *CafeOBJ Report: the Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, AMAST Series in Computing 6, World Scientific, (1998)

[11] J. Wordsworth, *Software Engineering with B*, Addison Wesley, (1996)

[12] R. Cleaveland, P.M. Lewis, S.A. Smolka, O. Sokolsky, The Concurrency Factory: a Development Environment for Concurrent Systems. *Proceedings Conference Computer-Aided Verification (CAV),* Springer LNCS Vol.1102, (1996) 398-401

[13] Formal Systems (Europe) Ltd. Failures-Divergence Refinement- FDR2 User Manual, 2000. Available via http://www.fsel.com/fdr2_manual.html

[14] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M. Srivas, *PVS: Combining Specification, Proof Checking, and Model Checking*, Springer LNCS Vol.1102, (1996) 411-414

[15] K.L. McMillan, *Symbolic Model Checking*, Kluwer, (1993)

[16] G.J. Holzmann, The Model Checker Spin, *IEEE Transactions on Software Engineering*, 23(5):279-295 (1997)

[17] R. Elmstrom, P.G. Larsen, P.B. Lassen, The IFAD VDN_SL Toolbox: a Practical Approach to Formal Specifications. *ACM SIGPLAN Notices*, 29(9), (1994)
[18] D. Craigen, I. Meisels, M. Saaltink, Analysing Z Specifications with Z/EVES. In: J.P. Bowen, M.G. Hinchey (eds.), *Industrial-Strength Formal Methods in Practice*, FACIT series Springer, (1999)
[19] I. Toyn, J.A. McDermid, CADiZ: an Architecture for Z Tools and its Implementation, *Software - Practice and Experience* 25(3):305-330 (1995)
[20] R.W.Sebesta, *Concepts of Programming Languages*, 5th edition, Addison-Wesley, (2002)
[21] E.M. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, Cambridge, MA, (1999)
[22] G. Myers, *The Art of Software Testing*, Wiley, (1979)
[23] S. Eilenberg, *Automata Machines and Languages*, Vol.A, Academic Press, (1974)
[24] E. Kehris, G. Eleftherakis, P. Kefalas, Using X-machines to Model and Test Discrete Event Simulation Programs, In: N. Mastorakis (ed.), *Systems and Control: Theory and Applications*, World Scientific, (2000) 163-168
[25] P. Kefalas, G. Eleftherakis, E. Kehris, *Communicating X-Machines: a Practical Approach for Formal and Modular Specification of Large Systems*, Information and Software Technology, 45(5):269-280 (2003)
[26] P. Kefalas, Formal Modelling of Reactive Agents as an Aggregation of Simple Behaviours, Springer LNAI Vol.2308 (I.P. Vlahavas, C.D. Spyropoulos, eds.), (2002) 461-472
[27] P. Kefalas, M. Holcombe, G. Eleftherakis, M. Gheorge, A Formal Method for the Development of Agent Based Systems, In: *Intelligent Agent Software Engineering*, V. Plekhanova (ed.), Idea Group, (2003) 68-98
[28] P. Kefalas, G. Eleftherakis, M. Holcombe, M. Gheorghe, Simulation and Verification of P Systems through Communicating X-Machines, *BioSystems*, (2003)
[29] M. Holcombe, F. Ipate, *Correct Systems: Building a Business Process Solution*, Springer (1998)
[30] P. Kefalas, XMDL User Manual, Version 1.6, Technical Report TR-CS07/00, Department of Computer Science, CITY College, (2000)
[31] P. Kefalas, E. Kapeti, A Design Language and Tool for X-machines Specification, In: D.I. Fotiadis, S.D. Nikolopoulos (eds.), *Advances in Informatics*, World Scientific Publishing, (2000) 134-145
[32] E.A. Emerson, J.Y. Halpern, Sometimes and not never Revisited: on Branching Time versus Linear Time, *Journal of the ACM*, 33:151-178 (1986)
[33] G. Eleftherakis, P. Kefalas, A. Sotiriadou, XmCTL: Extending Temporal Logic to Facilitate Formal Verification of X-Machines, *Matematica-Informatica*, Analele Universitatii Bucharest, 50, (2002) 79-95
[34] F. Ipate, M. Holcombe, Specification and Testing Using Generalised Machines: a Presentation and a Case Study, *Software Testing, Verification and Reliability*, 8,:61-81 (1998)
[35] M. Holcombe, X-machines as a Basis for Dynamic System Specification, *Software Engineering Journal*, 3(2):69-76 (1988)
[36] P. Kefalas, A. Sotiriadou, A Compiler that Transforms X-machines Specification to Z, Technical Report TR-CS06/00, Department of Computer Science, CITY College, (2000)
[37] N. Walkinshaw, Visual X-Machine Description Language (VXMDL), Department of Computer Science, University of Sheffield (2002)
[38] G. Eleftherakis, P. Kefalas, A. Sotiriadou, Formal Modelling and Verification of reactive Agents for Intelligent Control, *Proceedings 12th Intelligent Systems Application to Power Systems*, (2003)