MOSEL: A Flexible Toolset for Monadic Second-Order Logic

Peter Kelb, Tiziana Margaria, Michael Mendler, Claudia Gsottberger

Universität Passau, D-94032 Passau (Germany),
{kelb,tiziana,mendler,gsottber}@fmi.uni-passau.de

Abstract. MOSEL is a new tool-set for the analysis and verification in Monadic Second-order Logic. In this paper we concentrate on the system's design: MOSEL is a tool-set to include a flexible set of decision procedures for several theories of the logic complemented by a variety of support components for input format translations, visualization, and interfaces to other logics and tools. The main distinguishing features of MOSEL are its layered approach to the logic, based on a formal semantics for a minimal subset, its modular design, and its integration in a heterogeneous analysis and verification environment.

1 Introduction and Background

Already 30 years ago Alonzo Church proposed monadic second-order logic on strings (M2L(Str)) as an appropriate specification formalism for reasoning about sequences of bitvectors [9]. This logic is among the most succinct decidable logics known to capture finite state systems. It is decidable, however, only in non-elementary time: the worst-case complexity is a stack of exponentials of height proportional to the size of the formula, a good reason for it having been considered impractical for a long time. Known almost exclusively to theoreticians for a long time, recently this logic celebrates a certain renaissance: despite the worst-case computational 'intractability' of this logic, relevant practical problems are usually far better behaved and can be solved automatically in reasonable time. Fields of application have been the specification, verification, and synthesis, in a fully automatic manner, of relevant classes of parametric systems. In particular, the logic can be used profitably as a description language for model-based analysis of software [17] as well as hardware systems [2,16,18,19] and is therefore a good candidate formalism for hardware/software codesign. Some examples of distributed systems have been addressed too [14,15]. From an application point of view this logic conveniently combines two important features in a single formalism: It is both an abstract specification language and an effective programming language. Every specification can be translated into executable behaviour in the form of an equivalent finite state automaton.

In this paper we present MOSEL, a new system for the automatic analysis and verification in Monadic Second-order Logic. The accent here is put primarily on the system's design, rather than on individual algorithms: MOSEL is a

tool-set, which, in its complete realization, will include a flexible set of decision procedures for several theories of the logic (e.g., finite and infinite strings, and trees) complemented by a variety of support components to provide input format translations, visualization, and interfaces to other logics and other analysis, verification, and synthesis tools.

The availability and construction of composite, even heterogeneous tools is supported by the METAFrame®[1] concept [25,26], a system-level programming environment, and it is actively promoted in projects like the METAFrame-based Electronic Tool Integration platform which will be available in the coming Springer Journal on Software Tools for Technology Transfer. From these and other project experiences it has clearly emerged that single monolithic systems are becoming less and less adequate for the challenges of modern system-level verification, and that tool support must be granted in an increasingly flexible, application-specific, and user-friendly way. The decision of extending the repository of algorithms, tools, and support components already available in METAFrame to deal also with monadic second-order logic, therefore, was naturally linked with a component-based implementation, which increases software reusability and offers high flexibility for the overall environment.

At the moment we have implemented in MOSEL a semantic decision procedure for monadic second-order logic over finite strings and a set of interface modules. Decision procedures for other variants of the logic as well as further support components are planned to follow at need. Although in the rest of the paper we will refer only to the current tools of MOSEL, the design principles and the overall system concept are valid in general.

Related Work. Toolmakers have recently started to show interest in monadic second-order logic. To our knowledge, however, only few implementations are available at the moment. Two groups have been actively working on tools: in Århus the Mona [14] and Mona++ packages implement interpretations over strings and trees respectively, and in Kiel the AMoRE system [20] offers a decision procedure for the logic over trees. Decision procedures are also soon to be integrated into STeP [3], and interests in a similar development have been recently shown by the hardware groups at Berkeley and at Indiana University.

The current implementations successfully demonstrated on several interesting case studies, spanning diverse fields of application, that practical examples are often indeed much better behaved than the staggering theoretical worst case complexity would suggest. However, they are still closer to research prototypes, basically lacking documentation as well as flexible structuring and user interfaces, which severely restricts their value for general users.

We can best compare the current finite string implementation of MOSEL with Version 0.2 of the Mona tool [14]. In our experience during the last two years, in which it was also used actively by students in a graduate course on Formal Methods for System Design, the following weaknesses were observed:

[1] METAFrame® is a registered trademark.

- The intuitive definition of Mona's implemented constructs found in [14] has omissions (e.g., it misses predicate definitions) and leaves unclear the correspondence between the published and the implemented versions of the logic, for instance concerning empty strings. Distinguishing between primitive and derived constructs, with explicitly documented encodings, and a proof of the correctness of the semantics, would have avoided those problems. The Mona system is moving recently to a different, weak second-order, semantics which is closer to Büchi's original setup [8], but more difficult to implement. There was no official release of this new version at the time of writing.
- Rigid *user interface* of the tool, which is in pure textual form. Mona accepts only M2L(Str) formulas and delivers automata and counter examples only as lists of transitions. The need for direct input of an automaton (e.g. to use the tool as a model checker), and for some form of visualization of the generated automata is not attended. Not even the same format for automata descriptions delivered as output can be read and used again by the tool.
- Shallow *integratability* of the tool in larger environments. The embedding of Mona into METAFrame was limited by the rigid interface of the former, which forces a one-directional cooperation: since Mona has to run inside the ML interpreter, it was not possible to launch it from METAFrame. Thus we could not use Mona as planned, i.e., as an external decision procedure callable at need with a simple command, nor could we exploit METAFrame's powerful graph manipulation and the wealth of available analysis and verification tools given the user interface limitations.
- Very limited *reuse* of work. The repository of automata created by Mona has the lifespan of a single session. Leaving the tool means losing the library, which is an unexpected waste of computation time for a tool implementing a non-elementary decision procedure. Without access to the source files we had no possibility of eliminating this serious drawback.

The system requirements to MOSEL presented in the following section arose exactly from these points, which, to our knowledge, are not addressed by any other related project.

2 The MOSEL Concept: System Requirements

The following four main principles underly our design of MOSEL:

Definition of semantic models for a minimal subset of the logic. Having started with a M2L(Str) implementation where the logic is interpreted on strings of finite, but arbitrary length, the semantics is defined in terms of finite-state automata, as discussed in the following sections.

Layered approach to the logic. We introduce a hierarchy of logic layers, with increasingly powerful constructs, related by either direct embedding or more elaborate encodings as shown in Fig. 1.

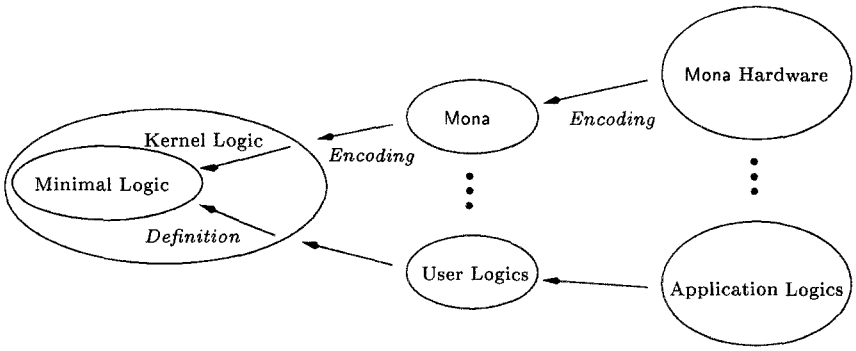

Fig. 1. Layered Logics in MOSEL

- The *minimal logic* contains a minimal set of primitives, for which the semantics is formally defined in terms of corresponding automata (Sect. 3). This set constitutes the reference language for proofs involving semantics, which is very economic e.g. for structural induction on the constructs of the language. While this is the ideal language for developing the theory, it is not adequate for practical use.
- 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 design of this extension is guided by considerations of efficiency of the computations required in the decision procedure, as discussed later in Sect. 4.
- A set of generic *user logics* corresponds to an application-independent layer. They extend the kernel logic by higher-level operators which are convenient for generic applications. We will discuss the syntax of Mona Version 0.2 as an extended example.
- A number of *application-specific logics*, each containing additional admissible predicates and constructs tailored to specific application domains. We have direct experience in the domain of verification and synthesis of hardware, where we deal with families of parametric sequential circuits (Sect. 6).

Due to the principle of implementing outer layers of the logic through successive encodings and definitional extensions to the unique minimal logic, and by making these explicit, the semantic coherence of richer logics with the minimal logic is ensured. The coherence of the kernel wrt. the minimal logic has also been proved to some extent automatically in MOSEL, as reported in Section 5.

Additional advantages of this principle are that the implemented set of primitives is transparent too, and that it is immediately clear which constructs are expensive, since this is determined by their (easily computable) definition in terms of kernel logic constructs.

$T ::= Id$	Second-Order Terms
$A ::= T \subseteq T \mid T + 1 = T$	Atoms
$F ::= A \mid \neg F \mid F \wedge F \mid \exists Id : F$	Formulas

Fig. 2. Minimal Syntax for M2L(Str)

Modular design. While other tools are available only as a single large component, MOSEL is a collection of modules which can be combined or exchanged at need. Following the METAFrame concept of a repository-based library of components, MOSEL supports flexible adaptation and extension to new input or output formalisms, as well as the interchange of some of its internal components (e.g., users may exchange the BDD package used in the decision procedure, or the automata minimization and determinization algorithms). The aim is that the best-fitting incarnation of the tool may be put together at need, on an application-driven basis, from the collection of existing components.

Integratability in a heterogeneous analysis and verification environment. Since the MOSEL tool-set is available within the METAFrame repository it is not merely a stand-alone tool: its decision procedure as well as each single component (compilers, interfacing and visualization components) are also available for use in complex, even heterogeneous METAFrame synthesized tools.

The following sections explain in detail the realization of these system requirements, starting with the introduction of the logic layers and their semantics (Sect. 3 to 7), followed by a description of the implementation principles (Sect. 8), and finally by the integration within METAFrame (Sect. 9).

3 The Minimal Logic

The minimal logic is a concrete syntactic version of the monadic second-order logic on finite strings, M2L(Str), using a minimal set of primitives. It serves as the reference logic for MOSEL, 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.

The syntax is shown in Fig. 2. Formulas F are constructed from atomic formulas A by means of the three logical connectives, negation $\neg$, conjunction $\wedge$ and existential quantification $\exists$. There are two types of atomic formulas, the binary inclusion $\subseteq$ and the successor relation $+1$. They can be applied to second-order terms T, which in the minimal logic can only be variables *Id.* This simple syntax is sufficient to derive other standard constructs of M2L(Str). In particular, these include first, second-order, and Boolean terms, quantification over first-order and Boolean variables, and derived logical connectives.

The second-order nature of the minimal logic becomes apparent in the definition of its semantics. Formulas F are interpreted as statements over subsets of natural numbers, which are second-order objects. More precisely, if F is a formula with the free second-order variables $X_1, X_2, \ldots X_k$, then F is a statement about $k+1$ tuples

$$\alpha = (n, M_1, M_2, \ldots, M_k),$$

where n is a natural number and every M_i, for $i = 1, \ldots, k$, is a subset of $\{0, 1, 2, \ldots, n-1\}$. Such an α is called an *F-structure*. We say that α *satisfies* F, or that α is a *model* of F, written $\alpha \models F$, according to the following inductive definition:

$$\begin{array}{lll}
\alpha \models X_i \subseteq X_j & \textit{iff} & M_i \textit{ is a subset of } M_j \\
\alpha \models X_i + 1 = X_j & \textit{iff} & \textit{if for all } 0 \leq x < n-1, x \textit{ in } M_i \textit{ iff } x+1 \textit{ in } M_j \\
\alpha \models \neg F & \textit{iff} & \textit{not } \alpha \models F \\
\alpha \models F_1 \wedge F_2 & \textit{iff} & \alpha \models F_1 \textit{ and } \alpha \models F_2 \\
\alpha \models \exists X_{k+1} : F & \textit{iff} & \textit{there is } M_{k+1} \subseteq \{0, 1, \ldots, n-1\} \textit{ with } \alpha \cdot M_{k+1} \models F,
\end{array}$$

where $\alpha \cdot M_{k+1}$ denotes the extension of α by the new set component M_{k+1}. A formula F is said to be *valid* if it is satisfied by every F-structure. It is *satisfiable*, if it is satisfied by some F-structure, that is, if it has a model. Through the definition of the satisfaction relation the language is interpreted as an ordinary predicate logic, in which the logical connectives are given their standard meaning in terms of truth values.

We associate with every F-structure $(n, M_1, \ldots, M_k)$ a word $a_0\, a_1 \cdots a_{n-1}$ of length n over the alphabet $\{0,1\}^k$, by putting $a_i = (a_{i1}, a_{i2}, \ldots, a_{ik})$ such that $a_{ij} = 1$ iff $i \in M_j$, for all $i \in \{0, 1, \ldots, n-1\}$ and $j \in \{1, 2, \ldots, k\}$. Vice versa, every word $a_0\, a_1 \cdots a_{n-1}$ over $\{0,1\}^k$ corresponds to the F-structure $(n, M_1, M_2, \ldots, M_k)$ where $M_j = \{i \mid a_{ij} = 1\}$. This induces a bijection between the words over the alphabet $\{0,1\}^k$ and F-structures. For instance, the word $(0,0,0)\ (0,1,1)\ (0,0,1)$ corresponds to the structure $(3, \{\}, \{1\}, \{1,2\})$, which satisfies the formula $X_2 \subseteq X_3 \wedge \neg\exists Z : Z \subseteq X_1 \wedge \neg X_1 \subseteq Z$.

From another point of view, the logic can be seen as a programming language for finite state automata. Is F a formula with free variables $X_1, X_2, \ldots X_k$, then we can associate with F a finite automaton $[\![F]\!]$ over the alphabet $A = \{0,1\}^k$. This automaton is constructed so that its language corresponds, via the identification of words with F-structures, to the set of F-structures that validate F. It was the seminal discovery of Büchi [8] that one can use automata-theoretic methods to decide the validity and satisfiability of formulas in monadic second-order arithmetic.

The automaton $[\![F]\!]$ associated with the formula F is obtained by induction on the structure of F. The left automaton seen there represents $[\![X_0 + 1 = X_1]\!]$ while the right one is $[\![X_0 \subseteq X_1]\!]$. The automata are displayed using METAFrame's ffgraphs [11], with transition labels 0 and 1 referring to the variables X_0 and

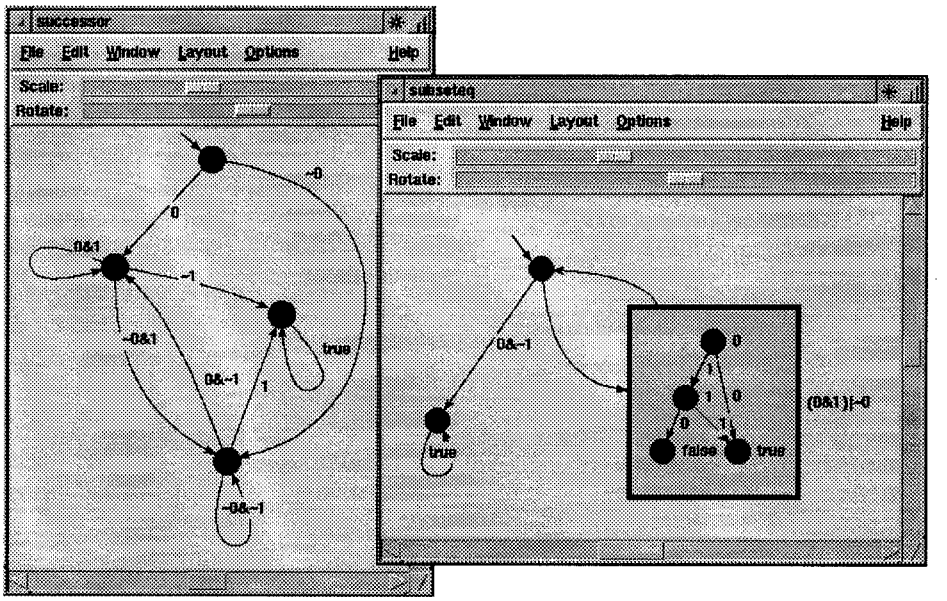

Fig. 3. Automata for Atomic Formulas

X_1, respectively. The top-most node is the start state and the accepting states are shown in grey. In the right-hand automaton $[\![X_0 \subseteq X_1]\!]$ the small window displays the BDD representation of the transition label `(0&1)|~0` for the loop in the start state. The logical connectives, then, amount to specific constructions that build more complex automata from these primitive ones. Concretely, $\neg$ realizes the complement, $\wedge$ the intersection, and $\exists$ the projection of the accepted regular languages. Now, if $[\![F]\!]$ is the automaton obtained from this construction, then we have for every F-structure α,

$$\alpha \in \mathcal{L}\,[\![F]\!] \quad \textit{iff} \quad \alpha \models F,$$

where $\mathcal{L}\,[\![F]\!]$ denotes the regular language accepted by $[\![F]\!]$. This observation is important, since it means we can use automata-theoretic decision procedures for the minimal logic. All extensions of the logical formalism we can build on and encode within the minimal logic, then, are guaranteed to be decidable too. More details on the semantics of monadic second-order logic and its relation to automata are contained in [27,28], which give a comprehensive survey.

4 The Kernel Logic

The advantage of a minimal logic is evident: the definition of the semantics is easier, and proofs of correctness and completeness for richer languages with the same expressive power can often be reduced to reasoning about the more manageable minimal language. The gain in conceptual clarity and confidence in

$$
\begin{array}{lll}
T ::= Id \mid \mathbf{all} \mid \mathbf{empty} \mid T \cup T \mid T \cap T \mid \overline{T} & & \text{Second-Order Terms} \\
A ::= \mathbf{sing}(T) \mid \neg\mathbf{sing}(T) \mid T \not\subset T \mid T \subset T \mid T \not\subseteq T & & \text{Atoms} \\
\quad T \subseteq T \mid T = T \mid T \neq T \mid T + 1 = T \mid T + 1 \neq T & & \\
\quad T < T \mid T \leq T \mid 0 \in T \mid 0 \notin T \mid \$ \in T \mid \$ \notin T & & \\
F ::= \mathbf{true} \mid \mathbf{false} \mid F \wedge F \mid F \vee F \mid F \Rightarrow F \mid F \Leftrightarrow F & & \text{Formulas} \\
\quad F \not\Leftrightarrow F \mid Id(T, \ldots, T) \mid \neg Id(T, \ldots, T) \mid \exists Id, \ldots, Id : F & & \\
\quad \mathbf{let}\ Id(Id, \ldots, Id) = F\ \mathbf{in}\ F \mid \neg\exists Id, \ldots, Id : F \mid A & &
\end{array}
$$

Fig. 4. The Kernel Syntax of MOSEL

the correctness of the concepts contrasts with a loss in implementation efficiency. The drawback of a very skinny language, like the minimal logic, is the fact that nearly all the constructs of a typical user-level language correspond to complex expressions. Since we cannot afford to break down frequently used user-level constructs to the few primitives of the minimal logic, every time they are used, we need to support a reasonable set of derived constructs by direct semantic translations. This means that the language actually implemented in the decision procedure will always be an extension of the minimal logic. This extended logic, called *kernel logic*, is not fixed once and for all, but may be enriched at need.

The current kernel syntax of MOSEL is given in Fig. 4. The semantics of formulas F and second-order terms T is implemented in MOSEL's decision procedure, and the atomic predicates A are precomputed by primitive automata, without going through an encoding via the minimal logic. The consistency of this "semantic bypassing" of the minimal logic has been verified by hand, and, to some extent, using MOSEL itself, as explained in the next section. The rest of this section discusses the most important extensions made in the kernel.

In the first class of syntactic constructs, directly implemented by the kernel for reasons of efficiency, there are a number of frequently used propositional expressions.

Example: Equivalence. The equivalence of formulas ϕ and ψ, written $\phi \Leftrightarrow \psi$, can be expressed in the minimal logic as:

$$F_1 \Leftrightarrow F_2 :\equiv \neg(\neg(F_1 \wedge F_2) \wedge \neg(\neg F_1 \wedge \neg F_2)).$$

This formulation is computationally expensive. As it involves 3 conjunctions and 5 negations, it requires 3 compositions and 5 complementations of automata. However, since, by a design decision, the automata in MOSEL are *deterministic* and *complete*, the automaton $[\![F_1 \Leftrightarrow F_2]\!]$ for the equivalence can be generated cheaply from the automata $[\![F_1]\!]$ and $[\![F_2]\!]$ in only one step.

The encoding of user-level features often requires special-purpose predicates to be introduced in some systematic way, which thus may appear a great number

of times in the translated formula. Given the complexity and the frequency of use, the explicit construction should be avoided and these predicates should feature as atoms in the kernel language.

Example: Singleton Predicate. The encoding of first-order variables in terms of second-order variables involves the predicate $sing(X)$ which expresses that the set denoted by X contains exactly one element. This predicate could be defined, in minimal syntax, as follows:

$$sing(X) :\equiv (\exists Y : Y \subseteq X \wedge \neg X \subseteq Y) \wedge \\ \neg\, \exists Y : Y \subseteq X \wedge \neg X \subseteq Y \wedge \exists Z : \neg Y \subseteq Z.$$

In the kernel the automaton $[\![sing(X)]\!]$ is precomputed, so that $sing(X)$ can be used as an atom.

A third type of useful atoms concerns implicit arithmetic operations that arise from interpreting second-order objects as binary coded natural numbers.

Example: Second-Order Less-Than Predicate. For a natural number n every subset $M \subseteq \{0, 1, \ldots, n-1\}$ may be taken as the n-bit binary encoding of a natural number. With 0 as the least significant bit, the number represented by M is $\sum_{i=0}^{n-1} a_i 2^i$ where $a_i = 1$ if $i \in M$, otherwise $a_i = 0$. When $n = 0$ there is only one subset $M = \emptyset$ which is taken to represent 0. Now, a predicate $X < Y$ is defined so that it holds for a structure (n, M_X, M_Y) if the number represented by M_X is strictly smaller than the number represented by M_Y. In minimal logic this predicate is

$$X < Y :\equiv (\exists K : sing(K) \wedge \neg\, K \subseteq X \wedge K \subseteq Y \wedge \\ \neg\, (\exists T : sing(T) \wedge \neg\, (T \subseteq X \equiv T \subseteq Y) \wedge \\ (\neg\, \exists Z : K \subseteq Z \wedge \exists W : Z + 1 = W \wedge W \subseteq Z \wedge \neg T \subseteq W))). \quad (1)$$

In the case that X and Y denote singletons, $X < Y$ restricts to the standard irreflexive ordering relation $<$ on natural numbers. This makes this predicate very useful for first-order specifications, too, and thus is profitably included in the kernel. The built-in automaton $[\![X < Y]\!]$, visualized in daVinci [10] is seen in Fig. 5. The multi-root reduced ordered BDD of its edge labels is also shown, in a separate window.

Another optimization idea to be mentioned concerns second-order terms. Although it would be possible to restrict the arguments of atomic formulas to second-order variables only, as in the minimal logic, in M2L(Str) usually terms such as union and intersection are allowed too. They need not be eliminated by formulas in the minimal logic, but can be implemented directly, as long as they do not involve the successor function.

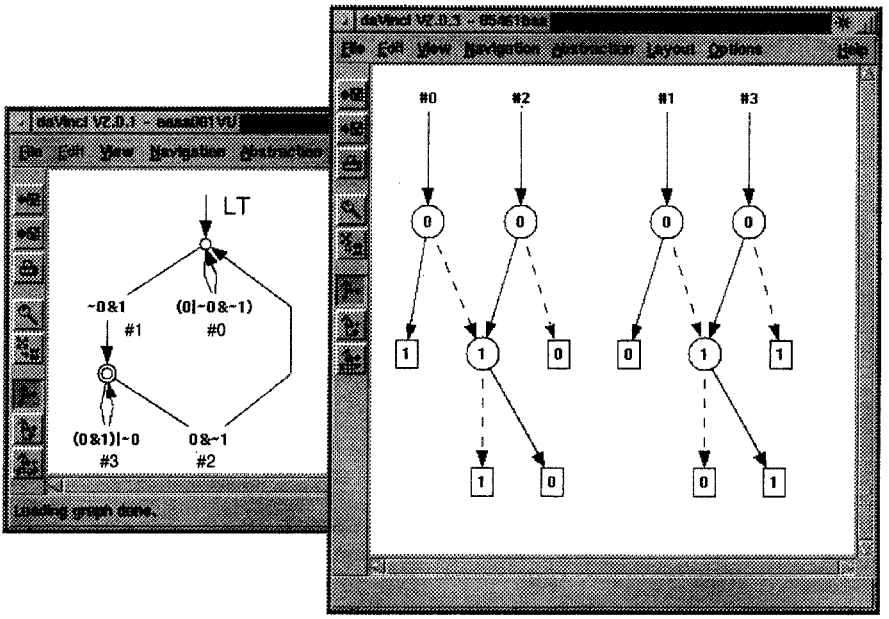

Fig. 5. Automata for $X < Y$

Example: Set Union The set union term $X \cup Y$ may be simulated in the minimal logic by a three-place formula $union(X, Y, Z)$ defined as

$$union(X, Y, Z) :\equiv X \subseteq Z \wedge Y \subseteq Z \wedge \neg\, \exists W : X \subseteq W \wedge Y \subseteq W \wedge \neg Z \subseteq W.$$

However, in a typical instance like $union(X, Y, Z) \wedge F[Z]$ this complicated computation can be avoided by taking account of the expression $X \cup Y$ directly in constructing the automaton for $F[Z]$, provided that we are able to handle complex labels (in this case $X \cup Y$ for Z).

The performance gain achievable by extending the second-order term language to the set shown in Fig. 4 is remarkable: instead of costly (global) operations on automata we need only local operations, on single edge labels. They are much cheaper, because the operators allowed in the complex labels have an efficient implementation in BDD based representations.

Finally, to avoid recomputations of common subexpressions a *let* construct is introduced, which binds formulas to names. Through lazy computation, the compiler ensures that the semantics of a formula is computed only when the formula is needed. The application of a predicate defined by a *let* construct is defined through the copy-rule: the predicate's automaton is copied and the second-order argument variables substituted, taking care of complex labels.

formula	CPU (s)
`X=Y <=> EQ(X,Y)`	.05
`X!=Y <=> NEQ(X,Y)`	.05
`subset(X,Y) <=> SUBSET(X,Y)`	.05
`~subset(X,Y) <=> NOT_SUBSET(X,Y)`	.06
`sing(X) <=> SING(X)).`	.16
`~sing(X) <=> NOT_SING(X)`	.17
`X<Y <=> LT_second(X,Y)`	.65
`X<=Y <=> LEQ_second(X,Y)`	.70
`0 in X <=> ZERO_IN(X)`	.56
`~0 in X <=> ZERO_NOT_IN(X)`	.56
`$ in X <=> DOLLAR_IN(X)`	.58
`~$ in X <=> DOLLAR_NOT_IN(X)`	.58
total	4.17
conjunction of all formulas	1.01

Table 1. Performance of the MOSEL Compiler on the Kernel Syntax Extensions

5 Correctness of the Kernel Logic Extension

To check the consistency of the additional atomic constructs as primitives in the kernel logic with respect to their definition in terms of minimal logic expressions, we have proved in MOSEL that the defined and the computed semantics are indeed equal. We have declared one explicit minimal logic predicate for each additional atomic construct, and have automatically proved in MOSEL the equivalence between those predicates and the built-in kernel logic primitives. Table 1 summarizes the performance of MOSEL on these theorems. For instance, it took 0.65s to verify the equivalence `X<Y <=> LT_second(X,Y)` where `X<Y` refers to the built-in primitive and `LT_second(X,Y)` to its explicit definition (1) in the minimal logic.

We checked each equivalence separately, and also the conjunction of all the equivalences at once, measuring the pure compilation time in seconds of CPU time on a Sparc 20. As expected, the total time required by the separate checks (4.17 s) exceeds by far the time needed to check the conjunction formula (1.01 s). The reason is that many intermediate automata occurring in several subexpressions are computed only once, saving a factor of 4 on the total time.

6 User Logics: The Mona Input Language

The Mona language, close to the form introduced in [14], is given in Fig. 6. It is a reasonably rich extension over the kernel logic that still keeps the generic flavour of a predicate logic, so as to be independent of any specific application domain. It maintains a three-fold type structure of first-order (t), second-order (T), and propositional objects (F), where the latter coincide with the syntactic class of formulas. At these type levels various term constructions are available, in

t ::= *Id* | `0` | `$` | t `+` i | t `-` i	First-Order Terms
T ::= *Id* | `all` | `empty` | `compl` T | T `inter` T | T `union` T | T `+` i | T `-` i	Second-Order Terms
F ::= *Id* | *Id* `(`A`, ... ,`A`)` | t `=` t | t `<` t | t `<=` t | T `=` T | T `sub` T | t `in` T | `~` F | F `&` F | F `|` F | F `=>` F | `All` *Id* `:` F | `Ex` *Id* `:` F	Formulas, or Propositional Terms
A ::= F | T | t	Predicate Arguments
D ::= F | *Id* `(`*Id*`, ...,`*Id*`)` `=` F `;` D	Declarations + Formula

Fig. 6. Basic Syntax for Mona

particular all usual predicate logic connectives at the propositional level. There is a set *Id* of identifiers which contains propositional, first-order, and second-order variables, as well as predicate constants locally declared in a declaration list. It is assumed that the type levels of the variables can be identified uniquely by the name of the quantified variable. Thus, depending on *Id*, the formula `All` *Id*`:`F can be a propositional, first-order, or second-order quantification.

The intuitive meaning of Mona's logic constructs has been presented in [14]. In MOSEL this semantics is implemented by a syntactic encoding into the kernel, making Mona a conservative extension of the kernel logic. There are three main translation steps involved, which are generic and generally useful for defining user logics.

Flattening. Some functions and constants must be replaced by relations, a simple example of which is the elimination of the successor function by the successor relation. For instance, a formula $F[t+1]$ with an occurrence of a first-order subterm $t+1$ is translated into $\exists\, s : s = t+1 \wedge F[s]$, where s is a fresh first-order variable that does not appear anywhere else in F or t. Considering $t+i$ as an abbreviation for the iterated successor function $t+1+1\cdots+1$, the process may be repeated until all occurrences of successor terms $t+i$ have been eliminated, and the only instances of a successor are of the form $x = y+1$ with x, y variables. A similar flattening is applied to occurrences of iterated second-order successors $F[T+i]$, and predecessors $F[t-i]$, $F[T-i]$. The first-order constants 0 and \$ are broken out in the same fashion, except if the 0 or \$ in $F[\$]$ or $F[0]$ occur in a subformula $\$ \in T$, $0 \in T$, which are implemented directly in the kernel. Otherwise, for instance, $F[\$]$ becomes $\exists s : s = \$ \wedge F[s]$. Another goal of the flattening is to break out formula arguments in predicate applications. For instance, if predicates $P(a,b)$, $Q(c)$ are declared with propositional variables a, b, c, then these may be instantiated with formula arguments as in $P(X \subseteq Y, Q(\$ \in Z))$. This subformula can be replaced by $\exists a, b, c : (a \Leftrightarrow X \subseteq Y) \wedge (b \Leftrightarrow Q(c)) \wedge (c \Leftrightarrow \$ \in Z) \wedge P(a,b)$. A logically equivalent but more efficient flattening is **let** $R(X,Y) = X \subseteq Y$ **in let** $S(Z) = \$ \in Z$ **in** $P[R(X,Y), Q[S(Z)]]$, where $P[\_,\_]$ and $Q[\_]$ represent the syntactic expansions of the definitions of P and Q.

Type Embedding. Propositional as well as first-order variables and quantifiers are encoded within the second-order fragment. Formally, every propositional variable $a, b, c, \ldots$ can be represented by some uniquely associated second-order variable $A, B, C, \ldots$ and every first-order variable $x, y, z, \ldots$ is represented by some associated second-order variable $X, Y, Z, \ldots$ The way these representation variables are introduced differs in both cases, following different semantic encoding techniques.

The idea in the propositional case is to represent the truth value of (the object referred to by variable) a by the last bit of (the bit sequence referred to by) the associated A. This divides the second-order objects into two equivalence classes, one representing *true* and the other representing *false*. The object A belongs to the *true* class if the predicate $true(A) := \$ \in A$ holds[2]. The predicate $true(A)$ is introduced to replace every occurrence of the associated propositional variable a as a subformula. To give an example, $\exists a, b : (a \Leftrightarrow X \subseteq Y) \wedge (b \Leftrightarrow Q(c)) \wedge P(a, b)$ is translated into $\exists A, B : (true(A) \Leftrightarrow X \subseteq Y) \wedge (true(B) \Leftrightarrow Q(c) \wedge P(A, B)$. Another method is to eliminate propositional quantifiers altogether, e.g. by replacing $\exists a : \phi$ by $\phi\{true/a\} \vee \phi\{false/a\}$, where *true* and *false* are some canonical choice for the propositional truth values. Which of the possibilities is more efficient depends on the particular case.

First-order objects are encoded by singleton subsets [27]. Technically speaking, we may say that the first-order type is treated as a *subtype* of the second-order type, whereas propositions are a *quotient*. Again, the representation variables $X, Y, Z, \ldots$ are substituted systematically for all occurrences of $x, y, z, \ldots$, but now some adjustment has to be made to the quantifiers. We replace $\exists x : F$ by $\exists X : sing(X) \wedge F$ and $\forall x : F$ by $\forall X : sing(X) \Rightarrow F$, where $sing(X)$ is the singleton predicate stating that the set denoted by X consists of exactly one element. Apart from translating variables and quantifiers we also need to replace the atomic predicates $t \in X$, $t = 0$, $t = \$$, by $T \subseteq X$, $0 \in T$, $\$ \in T$, respectively.

Normalization. We produce an equivalent positive normal form by standard methods. Thus, universal quantifiers $\forall$ are replaced by $\neg\exists\neg$, DeMorgan's laws are applied to push negation inwards, and double negations are removed.

Flattening, type embedding, and normalization, are only three examples of many different translation steps that may be necessary to compile a user logic formula into a well-formed formula of the kernel logic. This process is non-trivial and requires special care to ensure the soundness and efficiency of the decision procedure. In general, soundness of the compilation from a user logic into the kernel logic requires additional constraints. Logically speaking, we translate into a special theory of the kernel logic, rather than into the kernel logic itself. In the case of Mona, for instance, the standard semantics of the first-order constructs is not preserved by the translation unless the constraint $\exists X : \exists Y : X \neq Y$ elim-

[2] Alternatively, any other formula that partitions the second-order objects into two classes may be used instead.

inating empty models (strings) is added[3]. Concerning efficiency it is important not to hard-wire one fixed translation strategy, but to support a combination of logically equivalent methods, so that for a particular application or formula the optimal solution can be assembled.

7 Application Logics: Modelling Hardware in Mona-HW

Hardware primitives for sequential circuits, like e.g. elementary gates and memory elements (say D-type flip-flops) have been defined as predicate macros on top of Mona. An example of a fairly complete library can be found in [18]. Building on these predicates, higher abstraction levels can be captured too, so that more complex predicates represent entire circuits or families of circuits.

The expressive power of M2L(Str) captures only one-dimensional structures (linearly or circularly arranged). This is due to the interpretation of the logic over strings, which implies that the parameterization allowed to express generalized behaviours is limited to the generic "length" of strings. Since strings may be taken to assume different meanings (in [16,17] sampled waveforms for control circuits, in [18] the bitwidth of a datapath), a degree of freedom in the use of the logic is still left to the application designer.

The M2L(Str) logic and its fully automatic decision procedure have allowed us to capture, in a common framework, a wide spectrum of abstraction levels, ranging from generic architecture or protocol levels [17] to the hardware-oriented register transfer and gate levels [16,18]. In particular, both behavioural and structural description styles are supported, and from both it is possible to carry out model-based analysis, verification, and error detection. Moreover, register-transfer and gate level circuits can be automatically obtained from the models with current synthesis techniques.

The fully automatic treatment of relevant classes of parametric circuits offered by the M2L(Str) logic is a central feature for the practicability of the method in an industrial environment: only push-button techniques are in fact widely acceptable by hardware designers. Moreover, user interaction must be possible completely within the application level. We envisage a hierarchy of application level formalisms the syntax of which may coincide with decidable subsets of several widespread HDLs. Adequate candidates for automatic translation are e.g. register-transfer and gate level subsets of VHDL and of Verilog, which could be dealt with on the basis of our library of M2L(Str) predicates.[4] But also the full BLIFF language, a standard language for the definition of netlists and gate-level components has a semantics contained in M2L(Str), and its translation into MOSEL kernel logic has already been started.

Similarly, it is important to be able to visualize results in a standard way: in our case, hardware designers and design tools are accustomed to design and manipulate finite state automata, which are therefore one of MOSEL's input/output

[3] The absence of this constraint in Version 0.2 of Mona system caused some confusion concerning the actual semantics of formulas.

[4] Information is available at `http://brahms.fmi.uni-passau.de/bs/projects/`.

formalisms, and are used to BDD representations of Boolean functions, which are also generated and visualized by the system.

8 Implementation

The current implementation of MOSEL, which is object-oriented and programmed in C++, features three groups of components: *decision procedures*, *translators* between different logics, and graphical *visualization* modules.

Semantic Decision Procedures. The decision procedure Mosel_dpsf is implemented as a compiler that transforms the abstract syntax tree derived by translation from a kernel logic input formula into the corresponding semantic model, which is a finite state automaton:

$$\text{Mosel\_dpsf} : \text{abstract syntax tree} \longrightarrow \text{automaton}.$$

Abstract syntax tree and automata are implemented as C++ classes, each construct's specific compilation is implemented as a method of the abstract syntax tree class and the resulting automaton is an object of the automaton class. Following other hardware verification and synthesis tools, our decision procedure, thus, is largely independent from the concrete input syntax, since it works directly on an object-oriented abstract syntax tree.

Logic Translations. In order to couple this compilation kernel with the environment, a series of interfacing modules translate each of the several logics into the abstract syntax trees accepted by the actual compiler. Concretely, they cooperate to transform an ASCII representation of logic formulas into an object of the abstract syntax class.

Graphical Visualization. Several alternative graphic tools can be used to display the automata generated within MOSEL. As in Mona, it is possible to use the daVinci [10] tool to show the structure of the automata as well as the concrete BDD encoding of the edge labels. An example is given in Fig. 5. Moreover, the bidirectional link of MOSEL's automata to METAFrame's ffgraphs library [11] makes it possible to read and generate automata not only for display, but also to result from or to be fed into other algorithms and tools of the METAFrame environment. This way graphs are not a pure visualization commodity, but an alternative import/export mechanism, crucial for the cooperation between heterogeneous tools.

Additional powerful graph manipulation features of METAFrame like the window-in-window browser shown in Fig. 3 (right) are now accessible (see also [4]). As an example, by clicking on an edge of the graph, the representation of its label as BDD is shown locally, but in a separate virtual window which can be moved, edited, and steered through the menu bars and commands of the outer window. The graphical display of the automata is implemented by a method of the automata class.

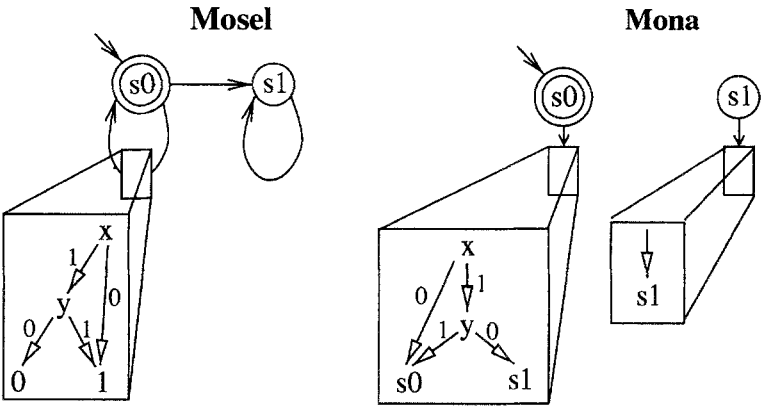

Fig. 7. Automata Representation in MOSEL and in Mona

Implementation of the Automata

Our choice of implementation for the automata is driven by considerations of memory efficiency and efficiency of the operations on the chosen data structure.

Automata Representation. The automata generated from M2L(Str) formulas are deterministic and complete. Both properties are essential for an efficient implementation of *negation.*

The size of the edge labels' alphabet being exponential in the number of used second-order variables, an explicit enumeration of the letters is excluded. Rather, edge labels are represented by Boolean functions characterizing the transitions, which in turn are implemented via BDD techniques [1,6,7,23]. The solution chosen here, as already for Mona, is a *hybrid representation* of the automata: graphs with edges encoded as Boolean functions and implemented via BDDs. This is widely sufficient if the complexity of the automata is mainly due to complex edge labels, as in our application, rather than to intricate structure.

Data Structure. An automaton has exactly one pointer to the start node. Each state of the automaton is a node of the graph. Each node has a flag to indicate whether it is a final state and a flag used during recursive traversal of the graph. Moreover, each node has a pointer to a list with an element for each outgoing edge. Each edge is represented by a pointer to the reduced ordered BDD representing its label, and a pointer to the corresponding successor node.

At the data structure level MOSEL and Mona differ: as visualized in Fig. 7 on a small example, instead of reduced ordered BDDs, Mona uses a kind of BDDs with multiterminal nodes. The edge label function, thus, is not available separately for each successor state, with transparent transitions between the nodes as in MOSEL, but for each node there is only one BDD whose terminals are the successor states, enumerated from 1 to n, and the graph is a collection of such isolated portions.

Our reason for not choosing this solution is that it is only efficient (i.e., admits sharing of BDDs) if there are several nodes whose *incoming* edges are identically labelled. Otherwise, MOSEL's solution is preferable, since *all* common subexpressions can be shared, independently of the state the edges lead to (this kind of sharing is in fact more likely to happen), and the graph structure is better preserved.

Automata Minimization Algorithm. Of the three nontrivial operations on automata, the *composition* and the *determinization* via powerset construction can be implemented in the standard way. The efficient $n \log n$ algorithms for *minimization* [12,22] cannot be used directly, since they presuppose the alphabet to be relatively small wrt. the automaton, so that the algorithm can be applied successively for all the letters of the alphabet. Since this is not feasible in our case, we have developed a generalization of the method to BDD-labeled transitions.

9 Embedding MOSEL in METAFrame

From a system-level point view, MOSEL is applicable in a broad spectrum of scenarios. Not only can each of its components be used as a stand-alone tool, it can also be used as a component in the construction of other complex heterogeneous analysis or verification tools within METAFrame. METAFrame [25] is an environment designed to support the *systematic* and *structured* computer aided generation, analysis, verification, and testing of application-specific complex systems from collections (repositories) of reusable components. In particular, it offers a *large grain* synthesis approach through its synthesis component [26].

To this end we have extended METAFrame's repository of available tools by checking in the Mosel_dpfs decision procedure and each of the new logic translation and output processing modules.

Example: Synthesis of a generic M2L(Str) checker in METAFrame

We show on a simple but concrete example how the user can synthesize with METAFrame a system for checking the correctness of M2L(Str) formulas.

Specification. The problem is informally described as

Display on screen the result of checking a formula given in Mona *syntax*

and `.mona, finally screen` is the corresponding input formula to the synthesis component. Since each module is interpreted as a transformer from its input formats into its output formats, characterized by means of their actual file extensions (e.g. module latex transforms a .tex input file into a .dvi output file), this formula means that, starting with an input contained in a .mona file, we want to reach eventually, possibly after an unspecified number of transformations, a display on a *screen*.

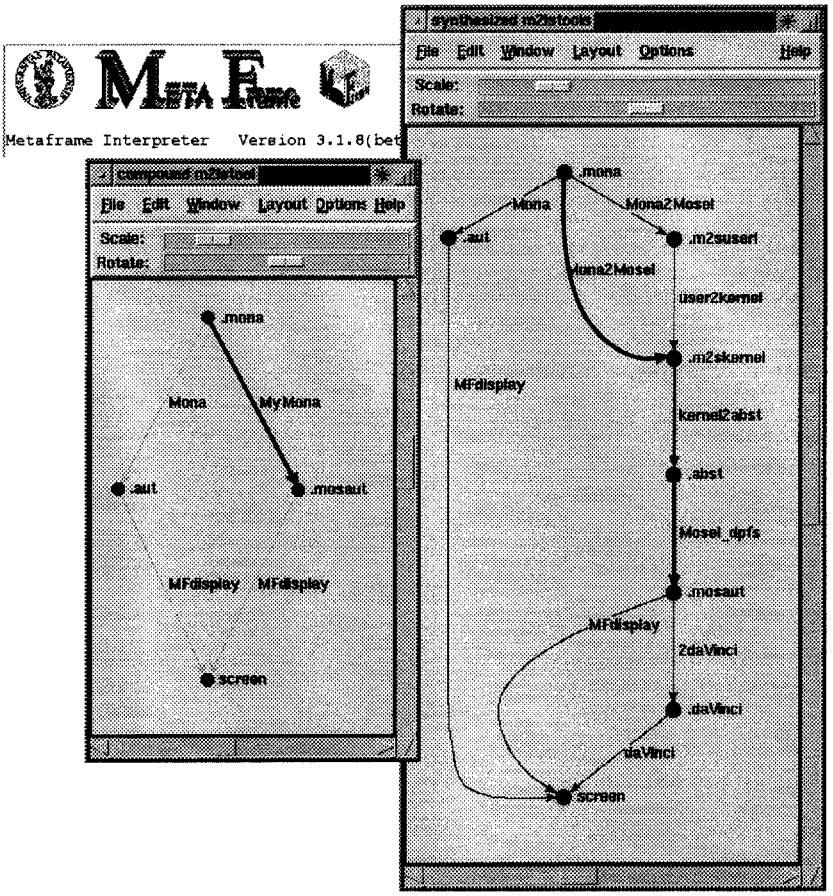

Fig. 8. Synthesizing M2L(Str) Tools in METAFrame

Synthesis. Given this specification, METAFrame provides the graph of Figure 8 which represents the set of all satisfying tool-compositions which can be built on the basis of the underlying repository. Every path from the start to the success node in the graph of Figure 8 (right) is a possible solution to our synthesis problem. A path represents a heterogeneous program whose transitions represent modules and whose nodes are labelled by intermediate types (here the file extensions).

While Mona offers only a single monolithic solution, as shown by the leftmost path, several alternatives are offered by the MOSEL tool-set: though sharing the Mosel_dpfs decision procedure, a flexible series of converters (even polymorphic, as in the case of Mona2Mosel) allows for alternatives. The display capabilities are enhanced too. Note that all the solutions contain (necessary) modules which are not mentioned in the specification, but are filled up at need by the synthesis algorithm.

Execution and Compilation. A direct execution of the chosen tool compositions is possible via interpretation of the corresponding program. Satisfactory solutions can be made persistent through compilation into a new module that can be saved in the component repository for later reuse. For example, it is possible to store the partial solution corresponding to the tool composition

$$\mathsf{Mona2Mosel} \rightarrow \mathsf{kernel2abst} \rightarrow \mathsf{Mosel\_dpfs}$$

highlighted with the thick line in Fig. 8 (right) to yield a MOSEL-based Mona simulator called MyMona.

Running the query a second time, and asking only for minimal length solution paths, we obtain the solution graph of Fig. 8 (left).

10 Future Work

Future work will follow several threads. As an alternative to the Mona user logic we plan to implement a typed predicate logic with facilities for user-specific extensions. At the application layer we are working on the embedding of standard HDL languages, like BLIFF. Furthermore, future extensions of MOSEL will support other semantic theories like finite trees, or finite sets.

The construction being still under way, it is too early to give a detailed assessment of the efficiency of MOSEL. Our first experiments show that our system outperforms Mona at the kernel level but that it is slower at the level of Mona syntax. Thus, in the further development of the system the focus will be primarily on improving the compilation algorithms rather than the basic decision procedures.

References

1. S. Akers: "*Binary Decision Diagrams*," IEEE Trans. on Comp. Vol.C-27, 1978, pp. 509-516.
2. D. Basin, N. Klarlund: "*Hardware verification using monadic second-order logic*," Proc. CAV '95, Liège (B), July 1995, LNCS N. 939, Springer Verlag, pp. 31-41.
3. N. Bjørner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. Sipma, T. Uribe: "*STeP: Deductive-algorithmic verification of reactive and real-time systems*," Proc. CAV'96, New Brunswick, NJ (USA), Aug. 1996, LNCS N. 1102, Springer Verlag, pp. 415-418.
4. M. von der Beeck, V. Braun, A. Claßen, A. Dannecker, C. Friedrich, D. Koschützki, T. Margaria, F. Schreiber and B. Steffen: "*Graphs in* METAFrame*: The Unifying Power of Polymorphism*," Proc. TACAS'97, Enschede (NL), April 1997, in this same Volume of LNCS, Springer.
5. J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang: "Symbolic model checking: 10^{20} states and beyond," Proc. LICS'90, Philadelphia, 1990, pp. 428-439.
6. R.E. Bryant: "*Graph-based algorithms for Boolean function manipulation*," IEEE Trans. Computing, vol. C-35(8), August 1986, pp. 677-691.
7. R. E. Bryant: "*Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*," ACM Computing Surveys Vol. 4, 1992, pp. 293-318.

8. J.R. Büchi: "*Weak second-order arithmetic and finite automata*," Z. Math. Logik Grundl. Math., Vol. 6, 1960, pp. 66-92.
9. A. Church: "*Logic, arithmetic and automata*," Proc. Int. Congr. Math., Almqvist and Wiksells, Uppsala 1963, pp. 23-35.
10. daVinci: the tool is available via ftp at site `ftp://ftp.uni-bremen.de/pub/graphics/daVinci`
11. C. Friedrich: "*The* ffgraph *library*," Techn. Rep. MIP-9520, Fakultät für Mathematik und Informatik, Universität Passau, December 1995.
12. J. Hopcroft: "*An n log n algorithm for minimizing states in a finite automaton*," Proc. Int. Symp. on Theory of Machines and Computations, Technion, Haifa (IL), Aug. 1971, pp.189-196.
13. P. Kelb: "*Abstraktionstechniken für automatische Verifikationsmethoden*," Ph.D. Diss., Univ. of Oldenburg (Germany), Dec. 1995, Shaker Verlag, Aachen (D).
14. J. Henriksen, J. Jensen, M. Jørgensen N. Klarlund, R. Paige, T. Rauhe, A. Sandholm: "*Mona: Monadic second-order logic in practice*," Proc. TACAS'95, Århus (DK), May 1995, LNCS 1019, Springer V., pp. 89-110.
15. N. Klarlund, M. Nielsen, K. Sunesen: "*A Case Study in Verification Based on Trace Abstractions*," in M. Broy, S. Merz, K. Spies (eds.), *Formal Systems Verification - The RPC-Memory Specification Case Study*, LNCS N. 1169, Springer V., Nov. 1996.
16. T. Margaria, M. Mendler: "*Automatic Treatment of Sequential Circuits in Second-Order Monadic Logic*", 4th GI/ITG/GME Worksh. on *Methoden des Entwurfs und der Verifikation digitaler Systeme*, Kreischa (D), March 1996, Shaker Verlag.
17. T. Margaria, M. Mendler: "*Model-based Automatic Synthesis and Analysis in Second-Order Monadic Logic*," Proc. AAS'97, ACM/SIGPLAN Int. Worksh. on Automated Analysis of Software, Paris (F), Jan. 1997, pp.99-112.
18. T. Margaria: "Fully Automatic Verification and Error Detection for Parameterized Iterative Sequential Circuits", Proc. TACAS'96, Passau (D), March 1996, LNCS N.1055, Springer Verlag, pp. 258-277.
19. T. Margaria: "*Verification of Systolic Arrays in M2L(Str)*," Techn. Rep. MIP-9613, Fakultät für Mathematik und Informatik, Universität Passau, July 1996.
20. O. Matz, A. Miller, A. Potthoff, W. Thomas, E. Valkema: "*Report on the Program AMoRE*", Techn. Rep. Nr. 9507, Inst. für Informatik und Praktische Mathematik, Universität Kiel (D), 1995.
21. J.K. Ousterhout: "*Tcl and the Tk Toolkit*," Addison–Wesley, April 1994.
22. R. Paige, R. Tarjan: "*Three partition refinement algorithms*," SIAM Journ. of Computation, Vol.16, N.6, Dec. 1987, pp.973-989.
23. K. S. Brace, R. L. Rudell, R. E. Bryant: "*Efficient Implementation of a BDD Package*," Proc. DAC'90, Orlando, FL, June 1990, pp. 40-45.
24. B. Steffen, T. Margaria, A. Claßen, V. Braun: "*Incremental Formalization: A Key to Industrial Success* ", In "SOFTWARE: Concepts and Tools", Vol. 17, No 2, pp. 78-91, Springer Verlag, July 1996.
25. B. Steffen, T. Margaria, A. Claßen, V. Braun: "*The* METAFrame *'95 Environment*", (Experience Report for the Industry Day), Proc. CAV'96, Juli-Aug. 1996, New Brunswick, NJ, USA, LNCS N.1102, Springer Verlag, pp.450-453.
26. B. Steffen, T. Margaria, A. Claßen: "*Heterogeneous Analysis and Verification for Distributed Systems*", In "SOFTWARE: Concepts and Tools", vol. 17, N.1, pp. 13-25, Springer Verlag, 1996.
27. W. Thomas: "*Automata on infinite objects*," In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, p. 133–191. MIT Press/Elsevier, 1990.
28. W. Thomas: "*Languages, automata, and objects*," to appear in the forthcoming new edition of the *Handbook of Theoretical Computer Science*, MIT Press/Elsevier.