Correct-by-Construction Techniques in the BIP Context

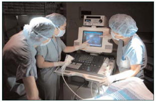
American University of Beirut
-
Faculty of Arts & Sciences - Department of Computer Science

May 17, 2017

Context

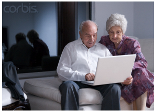
Computer systems are everywhere,

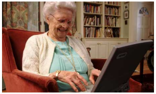





and are for all ages,





Challenges

Systems become more and more complex and the adoption of them is increasing exponentially.

Existing solutions

- Software engineering: verification, test, simulation, ...
- Programming and modeling languages: C/C++, Java, UML, GME, Simulink, .Net, SystemC, ...

But!?

Building *correct* and *efficient* systems is still time-consuming and hardly predictable

Challenges

Systems become more and more complex and the adoption of them is increasing exponentially.

Existing solutions

- Software engineering: verification, test, simulation, ...
- Programming and modeling languages: C/C++, Java, UML, GME, Simulink, .Net, SystemC, ...

But!?

Building *correct* and *efficient* systems is still time-consuming and hardly predictable

Programming and Modeling Languages

We can distinguish two different types of programming and modeling languages:

1. High-level design and modeling languages (Simulink, UML, . . .)
   - ++ Validation, simulation, . . .
   - −− Efficient implementation
2. Low-level modeling languages (C/C++, Java, SystemC, . . .)
   - ++ Efficient implementation
   - −− Validation

Still there is no language that encompasses everything !

Is it possible to define a unified modeling language such that: ++ validation, ++ simulation, ++ efficient implementation ?

Programming and Modeling Languages

We can distinguish two different types of programming and modeling languages:

1. High-level design and modeling languages (Simulink, UML, . . .)
   - ++ Validation, simulation, . . .
   - −− Efficient implementation
2. Low-level modeling languages (C/C++, Java, SystemC, . . .)
   - ++ Efficient implementation
   - −− Validation

Still there is no language that encompasses everything !

Is it possible to define a unified modeling language such that: ++ validation, ++ simulation, ++ efficient implementation ?

Requirements

Requirements for building efficient and correct implementations for complex systems.

1. Component framework (components + composition operators)
2. Abstraction (high-level primitives for modeling behaviors and communications)
3. Expressiveness (powerful primitives for modeling coordination between components)
4. Automated generation of correct and efficient implementations.

Difficulties

- Abstraction reduces efficiency
- Preserving equivalence between high-level model and implementation

Requirements

Requirements for building efficient and correct implementations for complex systems.

1. Component framework (components + composition operators)
2. Abstraction (high-level primitives for modeling behaviors and communications)
3. Expressiveness (powerful primitives for modeling coordination between components)
4. Automated generation of correct and efficient implementations.

Difficulties

- Abstraction reduces efficiency
- Preserving equivalence between high-level model and implementation

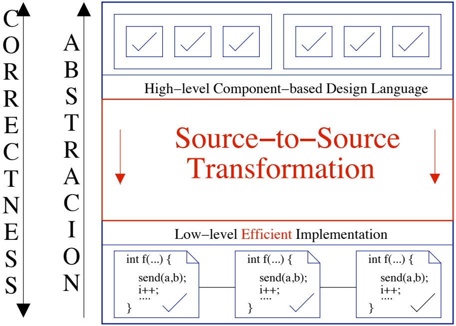
Solution (in general)

Formal Method and Theory

Engineering

Correct-by-Construction

Correct-by-Construction method for automatically generating correct and efficient implementations starting from a high-level model.



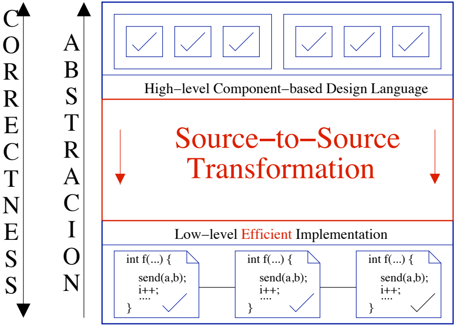
BIP Framework

- Component-based framework (BIP)
- High-level primitives + Expressiveness
- Rigorous semantics
- Strong theoretical backing

- Correct-by-Construction Transformation.
- Efficient implementation (centralized, distributed).

Correct-by-Construction

Correct-by-Construction method for automatically generating correct and efficient implementations starting from a high-level model.



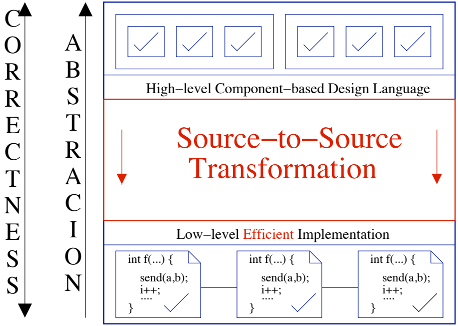
BIP Framework

- Component-based framework (BIP)
- High-level primitives + Expressiveness
- Rigorous semantics
- Strong theoretical backing

- Correct-by-Construction Transformation.
- Efficient implementation (centralized, distributed).

Correct-by-Construction

Correct-by-Construction method for automatically generating correct and efficient implementations starting from a high-level model.



BIP Framework

- Component-based framework (BIP)
- High-level primitives + Expressiveness
- Rigorous semantics
- Strong theoretical backing

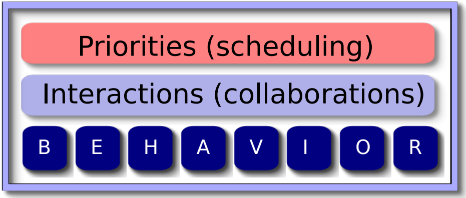
- Correct-by-Construction Transformation.
- Efficient implementation (centralized, distributed).

Outline

Outline

Overview of BIP

BIP is a component framework for modeling heterogeneous systems



BIP: Layered Component Model

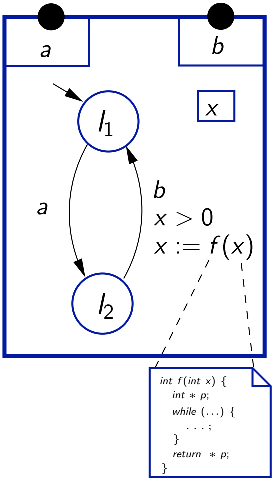
- **Behavior** - petri net extended with data and communication ports
- **Interactions** - set of interactions (interaction = set of ports)
- **Priorities** - partial order on interactions

Behavior

Atomic Component

It is a Petri net extended with data, it is composed of:

- a set of ports, e.g, $\{a, b\}$
- a set of control locations, e.g, $\{l_1, l_2\}$
- a set of variables, e.g, $\{x\}$
- a set of transitions

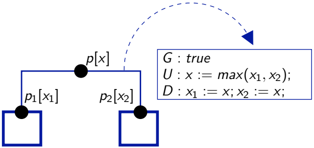


Connector

Connector

A connector is defined by:

- its port p and the associated variable x;
- its interaction defined by a set of ports, e.g, $\{p_1, p_2\}$
- upward update function U (specifying the flow of data upstream)
- downward update function D (specifying the flow of data downstream)



Bliudze and Sifakis

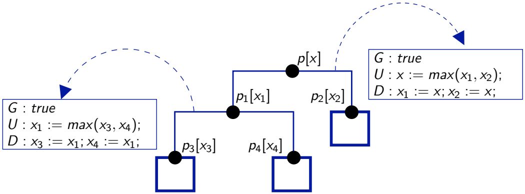
- Strong formalization of the Algebra of Connectors
- Interactions and priorities encompass the universal glue

Connector

Connector

A connector is defined by:

- its port p and the associated variable x;
- its interaction defined by a set of ports, e.g, $\{p_1, p_2\}$
- upward update function U (specifying the flow of data upstream)
- downward update function D (specifying the flow of data downstream)



Bliudze and Sifakis

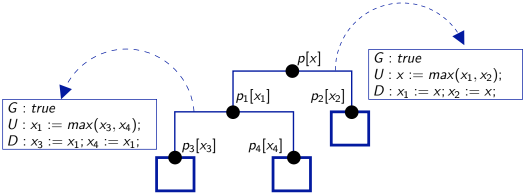
- Strong formalization of the Algebra of Connectors
- Interactions and priorities encompass the universal glue

Connector

Connector

A connector is defined by:

- its port p and the associated variable x;
- its interaction defined by a set of ports, e.g, $\{p_1, p_2\}$
- upward update function U (specifying the flow of data upstream)
- downward update function D (specifying the flow of data downstream)



Bliudze and Sifakis

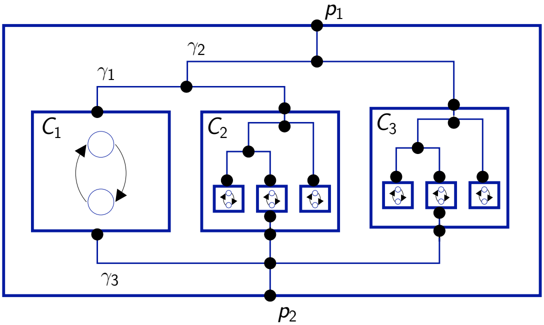
- Strong formalization of the Algebra of Connectors
- Interactions and priorities encompass the universal glue

Composite component

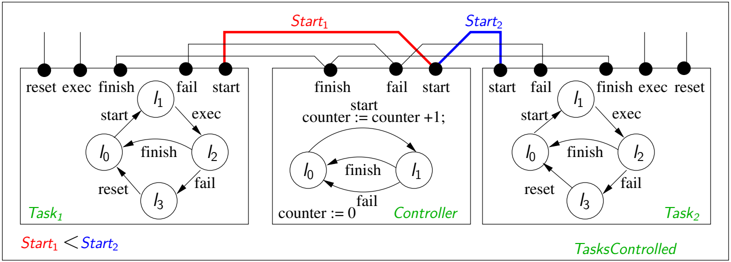
Composite component

A composite component is constructed from:

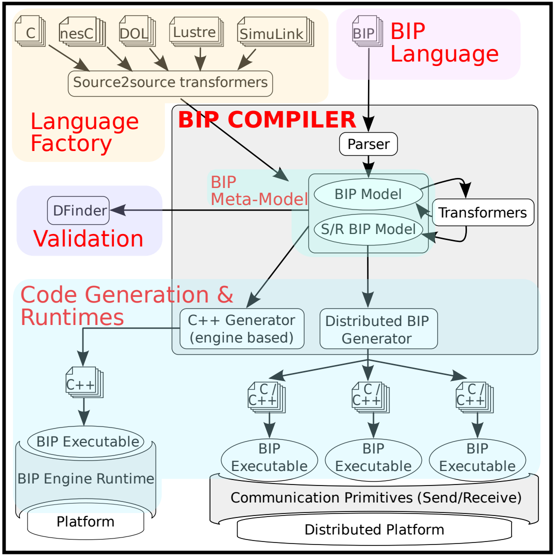
1. existing components, e.g, $\{C_1, C_2, C_3\}$
2. a set of connectors specifying interactions between components, e.g, $\{\gamma_1, \gamma_2, \gamma_3\}$
3. a set of exported ports, e.g, $\{p_1, p_2\}$



Composite Component Example



BIP Tool-chain

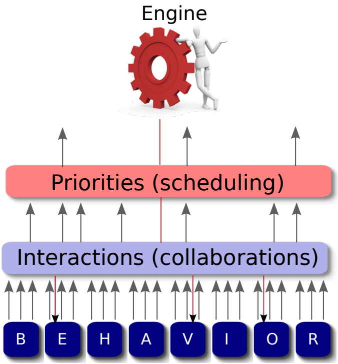


Outline

Problem statement

Engine Protocol

1. Atoms notify the engine of their active ports;
2. The engine enumerates the allowed interactions;
3. filters out low priorities ones;
4. Picks one among those left;
5. Notifies the atoms.



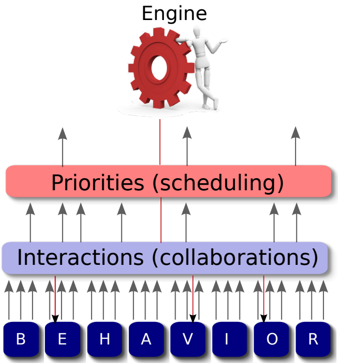
Problem

- clarity of models may be at the detriment of efficiency
- significant overhead in execution time wrt monolithic code

Problem statement

Engine Protocol

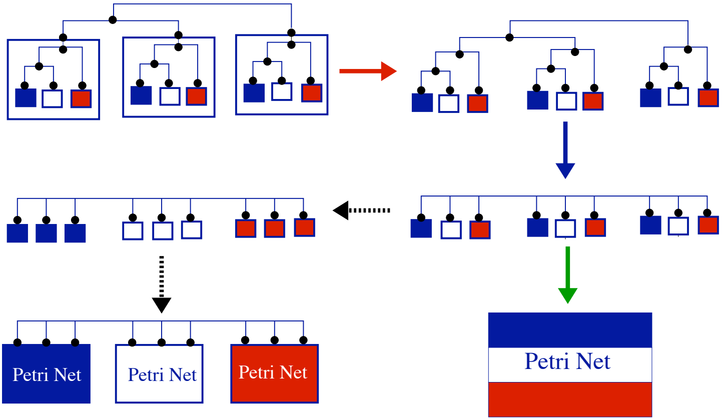
1. Atoms notify the engine of their active ports;
2. The engine enumerates the allowed interactions;
3. filters out low priorities ones;
4. Picks one among those left;
5. Notifies the atoms.



Problem

- clarity of models may be at the detriment of efficiency
- significant overhead in execution time wrt monolithic code

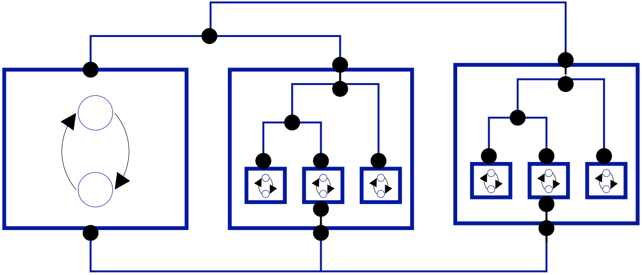
Source-to-Source transformations



Component flattening, *Connector flattening*, *Component composition*

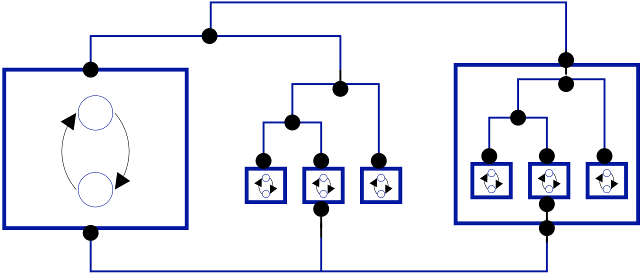
Component flattening

This transformation replaces each non atomic component C_j of C by its content (C is a composite component of $\{C_i\}_{i \in I}$).



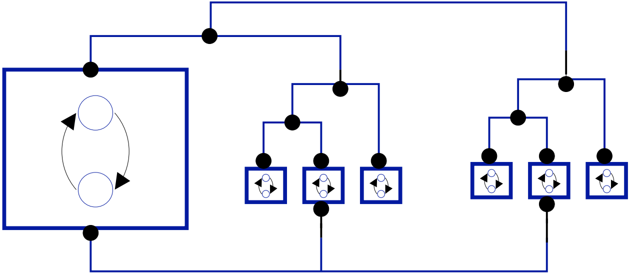
Component flattening

This transformation replaces each non atomic component C_j of C by its content (C is a composite component of $\{C_i\}_{i \in I}$).



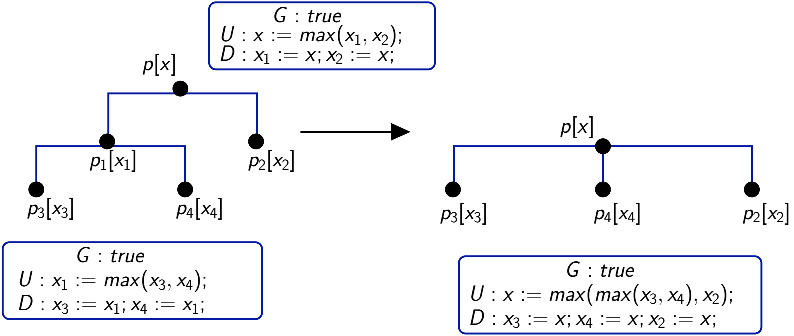
Component flattening

This transformation replaces each non atomic component C_j of C by its content (C is a composite component of $\{C_i\}_{i \in I}$).



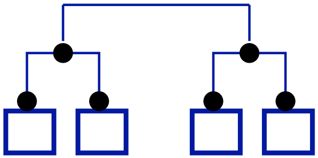
Connector flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j with $\gamma_i \rightarrow \gamma_j$ and produces an equivalent one.



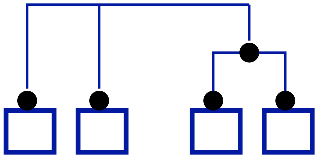
Connector flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j with $\gamma_i \to \gamma_j$ and produces an equivalent one.



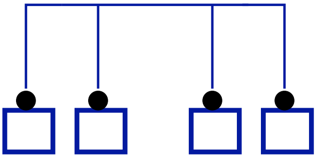
Connector flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j with $\gamma_i \rightarrow \gamma_j$ and produces an equivalent one.



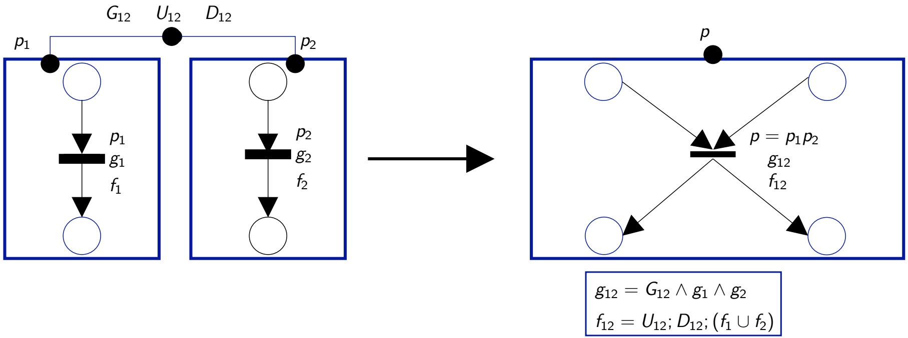
Connector flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j with $\gamma_i \rightarrow \gamma_j$ and produces an equivalent one.

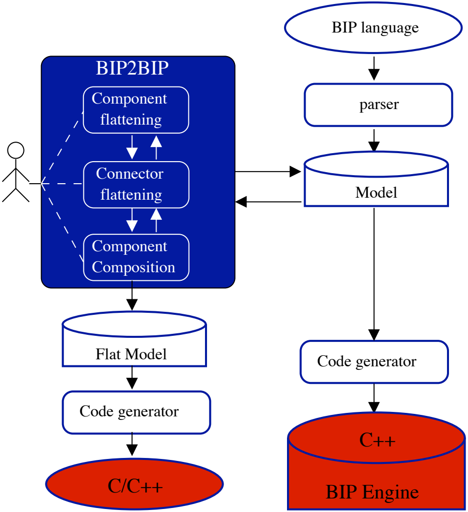


Component composition

This transformation consists in "glueing" together transitions from atomic components that are synchronized through connector.

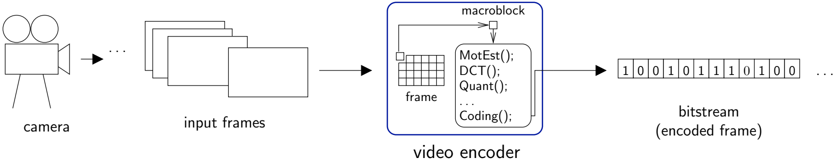


BIP2BIP tool



Example - MPEG video encoder

- collaboration with **STMicroelectronics** (**GaloGiC** project)
- **embedded video encoder**

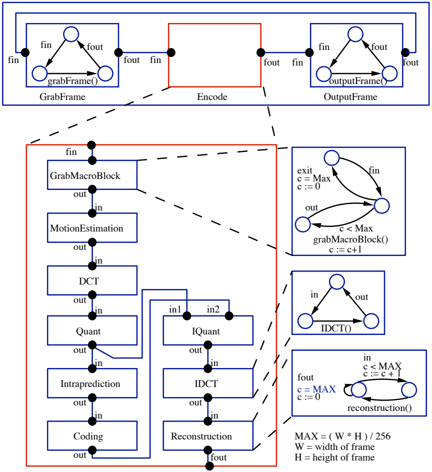


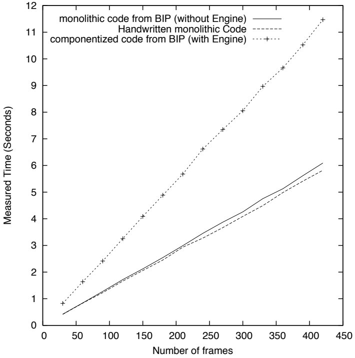
Transform the monolithic sequential program (12000 lines of C code) into a componentized one:

- ++ reusability, schedulability analysis, reconfigurability
- −− overhead in memory and execution time

MPEG video encoder

- GrabFrame: gets a frame and produces macroblocks
- Encode: encodes macroblocks
- OutputFrame: produces an encoded frame



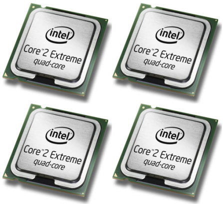


Outline

Motivation

Increase of computing power requires distributed platforms:

- Computer networks
- Multi-core processors
- Networks on chip



Motivation

Deriving from the high-level BIP model a *correct* and *efficient* distributed implementation, that allows:

- Parallelism between components
- Parallel execution between interactions

Challenges

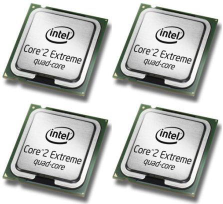
Adding implementation details involves many subtleties:

- Inherent concurrency
- Non-determinism
- Non-atomic actions of distributed systems

Motivation

Increase of computing power requires distributed platforms:

- Computer networks
- Multi-core processors
- Networks on chip



Motivation

Deriving from the high-level BIP model a *correct* and *efficient* distributed implementation, that allows:

- Parallelism between components
- Parallel execution between interactions

Challenges

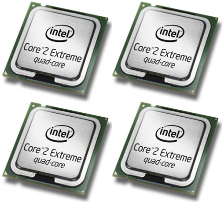
Adding implementation details involves many subtleties:

- Inherent concurrency
- Non-determinism
- Non-atomic actions of distributed systems

Motivation

Increase of computing power requires distributed platforms:

- Computer networks
- Multi-core processors
- Networks on chip



Motivation

Deriving from the high-level BIP model a *correct* and *efficient* distributed implementation, that allows:

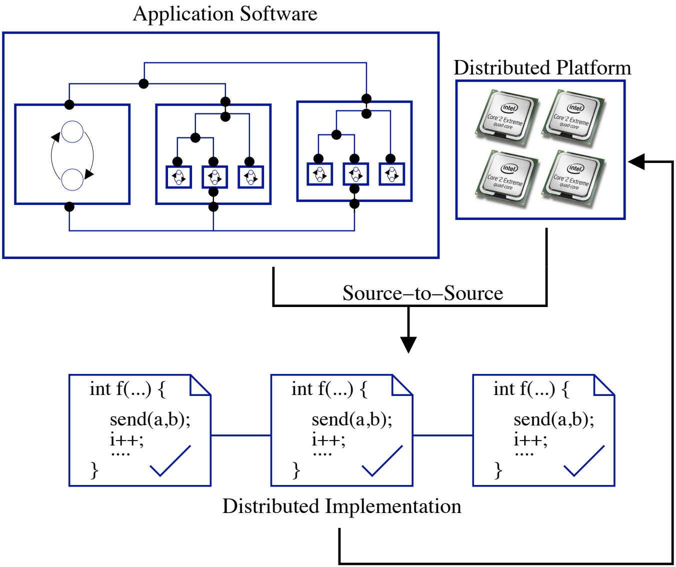
- Parallelism between components
- Parallel execution between interactions

Challenges

Adding implementation details involves many subtleties:

- Inherent concurrency
- Non-determinism
- Non-atomic actions of distributed systems

Motivation


Application Software
Distributed Platform
Source–to–Source
int f(...) {
send(a,b);
i++;
....
}
int f(...) {
send(a,b);
i++;
....
}
int f(...) {
send(a,b);
i++;
....
}
Distributed Implementation

Send/Receive-BIP

BIP is based on:

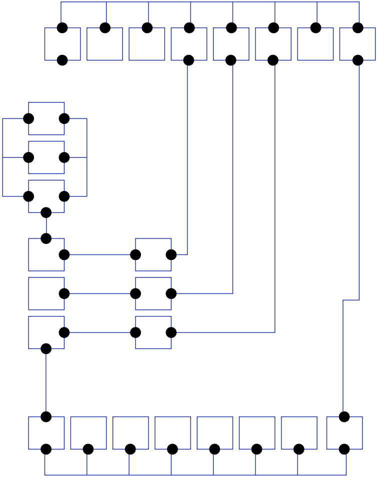
- Global state semantics, defined by operational semantics rules, implemented by the Engine
- Atomic multiparty interactions, e.g. by rendezvous or broadcast

Send/Receive-BIP

Translate BIP models into observationally equivalent Send/Receive-BIP

1. Collection of independent components intrinsically concurrent - No global state
2. Atomicity of transitions is broken by separating interaction from internal computation
3. Point to point communication by asynchronous message passing
4. Translation is correct-by-construction

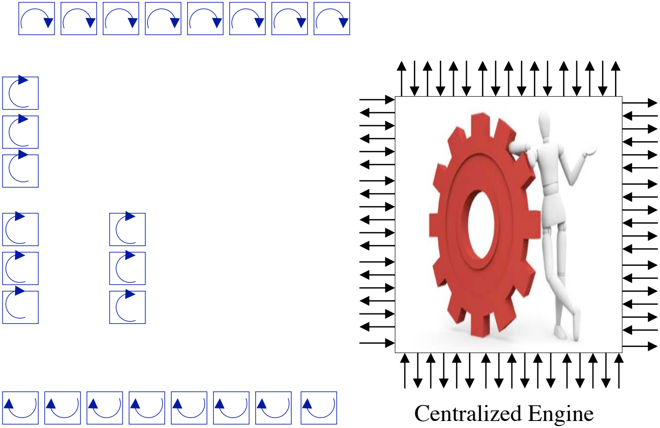
Straightforward solution



Centralized Engine

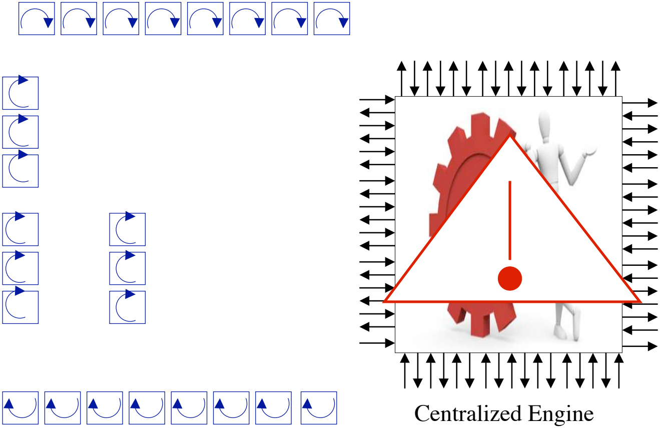
Congestion and no parallelism between interactions !

Straightforward solution



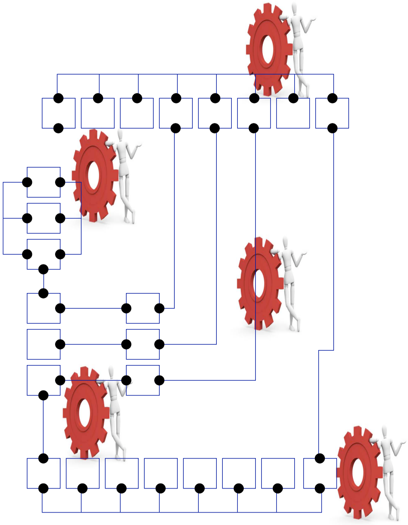
Congestion and no parallelism between interactions !

Straightforward solution



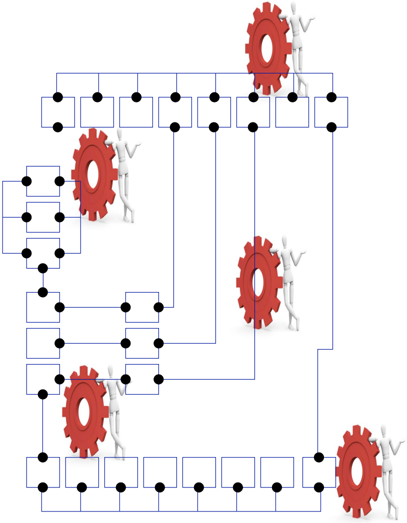
Congestion and no parallelism between interactions !

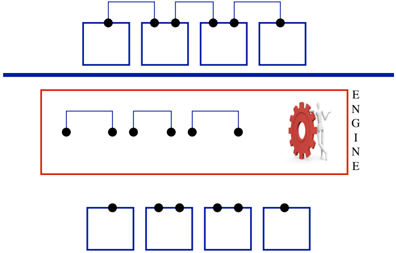
Distributed engines - Challenges



Decentralization requires separate engines: need to take care of **"conflicts"**

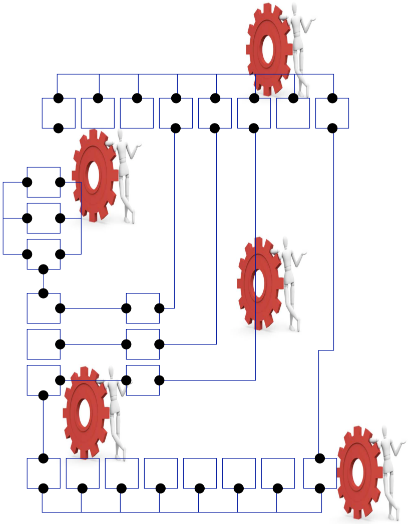
Distributed engines - Challenges

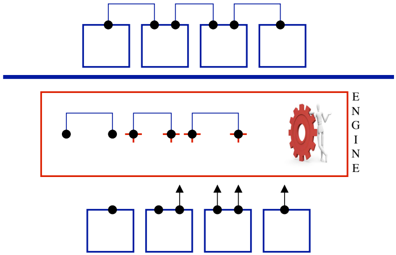




Decentralization requires separate engines: need to take care of "conflicts"

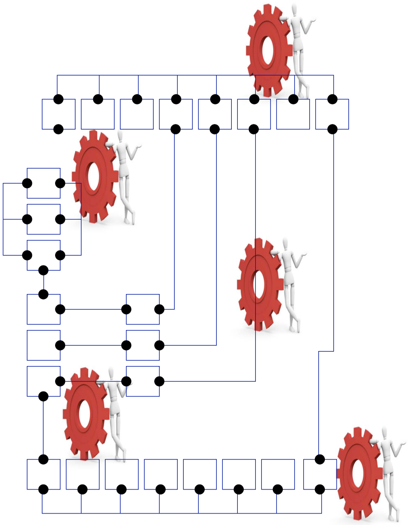
Distributed engines - Challenges

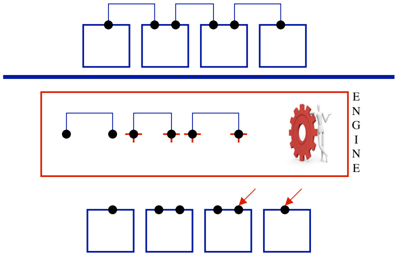




Decentralization requires separate engines: need to take care of "conflicts"

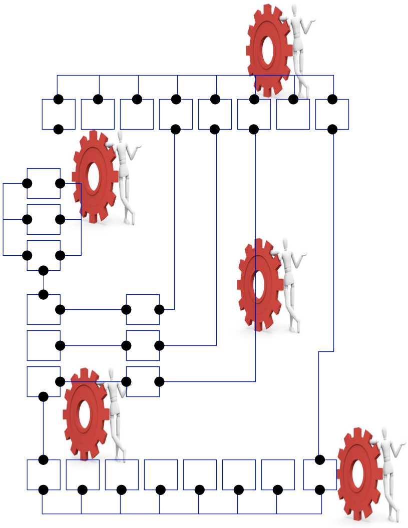
Distributed engines - Challenges

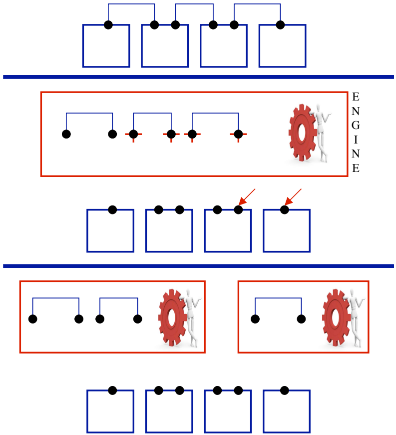




Decentralization requires separate engines: need to take care of "conflicts"

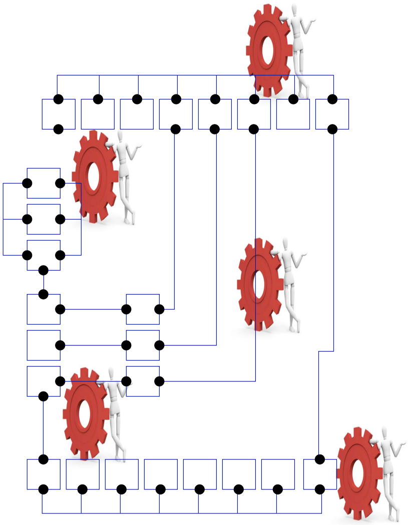
Distributed engines - Challenges

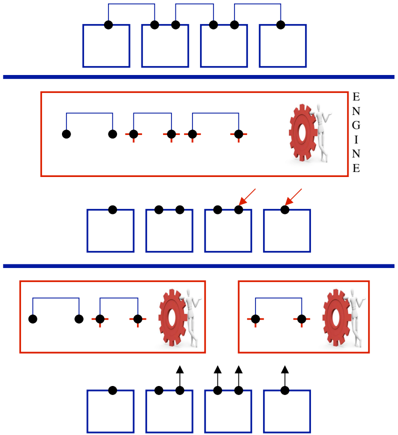




Decentralization requires separate engines: need to take care of "conflicts"

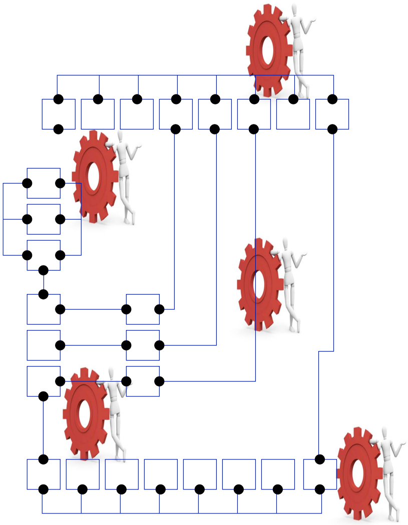
Distributed engines - Challenges

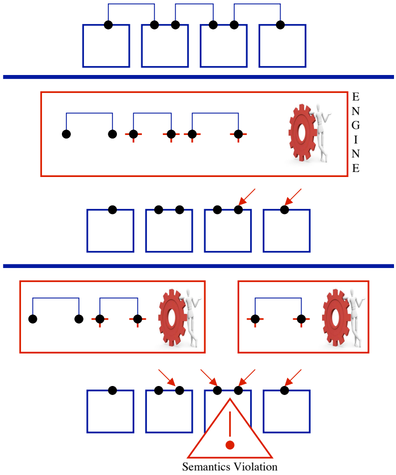




Decentralization requires separate engines: need to take care of "conflicts"

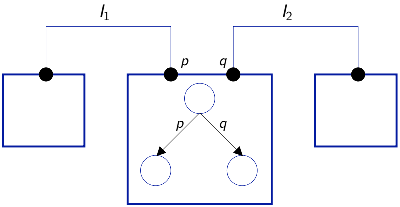
Distributed engines - Challenges

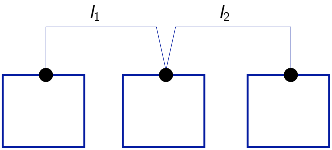




Decentralization requires separate engines: need to take care of **"conflicts"**

Conflicting interactions



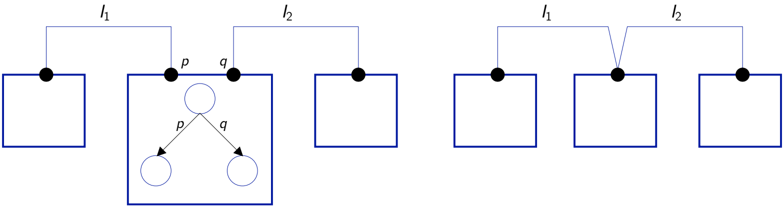


I_1 and I_2 are using both sides ports of a choice in a component

I_1 and I_2 share a common port

I_1 and I_2 are conflicting ($I_1 \# I_2$)

Conflicting interactions



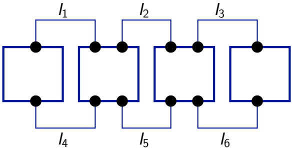
I_1 and I_2 are using both sides ports of a choice in a component

I_1 and I_2 share a common port

I_1 and I_2 are conflicting ($I_1 \# I_2$)

Conflict-free distributed engines

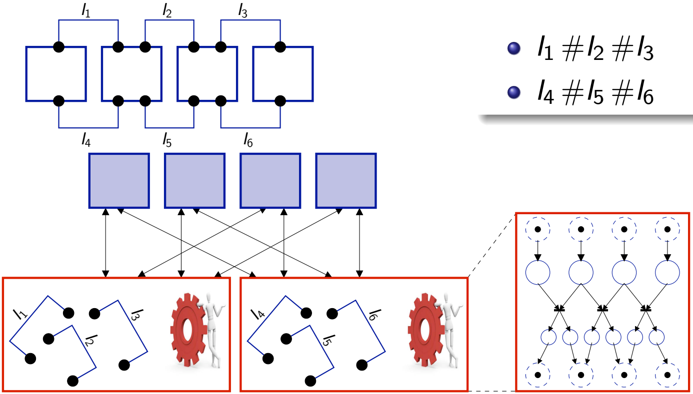
Distributed Engines Conflict-Free by Construction, by grouping interactions according to the transitive closure of the conflict relation #



- $I_1 \# I_2 \# I_3$
- $I_4 \# I_5 \# I_6$

Conflict-free distributed engines

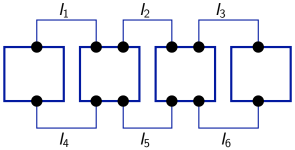
Distributed Engines Conflict-Free by Construction, by grouping interactions according to the transitive closure of the conflict relation #



- $I_1 \# I_2 \# I_3$
- $I_4 \# I_5 \# I_6$

Conflict-free distributed engines

Distributed Engines Conflict-Free by Construction, by grouping interactions according to the transitive closure of the conflict relation #



- $I_1 \# I_2 \# I_3$
- $I_4 \# I_5 \# I_6$

Drawbacks

Grouping conflicting interactions according to the transitive closure reduces drastically parallelism between interactions.

3—Tier architecture

Conflict Resolution Protocol

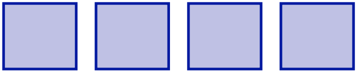
Resolves conflict between engines

Interaction Protocol

- Determined by a partition of the interactions
- Executes interactions

Atomic Components

- Send offers
- Wait for notifications
- Execute local computations



3—Tier architecture

Conflict Resolution Protocol

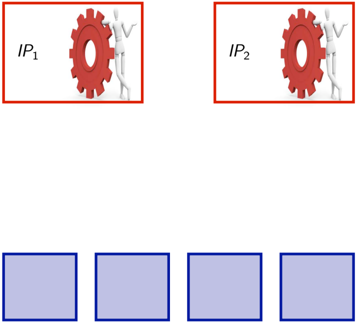
Resolves conflict between engines

Interaction Protocol

- Determined by a partition of the interactions
- Executes interactions

Atomic Components

- Send offers
- Wait for notifications
- Execute local computations



3—Tier architecture

Conflict Resolution Protocol

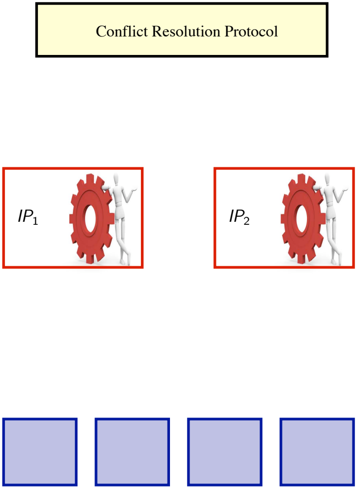
Resolves conflict between engines

Interaction Protocol

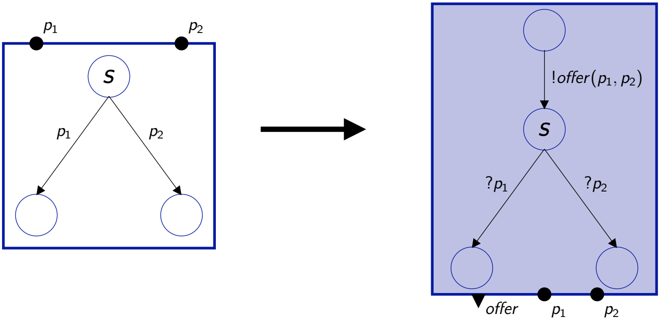
- Determined by a partition of the interactions
- Executes interactions

Atomic Components

- Send offers
- Wait for notifications
- Execute local computations



Transforming atomic components



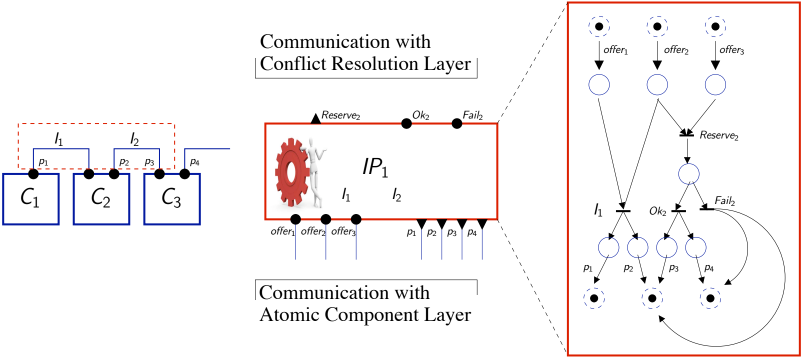
Global state model

Choice made by the global engine

Partial state model

1. sends an offer indicating the available ports
2. it waits for a notification to execute the corresponding transition

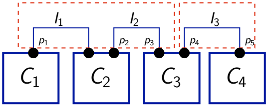
Interaction protocol

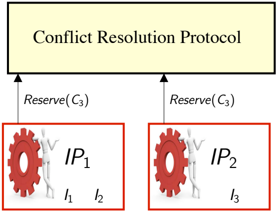


1. Receives offers
2. Detects enabled interactions and tries to execute:
   - interactions with only local conflicts (immediate execution)
   - interactions with external conflicts (request to the conflict resolution layer)
3. Notifies atomic components

Conflict resolution protocol

Each engine needs to reserve components in order to execute an externally conflicting interaction.

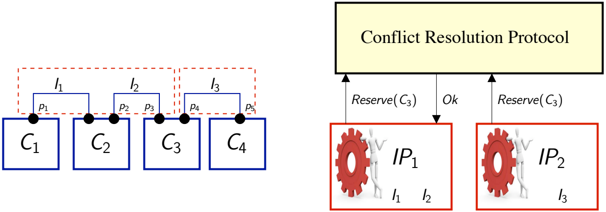




The protocol resolves conflict between Interaction Protocols (Engines)

Conflict resolution protocol

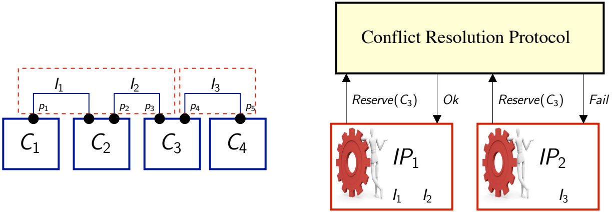
Each engine needs to reserve components in order to execute an externally conflicting interaction.



The protocol resolves conflict between Interaction Protocols (Engines)

Conflict resolution protocol

Each engine needs to reserve components in order to execute an externally conflicting interaction.



The protocol resolves conflict between Interaction Protocols (Engines)

Conflict resolution protocol variations

Centralized version

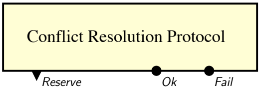
one component is responsible for solving all conflicts

Token ring

- Each component corresponds to an externally conflicting interaction
- A token circulates through all these components
- Only the owner of the token can confirm/deny reservation

Dining philosophers

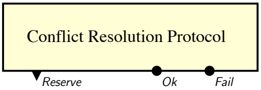
- Each component corresponds to an externally conflicting interaction
- Two interactions share a fork if they are conflicting
- To confirm/deny reservation, all forks from the neighborhood are required



Conflict resolution protocol variations

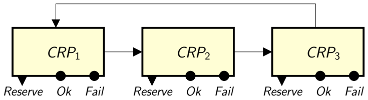
Centralized version

one component is responsible for solving all conflicts



Token ring

- Each component corresponds to an externally conflicting interaction
- A token circulates through all these components
- Only the owner of the token can confirm/deny reservation



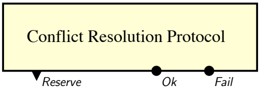
Dining philosophers

- Each component corresponds to an externally conflicting interaction
- Two interactions share a fork if they are conflicting
- To confirm/deny reservation, all forks from the neighborhood are required

Conflict resolution protocol variations

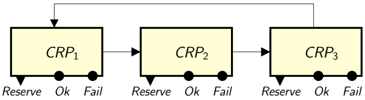
Centralized version

one component is responsible for solving all conflicts



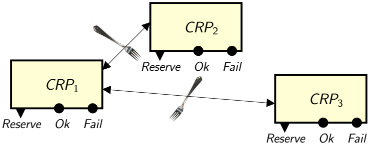
Token ring

- Each component corresponds to an externally conflicting interaction
- A token circulates through all these components
- Only the owner of the token can confirm/deny reservation

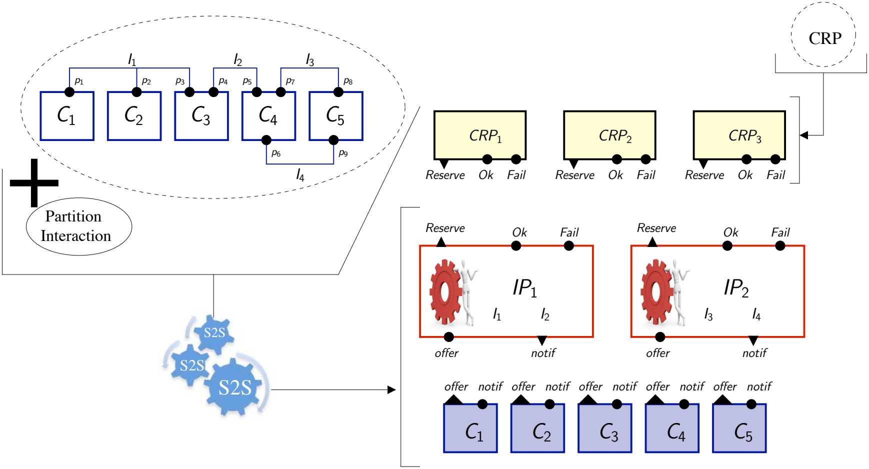


Dining philosophers

- Each component corresponds to an externally conflicting interaction
- Two interactions share a fork if they are conflicting
- To confirm/deny reservation, all forks from the neighborhood are required

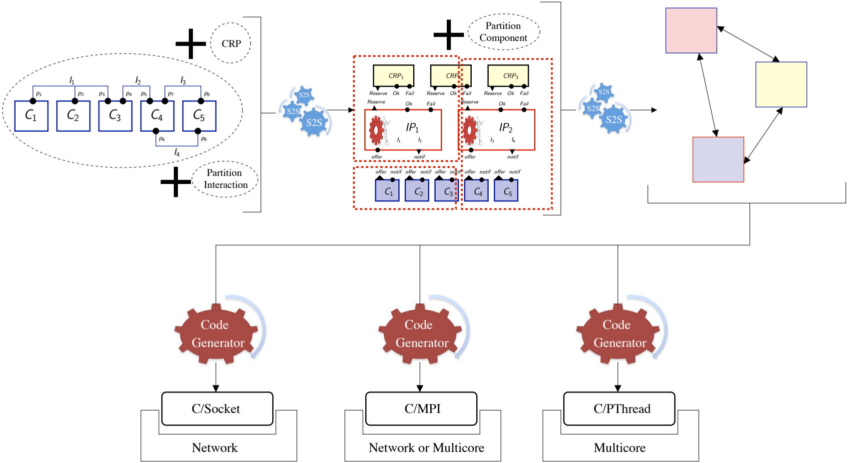


3−Tier architecture



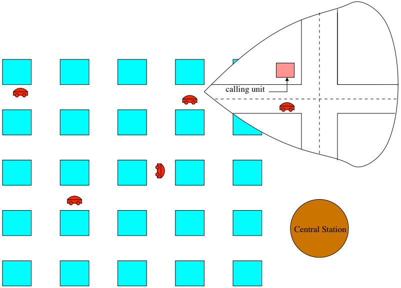
BIP and $BIP^{3-\text{Tier}}$ are Observationally equivalent when using Centralized protocol and we have Trace equivalent when using Token ring and Dining philosophers protocols

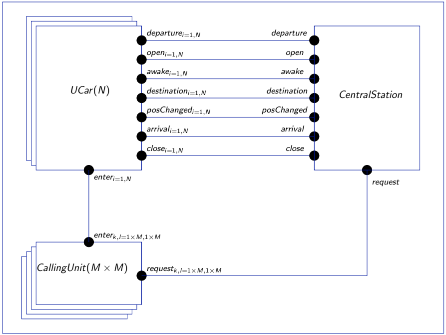
Design Methodology and code generator



Example - UTOPAR

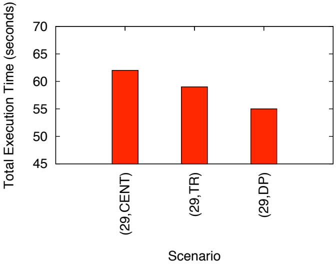
- Industrial case study of the Combest Project
- UTOPAR is an automated transportation system managing various requests for transportation



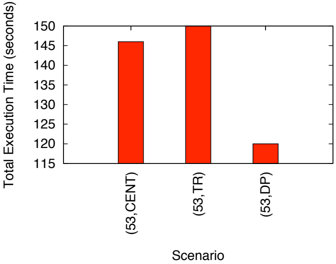


UTOPAR - Benchmarks

5×5 calling units and 4 cars and 29 Engines (Interaction protocols)



7×7 calling units and 4 cars and 53 Engines (Interaction protocols)

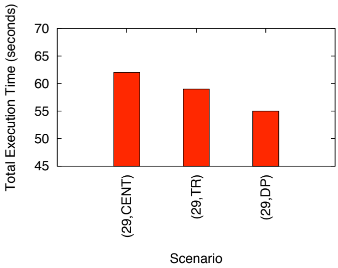


Performance of responding 10 requests per calling unit

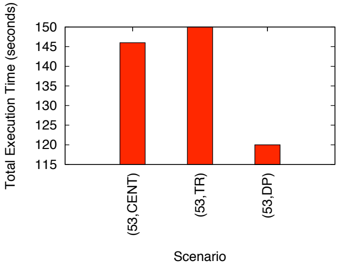
- Dining philosophers protocol outperforms other protocols
- Fully automated distributed C is generated

UTOPAR - Benchmarks

5 × 5 calling units and 4 cars and 29 Engines (Interaction protocols)



7 × 7 calling units and 4 cars and 53 Engines (Interaction protocols)



Performance of responding 10 requests per calling unit

- Dining philosophers protocol outperforms other protocols
- Fully automated distributed C is generated

Outline

Conclusion

It is possible to reconcile component-based incremental design and efficient code generation by applying a paradigm based on the combined use of:

1. A high-level modeling language, BIP, based on
   - a well-defined operational semantics, and
   - supporting powerful mechanisms for expressing structured coordination between components
2. Using the D-Finder tool, to generate and/or check invariants of the components and validate their properties
3. "**Correct-by-Construction**" transformations that allows to automatically generate efficient centralized or distributed implementations