Designers Work Less with Quality Formal Equivalence Checking

Orly Cohen, Moran Gordon, Michael Lifshits,
Alexander Nadel, and Vadim Ryvchin
Intel Corporation
P.O. Box 1659
Haifa 31015 Israel
{orly.cohen,gordon.moran,michael.lifshits,alexander.nadel,
ryvchin.vadim}@intel.com

ABSTRACT

Formal Equivalence Checking (FEC) is a technique that formally proves the equivalence of a schematics implementation against a golden RTL model. This equivalence must be guaranteed in light of possible multiple local hand-implemented changes in the schematics. To overcome capacity problems, FEC is usually performed on system sub-blocks, whereas the "environment" is modeled with assumptions written using a property specification language such as SVA. These assumptions must later be proved relative to the driving logic. The majority of FEC tools today are based on SAT-based model checking formal verification engines. In this paper, we describe an approach that can considerably reduce both the time and computational effort required to complete FEC activity in a project. It is based on an additional step introduced to complement the traditional SAT-based model checking algorithm. This step calculates a minimal set of required assumptions using a new SAT-based algorithm. Minimizing the set of assumptions greatly reduces the manual debugging effort required of designers, as well as reduces the number of iterative verifications.

1. INTRODUCTION

Formal Equivalence Checking (FEC) [12] is a powerful technique that formally proves the equivalence of a pair of design models. FEC is used in various places in the design flow, including functional equivalence comparison of the schematics implementation (which has been created either manually or by an automatic synthesis tool) against the golden Register Transfer Logic (RTL) model. This equivalence (or design implementation correctness) must be guaranteed in light of multiple local hand-implemented changes in the schematics that may have been introduced by, for example, timing and power optimizations. FEC is usually performed on system sub-blocks small enough to be suitable for the well-known capacity limitations of formal tools. Since the set of behaviors of a given block in isolation can be much larger than when the block is integrated in a circuit, any unexpected input that the block receives will result in an incorrect behavior. This is known as the Environment Problem [5]. Users overcome this lack of environment by applying assume-guarantee reasoning (e.g. [1]) which expresses restrictions the circuit places on the sub-circuit using a property specification language such as SVA. Formal verification tools treat these properties as assumptions which mimic the "essential" behavior of the environment. These assumptions must, of course, be proved relative to the driving logic, whence they are known as guarantees. By formulating the set of assume and guarantee properties, the correctness of the entire system is demonstrated by rendering the original verification problem into smaller verification problems involving the individual decoupled blocks. It follows that using a smaller set of assumptions is better, since ensuring the validity of the verification requires iteratively proving all the assumptions used. Seligman et al. describe the importance of assumptions verification in FEC and provide an example of a CPU project at Intel that arrived with a dead A0 silicon as a result of a missed assumption verification step [8].

The majority of FEC tools today (we used Seqver [13]) use model checking theory and algorithms [4] and are implemented with SAT-based formal verification engines (e.g. [3]). Generally speaking, they create a propositional boolean formula that is satisfiable if and only if the schematics is not equivalent to the RTL, that is, there is a bug. This formula is constructed as follows. A set of propositional clauses is generated for each circuit element, each assumption, and the equivalence requirement between the schematics and RTL. The propositional formula comprises the set of all the clauses. SAT solvers aim to prove the lack of a satisfying assignment to a given formula, which, in turn, proves the desired equivalence of the models. Modern SAT solvers based on the DLL backtrack search algorithm [6], such as Minisat [9] or Eureka [18], are able to deduce whether extremely large formulas are satisfiable, making it possible to solve verification problems on complex circuits with tens of thousands of sequential elements and millions of gates. If the models are equivalent, it is necessary to prove the assumptions relative to the guarantees. Proving all the assumptions would require excessive effort on the part of designers; however, it is sufficient to prove only such assumptions as were necessary for the proof of equivalence. Fortunately, modern SAT solvers can return all the clauses that were required for the proof. This is called the unsatisfiable core of the formula [11, 20]. The projection of the clauses in the core onto the assumptions represents the subset of assumptions whose proof is required.

However, the number of assumptions returned by applying standard algorithms [11, 20] is still large and can be reduced further. There exist SAT-based approaches for reducing the set of clauses in the unsatisfiable core [10], or even minimizing them [7, 17] in the sense that removal of any further clause from the core would make the problem satisfiable. However, these algorithms are unaware of the mapping between the assumptions and the clauses. Minimization at the SAT level does not imply that the number of assumptions in the core would be minimized (see Fig. 4).

We developed a more efficient algorithm to calculate a minimal set of assumptions required for the proof, in the sense that removing any assumption would make the models non-equivalent. Besides reducing the number of iterative verifications required, minimizing the set of assumptions can greatly reduce the manual effort required of designers. A major part of this effort during FEC consists of debugging property failures. However, the failure of a property usually indicates the lack of an assumption on the inputs of a block rather than a bug. Therefore, minimizing the number of assumptions (i.e. the number of potential failures that need to be debugged) will reduce the number of debugging cycles required and the amount of related designer effort, while achieving the same quality FEC. However, assumption minimization requires additional computation time; we shall demonstrate that it is highly beneficial at certain stages of the FEC flow, while at other stages it is too costly.

The rest of the paper is organized as follows. In Section 2, after briefly describing FEC, we give an overview of the algorithm that efficiently calculates the minimal assumptions set. In Section 3, we discuss the tradeoffs involved in various approaches to reducing the number of assumptions used at the various stages of FEC, and summarize the impact of our approach in a large microprocessor project at Intel. Conclusions and a number of recommendations for implementing the suggested approach follow in Section 4.

2. FEC – FORMAL EQUIVALENCE CHECKING

To overcome capacity limitations, in practice, formal verification is carried out using a divide and conquer approach, usually referred to as compositional verification [5]. A compositional approach to FEC was developed in [12] and [15], where the RTL and implementation models are decomposed into pairs of corresponding slices. From proving the equivalence of all the pairs one can infer the equivalence of the models. The slices need to be small enough to be within the capacity of the formal engines.

The constraints mimicking the "essential" behavior of the slice environment must be added to the slices' inputs using combinational or linear temporal formulas. The theory in [15], unlike [12], supports the tool's use of the input restrictions as assumptions when proving the equivalence of a slice pair. These assumptions must, of course, be proven relative to the driving logic, whence they are known as guarantees. By properly formulating the set of assume and guarantee properties, it is possible to demonstrate the correctness of the entire system, rendering the original verification problem into smaller verification problems involving only the individual decoupled slices. This divide and concur approach is taken almost any time formal verification is applied to real systems (see e.g. [1])

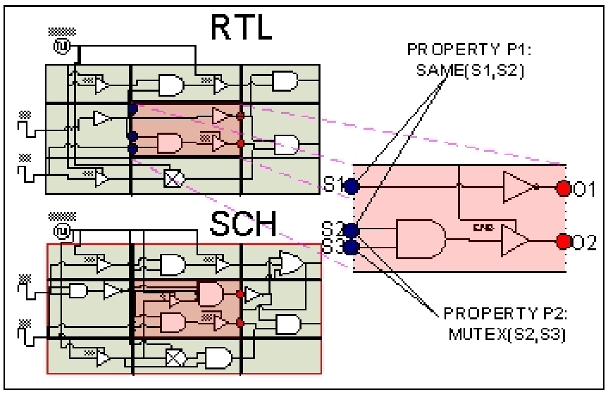
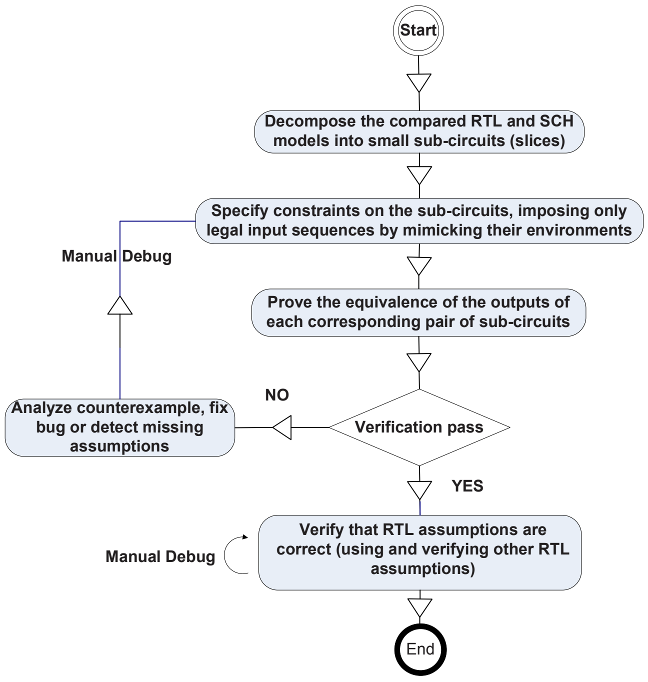


Figure 1: FEC between matching RTL and schematics slices


Start
Decompose the compared RTL and SCH models into small sub-circuits (slices)
Specify constraints on the sub-circuits, imposing only legal input sequences by mimicking their environments
Manual Debug
Prove the equivalence of the outputs of each corresponding pair of sub-circuits
NO
Analyze counterexample, fix bug or detect missing assumptions
Verification pass
YES
Manual Debug
Verify that RTL assumptions are correct (using and verifying other RTL assumptions)
End
Figure 2: Compositional FEC flow stages

The above process is outlined in Figure 1. FEC is performed between matching RTL and schematics slices (marked with darkened closed rectangles) on the left side of the Figure. Constraints $P1$ and $P2$, placed on the RTL slice inputs, are shown on the right side of the Figure (on the enlarged slice).

Figure 2 outlines the stages of the composition FEC flow that proves the equivalence of the specification (RTL) and implementation (circuit) models.

Note that the last stage includes multiple iterative verifications. First, the set of assumptions used in the previous stage is verified. This process uses other assumptions residing in the RTL following the assume-guarantee reasoning.

These assumptions are then verified using other assumptions and so on. Properties in the RTL come from various sources (for example, design intent properties are written in the RTL code, capturing intended design behavior) to enable Assertion-Based Verification (ABV) methodology. These properties are considered assumptions when used during verification as constraints, and therefore need to be verified to close the Assume-Guarantee loop. The latest work of Khasidashvili et al. [14] describes how these properties can be proved locally in RTL contexts that are different from the slices used in FEC.

Each verification ends with one of the following results: *Pass*, *Fail* (in this case a counterexample trace is returned by the tool), *Problematic* (due to lack of memory or timeout), and *Conditional* - the property itself passes but its correctness is conditionally dependent on the status of other properties (those used as assumptions for its verification). Failing and problematic properties should be manually addressed. The traces of failing properties should be debugged – an interactive and iterative process that requires the hands-on involvement of the designers, who must identify and resolve the root causes of the failures [16]. Although some approaches have been suggested to automate this time-consuming process [16], it still requires a substantial effort in practice. Debugging results in adding missing assumptions (in most cases) or in a bug fix. Various manual techniques for logic reduction are usually required to help formal verification tools resolve problematic results. Unfortunately, conditional properties need to be addressed as well – the counterexample traces of the failing assumptions are debugged by the designers. [8] describes a case where a CPU project at Intel came up with dead A0 silicon due to the fact that an engineer running FEV missed an assumption failure in the design and did not make sure that all used assumptions were fully verified.

2.1 Minimal Assumption Set Calculation

From the previous discussion it follows that using a smaller set of assumptions is better, since to ensure the validity of the equivalence verification one has to iteratively prove all the assumptions used. There exist various approaches to reducing the number of used assumptions in formal verification. In this section we review several approaches, mentioning their drawbacks to justify our regarding them as unsuitable for our purpose.

2.1.1 Static Structural Analysis

One can analyze the logical cone of influence of the property in the DUT and consider only those assumptions directly affecting the property – that is, whose cones intersect the cone of the property. Consider Figure 3. The cones of influence of the assumptions A1 and A2 (marked with dark grey) intersect the cone of the property P1, therefore affecting the behavior of DUT inputs relevant for property P1. However, assumption A3 implicitly affects property P1 through A2. This shows how such naive approaches are prone to under-constraining the DUT inputs, leading to false negatives and redundant debugging cycles.

2.1.2 Iterative Trial and Error

One can try an iterative verification method, gradually adding assumptions until they are sufficient for the proof. The steps of this approach are outlined in Algorithm 1.

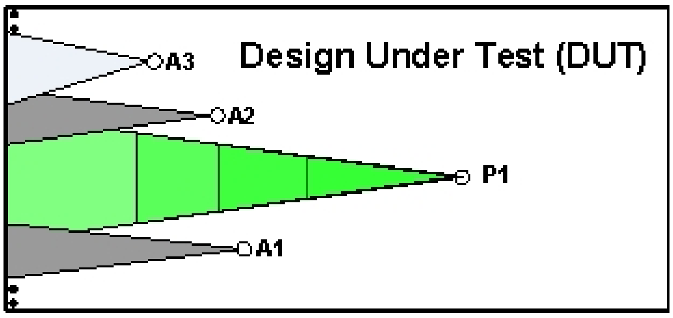


Figure 3: Assumption selection based on structural analysis

Algorithm 1 Trial and Error Iterative Assumption Minimization (*Assump*)

```
$MinAssump := \emptyset$. Try proving the equivalence without any assumptions
while verification fails and $MinAssump \neq Assump$ do
    try proving the equivalence using only assumptions in $MinAssump$
    use the counterexample (CEX) returned and find $A \in Assump$ : A not in $MinAssump$ and $A$ contradicts with CEX trace
    add (at most 20) such assumptions to $Assump$
return $MinAssump$
```

Although this method was used in the Intel FEC flow until recently, it has multiple drawbacks – it includes multiple verifications trying various sets of assumptions for each pair of design slices. Moreover, as we will see when comparing methods in Section 3, it results in a non-optimal set of assumptions.

2.1.3 SAT-based Algorithms

Here we describe SAT-based approaches to reducing the number of assumptions used to prove the equivalence between the schematics and RTL. Our novel algorithm for assumption minimization is based on the known technique of projecting the data calculated at the SAT solver level back onto the DUT.

Suppose that we are given a set of *entities*, where each entity is either a design element, an assumption, or the equivalence requirement between the schematics and RTL. Suppose that the schematics is equivalent to the RTL. Each entity *entity* can be translated to a set of clauses *Clss*(*entity*), called the *clause projection* of *entity*. We denote the set of all the assumptions by *Assump* and the union of all the circuit elements and the equivalence requirement by *Rest*. Consider Algorithm 2. It receives a set of assumptions *Assump* and a set containing the rest of the formula *Rest*. It returns a minimal set of assumptions required for the equivalence proof, *MinAssump*. In other words, after applying the algorithm, it is guaranteed that $Rest \cup MinAssump$ is unsatisfiable and that removing any assumption from $Rest \cup MinAssump$ would make it satisfiable. The algorithm finds an approximation of *MinAssump* by invoking a SAT solver and placing every assumption whose clause projection intersects with the unsatisfiable core into *MinAssump*. This approximation results in a reduced set of assumptions, and

Algorithm 2 Minimize Assumptions ($Assump$,$Rest$)

```
Solve $Clss(Rest \cup Assump)$ with SAT and extract the unsatisfiable core $UC$
$MinAssump := A \in Assump$: $Clss(A) \cap UC$ is non-empty;
for all $A \in MinAssump$: $Clss(A) \cap UC$ is non-empty do
    Solve $Clss(Rest \cup (MinAssump \setminus \{A\}))$ with SAT
    if the result is "unsatisfiable" then
        $MinAssump := MinAssump \setminus \{A\}$
return $MinAssump$
```

the reduced set can be sufficient if computation running time is a serious limiter. We refer to this algorithm as *assumption reduction by unsatisfiable core projection.*

Our algorithm further minimizes the core after the SAT solver computes its first approximation. This requires additional computation time, and hence it is justified mainly when minimizing the core is relatively more important than reducing the run-time. It iterates over all the assumptions remaining in $MinAssump$. For each assumption $A \in MinAssump$, the algorithm checks if it can be removed from $MinAssump$ by invoking the SAT solver on the clause projection of all the assumptions in $MinAssump$, except for A, and the rest of the formula $Clss(Rest \cup (MinAssump \setminus \{A\}))$. If the SAT solver concludes that this formula is unsatisfiable, it means that it found an equivalence proof without need of A. Hence, A can be removed from $MinAssump$.

Modern SAT solvers learn the so-called conflict clauses [2, 19] during the search. Conflict clauses are lemmas that make the subsequent search substantially faster. To improve the performance of our algorithm, it is essential to re-use, as much as possible, conflict clauses that were learned in the SAT solver's invocations. Roughly speaking, we re-use any conflict clause that was not derived using removed assumptions by providing it to any subsequent invocation of the SAT solver. The idea of re-using conflict clauses in the context of minimal unsatisfiable core extraction was proposed first in the context of the CRR algorithm for minimal unsatisfiable core extraction at the clause level [7, 17].

3. IMPACT OF THE ASSUMPTION MINIMIZATION STAGE ON FEC

In this section we describe the experiments we performed to quantitatively measure the impact of the proposed approach to assumption minimization in a large microprocessor project at Intel. Assumption minimization is important in all the verification stages involved in FEC (see the diagram in Fig. 2). Each verification stage introduces new assumptions which, in turn, should be verified at a later stage. The original equivalence verification is not complete until all the used assumption failures are cleared. It follows that minimizing the set of used assumptions is most important at the early stage, when the equivalence of the corresponding RTL and SCH pairs is proven (the first "manual debug" loop appears at this stage in Figure 2). It is also relevant for later stages, although in these stages a less accurate (and thus less time-consuming) approach can be considered.

Assumption minimization comes with a price – it requires additional computational effort, e.g. to iterate over the assumptions after unsatisfiable core calculation (see Algorithm 2). Next we describe the experiments we performed to estimate the computational overhead versus effectiveness of the various approaches to reducing the number of used assumptions. We further discuss the tradeoffs for achieving the right balance between desirable minimization characteristics and run-time at various FEC stages.

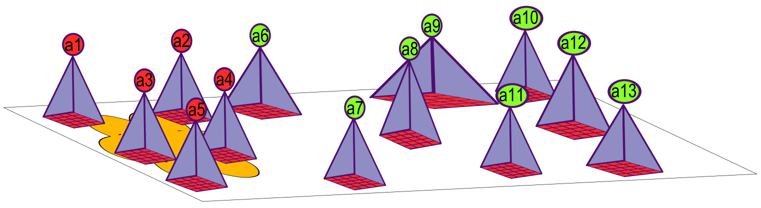
We compared the results for minimizing the set of used assumptions at the first, RTL and SCH equivalence verification FEC stage. A comparison of run-time and reduction results for 22 randomly chosen blocks from microprocessor design are shown in Figure 5. The red columns show the results of the "naive" iterative algorithm described in Algorithm 1, while the green columns indicate the results of the minimization algorithm based on the unsatisfiable core described in Algorithm 2. Numbers marked over each green/red column pair show the improvement of the minimization algorithm over the iterative one, where 100% means that both algorithms resulted in the same amount of used assumptions, 50% means that the minimization algorithm resulted in half as many assumptions and so on. The run-time [1] of both algorithms was similar (not shown here). The minimization algorithm did a much better reduction job, resulting in half as many assumptions in most cases, and dramatically fewer in some cases (e.g. B17 and B18).

The last FEC stage where RTL assumptions are verified is usually much more computationally intensive as compared to the other stages. This is because the cone of influence of ABV RTL properties, used in FEC as assumptions, stretches beyond the boundaries of the relatively small and mostly combinational slices of the design used for a proof in earlier FEC stages. It can reach out to the primary inputs of the DUT and contain large sequential logic. As a result, the assumption minimization algorithm may become practically unfeasible due to the long run-time.

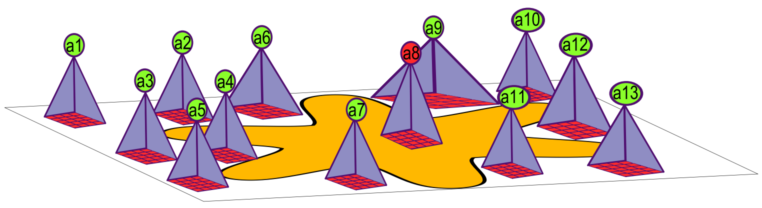
We compared the results and run-times of two approaches to reducing the set of used assumptions. The first approach calculates a subset of all assumptions using the projection of the unsatisfiable core calculated with standard algorithms [11, 20] onto the assumptions. See the illustration in Fig. 4a – assumptions $a1 \dots a5$ are returned instead of $a1 \dots a13$. The second approach uses our algorithm for unsatisfiable core minimization with respect to assumptions. See the illustration in Fig. 4b – only assumption $a8$ is returned instead of $a1 \dots a13$.

We show the results for 4 microprocessor design blocks with 71, 50, 75 and 86 RTL properties that needed to be proven to complete FEC. The number of assumptions used without applying any of the assumption reduction approaches resulted overall in 13019, 10431, 23836, and 31101 assumptions for these blocks, respectively. The results of the run-time and reduction comparison are shown in Fig. 6. Red columns refer to the standard UNSAT core projection algorithm, while the green columns indicate the result of the UNSAT core minimization algorithm 2. The lines indicate the run-times of the algorithms. It can be seen that both algorithms considerably reduced the number of assumptions – the UNSAT core projection algorithm resulted in about a fifth, and the UNSAT core minimization algorithm in about a twentieth, of the original number of assumptions.

[1]The compared slices are small in most cases, and the related equivalence verification does not take much time to complete, even when the longest assumption minimization algorithm is used. Therefore we haven't experimented with faster assumption reduction using unsatisfiable core projection.



(a) Minimizing SAT clauses



(b) Minimizing assumptions

Figure 4: Minimal unsatisfiable core with respect to SAT clauses (a) and DUT assumptions (b). The rectangle represents all the clauses within the SAT instance. Pyramides $a1 \ldots a13$ represent the assumptions. Unsatisfiable cores are shown with blobs that contain some rectangle space or some of the SAT clauses. Assumptions whose clauses (pyramid bases) intersect the unsatisfiable core are marked with red circles - they are returned by the algorithms.

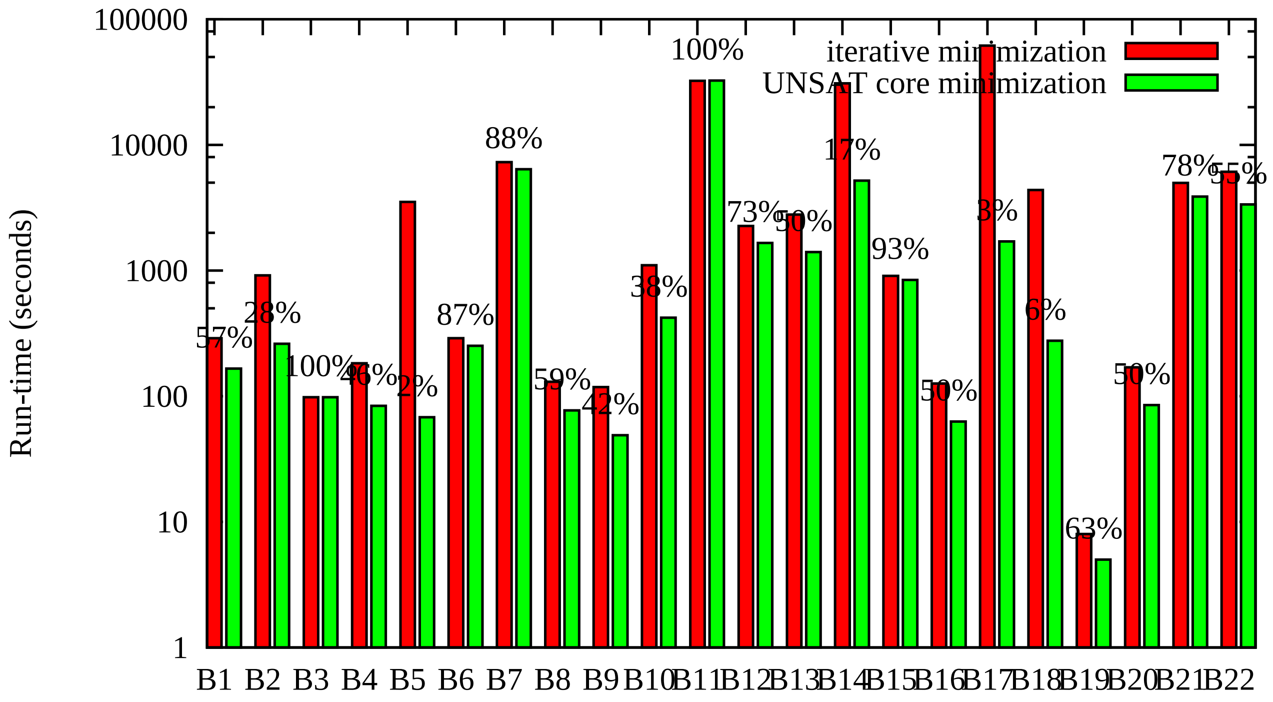


Figure 5: Assumption minimization with unsatisfiable core (see Alg. 2) vs. trial and error iterative minimization (see Alg. 1). The time is shown with logarithmic scale. Percents indicate the number of assumptions in minimization algorithm compared to the iterative one (e.g. 50 indicates $2\times$ reduction).

However the minimization algorithm took up to 40 hours to complete whereas the projection algorithm finished in a fraction of this time. Considering the above we decided to use the faster approach for this stage.

The following results demonstrate the amount of manual designer effort that can be saved using assumption minimization. We experimented with three blocks (let's denote them DUT1, DUT2 and DUT3) including thousands of RTL properties. Earlier FEC stages used 2200, 776, and 3200 RTL properties as assumptions, and thus they needed to be verified. Let's denote these as Used-By-FEC RTL properties (UBF). We ran the verification twice for all RTL properties, first with assumption reduction using the standard UNSAT core projection technique and then without assumption reduction.

Figure 7 summarizes the verification results for the three blocks – it shows the average number of assumptions used per property and the percentage of failing assumptions. The degree of assumption reduction varies between DUTs, but the number of used assumptions is in every case considerably reduced: by $\sim 50\times$ for DUT1 (153 vs. 3.8) and by $\sim 2\times$ for DUT2 and DUT3. Moreover, the rate of failing properties, those requiring manual debug, is also significantly reduced.

We used an SQL database to store the verification results and calculated the combined verification status for the UBF properties in the following way. We used a recursive function to calculate the full set of assumptions that each UBF property depends on. That is, for each UBF property P we queried for the set of assumptions ($Assump$) used to verify a property P, then for each assumption $A_i \in Assump$ we

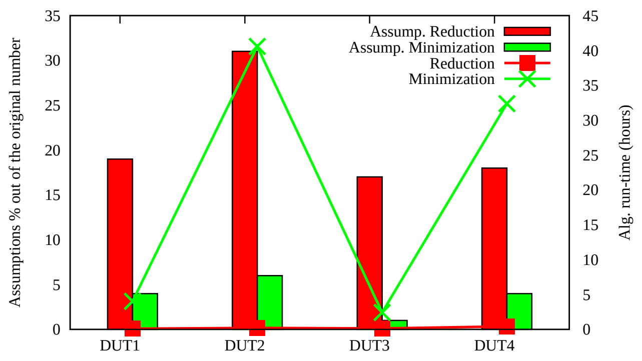


Figure 6: Assumption reduction using UNSAT core projection vs. UNSAT core minimization (see Alg. 1)

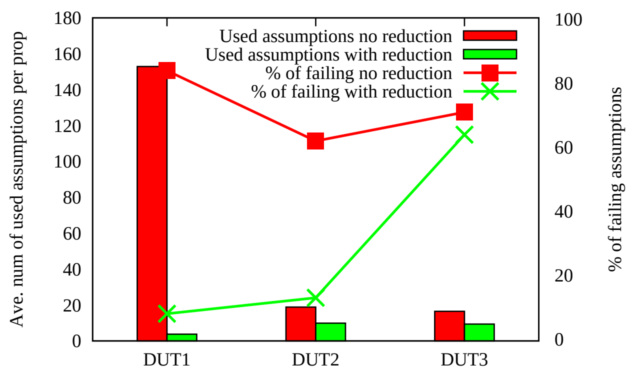


Figure 7: Impact of assumption reduction on the number of failures to debug

queried for the set of assumptions ($Assump_i$) used to verify a property A_i and so on. The final set that each UBF property depends on is received from unifying: $Assump_all = Assump \cup Assump_i \ldots \cup Assump_n$. UBF property passes iff all the properties within $Assump_all$ pass and has conditional verification result if any property within $Assump_all$ fails.

Figure 8 summarizes the averaged verification results for the three blocks.

It can be seen that, on average, an additional 36 percent of the properties passed when assumption reduction was used (they were conditional, i.e. dependent on an assumption that failed verification, when no assumption reduction was used). This is because $Assump_all$ sets were smaller when using assumption reduction and so too the chances of a failing assumption being included in the $Assump_all$ set. This implies that an additional 36 percent of the properties were formally proven and hence do not require further attention from designers. Considering the large numbers of properties used in FEC, this saves the design team a huge amount of manual effort.

4. CONCLUSION AND RECOMMENDATIONS

The summary of our findings is as follows.

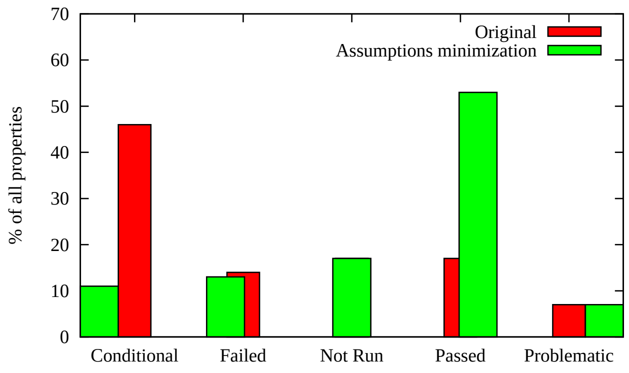


Figure 8: Impact of assumption reduction on the amount of conditional properties

- Reducing the set of used assumptions has a significant, positive efficiency impact on all FEC stages, decreasing both manual debug time and computational effort. The impact is greatest in the early FEC stages.
- Unsatisfiable core-based reduction techniques are much more effective than straightforward iterative or structural techniques.
- There is a tradeoff between the effectiveness of assumption reduction techniques and the amount of additional computation time they require.
- Different assumption reduction techniques should be applied at the various FEC stages as those stages vary in terms of the complexity of the verification (the complexity is much greater for RTL assumptions in the last stages than it is for proving RTL and schematics equivalence in the first stage) and the importance of assumption reduction (smaller for the verification of RTL assumptions than it is for proving RTL and schematics equivalence). Assumption minimization should be applied for RTL and schematics equivalence, and assumption reduction should be applied for RTL assumption verification.

We implemented a novel assumption minimization algorithm that calculates a minimal set of required assumptions using a new SAT-based algorithm. It achieves the best minimization results compared to all other approaches, but may take long time to complete in case of hard-to-solve verification instances. Based on our experience with the tradeoffs between the accuracy of assumption reduction and run-time, we suggest employing different algorithms at different FEC stages, thereby considerably reducing overall both the time and the computational effort required to complete the FEC activity in any project. Based on our learning, we believe that any design team can run FEC with fewer resources than is currently the practice and still achieve the same level of confidence.

Acknowledgment

The authors would like to thank Paul Inbar, Zurab Khasidashvili, Kim Joonyoung, Haim Kerem, Kaiss Daher and Amit Palti for their valuable suggestions, reviews, and support of our work.

5. REFERENCES

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[2] R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, 1997.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[4] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.

[5] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[7] N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In A. Biere and C. P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, 2006.

[8] J. K. E. Seligman. FevŠs greatest bloopers: False positives in formal equivalence. In *DVCon*, 2007.

[9] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *CAV*, pages 109–122, 2006.

[11] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10886, Washington, DC, USA, 2003. IEEE Computer Society.

[12] S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design DeBugging*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[13] D. Kaiss, S. Goldenberg, and Z. Khasidashvili. Seqver : A sequential equivalence verifier for hardware designs. In *ICCD*, 2006.

[14] Z. Khasidashvili, D. Kaiss, and D. Bustan. A compositional theory for observational equivalence checking of hardware. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2009.

[15] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-Aided Design*, pages 58–65, Washington, DC, USA, 2004. IEEE Computer Society.

[16] J. Moondanos. From error to error: Logic debugging in the many-core era. *Electron. Notes Theor. Comput. Sci.*, 174(4):3–7, 2007.

[17] A. Nadel. *Understanding and Improving a Modern SAT Solver*. PhD thesis, Tel Aviv University, August 2009.

[18] A. Nadel, M. Gordon, A. Palti, and Z. Hanna. Eureka-2006 SAT solver. In *Solvers description, SAT-race*, 2003.

[19] J. P. M. Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[20] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.