Abstract

We propose a novel technique to improve SAT-based Combinational Equivalence Checking (CEC) by statically adding meaningful clauses to the CNF formula of the miter circuit. A fast preprocessing quickly builds up the implication graph for the miter circuit under verification, resulting in a large set of direct, indirect and extended backward implications. The non-trivial implications are converted into two-literal clauses and added to the miter CNF database. These added clauses constrain the search space, and provide correlation among the different variables, which enhances the Boolean Constraint Propagation (BCP). Experimental results on ISCAS'85 CEC instances show that with the added clauses, an average speedup of more than 950x was achieved.

I. Introduction

In the past four decades, much progress has been made in the field of Boolean Satisfiability (SAT). Due to its numerous Electronic Design Automation (EDA) applications, such as Combinational Equivalence Checking (CEC) [5, 7, 14, 15], Bounded Model Checking (BMC) [8, 13] and Automatic Test Pattern Generation (ATPG) [11, 12], SAT continues to be a heavily studied area. The state-of-the-art SAT solvers [1-4] are usually based on the Conjunctive Normal Form (CNF). While trying to satisfy the given CNF, the SAT solver makes decisions based on a given set of variable selection heuristics [1-4]. It learns dynamically from the conflicts encountered during the search, and generates conflict-induced clauses [1, 2], that can subsequently constrain the search. However, the conflict clauses learnt dynamically have the following disadvantages:

- Not all learned clauses are useful, especially long clauses.
- Set of all learned clauses can grow very large.
- The clauses are learned gradually over the entire SAT search, which may take a long time.

Recently, efforts have been made to improve the SAT-based CEC by inducing useful information into the original CNF before the SAT solver starts. This overcomes the above disadvantages to some extent. In [5, 6], probable correlation among signal pairs is first obtained via random simulation on the miter circuit. Then, explicit learning is performed wherein the correlated signal pairs are assigned values, which most likely results in conflict. A SAT-solver is invoked to quickly learn a fixed number of conflict-induced clauses, corresponding to every correlated pair. Because of random simulation, only a subset of signal correlations is identified. In [7], equivalence and implication relations are learned within a set of observable variable list. This list of observable variables restricts the neighborhood over which related signals can be identified. In [8], which concentrates on improving SAT-based BMC, BDDs are built locally around the “seed” node, which can be selected statically or dynamically. The resulting BDD relations are converted into clauses, and added to the original CNF formula. However, the locally built BDDs are not helpful in extracting the global relations.

In our approach, unlike [5-8] we statically and efficiently identify useful non-trivial relations among signals (variables) over the entire miter circuit. We then augment the existing CNF formula by adding these relations as clauses, before the SAT solver starts. The preprocessing step quickly builds up the Implication Graph for the miter-circuit under verification. The resulting indirect and extended backward implications are converted into two-literal clauses, which are added to the CNF database. These added clauses prune the search space and provide correlation among different variables, which enhances the Boolean Constraint Propagation (BCP). Experimental results for combinational circuit equivalence checking showed that using the proposed method, an average speedup of more than 950x was achieved.

The rest of the paper is organized as follows. Section II gives the background of Static Implications that we have used in our implementation. Section III focuses on the interesting observations, when indirect and extended backward implications are applied to the problem at hand. Here we also show how to convert these implications into two-literal clause form. Section IV gives the implementation algorithm. Experimental results are discussed in Section V, and finally section VI concludes the paper with an insight into future work.

II. Preliminaries

Static Implications

Static implications are obtained by asserting the logic values ‘0’ and ‘1’, respectively, to every gate in the circuit. They are made up of direct, indirect and extended backward
implications. Direct implications can be easily determined whereas indirect and extended backward implications are non-trivial, and their discoveries require combination of simulation, transitive law and contrapositive law [9, 10]. An efficient way for representing the implication relations is through an implication graph. For a given circuit with N gates, the total number of nodes in this graph is \(N \times 2\), since each gate can take on a logic value of 0 or 1. The following terminology is used:

- \((N, v)\): Assign logic value v to gate N.
- \((N, v) \rightarrow (M, w)\): Assigning logic value v to gate N implies gate M would be assigned a value w.
- \(\text{impl}[N, v]\): Set of all implications resulting from assigning logic value v to gate N.

- Contrapositive law: If \((N, v) \rightarrow (M, w)\), then the contrapositive law states that \((M, w') \rightarrow (N, v')\), where \(w'\) and \(v'\) are the complementary values of \(w\) and \(v\), respectively. This property can be used to identify additional (possibly non-trivial) implications.
- Impossible/constant nodes: If \((M, w) \rightarrow (N, v)\) and \((M, w) \rightarrow (N, v')\) or if \((M, w) \rightarrow (M, w')\), then \((M, w)\) is impossible, i.e. gate M would never be able to acquire value w and would be a constant with value \(w'\).

The concepts of direct, indirect and extended backward implications can be illustrated through the example circuit shown in Figure 1.

![Figure 1: Example Circuit](image)

**Direct Implications:** Consider gate ‘f’ in Figure 1. When we assert a logic ‘0’ value on its output, the direct forward implications are \((g, 1)\) and \((h, 1)\). Similarly, the direct backward implications are \((e, 1)\) and \((c, 1)\). Therefore, \(\text{impl}[f, 0] = \{(f, 0), (g, 1), (h, 1), (e, 1), (c, 1)\}\).

**Indirect implications:** Consider the direct implications of \((f, 0)\) in Figure 1. We see that \((g, 1)\) or \((h, 1)\) individually, do not imply anything on gate ‘i’’. However, together they imply \((i, 1)\). Therefore \((f, 0) \rightarrow (i, 1)\) is an indirect implication, and can be computed by simply logic simulating the list \(\text{impl} \ [f, 0]\). These indirect implications are added to the implication graph of the circuit along with their corresponding contrapositive implications. Thus, \(\text{impl}[f, 0] = \{(f, 0), (g, 1), (h, 1), (e, 1), (c, 1), (i, 1)\}\).

**Extended Backward implications:** Extended Backward implications [9] are performed on unjustified gates in the implication list of the target gate. A target gate is one whose implications we are determining. Let \((G, v) \in \text{impl}[N, v]\), and suppose gate G has \(m\) inputs \((l_0, \ldots, l_m)\).

Here \(N\) is the target gate and \(G\) is the unjustified gate. Then, assuming \(G\) is an OR gate,

\[
\text{impl}[N, v] = \bigcup_{\forall i} \text{LogicSimulate(impl}[N, v] \cup \text{impl}[l_i, 0)]
\]

Here LogicSimulate() refers to performing logic simulation with the implications plugged onto the circuit.

Assuming \(G\) is an \(OR\) gate

\[
\text{impl}[N, v] = \bigcup_{\forall i} \text{LogicSimulate(impl}[N, v] \cup \text{impl}[l_i, 1])
\]

In the same way, extended backward implications can be computed for \(NAND\) and \(NOR\) gates.

Assuming \(G\) is a 2-input \(XOR\) gate,

\[
\text{impl}[N, v] = \bigcup_{\forall i, j} \{ \text{LogicSimulate(impl}[l_i, 0) \cup \text{impl}[l_j, 1]) \cap \text{LogicSimulate(impl}[l_i, 1) \cup \text{impl}[l_j, 0]) \}
\]

Similarly for \(XNOR\) gate.

It should be mentioned that since we are dealing with miter circuits, the extended backward implications pertaining to XOR/XNOR gates helps to identify many meaningful non-trivial implications, which in turn plays an important role in proving the equivalence of two circuits.

To illustrate the concept of extended backward implications, consider the example circuit of Figure 1. We see that \(\text{impl}[f, 0] = \{(f, 0), (g, 1), (h, 1), (e, 1), (c, 1), (i, 1)\}\). The implication list of \((f, 0)\) contains \((e, 1)\) and the \(OR\) gate ‘e’ is unjustified. Now justifying \(e=1\) by setting the fanin \(a=1\) yields \(XOR\) gate \(j=0\) and \(j=0 \rightarrow m=0\). Setting the fanin \(b=1\) results in \(NAND\) gate \(k=0\) and \(k=0 \rightarrow m=0\). Thus, if the \(OR\) gate ‘e’ is justified by any of the fanins, we get a common implication \(m=0\). Therefore, \(f=0 \rightarrow m=0\) is an extended backward implication of \((f, 0)\), and is appended to the list \(\text{impl} \ [f, 0]\). Intuitively, extended backward implications help to identify the hard-to-find implications, and hence are effective for various applications such as capturing additional untestable faults [9, 10].

It must be mentioned here that the concept of extended backward implications [9] is different from Recursive Learning [16, 17] of level (depth) ‘1’. The relations obtained through extended backward implications may take quite a few levels of recursion by the Recursive Learning
procedure. In fact, recursive learning of depth ‘1’ is equivalent to performing direct backward implications on each the fanins of the unjustified gate, and determining the common set of implications. This is done just once for all the unjustified gates. On the other hand, extended backward implications makes use of the implication list of the target gate (this implication list includes the unjustified gate and its corresponding implications), in addition to the implication list of the fanins of the unjustified gate and performs logic simulation to determine the common set of assignments. Note that the same unjustified gate may be processed more than once. Also, with Recursive Learning as the depth of recursion is increased, the time to compute the implications increases exponentially. Extended backward implications on the other hand, help to quickly identify the useful non-trivial implications; the computation complexity is linear to the number of unjustified gates in the implication list of the target gate.

III. Application of Static Implications to SAT

When a circuit is converted into its equivalent CNF-form, there is loss of circuit structural information. In our approach, we first compute the implications on the circuit netlist, and convert the meaningful implications into clause form. These clauses when added to the original CNF formula, induce signal correlation among the variables, which in turn enhances the SAT solver performance.

Consider the CNF formula for circuit of Figure 1.

\[ \neg a + e \neg b + e \neg e + a + b (f + e) f + c \]
\[ \neg f + e + c \neg f + g f + g \neg f + d + h f + h \]
\[ d + h g + i h + i \neg g + h + i j + a + i \]
\[ \neg j + a + i j + a + i h + k b + k \]
\[ \neg h + b + k f + m k + m m + j + k \]

When we assign \( i=0 \), and perform Boolean Constraint Propagation (BCP) [1-4], no unit clauses are obtained. However, from our implication engine, we know that \( f=0 \rightarrow i=1 \), and by contrapositive law \( i=0 \rightarrow f=1 \). The two-literal clause corresponding to this implication is \( (i + f) \). If we add this clause beforehand to the original CNF, setting \( i=0 \) will imply \( f=1 \) immediately, which in turn will imply \( g=0 \). Therefore, learning the information \( i=0 \rightarrow f=1 \), helps us to satisfy a total of 10 clauses instead of 4. This is illustrated in Figure 2 below.

![Figure 2. Implied values and satisfied clauses in the CNF formula, before and after adding the clause \((i + f)\).](image)

Identification of Equivalent/Complement Literals

The basis of Combinational Equivalence Checking (CEC) is to identify equivalent signals in the two circuits incrementally, proceeding from the primary inputs towards the primary outputs. In SAT-based CEC, identification of such equivalent signals helps to reduce the problem complexity; a decision on one of the signals in the equivalent pair implies a value on the other corresponding signal, which in turn enhances the BCP and reduces the number of decisions required to prove the satisfiability/unsatisfiability of the CNF formula. The implication graph that we build (as a preprocessing step) for the miter circuit under verification helps us to identify these equivalent signals, which are in turn added as two-literal clauses to the existing CNF database.

Consider the CNF for the circuit shown in Figure 3.

\[ (x + p)(y + p)(\neg x + \neg y + \neg p)(x + q)(y + q)(\neg x + \neg y + \neg q) \]

![Figure 3. Equivalent/Complement literal identification](image)

We see that the decision \( p=0 \) implies \( x=1 \), \( y=1 \), and finally \( q=0 \). Similarly, the decision \( q=0 \) implies \( x=1 \), \( y=1 \), and finally \( p=0 \). But neither \( p=1 \) implies anything on \( q \), nor \( q=1 \) implies anything on \( p \). Hence, we cannot deduce that the two signals \( p \) and \( q \) are equivalent. However our Implication Engine can deduce this relation. We see that impl \( [p, 0] = \{(p,0), (x,1), (y,1), (q,0)\} \), where \( (p, 0) \rightarrow (q, 0) \) is an indirect implication. By contrapositive law, \( (q, 1) \rightarrow (p, 1) \). Similarly, impl\( [q, 0] = \{(q,0), (x,1), (y,1), (p,0)\} \), such that \( (q, 0) \rightarrow (p, 0) \) is an indirect implication. Again, using the contrapositive law, \( (p, 1) \rightarrow (q, 1) \). Thus, \( p \leftrightarrow q \). Therefore, for the two indirect implications, \( (p, 0) \rightarrow (q, 0) \) and \( (q, 0) \rightarrow (p, 0) \), we add up the clauses \( (p + \neg q) \) and \( (q + \neg p) \), respectively. It should be noted that every two-literal clause we add, embeds in itself both the indirect implication as well as its contrapositive. Thus, addition of such two clauses proves the equivalence of two literals. In the similar manner, our approach can also identify complementary signals in the circuit. These relations between intermediate points of the circuit are propagated in the forward direction and help to identify additional relations and implications throughout the circuit. Also, this technique does not suffer from the false negative problem [18, 19], since the variables are tied through clauses and none of the variables are eliminated.

Identification of Constant/Impossible Nodes

In order to prove the equivalence of two circuits, the corresponding PO’s of the two circuits are XORed (i.e., a miter circuit is created), and the XOR outputs are then...
checked for tautology ‘0’. In our approach, building the implication graph for the miter circuit under verification deduces a few XOR outputs to be at constant ‘0’ value. This happens whenever implications of the following type are obtained:

- \((\text{XOR
gate, 1}) \rightarrow (\text{Any
gate, 0})\) and \((\text{XOR
gate, 1}) \rightarrow (\text{Any
gate, 1})\) or
- \((\text{XOR
gate, 1}) \rightarrow (\text{XOR
gate, 0})\).

From the implications of the above type, one can observe that \((\text{XOR
gate, 1})\) is impossible, i.e. \((\text{XOR
gate, 1})\) would never be able to acquire logic value ‘1’ and would always be a constant with logic value ‘0’. After the implication graph for the miter circuit under verification has been built, all the nodes identified as constants are added as unit literals to the original CNF database. This in turn prunes the search space of the SAT-solver engine, thereby enhancing its performance.

**Significance of Extended Backward Implications**

The concept of extended backward implications helps us to learn some very useful, non-trivial two-node implications. When added as two-literal clauses to the original CNF, they play a significant role. Let us show this by means of example circuit in Figure 1. Suppose our objective is to satisfy \(m=1\). Assume that we make the decisions \(m=1\) (given objective), followed by \(f=0\), and then \(a=0\). But, assigning \(a=0\) results in a conflict. Also, on backtracking \(a=1\) yields a conflict. We again backtrack and set \(f=1\), and finally the decisions \(d=0\), \(b=0\) make the formula satisfiable. The decision tree obtained is shown in Figure 4.

**IV. Implementation Algorithm**

The flow of our implementation is described by the algorithm below.

**Algorithm:**

1. Generate the CNF formula for the miter circuit under verification.
2. Initialize the implication graph by computing the direct implications for all the nodes in the circuit.
3. Compute the indirect implications for each of the nodes in a levelized fashion, from the primary inputs towards the primary outputs.
4. Convert the indirect implications obtained in step 3 above, into two-literal clauses, and append them to the CNF database. Also, add the nodes determined as constants as unit clauses.
5. If 25% or more of the mitered XOR outputs have been identified as constant ‘0’s, give the new CNF database to the SAT-solver for processing and stop, else go to step 6.
6. For each gate N,
   - Perform extended backward implications on all the unjustified gates in the implication list of the target gate.
7. Convert the extended backward implications obtained in step 6 above, into two-literal clauses, and append them to existing CNF formula. Also, add the new constants as unit clauses.
8. Give the modified CNF formula to the SAT-solver for processing.
9. Stop.
V. Experimental Results

The proposed concept was implemented in C++ in a preprocessing engine called IMP2C (Implications to Clauses). IMP2C builds the Implication Graph for the miter circuit under verification, and formulates the two-liter clauses corresponding to indirect and extended backward implications learnt. The experiments were run on a Pentium-4, 1.8GHz machine, with 512Mb of RAM and Linux as the operating system. ZChaff[1] is used as the SAT-solver for all instances. The results are reported in Tables 1 and 2. Two different types of miter circuits are verified for the terms of ZChaff + IMP2C preprocessing engine execution time taken by In Table 1, for each miter circuit, we first report the satisfy OR gate output to logic 1. OR all the mitered outputs, and ask the SAT solver to satisfy OR gate output to logic 1.

In Table 1, for each miter circuit, we first report the execution time taken by ZChaff alone, followed by our preprocessing engine IMP2C, and finally the time taken by ZChaff + IMP2C together. We also report the speedup ratio of ZChaff + IMP2C over ZChaff alone. In the third column, the terms ‘I’ and ‘EB’ in the parenthesis refer to indirect implications and extended backward implications, respectively. (I) indicates that only indirect implications were used to augment the original CNF formula, whereas (I+EB) indicates that both indirect and extended backward implications were used.

Table 1. Results with ZChaff alone and (ZChaff + IMP2C)

<table>
<thead>
<tr>
<th>Miter Circuit</th>
<th>ZChaff (secs)</th>
<th>IMP2C (secs)</th>
<th>ZChaff + IMP2C (secs)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>c499_equiv</td>
<td>6.70</td>
<td>0.22(I+EB)</td>
<td>0.22</td>
<td>30.77</td>
</tr>
<tr>
<td>c880_equiv</td>
<td>1.26</td>
<td>0.03(I)</td>
<td>0.03</td>
<td>42.00</td>
</tr>
<tr>
<td>c1355_equiv</td>
<td>0.88</td>
<td>0.13(I)</td>
<td>0.13</td>
<td>6.76</td>
</tr>
<tr>
<td>c1908_equiv</td>
<td>12.36</td>
<td>0.07(I)</td>
<td>0.07</td>
<td>176.57</td>
</tr>
<tr>
<td>c2670_equiv</td>
<td>2.88</td>
<td>0.31(I)</td>
<td>0.32</td>
<td>9.00</td>
</tr>
<tr>
<td>c3540_equiv</td>
<td>85.29</td>
<td>0.94(I)</td>
<td>1.10</td>
<td>77.53</td>
</tr>
<tr>
<td>c5315_equiv</td>
<td>37.01</td>
<td>0.68(I)</td>
<td>0.72</td>
<td>51.40</td>
</tr>
<tr>
<td>c6288_equiv</td>
<td>&gt;5000</td>
<td>0.35(I)</td>
<td>0.36</td>
<td>13889</td>
</tr>
<tr>
<td>c7552_equiv</td>
<td>105.77</td>
<td>1.71(I)</td>
<td>1.80</td>
<td>58.76</td>
</tr>
<tr>
<td>c880_opt</td>
<td>1.01</td>
<td>0.31(I+EB)</td>
<td>0.33</td>
<td>3.06</td>
</tr>
<tr>
<td>c1355_opt</td>
<td>10.04</td>
<td>0.75(I+EB)</td>
<td>0.78</td>
<td>12.87</td>
</tr>
<tr>
<td>C1908_opt</td>
<td>18.53</td>
<td>3.26(I+EB)</td>
<td>3.26</td>
<td>5.68</td>
</tr>
<tr>
<td>c3540_opt</td>
<td>62.74</td>
<td>0.82(I)</td>
<td>2.82</td>
<td>22.24</td>
</tr>
<tr>
<td>c5315_opt</td>
<td>28.8</td>
<td>16.24(I+EB)</td>
<td>16.24</td>
<td>1.77</td>
</tr>
<tr>
<td>c6288_opt</td>
<td>&gt;5000</td>
<td>3.88(I+EB)</td>
<td>3.89</td>
<td>1285.3</td>
</tr>
<tr>
<td>c7552_opt</td>
<td>212.76</td>
<td>38.47(I+EB)</td>
<td>38.48</td>
<td>5.52</td>
</tr>
</tbody>
</table>

Average speed up being 979.88x over all the 16 instances.

According to Table 1, the execution time for constructing the implication graph is very low, ranging from 0.03 seconds to 38.47 seconds. In some cases, once the implication relations are computed, the SAT-solver ZChaff can determine the formula to be unsatisfiable almost immediately. For instance, in the miter circuits c7552_equiv and c7552_opt, without any added clauses, ZChaff spent 105.77 seconds and 212.76 seconds, respectively. When we augment the CNF with the computed implications (1.71 seconds and 38.47 seconds, respectively), the SAT problem complexity is reduced significantly, with ZChaff taking only a fraction of a second.

Overall, the results for ZChaff + IMP2C are very encouraging, with speed-ups ranging from 1.77x for c5315_opt to 13889x for c6288_equiv. In fact, for c6288_equiv and c6288_opt, ZChaff alone could not finish even after 5000sec. An average speed up of 979.88x is obtained over all the 16 instances. It should however be noted that for most circuit_equiv versions, only the two-literal clauses resulting from indirect implications were sufficient to obtain significant improvements. The reason is that, since the two circuits mitered were exactly the same, most of the equivalent signals could be identified with indirect implications alone. However, for c499_equiv we needed both indirect and extended backward implications as the circuit contains many XOR gates, and equivalences were not easily determined. For most circuit_opt mitered versions, preprocessing due to indirect implications alone was not sufficient, and we added the two-literal clauses resulting from extended backward implications as well. Since performing extended backward implications requires additional computation, the speedups obtained were lower than those obtained for circuit_equiv versions.

In Table 2, we compare our results with C-SAT-Jnode [5] and P_EQ + Berkmin [7]. Please note that the experimental setups were over different platforms. In [5], the authors introduced incremental learn-from-conflict strategy. They divide the problem at hand into unsatisfiable sub-problems, and add the conflict-induced clauses resulting from solving these sub-problems to the original CNF. In [7], the authors branch on a small subset of variables (called observable variables), analyzing the results of unit propagation, and statically adding the clauses to the existing CNF. According to Table 2, our results are mostly on the same order of computation effort, and in a few cases better than [5] and [7]. It should however be noted, that we offer a novel technique of adding useful correlations over the entire miter circuit, rather than restricting to a subset of variables.

In [14, 15] Silva et al. incorporated the technique of Recursive Learning [16, 17] into SAT-solvers and applied the same to combinational equivalence checking. However, they preprocessed the CNF formula using only depth ‘1’ recursive learning which is different from extended backward implications [9] used in our approach. Hence, even if the computational time reported in their experimental results was scaled down, our technique will still be superior.
Table 2. Comparison of our results with [5] and [7]

<table>
<thead>
<tr>
<th>Miter Circuit</th>
<th>C-SAT-Jnode[5]* (secs)</th>
<th>P_EQ + Berkmin[7]@ (secs)</th>
<th>IMP2C + ZChaff (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1355_equiv</td>
<td>0.14</td>
<td>0.05</td>
<td>0.13</td>
</tr>
<tr>
<td>c1908_equiv</td>
<td>0.23</td>
<td>0.27</td>
<td>0.07</td>
</tr>
<tr>
<td>c2670_equiv</td>
<td>NA</td>
<td>0.17</td>
<td>0.32</td>
</tr>
<tr>
<td>c3540_equiv</td>
<td>2.01</td>
<td>0.83</td>
<td>1.10</td>
</tr>
<tr>
<td>c5315_equiv</td>
<td>0.46</td>
<td>0.61</td>
<td>0.72</td>
</tr>
<tr>
<td>c6288_equiv</td>
<td>9.6</td>
<td>0.17</td>
<td>0.36</td>
</tr>
<tr>
<td>c7552_equiv</td>
<td>3.88</td>
<td>0.87</td>
<td>1.80</td>
</tr>
</tbody>
</table>

*Expts. were run on P-3, 1GHz with 1Gb RAM [5].
@ Expts. were run on P-3, 700MHz with 640Mb RAM [7].

VI. Conclusion and Future Work

A new concept of augmenting the original CNF formula with static logic implications has been presented. Two-literal clauses resulting from indirect and extended backward implications are quickly computed and added to the existing CNF database. These added clauses serve as constraints and help the SAT-solver in the search process. Experimental results for combinational circuit equivalence checking show that using the same state-of-the-art SAT-solver, we achieved an average speedup of more than 950x with the added clauses.

As a future work, we would like to apply the preprocessing based on Static Sequential Logic Implications to improve the SAT-based Bounded Model Checking. The sequential implication relations (crossing time frame boundaries) can be easily computed from the original sequential circuit, and replicated throughout the unrolled circuit. This will induce useful structural information into the CNF formula of the unrolled circuit and hence aid the SAT-solver in proving the satisfiability/unsatisfiability of various safety/liveness properties.

References