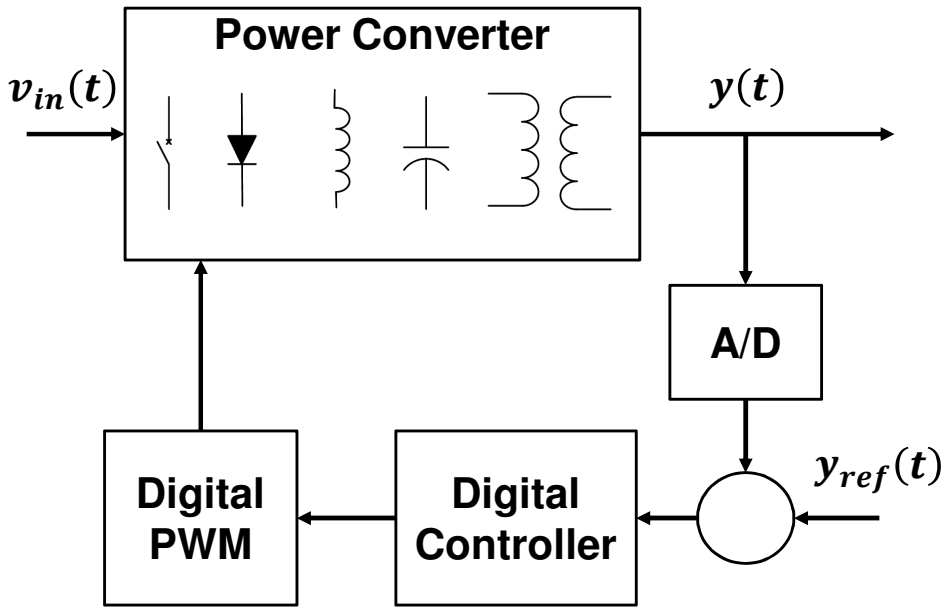
22nd International SPIN Symposium on Model Checking of Software

DSVerifier: A Bounded Model Checking Tool for Digital Systems

Hussama I. Ismail, **Iury V. Bessa**, Lucas C. Cordeiro
Eddie B. de Lima Filho, and João E. Chaves Filho

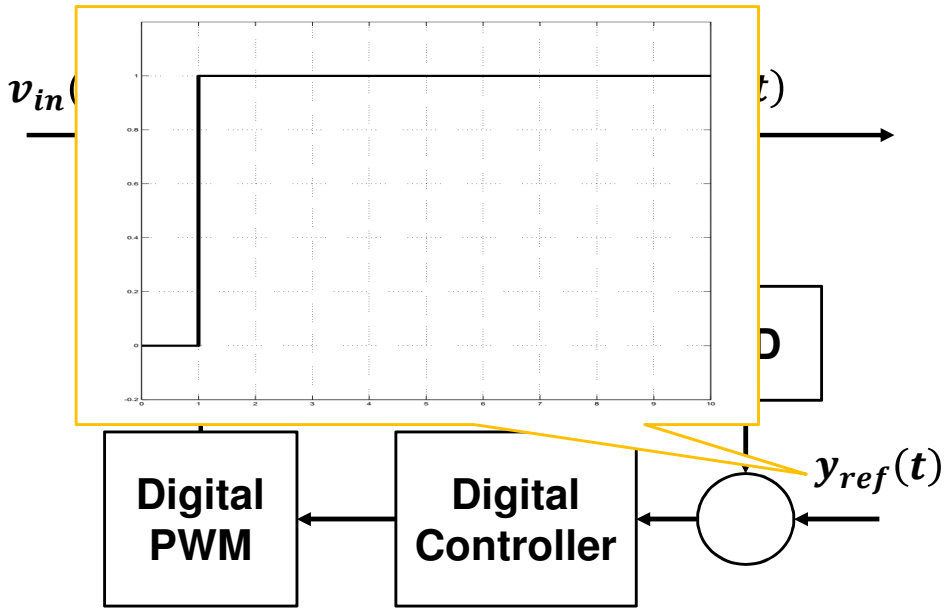
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



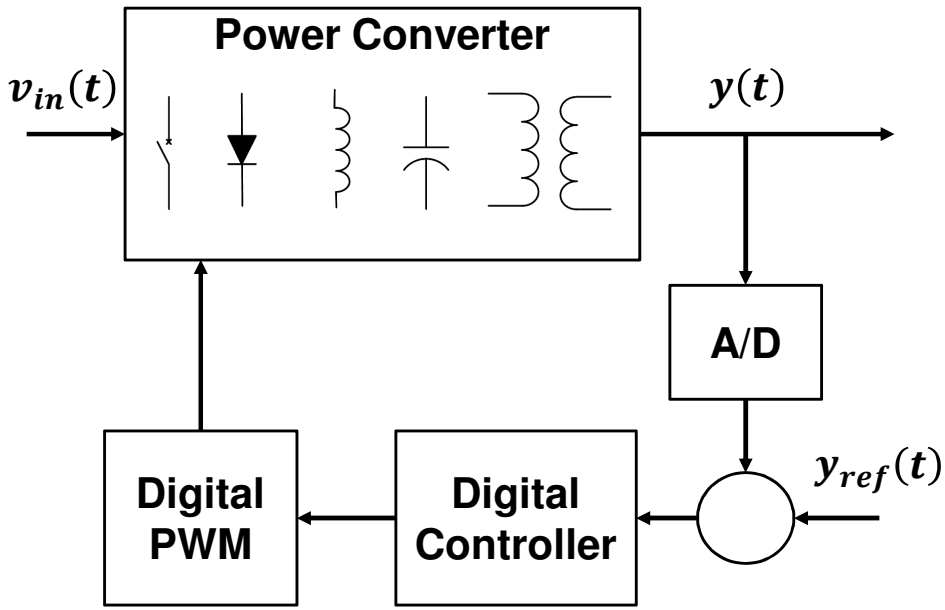
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



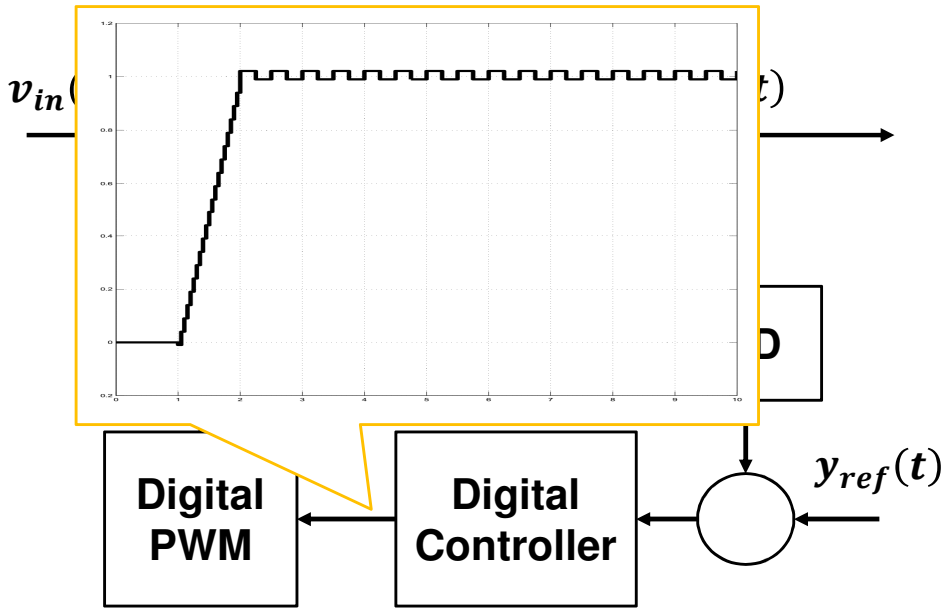
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



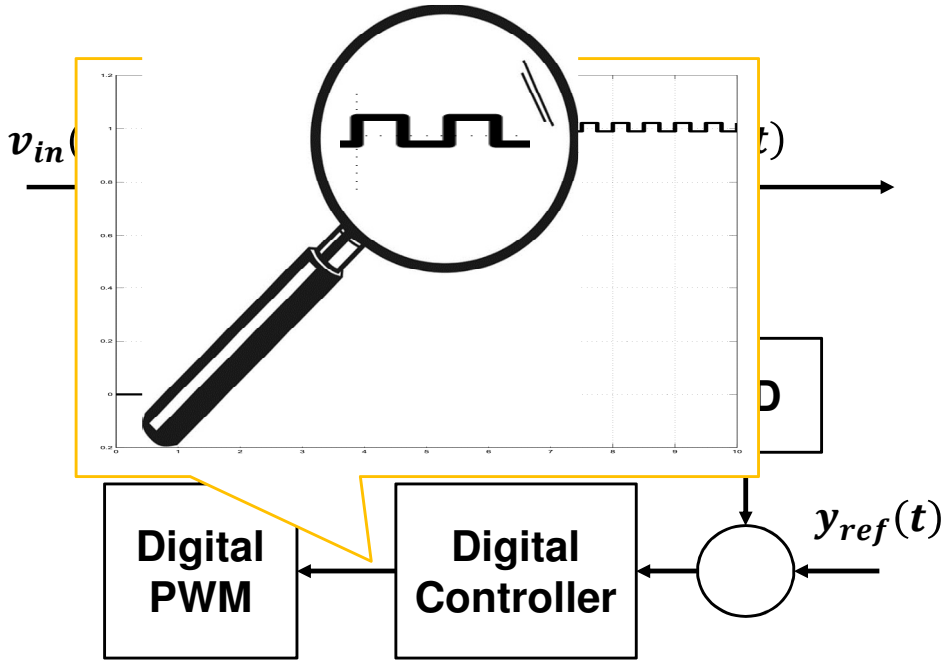
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



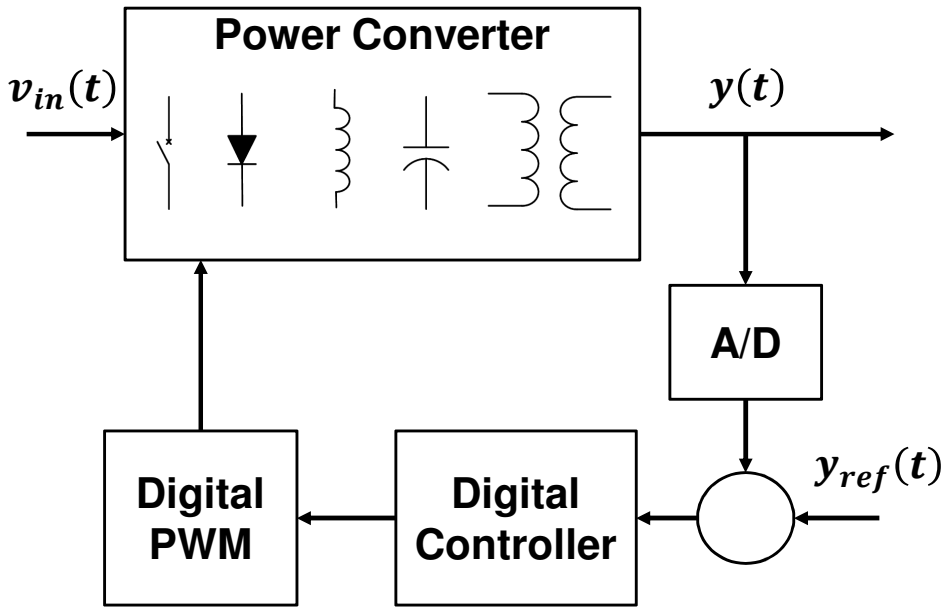
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



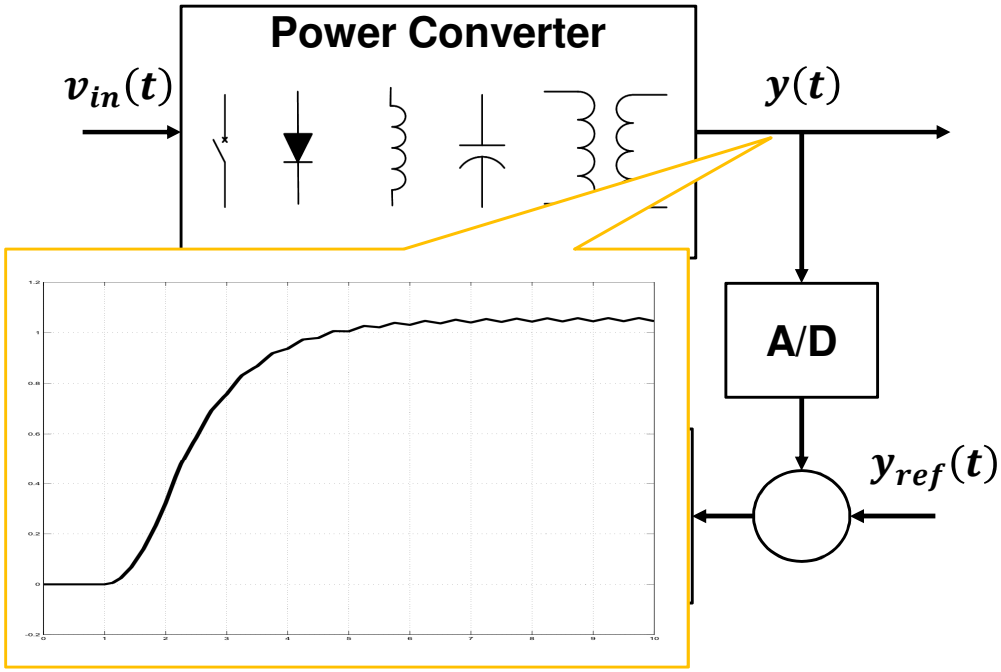
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



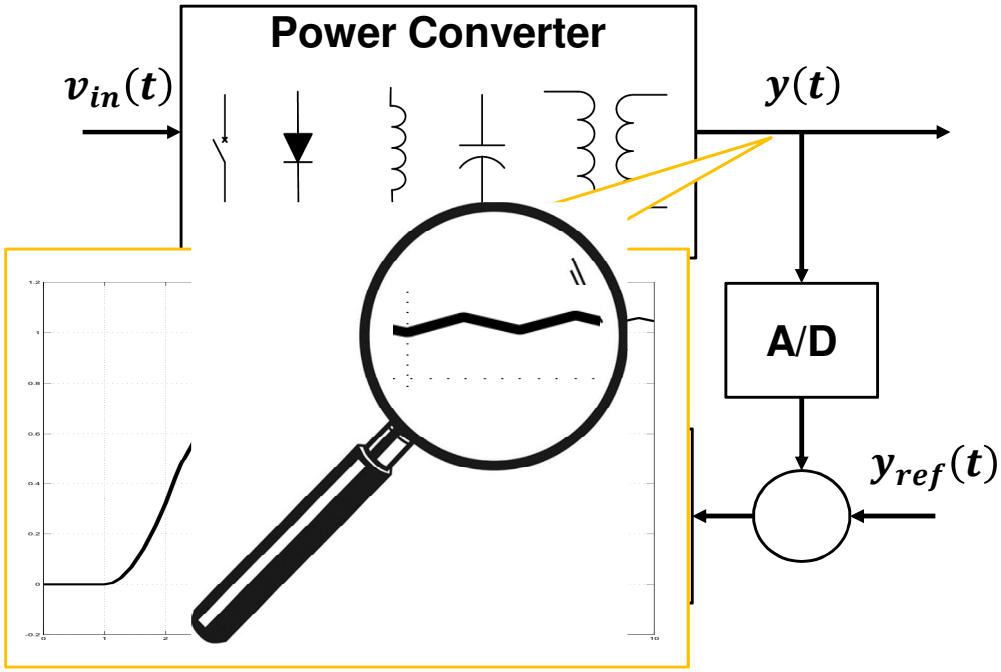
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



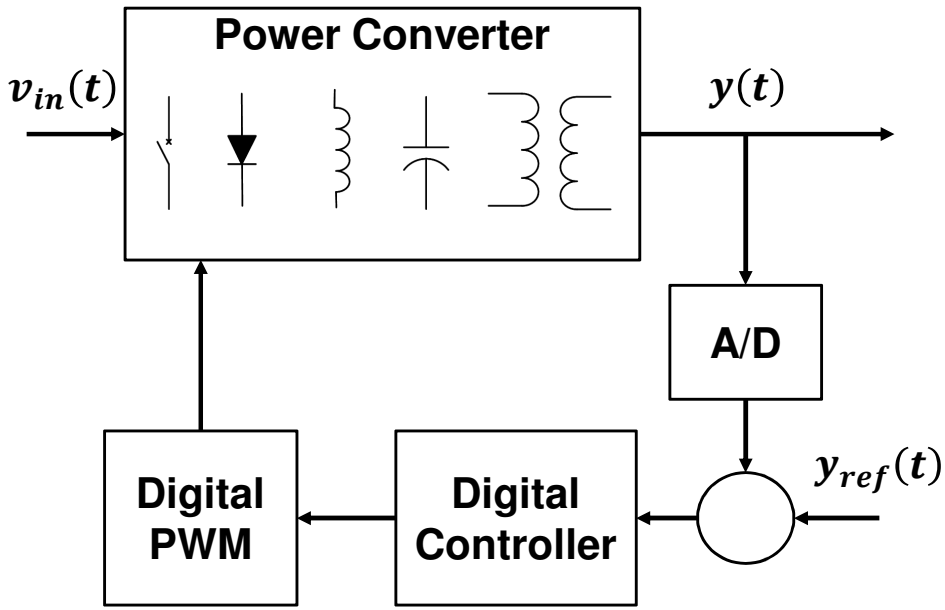
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



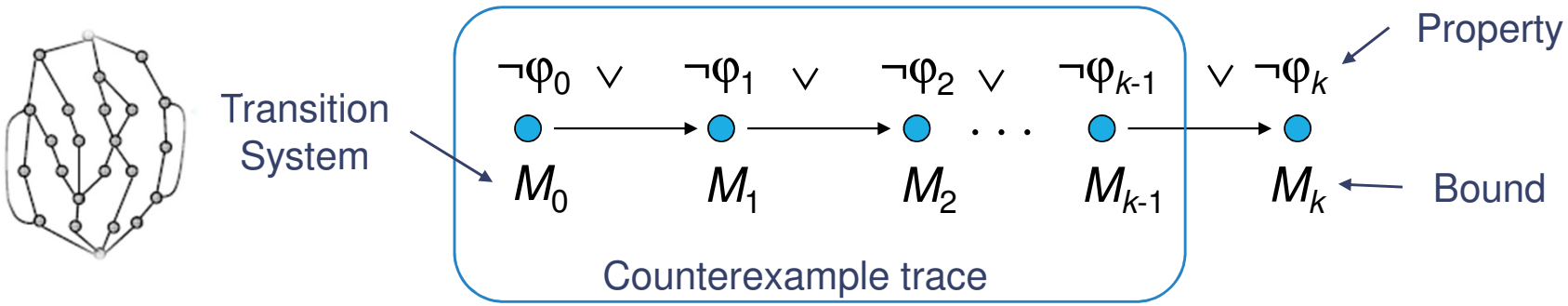
Digital Systems Applications and Limitations

- Digital filters and controllers are currently replacing many analog components
- Despite several advantages, they present limitations related to **finite-word length** (FWL) effects
- Limit cycle oscillations (LCOs) in power converters:
  - Oscillation in output voltage due to round-off and overflows
  - More **energy losses** and short **silicon lifespan**
  - LCOs are almost unavoidable and difficult to be detected
  - LCOs are typically detected via time-domain simulations



Bounded Model Checking (BMC)

- Basic Idea: given a transition system *M*, check negation of a given property *φ* up to given depth *k*



- Translated into a VC *ψ* such that: ***ψ* is satisfiable iff *φ* has counterexample of max. depth *k***
- BMC has been applied successfully to verify (embedded) software since early 2000's, but it has not been used to verify digital controllers

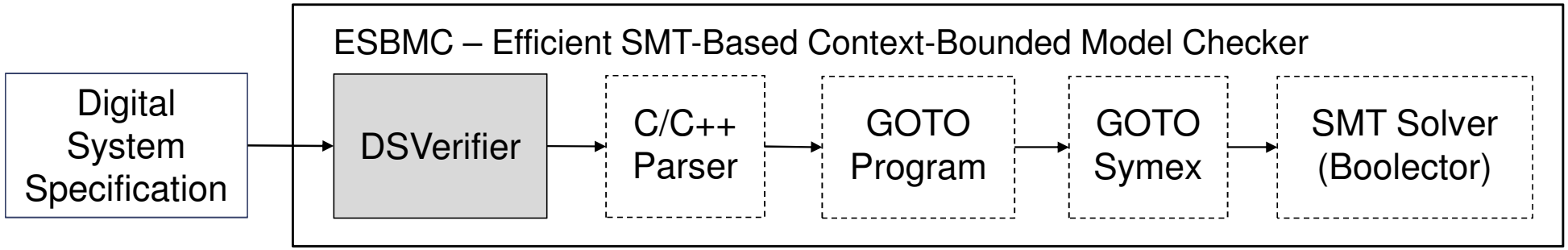
Objectives

BMC of digital systems implementations considering FWL effects

- Investigate FWL effects in fixed-point digital system (controllers and filters) implementations via BMC techniques
- Apply a design-aided verification methodology to digital systems, which is supported by the Digital-Systems Verifier (DSVerifier)
- Verify overflows, limit cycles, time constraints, stability, and mimimum phase in digital systems using standard benchmarks

The Digital-Systems Verifier (DSVerifier)

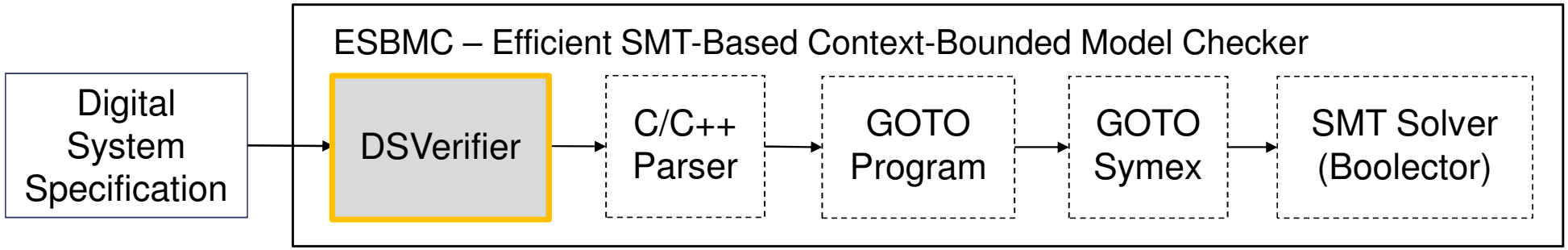
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



The complete tool includes four components from ESBMC
C Parser, GOTO Program, GOTO Symex, and SMT Solver

The Digital-Systems Verifier (DSVerifier)

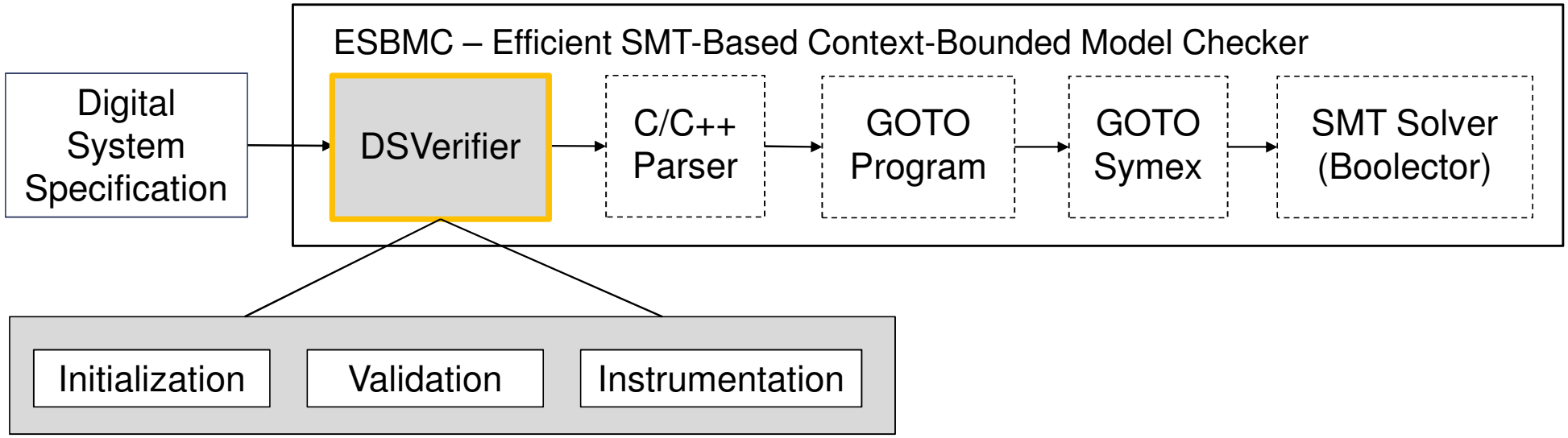
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



DSVerifier module is included before the ANSI-C parser, which provides functions related to quantization, digital-system realizations, and property verification

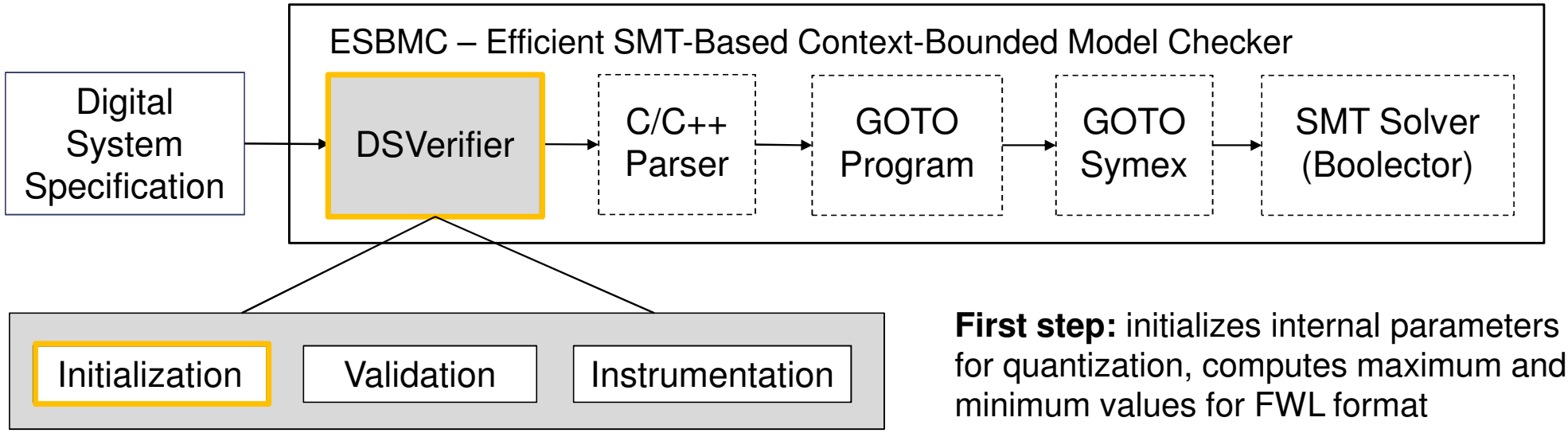
The Digital-Systems Verifier (DSVerifier)

- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



The Digital-Systems Verifier (DSVerifier)

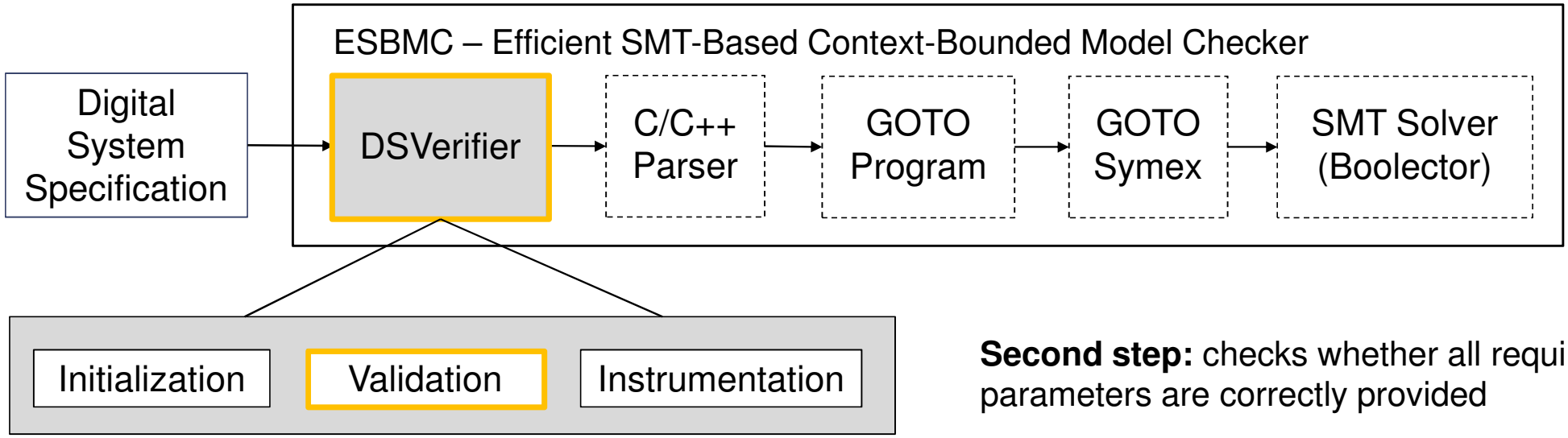
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



First step: initializes internal parameters for quantization, computes maximum and minimum values for FWL format

The Digital-Systems Verifier (DSVerifier)

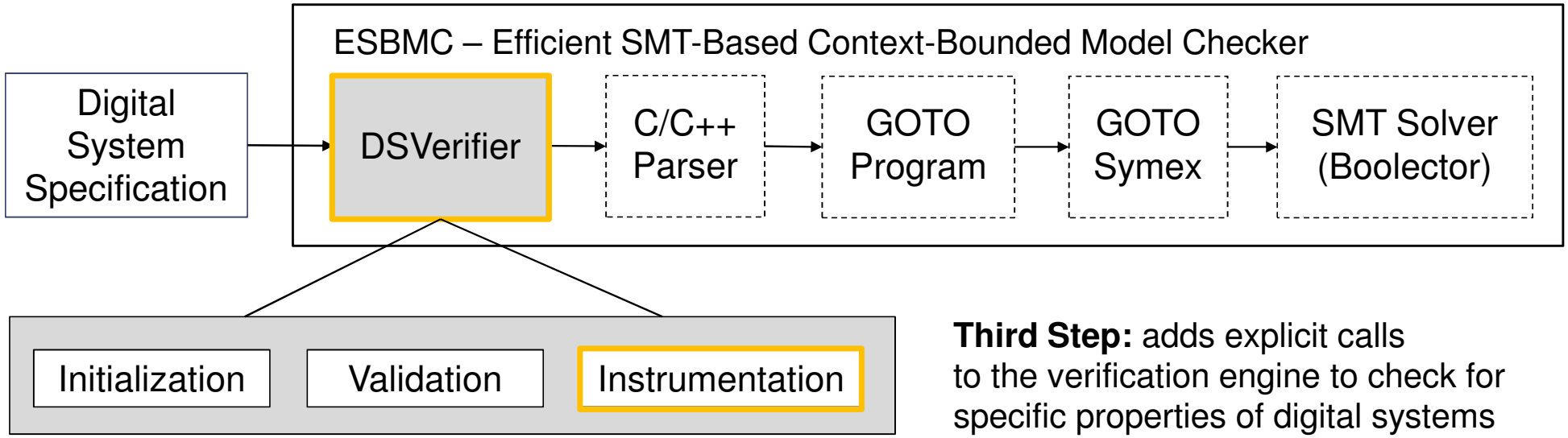
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



Second step: checks whether all required parameters are correctly provided

The Digital-Systems Verifier (DSVerifier)

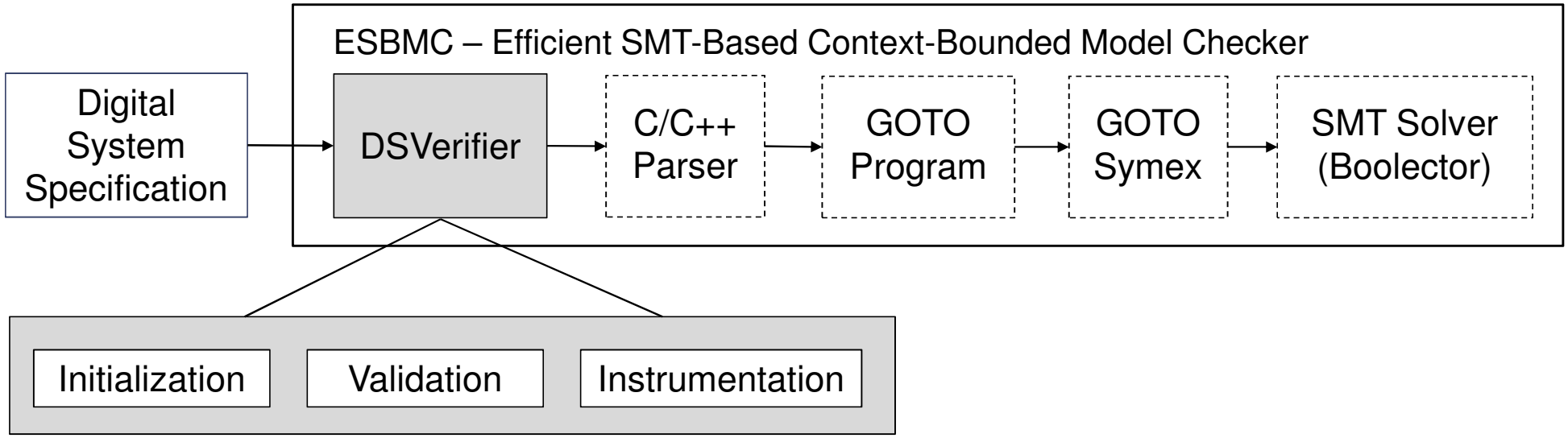
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



Third Step: adds explicit calls to the verification engine to check for specific properties of digital systems

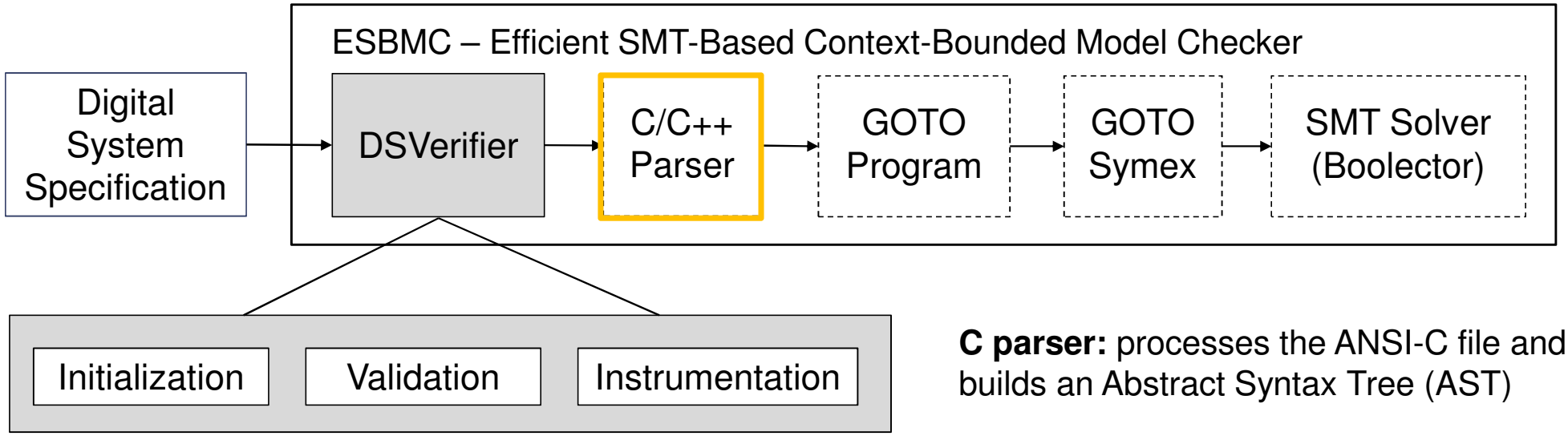
The Digital-Systems Verifier (DSVerifier)

- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



The Digital-Systems Verifier (DSVerifier)

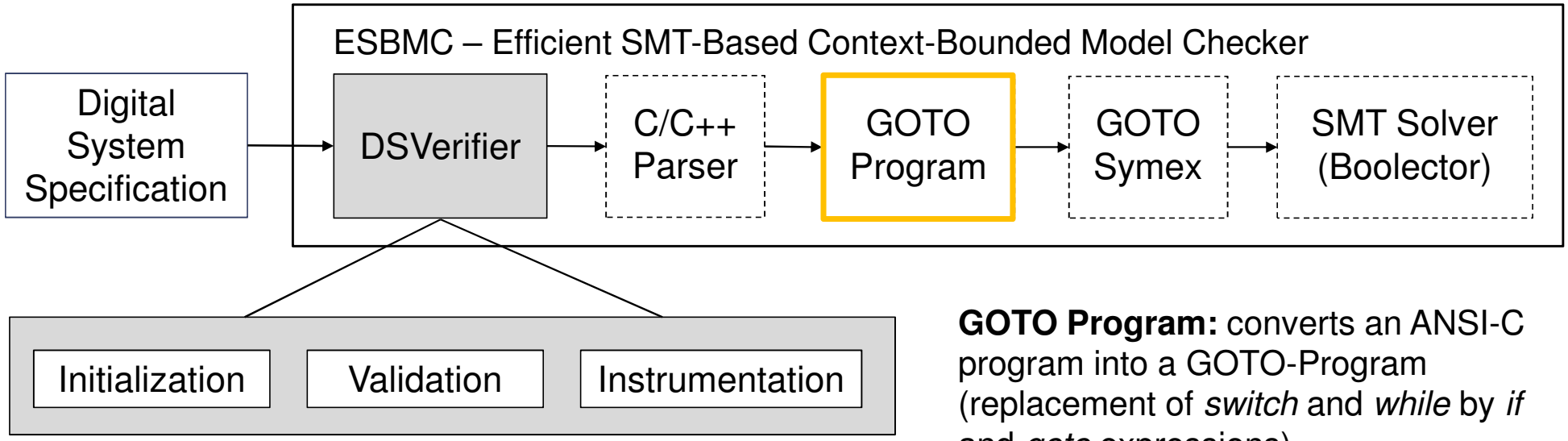
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



C parser: processes the ANSI-C file and builds an Abstract Syntax Tree (AST)

The Digital-Systems Verifier (DSVerifier)

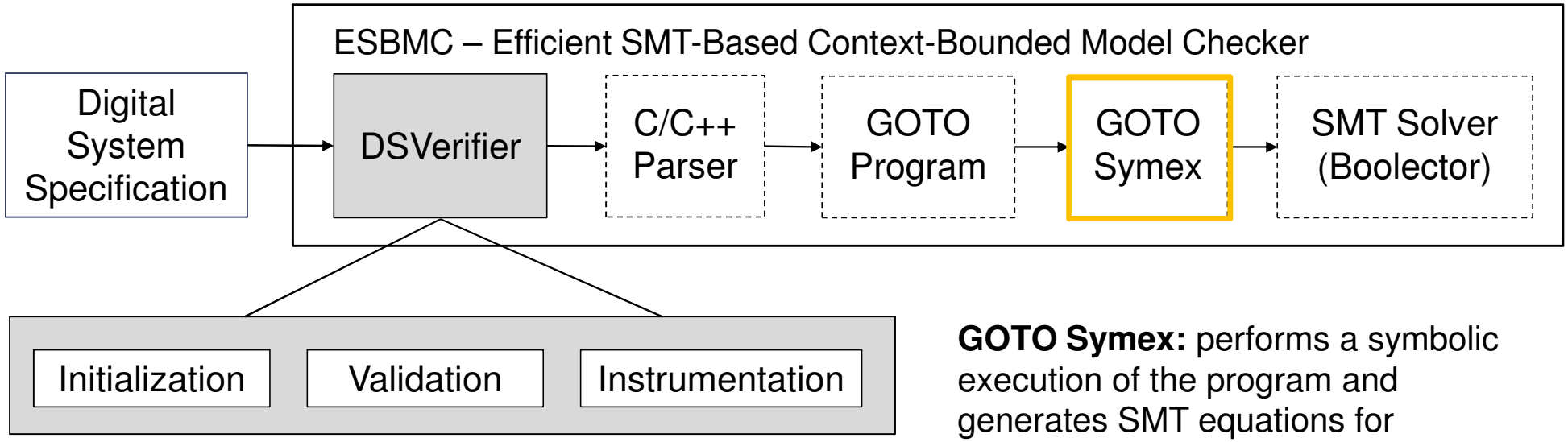
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



GOTO Program: converts an ANSI-C program into a GOTO-Program (replacement of *switch* and *while* by *if* and *goto* expressions)

The Digital-Systems Verifier (DSVerifier)

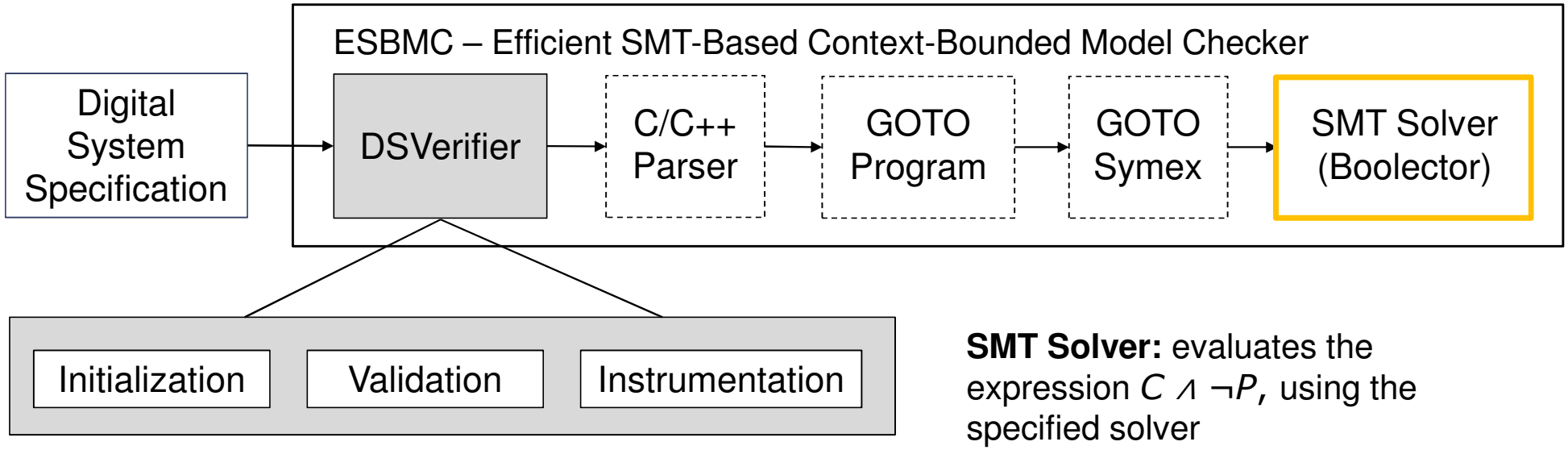
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



GOTO Symex: performs a symbolic execution of the program and generates SMT equations for constraints (*C*) and properties (*P*)

The Digital-Systems Verifier (DSVerifier)

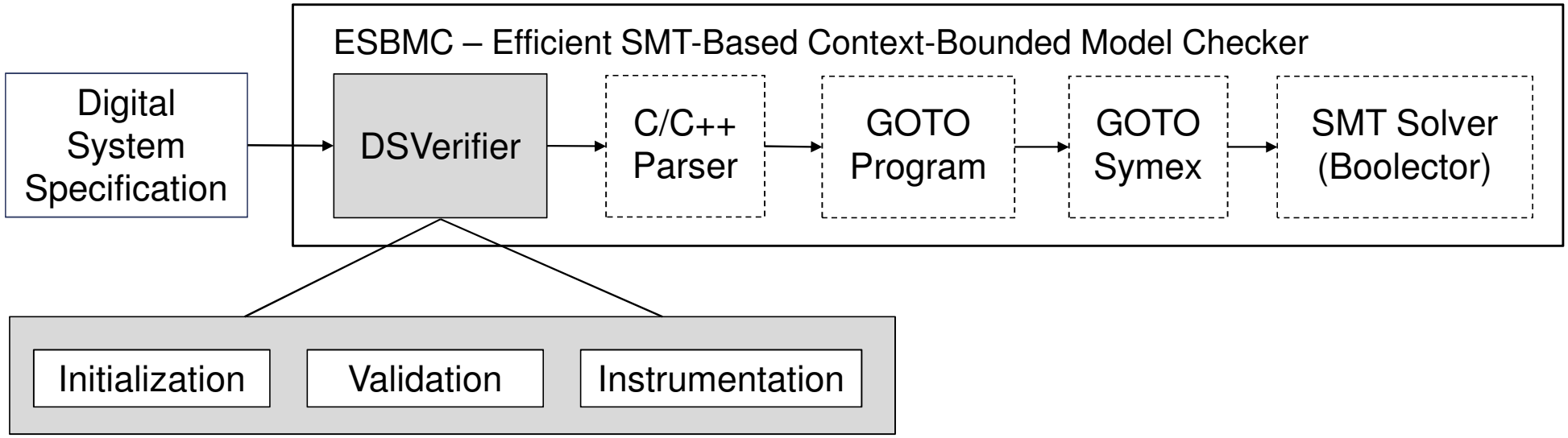
- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification



SMT Solver: evaluates the expression $C \wedge \neg P$, using the specified solver

The Digital-Systems Verifier (DSVerifier)

- DSVerifier is an additional module for the Efficient SMT-based Context-Bounded Model Checker (ESBMC) to add support for digital systems verification

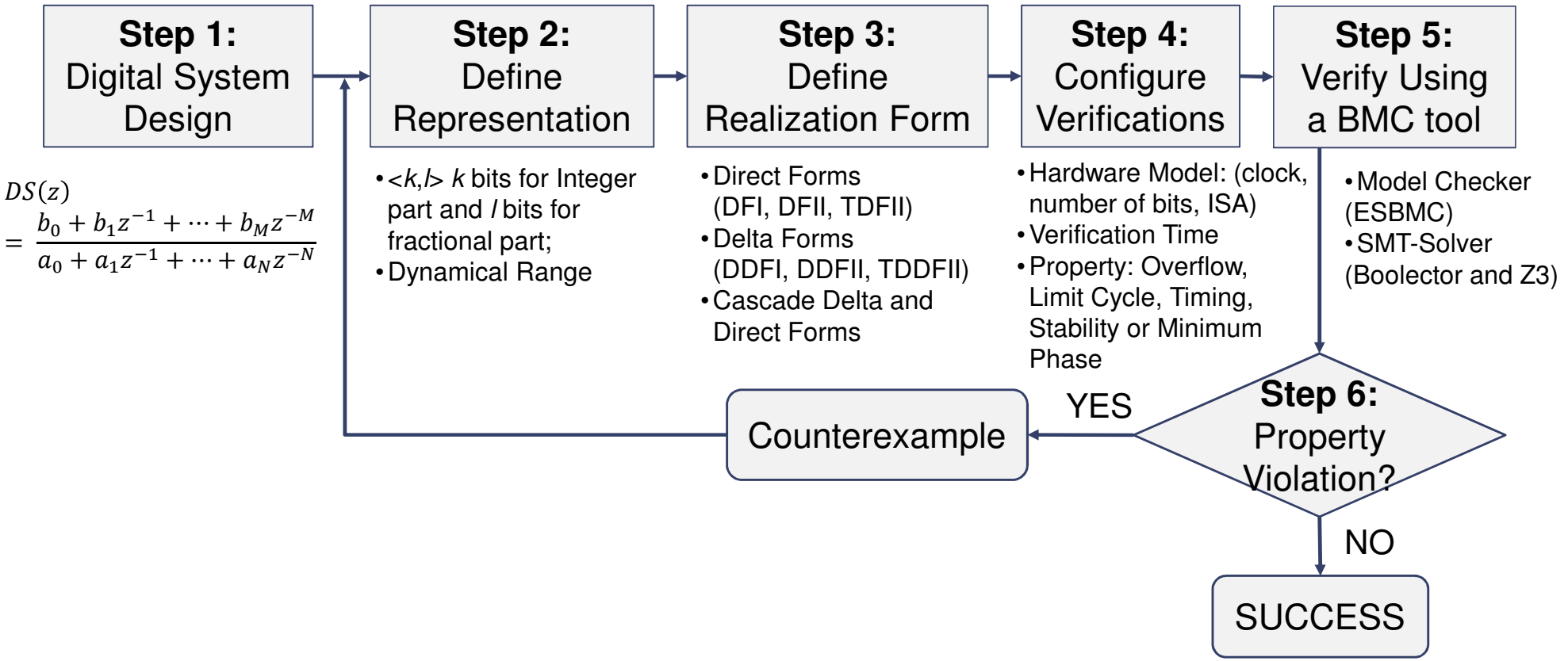


DSVerifier Features

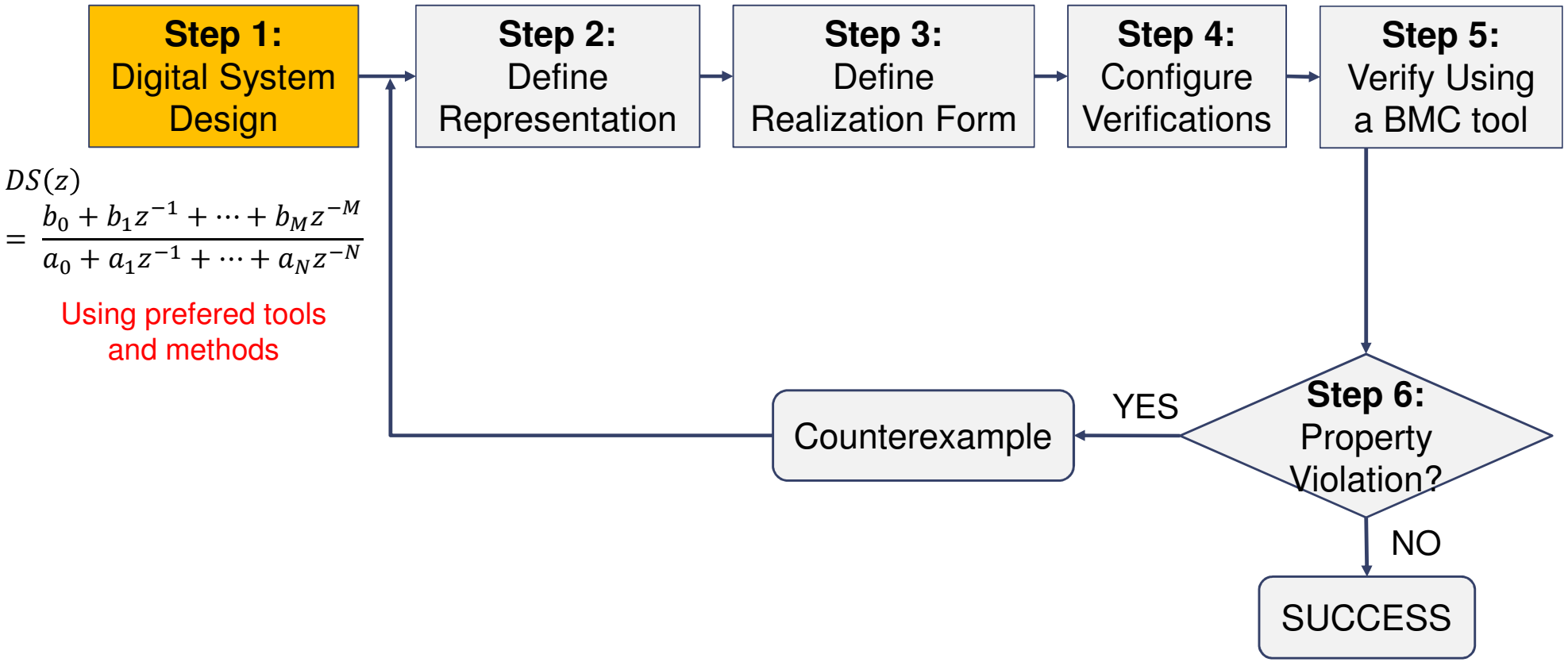
- DSVerifier supports five verification properties, considering three direct- and delta-form implementations, in addition to the cascade form

1. **Overflow:** if a sum or product exceeds the number representation
2. **Limit Cycle:** checks for zero-input limit cycles, for any initial condition
3. **Stability:** considers FWL effects on pole locations
4. **Minimum phase:** considers FWL effects on zero locations
5. **Time constraints:** checks whether a specific realization meets time constraints

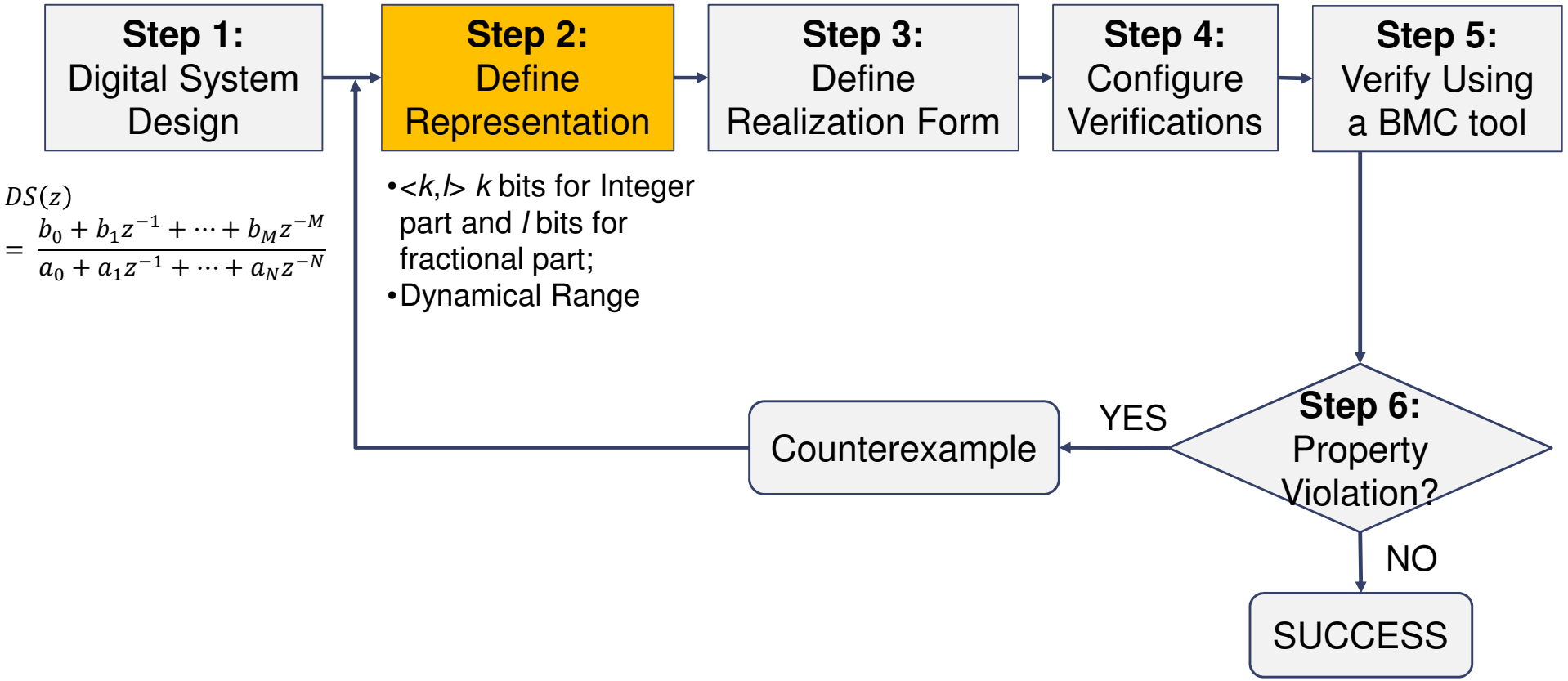
DSVerifier-Aided Verification Methodology



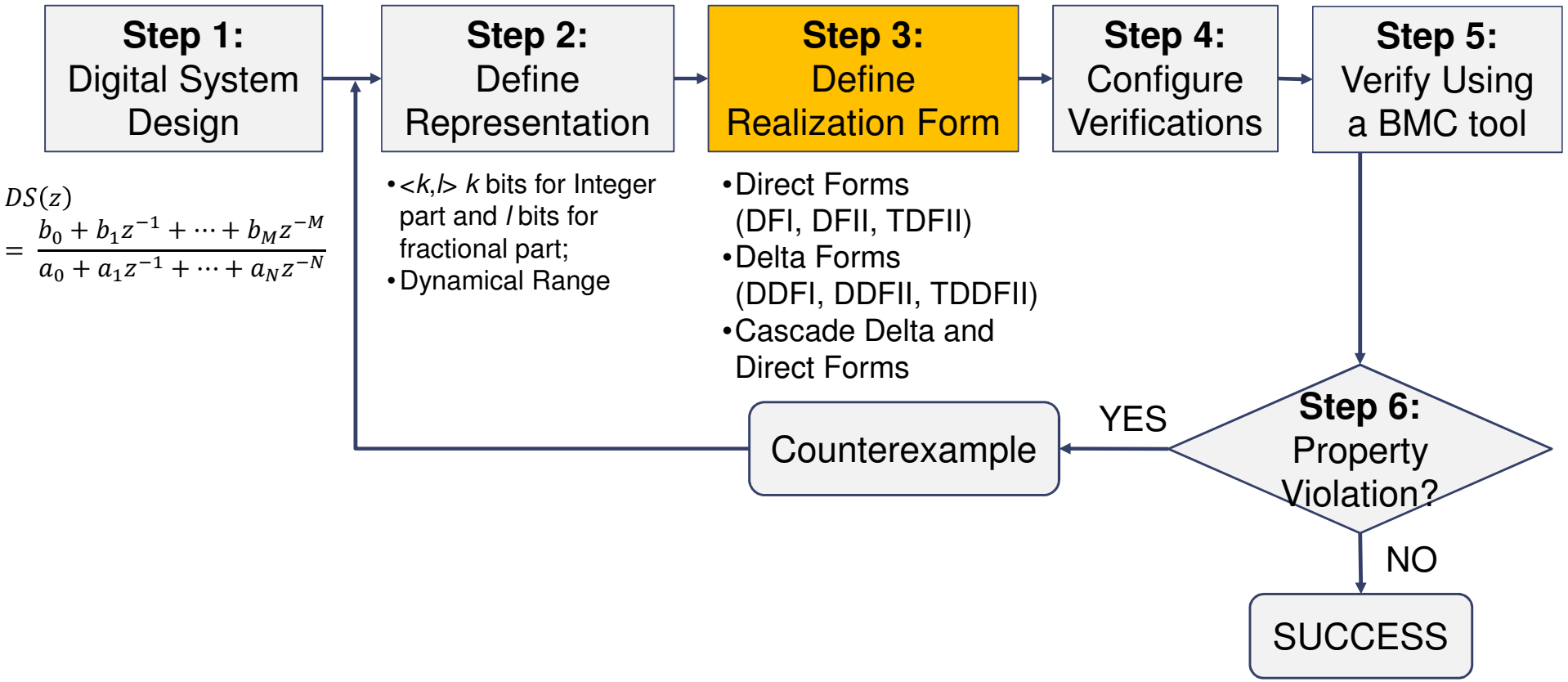
DSVerifier-Aided Verification Methodology



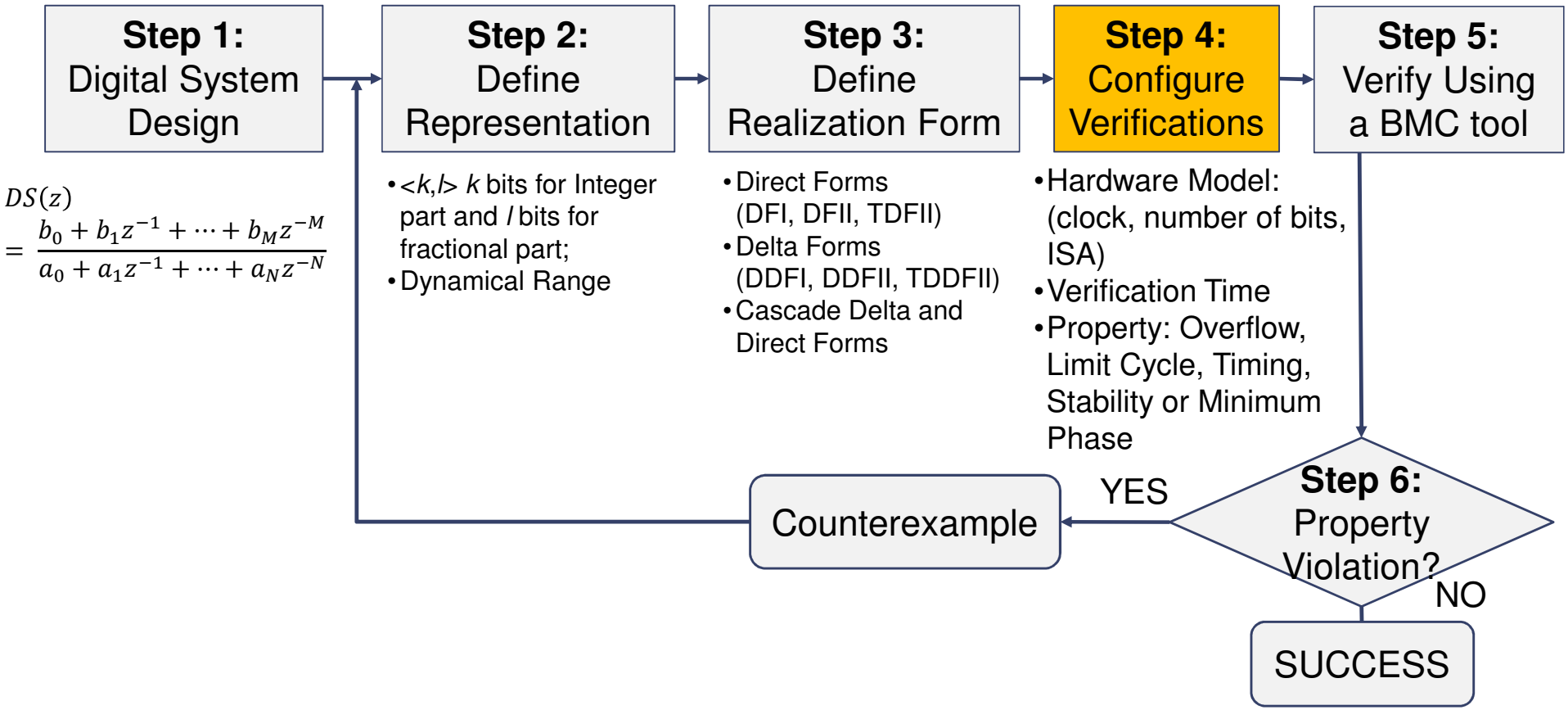
DSVerifier-Aided Verification Methodology



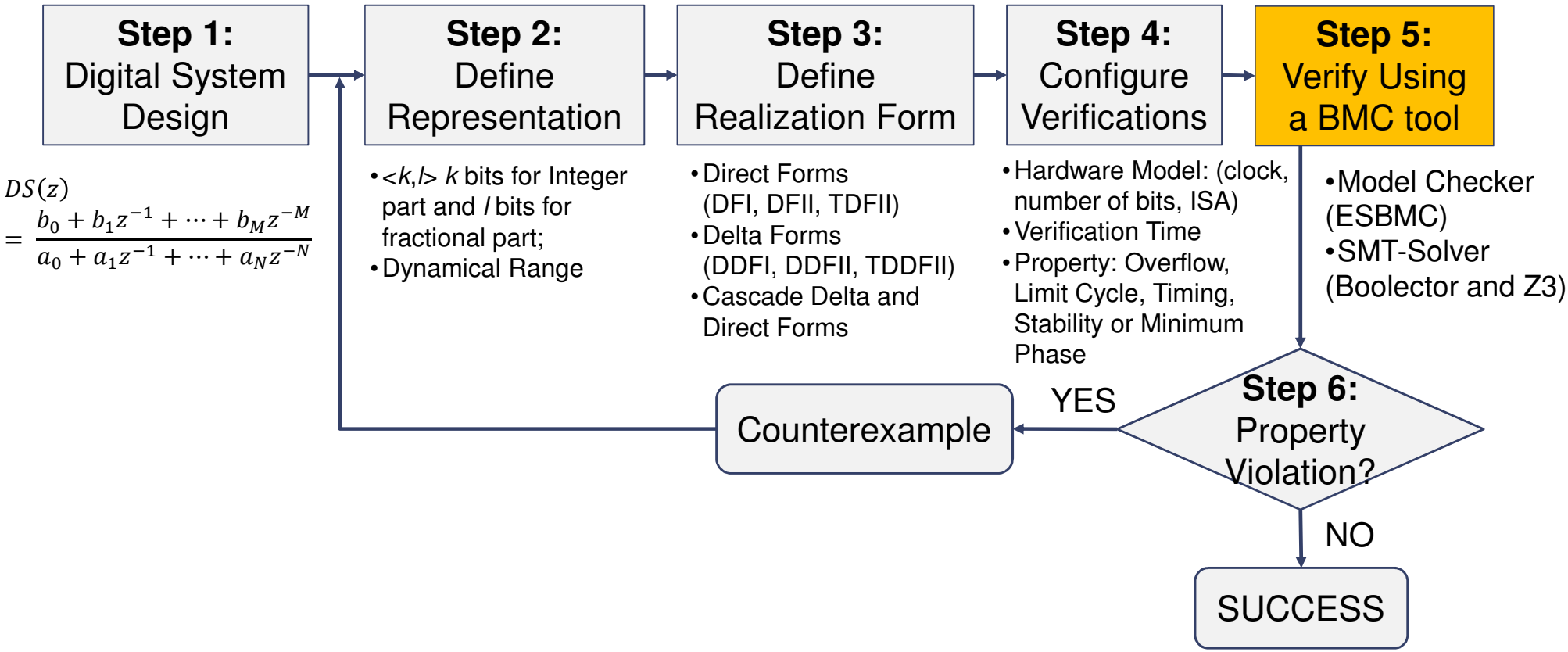
DSVerifier-Aided Verification Methodology



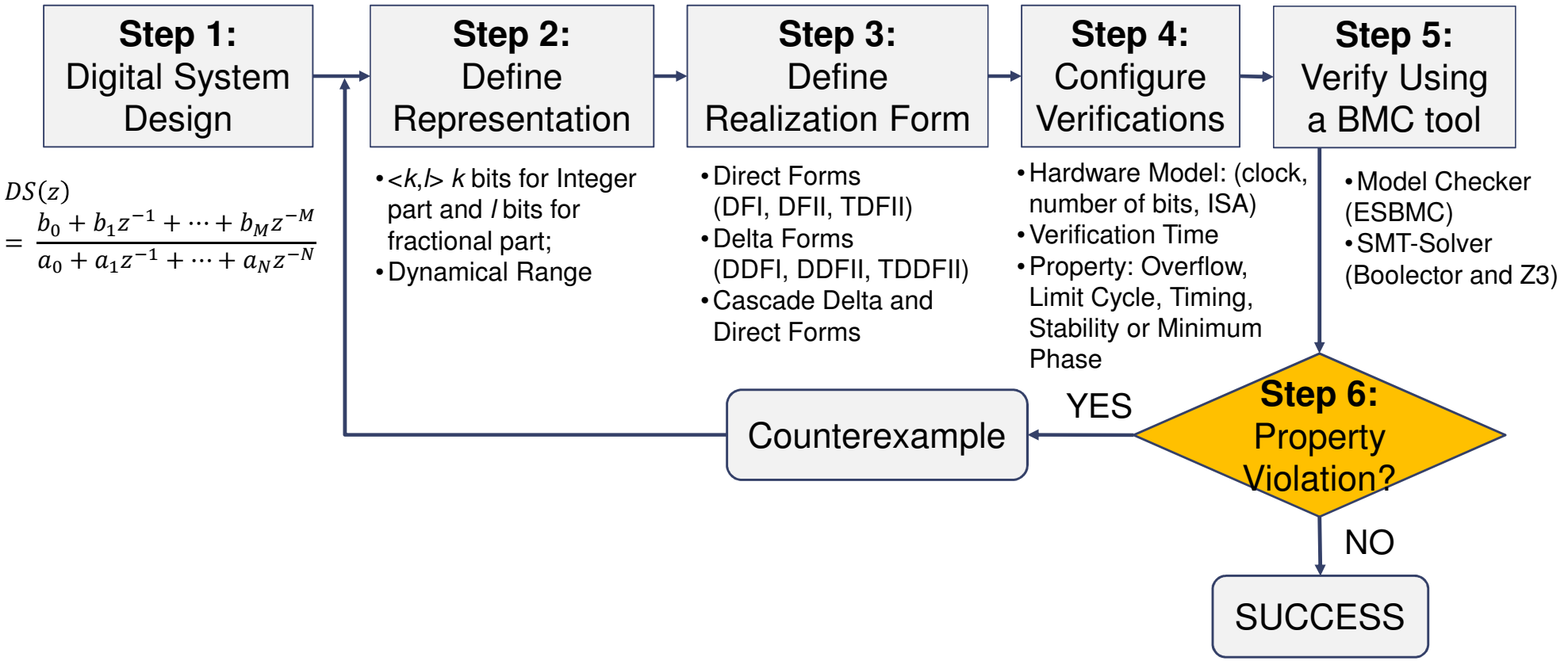
DSVerifier-Aided Verification Methodology



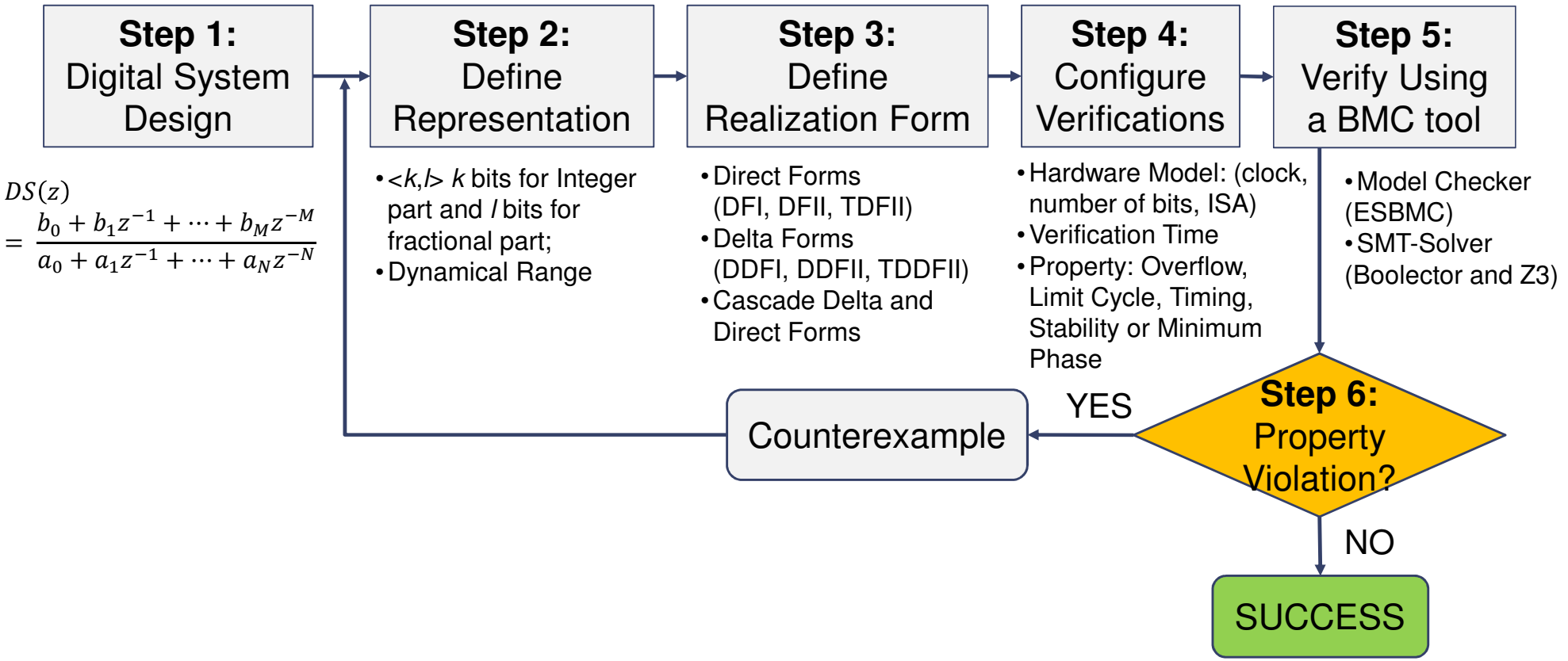
DSVerifier-Aided Verification Methodology



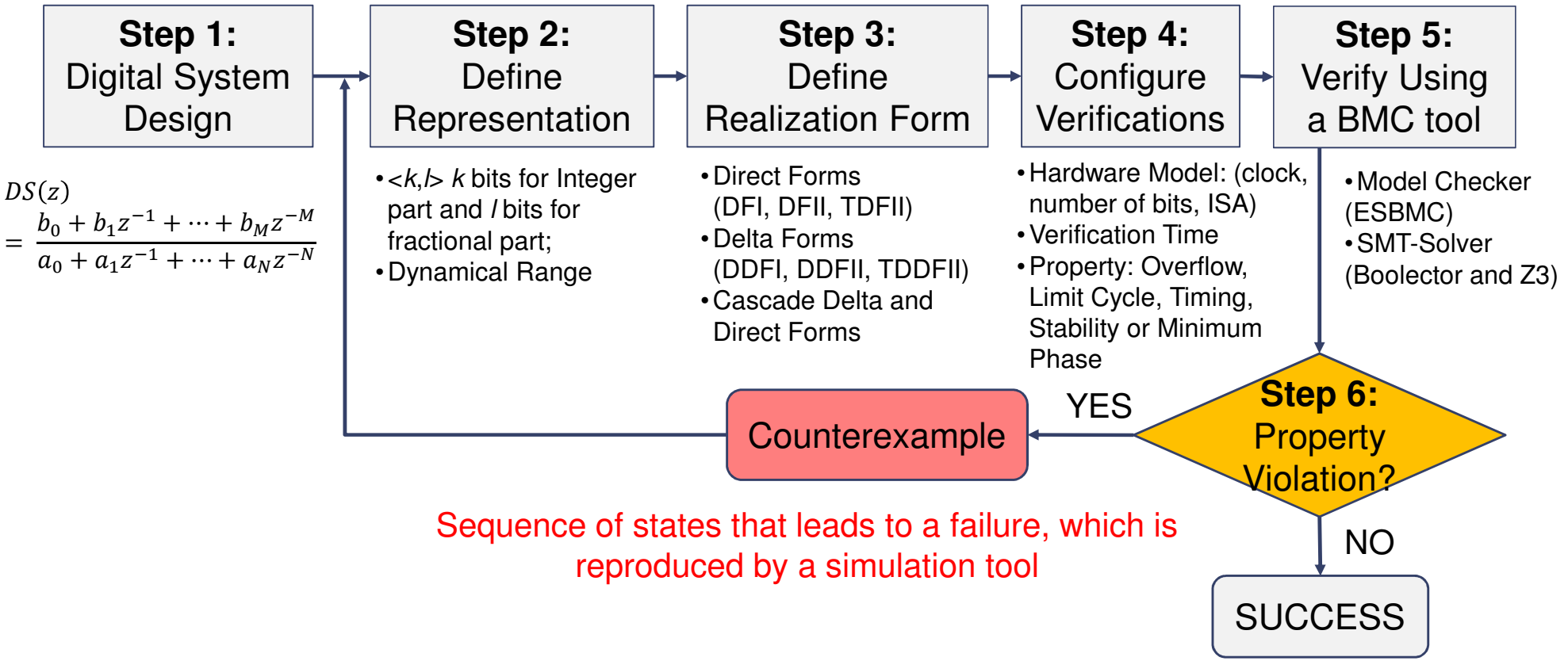
DSVerifier-Aided Verification Methodology



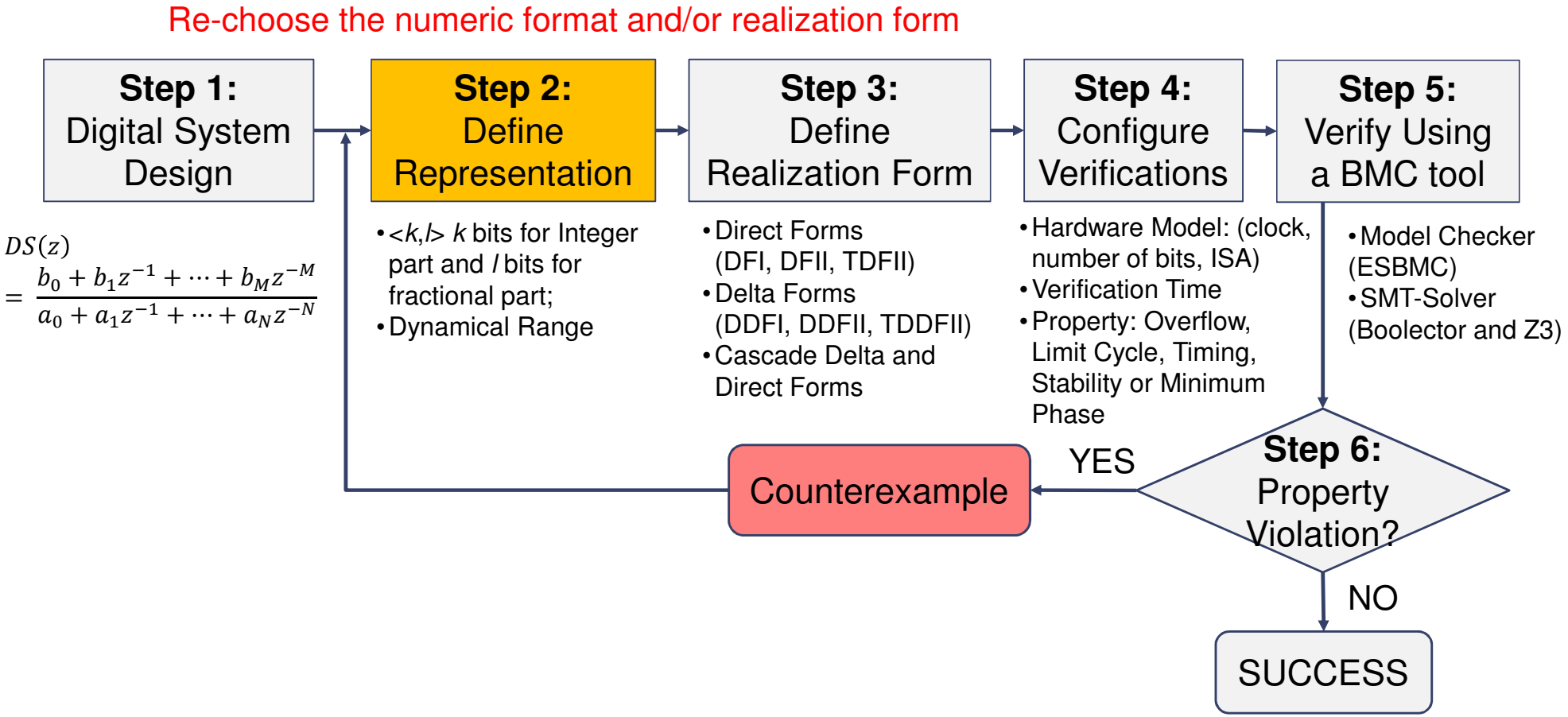
DSVerifier-Aided Verification Methodology



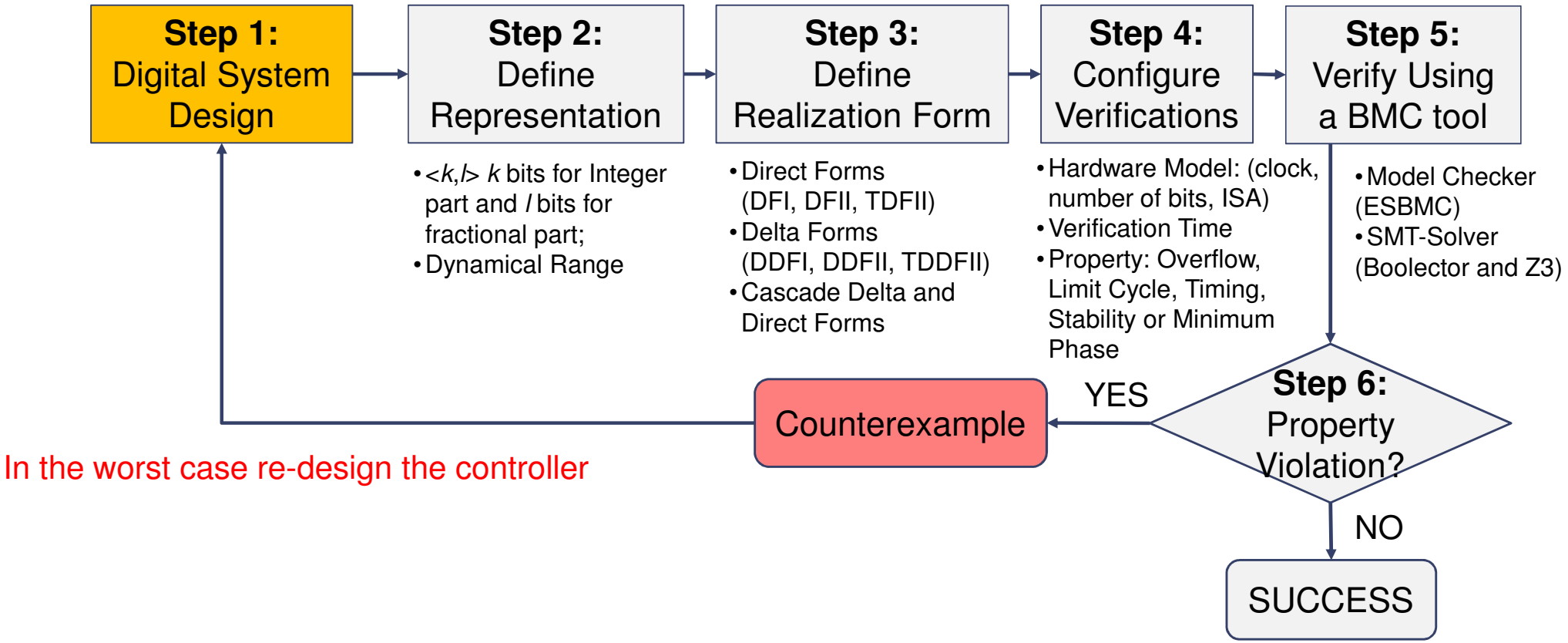
DSVerifier-Aided Verification Methodology



DSVerifier-Aided Verification Methodology



DSVerifier-Aided Verification Methodology



DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$
- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

Numeric format choosen based on impulse response sum and hardware limitations

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- DFII

Random first trial

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- DFII

- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- DFII

- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz

- ESBMC
- Bound: 10 samples

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$
- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]
- DFII
- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz
- ESBMC
- Bound: 10 samples
- Verification Failed

Failure due to a sum overflow (sum result = 2.0879 > 1).

Input sequence: {0.9995, -0.9995, 0.9995, 1, 1, 1, 0.9995, 0.9995, 0.9995, 0.9995, 1}

Redefine the implementation!

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

Maintain the Representation

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

Change the Realization Form
TDFII presents less sums and products

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz

- ESBMC
- Bound: 10 samples

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz

- ESBMC
- Bound: 10 samples

Repeat the test

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

- Verify overflow
- Verification time:3600
- MSP430 16-bit 16 MHz

- ESBMC
- Bound: 10 samples

- SUCCESS

The problem was solved

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

- Verify limit cycle
- Verification time:3600
- MSP430 16-bit 16 MHz

- ESBMC
- Bound: 10 samples

But verifing limit cycles...

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$

- $< 3,12 >$:3 bits for integer part and 12 bits for fractional part
- Dynamical Range: [-1,1]

- TDFII

- Verify limit cycle
- Verification time:3600
- MSP430 16-bit 16 MHz

- ESBMC
- Bound: 10 samples

- Verification Failed

Appears an oscillation: {-0.002, -0.002, -0.0015, -0.0015, -0.002, -0.002, -0.0015, -0.0015, -0.002, -0.002}.

Zero input sequence

Redefine the implementation!

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$
- $< 3,4 >$:3 bits for integer part and 4 bits for fractional part
- Dynamical Range: [-1,1]
- TDFII
- Verify limit cycle
- Verification time:3600
- MSP430 16-bit 16 MHz
- ESBMC
- Bound: 10 samples

Verifying with a different representation...
There is a trade off: the oscillation is solved; however, there is an accurate loss

DSVerifier-Aided Verification Example

- $C(z) = \frac{0.2(z^2-2z+1)}{z^2-0.25}$
- $< 3,4 >$:3 bits for integer part and 4 bits for fractional part
- Dynamical Range: [-1,1]
- TDFII
- Verify limit cycle
- Verification time:3600
- MSP430 16-bit 16 MHz
- ESBMC
- Bound: 10 samples
- SUCCESS

SUCCESS

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
  .b = {2.813, -0.0163, -1.872},
  .b_size = 3,
  .a = { 1.0, 1.068, 0.1239 },
  .a_size = 3
};
```

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
  .b = {2.813, -0.0163, -1.872},
  .b_size = 3,
  .a = { 1.0, 1.068, 0.1239 },
  .a_size = 3
};
```

→ Numerator Coefficients

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
  .b = {2.813, -0.0163, -1.872},
  .b_size = 3,
  .a = { 1.0, 1.068, 0.1239 },
  .a_size = 3
};
```

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
  .b = {2.813, -0.0163, -1.872},
  .b_size = 3,
  .a = { 1.0, 1.068, 0.1239 },
  .a_size = 3
};
```

→ Number of Coefficients

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
  .b = {2.813, -0.0163, -1.872},
  .b_size = 3,
  .a = { 1.0, 1.068, 0.1239 },
  .a_size = 3
};
```

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
  .b = {2.813, -0.0163, -1.872},
  .b_size = 3,
  .a = { 1.0, 1.068, 0.1239 },
  .a_size = 3
};
```

→ Denominator Coefficients

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

  14-bits architecture: 4 bits for integer and 10 bits for precision parts

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

  14-bits architecture: 4 bits for integer and 10 bits for precision parts

  Dynamical Range: between -5.0 and 5.0

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

  14-bits architecture: 4 bits for integer and 10 bits for precision parts

  Dynamical Range: between -5.0 and 5.0

- **DSVerifier is invoked as:**

./dsverifier *<file>*
  --realization *<i>* *--property <j>* *--x-size <k>*

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

  14-bits architecture: 4 bits for integer and 10 bits for precision parts

  Dynamical Range: between -5.0 and 5.0

- **DSVerifier is invoked as:**

./dsverifier *<file>*
  --realization *<i> --property <j> --x-size <k>*

e.g., DFI, DFII

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

  14-bits architecture: 4 bits for integer and 10 bits for precision parts

  Dynamical Range: between -5.0 and 5.0

- **DSVerifier is invoked as:**

./dsverifier *<file>*
  --realization *<i> --property <j> --x-size <k>*

e.g., DFI, DFII

e.g., OVERFLOW

DSVerifier Command-line Version

- The user provides the digital-system specification via an ANSI-C file
- Consider the following digital system: $H(z) = \frac{2.813z^2 - 0.0163z - 1.872}{z^2 + 1.068z + 0.1239}$

```
#include <dsverifier.h>

digital_system ds = {
   .b = {2.813, -0.0163, -1.872},
   .b_size = 3,
   .a = { 1.0, 1.068, 0.1239 },
   .a_size = 3
};
implementation impl = {
   .int_bits = 4,
   .frac_bits = 10,
   .min = -5,
   .max = 5
};
```

- **Implementation aspects:**

  14-bits architecture: 4 bits for integer and 10 bits for precision parts

  Dynamical Range: between -5.0 and 5.0

- **DSVerifier is invoked as:**

./dsverifier *<file>*
  --realization *<i> --property <j> --x-size <k>*

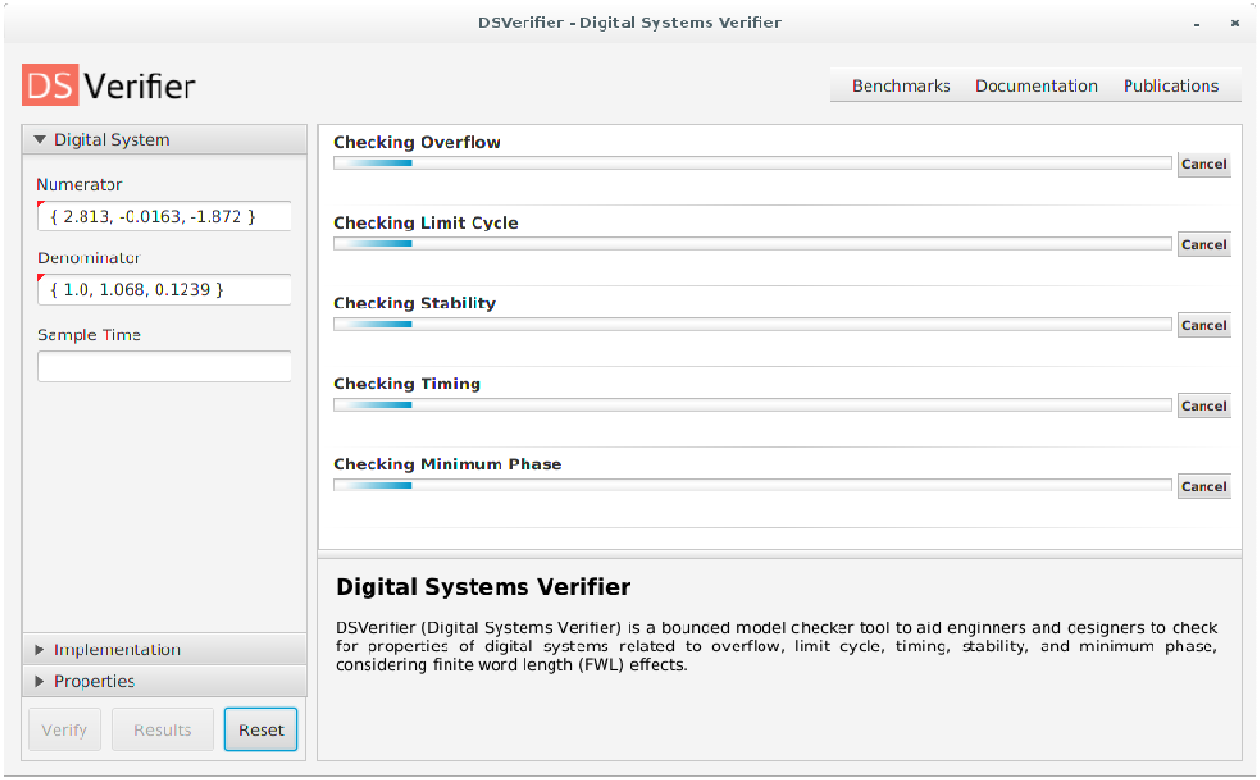
e.g., DFI, DFII

e.g., OVERFLOW

e.g., 10, 20, 30

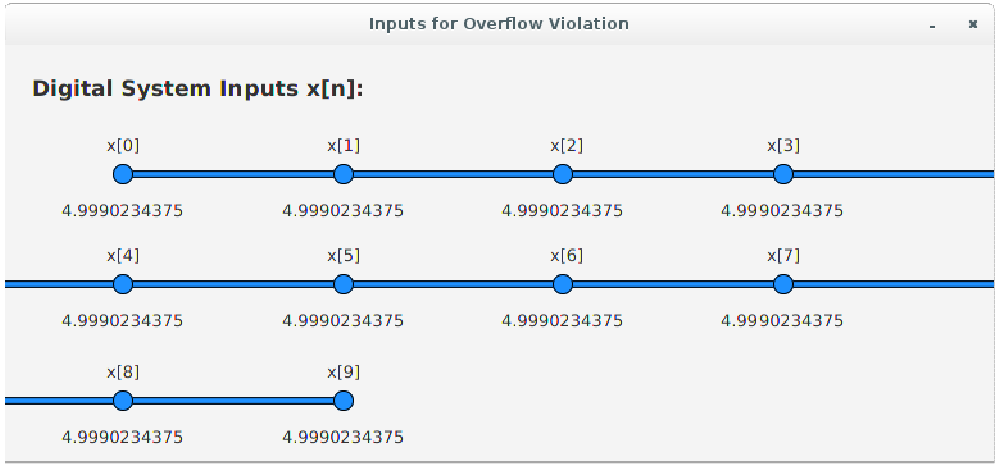
DSVerifier Usage (Graphical User Interface)

- The graphical user interface (GUI) improves usability and attracts more digital-system enginners
- Allows users to provide all required parameters for the verification
- Parallel execution of verification tasks, which is guided by properties



DSVerifier Usage (Graphical User Interface)

- Graphical verification results and counterexamples
- Access the documentation, benchmarks, and publications
- Developed using JavaFX
- Requires Java Runtime Environment Version 8.0 Update 40 (jre1.8.0 40)



Conclusions

- DSVerifier is able to verify digital systems and supports an extensive verification of different properties and realization forms
- DSVerifier can be regarded as an automated and reliable tool if compared to traditional simulation tools
  - An enginner can verify during design phase, if the digital-system presents the expected behavior

Future Work

- Support for closed-loop system verification, more system-level properties, realizations, hardware platforms, and BMC tools
- Source code, benchmarks, experimental results, and publications are available at **http://www.dsverifier.org**

Demonstration