Compositional Synthesis for High-level Design of System-Chips

A Personal Journey in HW-SW Co-Design

Rajesh K. Gupta
University of California, San Diego
INRIA International Chair.

MESL . UCSD . EDU
“Hardware” acceleration is no longer a choice, it is inevitable

- SOC is an inevitable destination for implementation in:
  - implanted, mobile, desktop, cloud, … systems
  - >100X more efficient GOPS/W, $$/part

Image Processing  Multimedia  Machine Learning  Web Search  Big Data Analytics

Deep Learning  Voice Recognition  Linear Algebra  Graphics

130 weeks, $20M problem
Outline

- Component composition beginnings
  - Synthesizability in reducing cost of design

- SOC architectural design for composition:
  - a tiered accelerator fabric in reducing time to design
  - designed to span the range from energy efficiency to flexibility

- Putting design & tools together for a new methodology
  - The Celerity chip.
“High-Level”: A personal journey

Life as circuit designer at Intel c. 1986 was ‘simple’
- Simulation tool reproduced hardware behavior faithfully
- Circuits hooked together: modularity and abstraction
- Designer design automation focused on methodological innovations (split runs, timing calculators, sanity checks)
- Real simple handoff (of printed C-size sheets)
- Local verifiability and updates through back annotations

Then things changed
- Design became data, and data exploded
- Programming paradigm percolated down to RTL
- Designers opened up to letting go of the clock boundary

HDL = HLL + Concurrency + Timing + Reactivity + Structure
- HardwareC, Radha-Ratan, Scenic → SystemC, BALBOA
From HLL to HDL: Semantic Needs

Concurrency
- model hardware parallelism, multiple clocks

Timing Determinism
- provide a “predictable” simulation behavior

Reactive programming
- provide mechanism to model non-terminating interaction with other components, watching, waiting, exceptions

Structural Abstraction
- provide a mechanism for building larger systems by composing smaller ones
An Algorithm for Synthesis of System-Level Interface Circuits

Ki-Seok Chung  Rajesh K. Gupta  C. L. Liu

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

We describe an algorithm for the synthesis and optimization of interface circuits for embedded system components such as microprocessors, memory ASIC, and network subsystems with fixed interfaces. The algorithm accepts the timing characteristics of two system components as input, and generates a combinational interface (glue logic) circuit. The algorithm consists of two parts. In the first part, we determine the direct pin-to-pin connections. There are several ways to connect the two chips in Figure 1-(a). We show two different ways in Figure 1-(b) and 1-(c). The costs of the interface circuits, however, are different in the two cases. One way to quantify the cost is to examine the number of signal transitions as it is directly related to power consumption in current technology. Also in Figure 1-(c), we can see that the $AS^+$ signal is used to drive three input pins. Such sharing of output signal by as many input pins as possible typically results in area minimization in the interface circuit, reducing the number of components.

Figure 2: A timing diagram and an equivalent signal transition graph

Figure 3: Overview of the interface synthesis algorithm.

Figure 5: Interfaces generated by SYNTERFACE for (a) MC6850 ACIA & MC68000 (b) INTEL 8085AH-2 & INTEL 8231 (c) INTEL 80186 & INTEL 82530.
Balboa: Structural Composition of IP Blocks

- Module as a top-level class
- Member functions:
  - model blocks
  - create compound blocks
  - connect component objects
  - set parameters

- A glorified schematic entry
  > set design [new Module]
  > set C0 [$design Component]
  > $design connect C0 C1
  > $design attach_list
  > $design copy_interface
  > $design attach_behavior
  > ...

Building a CCF

- Define compositional semantics across models of computation (MOCs)
  - enable easy system construction and its “formal” validation
  - “adequate”, hierarchical and verifiable composition
  - Create “Virtual” System Architectures

- Can be done through
  - Polymorphic interfaces and mixed compiled and interpreted programming components
  - Incorporating capabilities in the design technology for reflection and introspection

- Use type system for correctness
  - Capture “behavioral types” and model checking obligations

- Primary obstacle to composability
  - Semantic gap between silicon IP and their software models
Compositional Semantic Gap

**Hardware elements:**

![Diagram of hardware elements]

**Software models:**

- `sc_in<int>`
- `sc_in<cData>`
- `Port<char>`
- `Protocol<event>`
- `B2* b2_ptr;`  
  `Int write(int);`

**How to connect?**
Balboa CC: Key Technical Decisions

- A layered development and runtime environment
  - Functionality: describe & synthesize
  - Structure: capture & simulate
- Use an interpreted language for
  - Architecture description
  - Component integration
- Use compiled models for
  - Behavioral description, simulation
- Automatically link the two domains
  - Through a “split-level” interface
- Automatic code “wrapper” generation
  - For component reuse.
Language Layer: Compiled

Component Implementation in C++
- To execute the modeled behavior
- Can use object structure to replicate modeled structures
- Use modeling class library (in SystemC, C++) for
  - Concurrency
  - Bit-level data types
  - Model of time (variants, BFM, ISS etc.)
  - Model of structure
  - OS, Middleware services, abstractions
- Components are implemented by a component library designer, modeling *plus C++ programming*
Language Layers: CIL

- Script-like language based on Object Tcl
- Compose an architecture
  - Instantiate components
  - Connect components
  - Compose objects
  - Build test benches
- Introspection
  - Query its own structure
- Loose typing
  - Strings and numbers

Producer P
Consumer C
Queue Q

P query attributes ⇒ queue_out
C query attributes ⇒ queue_in

P.queue_out query methods ⇒ bind_to read

P.queue_out bind_to Q
…
Language Layers: BIDL

- Describe the component for usage with the CIL
- Exports the interface and internals details:
  - Attributes
  - Methods
  - Relationships
  - Non-functional properties
- Configure a Split-Level Interface (SLI)
  - A custom wrapper for manipulation of the C++ compiled object by the CIL
- Generate the Type System Extensions
  - For the CIL introspection and type inference
  - (Defines the “meta-level” for reflection)

template<class T>
class Producer {
    kind BEHAVIORAL;
public:
    Queue<T>* queue_out;
    unsigned int packet_count;
    void packet_generator.process();
};

INSTANCE (int)
OF_CLASS (Producer)
INSTANCE (BigPacket)
OF_CLASS (Producer)
INSTANCE (SmallPacket)
OF_CLASS (Producer)
Internal Component Architecture

CIL Commands

Split level Interface

Interpreted OTcl class with variables and methods

Type Adapter Bridge

Type system information

Internal compiled object

C++ Objects Interactions
Internal Component Architecture

- **Split-level interface**
  - Link between interpreted and compiled domain
  - Abstracts and manage the underlying C++ object
  - Implements heuristic for type inference
  - Maintains type checking for correct by construction validation
  - Implement the composition model, introspection & reflection

- **Type adapter bridge**
  - Provides a proxy to the internal C++ object
  - Specific for each C++ type
  - Generated by the BIDL

- **Type system information**
  - Specific to the C++ class, generated by the BIDL

- **Interpreted variables and methods**
  - The system architect can add interpreted parts to the component
Type System

- Compiled types are “weakened” in the CIL
  - Data types are abstracted from signal and ports
- Algorithm for data type inference
  - If a component is not typed in the CIL
    - The SLI delays the instantiation of the compiled internal object
    - Interpreted parts of the component are accessible
  - Verify if types are compatible when a relationship is set
    - If a compatible type is found, the SLI allocates the internal object and sets the relationship
    - If not, the link command is delayed until the types are solved

To understand this inferencing, let us look at typing…
Type System and Introspection

- **Fundamental purpose**
  - Static error checking in program composition
- **But, it can also support various “type abstractions”**
  - ML: assign types to functions and variables
  - Tcl/Perl: every variable is a string, convert to number if needed
  - C++: dispatch virtual methods
Type System and Introspection

- **Fundamental purpose**
  - Static error checking in program composition
- **But, it can also support various “type abstractions”**
  - ML: assign types to functions and variable
  - Tcl/Perl: every variable is a string, convert to number if needed
  - C++: dispatch virtual methods

- BALBOA Type system created with reification
  - Enables type inference
  - Enables checking of compositional correctness

- **Automatic inspection of type composition through introspection**
  - Data type checks
  - Protocol match checks
  - Adapter synthesis
An adder:

is polymorphic because its ports can have many type mappings:

\[
\begin{align*}
\text{ports}(c_1) & : \text{int} & \times \text{int} & \times \text{bool} & \times \text{int} & \times \text{bool} \\
\text{ports}(c_2) & : \text{bv8} & \times \text{bv8} & \times \text{bool} & \times \text{bv8} & \times \text{bool} \\
\text{ports}(c_3) & : \text{bv16} & \times \text{bv16} & \times \text{bool} & \times \text{bv16} & \times \text{bool}
\end{align*}
\]

The \( dt_p \) mapping function has 3 choice in assigning the ports to compiled types!

\textit{Mapping can be viewed as an IP selection}
Subtyping & Software Components

Substitutability (polymorphism):

If we replace A by B in the system, will correctness be maintained? (may be a different abstraction, language, required environment)

$\rightarrow$ Problem gets complex as the notion of substitutability is enhanced.
Subtyping Relations

- **Lattice ordering for subtyping and conversion relation**
  - Value subset semantics (range restriction)
  - Char $\leq$ Int $\leq$ Long $\leq$ Num

- **Goal: infer the most general type that will allow a program to be correct**
  - Exact static match
  - Lossless run-time conversion
Type System in Balboa

- **Semi-lattice type relationship:**
  - NP-hard to find a match for a netlist
    - Set P of ports partitioned into k sets (component)
    - Set S of signals
    - For each component, with its port vector p, assign a row from the TAT table such that if there is a signal set is compatible.
    - (One-in-Three Mono 3SAT can be reduced to Type Inference)
  - Full type resolution is not guaranteed
- **Solved as a constrained optimization problem**
  - If a component is not typed in the CIL
    - The SLI delays the instantiation of the compiled internal object
    - Interpreted parts of the component are accessible
  - Verify if types are compatible when a relationship is set
    - If a compatible type is found, the SLI allocates the internal object and sets the relationship
    - If not, the link command is delayed until the types are solved

Reference: TCAD, Dec 2003
What can we do to reduce design time by 100 weeks?”
CERTUS: SPEEDING UP SOC DESIGN BY 5X

Standard cell design flow

Initial RTL creation | Floorplan, RTL, constraint tuning, and physical synthesis | Physical implementation cycle #1 | Physical implementation cycle #2 | Physical implementation cycle #3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
</tr>
<tr>
<td>88</td>
<td>96</td>
<td>104</td>
<td>112</td>
<td>120</td>
</tr>
<tr>
<td>128</td>
<td>136</td>
<td>144</td>
<td>152</td>
<td>weeks</td>
</tr>
</tbody>
</table>

130-week implementation for advanced node design

10 weeks: Physical implementation

32 weeks: Front-end design

16 weeks: Analog design

8 weeks: Physical implementation

30 weeks: Risk of respin

Clinical trials: P&R, Floorplan/RTL/constraints tuning, equivalence checking, timing ECO

1 week: Timing closure

6 weeks: Foundry retargeting, dummy fill, mask flow

4 weeks: Full chip integration: IPs and IOs

2 weeks: Clock generator synthesis, dc/dc conversion

1 week: RTL coding, FPGA verification, software co-verification

Composition

Regularities

Deep Pipelining

Physical design, timing closure, signoff criteria, multiphysics analyses

GDS out

Each of these groups has direct interface to relevant external community (RISC V SIG, MOSIS, NGAS)
Several key innovations, no silver bullet

1: Start with an architectural template that provides for three key MOCs (MPU, GPU, FPGA)
General-purpose computations (OS, IO etc); exploit parallelism at coarse- and fine-grain and provide for custom accelerators.

2: Component Composition Framework (PyMTL)
Capability to rapidly compose and verify different languages (Python, SystemVerilog, System C) at different levels of abstraction (untimed, cycle approx., cycle accurate).

3: Regularizing High-level Synthesis (HLS)
Capability to take brand new application and produce accelerator in 12 weeks with rapid design exploration. ROCC interface and RSP programming model.

4: Backend Design and Verification Flow
Low-complexity, well-structured, rapidly portable, RTL-to-GDS plus analog and hierarchical flow.

Approach: Compartmentalized, fully scripted hierarchy flow for parallel iteration toward timing closure and verification.
5: Standard Template Library (STL) of IP Blocks

Capability: rapid development of traditional (non-HLS) architectural blocks using extremely heterogeneous composability through latency-insensitive interfaces, highly parameterized, standardized containers and components, representing all major HW design building blocks, large and small.

Support for regression testing that ensure that component changes do not break components with dependencies.

6: Fully synthesizable PLL and LDO analog blocks

100% std cell design and APR; easy to port. Use time as analog values instead of RLC's or charge. Achieved industry competitive metrics.

Highly scalable and rapidly deployable/portable digital LDO architecture due to digital PD control. Uses minimal number of custom cells. 16nm tapeout outperforms all other digital LDOs and analog LDOs.
A tiered parallel accelerator fabric that enables decomposition of tasks according to control, communications and memory needs

- **Control-plane** Tier with Five RISC-V Linux cores each capable of running a separate image of Linux
  - Cores used for general purpose control
- **Data-plane** Tier optimized for minimum energy operations for high throughput
  - 496-core (31x16 RISC-V tiled) manycore array
- **Acceleration-plane** Tier that is connected to dataplane and to control plane tiers
  - Dataplane can stream words to accelerators
  - BNN obviates need for dedicated SRAM banks by using dataplane as a programmable stream buffer

- Manycore deploys remote-store programming where the tiles can do word writes to other tiles’ memory spaces thus enabling stream programming
  - Out performs MIT RAW and Tilera with far less buffer space and unlimited deadlock free channels between tiles (10X more dense than RAW)

- Adaptive and response dynamic power management through 2x5 manycore array that provides housekeeping DVFS functions.
- Deploys a tiered memory system as well that supports direct access and DMA
**CELERITY CHIP OVERVIEW**

- **TSMC 16nm FinFET, 25 mm², ~360 million transistors**
- **511 RISC-V Cores**
  - 5 Linux-capable 64-bit “Rocket Cores” generated from Chisel: 5-stage, in-order scalar processor, DP floating point, 16 KB 4-way each I-cache and D-cache, RG64G ISA, 0.97 mm² per core @625MHz
  - 496 core mesh tiled array “Manycore”, distributed memory model, 4KB each instruction and data memory. RV32IM ISA, 80 Gbps duplex adjacent links, 0.024 mm² @1.05 GHz
  - 10 core mesh tiled array “Manycore” (low voltage)
- **1 Binarized Neural Network Accelerator, 0.356 mm²**
  - 13.4 MB model size with 9 layers
- **On-chip synthesizable PLLs and DC/DC LDO**
- **3 Clock Domains**
  - 400 MHz – DDR IO
  - 625 MHz – Rocket Core + BNN Accelerator
  - 1.05 GHz – Manycore Array
- **672-pin flip-chip BGA package**
- **9-month from PDK access to tapeout.**
XY-dimension network-on-chip (NOC)

Unlimited deadlock-free communications

Remote Store Programming Model
- Word writes into other tile’s data memory
- Off-chip communication uses same network
- MIMD programming
  - Fine grain parallelism through high-speed communication between tiles

Token-queue architectural primitive
- Reserves buffer space in remote core
- Ensures buffer is filled before accessed
- Tight producer-consumer synchronization
- Streaming programming model supporting producer-consumer parallelism
High Speed Source-Synchronous Communication Link

Northbridge

DDR-3

PCI-E

USB 3.0

Gig-E

Basejump Motherboard

10-core

Synthesizable DC/DCs

Synthesizable PLLs
Towards Highly Productive Hardware Specialization

- **Manual HDLs**
  - **~2-3X Efficiency gap**

- **C-to-gates**
  - **~10-20X Productivity gap**

- **High-Level Programs**
  - **Goal: Architect-Accessible High-Quality HLS**

- **Cross-layer optimizations**
- **Regularization synthesis**
- **Automated refinement checking**
- **Agile compositional methods**
Compositional Synthesis Toolflow

- Source-to-source transformation to generate optimized C++ or SystemC specification
  - Build on LLVM compiler infrastructure
  - Leverage commercial HLS tools
    - Vivado HLS for FPGA prototyping
    - Cadence C-to-Silicon Compiler for ASIC implementation

Technology used
CERTUS HLS framework based on open-source LLVM compiler infrastructure

Commercial CAD toolflow
Commercial HLS Tool
RTL2GDS Flow
Layout

Design flow
C++ design
C/C++ Front End
LLVM IR
CERTUS HLS XFORMS
Optimized IR
CERTUS HLS Code Gen
HW-friendly C++/SC design
Commercial HLS Tool
TL design

Synthesized Code
C/C++ Code
LLVM
Pattern Detection
Pattern/Resource Selection
HLS tool
Modified C/C++ Code
Modified C/C++ Code
Code Transformation

Cross-layer feedback
InnovaCon #1: Cross-Layer Synthesis Optimizations

- Uncover optimization opportunities by looking across abstraction layers
  - Address the interdependence between HLS and low-level synthesis
  - Intelligent sampling in queries to downstream tools

**HLS**
- Input is high-level CDFG containing cycles, user timing constraints, multi-bit values, and complex ops
  - *The actual hardware (control logic, etc.) has yet to be generated*

** RTL/logic synthesis**
- Part of RTL2GDS flow
  - Input is a low-level netlist with fixed register boundaries, single-bit values, and simple logic ops

---

**Conventional HLS**: 2-cycle latency

**Consider tech mapping**: 1 cycle (combinational)
Innovation #2: Regularizing Synthesis

- Regular structures can reduce the size of design task and optimize QOR.
- Regularity can be in function (similarity of operations), structure (similarity of connections) or topological (similarity of placement). Degree of regularity.
- Specification \( \equiv \) Implementation => They have the same set of execution sequences of visible instructions.
- Visible instructions are: Function call and return statements.
- Two function calls are equivalent if the state of globals and the arguments are the same. Two returns are equivalent if the state of the globals and the returned values are the same.
- Split the program state space into control flow and data flow states
  - CF state explored by traversing CFG generating constraints required for the visible instructions to be matched.
  - DF state explored using an automated theorem prover.

The property checker takes as input a high-level model and through a repeated process of checking and correction produces a golden reference model. The refinement or the equivalent checker takes as input a specification and an implementation and checks if the implementation preserves certain properties of the specification.
InnovaCon #4: Agile PyMTL/HLS Composition

**PyMTL Modeling Framework**
- **Test Source**: Digit 0, Digit 1, Digit 9
- **Test Sink**: 014
- **Golden Results**: 014
- **RTL Architecture**: Nearest Neighbor
- **Digital Circuits**: Digit 0, Digit 1, Digit 9
  - Digit 0 training set
  - Digit 1 training set
  - Digit 9 training set

**Python Hardware Model**
```python
from pymtl import *

class Digitrec(VerilogModel):
    def _init_(s):
        s.digit_strm = InValRdyBundle(Bits(49))
        s.out_strm = OutValRdyBundle(Bits(4))

        s.set_ports(
            'ap_clk': s.clk,
            'ap_rst': s.reset,
            'digit_strm V V': s.digit_strm.msg,
            'digit_strm V V ap_vld': s.digit_strm.val,
            'digit_strm V V ap_ack': s.digit_strm.rdy,
            'out_strm V V': s.out_strm.msg,
            'out_strm V V ap_vld': s.out_strm.val,
            'out_strm V V ap_ack': s.out_strm.rdy,
        )

    def line_trace(s):
        return "{} > {}",format("
```

**C++ Source**
```cpp
void Digitrec(
    hls::stream&lt;digit&gt;& digit_strm,
    hls::stream&lt;bit4&gt;& out_strm
);

void update_knn(
    digit test inst,
    digit train inst, bit6
    min_distances[K_CONST]
);
```

**Simulation Line Trace**

- 83: 0003041060800 >
- 2078: # > 1
- 2160: 000e1c1860f00 >
- 4155: # > 2
- 4237: 0006081810c00 >
- 6232: # > 5
- 6314: 0006123850e00 >
- 8309: . > 8
class ProcXcel ( Model ):

def __init__( s, ProcModel, XcelModel ):
    # Interface
    s.proc_imemreq = OutValRdyBundle ( MemReqMsg4B )
    s.proc_imemresp = InValRdyBundle ( MemRespMsg4B )
    s.proc_dmemreq = OutValRdyBundle ( MemReqMsg4B )
    s.proc_dmemresp = InValRdyBundle ( MemRespMsg4B )
    s.xcel_memreq = OutValRdyBundle ( MemReqMsg4B )
    s.xcel_memresp = InValRdyBundle ( MemRespMsg4B )

    # Child models
    s.proc = ProcModel()
    s.xcel = XcelModel()

    # Processor <-> Memory
    s.connect( s.proc_imemreq, s.proc.imemreq )
    s.connect( s.proc_imemresp, s.proc.imemresp )
    s.connect( s.proc_dmemreq, s.proc.dmemreq )
    s.connect( s.proc_dmemresp, s.proc.dmemresp )

    # Processor <-> Xcel
    s.connect( s.proc.xcelreq, s.xcel.xcelreq )
    s.connect( s.proc.xcelresp, s.xcel.xcelresp )

    # Xcel <-> Memory
    s.connect( s.xcel_memreq, s.xcel.memreq )
    s.connect( s.xcel_memresp, s.xcel.memresp )
Quantifying Results

- Extensive tracking of raw design effort data in person-hours at individual cells, blocks
- Goal: 5x reduction in design effort
  - bsg_comm_link reduction: 21.6X
  - Manycore2x10 reduction: 5.5x
- Goal: 30-week design time for a 10-person design team on a chip of logic blocks >200K gates, multiple mixed signal blocks, multiple SRAM blocks, 3rd party IP
  - Total project duration: 52 weeks from kickoff
  - Total engineering time spent: 12,892 hours or 322 person-weeks
- Top-level chip verification time of DRC, LVS: ~1 week
- Goal: Design productivity target: 50K gates/engineer-day, 1.0 analog block/engineer-week
  - 71K-377K gates/engineer-day, ~1 week for full PLL.
  - 0.71 person-week/day for Clock, 1.5 person-week/day for Power Supply
CERTUS Design Time

- Manycore 16x31 Array: 377K gates/engineer-day
- Rocket core: 71.5K gates/engineer-day
- BNN accelerator: 12.3 K/gates/engineer-day
  - (includes algorithm design, binarization reduces gate count but increases design time. A fixed-point CNN would be 30-50x larger and take less time)

Analog blocks: (Clock Generator: 2 blocks; SAR-LDO: 4 blocks)

- Design of the Clock Generator: 1.25 person-weeks; <1 person-week/block
- Prelayout of the Clock Generator: 2 person-weeks; 1 wk/block
- Postlayout verification of the clock generator: ~4 person-weeks; 2wk/block
- Design of DC/DC Power Supply block: 6 person-weeks; 1.5 person-week/block
- Prelayout verification of the power supply: 1.5 person weeks
- Post layout verification of the power supply: 9 person weeks.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Manycore Generator</th>
<th>Manycore</th>
<th>Coyote</th>
<th>BNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Time</td>
<td>376,903</td>
<td>376,903</td>
<td>71,511</td>
<td>12,294</td>
</tr>
<tr>
<td>Prelayout Verification</td>
<td>5,621</td>
<td>703,553</td>
<td>9,355</td>
<td>3,659</td>
</tr>
<tr>
<td>Postlayout Verification</td>
<td>3,822</td>
<td>191,878</td>
<td>54,097</td>
<td>12,806</td>
</tr>
</tbody>
</table>
SUMMARY AND OUTLOOK

- Nearly Three Decades of Computer Architecture and EDA advances have settled on a few commonly accepted dictums
  - No need for language wars for High-level Design
    - Sophisticated typing and validation tools instead.
  - No universally acceptable compute MOC
    - From Globally Shared Memory multi-threaded programming to NUMA models to direct hardware execution under explicit memory management
      - All three have a role in a realistic machine
  - Focus on Reuse, Modularization and Automation to reduce design time.
Acknowledgements

- BALBOA Team
  - Jean-Pierre Talpin, INRIA, Sandeep Shukla, IIT Kanpur, Fred Doucet, Cadence

- SystemC, SPARK, HLV Teams
  - Sudipta Kundu, Synopsys, Ali Dasdan, Google, Sumit Gupta, IBM, Sorin Lerner, Nik Dutt, Alex Nicolau, Nick Saviou

- CERTUS Team