Electronic Communications of the EASST
Volume 30 (2010)

International Colloquium on Graph and Model Transformation On the occasion of the 65th birthday of Hartmut Ehrig (GraMoT 2010)

Second-Order Value Numbering

Tiziana Margaria, Bernhard Steffen and Christian Topnik

15 pages

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Second-Order Value Numbering

Tiziana Margaria[1], Bernhard Steffen[2] and Christian Topnik[2]

Chair Service and Software Engineering, University of Potsdam
[1]margaria@cs.uni-potsdam.de,
Chair Programming Systems, TU Dortmund
[2]steffen@cs.uni-dortmund.de

Abstract: We present second-order value numbering, a new optimization method for suppressing redundancy, in a version tailored to the application for optimizing the decision procedure of jMosel, a verification tool set for monadic second-order logic on strings (M2L(Str)). The method extends the well-known concept of value numbering to consider not merely *values*, but *semantics transformers* that can be efficiently pre-computed and help to avoid redundancy at the 2nd-order level. Since decision procedures for M2L are non-elementary, an optimization method like this can have a great impact on the execution time, even though our decision procedure comprises the analysis and optimization time for second-order value numbering. This is illustrated considering a parametric family of hardware circuits, where we observed a performance gain by a factor of 3. This result is surprising, as the design of these circuits exploits already structural similarity.

Keywords: Program Analysis and Optimization, Monadic Second-Order Logic, (second-order) Value Numbering

1 Introduction

Value numbering is a well-known compiler optimization technique used to efficiently detect and eliminate redundant code by identifying equality of values [CS70, AWZ88]. Considering this (first order) concept there is a natural generalization to second-order (or even higher-order in general): rather than considering just values, one could lift the analysis to second-order by considering *semantics transformers*, which may then be efficiently pre-computed and help to avoid redundancy at the second-order level.

In this paper we introduce second-order value numbering and illustrate its impact by applying it to improve the decision procedure of jMosel [TWMS06], a verification toolset for monadic second-order logic on strings (M2L(Str)). M2L [Chu63] is an extremely expressive specification language with a non-elementary decision procedure. This makes jMosel a good candidate for our new optimization technique, as there is room even for ambitious optimizations due to the huge leverage potential. Our experiments support this judgement: we observed a performance gain of a factor of three when analyzing a parametric family of hardware circuits, despite the fact that the optimized decision procedure includes the analysis and optimization time for 2nd-order value numbering as well. This result is surprising, as the design of these circuits exploits already structural similarity. - Please note that our technique is quite general, and not restricted to the considered application domain.

This paper is organized as follows: Section 2 provides an introduction to the jMosel toolset including the definition of its syntax and semantics. First-order value numbering in the jMosel context is explained in Section 3, while Section 4 introduces second-order value numbering together with a detailed discussion of a minimal example. Subsequently, Section 5 illustrates our new method along a realistic case study, before we conclude with Section 6.

2 jMosel

jMosel is a toolset for M2L(Str) that computes the semantics of a formula in terms of a finite state automaton. In this sense, it can be seen as a compiler from this logic into automata models. A detailed presentation of the tool can be found e.g. in [TWMS06]. Its underlying concepts and the predecessor MoSeL have been presented in [Mar96, KMMG97]. The following subsections summarize the required background about jMosel and M2L.

2.1 Syntax

jMosel's several user-level logics are built on top of the following *Minimal Logic*, which already provides the full expressive power of M2L(Str):

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

In this BNF, the non-terminal `T` denotes 2nd-order terms in form of (2nd-order) variables `Id`. Atomic predicates `A` allow comparisons in terms of subset relation and equality after bit-shifting. jMosel's minimal logic formulas, denoted by the non-terminal and start symbol `F`, may be constructed using the standard operators of (a minimal) first-order logic. For convenience, we will later also use the usual derived operators like disjunction (here written $|$), implication and logical equivalence.

2.2 Semantics

In M2L(Str) formulas are interpreted as sets of (ordered) positions in a string of arbitrary, but finite length, which can be conveniently described as finite bitvectors, i.e. a finite word over the alphabet $\{0,1\}$. One often refers to the interpretation of these bitvectors as characteristic functions that describe subsets of a given ordered set. Typical is their interpretation as finite set of natural numbers, illustrated in Figure 1.

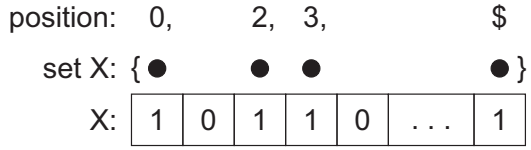


Figure 1: A set of positions `set X` and the corresponding bit vector `X`.

The bit vector corresponding to a string variable X has value 1 at position n iff $n \in \mathbb{N}_0$ is included in X, and value 0 otherwise. The figure shows a set X containing the positions 0, 2, 3 and \$, its representation as a characteristic set, and its corresponding bit vector. Here, the symbol \$ stands for the last position in the parametric string, and therefore marks the last bit in the bit vector; a special symbol for this last position is necessary since M2L(Str) allows reasoning about strings of finite but *arbitrary* length, a convenient model for parametric hardware components.

The following development will entirely foot on the bit vector interpretation of M2L(Str), which we formally define below.

Semantics of jMosel formulas

jMosel translates formulas into complete and deterministic finite automata (DFA) in such a way that the language recognized by one such automaton corresponds to the formula's interpretation as a bit vector. Accordingly, the semantics of a formula is defined via the function $[\![\ ]\!] : \varphi \longrightarrow \alpha$, where φ is the set of all jMosel formulas and α is the set of all complete DFAs.

Definition 1 (Boolean Automaton)
A Boolean Automaton A of α is defined as $A = (\Sigma, S, s_0, F, \delta)$, where

- Σ is the set of all edge labels, which themselves denote subsets of the set of free variable V in the considered formula. They are represented as bitvectors of length $|V|$.
- S is the set of all states.
- s_0 is the initial state, $s_0 \in S$.
- F is the set of accepting states $F \subseteq S$.
- δ is the transition function defined as $\delta : S \times \Sigma \longrightarrow S$.

The edge labels determine for every string variable the Boolean value at position n, whenever this label is taken as nth step of an accepting run. The number of edge labels is exponential in the size of the formula's free variables, since the value of every variable $v \in V$ has to be checked for equality with 0 or 1. Therefore, each label consists of a bit vector of length $|V|$.

Boolean Automata typically have very many edges between two nodes. We therefore construct the following equivalent Symbolic Automaton $\mathscr{A}_s$, whose edges are labelled with Boolean functions and therefore compactly represent a set of edges of the original automaton.

Definition 2 (Symbolic Automaton)
A symbolic automaton $\mathscr{A}_s$ is defined as $\mathscr{A}_s = (\mathscr{L}, \mathscr{S}, s_0, \mathscr{F}, \delta)$, where

- $\mathscr{L}$ is the set of all possible edge labels, consisting of Boolean functions.
- $\mathscr{S}$ is the set of all states.
- s_0 is the initial state, $s_0 \in \mathscr{S}$.
- $\mathscr{F}$ is the set of all accepting states, $\mathscr{F} \subseteq \mathscr{S}$.

T \ t	0	1	2	…	$\$-1$	$\$$
X	x_0	x_1	x_2	…	$x_{\$-1}$	$x_\$$
Y	y_0	y_1	y_2	…	$y_{\$-1}$	$y_\$$
⋮	⋮	⋮	⋮	⋱	⋮	⋮
Z	z_0	z_1	z_2	…	$z_{\$-1}$	$z_\$$

Figure 2: Values for 2nd-order variables

T \ t	…	i	…
W	…	0	…
X	…	1	…
Y	…	0	…
Z	…	1	…

Figure 3: Representation of edge label `~w & x & ~y & z`

- δ is the transition function defined as $\delta : \mathscr{S} \times \mathscr{L} \longrightarrow \mathscr{S}$.

To describe the transformation from A to $\mathscr{A}_s$, we observe that values for a jMosel formula's 2nd-order variables can be represented in table form, where a variable X is expressed as bit vector with position literals $x_0, x_1, ..., x_{\$-1}, x_\$$. The ordering of variables is arbitrary but fixed.

A *row* for a variable X in Fig. 2 represents a word of the language $[\![X]\!]$. Every column of the table specifies one input symbol of A and must therefore match an appropriate edge label. The position j in this label corresponds to the variable at position j in the ordering of variables. To convert A into an Automaton $\mathscr{A}_s$ with Boolean formulas as edge labels, the labels of A are first transformed as shown in Fig. 3. Subsequently, edges sharing the same source and target state are merged; the resulting edge is labelled with the disjunction of the merged edges' labels.

The formulas for the edge labels resulting from this transformation may be large, but their BDD representations are canonical and typically nice and concise [Bry86]. The jMosel toolset supports various BDD libraries to optimally exploit this observation.

Semantic Completeness:
Note that a symbolic automaton $\mathscr{A}_s$ composed this way is typically not complete: its input alphabet consists of all Boolean functions, but not every state considers the input of every possible Boolean function. However, the automaton is complete *at a semantical level*: the automaton A with bit vector labels it represents is always complete. It is this semantic notion of completeness and determinism which we will refer to in the sequel of the paper.

Convention:
In the following sections, the semantics of jMosel formulas will always be given in terms of symbolic automata $\mathscr{A}_s$. In this section we used the index "s" to better distinguish between the two types of automata, but we omit it from now on. In the figures depicting automata, the

following applies: an arrow marks the initial state, accepting states are denoted as double circles, non-accepting states as plain circles.

In the following, we first present the classical (first-order) value numbering for this application domain, before we lift to second order in Section .

3 First-Order Value Numbering

First-order value numbering is an analysis method that allows the detection and removal of redundant computations from a program [CS70]. This goal is achieved by assigning abstract identification values to computations that imply equality: as soon as an identification value reappears, it is certain that the corresponding computation has been already performed before, thus the previously computed result may be reused instead of performing the computation again. This 'classical' optimization is called DAGification in [KMS02].

3.1 Characterization of 1st-order Value Numbering

Given a syntax tree T of a jMosel formula in terms of

- $\mathscr{L}$ is the set of all labels for predicates, operators, and variables,
  $\mathscr{L} = \{\texttt{subseteq}, \texttt{shifteq}, \tilde{}, \texttt{\&}, \texttt{ex}\} \cup \{X, Y, Z, ...\}$
- $\mathscr{N}$ is the set of nodes of the syntax tree T under consideration
- $l : \mathscr{N} \longrightarrow \mathscr{L}$ maps every syntax node to its label.

the assignment of abstract identification values can be given by any function $v_{1st} : \mathscr{N} \longrightarrow \mathbb{N}$ that satisfies the following two characteristics:

For all nodes $n_1, n_2 \in \mathscr{N}$ of the syntax tree,

- $v_{1st}(n_1) = v_{1st}(n_2)$ implies $l(n_1) = l(n_2)$,
  i.e. the coincidence of their syntactic labels. In addition we require
- if n_1 and n_2 are internal nodes with children $c_1^1, ..., c_i^1 \in \mathscr{N}$ and $c_1^2, ..., c_j^2 \in \mathscr{N}$, respectively
  $i = j \wedge \forall k \in \{1, ..., i\} . v_{1st}(c_k^1) = v_{1st}(c_k^2)$

3.2 Example

As an example for the process of first-order value numbering, we consider the following jMosel formula:

$F =$ `(subseteq(X,Y)&shifteq(A,B))|(subseteq(X,Y)&shifteq(A,B))`

Fig. 4 shows its syntax tree after computation of the value numbers.

At compilation, the compiler can benefit from the fact that the subformulas with value numbers 3, 6, and 7 all occur twice by only calculating each of them once, storing the result of the computation, and referring to it when the corresponding value number occurs for the second time. We will illustrate the impact of this optimization in Section 5.

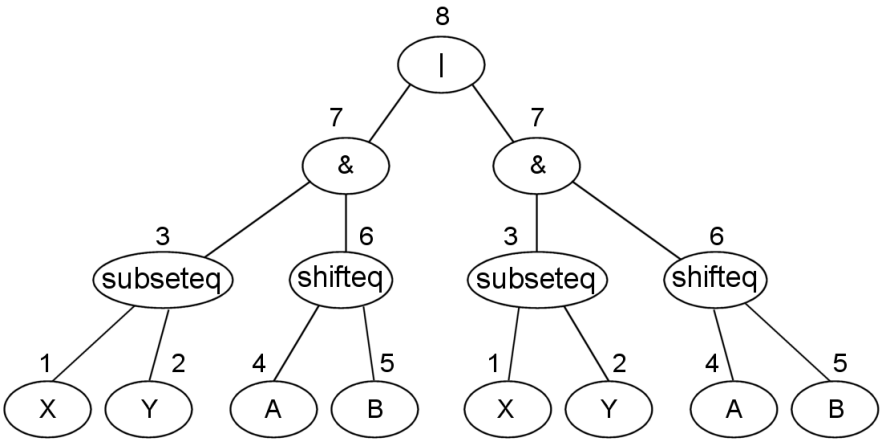


Figure 4: 1st order value numbering applied to the jMosel formula F.

4 Second-Order Value Numbering

While first-order value numbering is used to identify redundant computations and replacing them by previously computed results, the goal of second-order value numbering is to identify redundant *transformations* - the reason for the use of second-order here. As will be clear below, this analysis and its corresponding optimizations is only a bit more involved than in the first-order case, but has a far bigger impact, see Section 5.

4.1 Characterization of 2nd-order Value Numbering

The only difference in the characterization of the labelling function $v_{1st} : \mathcal{N} \longrightarrow \mathbb{N}$ concerns the treatment of atomic predicates, i.e., of $\mathscr{A} = \{\texttt{subseteq}, \texttt{shifteq}\}$. Their labelling does no longer require the second clause for internal nodes. This results in the following slightly modified characterization:

For all nodes $n_1, n_2 \in \mathcal{N}$ of the syntax tree,

- $v_{1st}(n_1) = v_{1st}(n_2)$ implies $l(n_1) = l(n_2)$,
  i.e. the coincidence of their syntactic labels. In addition we require
- if n_1 and n_2 are internal nodes with children $c_1^1, \ldots, c_i^1 \in \mathcal{N}$ and $c_1^2, \ldots, c_j^2 \in \mathcal{N}$, respectively $i = j \quad \wedge \quad \forall k \in \{1, \ldots, i\}.v_{1st}(c_k^1) = v_{1st}(c_k^2)$ unless they are labelled with $\mathscr{A} = \{\texttt{subseteq}, \texttt{shifteq}\}$

After this labelling, nodes sharing the same value number can be replaced by calls to a semantics transformer. However note that transformers should only be created for subtrees containing at least one logical operator, as otherwise the effect of the transformation is vacuous.

4.2 Semantics Transformers

When implementing second-order value numbering for jMosel, the semantics transformers can be implemented in terms of custom predicates similar to the atomic formulas `subseteq` and `shifteq`. This means that every identification of redundancy results in the automatic definition of a custom predicate. This process can be seen as an "on-the-fly enhancement" of the logic with newly identified predicates with multiple occurrences.

For the definition of semantics transformers and calls to these transformers, the syntax of jMosel is enhanced by the `let`-construct

$$\texttt{let } < predicatename > \texttt{(} < argumentlist > \texttt{) = } < definition > \\ \texttt{in } < formula >$$

that allows one to formulate formulas like:

```
let pred(X,Y) =subseteq(X,Y) & shifteq(Y,X)
       in pred(A,B) <-> pred(S,T).
```

where a new predicate `pred` with arguments `X` and `Y` is defined by the formula `subseteq(X,Y) & shifteq(Y,X)` and instantiated twice in the formula `pred(A,B) <-> pred(S,T).`

Definition 3 (Semantics of the *let*-Construct)
For formulas $f_1, f_2 \in F$ and a predicate `Pred(A1,...,An)` $\in P$, the semantics of the *let*-construct is defined as follows:

$$[\![\texttt{let Pred(arg1,...,argn) = } f_1 \texttt{ in } f_2]\!] =_{df} [\![f_2[f_1/\texttt{Pred(x1,...,xn)}]]\!]$$

where $\cdot[\cdot/\cdot] : F \times ID \times ID \longrightarrow F$ denotes the usual syntactic substitution.

We use the `let`-construct to implement second-order value numbering for jMosel. There, the definitions of and calls to semantics transformers are automatically inserted into the considered formula according to the value numbers assigned to the individual computations.

In the following we first illustrate on a very simple example how a semantics transformer is identified and inserted into the formula, then we consider a more complex case study in Section 5.

4.3 Example

As a short example for the process of second-order value numbering, we consider the following jMosel formula:

```
(subseteq(A,B) & shifteq(C,D)) | (subseteq(X,Y) & shifteq(V,W))
```

This formula is similar to the one of Section 3.2, but cannot benefit from first-order value numbering, since the atomic formulas `subseteq` and `shifteq` are called with different parameters. This is a very frequent case in practice: in hardware design, for example, circuits are composed of a small number of component types, each instance of which has the same abstract function, but is connected differently. Circuits would thus not be eligible for first order value numbering, but are an excellent application for second-order value numbering.

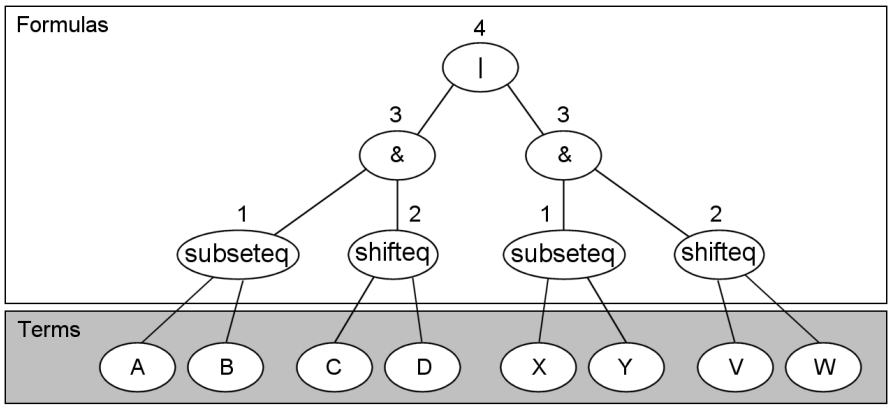


Figure 5: 2nd-order value numbering applied to a jMosel formula.

The formula's syntax tree after computation of second-order value numbers is shown in Fig. 5. The nodes of the tree have been divided into two sets, *formulas* and *terms*; nodes representing terms have not been numbered by the 2nd-order value-numbering procedure.

The two nodes labeled with "`&`" share the value number 3; this means they both perform the same set of computations and can therefore be replaced by calls to a same semantics transformer `st_3`. The nodes with the value numbers 1 and 2 are not taken into account, since they are labeled with atomic predicates.

The definition of the `st_3` transformer is isomorphic to the subtrees labeled with value number 3, but all occurring variables are replaced by fresh variables "`arg_n`". This transformer is inserted via let-construct into the formula, and the subtrees labelled with 3 are replaced by calls to `st_3` (see the corresponding syntax tree in Fig. 7), resulting in the formula:

```
let st_3(arg_1,arg_2,arg_3,arg_4) =
      subseteq(arg_1,arg_2) & shifteq(arg_3,arg_4)
  in st_3(A,B,C,D) | st_3(X,Y,V,W)
```

When compiling this formula, the conjunction of the predicates `subseteq` and `shifteq` is only computed once and stored as a semantics transformer, opposed to the original formula, where the conjunction is computed twice. The detailed course of the optimization and compilation is described in the next section.

4.4 The Optimizing Transformation

The optimization of the syntax tree for the formula

```
(subseteq(A,B) & shifteq(C,D)) | (subseteq(X,Y) & shifteq(V,W))
```

is performed in the following steps:

- Perform the numbering of the syntax tree, resulting in the labelling shown in Fig. 5.
- Identify the good targets for optimization: the nodes labelled with 3 qualify, as there exist more than one, and the corresponding subtrees contain logical operators.

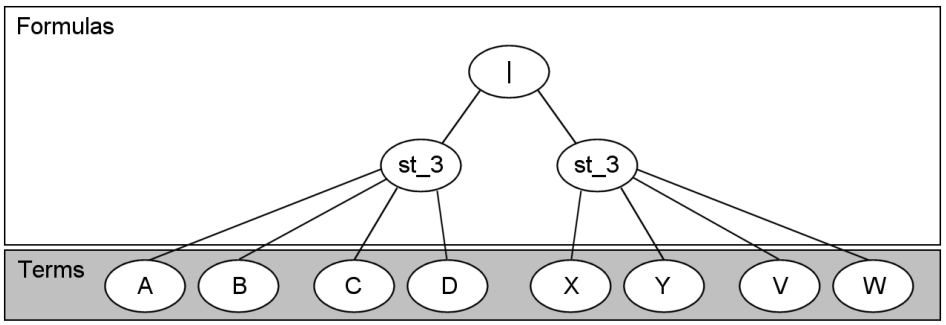


Figure 6: Syntax tree with calls to the semantics transformer.

- Create a semantics transformer `st_3` for the nodes labelled with 3 by duplicating one of the syntax trees of the corresponding subfunction.

- Replace the two occurrences of syntax nodes labelled with 3 by calls to the newly created semantics transformer `st_3` (Fig. 6).

- Add the definition of `st_3` to the top of the syntax tree (Fig. 7).

The compiler operates on the modified syntax tree as follows:

- At the "let" construct it compiles the semantics transformer's definition, identified by the subtree of the second child node of "let".

- At the nodes representing the atomic predicates `subseteq(arg_1,arg_2)` and `shifteq(arg_3,arg_4)` it constructs the corresponding basic automata $\mathbf{a}_1$ and $\mathbf{a}_2$.

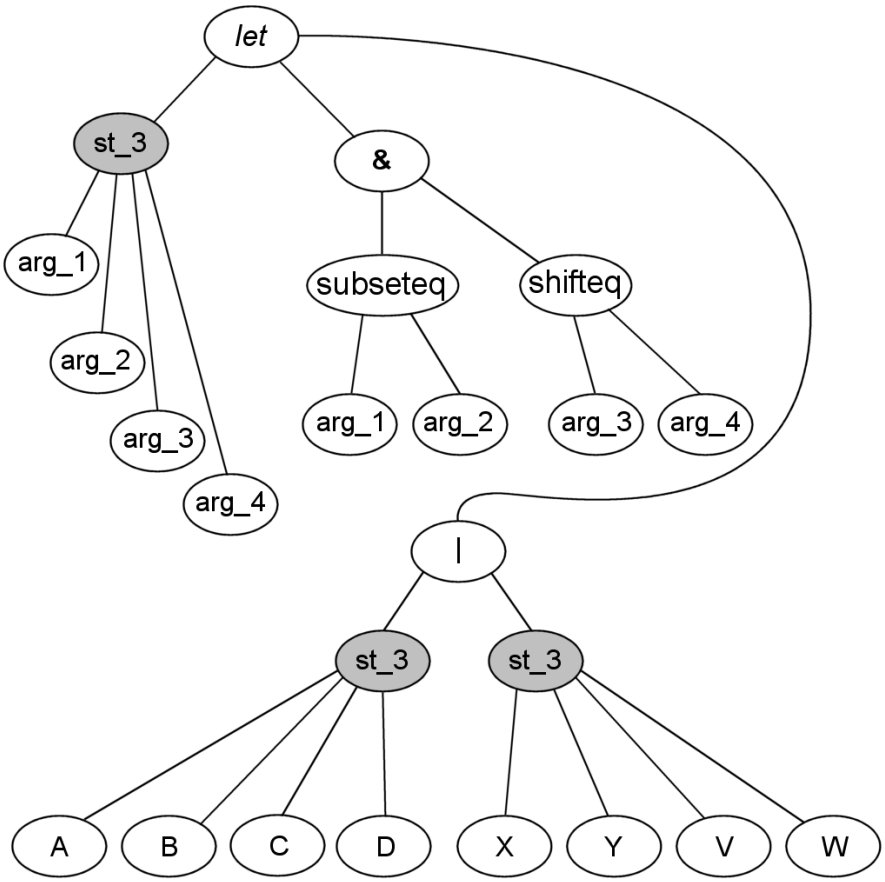
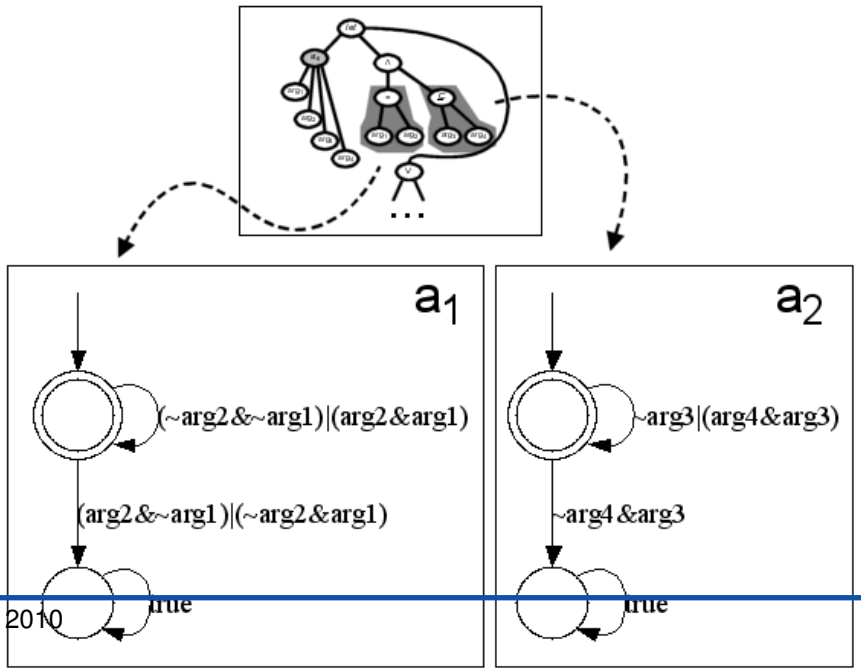


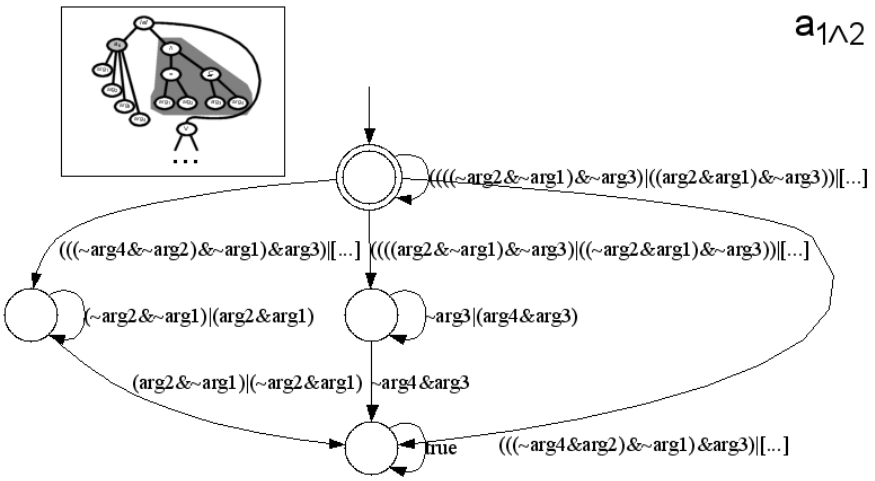
Figure 7: Syntax tree after optimization.



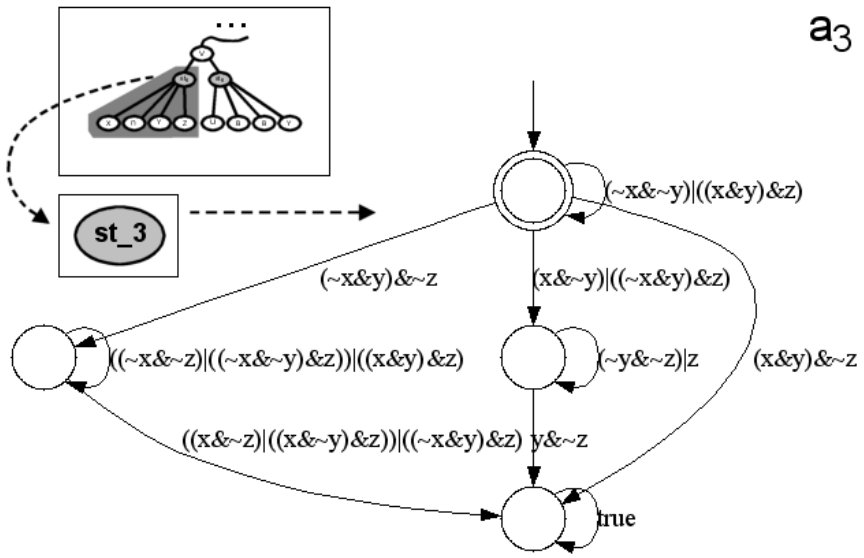
- At the node representing the formula

  ```
  subseteq(arg_1,arg_2) & shifteq(arg_3,arg_4)
  ```

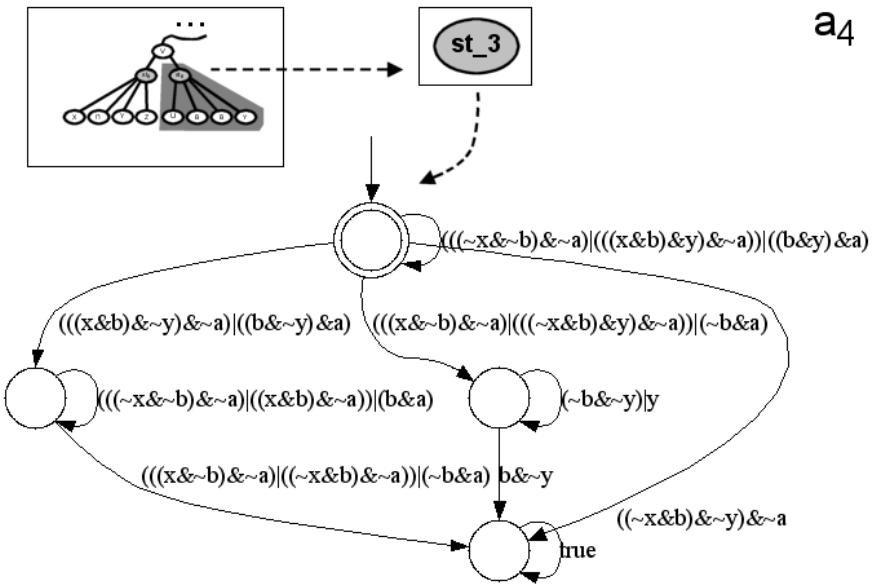
  it constructs the product automaton $\mathbf{a}_{1\wedge 2}$ representing the conjunction of $\mathbf{a}_1$ and $\mathbf{a}_2$.



- It stores the resulting automaton as a semantics transformer named `st_3` with arguments `arg_1,...,arg_4`.
- The compilation continues with the subtree of the third child node of "let".
- At the node representing the call of a semantics transformer `st_3(A,B,C,D)` the pre-computed definition of `st_3` is copied, replacing the arguments `arg_1, ...,arg_4` in the edge labels with the terms `A,B,C,D` to yield the result automaton $\mathbf{a}_3$.



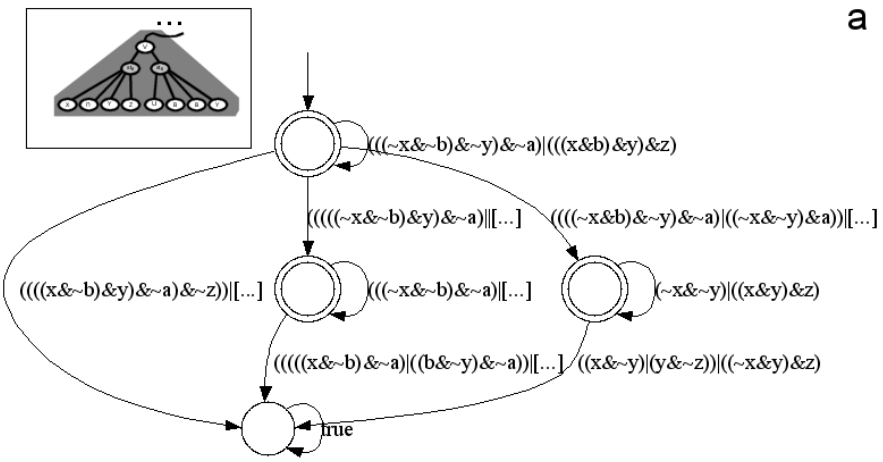
- At the node representing the call of a semantics transformer `st_3(X,Y,V,W)` the precomputed definition of `st_3` is copied again, this time the arguments `arg_1,...,arg_4` in the edge labels are replaced with the terms `X,Y,V,W` to form the result automaton $\mathbf{a}_4$.



- At the node representing the formula

```
st_3(A,B,C,D) | st_3(X,Y,V,W)
```

it constructs the product automaton $\mathbf{a}$ representing the disjunction of $\mathbf{a}_3$ and $\mathbf{a}_4$ and returns it as the compilation's result.



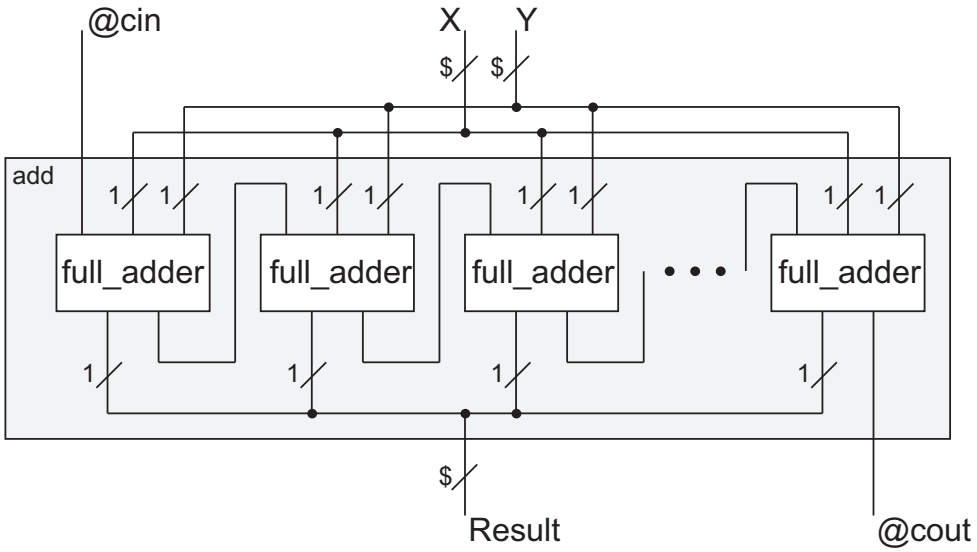


Figure 8: Structure of the parametric adder

5 Application and Performance Measuring

One of jMosel's main application areas is the specification and verification of parametric hardware systems. We tested the presented optimization with a "real-world" example, applying it to the structural description of a parametric adder that describes the family of adder circuits for bit vectors of length n.

Structural Description of a Parametric Adder

Fig. 8 shows the structure for this adder based on n interconnected full adders. The circuit adds two bit vectors X and Y and stores the result as the new vector *Result*. The Boolean variables *@cin* and *@cout* are the carry-in and carry-out bits.

The size of input formula and of the resulting automaton are too large for a detailed discussion in this paper, so we only present the results in terms of key data at this point. The compilation times have been measured on an Intel Centrino Duo System (2 x 2.16 GHz) with 1 GB of RAM:

Optimization	none	1st-ord. VN	2nd-ord. VN
Nodes in synt. tree	472	469	452
Depth of synt. tree	26	27	32
overall run time	11.50 sec	10.49 sec	3.47 sec
Sem. transformers	-	1	6

As we expected, first order value numbering does not contribute significatively to performance: the sharing is at the level of subcircuit types, not of fully instanced values.

The increased depth of the modified syntax tree is due to the fact that all definitions of semantics transformers are added to the top of the tree. By identifying 6 semantics transformers, the size of the tree could be reduced by 20 nodes. This does not seem too exciting at first sight; however, it has quite some impact: the overall run time of the decision process is accelerated by a factor of three.

The enormous speedup is quite surprising, since the adder's structural description already included user-defined predicates for frequently occurring constructs like the full adder and logical gates. This shows that even a carefully written formula and well structured circuits might still contain significant potential of redundancy, and therefore could benefit greatly from second-order value numbering.

6 Conclusion

We have presented second-order value numbering, a new optimization technique for suppressing redundancy, in a version tailored to the application for improving the decision procedure of jMosel, a verification tool set for monadic 2nd-order logic on strings. Our technique extends the well-known concept of value numbering to consider not merely values, but *semantics transformers* that can be efficiently pre-computed and help to avoid redundancy at a second-order level. We have illustrated the effect of this optimization for a parametric family of hardware circuits, where we observed a performance gain by a factor of 3. This result is surprising, as the design of these circuits exploits already structural similarity.

Currently we are working on a careful experimental analysis of the impact of our technique in practice using standard benchmarks and libraries. We conjecture that we will observe a growth of the improvement factor with the size of the system, i.e. a 'felt' superlinear speedup.

In a more general perspective, second-order value numbering can be regarded as a means for a specific semantic form of procedural abstraction [SHKN76, DWF+07] in a similar way as value numbering (or its generalization to Value Flow graphs) is a semantic support for code motion [SKR90]. Thus besides looking for further application domains for second-order value numbering, it would also be interesting to investigate how the structural generalization of value numbering presented in [SKR90] can be raised to second-order in order to achieve a truly semantic notion of procedure abstraction for imperative programs.

Bibliography

[AWZ88] B. Alpern, M. N. Wegman, F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Pp. 1–11. ACM Press, New York, NY, USA, 1988.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8):677–691, 1986.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. Intern. Congr. Math.* Pp. 23–35. Almqvist and Wiksells, 1963.

[CS70] J. Cocke, J. T. Schwartz. Programming Languages and their Compilers. Courant Institute of Mathematical Sciences, New York University, 1970.

[DWF+07] A. Dreweke, M. Wörlein, I. Fischer, D. Schell, T. Meinl, M. Philippsen. Graph-Based Procedural Abstraction. In Society (ed.), *Proc. of the 2007 CGO*. Pp. 259–270. IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[KMMG97] P. Kelb, T. Margaria, M. Mendler, C. Gsottberger. MOSEL: A Flexible Toolset for Monadic Second-Order Logic. In *Proc. TACAS'97*. Lecture Notes in Computer Science 1217, pp. 183–202. Springer Verlag, 1997.

[KMS02] N. Klarlund, A. Møller, M. Schwartzbach. MONA Implementation Secrets. *International Journal of Foundations of Computer Science*, 2002.

[Mar96] T. Margaria. Fully Automatic Verification and Error Detection for Parameterized Iterative Sequential Circuits. In *Proc. TACAS '96*. Lecture Notes in Computer Science 1055, pp. 258–277. Springer Verlag, 1996.

[SHKN76] T. Standish, D. Harriman, D. Kibler, J. Neighbors. The Irvine Program Transformation Catalogue. University of California, Irvine, 1976.

[SKR90] B. Steffen, J. Knoop, O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *European Symposium on Programming*. Pp. 389–405. 1990.

[TWMS06] C. Topnik, E. Wilhelm, T. Margaria, B. Steffen. jMosel: A Stand-Alone Tool and jABC Plugin for M2L(Str). In *Model Checking Software: 13th International SPIN Workshop, Vienna (Austria)*. LNCS 3925/2006, pp. 293–298. Springer-Verlag, 2006.