Avestan: A Declarative Modeling Language Based on SMT-LIB

Amirhossein Vakili and Nancy A. Day
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{*avakili,nday*}*@uwaterloo.ca*

***Abstract*—Avestan is a declarative modelling language compatible with SMT-LIB. SMT-LIB is an standard input language that is supported by the state-of-the-art satisfiability modulo theory solvers (SMT solvers). The recent advances in SMT solvers have introduced them as efficient analysis tools; as a result, they are becoming more popular in the verification and certification of digital products. SMT-LIB was designed to be machine readable rather than human readable. In this paper, we present Avestan, a declarative modelling language that is intended to be analyzed by SMT solvers and readable by humans. An Avestan model is translated to an SMT-LIB model so that it can be analyzed by different SMT solvers. Avestan has relational constructs that are heavily inspired by Alloy; we added these constructs to increase the readability of an Avestan model.**

***Keywords*-Alloy; Declarative Model; First-Order Logic; Relational Construct; SMT-LIB; SMT solver**

I. Introduction

In this paper, we present the declarative modelling language, Avestan[1], which is intended to create a readable modelling language fully compatible with the SMT-LIB language [1]. SMT-LIB was designed as a common interface language for SMT solvers, such as Z3 [2] and CVC3 [3]. However, SMT-LIB would not be the language of choice for a modeller. An SMT-LIB model consists of a set of S-expressions; the excessive use of parentheses in S-expressions, and their prefix format, makes SMT-LIB models machine readable rather than human readable. However, a general-purpose declarative modelling language that exactly matches the expressive power of SMT-LIB is lacking. Alloy [4] comes close, but is not fully compatible with SMT-LIB because it includes non-first order features (e.g., transitive closure). Avestan has been heavily influenced by Alloy and actually attempts to be more readable than Alloy for some problems because it is less influenced by an object-oriented focus.

We describe the key modelling features of Avestan through examples. The contribution of this work is mainly a utilitarian one rather than a technical contribution. With the trend that more and more software engineering problems can be described as SMT problems and solved by SMT solvers, many may find Avestan a very useful language. Our translator from Avestan to SMT-LIB is available on the web [5].

[1]It is named after the ancient Persian language, Avestan.

II. Features of Avestan

The syntax of Avestan is in ASCII form. Figure 2 is an Avestan model of binary trees. A block indicated by `/*...*/` or `//` to the end of the line are comments. We use this example to present the different features of Avestan.

In order to model a binary tree, `BTree`, we use two functions, `left_child` and `right_child`, to represent the left and right subtrees of a tree. A special binary tree is used to represent the `null` tree. The `size` of a binary tree is the number of nodes that occur in the tree. We use the function `leaf_number` to represent the number of leafs that a tree has; e.g., the size of the binary tree of Figure 1 is 5 and it has 3 leafs.

In general, an Avestan model consists of three parts:

1) Declarations: the declaration section declares the user-defined types that are used in modelling, and the entities that are present in the model.
2) Constraints: a set of first-order logic formulas that the entities of the model must satisfy.
3) Analysis command: what kind of analysis the modeller is interested in applying to the Avestan model.

For example, Lines 1-10 of Figure 2 constitute the declaration section, which declares the user-defined type, `BTree`, along with some functions and relations. Lines 19-41 are constraints that these entities must satisfy, and Lines 47-49 express the analysis command. In the following subsections, each line of the Avestan model of Figure 2 is explained in detail.

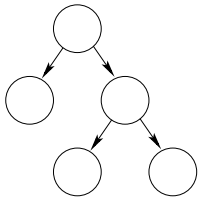

Figure 1. A binary tree with 3 leafs and 5 nodes

```
1 // Declarations:
2 sort BTree.   // Declaring a new type.
3
4 null: -> BTree. // Declaring a constant.
5
6 left_child, right_child: BTree -> BTree. //Declaring 2 functions of type BTree->BTree
7 size, leaf_number: BTree -> Int. //Declaring 2 functions of type BTree->BTree
8
9 child: BTree * BTree. // Declaring a binary relation over BTree.
10 proper_descendant: BTree * 2. // This is another way to declare a binary relation.
11
12 // is_leaf  is a macro.
13 macro is_leaf(t){
14   and[t != null, right_child(t) = null, left_child(t) = null]
15 }
16
17 // Constraints:
18 // The keyword definition is used to define the relation child.
19 definition child(t, c: BTree){
20   t != c.
21   left_child(t) = c or right_child(t) = c.
22 }
23
24 // The function size is defined by using the if-then-else-fi construct.
25 definition size(t: BTree): Int{
26   if t = null then 0
27   else size(left_child(t)) + size(right_child(t)) + 1 fi
28 }
29
30 // The keyword suppose is used to add constraints to an Avestan model.
31 suppose left_child(null) = null and right_child(null) = null.
32 suppose leaf_number(null) = 0.
33 suppose forall t:BTree| size(t) = 0 implies t = null.
34 suppose forall t:BTree| is_leaf(t) implies leaf_number(t) = 1.
35 suppose forall t:BTree|
36   is_leaf(t) or leaf_number(t) = leaf_number(left_child(t)) + leaf_number(right_child(t)).
37 suppose proper_descendant == expand(child, 4).
38 suppose not (exists t: BTree| t != null and proper_descendant(t, t)).
39
40 // This cardinality constraint states that there are only 4 binary trees in this model.
41 suppose |BTree| = 4.
42
43 // Analysis command:
44 /* This entailment block asks the question whether
45  * size(t) = 2 * leaf_number(t) - 1 holds for any nonempty tree.
46  */
47 entails ? {
48   forall t:BTree| t != null implies size(t) = 2 * leaf_number(t) - 1.
49 }
```

Figure 2. Modelling binary trees in Avestan

A. Type System

The keyword **sort** is used to define new types in an Avestan model. In Figure 2, Line 2, `BTree` is defined as a new type.

Besides user defined types, Avestan has three built-in types: **Bool**, **Int**, and **Real**. Conventional arithmetic and logical operators, such as multiplication, $*$, division, /, addition, +, subtraction, −, **and**, **or**, **not**, etc., are built-in. These types are included in Avestan for two reasons: 1) they are commonly used in modelling, 2) SMT solvers support the analysis of these types. Unlike in Alloy, the analysis support provided by SMT solvers for integers and real numbers can lead to unbounded analysis of Avestan models that use these types.

B. Entities

An entity in an Avestan model is either a *constant*, a *relation* (set), or a *function*. Constants and functions are found in SMT-LIB, but viewing relations as sets and having set operations for them in Avestan was influenced by Alloy.

Constants: a constant is a single element of a set. For example, in Figure 2, Line 4, `null` is declared as a single element of the set `BTree`. The following is the general form for declaring the constant `c` of type `TYPE` in Avestan:

```
c: -> TYPE.
```

`TYPE` can be a user-defined type or a built-in type.

In general, two constants of the same type can have equal values. The keyword **`distinct`** can be used to impose distinctness between constants:

```
distinct c1, c2: Real.
```

Unlike Alloy, which implements constants as singleton sets, Avestan, similar to SMT-LIB, implements constants as constant functions.

Functions: a function is a mapping from the elements of one set to another. In Figure 2, Line 6, `left_child` and `right_child` are functions from `BTree` to `BTree`. Line 7 declares that `size` and `leaf_number` are functions that map each binary tree to an integer.

The general form for declaring the *total*[2] function `f`, from the type `Domain` to the type `Range` is as follows:

```
f: Domain -> Range.
```

`Range` is a user-defined type or a built-in type; `Domain` is a user-defined type, a built-in type, or a sequence of user-defined and built-in types separated by `*`; e.g., the following defines the function `insert_int` as a function that takes a `List` and an **`Int`**eger and outputs another `List`:

```
insert_int: List * Int -> List.
```

Avestan does not support partial functions directly. A partial function must be defined as a relation that has functional properties.

Relations: in Avestan, a relation is a set of elements of the same type. The type is either a user-defined type, a built-in type, or a sequence of user-defined and built-in types separated by `*`; e.g., Line 9 of Figure 2 declares `child` as a relation over pairs of binary trees, `BTree * BTree`. The following is the general form for declaring the relation `r` of type `TYPE`:

```
r: TYPE.
```

Avestan has a set of built-in relational operators that allow a user to define new relations based on the existing relations in the model: union, `++`, intersection, `&&`, difference, `--`, Cartesian product, `**`, and **`join`**. The **`join`** operator of Avestan is much more flexible than Alloy's; it takes two relations `r1, r2`, and a positive integer `n` as input and the result is a relation that is equal to the join of `r1, r2` based on the last `n` columns of `r1` and the first `n` columns of `r2`. For example, if `r1` is of type `A * B * C`, and `r2` is of type `B * C * D`, **`join`**`(r1,r2,2)` satisfies the following property:

$$(a,d) \in \textbf{join}(\texttt{r1},\texttt{r2},2) \Leftrightarrow \exists b \in \texttt{B},\ \exists c \in \texttt{C} \mid \texttt{r1}(a,b,c) \wedge \texttt{r2}(b,c,d)$$

In Avestan, the type of a relation can be followed by "`* 2`". This declaration is equivalent to repeating the `TYPE` twice; e.g., the following two declarations are semantically equivalent:

```
r:  Int*Real*List*2
r:  Int*Real*List*Int*Real*List
```

A relation that can be declared using the "`* 2`" is called an *expandable* relation. Avestan has an operator that is only applicable to expandable relations, **`expand`**. This operator can be used as an approximation for the transitive-closure operator, which is not expressible in first-order logic. If `R` is an expandable relation with arity $2 \times \texttt{m}$, the **`expand`** operator satisfies the following two recursive equations:

```
expand(r,1)     = r
expand(r,2*n)   = join(expand(r,n),expand(r,n),m)
                                    ++ expand(r,n)
```

In Figure 2, Line 37, `proper_descendant` is defined to be equal to the relation `child` unfolded 4 times.

Expandable relations in Avestan are a more general form of binary relations; as a result, the **`expand`** operator of Avestan is more flexible than the transitive closure of Alloy, which can only be applied to relations having arity 2.

C. Constraint Language

After declaring the entities in an Avestan model, the keyword **`suppose`** can be used to express the constraints that the entities need to satisfy; e.g., the constraint in Line 31 of Figure 2 states that the left child and the right child of `null` are both `null`. The logic that Avestan provides for writing constraints is many-sorted first-order logic [6]. Avestan supports quantifiers, **`forall`**, **`exists`**, and the logical operators, **`and`**, **`or`**, **`not`**, **`xor`**, **`implies`**, **`iff`**. The connectives **`and`** and **`or`** can be used in prefix form where they are provided a list of constraints; this list should have at least two constraints. Line 14 of Figure 2 uses **`and`** in prefix format.

Avestan provides term comparison operators, `=`, `!=`, `>`, `<`, `>=`, `<=`; relational operators for checking the equality of two relations, `==`, `=!=`; and relational operators for checking the subset relation **`in`**, **`!in`**; e.g., Line 37 of Figure 2 states that the relation `proper_descendant` is equal to the relation `child` unfolded for 4 steps.

[2]A total function is defined for all the elements in its domain.

Another construct that Avestan provides for writing constraints is **definition**. Definitions are implicitly universally quantified; e.g., the definition in Lines 19-22 of Figure 2 can be replaced by the following constraint:

```
suppose forall t,c:BTree| child(t,c) iff
  t!=null and left_child(t)=c or right_child(t)=c.
```

The most important difference between definitions in Avestan and Alloy is that the definitions in Avestan can have terms using the **if-then-else-fi**, but Alloy does not allow this form. Modelling the **definition** in Lines 25-28 of Figure 2 in Alloy requires the expansion of the if-then-else construct and implicit use of quantifiers. This reduces the readability of a model. The following is the Alloy code for the **definition** in Lines 25-28 of Figure 2:

```
1sig BTree{ size: Int,
2  left_child, right_child: BTree}
3one sig null extends BTree {}
4fact { all t: BTree|
5  (t=null and size[t]=0) or
6  (t!=null and size[t]=size[left_child[t]]
7   +size[right_child[t]] + 1)}
```

Avestan provides macros to avoid writing the same constraints and terms over and over; e.g., Lines 13-15 of Figure 2 declares `is_leaf(t)` as the conjunction of three constraints, Line 14.

D. Analysis Command

The last part of an Avestan model is the analysis command. Avestan supports two types of analysis: 1) entailment, and 2) consistency. Entailment is used to check whether an Avestan model logically implies a set of constraints; e.g., in the Avestan model of Figure 2, the entailment block, Lines 47-49, asks whether it is possible to conclude that the `size` of every nonempty tree[3], `size(t)`, is equal to twice the number of its leafs, `leaf_number(t)`, minus one.

If an entailment block contains more than one constraint, the analyzer is instructed to check whether all the constraints are logical consequences of the model.

Analyzing an entailment command by an SMT solver has three possible outputs: 1) `Yes`: in this case, the constraints in the entailment block logically follow the specifications, 2) `No`: in this case, not all constraints are logical consequences of the specifications; the answer `No` is accompanied by a *counterexample*. A counterexample is an interpretation that satisfies the specifications of the model but does not satisfy all the constraints in th entailment block. 3) `Maybe`: in this case, the analyzer is not able to make a conclusion.

In consistency checking, the analyzer is asked to check whether the Avestan model is consistent. If the model is consistent, then there exists a satisfying interpretation for the Avetsan model. If the model is not consistent, then such an interpretation does not exist; e.g., we can replace Lines 47-49 of Figure 2 with the following command, and as a result, the analyzer is instructed to check for the consistency of the Avestan model:

```
is_consistent ?
```

Avestan provides another consistency checking command, **is_consistent_with**. This command must be followed by a set of constraints that are conjuncted with the constraints of the model, and then checked for consistency. This is similar to running a predicate in Alloy [7].

Similar to entailment, consistency checking by an SMT solver can lead to one of the following three outputs: 1) `Yes`: in this case, the constraints are satisfiable and the answer `Yes` is accompanied with a *witness*, which is an interpretation that satisfies the specification of the Avestan model. 2) `No`: in this case, the Avestan model is not consistent. 3) `Maybe`: in this case, the analyzer was not able to make a conclusion.

The analysis command, entailment or consistency, can be preceded by *cardinality constraints*. Cardinality constraints can *only* be used to make the user-defined sorts finite sets; e.g., Line 41 of Figure 2 is a cardinality constraint that states the number of binary tress, `|BTree|`, is 4.

In case the consistency checking results in the output `No`, the unsatisfiable core finding capability of SMT solvers may be applied to find the source of inconsistency in the model.

III. Translation

Figure 3 is the SMT-LIB output of the translation of Figure 2. Except for relational operators, every construct in Avestan has a corresponding S-expression in SMT-LIB; e.g., the constraint in Line 32 of Figure 2 is translated to the `assert`ion in Line 22 of Figure 3. Before translating Avestan to SMT-LIB, a preprocessor expands macros. e.g., the macro `is_leaf` is expanded, and the expansion occurs in Lines 27 and 31 of Figure 3.

Translation of relational constraints is done via rewriting them into their equivalent form in first-order logic; e.g., the equality of the two relations `proper_descendant` and **expand**`(child,4)`, which is expressed in Line 37 of Figure 2, is translated to the assertion in Lines 40-41 of Figure 3.

The operators **join** and **expand** are translated according to their definitions; e.g., Lines 34-38 of Figure 3 represent the translation of **expand**`(child,4)`.

For consistency checking, the translator simply adds the `(check-sat)` and `(model)` commands to the end of the translated model. In the case of entailment, the constraints in the entailment block are all conjuncted (if there is more than one) to form a single formula, and the negation of this formula is added to the translation along with `(check-sat)`, and `(model)` commands: if the SMT solver outputs `unsat`, the result of entailment

[3]A tree that is not `null`.

```
 1 (declare-datatypes ((BTree (BTree-1) (BTree-2) (BTree-3) (BTree-4))))
 2
 3 (declare-fun null () BTree)
 4
 5 (declare-fun left_child (BTree) BTree)
 6 (declare-fun right_child (BTree) BTree)
 7
 8 (declare-fun size (BTree) Int)
 9 (declare-fun leaf_number (BTree) Int)
10
11 (declare-fun child (BTree BTree) Bool)
12 (declare-fun proper_descendant (BTree BTree) Bool)
13
14 (assert (forall ((t BTree) (c BTree))
15   (= (and (not (= t c)) (or (= (left_child t) c) (= (right_child t) c))) (child t c))))
16
17 (assert (forall ((t BTree))
18   (=(size t) (ite (= t null) (0) (+ 1 (+ (size (right_child t)) (size (left_child t))))))))
19
20 (assert (and (= (left_child null) null) (= (right_child null) null)))
21
22 (assert (= (leaf_number null) 0))
23
24 (assert (forall ((t BTree)) (=> (= (size t) 0) (= t null))))
25
26 (assert (forall ((t BTree))
27   (=> (and (not (= t null)) (= (right_child t) null) (= (left_child t) null))
28            (= (leaf_number t) 1))))
29
30 (assert (forall ((t BTree))
31   (or (and (not (= t null)) (= (right_child t) null) (= (left_child t) null))
32       (= (leaf_number t) (+ (leaf_number (left_child t)) (leaf_number (right_child t)))))))
33
34 (define-fun expand-child-2 ((t1 BTree) (t2 BTree)) Bool
35   (or (child t1 t2) (exists ((t3 BTree)) (and (child t1 t3) (child t3 t2)))))
36 (define-fun expand-child-4 ((t1 BTree) (t2 BTree)) Bool
37   (or (expand-child-2 t1 t2) (exists ((t3 BTree))
38      (and (expand-child-2 t1 t3) (expand-child-2 t3 t2)))))
39
40 (assert (forall ((t1 BTree) (t2 BTree))
41    (iff (proper_descendant t1 t2) (expand-child-4 t1 t2))))
42
43 (assert (not (exists ((t BTree)) (and (not (= t null)) (proper_descendant t t)))))
44
45
46 (assert (not (forall ((t BTree))
47    (=> (not (= t null)) (= (size t) (- (* 2 (leaf_number t)) 1))))))
48
49 (check-sat)
50 (model)
```

Figure 3. SMT-LIB model of Figure 2

checking is `Yes`; otherwise, if the output of the analyzer is `sat`, the result of entailment checking is `No`, and the valid interpretation that is given by the SMT solver is a counterexample. A simple method can transform the output of the SMT solver to an instance for Avestan. This method needs to eliminate some extra information that has been introduced during the translation. Lines 46-50 of Figure 3 are the translation of the entailment block of Figure 2.

By running the SMT-LIB model of Figure 3 in Z3, we see that the solver produces the tree of Figure 4 as a counterexample. This counterexample shows that we have underspecified binary trees in the Avestan model of Figure 2; constraints stating that a tree must have a root, and every node has at most of parent must be added to the model.

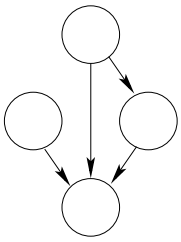

Figure 4. Counterexample for the Avestan model of Figure 2

IV. Related Work

Z [8] is a set-based modelling language that has inspired many other modelling languages, such as Alloy and S [9]. Its analyzers are focused on theorem proving rather than finite scope analysis. These theorem provers, such as Proof-Power [10], are not fully automatic. Z is more expressive than Avestan, but the notation is not in ASCII form, and its models cannot be analyzed automatically, unless sets are finite.

Alloy is a lightweight modeling language that has static type checking. The logic that Alloy provides for modeling is first-order logic with the transitive closure operator. An Alloy model consists of a set of declarations, which specifies the sets, relations, and functions in a model, and a set of constraints, which are logical formulas. In general, first-order logic is undecidable, and as a result, consistency checking of Alloy models is not possible. The Alloy Analyzer, the main analysis tool for Alloy models, provides finite scope analysis: a user is required to fix the size of the sets in the model to constant numbers and then, the Alloy Analyzer translates the model to a CNF formula, which is then handed to a SAT solver for consistency checking. The Alloy language together with its analyzer is unique in that it combines the ability to write high-level declarative specifications with fully automated analysis via SAT solving for finite scopes. Without the transitive closure, Avestan is as expressive as Alloy; having transitive closure in Alloy makes it a more expressive notation, but since Alloy models are analyzed for finite scopes, the **`expand`** operator of Avestan provide the same readability as the transitive closure of Alloy; moreover, Avestan models are analyzed by SMT solvers rather than SAT-solvers. This makes it possible to analyze Avestan models for infinite scopes. Modelling a general consistency checking problem in Alloy often results in not very readable models, due to the object-oriented style of Alloy. For this reason, Avestan's syntax is closer to logical notations. At the same time, relational constructs and set operators in Avestan make it suitable for object-oriented style modelling.

Ghazi and Taghdiri show how Alloy models can be analyzed by an SMT solver [11]. Their translation uses the λ-expressions and quantifiers of Yices [12], and as a result, the analysis can be unsound.

Cadoli and Schaerf present a compiler that translates declarative specifications into SAT [13]. The language of the compiler is NP-SPEC, which can express NP problems. Avestan is more expressive than NP-SPEC since it supports infinite domains and any PSPACE problem can also be expressed in Avestan.

S [9] is a modelling language based on higher order logic. This language was developed to be used in an air traffic control system project. The main motivation for developing S was that existing modelling notations, such as Z, were not suitable for the project; as a result, S, a gentler and kinder Z, was developed to be machine readable and applicable to industrial projects. Theorem proving is the main technique for analyzing S models; an S model is translated to the input language of theorem provers, such as HOL [14].

The Abstract State Machine (ASM) method [15] is for high-level system design and analysis. The ASM method is used to specify software systems. Analysis techniques for the ASM method include theorem proving [16], [17], and model checking [18], which consists of translating a certain class of ASMs to SMV. Translation of abstract models to the input language of state-of-the-art model checkers may result in large models.

B [19] is a modeling language that has many similarities with Alloy. Models developed in B are called B *machines*, and the variables used to define the state space can be sets and relations. ProB [20] is a tool for analyzing B machines, in particular, model checking and automatic refinement checking of B machines. ProB provides model checking of safety properties, which are presented as invariants in B machines. These properties are checked by explicit state-space search. Since each single state in a B machine represents some sets and relations, computing the set of the next states of a single state is computationally very costly. Similar to Alloy, B machines are analyzed for finite scopes in ProB.

V. Future Work

We are extending Avestan in two ways: 1) the language itself, 2) the translator from Avestan to SMT-LIB. The main language extension we plan is constructs for modelling dynamic behavior of systems; in particular, modelling transition systems [21]. Transition systems that are developed in first-order logic and relational calculus are often not readable. In the future, Avestan will include constructs that can be used to refer to the values of variables in the next state of a transition system.

Currently, the translator from Avestan to SMT-LIB does not do any heavy preprocessing of the input. In the future, our translator will include a "simplifier" that optimizes Avestan models based on optimization methods for declarative models [22].

VI. Conclusion

Avestan is a declarative modelling language that has been inspired by SMT-LIB and Alloy. The main motivation for developing this language is to make the SMT solver technology more accessible to the world of model driven engineering [23]. SMT solvers are analysis tools that check the satisfiability of a set of logical formulas in the presence of some theories. These theories provide interesting types, such as integers and real numbers, that can be used in modelling.

SMT-LIB, which is a standard notation for the input to SMT solvers, is not readable: an SMT-LIB model consists of a set of S-expressions; the excessive use of parentheses in S-expressions, and their prefix format, makes SMT-LIB models machine readable rather than human readable. Avestan overcomes this deficiency by providing a human readable syntax, and relational and set operators to facilitate modelling declarative models; moreover, the logic based notation of Avestan; the flexibility of its operators, such as **`join`**, **`expand`**; its more general form of constructs such as **`if-then-else-fi`**; its macros; and built-in types, such as **`Int`** and **`Real`**, make Avestan models more readable than Alloy for modelling a general consistency checking problem.

To analyze an Avestan model, first, the Avestan model is translated to SMT-LIB and then an SMT solver is used to analyze the translated model. The main motivation for taking this approach for analyzing Avestan models is to provide a user with different choices for the SMT solver. The translator for Avestan is developed in SWI-Prolog [24]. Detailed documentation of the language and the translator are available on the web [5].

References

[1] C. Barrett, A. Stump, and C. Tinelli, *The SMT-LIB Standard Version 2.0 Reference Manual*, Jan. 2010. [Online]. Available: www.SMT-LIB.org

[2] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2008, vol. 4963, pp. 337–340.

[3] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th international conference on Computer aided verification*, ser. CAV'07. Springer-Verlag, 2007, pp. 298–302.

[4] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM TOSEM*, vol. 11, no. 2, pp. 256–290, 2002.

[5] "http://www.cs.uwaterloo.ca/~avakili/projects/avestan."

[6] M. Manzano, *Introduction to many-sorted logic*. John Wiley & Sons, Inc., 1993, pp. 3–86.

[7] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[8] *Information Technology Z Formal Specification Notation Syntax, Type System and Semantics*, International Organisation for Standardization, 2000.

[9] J. Joyce, N. Day, and M. Donat, "S: A machine readable specification notation based on higher order logic," in *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*. Valletta, Malta: Springer-Verlag, September 1994, pp. 285–299.

[10] *ProofPower*. [Online]. Available: http://www.lemma-one.com/ProofPower/index/index.html

[11] A. A. E. Ghazi and M. Taghdiri, "Analyzing Alloy Constraints using an SMT Solver: A Case Study," in *Automated Formal Methods (AFM) workshop*, 2010.

[12] B. Dutertre and L. De Moura, "The Yices SMT Solver," *Tool paper at httpyices csl sri comtoolpaper*, pp. 1–5, 2006.

[13] M. Cadoli and A. Schaerf, "Compiling problem specifications into sat," *Artificial Intelligence*, vol. 162, no. 12, pp. 89 – 120, 2005.

[14] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[15] E. Börger, "The ASM Method for System Design and Analysis. A Tutorial Introduction," in *Frontiers of Combining Systems*, ser. LNCS. Springer, 2005, vol. 3717, pp. 264–283.

[16] G. Schellhorn and W. Ahrendt, "Reasoning about Abstract State Machines: The WAM Case Study," *Journal of Universal Computer Science*, vol. 3, no. 4, pp. 377–413, 1997.

[17] A. Dold, "A Formal Representation of Abstract State Machines Using PVS," Universität Ulm, Verifix Technical Report Ulm/6.2, Jul. 1998.

[18] G. Del Castillo and K. Winter, "Model Checking Support for the ASM High-Level Language," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2000, vol. 1785, pp. 331–346.

[19] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[20] M. Leuschel and M. Butler, "ProB: A Model Checker for B," in *FME 2003: Formal Methods*, ser. LNCS. Springer, 2003, vol. 2805, pp. 855–874.

[21] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[22] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard, "Optimizations for Compiling Declarative Models into Boolean Formulas," in *Theory and Applications of Satisfiability Testing*, ser. LNCS. Springer, 2005, vol. 3569.

[23] B. Selic, "From Model-Driven Development to Model-Driven Engineering," in *ECRTS*. IEEE Computer Society, 2007.

[24] J. Wielemaker, "SWI-Prolog 5.3 Reference Manual," 2004.