AC/3 V1.00
A Tool for Automatic Error Correction of Combinatorial Circuits*

Dirk W. Hoffmann and Thomas Kropf

Institute for Computer Design and Fault Tolerance,
Prof. D. Schmid
Universität Karlsruhe, D-76128 Karlsruhe, Germany
hoff@ira.uka.de kropf@ira.uka.de
http://goethe.ira.uka.de/hvg

Abstract

AC/3 is a tool for performing automatic error correction in combinatorial circuits. Two circuits must be provided to the system where one serves as the specification circuit and the other one as the current implementation. **AC/3** tries to prove equivalence between both designs and performs automatic error correction if equivalence does not hold. The tool is based on the rectification theory developed in [7].

keywords: Automatic error correction, equivalence checking, BDDs

1 Introduction

In recent years, formal verification techniques [6] have become more and more sophisticated and for several application domains they have already found their way into industrial environments. Boolean equivalence checking [8, 1, 9], mostly based on BDDs [4, 5], is unquestionably one of these techniques and usually applied during the optimization process to ensure that an optimized circuit still exhibits the same behavior as the original "golden" design. Using BDDs for representing Boolean functions, the verification task mainly consists of creating a BDD for the Boolean function of each output-signal. Then, due to the normal form property of BDDs, both signals implement the same function if and only if they have the same BDD representation. Hence, equivalence can be decided by simply comparing both BDDs.

A lot of professional tools have been proposed in recent years and they have already been able to prove their practical usefulness in a short period of time. Many companies are starting to apply equivalence checking and in a few years this method will undoubtedly be a fully accepted and integrated part of the design cycle.

*This work is supported by the ESPRIT LTR Project 26241

A major requirement of formal methods to be applied successfully in industrial environments is that a verification tool provides useful information even if the verification task fails. Then, the application domain of formal verification is no longer restricted to approve correctness of a specific design, it can also serve as a powerful debugging technique and therefore helps speeding up the whole design cycle.

If equivalence checking fails, most verification tools only allow to compute a counterexample in form of a combination of input values for which the output of the optimized circuit differs from its specification. Therefore, it often remains extremely hard to detect the error causing components. Counter examples as produced by most equivalence checkers can only serve as hints for debugging a circuit and a deeper understanding of the design is still essential.

In recent years, several approaches have been presented for extending equivalence checkers with capabilities not only to compute counter examples, but to locate and rectify errors in the provided design. The applicability of such a method is strongly influenced by the following aspects:

- Which types of errors can be found ?
- Does the method scale to large circuits ?
- How many changes does the computed solution require ?
- Does the method perform well even if both circuits are structurally different ?

AC/3 is a tool for automatic error localization and rectification of combinatorial circuits and based on the rectification theory developed in [7]. Basically, **AC/3** tries to determine the smallest component containing the erroneous parts in the optimized circuit. Once such a component has been localized, a circuit fix is computed and suggested to the designer.

The rectification method implemented in **AC/3** does not assume any error model and therefore, arbitrary design errors can be detected. Moreover, when computing a circuit rectification, **AC/3** tries to incorporate as many sub-parts of the circuit as possible in order to minimize the number of modifications. The underlying method directly works on BDDs, and during the rectification process, only the abstract BDD representation of the specification-circuit is considered. Thus, the success of our algorithm does not depend on any structural similarity between the implementation and the specification.

This paper is organized as follows: In Section 2, we provide a quick tutorial on **AC/3**. The basic steps for rectifying a circuit are illustrated by a small example. Section 3 describes the tools in more detail. A formal definition of the input language including a formally defined semantics is given in Section 3.1 and 3.2. Section 3.3 provides a complete description of all available commands. The currently supported flags are described in Section 3.4. In Section 4, we give a brief description of additional tools of **AC/3**.

2 A Tutorial Example

This Section demonstrates the usage of **AC/3** with a small tutorial example. A complete and more detailed description of the system will be presented in Section 3. To start **AC/3**, simply type

```
checker
```

at the Unix command prompt. If the system has been build successfully, **AC/3** answers with the following message:

```
Circuit-Rectifier V1.00b, build on Fri Dec 11 19:05:33 1998
Dirk W. Hoffmann,  (C)opyright 1998 University of Karlsruhe

Type '?' for help...

Rectifier>
```

Whenever the prompt appears, typing `?` brings up a list of currently available commands:

```
Options:
--------
?                   : this help message
info                : about this program
exit                : exit program
settings            : display current settings
impfile  <file>     : select implementation file
specfile <file>     : select specification file
imppin  <name>      : select implementation out-pin
specpin <name>      : select specification out-pin
viewimp             : view implementation file
viewspec            : view specification file
prove               : start equivalence checking
profile             : print st about parsed files
solution            : select a solution
viewsol             : view selected solution
writesol            : write rectified circuit to a file
set <flag> <value> : set flag
    possible flags and values:
    caching {on|off}
    tempcaching {on|off}
    precomputation {on|off}
    granularity {low|medium|high}
    solution_type {main_inputs|gate_inputs|comp_inputs}
```

In this tutorial section, we will only use some of these commands. A complete description of all commands can be found in Section 3.3.

```
COMPONENT CR_ADDER (a1,a2,b1,b2) --> (c1,c2,c3)
        COMPONENT H_ADDER (a,b) --> (sum,carry)
                sum   := (a <-> b);
                carry := (a /\ b);
        END
        COMPONENT FULL_ADDER (a,b,c) --> (sum,carry)
                sum   := a XOR b XOR c;
                carry := (a /\ b) \/ (a /\ c) \/ (b /\ c);
        END
        H_ADDER.a := a1;
        H_ADDER.b := b1;
        FULL_ADDER.a := a2;
        FULL_ADDER.b := b2;
        FULL_ADDER.c := H_ADDER.carry;
        c1 := H_ADDER.sum;
        c2 := FULL_ADDER.sum;
        c3 := FULL_ADDER.carry;
END
```

Figure 1: Example: A two bit Carry-Ripple-Adder

For this tutorial session, we want to rectify a very small circuit implementing a two-bit carry-ripple adder. The implementation circuit is shown in Fig. 1. The circuit has four global input-signals a_1, a_2, b_1, b_2 and three output signals c_1, c_2, c_3. Using a half-adder (component `H_ADDER`) and a full-adder (component `FULL_ADDER`), the circuit computes the sum $(c_3, c_2, c_1) = (0, a_2, a_1) + (0, b_2, b_1)$.

The specification is shown in Fig. 2. Unlike the implementation, the specification defines its output-signals by Boolean functions being derived directly from the truth-table of Boolean addition.

To load the implementation circuit and the specification circuit, we use the `impfile` and `specfile` command, respectively:

```
Rectifier> impfile carryripple.imp
Parsing file... done
```

```
COMPONENT CR_ADDER (a1,a2,b1,b2) --> (c1,c2,c3)
        c1 := (a1 XOR b1);
        c2 := (a2 XOR b2) XOR (a1 /\ b1);
        c3 := (a2 /\ b2) \/ (a1 /\ b2 /\ b1) \/ (a1 /\ a2 /\ b1);
        END
```

Figure 2: Specification for the 2-bit adder-circuit.

```
file carryripple.imp loaded...

Rectifier> specfile carryripple.spec
Parsing file... done
file carryripple.spec loaded...
```

`carryripple.imp` and `varryripple.spec` are the filenames on disk.

Now, we have to specify a pair of output signals we want to prove equivalent. Output signals can be selected with the `imppin` and `specpin` command. For now, we choose output `c2` in both circuits.

```
Rectifier> imppin c2
Rectifier> specpin c2
```

With the `settings` command, we can display the current configuration:

```
Rectifier> settings
Specification:
--------------
  File:    carryripple.spec
  Out-pin: CR_ADDER.c2
Implementation:
---------------
  File:    carryripple.imp
  Out-pin: CR_ADDER.c2
Solution:
---------
  currently selected: <none>
Flags:
------
bdd caching         : on
temporary caching   : on
precompute solutions: off
solutions formulas reuse signals from  : current component
search granularity                     : low
```

Before starting the rectification process, we change some of the flags. Since we deal with a very small example, we set the `precomputation` flag.

```
Rectifier> set precomputation on
```

This causes the rectifier to immediately compute a solution whenever a rectifiable sub-component has been localized. If the flag is switched off, solution-computation is delayed. Precomputation should only be switched on if the designs to be rectified are not too large since it can considerably slow down the rectification process. We also choose high search-granularity to find a maximum number of solutions.

```
Rectifier> set granularity high
```

To invoke the equivalence checker, simply type `prove`. **AC/3** now creates the BDD representation for the selected output signals and checks for equivalence:

```
Rectifier> prove
Symbolic simulation in progress...
  BDD1: at 0xcdd90 (9 nodes)
  BDD2: at 0xcdd90 (9 nodes)

CR_ADDER.c2 of carryripple.spec and
CR_ADDER.c2 of carryripple.imp are equivalent :-)
Elapsed time: 0 sec.
```

For output `c2`, equivalence has been proven without changing the design. Signal `c3` can also be proven equivalent on the first try. However, for signal c_1, equivalence checking fails and the verification tool tries to rectify the circuit automatically.

```
Rectifier> imppin c1
Rectifier> specpin c1
Rectifier> prove
Symbolic simulation in progress...
  BDD1: at 0xcddb1 (5 nodes)
  BDD2: at 0xcddb0 (5 nodes)

CR_ADDER.c1 of carryripple.spec and
CR_ADDER.c1 of carryripple.imp are different :-(
Elapsed time: 0 sec.

Trying to fix circuit...
Checking result...
Constructing solution...
Calling the construct algorithm...
Number of signals for reuse: 2
done (5nodes)
Rectify completet...

2 possible circuit fixes found
Rectification time: 0 sec.

Total BDD nodes:    51
Garbage collection...
done
Total BDD nodes:    51
```

We can now choose a solution with the `solution` command.

```
Rectifier> solution
(1) : 2 changes in CR_ADDER (0 sec, 3 nodes)
(2) : 2 changes in CR_ADDER.H_ADDER (0 sec, 3 nodes)
```

```
(3) : 2 changes in CR_ADDER (0.01 sec, 3 nodes)
(4) : 2 changes in CR_ADDER.H_ADDER (0 sec, 3 nodes)
(5) : 9 changes in CR_ADDER.H_ADDER (0 sec, 5 nodes))
(6) : 9 changes in CR_ADDER (0 sec, 5 nodes)
type in number: 5
```

We have selected solution 5. The `viewsol` command automatically applies the selected solution to the circuit and displays the rectified design.

```
Rectifier> viewsol
Changes in CR_ADDER

COMPONENT CR_ADDER (a1,a2,b1,b2) --> (c1,c2,c3)
        COMPONENT H_ADDER (a,b) --> (sum,carry)
                sum := ((b /\ ~a) \/ (~b /\ a));
                carry := (a /\ b);
        END
        H_ADDER.a := a1;
        H_ADDER.b := b1;
        COMPONENT FULL_ADDER (a,b,c) --> (sum,carry)
                sum := ((a != b) != c);
                carry := (((a /\ b) \/ (a /\ c)) \/ (b /\ c));
        END
        FULL_ADDER.a := a2;
        FULL_ADDER.b := b2;
        FULL_ADDER.c := H_ADDER.carry;
        c1 := H_ADDER.sum;
        c2 := FULL_ADDER.sum;
        c3 := FULL_ADDER.carry;
END
```

The circuit has been modified in component `H_ADDER` by changing the definition of signal `sum`.

Table 1 shows a complete list of all computed solutions. The second and third column contain name of the sub-component and name of the signal to be modified, respectively. Column 4 shows the old signal definition while column 5 contains the suggested replacement. Comparing the rectified circuit with the original implementation in Fig. 1, it turns out that the major design error has been made in component `H_ADDER`. Output-signal `sum` computes a false value due to a wrong logical connective. Instead of performing an XOR-operation, the equivalence-operator is applied. Solution 5 exactly suggests to replace this logical connective, but all other solutions also fix the circuit even if some of them actually do not reflect the designer's original intention. Since the verification tool does not have any semantical knowledge about the half-adder, it cannot distinguish between these solutions. In general, the solution that requires the minimal number of changes is considered best. Solutions 1 to 4 prove that the circuit can even be rectified by inserting one additional NOT gate, only.

Using the `writesol` command allows to write back the rectified circuit to a file. After saving the circuit, **AC/3** can be quitted by typing `exit`.

Nr	Component	Signal	old definition	suggested replacement
1	H_ADDER	sum	a $\leftrightarrow$ b	a $\leftrightarrow \neg$b
2	H_ADDER	sum	a $\leftrightarrow$ b	$\neg$a $\leftrightarrow$ b
3	CR_ADDER	H_ADDER.b	b1	$\neg$b1
4	CR_ADDER	H_ADDER.a	a1	$\neg$a1
5	H_ADDER	sum	a $\leftrightarrow$ b	(a $\wedge \neg$b) $\vee$ ($\neg$a $\wedge$ b)
6	CR_ADDER	c1	H_ADDER.sum	$\neg$H_ADDER.sum

Table 1: Complete list of all computed circuit fixes for the carry-ripple adder.

```
Rectifier> writesol
Enter file name: carryripple.fix
Rectifier> exit
Have a nice day!
```

3 Tool Description

The following description refers to **AC/3 V1.00**. **AC/3** has been written in C++ and documented via the DOC++ standard. **AC/3** should compile on every UNIX-platform and every C++ compiler supporting the ANSI standard. D.E. Long's BDD library is required for compilation and can be downloaded freely.

In particular, we have successfully build the system with the following configuration:

Solaris	5.6 and 5.5.1	
gcc,g++	2.7.2	
flex	2.5.4	
bison	1.22	
BDD lib	dated 06/11/93	(package by D. E. Long)
DOC++	3.01	(for extracting the developer-documentation)

For installing the system, please refer to the installation notes that come with the system. To start **AC/3**, simply type

```
checker
```

at the command prompt.

In the next section, we describe the input language of **AC/3 V1.00** in more detail. Section 3.3 gives a detailed description of the available commands, and Section 3.4 describes currently available flags.

3.1 The Input Language

A first impression of the input language of **AC/3** has been given in Fig. 1 and Fig. 2. Basically, every input file describes a combinatorial circuits in form of hierarchical net list. To achieve a hierarchical description, several *components*

can be declared, each consisting of an *interface declaration* and a *component body*. Input and output signals are declared in the interface declaration. In the component body, sub-components can be declared together with Boolean formulas defining the component's behavior.

Throughout the input file, comments can be inserted by typing two slashes. After `//`, everything is ignored until a new-line character occurs. Spaces, tabulators, and blank lines can occur everywhere and are skipped by the parser.

We now give a more precise definition of the input-language using regular expressions and BNF notation.

Identifiers are used to specify names. An identifier can represent a component name, a signal name, or a name of an external variable. Formally, we define

$$\text{<ident>} \quad ::= \quad \texttt{[0-9a-zA-Z\_]}^{+}$$

We explicitely make the exception that identifiers must not be a reserved word. Reserved words are `COMPONENT`, `EXTERN`, `END`, `NOT`, `AND`, `OR`, `XOR`, `IMP`, and `EQUIV`.

References represent names of signals.

$$\begin{array}{lcl} \text{<reference>} & ::= & \text{<ident>} \\ & | & \text{<ident> . <ident>} \end{array}$$

The right most identifier is the signal name and the optional identifier can be used to specify a distinct component where the signal occurs in.

Expressions represent Boolean functions and are defined as follows:

<expr>	::=	`true`	
	|	`false`	
	|	<reference>	(internal signal)
	|	EXTERN <ident>	(external input)
	|	$\sim$ <expr>	(negation)
	|	NOT <expr>	(negation)
	|	<expr> /\ <expr>	(conjunction)
	|	<expr> AND <expr>	(conjunction)
	|	<expr> \/ <expr>	(disjunction)
	|	<expr> OR <expr>	(disjunction)
	|	<expr> - > <expr>	(implication)
	|	<expr> IMP <expr>	(implication)
	|	<expr> <-> <expr>	(logical equivalence)
	|	<expr> EQUIV <expr>	(logical equivalence)
	|	<expr> XOR <expr>	(exclusive-or)
	|	(<expr>)	

Expressions can either be a reference to a signal, an external input (keyword EXTERN), or a combination of one or more expressions with a logical connective.

Assignments allow to assign an expression to a signal:

$$\text{<assignment>} \quad ::= \quad \text{<reference> := <expr> ;}$$

Components are the core objects of the input language. As mentioned before, each component consists of an interface declaration, a list of sub components and a list of assignments. Component declarations can be arbitrarily nested which leads to hierarchical circuit descriptions.

```
<component>   ::=   <interface>
                    <component>*    (sub components)
                    <assignment>*   (assignments)
                    END
```

The *component interface* consists of a component name, a declaration of input signals, and a declaration of output signals:

```
<interface>   ::=   COMPONENT <ident> (<id_list>) --> (<id_list>)

<id_list>     ::=   <ident>
                  |   <ident> , <id_list>
```

Finally, a valid input file consists of the declaration of a single component which is called the *main component*:

```
<input-file>   ::=   <component>   (main component)
```

The input signals of the main component are implicitly considered to be external inputs. In all other components, external signals have to be defined using the EXTERN keyword.

3.2 Formal Semantics

In this section, we provide a formal semantics for the input language defined in the previous section. The semantics is given in form of a function that maps a given input-file p onto a corresponding Boolean formula $[\![p]\!]_A$ describing the output-signals of p. $[\![p]\!]_A$ is called the *denotation* of p. The additional subscript A is a simple string carrying the component-name where p is defined in. If the subscript is omitted, we implicitly assume A to be the empty string.

We use a *syntax directed* semantics which means that the semantics of a program-construct is defined in terms of the semantics of its syntactic components. Formally, we define the denotation of **AC/3** input-files as follows:

- *Identifiers:*

$$[\![\ ident\ ]\!]_A \quad := \quad A.ident$$

  Note that $A.ident$ is the name of a single variable. The prefix A is added to the variable name in order to avoid name clashes between variables defined in different components.

- *References:*

$$\begin{aligned} [\![\ ident\ ]\!]_A &:= A.ident \\ [\![\ ident1\ .\ ident2\ ]\!]_A &:= A.ident1.ident2 \end{aligned}$$

- *Expressions:*

$$
\begin{aligned}
[\![\texttt{true}]\!]_A &:= \texttt{true} \\
[\![\texttt{false}]\!]_A &:= \texttt{false} \\
[\![\texttt{EXTERN } \mathit{ident}\,]\!]_A &:= \mathit{ident} \\
[\![\sim\ \mathit{expr}\,]\!]_A &:= \neg[\![\,\mathit{expr}\,]\!]_A \\
[\![\texttt{NOT } \mathit{expr}\,]\!]_A &:= \neg[\![\,\mathit{expr}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{/\textbackslash}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \wedge [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{AND}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \wedge [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{\textbackslash/}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \vee [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{OR}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \vee [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{->}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \rightarrow [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{IMP}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \rightarrow [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{<->}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \leftrightarrow [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{EQUIV}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \leftrightarrow [\![\,\mathit{expr2}\,]\!]_A \\
[\![\,\mathit{expr1}\ \texttt{XOR}\ \mathit{expr2}\,]\!]_A &:= [\![\,\mathit{expr1}\,]\!]_A \oplus [\![\,\mathit{expr2}\,]\!]_A \\
[\![\texttt{(}\ \mathit{expr}\ \texttt{)}]\!]_A &:= ([\![\,\mathit{expr}\,]\!]_A)
\end{aligned}
$$

  The logical operators $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\oplus$ stand for logical conjunction, disjunction, implication, equivalence, and exclusive-or, respectively, and are defined by their usual truth tables.

- *Assignments:*

$$[\![\,\mathit{reference} := \mathit{expr}\,]\!]_A := [\![\,\mathit{reference}\,]\!]_A = [\![\,\mathit{expr}\,]\!]_A$$

- *Components:*

$$
\begin{aligned}
[\![\,\mathit{interface}\ \ \mathit{cmp\_list}\ \ \mathit{ass\_list}\,]\!]_A := {} & [\![\,\mathit{cmp\_list}\,]\!]_{A.name} \\
& \wedge [\![\,\mathit{ass\_list}\,]\!]_{A.name}
\end{aligned}
$$

  where *name* is the component-name specified in the interface.

- *Lists of components:*

$$
\begin{aligned}
[\![\,\mathit{empty\ list}\,]\!]_A &:= \texttt{true} \\
[\![\,\mathit{comp1} \mid \mathit{rest}\,]\!]_A &:= [\![\,\mathit{comp1}\,]\!]_A \wedge [\![\,\mathit{rest}\,]\!]_A
\end{aligned}
$$

- *Lists of assignments:*

$$
\begin{aligned}
[\![\,\mathit{empty\ list}\,]\!]_A &:= \texttt{true} \\
[\![\,\mathit{ass1} \mid \mathit{rest}\,]\!]_A &:= [\![\,\mathit{ass1}\,]\!]_A \wedge [\![\,\mathit{rest}\,]\!]_A
\end{aligned}
$$

- *Input file:*

  Before defining the semantics of a complete AC/3 program, we introduce the $sigs_A$ function mapping components to sets of variables:

$$sigs_A(\ name\ :(i_1,\ldots,i_n)\texttt{-->}(o_1,\ldots,o_m)\ \ cmp\_list\ \ ass\_list\ ) := \{A.i_1,\ldots,A.i_n,A.o_1,\ldots,A.o_m\} \cup sigs_{A.name}(\ cmp\_list\ )$$

  $sigs(C)$ collects the names of all signals occurring either in C itself or in one of its sub-components. We extend the definition to component-lists as usual:

$$\begin{aligned} sigs_A(\ empty\ list\ ) &:= \emptyset \\ sigs_A(\ comp1 \mid rest\ ) &:= sigs_A(\ comp1\ ) \cup sigs_A(\ rest\ ) \end{aligned}$$

  Using the *sigs* function, we define the semantics of a complete input-file as

$$\begin{aligned} [\![\ interface\ \ cmp\_list\ \ ass\_list\ ]\!] \ :=\ & \exists s_1,\ldots,s_k : \\ & [\![\ cmp\_list\ ]\!]_{name} \\ & \wedge [\![\ ass\_list\ ]\!]_{name} \end{aligned}$$

  where *name* is the component-name specified in the interface and $(s_1,\ldots,s_k) = sigs_{name}(\ cmp\_list\ )$. In the resulting formula, all intermediate signals are hidden by the "$\exists$" operator denoting standard existential quantification for Boolean formulas, i.e.,

$$(\exists x : f) = (f[x \leftarrow 0] \vee f[x \leftarrow 1])$$

  Thus, the denotational semantics for an AC/3 program is a Boolean formula only containing the input and output signals of the *main* component as variables.

3.3 Program Commands

After **AC/3** has been started, the following message comes up:

```
Circuit-Rectifier V1.00b, build on Fri Dec 11 19:05:33 1998
Dirk W. Hoffmann,  (C)opyright 1998 University of Karlsruhe

Type '?' for help...

Rectifier>
```

This is the text interface of **AC/3**. A graphical user interface is currently under development. Entering "`?`" at the command prompt brings up a summary of all available commands for driving the equivalence checker and rectification engine. The following commands are currently supported:

- `exit`

- `impfile`
- `info`
- `profile`
- `prove`
- `set`
- `settings`
- `solution`
- `specfile`
- `specpin`
- `viewimp`
- `viewsol`
- `viewspec`
- `writesol`

Now, we describe the commands in more detail:

`exit`

Parameters: *none*
Purpose: Terminates **AC/3**.
Requires: —

`impfile`

Parameters: name
Purpose: Loads the specified implementation circuit from disc. The input file must follow the language definition given in Section 3.1
Example: `impfile carryripple.imp`
Requires: —

`imppin`

Parameters: name
Purpose: Selects a signal in the implementation circuit that is going to be compared with the corresponding signal in the specification circuit.
Example: `imppin c1`

Requires: Implementation circuit has already been loaded.
(see `impfile`-command)

`info`

Parameters: *none*

Purpose: Shows information about this program.

Requires: —

`profile`

Parameters: *none*

Purpose: Shows detailed information about the currently loaded circuits. This function has been implemented for debugging purposes, but can also be invoked directly.

Requires: —

`prove`

Parameters: *none*

Purpose: This command calls the core algorithms of **AC/3**. First, the selected output signals in the specification and implementation circuit are checked for equivalence. If equivalence does not hold, the rectification engine is invoked automatically. For a detailed description of the underlying algorithms, see [7].

Requires: Former calls to `impfile`, `specfile`, `imppin`, `specpin`

`set`

Parameters: flag value

Purpose: This command allows to set a flag to the specified value. A list of available flags together with a description of their meaning is provided in Section 3.4.

Example: `set solution_type gate_inputs`

Requires: —

`settings`

Parameters: *none*

Purpose: Shows the current program configuration. Displayed items include name of loaded circuits, name of selected signals, and current flag-values. After a flag is changed, this command is invoked automatically.

Requires: —

`solution`

Parameters: *none*

Purpose: This command displays the list of computed solutions and allows the user to choose a specific circuit fix which is then applied to the implementation circuit.

Requires: Former call to `prove`

`specfile`

Parameters: name

Purpose: Loads the specified specification circuit from disc. The input file must follow the language definition given in Section 3.1

Example: `specfile carryripple.spec`

Requires: —

`specpin`

Parameters: name

Purpose: Selects a signal in the specification circuit that is going to be compared with the corresponding signal in the implementation circuit.

Example: `specpin c1`

Requires: Specification circuit has already been loaded.
(see `specfile`-command)

`viewimp`

Parameters: *none*

Purpose: Prints the description of the implementation circuit.

Requires: Implementation circuit has already been loaded.

`viewsol`

Parameters: *none*

Purpose: Prints the rectified implementation circuit. This command may only be used after a solution has been selected.

Requires: Former call to the `solution`-command.

`viewspec`

Parameters: *none*

Purpose: Prints the description of the specification circuit.

Requires: Specification circuits has already been loaded.

`writesol`

Parameters: name

Purpose: Writes the rectified implementation circuit back to a file. The file name must be provided as argument.

Example: `writesol carryripple.fixed`

Requires: Some solution must have been selected. (see `solution`-command)

3.4 Flags

AC/3 provides various flags which influence the rectification process. At the moment, the following flags are supported:

- `caching`
- `tempcaching`
- `solution_type`
- `precomputation`
- `granularity`

The current value of the flags can be displayed with the `settings` command and changes with the `set` command (see Section 3.3).

We now describe all supported flags in detail. For each flag, possible values, default value, and meaning is described:

`caching`

Possible values: `on`, `off`

Default value: `on`

Purpose: This flag enables or disables the global caching mechanism for BDDs. Caching of BDDs is used when traversing the circuit net list and can dramatically decrease computation time.
Since caching does not influence the computed results, it should only be switched off for debugging purposes. This flag only influences caching in the rectification engine and does not influence caching performed within the underlying BDD package.

Warning: Disabling the caching mechanism can cause an exponential blow-up in runtime.

`tempcaching`

Possible values: `on`, `off`

Default value: `on`

Purpose: This flag enables or disables the temporary caching mechanism for BDDs. A temporary cache is used in addition to the global cache in some functions of the rectification engine. This can further decrease computation time considerably.

Since caching does not influence the computed results, it should only be switched off for debugging purposes. This flag only influences caching in the rectification algorithm and does not influence caching performed within the underlying BDD package.

Warning: Disabling the caching mechanism can cause an exponential blow-up in runtime.

`solution_type`

Possible values: `main_inputs`, `gate_inputs`, `comp_inputs`

Default value: `comp_inputs`

Purpose: This flag influence the structure of the computed circuit rectifications. To keep modifications small, **AC/3** tries to compute solutions that reuse as many signals of the old circuit as possible. The solution type determines the signals that are going to be reused. The user can choose out of three possible solution types:

`main_inputs`: The solution formula is constructed out of external signals, only. Thus, there is no reuse of any intermediate signal. This solution type should only be chosen if the implementation circuit is a flat design and does not exhibit any hierarchy. This solution type can also be used for debugging purposes since it excludes the call to the construct-algorithm (see [7] for details).

`gate_inputs`: The solution formula is constructed out of external signals and the immediate input signals to the part of the circuit that is going to be substituted.

`comp_inputs`: The solution formula is constructed out of external inputs and the input signals of the component where the modification occurs.

`precomputation`

Possible values: `on`, `off`

Default value: `off`

Purpose: If precomputation is enabled, the rectification-algorithm immediately computes a solution whenever a rectifiable subcomponent has been localized. If precomputation is disabled, solution-computation is delayed and only computed after a specific solution has been selected with the `solution` command described in Section 3.3. Precomputation can be switched off for rectifying large designs. This can accelerate the rectification process considerably. However, to estimate the quality of a solution, we have to count the number of modifications that

have to be applied to the implementation circuit. This can only be done after the solution formulas have explicitely been computed.

`granularity`

Possible values: `low`, `medium`, `high`

Default value: `low`

Purpose: The search granularity determines which parts of the implementation circuit are checked for rectifiability. We distinguish three types of different circuit-rectifications:

1. rectifications that substitute an output-signal of some component
2. rectifications that substitute an input-signal of some component
3. rectifications that substitute an inner part of a component

According to the selected search granularity, only rectifications of a specific type are computed. In particular:

- `low` granularity only computes type 1 rectifications
- `medium` granularity only computes type 1 and type 3 rectifications
- `high` granularity computes all types of solutions

4 Additional Tools

AC/3 provides two additional tools for

- statistical analysis of the input files
- converting ISCAS89 format

Statistical Analysis

Usage: `statistics file`

Purpose: The statistics tool parses the specified input file and first performs some consistency checks. Besides searching missing signal definitions, the input circuit is checked for loop-freeness. After checking consistency, some statistical information is computed, i.e.,

- the number of logical connectives,
- the number of internal references, and
- the input cones[1]

[1] the set of external input variables occuring in the definition of the observed signal.

are determined for each output signal of the main component.

Converter from ISCAS89 format

Usage: `conv89 file`

Purpose: Reads in the specified file in ISCAS89 format and prints it in the input language of **AC/3** to stdout. See Section 3.1 for a detailed description of the input language. The ISCAS89 converter allows to get access to a broad range of circuits (e.g., the ISCAS85 benchmarks [3] and the Berkeley benchmarks circuits [2]).

A Copyright Notice, License, and Disclaimer

AC/3 V1.00b
==========

Author: Dirk Hoffmann
email: hoff@ira.uka.de
WWW: http://goethe.ira.uka.de/~hoff

References

[1] D. Brand. Verification of Large Synthesized Designs. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 534–537, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[2] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1986.

[3] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN. In *Int. Symposium on Circuits and Systems, Special Session on ATPG and Fault Simulation*, 1985.

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

[5] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] A. Gupta. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.

[7] D. W. Hoffmann and T. Kropf. Using BDD-based decomposition for automatic error correction of combinatorial circuits. Technical Report 6/99, University of Karlsruhe, March 1999.

[8] Alan J. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 677–682, October 1997.

[9] S.M. Reddy, W. Kunz, and D.K. Pradhan. Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment. In *ACM/IEEE Design Automation Conference*, pages 414–419, 1995.