jMosel: A Flexible Tool-Set for Monadic Second-Order Logic on Strings

Christian Topnik[1] Eva Wilhelm[1] Tiziana Margaria[2]
Bernhard Steffen[1]

[1]Universität Dortmund, FB Informatik, Lehrstuhl 5,
{christian.topnik,eva.wilhelm}@uni-dortmund.de
steffen@cs.uni-dortmund.de
[2]Universität Göttingen, Service Engineering for Distributed Systems,
margaria@cs.uni-goettingen.de

Abstract. *jMosel* is a tool-set for the analysis and verification of linear parametric systems in monadic second-order logic on strings. In this paper we concentrate on the presentation of the core tool which supports several input and output formats as well as the interchange of the tool-set's internal components. The main design principles of *jMosel* are its layered approach to the logic, the definition of a formal semantics for a minimal subset, its modular design and its integration into the *jABC* application design environment. The tool demonstration in the appendix shows how to use *jMosel* as a stand-alone tool and as a plugin for the *jABC* environment.

1 Introduction

Monadic second-order logic on strings (*M2L(Str)*) was proposed as an appropriate formalism for reasoning about bit vector sequences by A. Church in the 1960's [3]. It is expressive enough to capture parametric finite-state systems and it is also decidable, though in non-elementary time. However, many relevant practical problems have proved to be solvable in reasonable time.

A convenient characteristic of this logic is that it is both an abstract specification language and a powerful programming language, since every specification corresponds to a finite-state automaton representing an executable behaviour.

It has been applied for the specification and verification of classes of parametric hardware systems [2, 6, 7, 9] and software systems [10, 11], in which the logic can serve, for example, as a description language for model-based analysis.

This paper introduces the new tool *jMosel*, that offers a tool-set for handling M2L(Str) formulas, i.e. constructing the automata representing the desired semantics and providing the means to further work with them in different contexts. *jMosel* is designed to include a flexible set of decision procedures for several theories of the logic, complemented by different input and output formats, as well as by interfaces to other logics and tools for analysis, verification and synthesis. The semantics for the Minimal Logic is defined via finite-state automata.

It is developed as the successor to the MoSeL tool-set from the 1990's, but using current technologies like Java and XML. The emphasis is placed on *flexibility*, to allow the *customisation* of nearly every aspect of the tool's properties, which also is one of *jMosel's* unique features distinguishing it from other implementations like the ones mentioned above. The input syntax, the output format, and libraries referenced by the compiler unit can be exchanged easily, which already enables *jMosel* to be combined with a variety of other applications.

2 The jMosel Concept

2.1 Design Principles

Two central design principle of *jMosel* are the layered approach to the logic and the library-based design.

Layered approach to the logic. *jMosel's* logic is built according to a hierarchy of logic layers (see figure 1) with increasingly powerful constructs. These layers are related by either direct embedding or more elaborate encodings realized in form of a more or less complex compiler.

- The *Minimal Logic* contains a minimal set of primitives, for which the semantics is formally defined in terms of corresponding automata.
- The *Kernel Logic* extends the Minimal Logic by additional derived constructs and coincides with the set of constructs actually implemented as primitives in the semantic decision procedure. The goal is here an optimisation of the execution time by offering more powerful constructs.
- A set of *user logics* can be encoded in the Kernel Logic. They are either convenient for generic applications or tailored to a specific application domain.

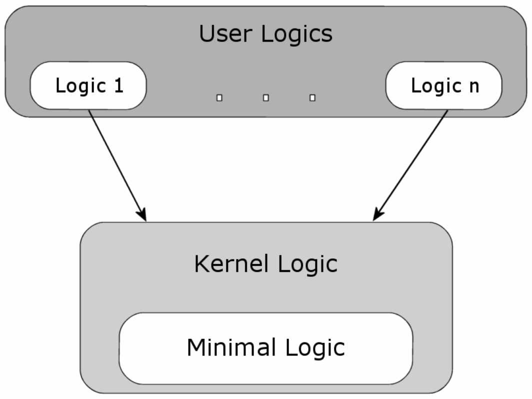


Fig. 1. The Hierarchy of Logics

Library-based design. *jMosel* supports flexible adaptation and extension to new input or output formalisms, as well as the interchange of many components, including the most crucial algorithms and libraries. This allows the user to experiment with a great variety of technologies. The aim is that the best-fitting incarnation of the tool for a specific application area may be put together at need from the collection of existing components.

– Exchangeability concerns e.g. the **BDD libraries**. The performance of the operations on the BDDs, which label the edges of a *jMosel* automaton, has a great impact on the overall performance of the compilation process. To encourage the user to compare different implementations, *jMosel* already supports the use of the following BDD libraries: CUDD [17], BuDDy [5], CAL BDD [14], JavaBDD (Java port of BuDDy) [19], JavaBDD Micro (Java port of BuDDy) [19], JDD [18]. Switching between these libraries can be done easily by changing the corresponding *lib* parameter of the command line tool (see appendix).
– Even such fundamental **compiler parts**, like the algorithms for determinisation and minimisation, can be easily changed or replaced.
– Various **visualisation tools**, like Graphviz or our own *jABC* libraries for graph layout and rendering, can be used - as shown in the demonstration (see appendix).
– Furthermore, *jMosel* has been integrated as a **plugin** in *jABC* [12], a modular general purpose modelling and design tool for heterogenous software systems.
– The *jABC* can be itself used as a plugin to the **Eclipse** environment, thus *jMosel* can be directly accessed within a generic software development environment.

2.2 The Syntax

jMosel's syntax slightly differs from the original MoSeL syntax of [4]. Changes have been introduced to uniform it with (modern) standard notations.

Minimal logic The Minimal Logic (figure 2) is a concrete syntactic version of the monadic second-order logic on strings, using a minimal set of primitives. It serves as the reference logic for *jMosel*, relative to which the correctness and completeness of the implementation can be verified and the semantics of the various extensions of the language can be defined.

Kernel Logic The drawback of a very tiny language, like the Minimal Logic, is the fact that nearly all the constructs of a typical user-level language correspond to complex expressions. To avoid the cost of breaking down frequently used user-level constructs to the few primitives of the Minimal Logic every time they are used, the Kernel Logic (see figure 3) was defined to support a reasonable set of derived constructs by direct semantic translations.

```
T ::= Id
A ::= subseteq(T,T)  |  shifteq(T,T)
F ::= A  |  ~F  |  F & F  |  ex Id: F  |  (F)
```

Fig. 2. The Minimal Logic of the *jMosel* Tool

```
 T ::= Id  |  all  |  empty  |  union(T,T)  |  inter(T,T)  |
       comp(T)  |  (T)
 A ::= sing(T)  |  ~sing(T)  |  subset(T,T)  |  ~subset(T,T)  |
       subseteq(T,T)  |  ~subseteq(T,T)  |  T = T  |  T ~= T  |
       shifteq(T,T)   |  ~shifteq(T,T)   |  T < T  |  T <= T  |
       roteq(T,T)   |  ~roteq(T,T)   |
       0 in T  |  0 ~in T  |  $ in T  |  $ ~in T
 F ::= true  |  false  |  F & F  |  F | F  |  F -> F  |
       F <-> F  |  F ^ F  | Id(T,...,T)  |  ~Id(T,...,T)  |
       ex Id,...,Id: F  |  ~ex Id,...,Id: F  |
       "<automaton_filename>"(T,...,T)  |
       ~"<automaton_filename>"(T,...,T)  |
       let Id(Id,...,Id) in F  |  A  |  (F)
```

Fig. 3. The Kernel Logic of the *jMosel* Tool

2.3 The Semantics

The *jMosel* formulas are transformed into complete and deterministic finite-state automata in such a way that the language recognized by an automaton corresponds to the interpretation of the represented formula. The semantics of the two atomic formulas of the Minimal Logic `shifteq` and `subseteq` are given in figure 4 and figure 6 respectively. These automata are not constructed, but primitive.

3 Implementation

jMosel is implemented in *Java*, for easy maintainance of the code and to make the tool instantly available on nearly every important hardware and operating system. Only for the most crucial and time consuming part, the potentially complex edge labels represented by BDDs, *C++* libraries can be referenced, to ensure fast calculations with minimum overhead. At need, the same mechanism can be used to access packages written in other languages or in assembler - but this would then reintroduce platform-dependence.

The architecture of *jMosel's* parser/compiler unit is shown in figure 5. A formula (given as a parameter string or a reference to a `*.mos` - file) is passed to the command line tool *jMoselC*, which invokes the parsing of the corresponding

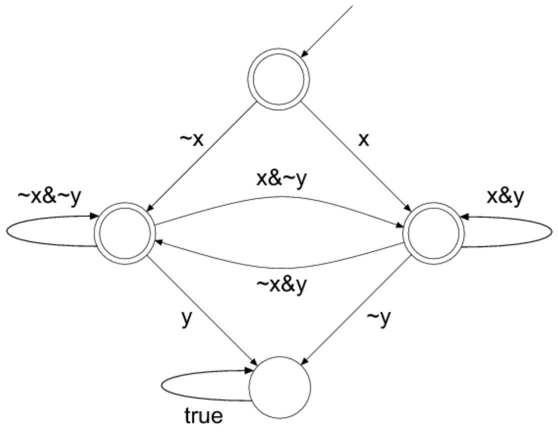


Fig. 4. Automaton for "shifteq(x,y)"

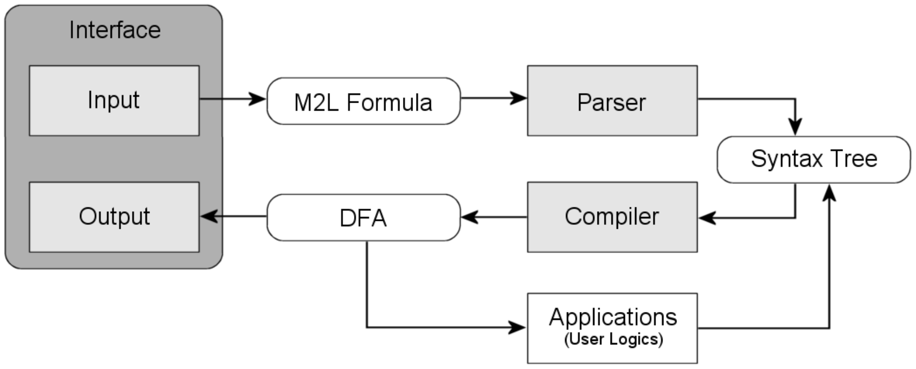


Fig. 5. The Parser/Compiler Architecture

syntax tree. The compiler traverses the tree to create the automaton representing the formula's semantics, which is finally translated into the desired output format.

4 Future Work

The main emphasis of the development within the jMosel group is placed on the following tasks:

– The *jETI System* [8, 13] will enable *jMosel* to be remotely executed over the internet as a WebService.
– An extension of the Kernel Logic will contain first-order and bit variables, e.g. as arguments to quantifiers.

– A PSL [1] user logic will offer means for the formulation and verification of PSL assertions based on regular expressions.
– An exhaustive library of hardware circuits will allow the intuitive description and verification of parametric hardware systems at register-transfer and gate level. This approach abstracts from the underlying logic layers and therefore is accessible for users unfamiliar with M2L(Str).

References

1. Accellera Organization, Inc. *Accellera Property Specification Language 1.1 Reference Manual*, 2004.
2. David A. Basin and Nils Klarlund. Hardware Verification using Monadic Second-Order Logic. In *Proc. CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 31–41. Springer Verlag, 1995.
3. Alonzo Church. Logic, arithmetic and automata. In *Proc. Intern. Congr. Math.*, pages 23–35. Almqvist and Wiksells, 1963.
4. Peter Kelb, Tiziana Margaria, Michael Mendler, and Claudia Gsottberger. MOSEL: A Flexible Toolset for Monadic Second-Order Logic. In *Proc. TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 183–202. Springer Verlag, 1997.
5. Jørn Lind-Nielsen. BuDDy. `http://sourceforge.net/projects/buddy`. 14. 12. 2005.
6. Tiziana Margaria. Fully Automatic Verification and Error Detection for Parameterized Iterative Sequential Circuits. In *Proc. TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 258–277. Springer Verlag, 1996.
7. Tiziana Margaria. Verification of Systolic Arrays in M2L(Str). In *Technical Report MIP-9613*. Fakultät für Mathematik und Informatik, Universität Passau, 1996.
8. Tiziana Margaria. Web Services-based Tool-Integration in the ETI Platform. In *SoSyM, International Journal on Software and System Modelling*. Springer Verlag, 2004.
9. Tiziana Margaria and Michael Mendler. Automatic Treatment of Sequential Circuits in Second-Order Monadic Logic. In *Proc. 4th GI/ITG/GME Workshop on "Methoden des Entwurfs und der Verifikation digitaler Systeme"*, pages 21–30. Shaker Verlag, 1996.
10. Tiziana Margaria and Michael Mendler. Model-based Automatic Synthesis and Analysis in Second-Order Monadic Logic. In *Proccedings AAS'97, ACM/SIGPLAN Int. Worksh. on Automated Analysis of Software*, pages 99–112, 1997.
11. Anders Møller. Program Verification with Monadic Second-Order Logic & Languages for Web Service Development. Technical report, Brics, Daimi, 2002. PhD thesis.
12. Ralf Nagel. jABC. `http://jabc.cs.uni-dortmund.de`. 14. 12. 2005.
13. Ralf Nagel. jETI. `http://jeti.cs.uni-dortmund.de`. 14. 12. 2005.
14. Rajeev Ranjan. CAL BDD. `http://www-cad.eecs.berkeley.edu/Research/cal_bdd/`. 14. 12. 2005.
15. AT&T Research. Dot format. `http://www.graphviz.org/cvs/doc/info/lang.html`. 14. 12. 2005.
16. AT&T Research. Graphviz. `http://www.graphviz.org/`. 14. 12. 2005.
17. Fabio Somenzi. CUDD. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`. 14. 12. 2005.

18. Arash Vahidi. JDD. `http://javaddlib.sourceforge.net/jdd/index.html`. 14. 12. 2005.
19. John Whaley. JavaBDD. `http://javabdd.sourceforge.net/`. 14. 12. 2005.

A Tool Demonstration

A.1 The Command Line Tool (for *jETI* Integration)

This short demonstration of the command line tool shows the functionality that will be integrated in *jETI* as a WebService and at the same time puts emphasis on two of the different output formats (*dot*, *XML*).

Visualizing an automaton with Graphviz
The command

```
jmoselc -in cmd "subseteq(x,y)." -out dot graph.dot
```

invokes the generation of the following output in the *dot format* [15] that is stored in the file "graph.dot":

```
/* Automaton generated by JMoselC */

digraph Automaton
{
   void [style=invis]; /* Used for starting pointer */
   void -> 0;
   0 [label="", shape=circle, peripheries=2];
   0 -> 0 [label="(~x&~y)|y"];
   0 -> 1 [label="x&~y"];
   1 [label="", shape=circle, peripheries=1];
   1 -> 1 [label="true"];
}
```

When viewed with a tool like *Graphviz* [16], the automaton looks like the one displayed in figure 6.

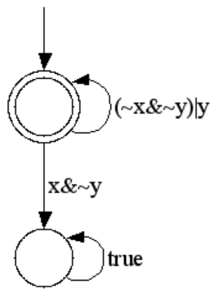


Fig. 6. Automaton for "subseteq(x,y)"

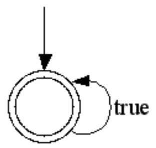


Fig. 7. Constant *true* automaton

Changing the BDD package and producing XML output
The invocation of the tool via the command

```
jmoselc -in cmd "subseteq(x,y)." -out xml graph.xml -lib cudd
```

computes the same automaton, but now using the CUDD library during compilation (instead of the standard Java BDD implementation), and changing the output format to XML. The resulting file looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Automaton generated by jMosel -->

<moselautomaton initialstate="s0">
    <state id="s0" accepting="yes">
      <edge to="s0">
         <label>
            (~x&~y)|y
         </label>
      </edge>
      <edge to="s1">
         <label>
            x&~y
         </label>
      </edge>
    </state>
    <state id="s1" accepting="no">
      <edge to="s1">
         <label>
            true
         </label>
      </edge>
    </state>
</moselautomaton>
```

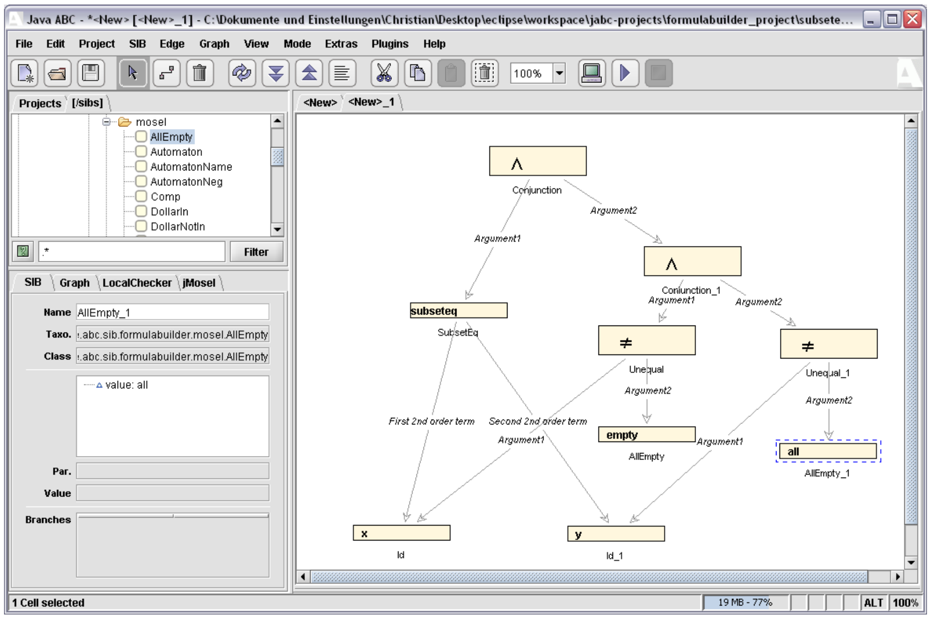
The exact format of the XML output could be customized further, depending on what information is needed by a target application. For example, it is possible to store a representation of the BDDs within the <*label*> tags, or to include a representation of the formula out of which the automaton was computed.

A.2 *jMosel* as *jABC* plugin

In this second part of the demonstration we show how to use *jMosel* inside the *jABC*, where it has been integrated as a plugin. In particular,

1. We construct a *jMosel* formula by drawing it as a SIB-Graph inside the *Formula Builder*. Fig. 8 shows the formula

```
subseteq(x,y) & x~= empty & y~= all.
```

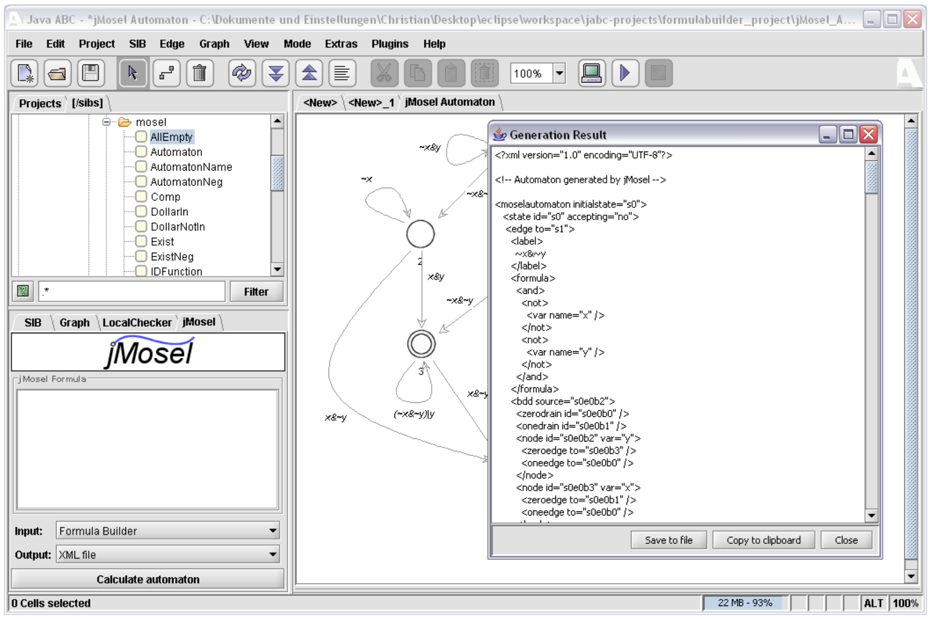


Name AllEmpty_1
Taxo. ·.abc.sib.formulabuilder.mosel.AllEmpty
Class ·.abc.sib.formulabuilder.mosel.AllEmpty
value: all
Par.
Value
Branches

Fig. 8. Creating a formula graphically

which accepts all the nontrivial x-subsets of y.

2. The automaton to that formula is transformed into a SIB-Graph and visualized in the *jABC* (figure 9), where we also see the XML export format.
3. We show here how the Kernel Logic constructs can be verified to be correct wrt. the Minimal Logic. In figure 10 the definition of `sing` is typed in the *jMosel* input field, it is translated into an automaton and exported as a *dot* file.



Input: Formula Builder

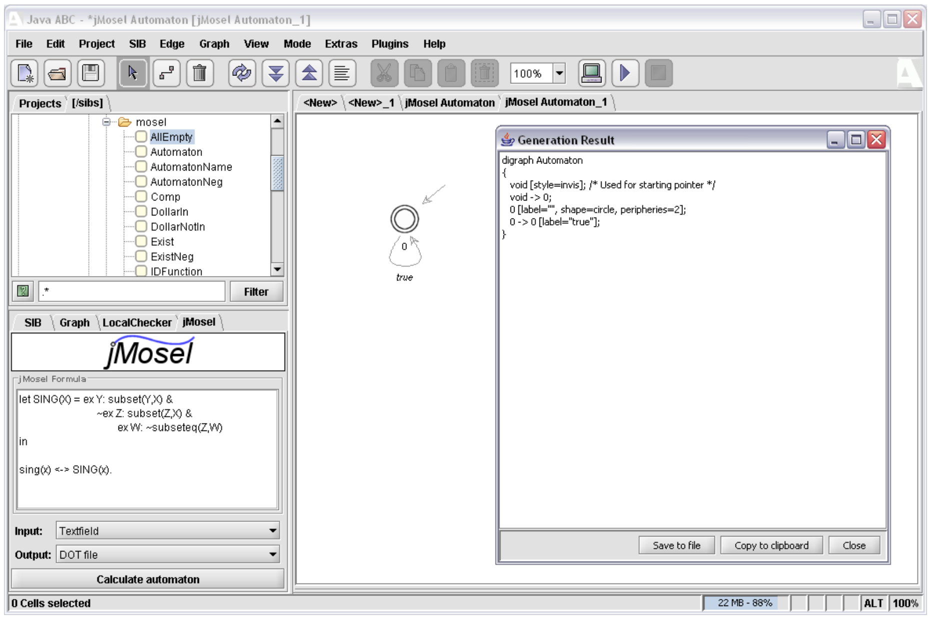
Output: XML file

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Automaton generated by jMosel -->

<moselautomaton initialstate="s0">
  <state id="s0" accepting="no">
   <edge to="s1">
    <label>
     ~x&~y
    </label>
    <formula>
     <and>
      <not>
       <var name="x" />
      </not>
      <not>
       <var name="y" />
      </not>
     </and>
    </formula>
    <bdd source="s0e0b2">
     <zerodrain id="s0e0b0" />
     <onedrain id="s0e0b1" />
     <node id="s0e0b2" var="y">
      <zeroedge to="s0e0b3" />
      <oneedge to="s0e0b0" />
     </node>
     <node id="s0e0b3" var="x">
      <zeroedge to="s0e0b1" />
      <oneedge to="s0e0b0" />
```

Fig. 9. Visualizing the automaton in *jABC* and XML export



```
let SING(X) = ex Y: subset(Y,X) &
                  ~ex Z: subset(Z,X) &
                       ex W: ~subseteq(Z,W)
in

sing(x) <-> SING(x).
```

Input: Textfield

Output: DOT file

```
digraph Automaton
{
  void [style=invis]; /* Used for starting pointer */
  void -> 0;
  0 [label="", shape=circle, peripheries=2];
  0 -> 0 [label="true"];
}
```

Fig. 10. Consistency proof of the `sing` construct (formula input on the left) and *dot* export