Speaking Bluntly about SharpHDL:
Some Old Stuff and Some Other Proposed Future Extensions

Gordon J. Pace & Christine Vella

Synchron'05

Malta, November 2005

Part I:
SharpHDL – An Object-Oriented Approach to Circuit Description

In the beginning there was…

- Lava
  - A HDL embedded in Haskell – a functional programming language.
  - Circuits are expressed as functions.
  - It allows simulation and formal verification of circuits.
  - It includes implementations of various *Connection Patterns.*
  - Circuit descriptions are concise and elegant.

… and there was …

- JHDL
  - A HDL embedded in Java – an object-oriented programming language.
  - Circuits are treated as objects and classes.
  - It provides simulation and analysis tools and also a CAD suite.
  - … but
    - It does not provide formal verification
    - No *connection patterns* are implemented.

SharpHDL – An Introduction

- SharpHDL is an object-oriented structural HDL embedded in C#
- It consists of:
  - A library of basic components which are needed to build a circuit structurally, such as wires, logic gates and ports;
  - A *Connection Patterns* library;
  - The ability to perform model-checking via external tools (right now, we only link to to SMV);
  - The ability to convert SharpHDL circuit description to standard HDLs or netlist (right now, we can only export to Verilog).

Example I:
Regular Expression Circuits

- Regular expression circuits are circuits that describe regular expressions. The language is embedded in SharpHDL.

- Being that it is a SharpHDL-embedded language
  - The new language is an embedded language,
  - Its library constructs are objects in SharpHDL,
  - It can use the various facilities offered by SharpHDL, including tools like generating SMV code which can be used for verification.

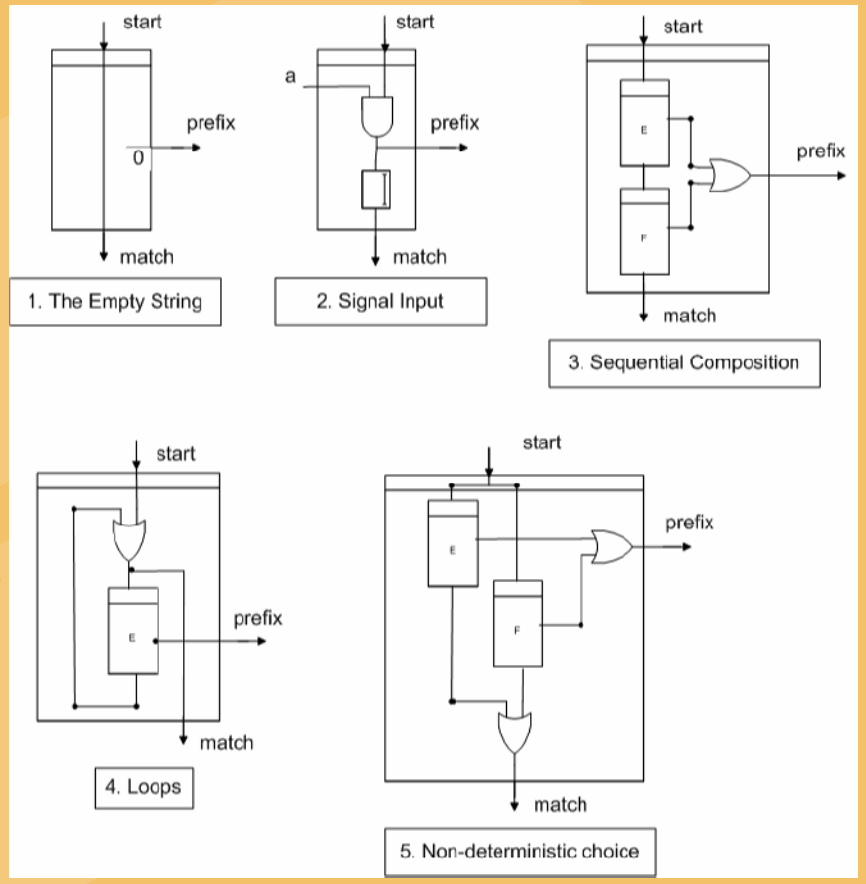
- Creation of language hierarchy:

  C# - SharpHDL - Regular Expression Circuits

Regular Expression Language Implemented Constructs

- The implemented constructs are:
  - The Empty String
  - Signal Input
  - Sequential Composition
  - Iteration
  - Non-Deterministic Choice

- All constructs compile down to a circuit having one input *start* and two outputs *match* and *prefix*, with each construct having its own structure.



Example II:
Fast Fourier Transform Circuits

- Fast Fourier Transform (FFT) is a computationally efficient algorithm for implementing the Discrete Fourier Transform which is used to transform discrete data from the domain of time or space to the frequency domain.

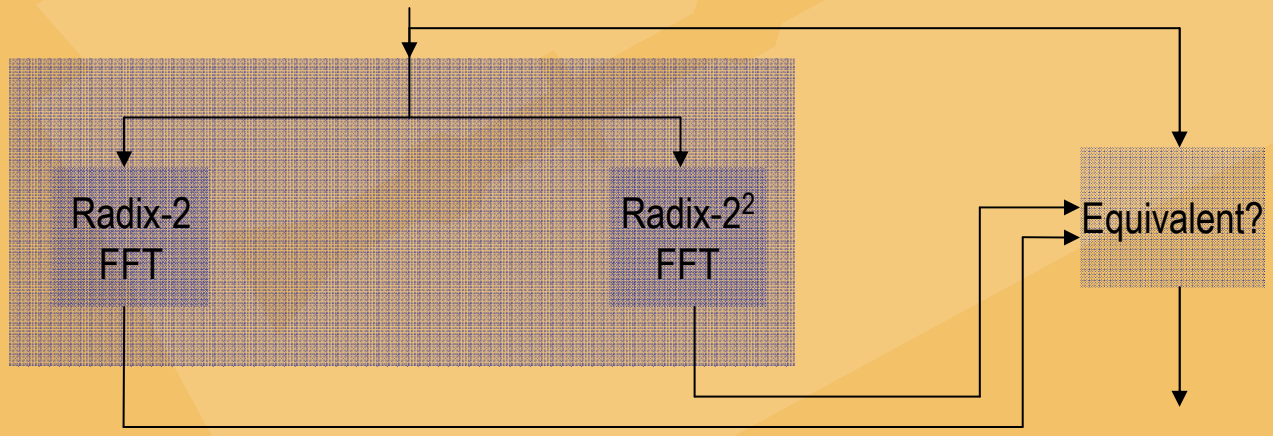
- Various FFT algorithms exist including *radix-2* and *radix-2*^{2} algorithms. These are the most popular due to their simple structure with regular butterfly geometry.

- SharpHDL was used to implement and verify the circuits describing these Fast Fourier Transform algorithms.

- Butterfly circuits are implemented in SharpHDL as another type of generic circuit having a recursive structure.

Verifying the equivalence of two FFT circuits

- The two FFT circuits were verified to be equivalent via a translation to SMV.



Advantages of the SharpHDL approach

- As with all embedded languages, reusability of
  - The host language's syntax;
  - The powerful features;
  - The well-tested tools like parser generators, lexical analyzers etc.
- SharpHDL circuits are treated as objects in the C# programming language, hence enabling straightforward hierarchical organization.
- Imperative nature of C# gives us names of objects for free, and updating of blocks is easier than in Lava-like FP based languages.

Conclusions for Part I

- Original aim was to explore a Lava-like but more imperative approach to hardware description and design.
- We did not manage to sufficiently exploit the imperative-nature, inheritance, etc to enable more abstract descriptions.
- Haskell's syntax, and (syntactically) lightweight high-level abstraction techniques are difficult to beat.

A Short Part II: What about aspects of circuits which may be changed/updated?

Specifications

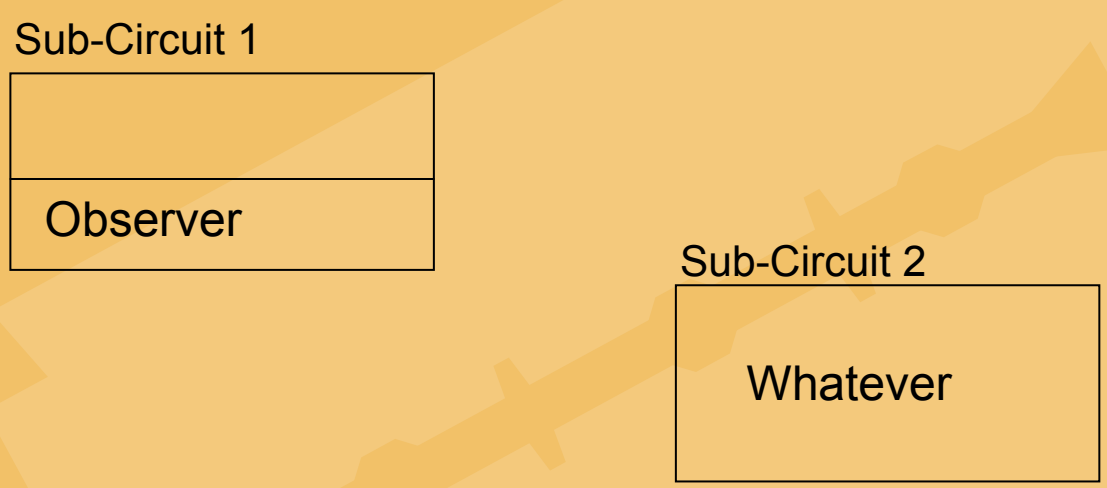
- We chose to look at the refinement process – a meta-aspect of the circuit.
- We enrich our embedded language to enable the designer to specify and document the refinement process, also enabling compositional verification.
- Eventually we would like to give the designer more control over the verification process.

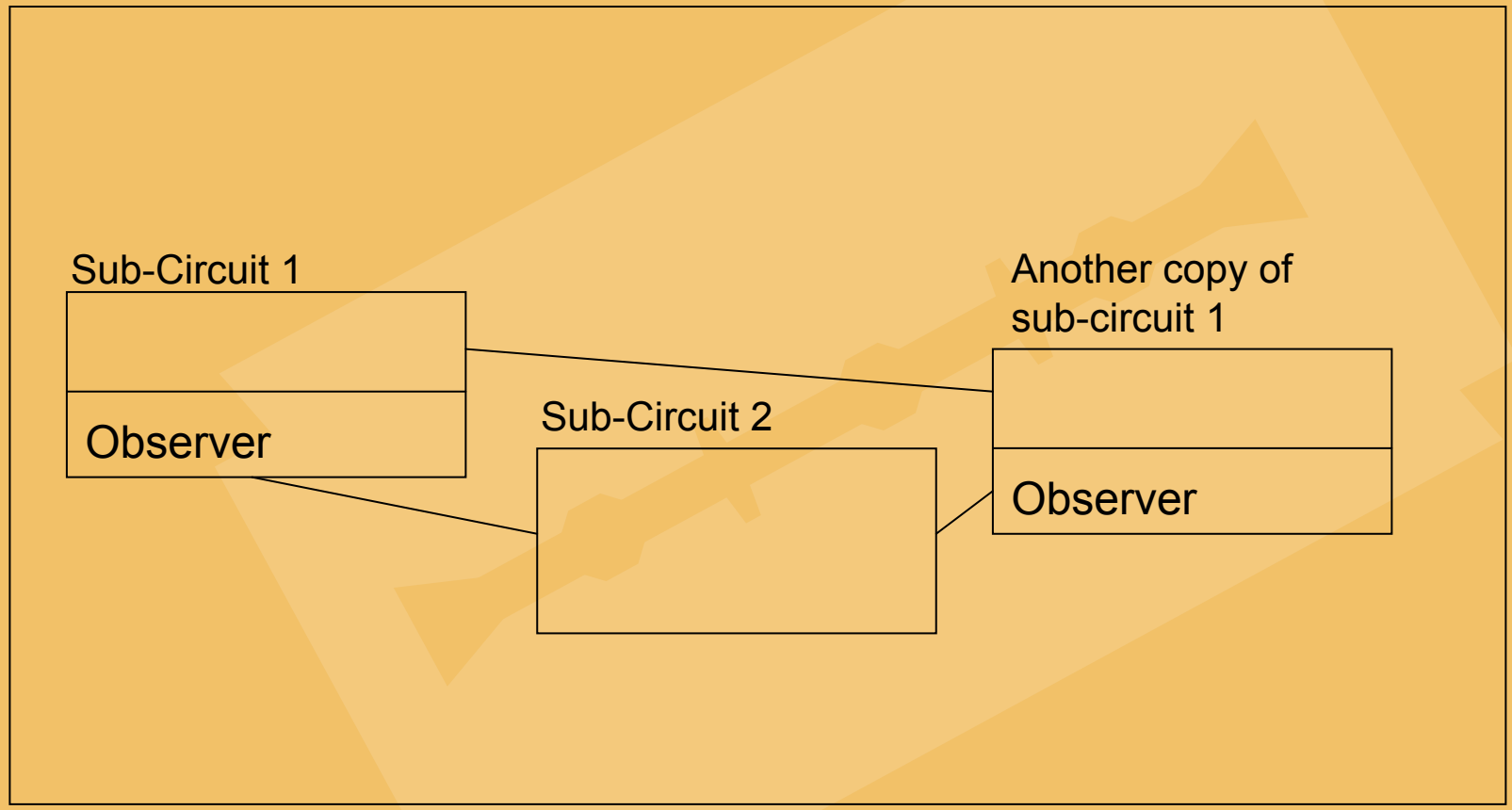
Specifications

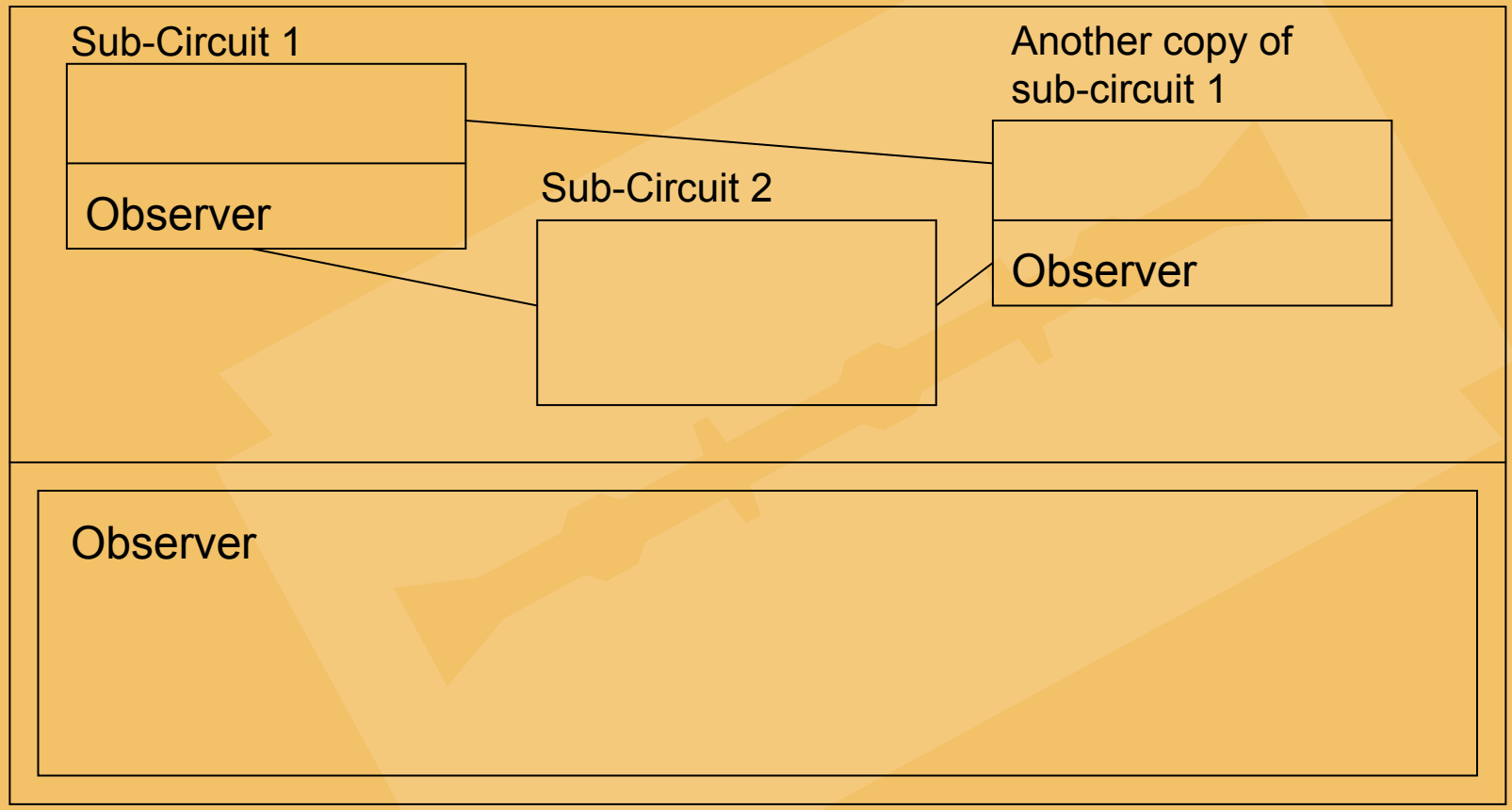
- We extend circuit descriptions by a specification construct – allowing us to create a circuit assumed to behave according to synchronous observer;
- We *cheat* by viewing the specification in both senses – either we assume a block to behave as specified, and verify it when refined. We use it both as an assumption and a proof obligation.

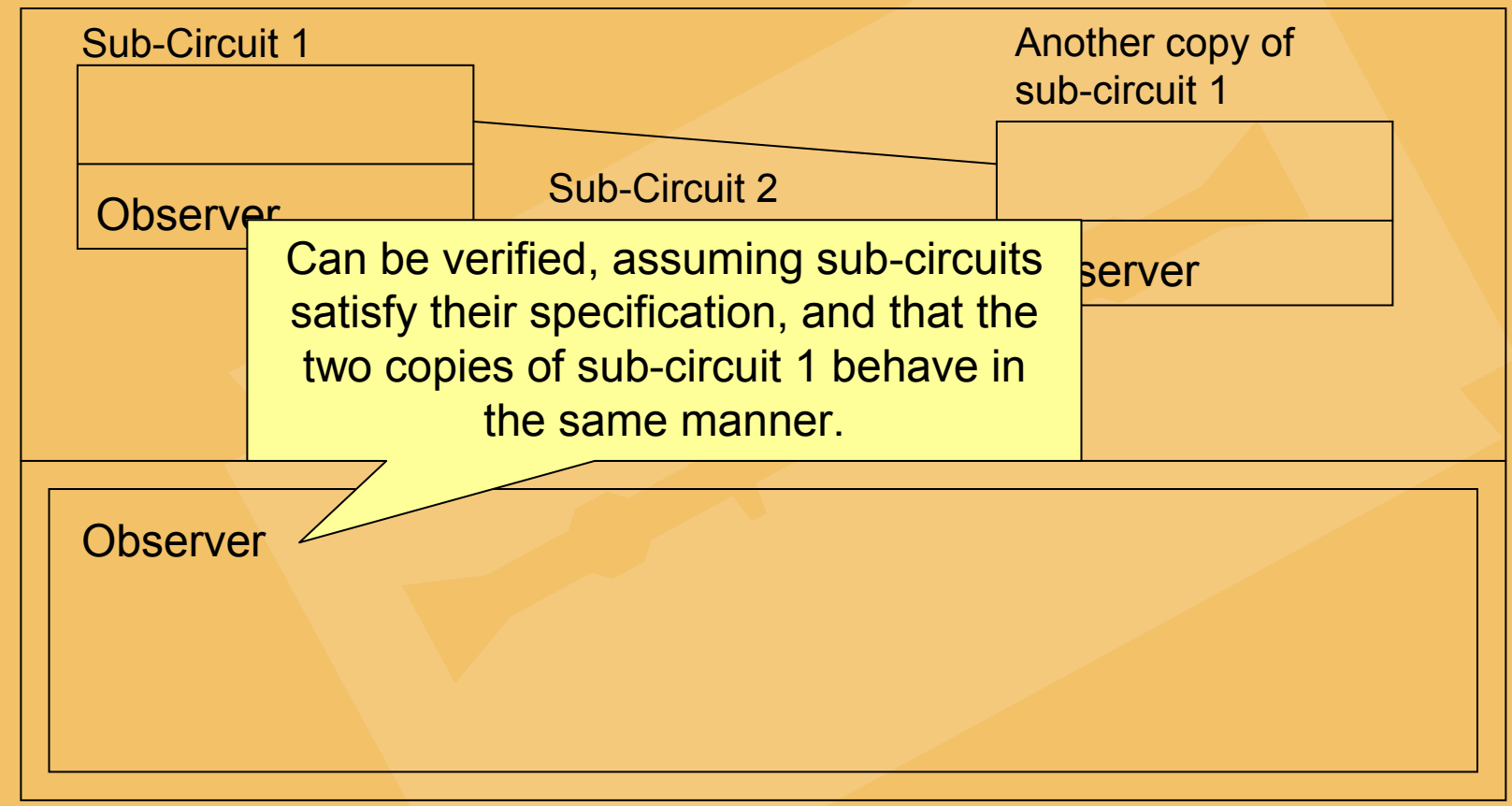
Specifications

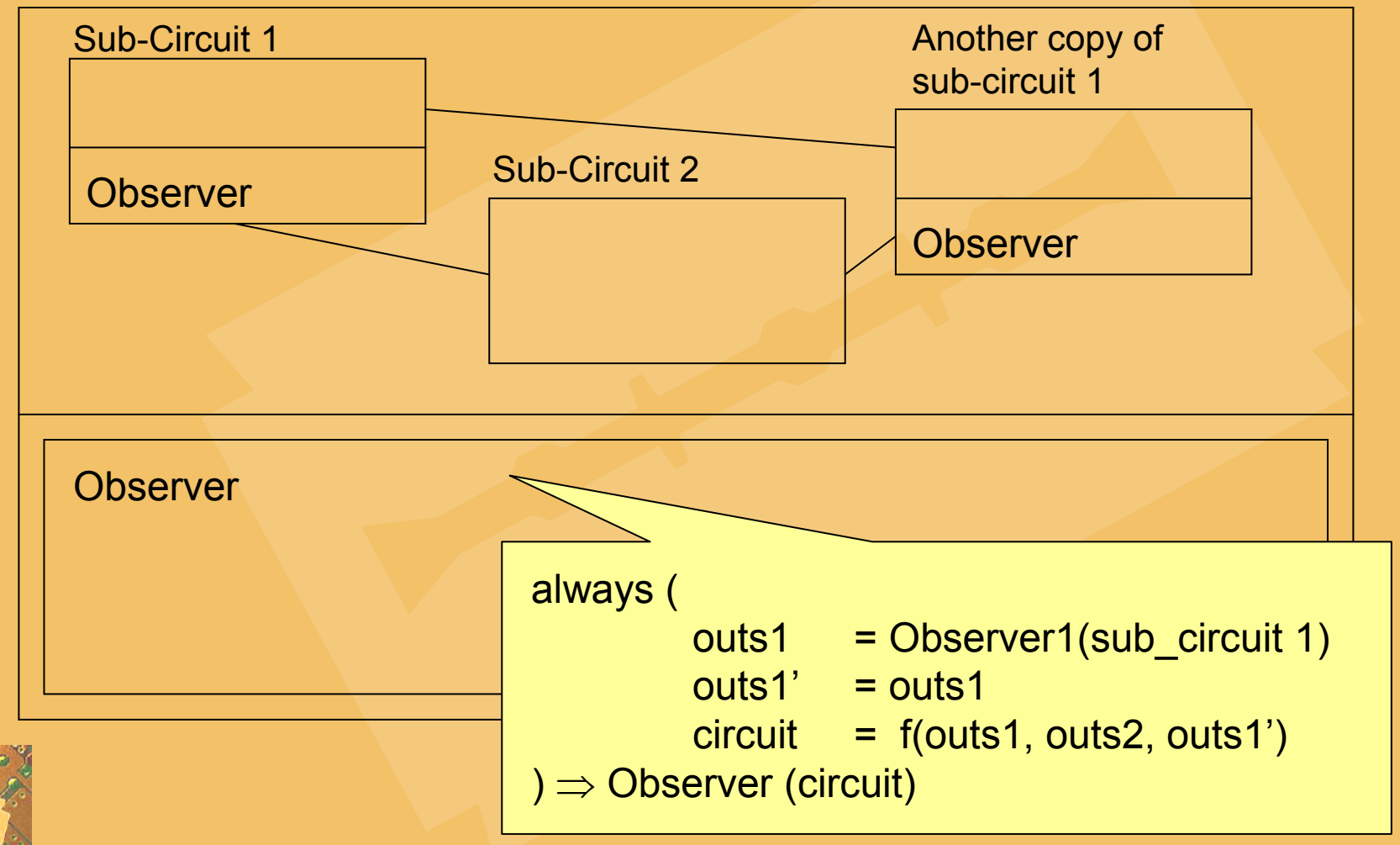
- Model-checking systems not only returns success/failure/etc, but also internally stores what has been proved.
- Refinement of sub-circuits will open up new proof-obligations. However, reproof of a higher-level circuit may (sometimes) be avoided thanks to compositionality and monotonicity of the refinement relation.
- Since we assume that the description will be compiled into a deterministic synchronous circuit we safely assume that common blocks have the same behaviour all over the circuit (for different refinements, the user would be creating two instances of the object).

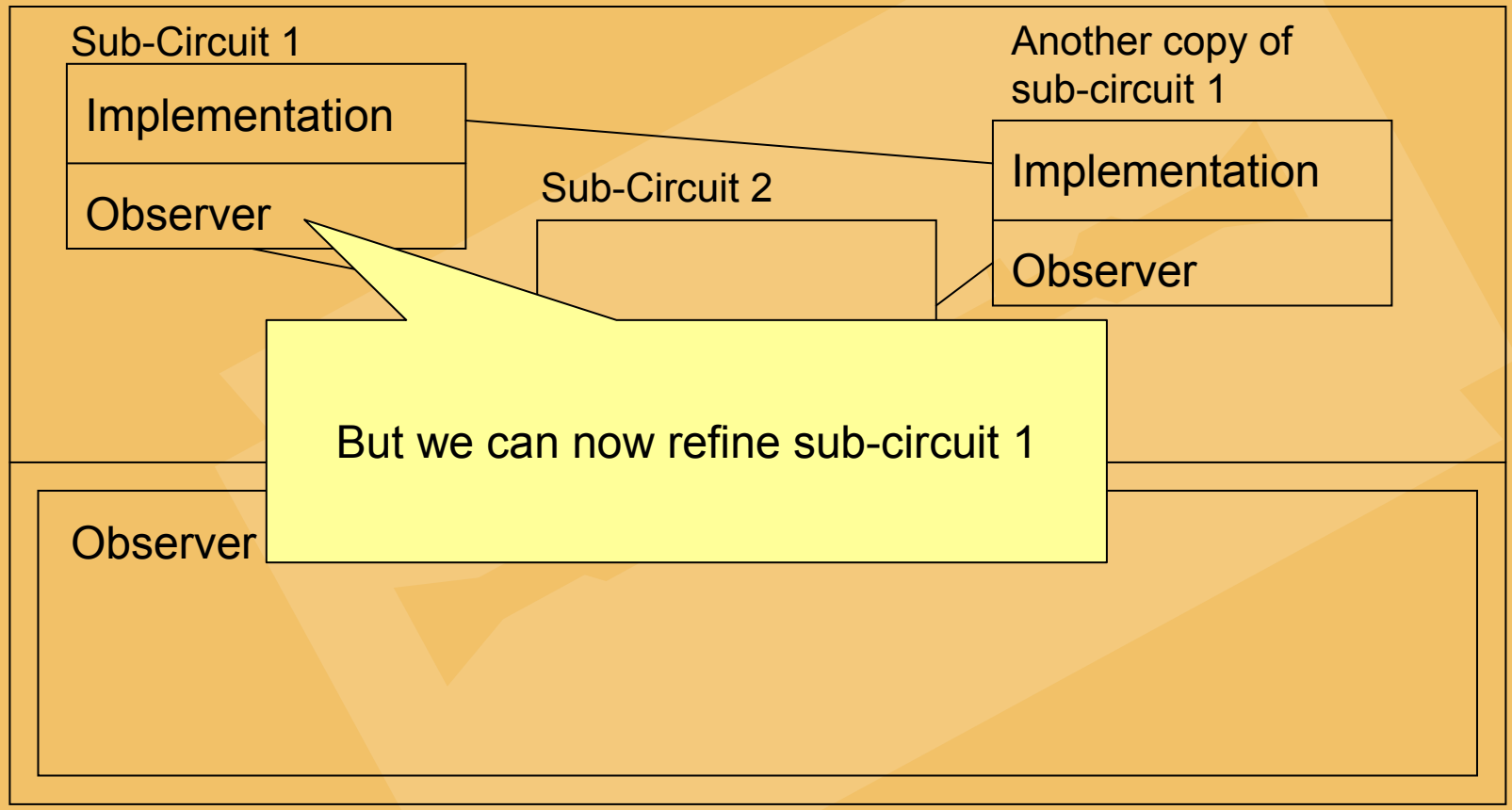

Sub-Circuit 1
Observer
Sub-Circuit 2
Whatever

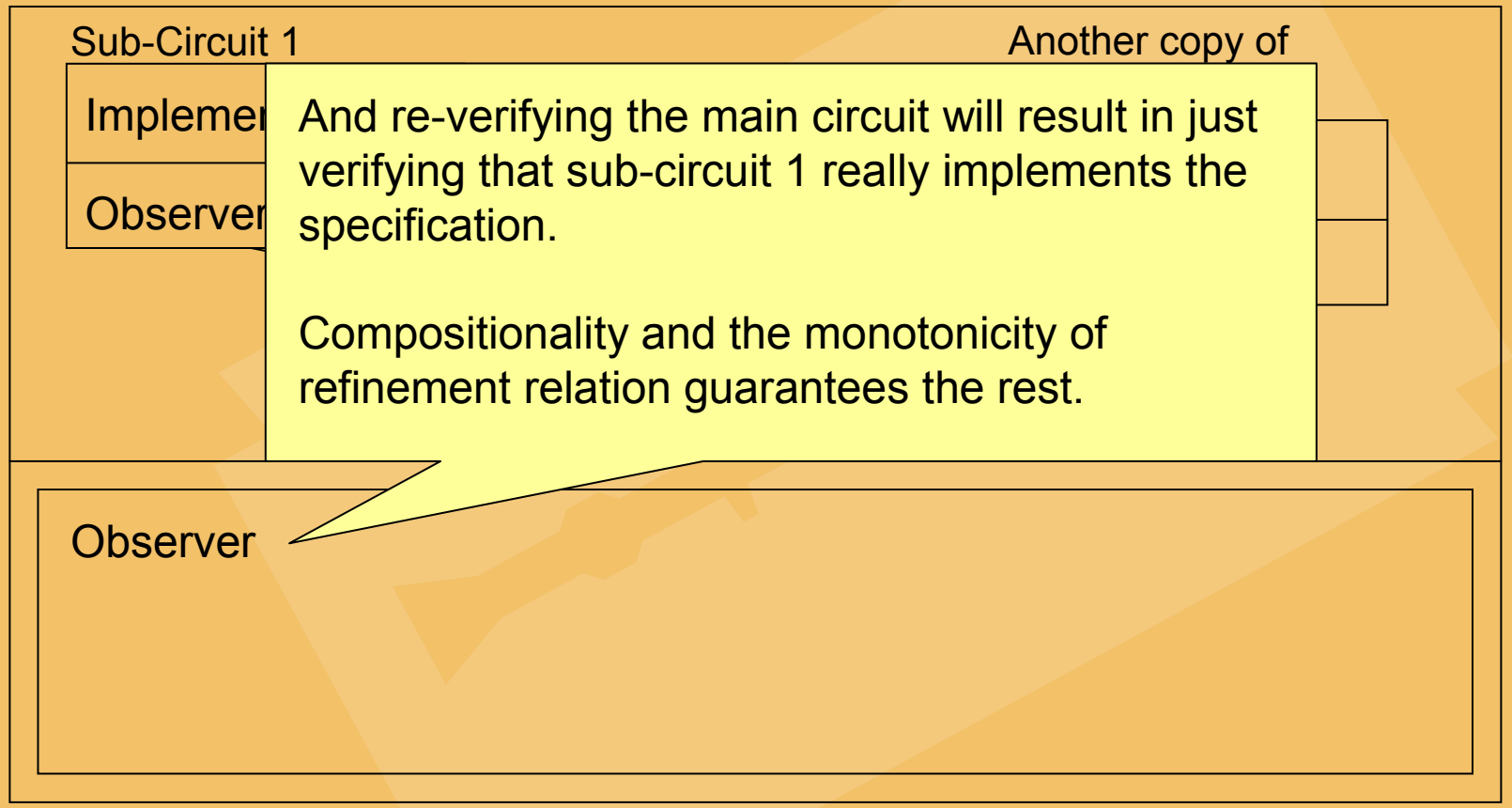

Sub-Circuit 1
Observer
Another copy of
sub-circuit 1
Sub-Circuit 2
Observer


Sub-Circuit 1
Observer
Another copy of
sub-circuit 1
Sub-Circuit 2
Observer
Observer


Sub-Circuit 1
Another copy of sub-circuit 1
Sub-Circuit 2
Observer
server
Can be verified, assuming sub-circuits satisfy their specification, and that the two copies of sub-circuit 1 behave in the same manner.
Observer


Sub-Circuit 1
Observer
Another copy of
sub-circuit 1
Sub-Circuit 2
Observer
Observer
always (
outs1 = Observer1(sub_circuit 1)
outs1' = outs1
circuit = f(outs1, outs2, outs1')
) ⇒ Observer (circuit)


Sub-Circuit 1
Implementation
Observer
Another copy of sub-circuit 1
Implementation
Observer
Sub-Circuit 2
But we can now refine sub-circuit 1
Observer

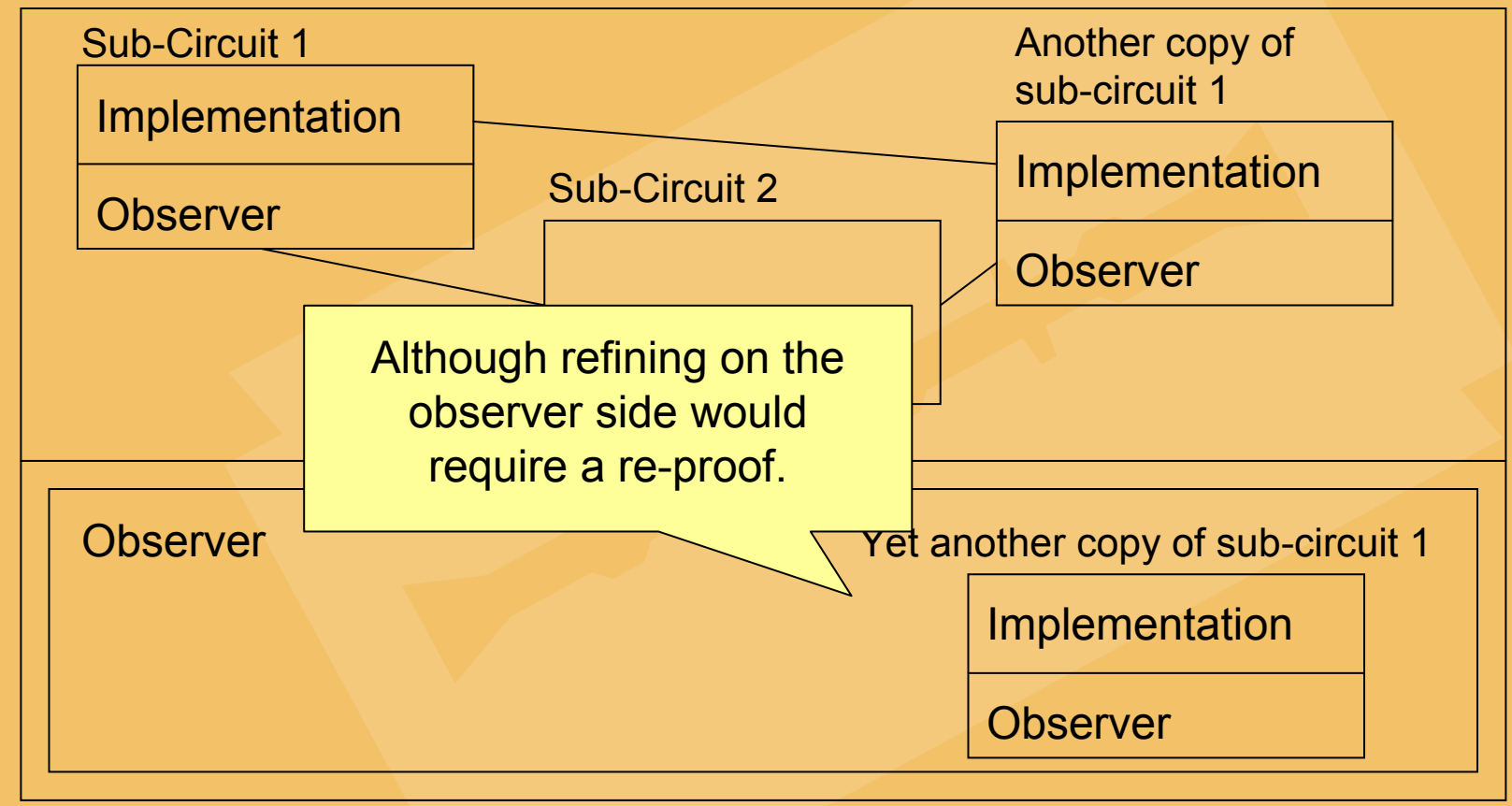

Sub-Circuit 1
Another copy of
Implemer
Observer
And re-verifying the main circuit will result in just verifying that sub-circuit 1 really implements the specification.
Compositionality and the monotonicity of refinement relation guarantees the rest.
Observer

The sort of thing we write

```
c_sub.property(sub_observer);

c.circuit(...,c_sub,...);
c.property(observer);

if (c.verify()) then {
  c_sub.refine(sub_observer');
  c.verify();
  ...
} else {
  ...
}
```


Sub-Circuit 1
Implementation
Observer
Another copy of sub-circuit 1
Implementation
Observer
Sub-Circuit 2
Although refining on the observer side would require a re-proof.
Observer
Yet another copy of sub-circuit 1
Implementation
Observer

Where Are We Going?

- Build a full language for refinement and interaction with model-checking tools.
- We're still looking for a cool case-study to show the use of this library on an interesting situation.
- As already noted, refining stuff on the observer side will require reproof. Should we allow refinement on the observer?

Longer Term Goals & Extensions

- Add more verification tools, and control over their use (eg timeouts, memory usage, etc) to be able to interact with via this same embedded language (a la Koen Claessen, Per Bjesse, Prover);
- Abstraction refinement via the embedded language;
- What about looking at interaction with testing instead of verification?