# APLCTRANS ALGORITHM FOR PLC VERIFICATION

**R. Šusta** *

* *Department of Control Engineering, CTU-FEE, Technická 2, Praque 6*
*fax : +420 224 918 646 and e-mail : susta@control.felk.cvut.cz*

Abstract: Before verifying any program we must convert its source code into some universal language acceptable by a chosen model checker. The presented APLCTRANS algorithm is based on transfer sets that allows associative composition of a subset of PLC programs to mathematical formulas. The formulas are usually accepted by many checkers in some form and they can be also used for fast parallel decomposition of a PLC program. APLCTRANS algorithm converts a program in linear time in the size of source code at the most cases, though the converted program has an exponential complexity of its execution time...

Keywords: PLC programs, verification and validation, conversion of PLC, APLCTRANS.

## 1. INTRODUCTION

Programmable logical controllers (PLCs) have proven their worth in countless industrial applications, but the safety of their software is an unknown element in general and the formal validation is needed. However, before verifying any PLC program we must convert its code into some universal language acceptable by chosen model checker. Therefore we deal with developing the conversion method that will come close as possible to an ideal situation: enter a PLC program, propositions and read results. That means to create algorithms allowing both effective conversions of PLC programs and their decomposition to simple blocks verifiable in reasonable time.

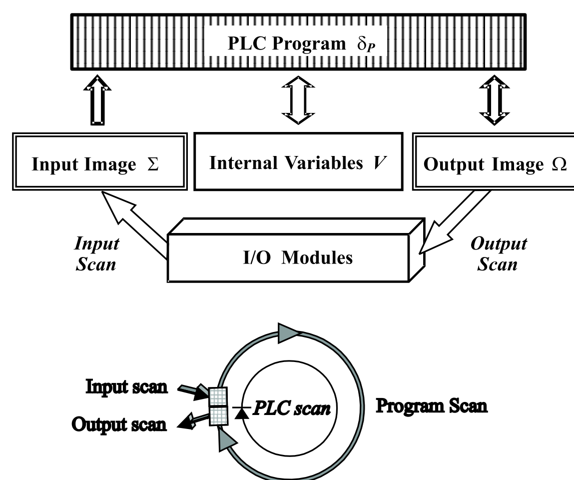



Fig. 1. PLC — Programmable Logic Controller

## 2. PROBLEM OF PLC COMPOSITION AND RELATED WORKS

In most cases, PLCs (Programmable Logical Controllers) operate in the cyclic mode depicted in Figure 1, or they contain at least one program executed cyclically (called continuous task).

PLC storage $S$ incorporates three sets of variables: $\Sigma$ — finite set of PLC inputs, $\Omega$ — finite set of PLC outputs, and $V$ — internal memory, to which we also add internal PLC registers — at least

boolean $f_{reg}$ flag register [1] and the evaluation stack $E_{stack}$ for $f_{reg}$ values, so $S = \Sigma \cup \Omega \cup V \cup \{f_{reg}\} \cup E_{stack}$ .

The cyclic behavior is suitable for verifying a PLC program as an automaton by a model checker. We only need to express the operations of PLC program scan as the mapping $\delta_P : S \rightarrow S$. Even if many publications concern PLCs and their verification, few works include some conversion methods. We have only found out algorithms based on converting single rungs of the ladder diagram.

The algorithm in (Rausch and Krogh 1998) converts each rung [2] into separated module of SMV (see (McMillan 1997)), which joins all modules into its inner model. The similar approach was found in (Rossi and Schnoebelen 2000), where it was limited to one ladders diagram and it prohibited using subroutines.

The method published in (Chambers *et al.* 2001) is based on X-machine, which also emulates each rung separately. It requires functional compositions in a form $f_5(f_4(f_3(f_2(f_1(x)))))$ and generating trees of possible executional paths, which can result in exponential complexity.

This paper presents new approach based on transfer sets with associative composition property, which allow composing PLC program from its bottom to top. Whole program is represented by mathematical formulas that may be used either for modeling the program as one automaton or for decomposing it into independent automata.

The transfer sets were described in (Šusta 2002*b*), where is also presented the decomposition of the final result with $O(|S|^3)$ complexity. This decomposition algorithm itself was also published in (Šusta 2002*a*), where is more detailed description of PLCs.

In this paper, the transfer sets are defined by another more comprehensive way.

## 3. TRANSFER SETS

In this section, we introduce the representation of PLC instructions as the *transfer sets* of *transfer functions* and we show that the set of all transfer sets with the operation of their composition is a monoid. To simplify definitions for reducing the size of this paper, we limit our considerations only to boolean operations which are the simplest case because they are consistent with ordinary boolean functions. We will briefly outline the extension of transfer sets to arithmetic operations and timers at the end of the paper.

We denote an ordered finite set of integer numbers by $I$ i.e., $I \stackrel{df}{=} \{1, 2, \ldots, |I|\}$ , and the set of all boolean variables by $\mathcal{B}$. We will suppose that a storage $S$ of PLC contains only boolean variables, i.e. $S \subset \mathcal{B}$. Boolean transfer functions have non-empty boolean expressions generated by the grammar:

$$\begin{aligned} Gbexp ::= 1 \mid 0 \mid b \mid \neg Gbexp \mid (Gbexp_1 \wedge Gbexp_2) \mid \\ (Gbexp_1 \vee Gbexp_2) \mid (Gbexp_1 \equiv Gbexp_2) \\ \mid (Gbexp_1 \not\equiv Gbexp_2) \mid (Gbexp) \qquad (1) \end{aligned}$$

where $b \in S$, $b \in \mathcal{B}$ ranges over all boolean variables, and 0 and 1 stand for logical constants. We will suppose that $0 \equiv false$ and $1 \equiv true$, which is common practice in PLC programs. The language generated by $Gbexp$ will be denoted by $Bexp^+$ and its elements by $bexp$, i.e., $bexp \in Bexp^+$. All elements can be primed or subscripted, for example, $bexp$, $bexp_1$, $bexp_2$ all stands for three (possible different) boolean expressions.

To manipulate with transfer functions without cumulating too many symbols, we use special notation for variables and their transfer functions.

*Definition 1.* Let $x \in S$ be any variable in PLC storage $S$ and $bexp \in Bexp^+$ be any expression generated by grammar $Gbexp$ then $\hat{f}(x) [\![bexp]\!]$ is called *boolean transfer function for variable $x$ on storage $S$* where $x$ value is defined by the assignment $[\![x := bexp]\!] S$, i.e., $x$ equals to a value of expression $bexp$ evaluated with respect of to momentary state of storage $S$.

Transfer functions of the type $\hat{f}(x) [\![x]\!]$, where $x \in S$ is any variable, are called *canonical transfer functions*. The set of all boolean transfer functions for variables in $S$ is denoted by $\widehat{\mathcal{B}}(S)$.

To simplify orientation in the following paragraphs, we will also hat-accent all further objects related to transfer functions or transfer sets.

To distinguish among several transfer functions for one identical variable, we mark such functions by subscripts. Symbols $\widehat{f_1}(x)$ and $\widehat{f_2}(x)$ stand for two (possibly different) transfer functions for $x$, and $\hat{f}(x_1)$ and $\hat{f}(x_2)$ stand for transfer functions for two (possibly different) variables $x_1$ and $x_2$.

Any $\hat{f}(x) [\![bexp]\!] \in \widehat{\mathcal{B}}(S)$ is fully specified by the variable, which it belongs to, and by non-empty

[1] $f_{reg}$ is utilized by PLC for evaluating logical conditions that affect the execution of instructions in ladder or function block diagrams.

[2] In PLC terminology, a rung means a continuous part of PLC instructions that correspond to assignment operations, which depend on same input conditions. Each rung is depicted by one continuous graph, called rung, in a ladder diagram diagram or it is described by one network block in a statement list.

expression *bexp* that represents its value, but the function can be abbreviated according to our momentary assumption about its structure. $\hat{f}(x)$ stands for any transfer function for $x$ variable with any expression. On the contrary, $\hat{f}(x)\,[\![x \vee y]\!]$ specifies the transfer function corresponding to the assignment $[\![x := x \vee y]\!]\, S$.

*Definition 2.* Let $\hat{f}(x)\,[\![bexp]\!] \in \widehat{\mathcal{B}}(S)$ be a transfer function then its domain is defined as the following:

$$\mathrm{dom}(\hat{f}(x)\,[\![bexp]\!]) \stackrel{df}{=} \{b_i \in \mathcal{B} \mid b_i \text{ is used in } bexp\} \quad (2)$$

The equality for boolean transfer function is determined by belonging to the same variable and their equivalent values in the meaning of boolean equivalence.

*Definition 3.* Let $\hat{f}(x)\,[\![bexp_x]\!] \in \widehat{\mathcal{B}}(S)$ and also $\hat{f}(y)\,[\![bexp_y]\!] \in \widehat{\mathcal{B}}(S)$ be two boolean transfer functions. We define the relation $\hat{f}(x) \mathrel{\widehat{=}} \hat{f}(y)$ as the concurrent satisfaction of two conditions $x = y$ and $bexp_x \equiv bexp_y$, otherwise $\hat{f}(x) \mathrel{\widehat{\neq}} \hat{f}(y)$.

*Lemma 3.1.* Binary relation $\widehat{=}$ on set $\widehat{\mathcal{B}}(S)$ is the equivalence.

*Proof:* The relation is certainly reflexive and symmetric due to comparing by $=$. Transitivity can be proved by direct applying Definition 3. □

The equivalence allows the decomposition defined as the classes of equivalence on $\widehat{\mathcal{B}}(S)$, which split $\widehat{\mathcal{B}}(S)$ into disjunctive subsets (the property of the classes of equivalence). Each class of equivalence

$$\widetilde{R}(\hat{f}(x)) \stackrel{df}{=} \left\{ \widehat{f}_i(x) \in \widehat{\mathcal{B}}(S) \mid \hat{f}(x) \mathrel{\widehat{=}} \widehat{f}_i(x) \right\} \quad (3)$$

can be considered as one element written in several different forms, but always with equal value.

The composition of more transfer functions at once requires the definition of the transfer sets to specify required replacements.

*Definition 4.* A subset $\widehat{X} \subset \widehat{\mathcal{B}}(S)$ is called a *transfer set on* $S$ if $\widehat{X}$ satisfies for all $\widehat{f}_i(x_i), \widehat{f}_j(x_j) \in \widehat{X}$ that $x_i = x_j$ implies $i = j$. The set of all transfer sets for $S$ variables is denoted by $\widehat{\mathcal{S}}(S)$ i.e., $\widehat{X} \in \widehat{\mathcal{S}}(S)$.

In other words, any transfer set contains at most one transfer function for each variable in $S$.

*Example* 3.1 Let $S = \{x, y, z\}$ be a PLC storage then $\widehat{X} = \{\hat{f}(x), \hat{f}(y)\}$ is a transfer set on $S$, i.e., $\widehat{X} \in \widehat{\mathcal{S}}(S)$, but $Y = \{\widehat{f}_1(x), \widehat{f}_2(x)\}$ is never a transfer set, because $Y$ contains two transfer functions for $x$, and $\widehat{Z} = \{\hat{f}(x), \hat{f}(a)\}$ is not transfer sets on $S$, $\widehat{Z} \notin \widehat{\mathcal{S}}(S)$, because $a \notin S$.

Manipulation with transfer set requires testing the presence of a transfer function for given variable $x \in S$.

*Definition 5.* Binary relation $\widehat{\in}$ on sets $S$ and $\widehat{\mathcal{B}}(S)$ is defined for all $\widehat{X} \in \widehat{\mathcal{S}}(S)$ and $x \in S$ as $x \mathrel{\widehat{\in}} \widehat{X}$ if $\exists\, \hat{f}(x) \in \widehat{X}$, otherwise $x \mathrel{\widehat{\notin}} \widehat{X}$.

The composition of transfer sets is based on the *concurrent substitution* defined as mapping from variables in $S$ to terms of *Gbexp* grammar. The substitution replaces all occurrences of variables, which appear in both the term and the domain of the substitution.

*Definition 6.* Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set and $bexp_0 \in Bexp^+$ be an expression. *Concurrent substitution* $\widehat{X} \rightsquigarrow bexp_0$ is defined as such operation whose result is logically equivalent to the expression obtained by these two consecutive steps:

1. For all $\hat{f}(x_i)\,[\![bexp_i]\!] \in \widehat{X}$, if $x_i \in \mathrm{dom}(bexp_0)$ then all such occurrences of $x_i$ are replaced by unique references to $\hat{f}(x_i)$.
2. For all $\hat{f}(x_i)\,[\![bexp_i]\!] \in \widehat{X}$, if $bexp_0$ contains a reference to $\hat{f}(x_i)$ then all such references are replaced by "$(bexp_i)$" i.e., the value of $\hat{f}(x_i)$ enclosed inside parentheses.

The main purpose of the definition above is to exclude cyclic substitutions without detailed reasoning about an algorithm for this operation.

*Example* 3.2 If a concurrent substitution is given $\{\widehat{f}_1(x)\,[\![x \wedge y]\!], \widehat{f}_2(y)\,[\![\neg x \wedge \neg y]\!]\} \rightsquigarrow (x \vee y)$ then direct application of the first step described in the definition above yields $(\underline{f}_1 \vee \underline{f}_2)$ where underlinings emphasize the fact that we have replaced the variables $x$ and $y$ in $(x \vee y)$ by some unique references to the transfer functions and these references need not be their identifiers necessarily.

The second step yields $((x \wedge y) \vee (\neg x \wedge \neg y))$ .

*Definition 7.* (Weak composition). Weak composition $\widehat{Z} = \widehat{X} \circ \widehat{Y}$ of two given transfer sets $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ is the transfer set $\widehat{Z} \in \widehat{\mathcal{S}}(S)$ containing transfer functions $\widehat{h}(x_i)\,[\![bexp_i]\!] \in \widehat{Z}$ defined by:

$$\widehat{h}(x_i)\,[\![bexp_i]\!] \stackrel{df}{=} \widehat{f}(x_i)\,\left[\!\!\left[\widehat{Y} \rightsquigarrow bexp_{x,i}\right]\!\!\right] \text{ for all } x_i \,\widehat{\in}\, \widehat{X}, i \in I, |I| = |\widehat{Z}| = |\widehat{X}|$$

where $bexp_{x,i}$ belongs to the transfer function $\widehat{f}(x_i)\,[\![bexp_{x,i}]\!] \in \widehat{X}$ for $x_i$ variable.

*Lemma 3.2.* The weak composition is not associative on $\widehat{\mathcal{B}}(S)$.

*Proof:* We prove the lemma by the example. Let $S = \{x, y\}$ be storage and $\widehat{X} = \{\hat{f}(x)\,[\![x \wedge y]\!]\}$, $\widehat{Y} = \{\hat{f}(y)\,[\![x]\!]\}$, and $\widehat{Z} = \{\hat{f}(x)\,[\![\neg x]\!]\}$ three transfer sets on $S$ then $((\widehat{X} \circ \widehat{Y}) \circ \widehat{Z}) = \{\widehat{f}(x)\,[\![\neg x]\!]\}$. On the contrary, $(\widehat{X} \circ (\widehat{Y} \circ \widehat{Z})) = \{\widehat{f}(x)\,[\![0]\!]\}$. □

The non-associative behavior has appeared in the example due to different variables affected by transfer sets: $y \,\widehat{\in}\, \widehat{Y}$ and $x \,\widehat{\in}\, \widehat{X}$, $x \,\widehat{\in}\, \widehat{Z}$. The restriction of $\widehat{\mathcal{B}}(S)$ to the subset of transfer function for a given variable $x \in S$ offers the simplest solution.

This subset, denoted by $\widehat{\mathcal{B}}(S/x)$, is defined by

$$\widehat{\mathcal{B}}(S/x) \stackrel{df}{=} \left\{ \widehat{f}_i(x_i) \in \widehat{\mathcal{B}} \mid x_i = x \right\} \tag{4}$$

and it is possible to prove that the weak composition is associative on $\widehat{\mathcal{B}}(S/x)$ and $\widehat{\mathbb{G}}(S/x) = (\widehat{\mathcal{B}}(S/x)/\widehat{=}, \circ)$ is semigroup but this theoretical result does not allow the construction of an effective algorithm.

To create an associative composition, we define the operator that appends canonical transfer functions (see page 2) for $S$ variables, which transfer functions are missing in a transfer set.

*Definition 8.* Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set on storage $S$. The extension of $\widehat{X}$, denoted by $\widehat{X} \uparrow S$, is the set with cardinality $|\widehat{X} \uparrow S| = |S|$ whose members are $\hat{f}(x_i)$ transfer functions defined for all $x_i \in S$ by cases:

$$\hat{f}(x_i) \stackrel{df}{=} \begin{cases} \hat{f}(x_i) & \text{if } x_i \,\widehat{\in}\, \widehat{X} \\ & \quad \text{and thus } \exists \hat{f}(x_i) \in \widehat{X} \\ \hat{f}(x_i)\,[\![x_i]\!] & \text{if } x_i \,\widehat{\notin}\, \widehat{X} \end{cases} \tag{5}$$

The set $\emptyset \uparrow S$, called *canonical transfer set* on $S$, contains canonical transfer functions for all $S$ variables. To emphasize its fundamental importance, let us denoted this set by $\widehat{\mathcal{E}}^S$.

*Definition 9.* Let $\widehat{X} \in \widehat{\mathcal{S}}(S)$ be a transfer set on storage $S$. The compression of $\widehat{X}$, denoted by $\widehat{X}\downarrow$, is defined by

$$\widehat{X}\downarrow \stackrel{df}{=} \left\{ \hat{f}(x_i) \in \widehat{X} \mid \hat{f}(x_i) \notin \widehat{\mathcal{E}}^S \right\} \tag{6}$$

The compression expresses the meaning of $\downarrow$ operator that serves for packing transfer sets to reduce allocated memory, but primary purpose of $\downarrow$ consists in portability among different $S$. We define a transfer set on a subset $S_0 \subseteq S$, which includes operands of a converted instruction, then it is possible to extend this transfer set by $\uparrow$ operator to any superset of $S_0$.

*Example* 3.3 Let storage be $S = \{x, y, z\}$. If $\widehat{T} \in \widehat{\mathcal{S}}(S)$ is given as

$$\widehat{T} = \left\{ \begin{array}{l} \hat{f}(x)\,[\![x]\!]\,, \\ \hat{f}(y)\,[\![x \vee y]\!] \end{array} \right\} \text{ then}$$

$$\widehat{T}\uparrow S = \left\{ \begin{array}{l} \hat{f}(x)\,[\![x]\!]\,, \\ \hat{f}(y)\,[\![x \vee y]\!]\,, \\ \hat{f}(z)\,[\![z]\!] \end{array} \right\} \text{ and}$$

$$\widehat{T}\downarrow = \left\{ \hat{f}(y)\,[\![x \vee y]\!] \right\}$$

The expansion and compression are generally not inverse operations but they will be if we narrow $\widehat{\mathcal{S}}(S)$ to subsets invariable with respect to these operations.

$$\widehat{\mathcal{S}}(S)\downarrow \stackrel{df}{=} \left\{ \widehat{X} \in \widehat{\mathcal{S}}(S) \mid \widehat{X} = \widehat{X}\downarrow \right\} \tag{7}$$

$$\widehat{\mathcal{S}}(S)\uparrow S \stackrel{df}{=} \left\{ \widehat{X} \in \widehat{\mathcal{S}}(S) \mid \widehat{X} = \widehat{X}\uparrow S \right\} \tag{8}$$

*Lemma 3.3.* $\uparrow S$ and $\downarrow$ define bijective mapping between $\widehat{\mathcal{S}}(S)\downarrow$ and $\widehat{\mathcal{S}}(S)\uparrow S$.

The proof follows directly from definition. The lemma allows extending the equivalence of transfer function introduced in Definition 3 to transfer sets.

*Definition 10.* Let $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ be two transfer sets on $S$ then $\widehat{X} \,\widehat{=}\, \widehat{Y}$ if, for all $\hat{f}(x_i) \in \widehat{X} \uparrow S$, there exists $\hat{f}(x_i) \in \widehat{Y} \uparrow S$ such that $\hat{f}(x_i) \,\widehat{=}\, \hat{f}(x_i)$

Notice that exactly one transfer function always exists for any $x_i \in S$ in every transfer set extended by "$\uparrow S$" operation. This allows to define the main operation with transfer sets.

*Definition 11.* (Composition). Let $\widehat{X}, \widehat{Y} \in \widehat{\mathcal{S}}(S)$ be two transfer sets on $S$ with extensions $\widehat{X}' = (\widehat{X} \uparrow S)$ and $\widehat{Y}' = (\widehat{Y} \uparrow S)$. Using this notation and Definition 7 of the weak composition we define $\circledcirc$ composition by:

$$\widehat{X} \circledcirc \widehat{Y} \stackrel{df}{=} \{ \hat{f}(x_i) \circ \widehat{Y}' \mid \hat{f}(x_i) \in \widehat{X}' \text{ where } x_i \in S \}\downarrow \tag{9}$$

Now we present the main theorem of this part for which validity we have created transfer sets.

*Proposition 3.1.* The composition $\circledcirc$ is the associative operation on $\widehat{\mathcal{S}}(S)$.

*Proof:* The proof is not difficult, but long, so we only outline it. We must prove that the equation $\widehat{X} \circledcirc (\widehat{Y} \circledcirc \widehat{Z}) = (\widehat{X} \circledcirc \widehat{Y}) \circledcirc \widehat{Z}$ holds for any $\widehat{X}, \widehat{Y}, \widehat{Z} \in \widehat{\mathcal{S}}(S)$. First, the flow of symbols in terms of languages is considered to prove that the expressions of the result contains only variable terms imported from a rightmost transfer set $\widehat{Z}$. Then, it is shown that the associativity holds if no minimization of boolean expressions are provided, and finally, the influence of minimization is considered. □

*Example* 3.4 We will return to the example presented in Lemma 3.2 (see page 4) for proving that weak composition is not associative. We evaluate the same transfer sets, but we will compose them by $\circledcirc$ defined on storage $S = \{x, y\}$.

$$\left(\widehat{X} = \{\hat{f}(x)\, [\![x \wedge y]\!]\}\right) \circledcirc \left(\widehat{Y} = \{\hat{f}(y)\, [\![x]\!]\ \}\right) \circledcirc \left(\widehat{Z} = \{\hat{f}(x)\, [\![\neg x]\!]\}\right)$$

Using bijective property of $\uparrow$ and $\downarrow$ operators we expand Equation 10 applying $\uparrow$ operator to all members and $\downarrow$ to the result, which yields

$$\left(\left\{\begin{array}{c}\hat{f}(x)\, [\![x \wedge y]\!]\,,\\ \hat{f}(y)\, [\![y]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{f}(x)\, [\![x]\!]\,,\\ \hat{f}(y)\, [\![x]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{f}(x)\, [\![\neg x]\!]\,,\\ \hat{f}(y)\, [\![y]\!]\end{array}\right\}\right)\downarrow \quad (10)$$

Composing first two leftmost terms in Equation 10 yields

$$\left(\left\{\begin{array}{c}\hat{f}(x)\, [\![x \wedge x]\!]\,,\\ \hat{f}(y)\, [\![x]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{f}(x)\, [\![\neg x]\!]\,,\\ \hat{f}(y)\, [\![y]\!]\end{array}\right\}\right)\downarrow = \left\{\begin{array}{c}\hat{f}(x)\, [\![\neg x]\!]\,,\\ \hat{f}(y)\, [\![\neg x]\!]\end{array}\right\}\downarrow = \widehat{\Theta}(\neg x)^S$$

and the same result is obtained if we begin with two rightmost terms

$$\left(\left\{\begin{array}{c}\hat{f}(x)\, [\![x \wedge y]\!]\,,\\ \hat{f}(y)\, [\![y]\!]\end{array}\right\} \circledcirc \left\{\begin{array}{c}\hat{f}(x)\, [\![\neg x]\!]\,,\\ \hat{f}(y)\, [\![\neg x]\!]\end{array}\right\}\right)\downarrow = \left\{\begin{array}{c}\hat{f}(x)\, [\![\neg x \wedge \neg x]\!]\,,\\ \hat{f}(y)\, [\![\neg x]\!]\end{array}\right\}\downarrow \ = \widehat{\Theta}(\neg x)^S$$

where $\widehat{\Theta}(\neg x)^S$ denotes the transfer set defined for any $bexp \in Bexp^+$ by the following equation $\widehat{\Theta}(bexp)^S \stackrel{df}{=} \left\{\hat{f}(x_i)\, [\![bexp]\!] \;\middle|\; \text{for all } x_i \in S\right\}$.

The composition is associative since all variables propagate to the final result as shown in Figure 3

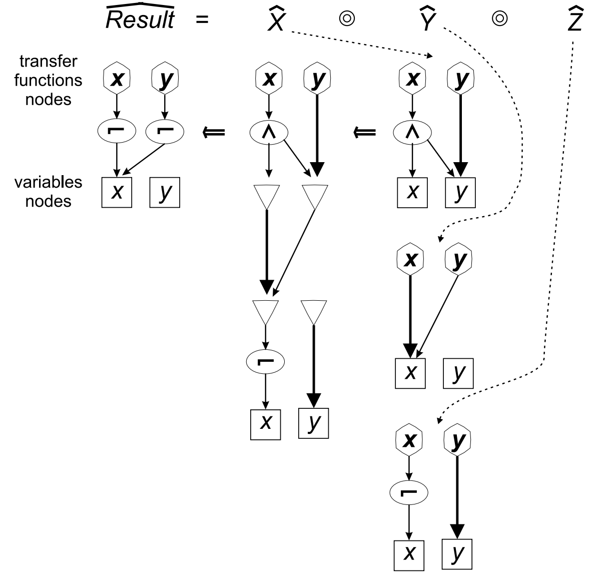



Fig. 2. Diagrams of Composition of Transfer Set in Example 3.4

that depicts the structure of the composition provided in the example above. The transfer set graphs are based on BEDs (Williams 2000).

The bottom nodes represent variables in $S$ and the upper nodes (the roots) stand for their transfer functions. Unlike tree graphs of BEDs, transfer sets are represented by forest graphs and the operand nodes are shared among all transfer functions in one transfer set. The thicker arrows emphasize the canonical transfer functions added by $\uparrow$ operator and their presence explain why $\circledcirc$ composition is associative.

*Remark 3.1.* From the amount of allocated memory, BED-like structures are also the optimal representation of transfer sets because the terms in expressions are not duplicated by $\circledcirc$ composition.

To reduce the sizes of the structures, the canonical transfer functions need not be stored in memory. We can represent only transfer sets compressed by $\downarrow$ operator. In this case, the size of final transfer set, which expresses the operations of a PLC program, will be linear in the size of PLC source code.

Canonical transfer set $\widehat{\mathcal{E}}^S$ (see Definition 4) represents the identical element of $\circledcirc$ operation so we may close this part by the one definition and two propositions.

*Proposition 3.2.* (Monoid).
$\widehat{\mathbb{M}}(S) = \left(\widehat{\mathcal{S}}(S)/\widehat{=}\,, \circledcirc\,, \widehat{\mathcal{E}}^S\right)$ is monoid.

*Definition 12.* Let $\widehat{X_1}, \widehat{X_2} \in \widehat{\mathcal{S}}(S)$ be two transfer sets on $S$ with extensions $\widehat{X_1}' = \widehat{X_1} \uparrow S$, $\widehat{X_2}' = \widehat{X_2} \uparrow S$. For any $x_i \in S$, one trans-

fer function always exists in every extended set: $\widehat{f}_1(x_i)\ [\![bexp_{1,i}]\!] \in \widehat{X_1}'$ and $\widehat{f}_2(x_i)\ [\![bexp_{2,i}]\!] \in \widehat{X_2}'$. Using this notation we define the following operations:

$$\widehat{X_1} \odot \widehat{X_2} \stackrel{df}{=} \{\hat{f}(x_i)\ [\![bexp_{1,i} \odot bexp_{2,i}]\!] \mid x_i \in S, i \in I, |I| = |S|\ \}\downarrow$$

$$\neg\widehat{X_1} \stackrel{df}{=} \{\hat{f}(x_i)\ [\![\neg bexp_{1,i}]\!] \mid x_i \in S, i \in I, |I| = |S|\ \}\downarrow$$

where $\odot$ stands for any boolean binary operation of *Gbexp* (see page 2).

*Proposition 3.3.* Lattice
$\widehat{\mathbb{A}}(S) \stackrel{df}{=} (\widehat{\mathcal{S}}(S)/\widehat{=}, \wedge, \vee, \widehat{\Theta}(1)^S, \widehat{\Theta}(0)^S)$
is Boolean algebra of transfer sets.

# 4. APLCTRAN: ABSTRACT PLC CONVERSION TO TRANSFER SETS

Here we outline APLCTRANS, the effective and fast algorithm for representing PLC program operations as one transfer set. A converted binary PLC program, or its part, must satisfy the following conditions:

(1) All operations are expressible by transfer sets, i.e., the program is rewritable into the code of abstract transfer set PLC.
*Most PLC instructions have transfer set analogies. The conversion tables were presented in (Šusta 2002*b*).*
(2) The program is reorganized by such way that code contains only jumps and calls, which point to higher addresses, as depicted in Figure 4.
*This prohibits loops and recursive subroutines. Fortunately, PLC programs used them rarely.*
(3) The instructions do not use indirect addresses in any form, i.e. their operands are either direct addresses of variables or constants.
*Unfortunately, this restriction excludes significant part of programs. Some indexes can be replaced by duplicating corresponding code, for example if indexes are used for transferring data into subroutines, but this method is not applicable to all index operations in PLCs. Weakening this restriction is the matter of further research.*

After converting operations of a PLC program into abstract code of transfer sets we may compose all transfer sets into single transfer set that will represent operation provided in one PLC scan. Because $\circledcirc$ composition of transfer sets is associative we start from the bottom to the top.

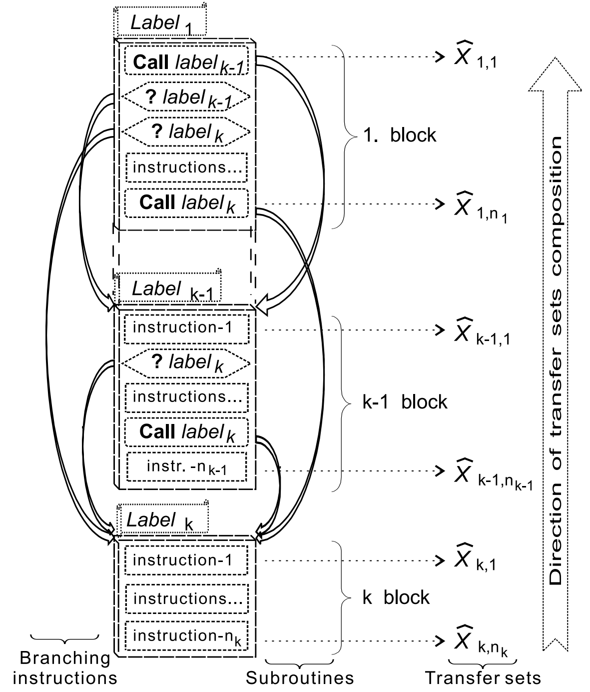



Fig. 3. Outline of APLCTRANS Algorithm

We divide the program into $k$ blocks according to used labels (see Figure 4) and begin by composing $k$ block as: $\widehat{X}_k = \widehat{X}_{k,n_k} \circledcirc \widehat{X}_{k,n_k-1} \circledcirc \ldots \widehat{X}_{k,2} \circledcirc \widehat{X}_{k,1}$ . Notice that $\circledcirc$ composition substitutes *from right to left* transfer set, therefore $\widehat{X}_{k,1}$ transfer set corresponding to the first instruction of $k$ block is on the right side of $\widehat{X}_k$ equation.

We store transfer set $\widehat{X}_k$ into memory and proceed to previous block. Unlike the last $k$ block, $k-1$ block may contain calls and jumps, but they will all point to already evaluated block $k$ due to the restriction above. We compose $k-1$ block as $\widehat{X}_{k-1} = \widehat{X}_{k-1,n_k} \circledcirc \widehat{X}_{k-1,n_k-1} \circledcirc \ldots \circledcirc \widehat{X}_{k-1,2} \circledcirc \widehat{X}_{k-1,1}$ or if it continues to the last block we insert $\widehat{X}_k$ in the front of the equation above, i.e., $\widehat{X}_{k-1} = \widehat{X}_k \circledcirc \widehat{X}_{k-1,n_k} \ldots$

$\widehat{X}_{k-1,n_k-1}$ transfer set corresponds to calling $k$ block as the subroutine. If the call in unconditional then $\widehat{X}_{k-1,n_k-1} = \widehat{X}_k$, otherwise we used modified $\widehat{X}_k$. If the condition is given by $f_{reg}$ (usual case) then $\widehat{X}_{k-1,n_k-1} = \widehat{\Theta}(f_{reg})^S \wedge \widehat{X}_k$.

We apply the similar procedure to $\widehat{X}_{k-1,2}$ (jump). If the jump is unconditional then $\widehat{X}_{k-1,2} = \widehat{X}_k$, otherwise

$$\widehat{X}_{k-1,2} = (\widehat{X}_k \wedge \widehat{\Theta}(f_{reg})^S) \vee (\widehat{X}_a \wedge \widehat{\Theta}(\neg f_{reg})^S) \qquad (11)$$

where $\widehat{X}_a$ corresponds to transfer sets after jump, composed from $\widehat{X}_{k-1,3}$ to the end of the program. Because we proceed from bottom to top, $\widehat{X}_a$ is already evaluated.

After finishing $k-1$ block we move to the previous block until we compose the first block, which represents the transfer set of whole PLC program.

### 4.1 *Example of Conversion*

Figure 4 displays the program written under developing environment RSLogix 5 for PLC-5 family of PLCs produced by Rockwell Automation.

The middle part of the figure gives the listing of PLC processor codes. RSLogix 5 allows editing these instructions in the graphical form of the ladder diagram depicted at the top of Figure 4. The corresponding transfer sets and APLC instructions are listed at the bottom. The example is analogous to Pascal program: [3]

```
var x, y, z, m: boolean;
begin   if not x then m:=y;
        y:=z;
end;
```

The simple program does not utilize evaluation stack. We will suppose that $\Sigma = \{x, y\}$, $\Omega = \{z\}$, and $V = \{m\}$, thus PLC storage will be $S = \Sigma \cup V \cup \Omega = \{x, y, z, m, f_{reg}\}$. The composition of the last block as the following:

$$\begin{aligned}\widehat{X}_2 &= \widehat{X}_{2,3} \circledcirc \widehat{X}_{2,2} \circledcirc \widehat{X}_{2,1} \\ &= \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\} \circledcirc \{\, \hat{f}(z) \llbracket f_{reg} \rrbracket \,\} \circledcirc \\ &\quad \{\, \hat{f}(f_{reg}) \llbracket f_{reg} \wedge y \rrbracket \,\} \\ &= \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket, \hat{f}(z) \llbracket f_{reg} \wedge y \rrbracket \,\}\end{aligned}$$

After the last block, the previous one is computed from the end to JMP instruction

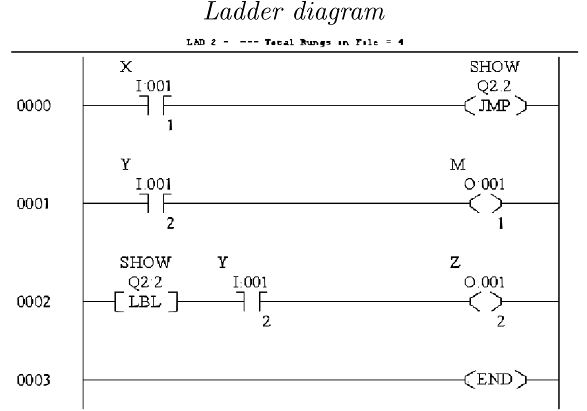
$$\begin{aligned}\widehat{X}_{1b} &= \widehat{X}_{1,9} \circledcirc \widehat{X}_{1,8} \circledcirc \widehat{X}_{1,7} \circledcirc \widehat{X}_{1,6} \circledcirc \widehat{X}_{1,5} \circledcirc \widehat{X}_{1,4} \\ &= \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\} \circledcirc \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\} \circledcirc \\ &\quad \{\, \hat{f}(m) \llbracket f_{reg} \rrbracket \,\} \circledcirc \{\, \hat{f}(f_{reg}) \llbracket f_{reg} \wedge y \rrbracket \,\} \circledcirc \\ &\quad \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\} \circledcirc \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\} \\ &= \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket, \hat{f}(m) \llbracket y \rrbracket \,\}\end{aligned}$$

Block 1 continues to block 2 and we must compose them together:

$$\begin{aligned}\widehat{X}_{1B} &= \widehat{X}_2 \circledcirc \widehat{X}_{1b} \\ &= \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket, \hat{f}(z) \llbracket f_{reg} \wedge y \rrbracket \,\} \circledcirc \\ &\quad \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket, \hat{f}(m) \llbracket f_{reg} \wedge y \rrbracket \,\} \\ &= \{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket, \hat{f}(z) \llbracket y \rrbracket, \hat{f}(m) \llbracket y \rrbracket \,\}\end{aligned}$$

[3] The presented example is very bad PLC program that was created intentionally only for demonstrating conditional JMP composition — the most complex case. The example could be programmed by much better ways, but such programs have appeared too unreadable for any manual composition.

*Ladder diagram*




*Listing created by PLC 5 development environment RSLogix 5*

```
PROJECT "EXAMPLE"
LADDER 2
% Rung: 0 %
SOR XIC I:001/1 JMP 2 EOR
% Rung: 1 %
SOR XIC I:001/2 OTE O:001/1 EOR
% Rung: 2 %
SOR LBL 2 XIC I:001/2 OTE O:001/2 EOR
    ADDRESS..SYMBOL
    O:001/2    Z
    O:001/1    M
    I:001/2    Y
    I:001/1    X
```

| PLC | Transfer set | | |
|---|---|---|---|
| SOR | $\widehat{X}_{1,1}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\}$ |
| XIC $X$ | $\widehat{X}_{1,2}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket f_{reg} \wedge x \rrbracket \,\}$ |
| JMP 2 | $\widehat{X}_{1,3}$ | = | Equation 11 |
| EOR | $\widehat{X}_{1,4}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\}$ |
| SOR | $\widehat{X}_{1,5}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\}$ |
| XIC $Y$ | $\widehat{X}_{1,6}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket f_{reg} \wedge y \rrbracket \,\}$ |
| OTE $M$ | $\widehat{X}_{1,7}$ | = | $\{\, \hat{f}(m) \llbracket f_{reg} \rrbracket \,\}$ |
| EOR | $\widehat{X}_{1,8}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\}$ |
| SOR | $\widehat{X}_{1,9}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\}$ |
| LBL 2 | | | |
| XIC $Y$ | $\widehat{X}_{2,1}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket f_{reg} \wedge y \rrbracket \,\}$ |
| OTE $Z$ | $\widehat{X}_{2,2}$ | = | $\{\, \hat{z} \llbracket f_{reg} \rrbracket \,\}$ |
| EOR | $\widehat{X}_{2,3}$ | = | $\{\, \hat{f}(f_{reg}) \llbracket 1 \rrbracket \,\}$ |

Fig. 4. Example of Simple PLC Program

After obtaining this result, APLCTRANS evaluates JMP according to Equation 11 as:

$$\begin{aligned}\widehat{X}_{1,3} &= \left( \left( \widehat{X}_2 \wedge \widehat{\Theta}(f_{reg})^S \right) \vee \left( \widehat{X}_{1B} \wedge \widehat{\Theta}(\neg f_{reg})^S \right) \right) \\ &= \left( \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket 1 \rrbracket, \\ \hat{f}(z) \llbracket f_{reg} \wedge y \rrbracket \end{array} \right\} \wedge \widehat{\Theta}(f_{reg})^S \right) \vee \\ &\quad \left( \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket 1 \rrbracket, \\ \hat{f}(z) \llbracket y \rrbracket, \\ \hat{f}(m) \llbracket y \rrbracket \end{array} \right\} \wedge \widehat{\Theta}(\neg f_{reg})^S \right)\end{aligned}$$

$$
= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket f_{reg} \rrbracket , \\ \hat{f}(x) \llbracket x \wedge f_{reg} \rrbracket , \\ \hat{f}(y) \llbracket y \wedge f_{reg} \rrbracket , \\ \hat{f}(z) \llbracket y \wedge f_{reg} \rrbracket , \\ \hat{f}(m) \llbracket m \wedge x \wedge f_{reg} \rrbracket \end{array} \right\} \vee
\left\{ \begin{array}{c} \hat{f}_{reg} \llbracket \neg f_{reg} \rrbracket , \\ \hat{f}(x) \llbracket 0 \rrbracket , \\ \hat{f}(y) \llbracket y \wedge \neg f_{reg} \rrbracket , \\ \hat{f}(z) \llbracket y \wedge \neg f_{reg} \rrbracket , \\ \hat{f}(m) \llbracket y \wedge \neg f_{reg} \rrbracket \end{array} \right\}
$$

$$
= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket f_{reg} \vee \neg f_{reg} \rrbracket , \\ \hat{f}(x) \llbracket (x \wedge f_{reg}) \vee 0 \rrbracket , \\ \hat{f}(y) \llbracket (y \wedge f_{reg}) \vee (y \wedge \neg f_{reg}) \rrbracket , \\ \hat{f}(z) \llbracket (y \wedge f_{reg}) \vee (y \wedge \neg f_{reg}) \rrbracket , \\ \hat{f}(m) \llbracket (m \wedge x \wedge f_{reg}) \vee (y \wedge \neg f_{reg}) \rrbracket \end{array} \right\}
$$

$$
= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket 1 \rrbracket , \\ \hat{f}(x) \llbracket x \wedge f_{reg} \rrbracket , \\ \hat{f}(z) \llbracket y \rrbracket , \\ \hat{f}(m) \llbracket (m \wedge x \wedge f_{reg}) \vee (y \wedge \neg f_{reg}) \rrbracket \end{array} \right\}
$$

Notice deleting canonical transfer function $\hat{f}(y) \llbracket y \rrbracket$ in the result by the compression (see Definition 9 on page 4).

Now APLCTRANS can compose the whole first block 1:

$$
\begin{aligned}
\widehat{X}_1 &= \widehat{X}_{1,3} \circledcirc \widehat{X}_{1,2} \circledcirc \widehat{X}_{1,1} \\
&= \widehat{X}_{1,3} \circledcirc \{ \hat{f}(f_{reg}) \llbracket f_{reg} \wedge x \rrbracket \} \circledcirc \{ \hat{f}(f_{reg}) \llbracket 1 \rrbracket \} \\
&= \widehat{X}_{1,3} \circledcirc \{ \hat{f}(f_{reg}) \llbracket x \rrbracket \} \\
&= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket 1 \rrbracket , \\ \hat{f}(x) \llbracket x \wedge f_{reg} \rrbracket , \\ \hat{f}(z) \llbracket y \rrbracket , \\ \hat{f}(m) \llbracket (m \wedge x \wedge f_{reg}) \vee (y \wedge \neg f_{reg}) \rrbracket \end{array} \right\} \\
&\quad \circledcirc \{ \hat{f}(f_{reg}) \llbracket x \rrbracket \} \\
&= \left\{ \begin{array}{c} \hat{f}_{reg} \llbracket 1 \rrbracket , \\ \hat{f}(z) \llbracket y \rrbracket , \\ \hat{f}(m) \llbracket (m \wedge x) \vee (y \wedge \neg x) \rrbracket \end{array} \right\}
\end{aligned}
$$

APLCTRANS terminates and transfer set $\widehat{X}_1$ contains transfer functions describing the operations of whole APLC program.

### 4.2 *Experimental Results*

To analyze the behavior of APLCTRANS, surveyed in Section 4, a test version of algorithm was created. It represents transfer sets as byte arrays and the composition was provided in the form close to symbolic formulas. BED-like structures, depicted in Figure 3, were not used because of several implementation problems but they are preparing for new version of APLCTRANS.

APLCTRANS algorithm implementation gave for the example in Subsection 4.1 the following result written in the notation suitable for text files:

```
Composed in 0.18 seconds.
Result={ @f[(1.(1.x))+(1.!(1.x))],
         m[(m.(1.x))+(((1.y)).!(1.x))],
         z[((((1.x).y)).(1.x))
           +(((1.y)).!(1.x))] }
Minimized in 0.00 seconds.
MinResult={ @f[1], m[x.m+!x.y], z[y] }
```

where symbol '@f' stands for $f_{reg}$. The prefix '@' is reserved code for APLC machine internal variables to distinguish them from all names used in programs.

The test were provided with 19 fragments extracted from real PLC programs, they were written by different programmers. Unfortunately, we could compose only fragments, because the test version has limit $|S| < 256$. Thus the sample is not so weighty as we would wish, and no statistically significant relation can be deduced.

Even if a highly non-optimal representation of transfer sets in memory was used, only one program (with 1990 instructions) produces the exponential growth of data size that has catapulted its conversion time to 954 s. The other programs have conversion time below 51 s (the longest fragment with 8479 instructions). The smallest sample has 438 instructions and shortest conversion time (2.2 s).

The tests with APLCTRANS also revealed that logical minimizations of the final results are many times slower than conversion itself and a partial optimization of formula must be provided during the composition.

## 5. CONCLUSION

In this paper, we presented transfer sets defined on boolean variables. Generally, the principles of concurrent substitution and the composition are preserved for many binary and unary operations, so the transfer set may be extended for expressing them. In this case, the modification of transfers set for composing calls and jumps must be generally provided by conditional assignments, like "$x$ :=*condition* ? *true-value* : *false-value*" known from C language.

The utilization of APLCTRANS for expressing timers was presented in and the result is convertible into a timed automaton.

In sum, our preliminary results are encouraging. We hope that further research will enable the potential advantages of APLCTRANS to apply to a much wider range of PLC programs.

## 6. REFERENCES


Chambers, Colin, Mike Holcombe and Judith Barnard (2001). Introducing X-machine models to verify PLC ladder diagrams. *Computers in Industry* **45**, 277–290.

McMillan, K. L. (1997). *Getting started with SMV*. Cadence Berkeley Labs. Document available at URL: http://www-cad.eecs.berkeley.edu/ kenmcmil/.

Rausch, M. and B. H. Krogh (1998). Formal verification of PLC programs. In: *American Control Conference, Philadelphia, PA, USA*.

Rossi, O. and Ph. Schnoebelen (2000). Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In: *Proc. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM'2000), Dortmund, Germany, Sept. 2000*. Shaker Verlag, Aachen, Germany. pp. 177–182.

Šusta, Richard (2002*a*). Paralel abstraction of PLC program. In: *The 5th International Scientific -Technical Conference Process Control 2002, Říp 2002*. CD R058 1-13.

Šusta, Richard (2002*b*). Verification of PLC Programs. PhD thesis. CTU-FEE Prague. submitted for publication.

Williams, Poul Frederick (2000). Formal Verification Based on Boolean Expression Diagrams. PhD thesis. DTU Tryk, Lyngby.