Hardware-Verification using First Order BDDs\textsuperscript{1}

Klaus Schneider*, Ramayya Kumar**, and Thomas Kropf* 

*Institut für Rechenerentwurf und Fehlertoleranz, (Prof. Dr. D. Schmid), Universitàt Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany, e-mail:{schneide,kropf} @ira.uka.de 
** Forschungszentrum Informatik, Haid-und-Neustraße 10-14 76131 Karlsruhe, Germany, e-mail:kumar @fzi.de 

Abstract

Binary decision diagrams (BDDs) are a well known method for representing and comparing boolean functions. Although BDDs are known to be very compact, in all known approaches for hardware verification, BDD-based calculi are restricted to propositional logic. This logic is insufficient for the verification of abstract data types, time abstraction and also for hierarchical verification. In this paper, the lifting of graphs based on shannon expansions and the related binary decision diagrams to first order logic is described and the soundness and correctness theorems are stated. The power of these techniques in the domain of hardware verification is shown by a case study using a hierarchical circuit.

Keyword Codes: I.2.3; F.4.1
Keywords: Hardware Verification; Deduction and Theorem Proving; Mathematical Logic

1 Introduction

Most automated approaches to hardware-verification are limited to propositional logic or temporal extensions of it (e.g. [BCMD90]), since these logics are decidable. Almost all such approaches rely on binary decision diagrams (BDDs) as the underlying representation form, due to the compactness of the representation and the efficiency of the related proof techniques. However, such methods suffer from a lack of expressiveness for describing high-level specifications, which can be conveniently done in first- or even better in higher-order logic. For example, the use of complex datatypes such as natural numbers, integers, lists, stacks, etc. and hierarchical verification supplemented with data and time abstractions are not supported in a satisfactory way by these logics. Furthermore, the size of the

\textsuperscript{1}This work has been partly financed by a German national grant, project Automated System Design, SFB No.358
formulae to be proved in first-order logic is much smaller and they are more readable than the propositional ones. A major advantage over propositional logic is the capability of specifying and proving the correctness of generalized \( n \)-bit implementations for regular structures once and for all, as opposed to repeated proofs for specific bitwidths [SeKK92c].

For these reasons, approaches in first and higher-order logic become more and more attractive. These logics provide a sufficient level of expressiveness, but they are in general undecidable. Fortunately, first order logic is semidecidable and a lot of proof calculi have been formulated for it up to now. However, the calculi based on tableau or sequent calculus, are quite inefficient in terms of space and time requirements. Although proofs in the domain of hardware verification are not very difficult to perform, they are highly memory intensive. This has motivated us to consider BDDs for first-order logic, so as to allow compact representations and to obtain efficient proofs.

In this paper, we present how graphs based on shannon expansions and the related binary decision diagrams can be lifted to first order logic in order to obtain automated theorem proving procedures for it. In section 3, we show the verification of a hierarchical circuit which uses abstract datatypes which correspond to operations over natural numbers. The section 4 gives some experimental results and finally the paper is concluded with a summary.

# 2 First Order BDD-like Calculi

In this section we define two main approaches that we have investigated for first order proof calculi, based on binary decision diagrams: the first order shannon graph calculus \( C_{SG} \), and the first order binary decision diagram calculus \( C_{BDD} \) both of which use graphs as the underlying representation of the formula. We also introduce a more efficiently implementable version of \( C_{BDD} \) called \( C_{ORB} \) which is based on ordered reduced BDDs.

Before we formalize the ideas, we list the notation which is used (for detailed definitions see any logic textbook, for example [Fitl90]):

\[
\begin{align*}
V & \quad \text{set of object variables} \\
T_\Sigma & \quad \text{set of first order terms over a signature } \Sigma = (P_\Sigma, F_\Sigma, \alpha_\Sigma), \\
& \quad \text{where } P_\Sigma, F_\Sigma \text{ denote the set of predicate and function symbols,} \\
& \quad \text{respectively and } \alpha_\Sigma : P_\Sigma \cup F_\Sigma \rightarrow \text{IN} \text{ is the arity function} \\
At_\Sigma & \quad \text{set of atomic formulae over a signature } \Sigma \\
For_\Sigma & \quad \text{set of first order formulae over a signature } \Sigma \\
\text{true,false} & \quad \text{truth values} \\
[\varphi]_x^\tau & \quad \text{substitution of the variable } x \text{ in the term or formula } \varphi \text{ by } \tau \\
\xi^\mu_\varphi & \quad \text{fixed assignment: } \xi^\mu_\varphi(\mu) := d \text{ and } \xi^\mu_\varphi(x) := \xi(\mu) \text{ if } x \neq \mu \\
\omega^{D,I,\xi}() & \quad \text{evaluation function of an interpretation } (D, I) \text{ where } D \text{ is the domain,} \\
& \quad I \text{ is the interpretation function, and } \xi \text{ an assignment} \\
\wp(M) & \quad \text{set of all subsets of a set } M
\end{align*}
\]
2.1 The Shannon Graph Calculus $\mathcal{CSG}$

The Shannon graph calculus $\mathcal{CSG} = (\mathcal{SG}_\Sigma, \mathcal{CSG}_\Sigma, \{ \vdash^e \})$ consists of a formal language $\mathcal{SG}_\Sigma$, a subset $\mathcal{CSG}_\Sigma$ of it containing the axioms of $\mathcal{CSG}$ and the only rule $\vdash^e$ which maps elements of the language $\mathcal{SG}_\Sigma$ to other elements of it. In general, a proof in $\mathcal{CSG}$ proceeds as follows: First, for the given first order formula a corresponding element of $\mathcal{SG}_\Sigma$ is computed, then the rule $\vdash^e$ is applied until an axiom belonging to $\mathcal{CSG}_\Sigma$ is obtained. However, implementations of $\mathcal{CSG}$ are based on graphs and work more sophisticatedly as described later. These graphs intuitively correspond to case splits on each atomic formula assuming it to be true or false and stem from Shannon expansions of the formula. However, quantified formulae have to be handled differently.

First we define the language $\mathcal{SG}_\Sigma$ of the calculus, which uses the operator $(\varphi \mapsto \varphi_0 \mid \varphi_1)$, interpreted as if not $\varphi$ then $\varphi_0$ else $\varphi_1$.

**Definition 2.1 (First Order Shannon Graphs $\mathcal{SG}_\Sigma$)** The following rules define the set of first order Shannon graphs $\mathcal{SG}_\Sigma$ over a given signature $\Sigma$:

1. $0, 1 \in \mathcal{SG}_\Sigma$, where 0, 1 are called the 0- and 1-graph, respectively.
2. $(\varphi \mapsto S_0 \mid S_1) \in \mathcal{SG}_\Sigma$, where $\varphi \in \mathcal{AL}_\Sigma$ and $S_0, S_1 \in \mathcal{SG}_\Sigma$.
3. $((S)_\mu \mapsto S_0 \mid S_1) \in \mathcal{SG}_\Sigma$, where $S, S_0, S_1 \in \mathcal{SG}_\Sigma$ and $\mu \in V$.

In the first order Shannon graph $((S)_\mu \mapsto S_0 \mid S_1)$, $(S)_\mu$ is called a quantified subgraph and $S$ is said to be bounded by the variable $\mu$. In $(\varphi \mapsto S_0 \mid S_1)$ and $((S)_\mu \mapsto S_0 \mid S_1)$, $S_0$ is called the false-subgraph and $S_1$ is called the true-subgraph.

In graphical representations and in the implementation as well, we use structure sharing to avoid exponential growth of the graphs. This structure sharing has the consequence that each Shannon-graph has at most one 1-leaf and at most one 0-leaf. However, this structure sharing is not visible in definition 2.1. For example, figure 1 shows the pictorial representation of the Shannon graph $(((P \mu \mapsto 0 \mid (Q \mu \mapsto 0 \mid 1))_\mu \mapsto 0 \mid (P a \mapsto 1 \mid (Q b \mapsto 1 \mid 0)))$. The edges labelled with ‘+’ lead to the true-subgraph and the edges labelled with ‘−’ lead to the false-subgraph.

The next definition shows how first order Shannon graphs of a given formula which are called the initial graphs can be computed. In order to define these graphs, we first have to define the append operator $\&$. $\& (S, S_0, S_1)$ simultaneously replaces each 0 subgraph in $S$ by $S_0$ and each 1 subgraph of $S$ by $S_1$ as shown in figure 2.

**Definition 2.2 (First Order Shannon Graph of a formula)** Formally, $\& : \mathcal{SG}_\Sigma \times \mathcal{SG}_\Sigma \times \mathcal{SG}_\Sigma \rightarrow \mathcal{SG}_\Sigma$ is defined recursively as follows:

\[
\begin{align*}
\& (0, S_0, S_1) & := S_0 \\
\& (1, S_0, S_1) & := S_1 \\
\& (\varphi \mapsto B_0 \mid B_1, S_0, S_1) & := (\varphi \mapsto \& (B_0, S_0, S_1) \mid \& (B_1, S_0, S_1)) \\
\& ((S)_\mu \mapsto B_0 \mid B_1, S_0, S_1) & := ((S)_\mu \mapsto \& (B_0, S_0, S_1) \mid \& (B_1, S_0, S_1))
\end{align*}
\]
The functions \( \mathfrak{S}_0, \mathfrak{S}_1 : \mathcal{F}_\Sigma \to \mathcal{G}_\Sigma \) is obtained from \( \Sigma \) by adding some skolem function symbols) assign to each formula \( \Phi \in \mathcal{F}_\Sigma \), first order shannon graphs \( \mathfrak{S}_0(\Phi) \) and \( \mathfrak{S}_1(\Phi) \), respectively, according to the following rules:

\[
\begin{align*}
\mathfrak{S}_1(P(\tau_1, \ldots, \tau_n)) &:= P(\tau_1, \ldots, \tau_n) \mapsto 0 | 1 \quad \mathfrak{S}_1(P(\tau_1, \ldots, \tau_n)) := P(\tau_1, \ldots, \tau_n) \mapsto 1 | 0 \\
\mathfrak{S}_1(\neg \varphi) &:= \mathfrak{S}_0(\varphi) \quad \mathfrak{S}_0(\neg \varphi) := \mathfrak{S}_1(\varphi) \\
\mathfrak{S}_1(\varphi \land \psi) &:= \mathfrak{N}(\mathfrak{S}_1(\varphi), 0, \mathfrak{S}_1(\psi)) \quad \mathfrak{S}_0(\varphi \land \psi) := \mathfrak{N}(\mathfrak{S}_0(\varphi), 0, \mathfrak{S}_0(\psi), 1) \\
\mathfrak{S}_1(\varphi \lor \psi) &:= \mathfrak{N}(\mathfrak{S}_1(\varphi), 1, \mathfrak{S}_1(\psi)) \quad \mathfrak{S}_0(\varphi \lor \psi) := \mathfrak{N}(\mathfrak{S}_0(\varphi), 0, \mathfrak{S}_0(\psi)) \\
\mathfrak{S}_1(\forall x. \varphi) &:= ((\mathfrak{S}_1([\varphi]^x_{\mu_1, \ldots, \mu_n}))_x \mapsto 0 | 1) \quad \mathfrak{S}_0(\forall x. \varphi) := (\mathfrak{S}_0([\varphi]^x_{\mu_1, \ldots, \mu_n}))_x \mapsto 0 | 1 \\
\mathfrak{S}_1(\exists x. \varphi) &:= \mathfrak{S}_1([\varphi]^x_{\mu_1, \ldots, \mu_n}) \\
\mathfrak{S}_0(\varphi) & \text{ is called the negative initial graph and } \mathfrak{S}_1(\varphi) \text{ is called the positive initial graph.}
\end{align*}
\]

\[\mathfrak{N}(\begin{array}{c}
S \\
0 \ 1
\end{array}, \begin{array}{c}
S_0 \\
0 \ 1
\end{array}, \begin{array}{c}
S_1 \\
0 \ 1
\end{array}) := \begin{array}{c}
S \\
0 \ 1
\end{array}\]

Figure 2: Definition of the append operator \( \mathfrak{N} \).

It has to be noted that the two initial graphs \( \mathfrak{S}_0(\varphi) \) and \( \mathfrak{S}_1(\varphi) \), differ only in the leaf nodes: When the 1 and the 0-leaves of one of the initial graphs are exchanged, the other initial graph is obtained, i.e. \( \mathfrak{S}_0(\varphi) = \mathfrak{N}(\mathfrak{S}_1(\varphi), 0, 1) \) and \( \mathfrak{S}_1(\varphi) = \mathfrak{N}(\mathfrak{S}_0(\varphi), 1, 0) \). The negative initial graph of \( (\forall x. Px \land Qx) \rightarrow (Pa \land Qb) \) is shown in figure 1.

The calculus \( \mathcal{C}_{\mathcal{G}} \) is a path-oriented first order proof calculus. Therefore we have to introduce the notion of paths in the graphs of \( \mathcal{G}_\Sigma \).

**Definition 2.3 (Paths of a First Order Shannon Graph)**

The path sets \( \mathcal{P}\mathfrak{S}_0(S) \) and \( \mathcal{P}\mathfrak{S}_1(S) \) of a first order shannon graph \( S \in \mathcal{G}_\Sigma \) corresponding to the paths leading to the 0-leaf and 1-leaf, respectively, are defined as follows:

\[
\begin{align*}
\mathcal{P}\mathfrak{S}_0(0) &:= \{\} \\
\mathcal{P}\mathfrak{S}_0(1) &:= \{\} \\
\mathcal{P}\mathfrak{S}_0(\varphi \rightarrow \mathfrak{B}_0 | \mathfrak{B}_1) &:= \{\neg \varphi \} \cup \{p \mid p \in \mathcal{P}\mathfrak{S}_0(\mathfrak{B}_0)\} \cup \{\varphi \} \cup \{p \mid p \in \mathcal{P}\mathfrak{S}_0(\mathfrak{B}_1)\} \\
\mathcal{P}\mathfrak{S}_0((S)_\mu \rightarrow \mathfrak{B}_0 | \mathfrak{B}_1) &:= \{P_1 \cup P_2 \mid P_1 \in \mathcal{P}\mathfrak{S}_0(S), P_2 \in \mathcal{P}\mathfrak{S}_0(\mathfrak{B}_0)\} \\
&\quad \cup \{P_1 \cup P_2 \mid P_1 \in \mathcal{P}\mathfrak{S}_1(S), P_2 \in \mathcal{P}\mathfrak{S}_0(\mathfrak{B}_1)\}
\end{align*}
\]

\[\text{We excluded the equivalence operator from our logical language. However, it is straightforward to add the rules: } \mathfrak{S}_1(\varphi \leftrightarrow \psi) := \mathfrak{S}_1((\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi)) \quad \mathfrak{S}_0(\varphi \leftrightarrow \psi) := \mathfrak{S}_0((\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi))\]
\[ \mathcal{PS}_1(0) := \{\} \]
\[ \mathcal{PS}_1(1) := \{\{\}\} \]
\[ \mathcal{PS}_1((\varphi \mapsto B_0 \mid B_1)) := \{\{\neg\varphi\} \cup p \mid p \in \mathcal{PS}_1(B_0)\} \cup \{\{\varphi\} \cup p \mid p \in \mathcal{PS}_1(B_1)\} \]
\[ \mathcal{PS}_1(((S)_\mu \mapsto B_0 \mid B_1)) := \{p_1 \cup p_2 \mid p_1 \in \mathcal{PS}_1(S), p_2 \in \mathcal{PS}_1(B_0)\} \]
\[ \cup \{p_1 \cup p_2 \mid p_1 \in \mathcal{PS}_1(S), p_2 \in \mathcal{PS}_1(B_1)\} \]

If one reaches a 0-leaf while traversing a path in a quantified subgraph, the traversal proceeds into the false-graph of the quantified subgraph. Analogously, if a 1-leaf is reached in a quantified subgraph, one proceeds in the true-subgraph of this quantified subgraph. For example, the negative initial graph of \((\forall x.Px \land Qx) \rightarrow (Pa \land Qb)\) given in figure 1 has the path sets as shown below:

\[ \mathcal{PS}_0(S) = \{\neg P\mu\}, \{P\mu, \neg Q\mu\}, \{P\mu, Q\mu, Pa, Qb\} \]
\[ \mathcal{PS}_1(S) = \{P\mu, Q\mu, Pa, \neg Qb\}, \{P\mu, Q\mu, \neg Pa\} \]

The next definition characterizes the axioms of the calculus \(\mathcal{CSG}\).

**Definition 2.4 (Closed First Order Shannon Graphs \(\mathcal{CSG}\))** A shannon graph \(S \in \mathcal{CSG}\) is closed, if there is a substitution \(\sigma\) such that each path \(p \in \mathcal{PS}_1(S)\) contains two formulae \(\varphi_p, \neg \psi_p\) with \(\sigma(\varphi_p) = \sigma(\psi_p)\). \(\mathcal{CSG}\) denotes the set of all closed shannon graphs over a signature \(\Sigma\).

The instantiation of a shannon graph \(S \in \mathcal{CSG}\) with a given substitution \(\sigma\) can be defined as follows:

**Definition 2.5 (Instantiation of Shannon Graphs)** For any substitution \(\sigma : V \rightarrow T\Sigma\), we define \(\hat{\sigma} : \mathcal{CSG} \rightarrow \mathcal{CSG}\) according to the following rules:

1. \(\hat{\sigma}(0) := 0\) and \(\hat{\sigma}(1) := 1\).
2. \(\hat{\sigma}((\varphi \mapsto S_0 \mid S_1)) := (\sigma(\varphi) \mapsto \hat{\sigma}(S_0) \mid \hat{\sigma}(S_1))\)
3. \(\hat{\sigma}(((S)_\mu \mapsto S_0 \mid S_1)) := \begin{cases} \{\mathcal{H}(\hat{\sigma}(S), \hat{\sigma}(S_0), \hat{\sigma}(S_1))\} & : \text{if } \sigma(\mu) \neq \mu \\ \{((\hat{\sigma}(S))_\mu \mapsto \hat{\sigma}(S_0) \mid \hat{\sigma}(S_1))\} & : \text{if } \sigma(\mu) = \mu \end{cases}\)

In case of propositional logic, no further rules are necessary. In order to prove a formula \(\varphi\) to be valid, we just have to compute the negative initial graph \(\mathcal{H}_0(\varphi)\) of it and to check if it is closed under \(id\), i.e. if each path \(p \in \mathcal{PS}_1(\mathcal{H}_0(\varphi))\) contains a formula \(\varphi\) and its negation \(\neg \varphi\). For first order logic, however, this is not sufficient: There are a lot of formulae which are theorems, but whose initial graphs cannot be closed under any substitution. In order to achieve completeness for this logic, we have to define the following rule which is the only rule of \(\mathcal{CSG}\). This rule is based on the law \((\forall x.\varphi) \Leftrightarrow (\forall x.\varphi) \land (\forall y.[\varphi]_y)\) which allows to add a new variant \((S)_\mu [\varphi]_\mu\) of a quantified subgraph \((S)_\mu\) to the graph.

**Definition 2.6 (Extension rule)** The extension relation \(\vdash^e\) is defined as follows:

1. \((S \mapsto S_0 \mid S_1) \vdash^e (S \mapsto S'_0 \mid S'_1), \text{ if } S_0 \vdash^e S'_0\).
2. \((S \mapsto S_0 \mid S_1) \vdash^e (S \mapsto S_0 \mid S'_1), \text{ if } S_1 \vdash^e S'_1\).
3. \(((S)_\mu \mapsto S_0 \mid S_1) \vdash^e ((S')_\mu \mapsto S_0 \mid S_1), \text{ if } S \vdash^e S'.\)
4. \((S)_\mu \rightarrow S_0 \mid S_1\) \vdash \varepsilon \left( (S)_\mu \rightarrow S_0 \mid (\lfloor S \rfloor_\xi_\mu \rightarrow 0 \mid S_1) \right).

Rules 1, 2 and 3 are used to traverse the nodes of the graph so that rule 4 can be applied to generate a new variant. This variant is placed immediately after the original subgraph as in the definition given above and in figure 3.

Figure 3 illustrates the extension rule. The extension rule is now used to derive the formula which we want to prove:

**Definition 2.7 (Derivation in \(C_{\Sigma} \))** A formula \(\varphi\) is derivable in \(C_{\Sigma}\) iff there exists a finite list of shannon graphs \(S_0, \ldots, S_n\) such that \(\exists_0(\varphi) = S_0 \vdash \varepsilon \quad \ldots \quad \vdash \varepsilon \quad S_n \in C_{\Sigma} \).

A proof in \(C_{\Sigma}\) can now be obtained as follows: First the initial graph \(\exists_0(\varphi)\) of the formula which has to be proven has to be computed. Then a finite number of extension rules is applied until an element of \(C_{\Sigma} \) is obtained. Figure 4 shows a proof of \((\forall x. P x \land Q x) \rightarrow (P a \land Q b)\). First the initial graph \(\exists_0((\forall x. P x \land Q x) \rightarrow (P a \land Q b))\), given in the left side of figure 4, has to be extended to the graph given in the middle of figure 4. Then a substitution \(\sigma = [\mu \leftarrow a][\xi \leftarrow b]\) is found under which the resulting graph is closed. The graph on the right side shows the instance of the extension under \(\sigma\). Note that only the paths of \(\mathcal{P}S_1(S)\) have to be checked for closure.

Note that all given operations are only *syntactical* operations and the semantics of the shannon graphs is given below:

**Definition 2.8 (Semantics of First Order Shannon Graphs)** Given an interpretation \((D, I)\) and a variable assignment \(\xi\), the function \(\Omega^{D, I, \xi} : \Sigma_{\xi} \rightarrow \{\text{true, false}\}\) is defined as follows:

1. \(\Omega^{D, I, \xi}(0) := \text{false}\)
2. \(\Omega^{D, I, \xi}(1) := \text{true}\)
3. \(\Omega^{D, I, \xi}(\varphi \leftrightarrow B_0 \mid B_1) := \left\{ \begin{array}{ll} \Omega^{D, I, \xi}(B_1) : & \omega^{D, I, \xi}(\varphi) = \text{true} \\ \Omega^{D, I, \xi}(B_0) : & \omega^{D, I, \xi}(\varphi) = \text{false} \end{array} \right\} \)
Figure 4: A proof in $C_S$.

4. $\Omega^{D,\mu}((S)_\mu \mapsto B_0 \mid B_1) := \begin{cases} 
\Omega^{D,\mu}(B_1) : \text{true for all } d \in D \\
\Omega^{D,\mu}(B_0) : \text{otherwise}
\end{cases}$

Lemma 2.1 (Properties of First Order Shannon Graphs) Let $|\varphi|$ and $|S|$ denote the number of atomic formulae occurring in the formula $\varphi$ or in the Shannon graph $S$, respectively.\footnote{Due to lack of space we give no proofs of the listed theorems but the required proofs are quite easy. For completeness theorems, the theorem of Gödel, Herbrand and Skolem is sufficient, for the correctness, lemma 2.1 will provide the theoretical basis.}

1. $|\mathfrak{N}(3_k(\varphi), S_0, S_1)| = |3_k(\varphi)| + |S_0| + |S_1|$ (where $k \in \{0, 1\}$), thus we can conclude that $|3_k(\varphi)| = |\varphi|$, or even: the initial graphs contain the atoms which occur in the original formula where variables might have been substituted by skolem terms.
2. The computation of the initial graphs is correct, that is
   
   (a) \( \varphi \) is unsatisfiable iff \( \exists_1(\varphi) \) is unsatisfiable,

   (b) \( \varphi \) is valid iff \( \exists_0(\varphi) \) is unsatisfiable.

   If the formula \( \varphi \) is skolemized, we can even formulate the following properties:

   \[
   \omega^{D,1,\xi}(\varphi) = \Omega^{D,1,\xi}(\exists_1(\varphi)) \text{ and } \omega^{D,1,\xi}(\neg \varphi) = \Omega^{D,1,\xi}(\exists_0(\varphi))
   \]

3. The extension rule is a correct rule:

   \[
   \Omega^{D,1,\xi}(S) = \Omega^{D,1,\xi}(S') \text{ if } S \vdash^{\xi} S'
   \]

4. All closed Shannon graphs are unsatisfiable:

   \[ S \in CSG \Rightarrow \Omega^{D,1,\xi}(S) = \text{false for all } D, I, \xi \]

**Theorem 2.1 (Soundness and Completeness of \( C_{SG} \))** A formula \( \varphi \in \text{For}_{\Sigma} \) is valid iff there is a derivation for \( \varphi \) in \( C_{SG} \), i.e., there is a substitution \( \sigma \) and a finite number of extensions \( S_0, \ldots, S_n \) such that

1. \( \exists_0(\varphi) = S_0 \vdash^{\xi} \ldots \vdash^{\xi} S_n \)

2. \( S_n \) is closed under \( \sigma \), i.e., each path \( p \in \mathcal{P}S_1(S_n) \) contains at least one pair \( (\varphi_p, \neg \psi_p) \) such that \( \sigma(\varphi_p) = \sigma(\psi_p) \).

### 2.2 First Order BDD Calculi: \( \mathcal{C}_{BDD} \) and \( \mathcal{C}_{OBDD} \)

If two propositional formulae \( \varphi_1 \) and \( \varphi_2 \) are equivalent, one could also state that they represent the same boolean function. Unfortunately, this semantic equivalence does not correspond to the syntactical equivalence. Even if we restrict the syntax on the single Shannon operator \( (\cdot \leftarrow \cdot | .) \), no normal forms are available. However, if the propositional constants are ordered and the Shannon graphs are reduced, Bryant [Brya86] has shown that the resulting OBDD (ordered binary decision diagram) is unique for all equivalent propositional formulae. These OBDDs are often used to represent boolean functions or sets by their characteristic functions in a lot of implementations for different purposes.

In this section, we will define two further calculi \( \mathcal{C}_{BDD} \) and \( \mathcal{C}_{OBDD} \) which are both based on BDDs. Similar to \( C_{SG} \), \( C_{BDD} = (BDD_{\Sigma}, \{0\}, \{\vdash^{\xi}_\sigma, R\}) \) consists of a language \( BDD_{\Sigma} \) on which it operates, a set of axioms \( \{0\} \) which contains as the only axiom the Shannon graph \( 0 \) and the two rules \( \{\vdash^{\xi}_\sigma, R\} \). The extension relation \( \vdash^{\xi}_\sigma \) is defined as in \( C_{SG} \) except that a substitution \( \sigma \) is immediately applied to the added subgraph and a new rule \( R \) has to be applied for reducing the extended Shannon graphs to a First Order BDD. This rule eliminates redundancy in the graphs. The application of \( R \) to all graphs of \( S_{G_{\Sigma}} \) yields the set of first order binary decision diagrams \( BDD_{\Sigma} \), thus \( BDD_{\Sigma} = R(S_{G_{\Sigma}}) \). In the following, we will also define an ordered version of \( C_{BDD} \). This calculus denoted by \( \mathcal{C}_{OBDD} \) is more efficient to implement than \( \mathcal{C}_{BDD} \) because of its more restricted representation. To see this, consider the reduction rule of \( C_{BDD} \):
Definition 2.9 (Reduction rule) In order to define the reduction operator \( \mathcal{R} \) we have to introduce the operator \( \mathcal{R}_p^N(S) \), which is supplied with two further arguments \( P, N \) which represent the positive and negative literals on the considered path up to the root.

1. \( \mathcal{R}_p^N(0) := 0 \)
2. \( \mathcal{R}_p^N(1) := 1 \)
3. \( \mathcal{R}_p^N((\varphi \mapsto S_0 \mid S_1)) = \begin{cases} \mathcal{R}_p^N(S_0) & : \text{if } \varphi \in N \\ \mathcal{R}_p^N(S_1) & : \text{if } \varphi \in P \\ \mathcal{R}_p^{Nu(\neg \varphi)}(S_1) & : \text{if } \mathcal{R}_p^{Nu(\neg \varphi)}(S_1) = \mathcal{R}_p^{Nu(\neg \varphi)}(S_0) \\ \left( \varphi \mapsto \mathcal{R}_p^{Nu(\neg \varphi)}(S_1) \mid \mathcal{R}_p^{Nu(\neg \varphi)}(S_1) \right) & : \text{otherwise} \end{cases} \)
4. \( \mathcal{R}_p^N((S)_\mu \mapsto S_0 \mid S_1)) = \left( (\mathcal{R}_p^N(S))_\mu \mapsto \mathcal{R}_p^N(S_0) \mid \mathcal{R}_p^N(S_1) \right) \)

The reduce operator is now defined as \( \mathcal{R}(S) := \mathcal{R}_i^N(S) \).

A proof in \( \mathcal{C}_{\text{BDD}} \) will now proceed as follows: First the initial shannon graph \( \exists_0(\varphi) \) of a given formula \( \varphi \) has to be computed according to definition 2.2\(^3\). Then we compute a first order binary decision diagram \( \mathcal{R}(\exists_0(\varphi)) \) by application of the reduce operator. As long as the axiom \( \mathcal{A} \) is not obtained by the reduction, an extension rule is applied with a substitution \( \sigma \) and the result is reduced afterwards. \( \sigma \) can be computed via usual unification algorithms.

Figure 5 shows an example proof of \( (\forall x. Px \land Qx) \rightarrow (Pa \land Qb) \).

Theorem 2.2 (Soundness and Completeness of \( \mathcal{C}_{\text{BDD}} \)) A formula \( \varphi \in \text{For}_\Sigma \) is valid if and only if there is a derivation for \( \varphi \) in \( \mathcal{C}_{\text{BDD}} \), i.e. there are substitutions \( \sigma_0, \ldots, \sigma_n \) and a finite number of first order BDDs \( B_0, \ldots, B_{n+1} \) such that

- \( B_0 = \mathcal{R}(\exists_0(\varphi)) \)
- \( B_i \vdash_\Sigma S_{i+1} \) and \( B_{i+1} = \mathcal{R}(S_{i+1}) \) for \( i \in \{1, \ldots, n\} \)
- \( B_{n+1} = 0 \).

Theorem 2.3 (Relation of \( \mathcal{C}_{\Sigma \varphi} \) and \( \mathcal{C}_{\text{BDD}} \)) \( S \in \mathcal{S}_\Sigma \) is closed under \( \sigma \) if and only if \( \mathcal{R}(\dot{\sigma}(S)) = 0 \).

The syntactical test \( \mathcal{R}_{p,\dot{\mu}(\varphi)}^N(S_1) = \mathcal{R}_{p,\dot{\mu}(\neg \varphi)}^N(S_0) \) in item 3 of definition 2.9 could also be replaced by a semantical one, i.e. prove \( \mathcal{R}_{p,\dot{\mu}(\varphi)}^N(S_1) \leftrightarrow \mathcal{R}_{p,\dot{\mu}(\neg \varphi)}^N(S_0) \). In propositional logic this means the same, but as no normal form is available in first order logic, the syntactical test may not hold even if the graphs are equivalent. However, checking the semantical equivalence is in general quite expensive. This motivates us to define the calculus \( \mathcal{C}_{\text{OBDD}} \).

The definitions we give in this section for the calculus \( \mathcal{C}_{\text{BDD}} \) are not rigorous enough to obtain unique normal forms\(^4\), but the graphs are more restricted than those in \( \mathcal{C}_{\text{BDD}} \), i.e. the syntactical test \( \mathcal{R}_{p,\dot{\mu}(\varphi)}^N(S_1) = \mathcal{R}_{p,\dot{\mu}(\neg \varphi)}^N(S_0) \) holds more often for semantically equivalent graphs of \( \mathcal{C}_{\text{OBDD}} \) than in \( \mathcal{C}_{\text{BDD}} \).

The set \( \mathcal{OBDD}_\Sigma \) depends on a given order \( \prec \) on the subgraphs, e.g. choses the following:

\(^3\)Note that \( \varphi \) is a theorem if and only if \( \neg \varphi \) is unsatisfiable.

\(^4\)In general this is not possible in first-order logic, because then we could conclude the decidability of this logic, which can not be proven due to Church’s Theorem.
atomic formulae are ordered by the lexicographical order of their syntactical string representation

$(S)_\mu \prec (B)_\xi \iff \mu \prec \xi$ and $(S)_\mu \prec \varphi$ for all $\varphi \in \text{At}_\Sigma$

A lot of more sophisticated orderings have been defined for term rewriting systems which can also be used [Ders87].

**Definition 2.10 (Initial ordered graph of a formula)** Given any order $\prec$, we define the merge operator $\mathcal{M}_*(S_1, S_2)$ ($* \in \{\land, \lor\}$) as follows:

1. $\mathcal{M}_\land(0, S) := \mathcal{M}_\land(S, 0) := 0$ and $\mathcal{M}_\lor(0, S) := \mathcal{M}_\lor(S, 0) := S$.
2. $\mathcal{M}_\land(1, S) := \mathcal{M}_\land(S, 1) := S$ and $\mathcal{M}_\lor(1, S) := \mathcal{M}_\lor(S, 1) := 1$.
3. $\mathcal{M}_*(\alpha_1 \mapsto \beta_1 \mid \gamma_1), (\alpha_2 \mapsto \beta_2 \mid \gamma_2))$

\[
\begin{cases}
(\alpha_1 \mapsto \mathcal{M}_*(\beta_1, \beta_2) \mid \mathcal{M}_*(\gamma_1, \gamma_2)) & : \text{if } \alpha_1 = \alpha_2 \\
(\alpha_1 \mapsto \mathcal{M}_*(\beta_1, (\alpha_2 \mapsto \beta_2 \mid \gamma_2)) \mid \mathcal{M}_*(\gamma_1, (\alpha_2 \mapsto \beta_2 \mid \gamma_2))) & : \text{if } \alpha_1 \prec \alpha_2 \\
(\alpha_2 \mapsto \mathcal{M}_*((\alpha_1 \mapsto \beta_1 \mid \gamma_1), \beta_2) \mid \mathcal{M}_*((\alpha_1 \mapsto \beta_1 \mid \gamma_1), \gamma_2)) & : \text{if } \alpha_2 \prec \alpha_1
\end{cases}
\]

Now the initial graphs $\Theta_0(\varphi)$ and $\Theta_1(\varphi)$ are defined as follows:
\[ \Theta_1(P(\tau_1, \ldots, \tau_n)) := P(\tau_1, \ldots, \tau_n) \rightarrow 0 | 1 \quad \Theta_0(P(\tau_1, \ldots, \tau_n)) := P(\tau_1, \ldots, \tau_n) \rightarrow 1 | 0 \]

For propositional logic, we have: \( \varphi \) is valid iff \( \Theta_0(\varphi) = 0 \) or \( \varphi \) is valid iff \( \Theta_1(\varphi) = 1 \). For first order logic, we have again to introduce an extension rule:

**Definition 2.11 (Extension of ordered BDDs)** The extension relation \( \vdash^e_\sigma \) is defined as follows. Rules 1-3 are analogous to definition 2.6, rule 4 is replaced by

\[
((S)_{\mu} \rightarrow S_0 \mid S_1) \vdash^e_\sigma ((S)_{\mu} \rightarrow S_0 \mid \mathcal{M}_\lambda(\hat{\sigma}, (S), S_1)).
\]

This instance is conjunctively ‘merged’ with the true-subgraph. Again we can formulate a soundness and completeness theorem:

**Theorem 2.4 (Soundness and Completeness of \( \mathcal{C}_{\text{OBDD}} \))**

A formula \( \varphi \in \text{For}_S \) is valid iff there is a derivation for \( \varphi \) in \( \mathcal{C}_{\text{OBDD}} \), i.e. there are substitutions \( \sigma_0, \ldots, \sigma_n \) and a finite number of extensions \( S_0, \ldots, S_n \) such that

\[ \Theta_0(\varphi) = S_0 \vdash^e_{\sigma_0} \ldots \vdash^e_{\sigma_n} S_n = 0 \]

### 2.3 Proof Procedures based on \( \mathcal{C}_{SG} \) and \( \mathcal{C}_{OBDD} \)

The Shannon graph calculus is very closely related to the tableau graph calculus \( \mathcal{C}_{TG} \) given in [ScKK92b] and it can be similarly implemented (see also [ScKK92b]). Instead of giving all details of an implementation, we will refer instead to section 3 and section 4 of [ScKK92b] since most of the items listed there carry over without any changes to an implementation of \( \mathcal{C}_{SG} \). Thus a proof procedure based on \( \mathcal{C}_{SG} \) will work as follows: First the initial graph augmented with a link structure connecting complementary pairs \( e.g., \varphi, \neg \varphi \) has to be computed. Then the closure of the graph has to be checked. If a path \( p \in \mathcal{P}S_1(S) \) contains no complementary pair (and no stopping criteria [ScKK92b] holds), the path \( p \) is extended by a quantified subgraph on it. Then the link structure is updated and the closure has to be checked again.

Similar to \( \mathcal{C}_{TG} \), we also have the choice between a depth-first path traversal and a breadth-first path traversal. Another implementation decision lies in the management of variable bindings: we can choose between a backtracking substitution search algorithm and a non-backtracking one as outlined in detail in [ScKK92b].

The extension rule as presented places the variant \( ([S]_{\mu}^e)_{\mu'} \) of a quantified subgraph \( (S)_{\mu} \) immediately after the original graph \( (S)_{\mu} \). The number of paths has decreased since all paths leaving \( (S)_{\mu} \) positively have been extended by \( ([S]_{\mu}^e)_{\mu'} \) this way. However, some of these paths could have also been closed without this extension, thus implementing extensions this way is very inefficient. Instead, we could also extend just one path ending
in a 1-leaf by the variant. In other words, after the detection of a path which cannot be closed, a variant of a quantified subgraph on the path is added exclusively to this path. This makes the prover more efficient as the number of paths to be checked is kept smaller this way.

Implementations of \( \mathcal{C}_{3PP} \), however, work differently: The extension rule has to immediately instantiate the variant to be added in order to allow the reductions to take place. Then the merging operator is used to merge the resulting BDDs together as given in definition 2.11.

3 A Hierarchical Hardware Case Study

In this section we illustrate the power of the calculi in the domain of hardware verification, by using an example where the descriptions are hierarchical. The hierarchy is exploited in proving the equivalence between the implementations and specifications, thus curtailing the size of the proof goals. Additionally the specifications use certain abstract datatypes for representing natural numbers. These abstract datatypes are defined recursively and hence the use of first order calculi allows us to automatically solve inductive goals which occur in parametrizable circuits.

The hardware environment called MEPHISTO has been implemented in the Standard ML version of the higher-order theorem prover called HOL90 [Gord88]. MEPHISTO contains an interface for converting netlists automatically into logical formulas, performs an expansion of the abbreviations used for the modules, simplifies the resultant formula by removing all possible internal lines and finally calls the first-order prover based on \( \mathcal{C}_{3G} \). The details of MEPHISTO can be found in [KuSK93].

The example used is called a ‘delta’ circuit whose informal specification can be stated as \( c < a + b \). This informal specification is then converted automatically into a formal specification (as shown below)\(^5\) which uses predefined and prevalidated \( n \)-bit operators as described in [ScKK92c].

```plaintext
val DELTA_N_SPEC = 
  |~ !n a b c out.
  DELTA_N_SPEC n a b c out =
    (?l1. PLUS n a b l1 F /
      (out = LS n c l1 /
       ~(PCARRY n a b F))) :thm
```

The informal specification is converted into a 3-address format with an internal variable \( 11 \). The predicate \( PLUS \) is true if \( 11 \) is the sum of the bitvectors \( a, b \) and the carry-bit, which is set to ‘false’. The predicate \( PCARRY \) corresponds to the overflow bit and \( LS \) to the less than operator.

The delta circuit can be implemented by using an \( n \)-bit adder, \( n \)-bit comparator and some gates. The design and verification of an \( n \)-bit ‘less than’ circuit is first illustrated, before the delta circuit is verified. The specification of this circuit uses the predefined operator \( LS \).

\(^5\)The numbered boxes correspond to actual sessions in HOL90. The symbols ‘!’, ‘?’ and ‘~’, correspond to forall, exists and negation operations, respectively. It is to be noted that, only the essential steps needed for clarifying the example have been listed in the boxes.
In implementing \( n \)-bit components, a recursive implementation scheme is given which is built up of an 1-bit component and an \( n \)-bit component to give an \( n+1 \)-bit component. Figure 6 shows a schematic for the less than component and the corresponding formal description is given in box 3.

This \( n \)-bit implementation ‘LESS_N_IMP’ can then be verified against its specification ‘LESS_N_SPEC’.

Having set the goal to be verified, the various steps in automatically proving the goal are illustrated. Applying the induction tactic within HOL90, yields two subgoals as shown below:
The base case can be solved by first expanding the definitions of the specification and the implementations, i.e. LESS_N_IMP, LESS_I_IMP and the library components INV, AND and OR2, and then finally rewriting the internal lines away by their definitions [KuSK93].

```
- e (PURE_ASM_REWRITE_TAC defs THEN SIMPLIFY_TAC);

Goal proved.
|- !a b lessOut.
  (?11 12 13 14 15.
   (11 = ~a 0) /
   (12 = 11 /\ b 0) /
   (13 = 11 /\ F) /
   (15 = F /\ b 0) /
   (14 = 12 \ 13) /
   (lessOut = 14 \ 15)) =
   lessOut =
   ~(a 0) /\ b 0

Goal proved.
|- !a b lessOut. LESS_N_IMP 0 a b lessOut = LESS_N_SPEC 0 a b lessOut

Remaining subgoals:
!a b lessOut.
LESS_N_IMP (SUC n) a b lessOut = LESS_N_SPEC (SUC n) a b lessOut
  (-->'!a b lessOut.
   LESS_N_IMP n a b lessOut = LESS_N_SPEC n a b lessOut'--)
```

By proceeding in a manner similar to the base case, the subgoal corresponding to the step case can also be simplified as shown below. It is to be noted that in general the definitions corresponding to the predefined operators on bitvectors have also to be used after using the inductive assumption, i.e. “LESS_N_IMP n a b lessOut = LESS_N_SPEC n a b lessOut”.

```
- e (PURE_ASM_REWRITE_TAC defs THEN SIMPLIFY_TAC);

OK..
1 subgoal:
(lessOut =
 ((a (SUC n)) /\ b (SUC n) \/ ~(a (SUC n)) /\ LS n a b) /\
 LS n a b /\ b (SUC n)) =
lessOut =
 ((a (SUC n) = b (SUC n)) ==> LS n a b) /\
 ~(a (SUC n) = b (SUC n)) ==> ~(a (SUC n)) /\ b (SUC n))
  (-->'!a b lessOut.
   LESS_N_IMP n a b lessOut = LESS_N_SPEC n a b lessOut'--)
```

The simplified subgoal is then directly fed to the first-order prover based on CSG, which solves the goal in less than 10 milliseconds. This entire process can be performed by using a single tactic called ‘BDD_REC_COMP_TAC’, and stored within the theory as the theorem ‘LESS_N_CORRECT’ for future use.
- e(PRED_BDD_TAC);

OK..

Goal proved.
|\- (lessOut =
  (~a (SUC n)) \ b (SUC n) \ (~a (SUC n)) \ LS n a b) \/
  LS n a b \ b (SUC n)) =
lessOut =
  ((a (SUC n) = b (SUC n)) => LS n a b) \
  (~a (SUC n) = b (SUC n)) => (~a (SUC n)) \ b (SUC n))

Goal proved.
|\- !n a b lessOut.
  LESS_N_IMP n a b lessOut = LESS_N_SPEC n a b lessOut

Top goal proved.
val it = () :unit

- save_thm ("LESS_N_CORRECT", TAC_PROOF (g, BDD_REC_COMP_TAC defs));
val LESS_N_CORRECT =
|\- !n a b lessOut.
  LESS_N_IMP n a b lessOut = LESS_N_SPEC n a b lessOut

- closetheory();
val it = () :unit
- export_theory();
val it = () :unit

The process similar to the less than circuit is then repeated for the n—bit adder circuit and stored within the theory ‘adder’. It is to be noted that the same tactic called ‘BDD_REC_COMP_TAC’ is once again used to solve the goal, that the recursive n—bit adder implementation corresponds to the specification using the predefined operators. We omit its details due to lack of space.

The theories ‘less’ and ‘adder’ which have been created can then be used to describe the implementation of the delta circuit and verify it against its specification. The automatically extracted formal description of the implementation is given below:

val DELTA_N_IMP =
|\- !n a b c out.
  DELTA_N_IMP n a b c out =
  (?11 12 13 14.
   ADDER_N_IMP n a b 11 12 \/
   LESS_N_IMP n c 11 13 \/
   INV (12,14) \/
   AND (13,14,out)) :thm

Such goals can always be automatically solved by the same sequence of tactics as shown below and can be put together into a single tactic called ‘BDD_SIMP_COMP_TAC’.
val goal =
(\[],
(\(\text{\textbf{!}}\text{n a b c out}.
\text{DELTA\_N\_IMP } \text{n a b c out } = \text{DELTA\_N\_SPEC n a b c out}\)\)\)
):\text{a list } * \text{ term}
- \text{save_thm}("\text{DELTA\_N\_CORRECT},
= \text{TAC\_PROOF (goal, (\text{REPEAT GEN\_TAC
= \text{THEN REWRITE\_TAC def
= \text{THEN SIMPLIFY\_TAC
= \text{THEN PRED\_BDD\_TAC))});
val \text{DELTA\_N\_CORRECT} =
\text{\textbf{!}}\text{n a b c out. } \text{DELTA\_N\_IMP } \text{n a b c out } = \text{DELTA\_N\_SPEC n a b c out}
:\text{thm}

The correctness theorem and the definitions of the specification and the implementation can then be used to automatically generate a specialized implementation corresponding to a specified bit width. This implementation can then be converted into netlists and the circuit designer can use a verified component in his design.

To summarize this section, we can see that the power of the first-order BDD prover can be exploited to more or less automatically verify hierarchical circuits whose specifications use predefined operators which reflect the semantics of natural numbers. In the next section some more experimental results are given to illustrate the speed of the BDD prover as against the older implementation, FAUST [ScKK92b], which was based on a modified tableau calculus.

4 Experimental Results

At present we have just implemented a preliminary version of $C_{\text{SG}}$, where the unifier search algorithm is replaced by a 'guessing procedure' which tries out a term that seems to be the likely one for obtaining a closed graph. We therefore list the results of this preliminary version, which are nevertheless quite promising.

The implementation of $C_{\text{SG}}$ has been embedded within the hardware verification environment called MEPHISTO ([KuSK93]) which performs major simplifications before the prover is called. The power of the prover is illustrated by an example shown in the HOL90 boxed session 11. This higher-order formula which occurs in the proof of the goal while proving the correctness of the implementation of a sampler circuit against its specification is solved by the prover in a matter of 130 milliseconds. Converting this formula into a propositional one after deriving the appropriate instantiations for the quantified variable $t$, would result in a formula containing 22 variables instead of the six variables — $\text{in1}$, $\text{sel}$, $\text{out0}$, $\text{out1}$, $\text{out2}$, $\text{out3}$.

The results listed in table 1 correspond to the proofs of specifications against implementations and the column goal-type indicates if an equivalence or an implication has been proved. The column $C_{\text{SG}}$, corresponds to the overall time taken for constructing the BDD and the actual proof. These times are much better than the time taken by FAUST as shown in the column $C_{\text{FG}}$.

\footnote{The goal is of an implication type if the specification is partial.}
5 Summary

We have shown, how shannon graphs and the related binary decision diagrams can be lifted to first order logic and how proof calculi based on these graphs can be defined. We have elaborated on an example and have shown how these calculi can be exploited in a hardware-verification environment for automatically proving hierarchical designs which use abstract datatypes in their specifications. The preliminary experimental results using such an approach has also been indicated.

We are currently implementing a prover based on $C_{OBD}$ and are testing how the heuristics given for propositional BDDs carry over to them. Additionally, we will compare the implementations of $C_{SG}$ and $C_{OBD}$ as far as their efficiency is concerned.

```
val t =
  (|--'(!t.
    (out_3 0 = ~(sel 0) \~(sel 0)) /
    (out_2 0 = ~(sel 0) \~(sel 0)) /
    (out_2 1 = ~(sel 1) \~(sel 1)) /
    (out_1 0 = ~(sel 0) \~(sel 0)) /
    (out_1 1 = ~(sel 1) \~(sel 1)) /
    (out_1 2 = ~(sel 2) \~(sel 2)) /
    (out_0 0 = ~(sel 0) \~(sel 0)) /
    (out_0 1 = ~(sel 1) \~(sel 1)) /
    (out_0 2 = ~(sel 2) \~(sel 2)) /
    (out_03) =
    ~(sel 3) \~(sel 3)) /
    (out_3(t+1)) =
    (sel (t+1) ==> in1 t) \~(sel (t+1)) /
    (out_2(t+2)) =
    (sel (t+2) ==> in1 t) \~(sel (t+2)) /
    (out_1(t+3)) =
    (sel (t+3) ==> in1 t) \~(sel (t+3)) /
    (out_0(t+4)) =
    (sel (t+4) ==> in1 t) \~(sel (t+4))) ==>
    (!t.
    (out_0 (t+4) =
    (sel (t+4) ==> in1 t) \~(sel (t+4))) /
    (out_1 (t+3) =
    (sel (t+3) ==> in1 t) \~(sel (t+3))) /
    (out_2 (t+2) =
    (sel (t+2) ==> in1 t) \~(sel (t+2))) /
    (out_3 (t+1) =
    (sel (t+1) ==> in1 t) \~(sel (t+1)))'-- :term
- predprove_time t;
```

Translating Term : 0.030000
Constructing BDD : 0.010000
Closing BDD : 0.120000
Proof Time : 0.130000
<table>
<thead>
<tr>
<th>circuit</th>
<th>goal-type</th>
<th>$C_{SC}$ [msec]</th>
<th>$C_{TG}$ [msec]</th>
<th>circuit</th>
<th>goal-type</th>
<th>$C_{SC}$ [msec]</th>
<th>$C_{TG}$ [msec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD2</td>
<td>$\leftarrow$</td>
<td>90</td>
<td>250</td>
<td>BCD_CORRECT</td>
<td>$\rightarrow$</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>C_COUNT</td>
<td>$\rightarrow$</td>
<td>40</td>
<td>70</td>
<td>DETECT111</td>
<td>$\rightarrow$</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>DETECT110</td>
<td>$\rightarrow$</td>
<td>90</td>
<td>580</td>
<td>DMUX</td>
<td>$\rightarrow$</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>JK1</td>
<td>$\leftarrow$</td>
<td>10</td>
<td>30</td>
<td>JK2</td>
<td>$\leftarrow$</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>MUX</td>
<td>$\leftarrow$</td>
<td>10</td>
<td>30</td>
<td>SPARITY</td>
<td>$\leftarrow$</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>PAR_SER</td>
<td>$\rightarrow$</td>
<td>70</td>
<td>670</td>
<td>RESET_REG</td>
<td>$\leftarrow$</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>SADDER</td>
<td>$\leftarrow$</td>
<td>30</td>
<td>250</td>
<td>SAMPLER</td>
<td>$\rightarrow$</td>
<td>130</td>
<td>2020</td>
</tr>
<tr>
<td>SAMPLER1</td>
<td>$\rightarrow$</td>
<td>140</td>
<td>2010</td>
<td>SREG4</td>
<td>$\rightarrow$</td>
<td>60</td>
<td>160</td>
</tr>
<tr>
<td>TRC_CORRECT</td>
<td>$\leftarrow$</td>
<td>60</td>
<td>140</td>
<td>TRC_011</td>
<td>$\leftarrow$</td>
<td>40</td>
<td>70</td>
</tr>
<tr>
<td>TRC_100</td>
<td>$\leftarrow$</td>
<td>40</td>
<td>70</td>
<td>XOR</td>
<td>$\rightarrow$</td>
<td>10</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 1: Verification times of various circuits

References


