A novel formal verification technique to System verification

Surinder Sood

Motivation and Problem statement

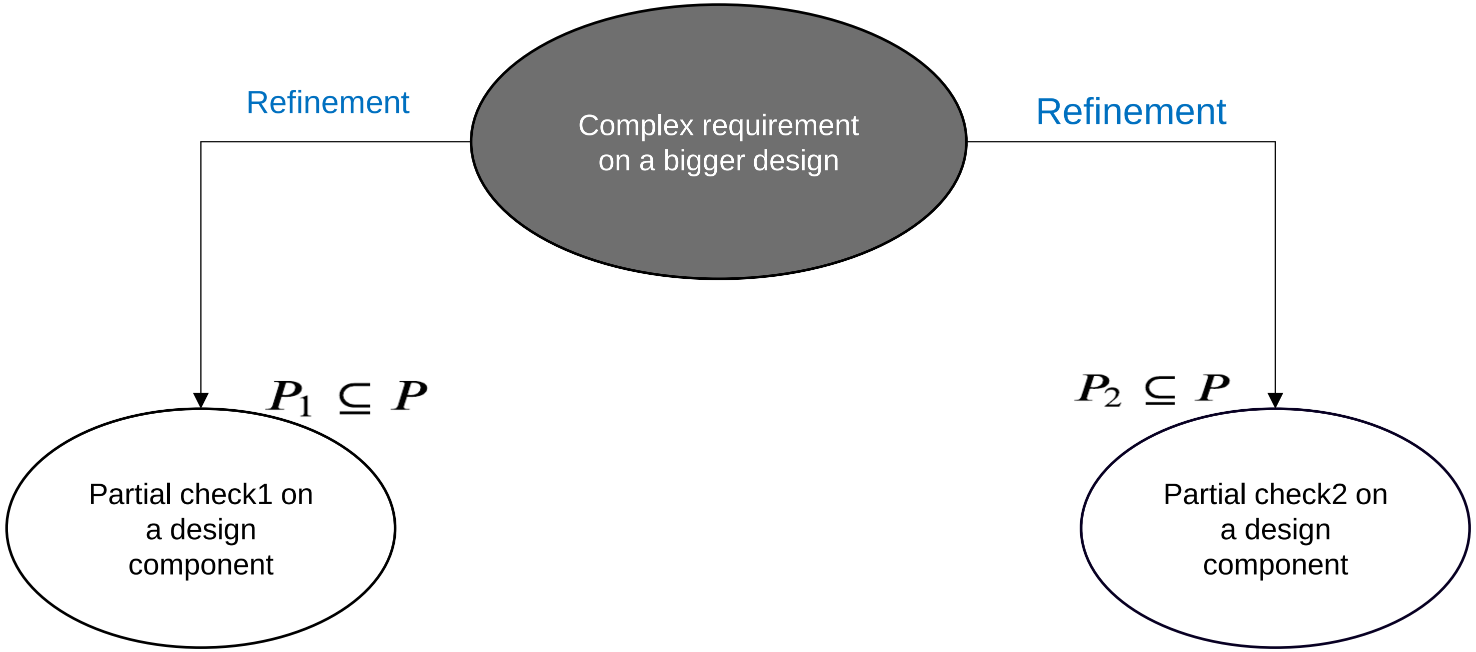
- There is no technique available to verify specific system[2] behavior which should guarantee
  - Completeness
  - Correctness
  - Consistency
- Existing formal techniques are limited only to component level designs, but guarantee 3Cs (mentioned above)[1]
- Proof convergence on larger designs is difficult to achieve using available state-of-the-art formal techniques

[1] https://dvcon-proceedings.org/document/lets-be-formal-while-talking-about-verification-quality-a-novel-approach-to-qualify-assertion-based-vips/
[2] A system is defined as an integration of two or more components

arm

Main Idea: Proof convergence using decomposition and refinement

$$P = System\, level\, property\, having\, assumptions\,(A)\, and\, guarantees\,(G)$$



Sub-step results together make overall system level proof

Refinement Explained

Suppose we have a **system-level requirement**:**"If a request is made, it will be acknowledged within 3 cycles."**

- This can be written as a temporal property: *P: always (req |=> ##[1:3] ack)*

This is our top-level contract:
- **Assumption (A)**: none
- **Guarantee (G):** *req |=> ##[1:3] ack*

<u>Decomposed</u> into Subcomponents

Suppose the system has two submodules

1.Request Manager (generates) *req_out* when *req_in* is received)

Contract P1 for Request Manager

P1: always (req_in |=> ##1 req_out)

- **Assumption (A1):** none
- **Guarantee (G2):** req_in |=> ##1 req_out

This refines the behavior: *req_in* leads to *req_out* in 1 cycle.

2.Responder (acknowledges req_out with ack)

Contract P2 for Responder

P2: always (req_out |=> ##[1:2] ack)

- **Assumption (A2):** *req_out follows req_in* quickly
- **Guarantee (G2):** ack within 1–2 cycles after req_out

It refines the part of the system that converts *req_out* into *ack*

✅ Refinement Validity

Now we check:

- $A = A_1 \cup A_2 =$ none ∪ (req_out follows req_in) ⇒ okay
- $G = G_1 \cap G_2 \Rightarrow$ `req_in |=> ##1 req_out` and `req_out |=> ##[1:2] ack` together imply `req_in |=> ##[2:3] ack` ⇒ matches system-level `P`

Hence, P_1 **and** P_2 together **refine** P.

Example: Assumption weakening

- **Definition**: Making the assumptions on the environment *less restrictive*, i.e., the module must work in *more scenarios*.

Example 1:

Original assumption:

*This means the environment **must** respond to a request with an acknowledgment in 1–2 cycles.*

```
assume property (req_valid |-> ##[1:2] ack_received);
```

Weakened assumption:

*Now, the environment may take **up to 5 cycles** — this is weaker because it allows more behaviors*

```
assume property (req_valid |-> ##[1:5] ack_received);
```

Example 2— weaken data constraints:

Original : assume property (req_valid |-> (req_data == 8'hFF));

Weakened: assume property (req_valid |-> (req_data[7:4] == 4'hF)); // only high nibble fixed

The component can tolerate more behaviors from its environment than the global system assumption. This is called **weakening the assumption** — i.e., it assumes *less*. Example: A2⊒A

Example: Guarantee Strengthening

- **Definition**: Making the guarantees of a component *more demanding*, i.e., the module promises to behave *better* or under *broader* conditions.

Example 1:

Original guarantee: *The module guarantees to issue a grant **within 1–3 cycles**.*

```
assert property (req_valid |-> ##[1:3] grant);
```

Strengthened Guarantee: Now, the module guarantees to respond *sooner* — this is stronger.

```
assert property (req_valid |-> ##[1:2] grant);
```

Example 2: Another form of strengthening:

Original guarantee: *Only guarantees grant when req_valid.*

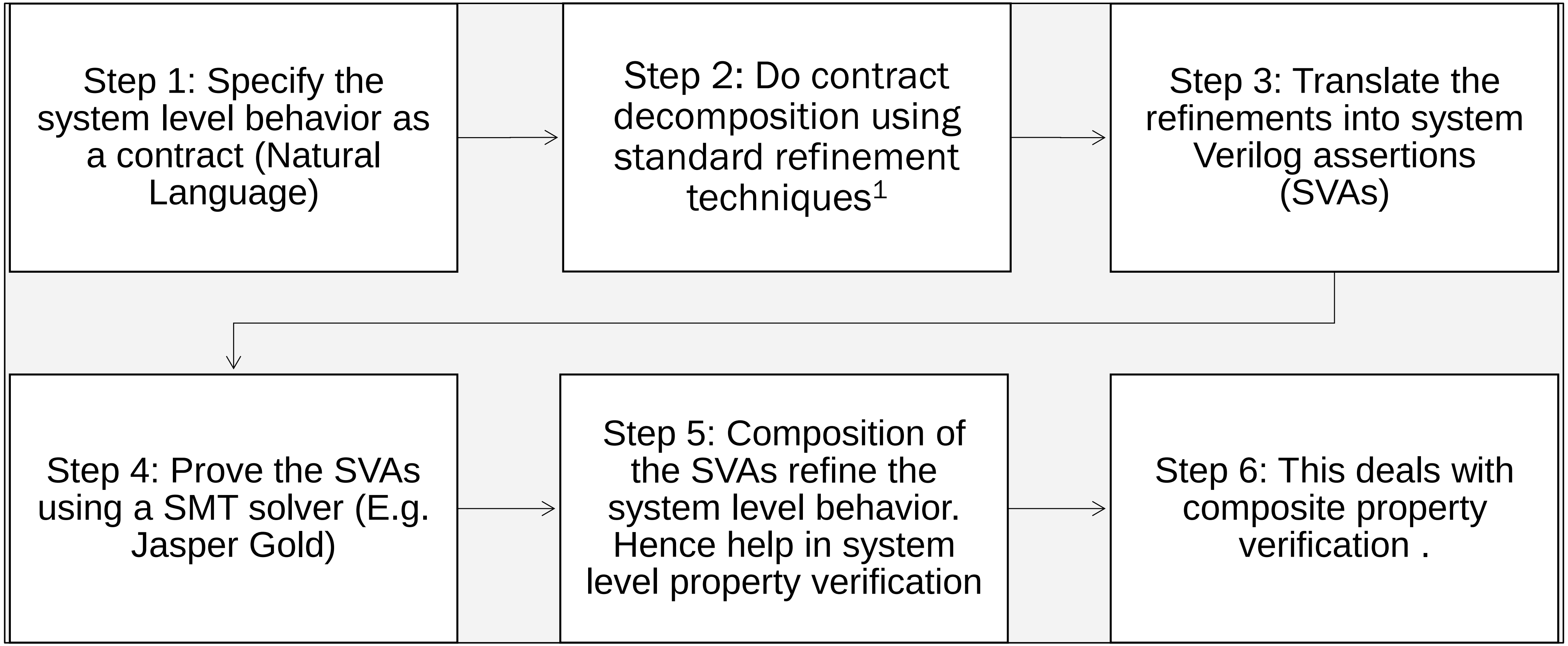
```
assert property (req_valid |-> (grant == 1'b1));
```

Strengthened Guarantee: *Guarantees grant under more general condition (e.g., not just current request, but any pending one too).*

```
assert property ((req_valid || pending_req) |-> (grant == 1'b1));
```

The component provides at least the same guarantees as the system-level property — possibly more. This is called strengthening the guarantee. Example: $G2 \subseteq G$

Proposed Solution



[1]A. Cimatti and S. Tonetta, “Contracts-refinement proof system for component-based embedded systems,” Science of computer programming, vol. 97, pp. 333–348, 2015.

Evidence: Formal verification of a System memory management unit

- A SMMU has in build TLBs and caches which provide faster access to various DMA requests from I/O devices before they are passed to the system interconnect .
- These caches behave in a similar way as the processor caches

- Example system level behaviors for SMMU
  - Any invalidation request also invalidate the corresponding caches [E1]
  - Any system level invalidation/Sync request is eventually Acknowledged [E2]
  - A design Bug introduced in E2[E3]

System level behavior	Time taken (conventional formal technique/Proposed technique)	Property converging: Conventional/Proposed technique	Comments
E1	122271/106541 seconds (13 % improvement)	No/Yes	The system level property did not converge the property
E2	5200/4100 seconds (21 % performance improvement	Yes/yes	Proposed technique converged property faster
E3	49588/48849 seconds (2 % improvement)	No/Yes	Proposed solution caught the bug, while conventional solution did not converge at all

Evidence: Formal verification of Granule protection check wrapper block

A GPC logic checks that a memory access to a physical address space (PAS) is allowed, according to the Granule Protection Tables (GPTs). It retrieves the relevant GPT entries from either the Granule Cache, which caches recently used GPT entries, or system memory. If a GPC does not pass, this is referred to as a Granule Protection Fault (GPF).

System level behavior	Time taken (conventional formal technique/Proposed technique)	Bound achieved	Comments
E1	45319/46571 seconds (3% improvement)	14/24(Got Cex)	**invalidate_all_must_depart_after_gmb_inv_ack**: All Invalidate operations must follow the GPC invalidations
E2	13888/23449 seconds (40 % improvement)	14/20	**sync_must_not_overtake_trans_faults:** A synchronization operation must not overtake translation faults Using conventional approach bound was not increasing, but with our approach the property bound showed increase, although property did not converge

Summary

1. Standalone properties consume more resources, their probability of convergence is still less
2. Compositional system level properties are automatically proven if their corresponding component level refinement properties are proven
3. Better refinement strategy gives faster convergence
4. If the system level property does not converge with the proposed technique, it still gives a better bound as compared to standalone properties
5. Further research is required to create more efficient refinement techniques and deployment of AI can also be done for faster convergence

arm