CITADEL

CRITICAL INFRASTRUCTURE PROTECTION
USING ADAPTIVE MILS
www.citadel-project.org

Towards Adaptive MILS Systems: Model-Based Design, Verification and Run-Time Adaptation

Alessandro Cimatti*, Rance DeLong†, Ivan Stojic*
and Stefano Tonetta*

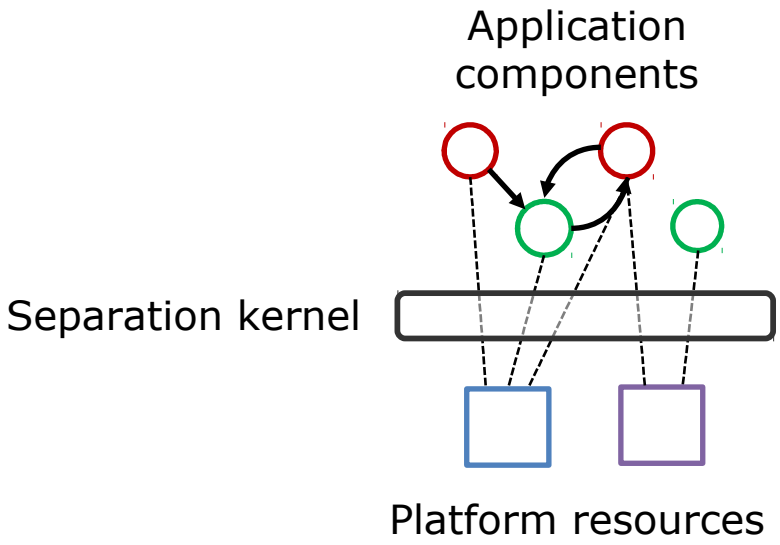
* FONDAZIONE BRUNO KESSLER

† THE Open GROUP

Introduction

MILS

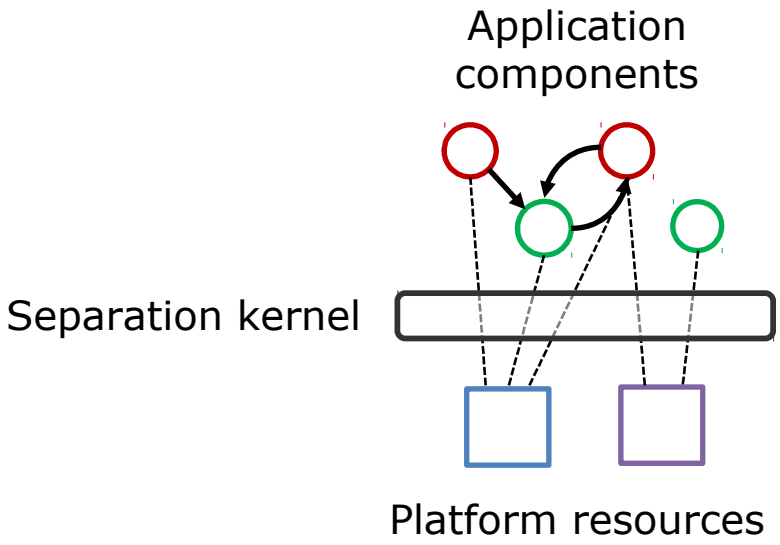
- MILS comprises:
  - a philosophy of design
  - a platform for deployment
  - a set of tools for
    - specification
    - verification
    - configuration
    - assurance



- Overarching objective of MILS: the ability to provide **demonstrable assurance**, necessitating
  - design-time rigor
  - analysability

MILS

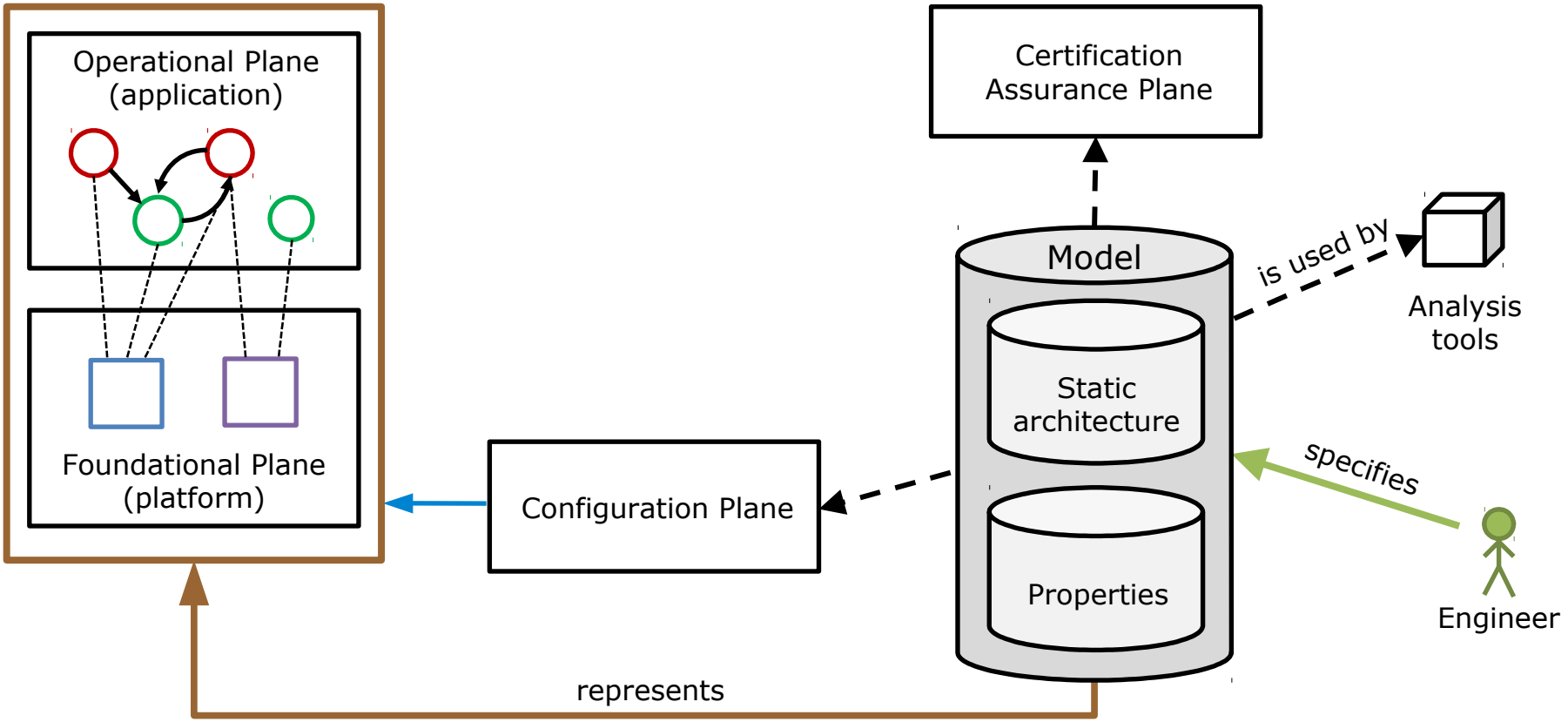
- Traditional MILS principles
  - simplicity
  - smallness
  - isolation
  - static configuration



- However, some applications
  - require high assurance
  - entail requirements antithetical to some of the above principles
- In Distributed MILS Project (www.d-mils.org), isolated and static MILS was extended to distributed systems of multiple MILS nodes, while preserving deterministic execution characteristics of MILS systems

Formal methods in D-MILS

- In Distributed MILS, distributed applications with static architectures are formally modeled in order to support system analysis, (initial) configuration, and system certification



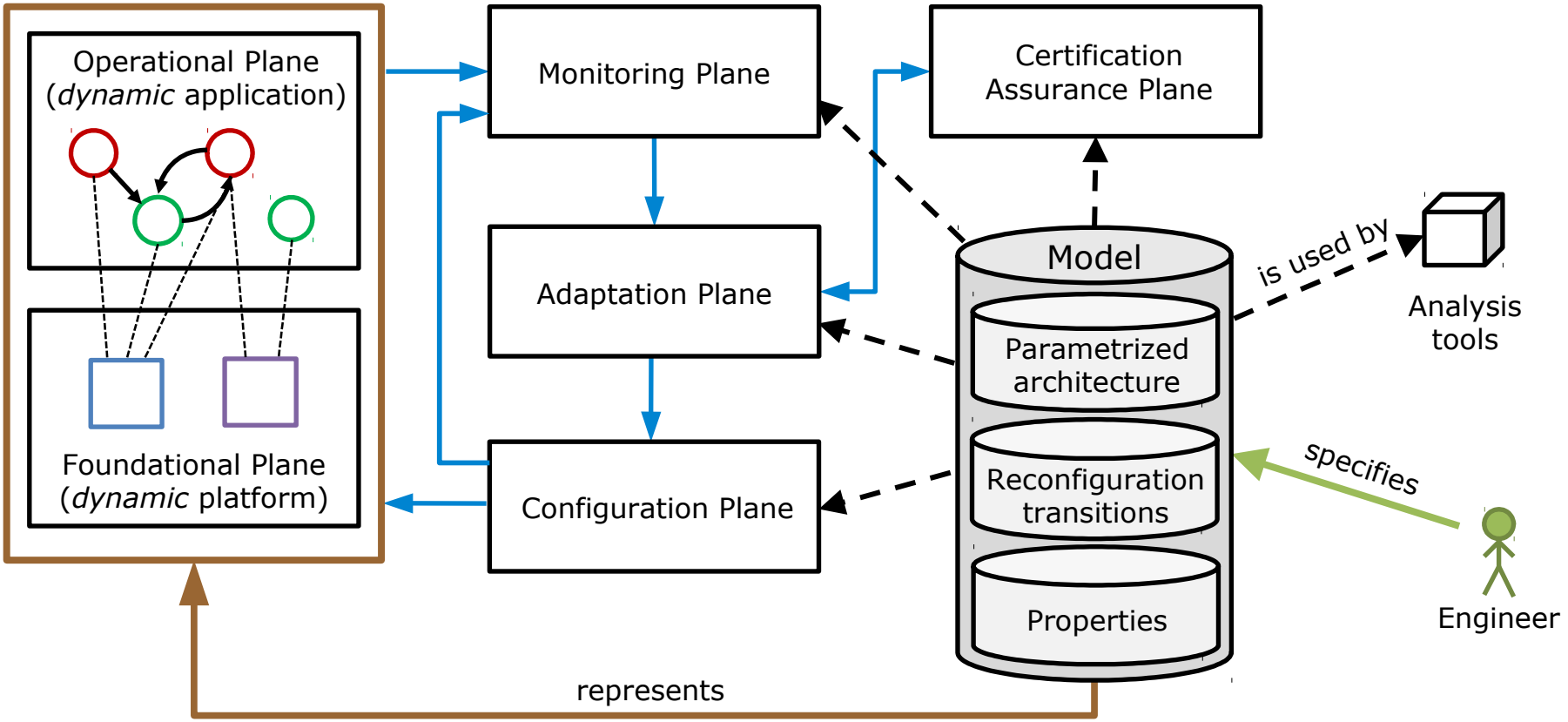
Adaptive MILS

CITADEL Project

- In CITADEL Project, distributed MILS is extended to **dynamically reconfigurable self-adaptive** systems, while
  - preserving analysability
  - providing demonstrably assurable adaptation
- Self-adaptation
  - effective approach to deal with modern highly complex and dynamic software systems
  - major challenge: provide guarantees about the properties of self-adaptive systems
    - solution: use formal methods to
      - design the system
      - verify the system
      - (assurably) safely adapt the system at run-time

CITADEL Framework and Model

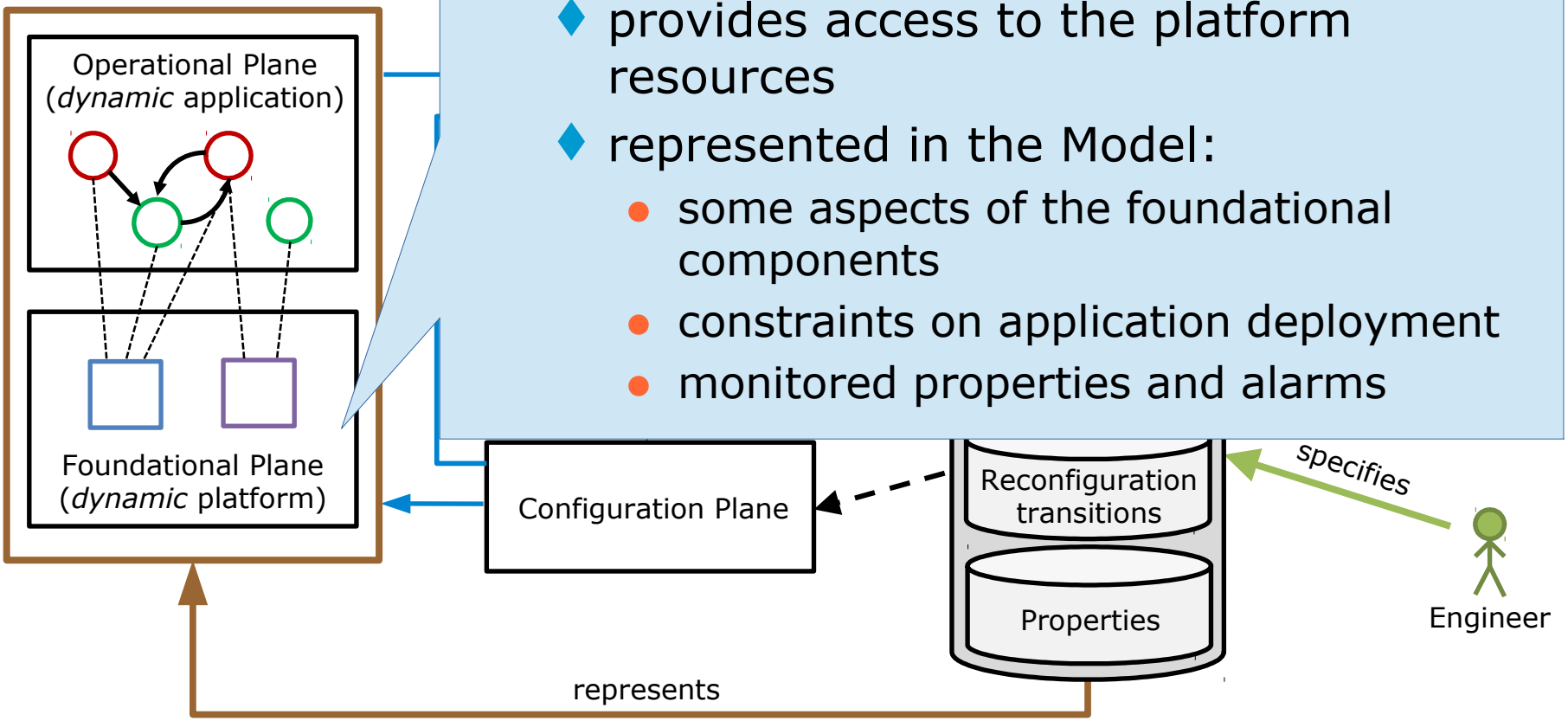
- In CITADEL, applications with dynamic architectures are formally modeled in order to additionally support dynamic reconfiguration, self-adaptation, and run-time certification



CITADEL Framework and Model

- In CITADEL, a
  formally mode
  reconfiguratio

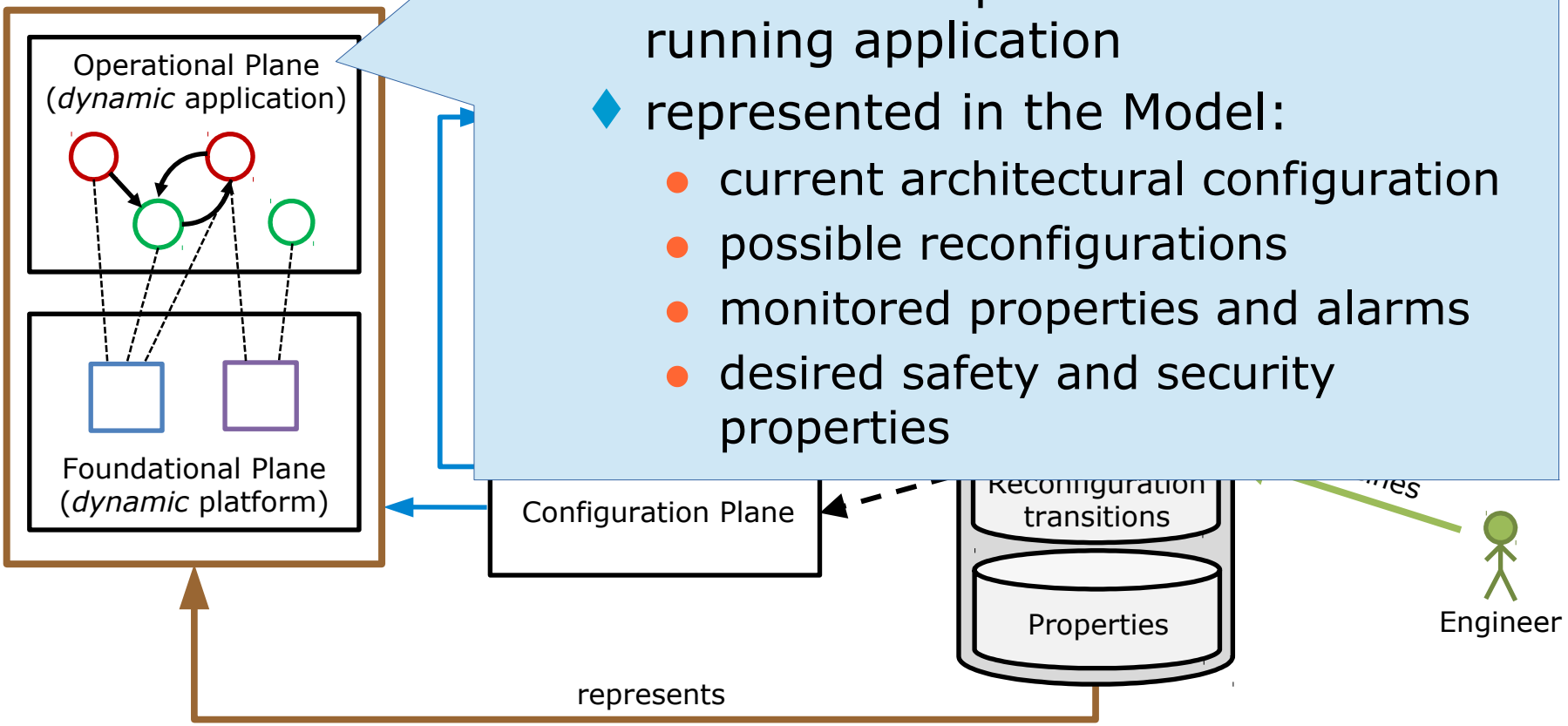
- Foundational Plane
  - composition of MILS foundational components based on a separation kernel
  - provides access to the platform resources
  - represented in the Model:
    - some aspects of the foundational components
    - constraints on application deployment
    - monitored properties and alarms



CITADEL Framework and Model

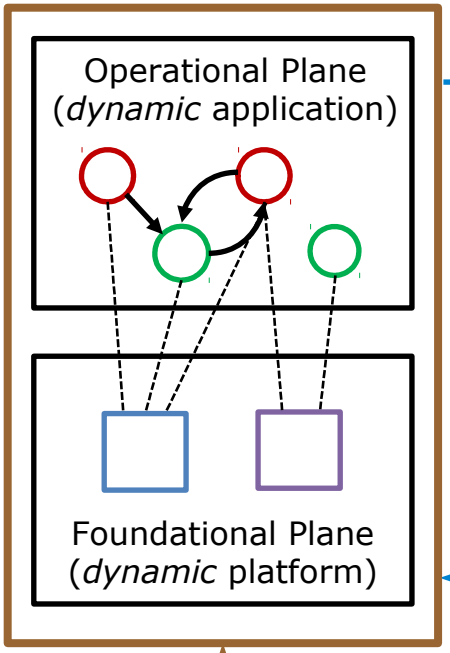
- In CITADEL, applications with dynamic architectures are formally model
  reconfiguration

  - Operational Plane
    - contains components of the running application
    - represented in the Model:
      - current architectural configuration
      - possible reconfigurations
      - monitored properties and alarms
      - desired safety and security properties



CITADEL Framework and Model

- In CITADEL, applications with dynamic architectures are formally modelled in order to additionally support dynamic reconfiguration, self-adaptation, and run-time certification



Monitoring Plane

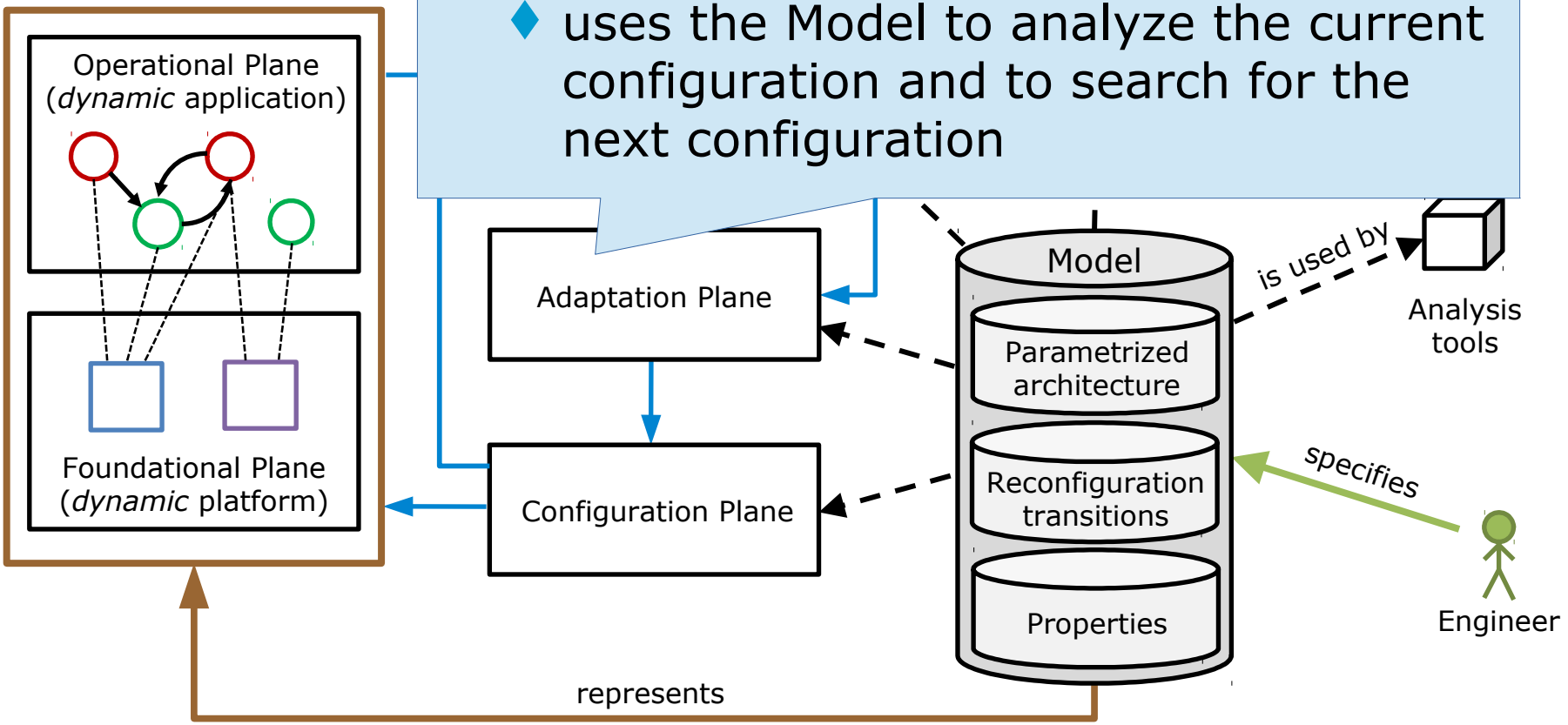
Certification Assurance Plane

- Monitoring Plane
  - contains monitors and sensors which monitor components in the OP and resources in the FP
  - generates alarms when it detects faulty or suspicious behaviour
  - uses data from the Model:
    - monitored properties and alarms

CITADEL Framework and Model

- In CITADEL, formally moc… reconfiguratio… c … n

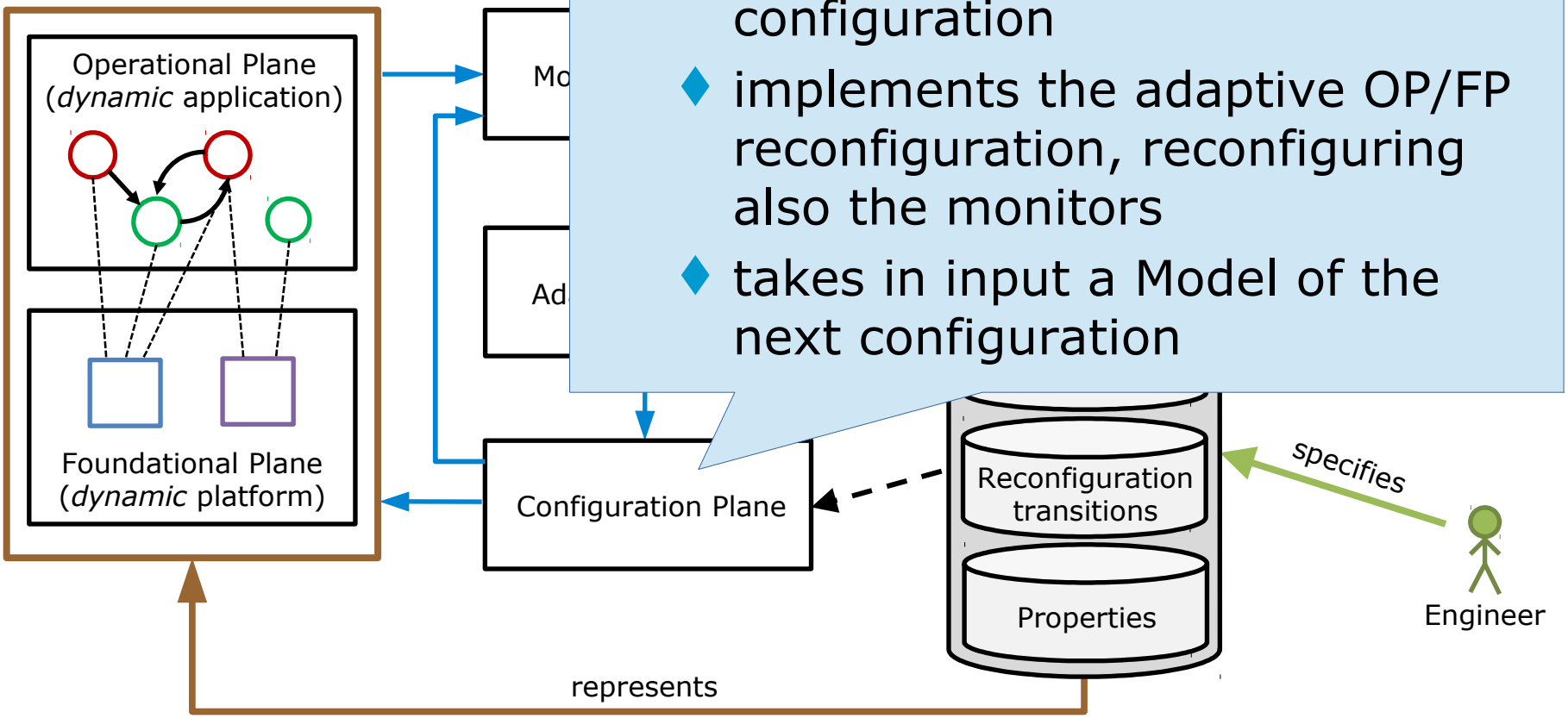
- Adaptation Plane
  - performs reasoning about adaptive reconfigurations of the OP and the FP
  - uses the Model to analyze the current configuration and to search for the next configuration



CITADEL Framework and Model

- In CITADEL, applications with dynamic architectures are formally modelled i
  reconfiguration, sel

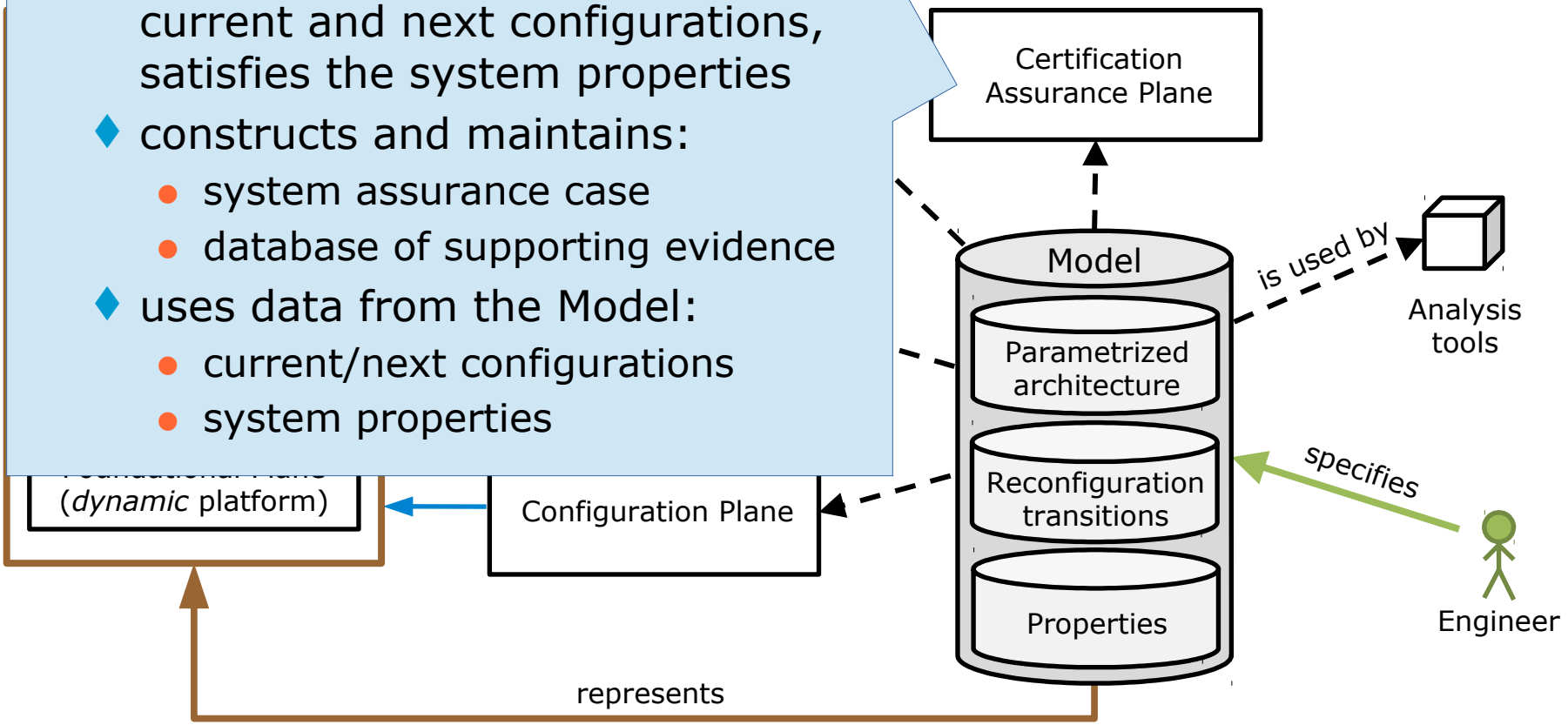
- Configuration Plane
  - implements the initial OP/FP configuration
  - implements the adaptive OP/FP reconfiguration, reconfiguring also the monitors
  - takes in input a Model of the next configuration



CITADEL Framework and Model

- In CITADEL, applications with dynamic architectures are formally modelled in order to additionally support dynamic and run-time certification

- Certification Assurance Plane
  - verifies that the model, in the current and next configurations, satisfies the system properties
  - constructs and maintains:
    - system assurance case
    - database of supporting evidence
  - uses data from the Model:
    - current/next configurations
    - system properties



Adaptive MILS

Model-Based Design

Architecture Analysis & Design Language (AADL)

- Architecture description language
- Standardized by SAE International
- A hierarchical architecture can be modeled compositionally by specifying:
  - component types (interfaces)
    - event ports
    - data ports (of some datatype)
  - component implementations
    - subcomponents
    - connections of ports of subcomponents
    - can be empty (leaf components)

Component types in AADL

Component category

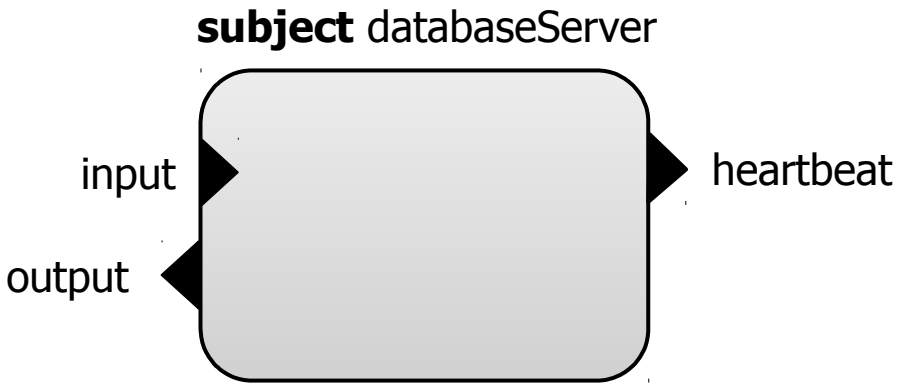
Component type name

Port name

Port type

```
subject databaseServer
   features
      input: in event data port sqlRequest.Data;
      output: out event data port sqlResponse.Data;
      heartbeat: out event port;
end databaseServer;

subject implementation databaseServer.impl
   -- This implementation is empty.
end databaseServer.impl;
```



Component implementations in AADL

```
system sys
    -- This type is empty.
end sys;

system implementation sys.impl
    subcomponents
        client: subject client.impl;
        server: subject applicationServer.impl;
        database: subject databaseServer.impl;
    connections
        port server.db_output -> database.input;
        port database.output -> server.db_input;
        port client.output -> server.input;
        port server.output -> client.input;
end sys.impl;
```

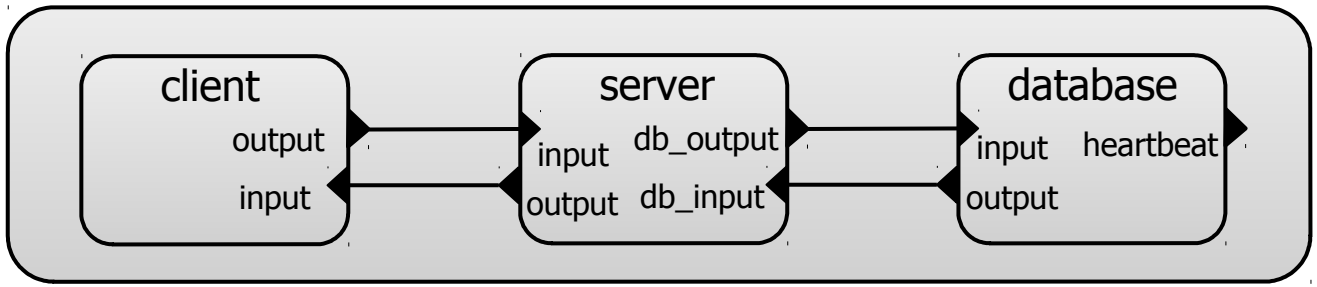
Component implementation name

Subcomponent name

Subcomponent implementation

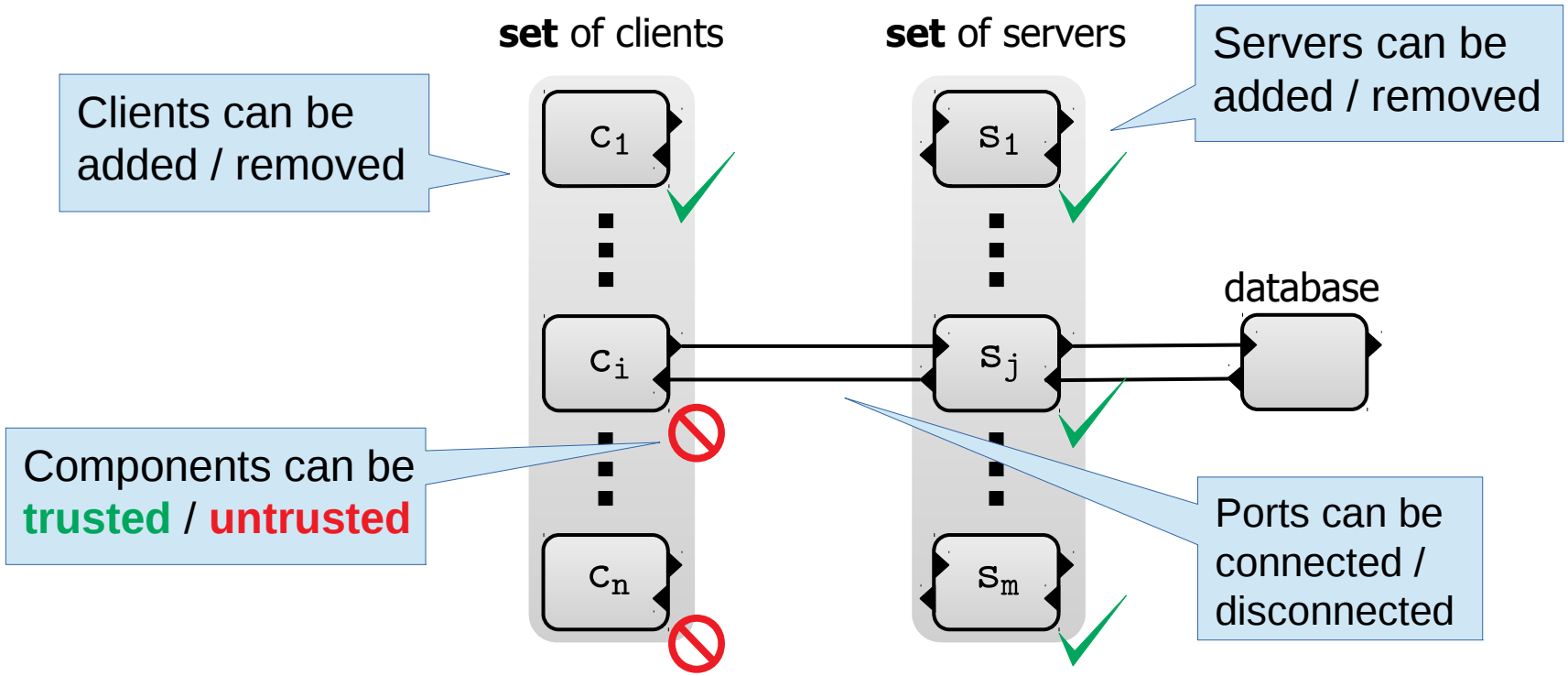
Port connection

system implementation sys.impl



Modeling dynamic architectures?

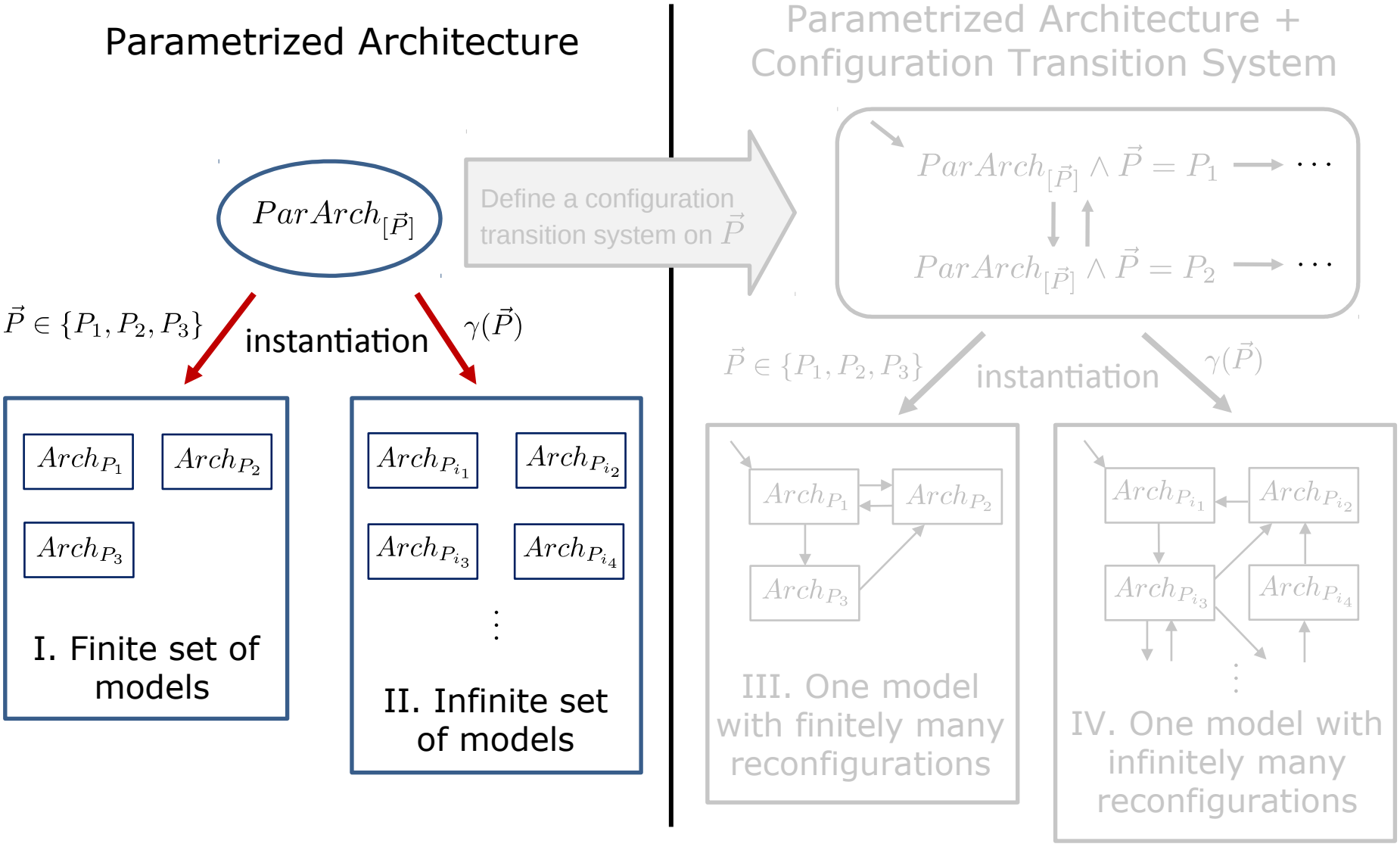
- For Adaptive MILS we want to model:
  - dynamic sets of components
  - dynamic connections
  - additional data associated with components/connections



CITADEL Modeling Language

- Based on AADL/SLIM
  - SLIM (System-Level Integrated Modeling language) is an extension of AADL
    - Nominal component behaviour (hybrid automata)
    - Error behaviour (probabilistic)
- CITADEL Modeling Language features:
  - Parametrized system architecture
  - Architectural reconfigurations
  - Component types and implementations
  - Component behaviour
  - Properties associated with model elements

Modeling language and its semantics



Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject databaseServer.impl;
        clients: set indexed by C of subject client.impl;
        servers: set indexed by S of subject applicationServer.impl;
    connections
        port database.output -> servers[s].db_input if trustedServers[s] for s in S;
        port servers[s].db_output -> database.input if trustedServers[s] for s in S;
        port servers[s].output -> clients[c].input if s = connectedTo[c] for s in S, c in C;
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject database
        clients: set indexed by C of
        servers: set indexed by S o
    connections
        port database.output -> serv
        port servers[s].db_output ->
        port servers[s].output -> clie
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

- Parameters can be of
  - simple types: **index**, **bool**, **int**, **real**, **enum**(id_1, …, id_n)
  - set types: **set of** <simple_type>
  - indexed set type: **set indexed by** <index set> **of** <simple_type, set_type>

Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject database
        clients: set indexed by C of
        servers: set indexed by S o
    connections
        port database.output -> serv
        port servers[s].db_output ->
        port servers[s].output -> clie
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Value of C is a set $\{c_1, c_2, \ldots, c_n\}$

- Parameters can be of
  - simple types: **index**, **bool**, **int**, **real**, **enum**(id_1, …, id_n)
  - set types: **set of** <simple_type>
  - indexed set type: **set indexed by** <index set> **of** <simple_type, set_type>

Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject database
        clients: set indexed by C of
        servers: set indexed by S o
    connections
        port database.output -> serv
        port servers[s].db_output ->
        port servers[s].output -> clie
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Value of C is a set $\{c_1, c_2, \ldots, c_n\}$

Set of Boolean values {trustedClients[c] : c in C}

- Parameters can be of
  - simple types: **index**, **bool**, **int**, **real**, **enum**(id_1, …, id_n)
  - set types: **set of** <simple_type>
  - indexed set type: **set indexed by** <index set> **of** <simple_type, set_type>

Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    sub  mponents
         base: subject databaseS
    connections
        port database.output -> serv
        port servers[s].db_output ->
        port servers[s].output -> clie
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Value of C is a set $\{c_1, c_2, \ldots, c_n\}$

Set of Boolean values {trustedClients[c] : c in C}

First-order logical formula over parameters

- Parameters can be of
  - simple types: **index**, **bool**, **int**, **real**, **enum**(id_1, …, id_n)
  - set types: **set of** <simple_type>
  - indexed set type: **set indexed by** <index set> **of** <simple_type, set_type>

Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject databaseServer.impl;
        clients: set indexed by C of subject client.impl;
        servers: set indexed by S of subject applicationServer.impl;
    connections
        port database.output -> servers[s].db_input if trustedServers[s] for s in S;
        port servers[s].db_output -> database.input if trustedServers[s] for s in S;
        port servers[s].output -> clients[c].input if s = connectedTo[c] for s in S, c in C;
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Indexed sets of subcomponents

Parametrized architecture

```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject databaseServer.impl;
        clients: set indexed by C of subject client.impl;
        servers: set indexed by S of subject applicationServer.impl;
    connections
        port database.output -> servers[s].db_input if trustedServers[s] for s in S;
        port servers[s].db_output -> database.input if trustedServers[s] for s in S;
        port servers[s].output -> clients[c].input if s = connectedTo[c] for s in S, c in C;
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Indexed sets of subcomponents

Specification of multiple connections

Parametrized architecture

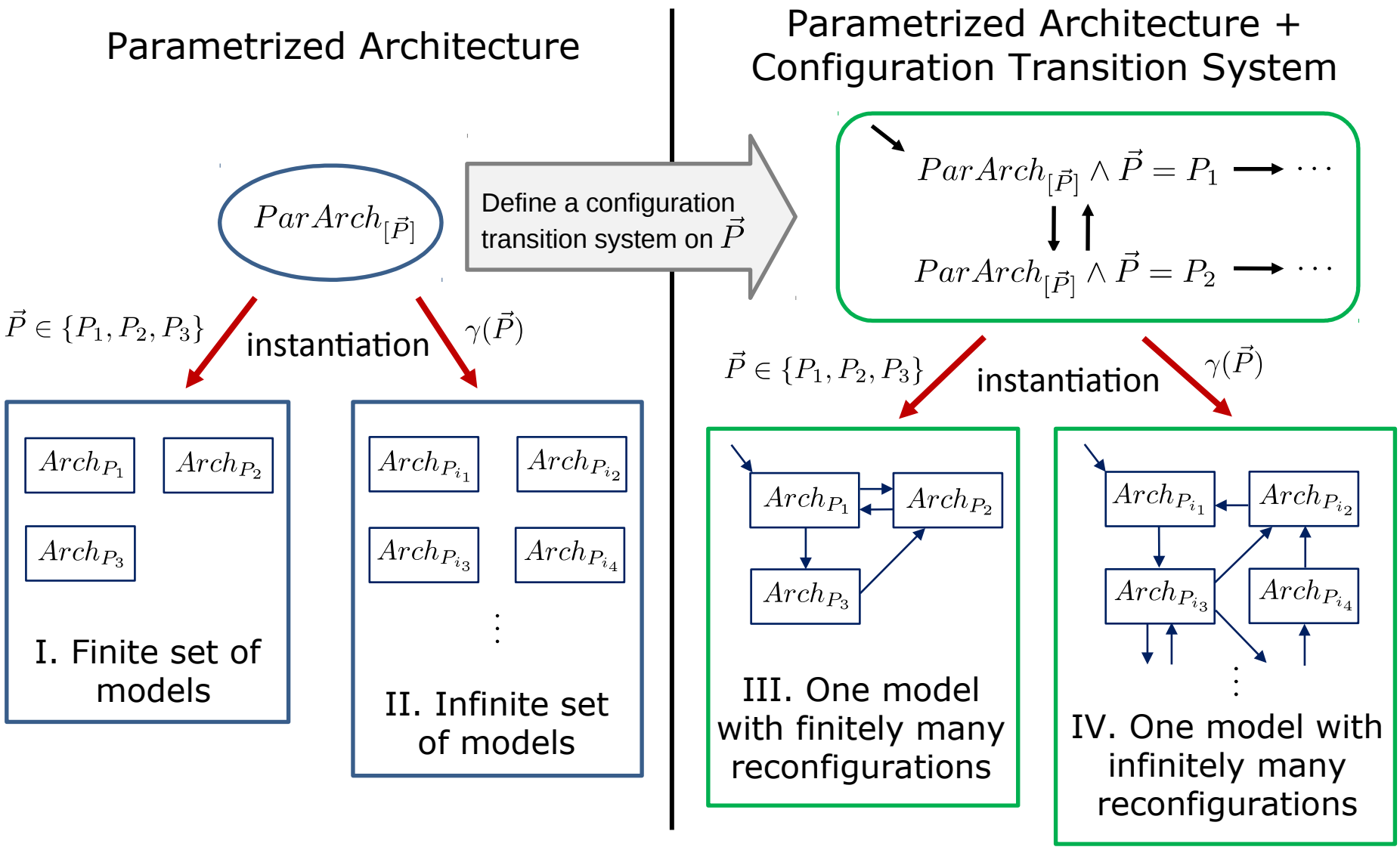
```
system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject databaseServer.impl;
        clients: set indexed by C of subject client.impl;
        servers: set indexed by S of subject applicationServer.impl;
    connections
        port database.output -> servers[s].db_input if trustedServers[s] for s in S;
        port servers[s].db_output -> database.input if trustedServers[s] for s in S;
        port servers[s].output -> clients[c].input if s = connectedTo[c] for s in S, c in C;
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
end sys.impl;
```

Indexed sets of subcomponents

Connection guard (first-order logical formula over parameters)

Specification of multiple connections

Modeling language and its semantics



Configuration transition system

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.trustedClients[c] and s = a.connectedTo[c])
                                            implies (not a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;

        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
end sys_cts;
```

Configuration transition system

Referenced parametrized architecture, with a label

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.trustedClients[c] and s = a.connectedTo[c])
                                            implies (not a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;

        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
end sys_cts;
```

Configuration transition system

Referenced parametrized architecture, with a label

First-order logical formula over parameters, defining the set of initial architectures

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.trustedClients[c] and s = a.connectedTo[c])
                                            implies (not a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;

        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
end sys_cts;
```

Configuration transition system

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.trustedClients[c] and s = a.connectedTo[c])
                                            implies (not a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;
        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
end sys_cts;
```

Transition label

Configuration transition system

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.trustedClients[c] and s = a.connectedTo[c])
                                            implies (not a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;

        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
end sys_cts;
```

Transition label

Specification of multiple transitions

Configuration transition system

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.tr...lients[c] and s = a.connectedTo[c])
                                            implies (n... a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;
        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
end sys_cts;
```

Formula specifying the transition step (functional dependency, implicitly includes frame condition)

Transition label

Specification of multiple transitions

Configuration transition system

```
CTS sys_cts
    architecture
        a: sys.impl;
    initial
        forall(c in a.C, forall (s in a.S, (not a.tr...lients[c] and s = a.connectedTo[c])
                                            implies (n... a.trustedServers[s])));
    transitions
        make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                                  for s in a.S;

        add_trusted_server[s]: step(next(a.S) = add(a.S, {s}) and
                                    next(a.trustedServers[s]) = true)
                               for s not in a.S;

        add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c}) and
                                         next(a.connectedTo[c]) = s and
                                         next(a.trustedClients[c]) = false)
                                    when (not a.trustedServers[s])
                                    for c not in a.C, s in a.S;
```

Formula specifying the transition step (functional dependency, implicitly includes frame condition)

Transition label

Specification of multiple transitions

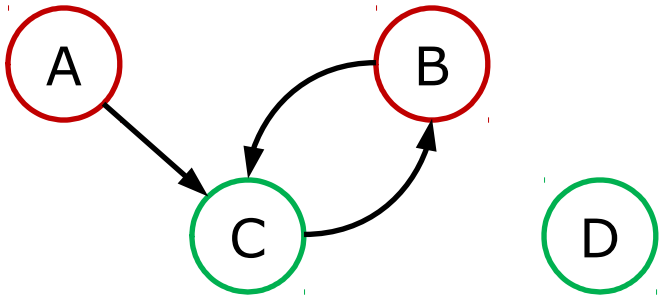
First-order formula over parameters specifying the transition guard (component states can be referenced)

Adaptive MILS

Verification

Information flows in classic MILS

- The classic MILS approach relies on strictly controlled information flows in order to enable compositional assurance of systems

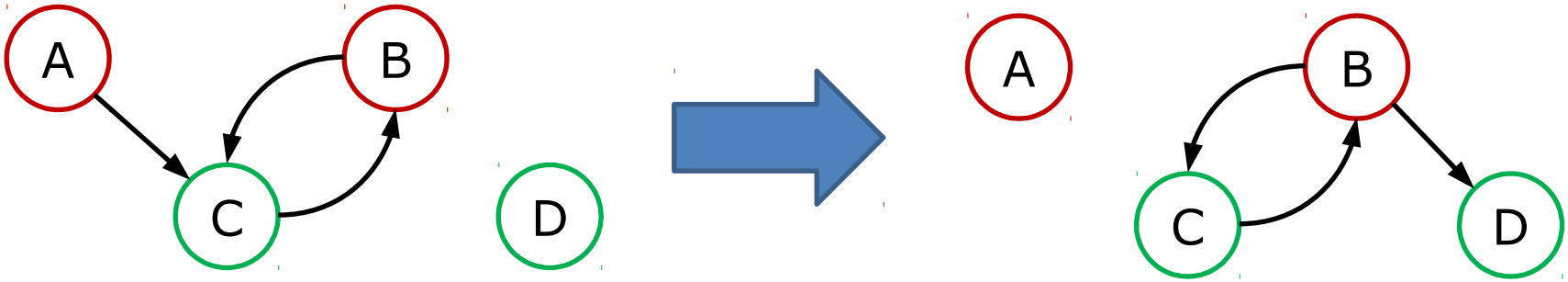


Static MILS policy architecture

- The problem: Decide whether information can flow from a source component to a destination component
- In classic MILS, the architecture is static and it is easy to verify information flow properties, such as “there is no information flow from A to D”

Adaptive MILS information flows

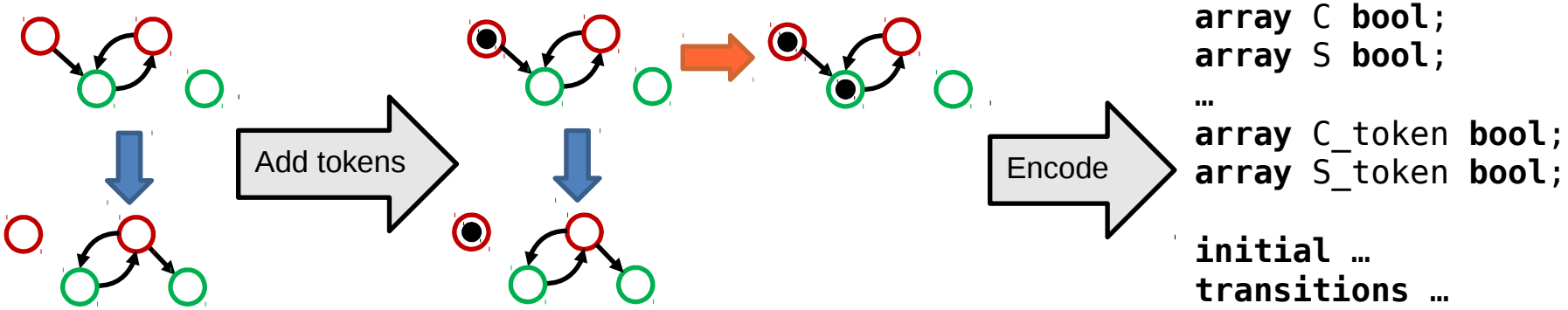
- In adaptive MILS, verifying information flow is more difficult, due to
  - Dynamic connections
  - Addition and removal of components
  - Potentially infinite number of architectural configurations (and unbounded number of variables)



Information can flow from A to D across architectural reconfigurations

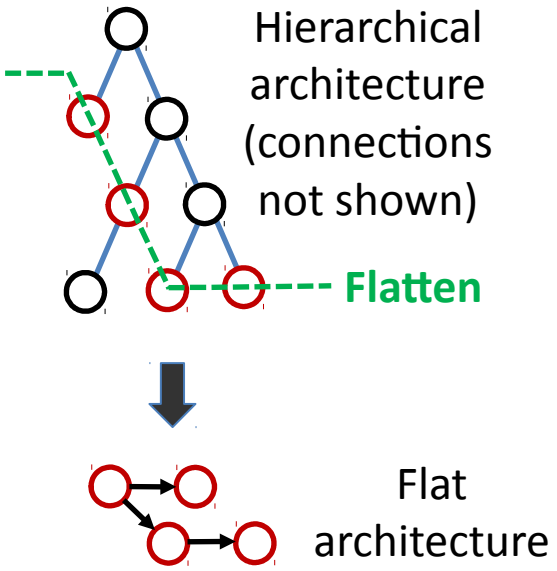
Information flow verification

- Approach
  - Consider a fragment of the CITADEL modeling language
    - No architecture hierarchy
    - No component behaviour
    - Some restrictions on formulas in PA and CTS
  - Encode the model and the information flow property using the theory of array, for model checker MCMT (users.mat.unimi.it/users/ghilardi/mcmt/)



Verification results

- Results
  - We were able to specify and automatically verify several non-trivial examples of dynamic architectures
  - The approach is feasible and promising
- Ongoing/Future work
  - Target other model checkers
    - CUBICLE (cubicle.lri.fr)
    - nuXmv (nuxmv.fbk.eu)
  - Generate proof certificates
  - Extend the approach
    - Hierarchical architectures
    - Trusted (filtering) components
    - Component behaviour
    - Checking of general properties
  - Evaluate on realistic problems

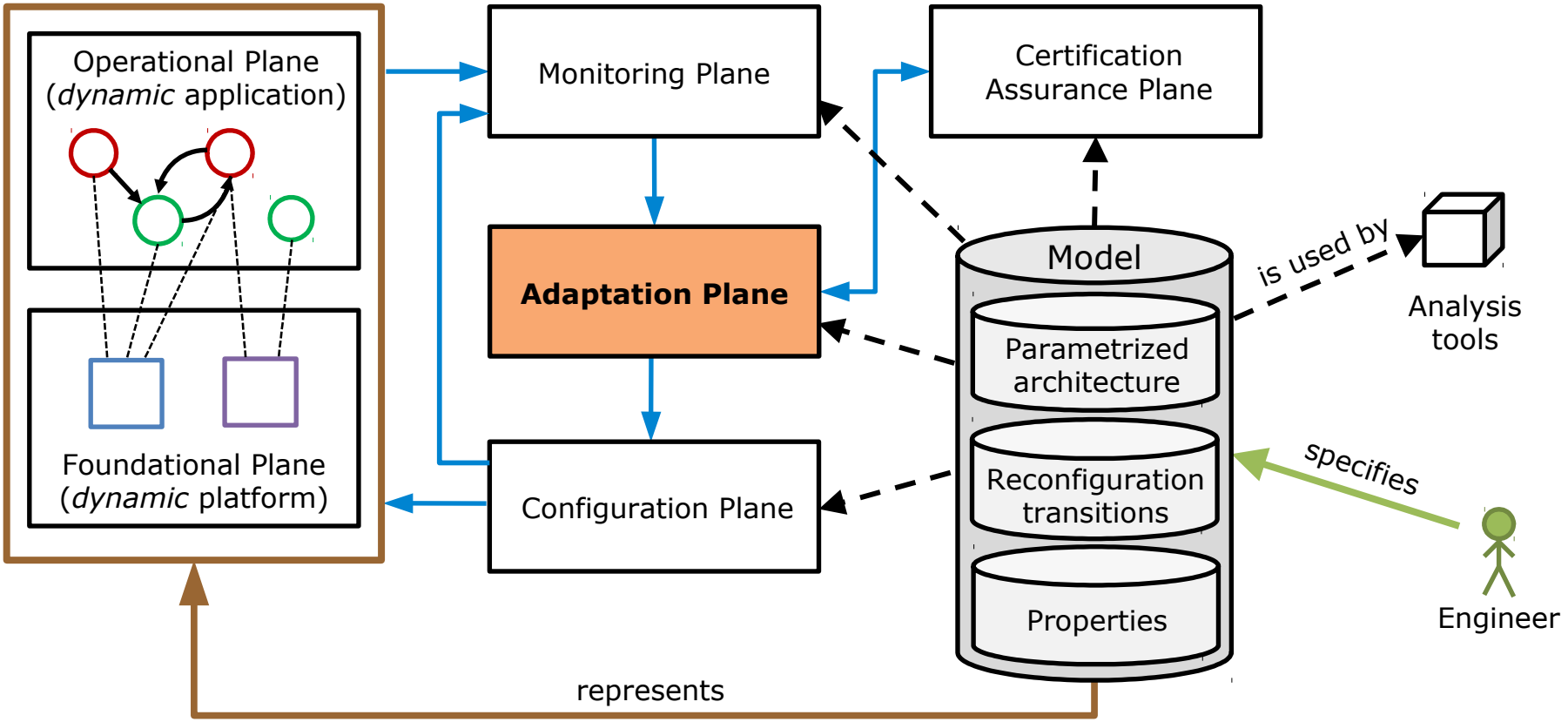


Adaptive MILS

Run-Time Adaptation

Adaptation Plane in the CITADEL Framework

- The Adaptation Plane receives alarms from the Monitoring Plane, decides on the next architectural configuration, synthesizes its model and sends it to the Configuration Plane.



Adaptation Plane

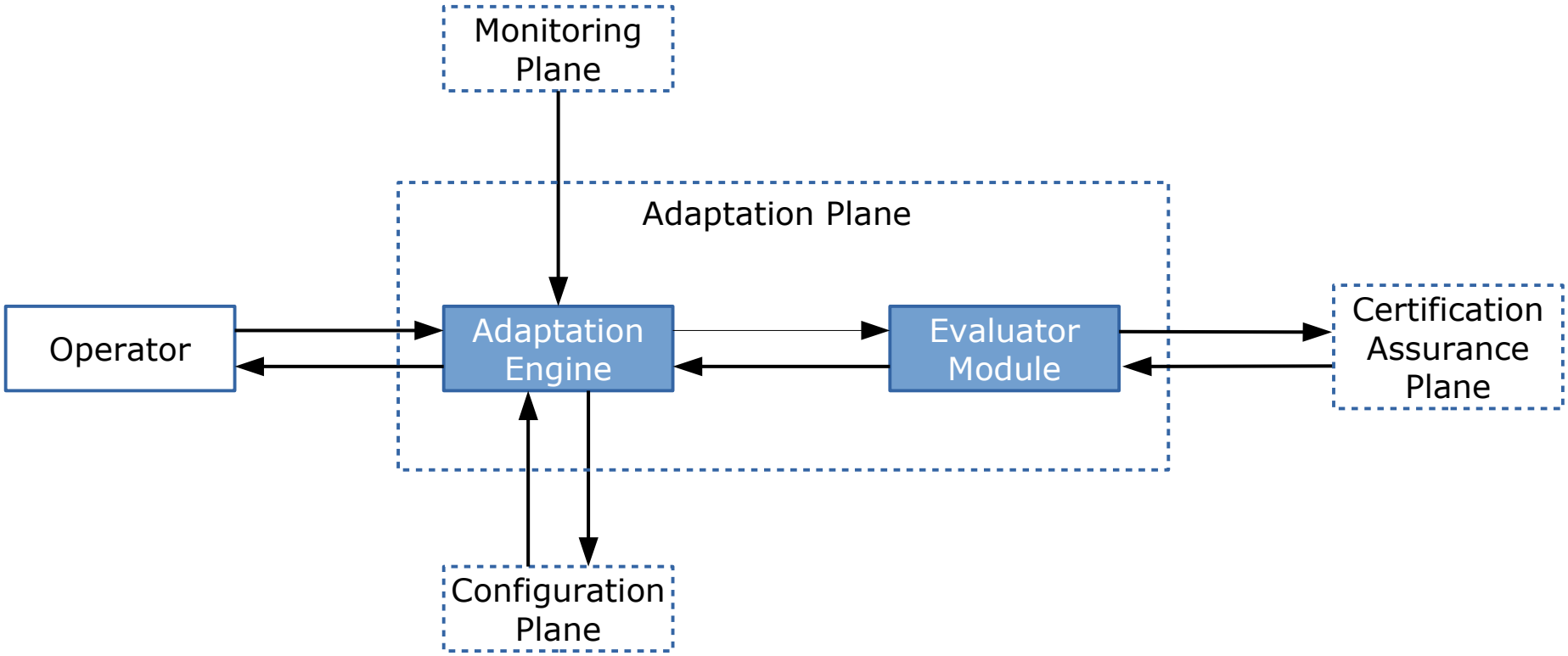
- Purpose
  - Listen to alarms from the Monitoring Plane
  - Decide the next architectural configuration
  - Communicate it to the Configuration Plane and to the Certification Assurance Plane
- Alarms and architectural reconfigurations are specified by the designer in the system model
- Next architectural reconfiguration is decided based on a reconfiguration strategy
  - Specified in the reconfiguration rule table
  - Maps alarms to reconfiguration actions
  - A reconfiguration action decides the next architectural configuration
- The Strategy is implemented by the Adaptation Engine
- The evaluation/reasoning is performed by the Evaluator Module

Adaptation Plane subcomponents

- Adaptation Engine
  - Implements the reconfiguration strategy
  - Based on the reconfiguration rule table
  - Decides on adaptation actions, which may be
    - specific reconfigurations (CTS transitions)
    - reasoning-based adaptation
    - reconfiguration obtained from an Operator
- Evaluator Module
  - Checks and evaluates the adaptation actions generated by the Adaptation Engine
    - Performing the requested reconfiguration action may be impossible in the current circumstances
  - Computes the (instantiated) model of the next configuration of the system

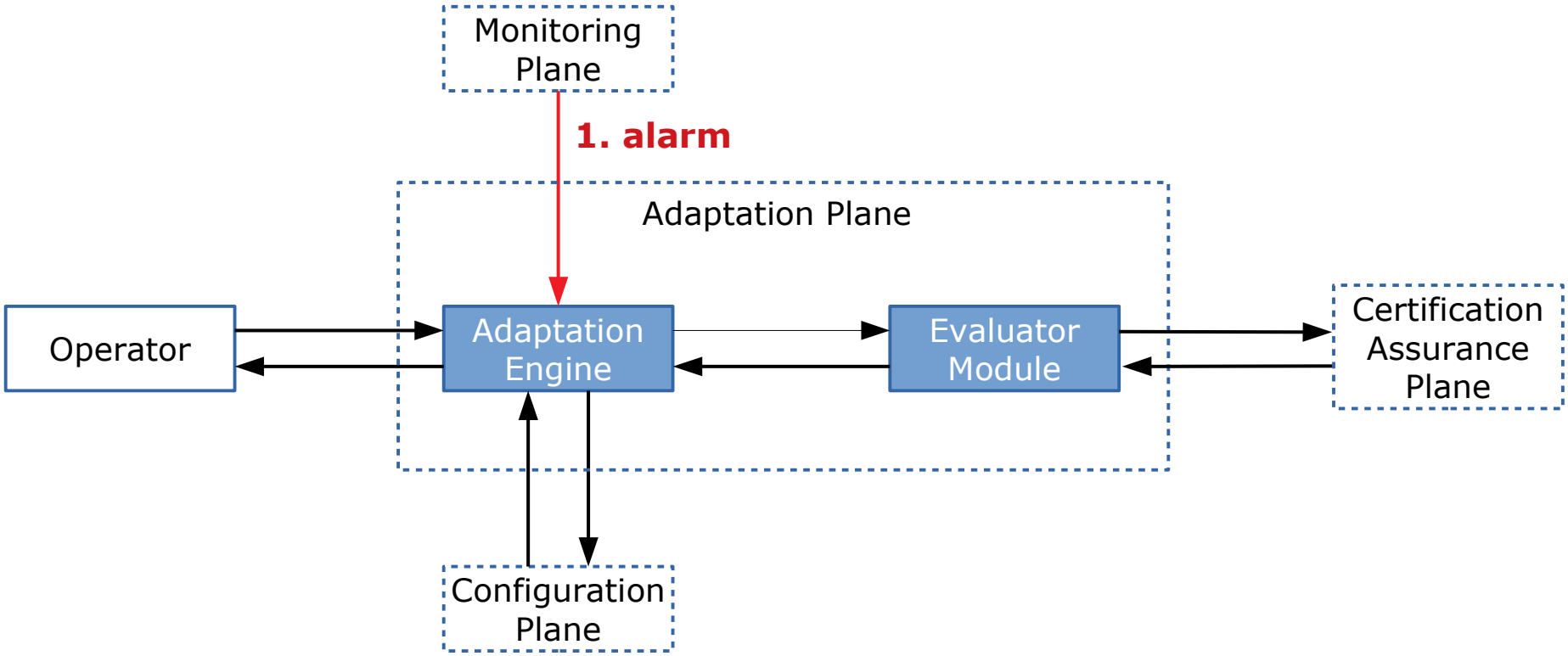
Adaptation Plane subcomponents

- Location of the Adaptation Engine and the Evaluator Module in the system:



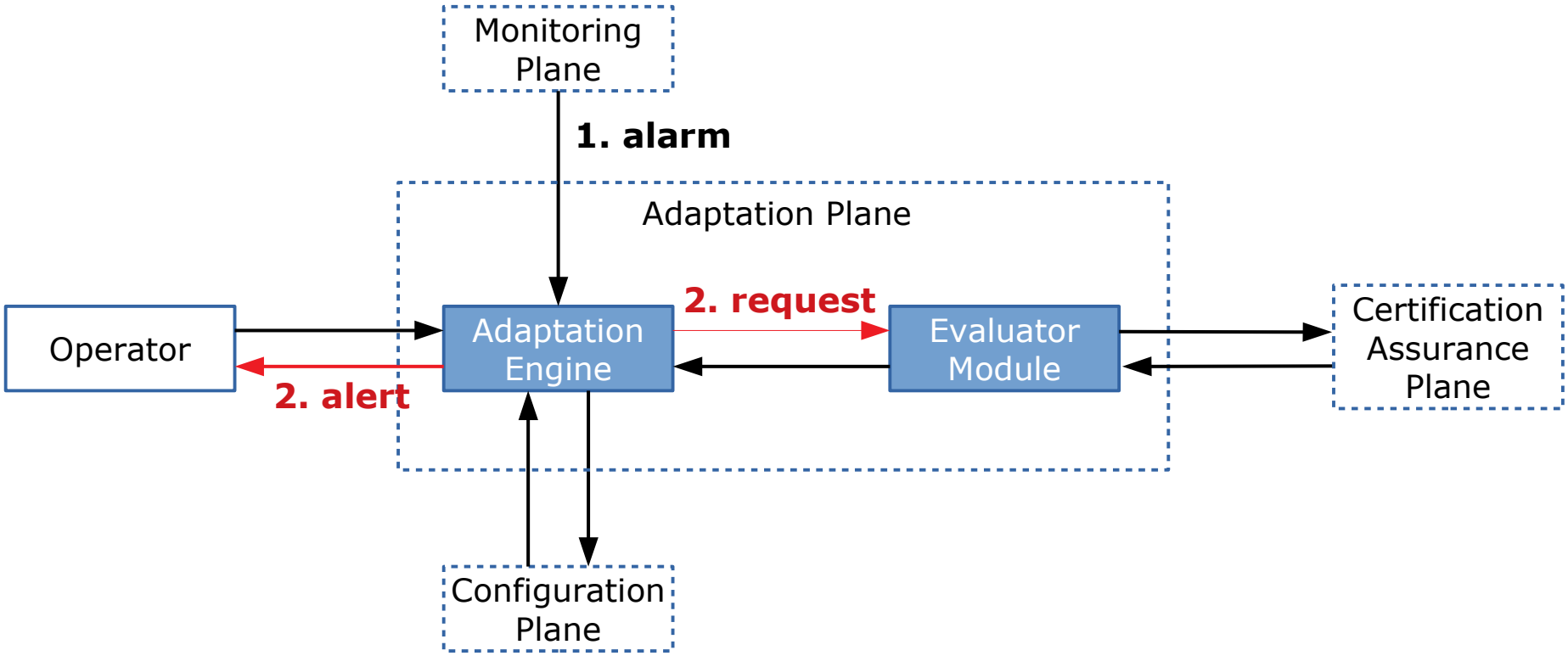
Nominal behaviour

- Monitoring Plane sends an alarm to the Adaptation Engine



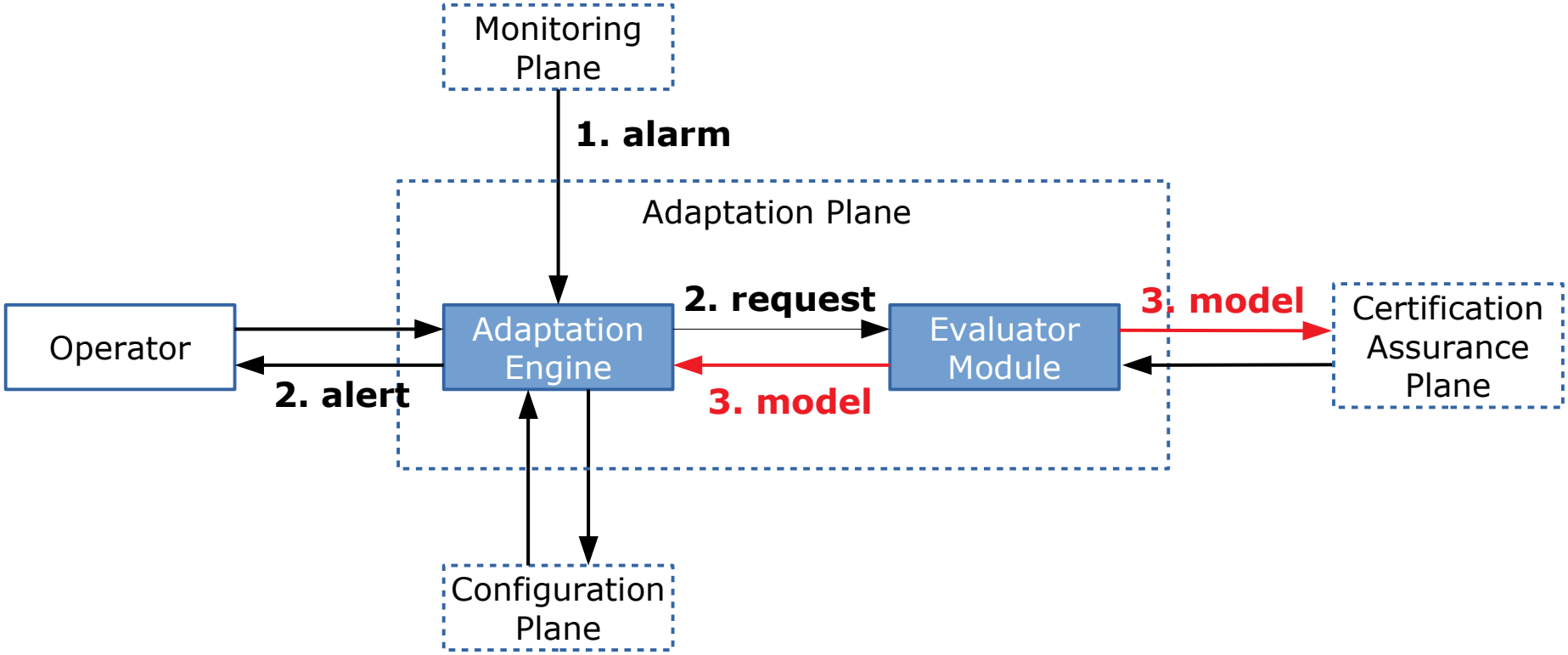
Nominal behaviour

- Adaptation Engine alerts the Operator, decides on the reconfiguration action and requests its evaluation from the Evaluator Module



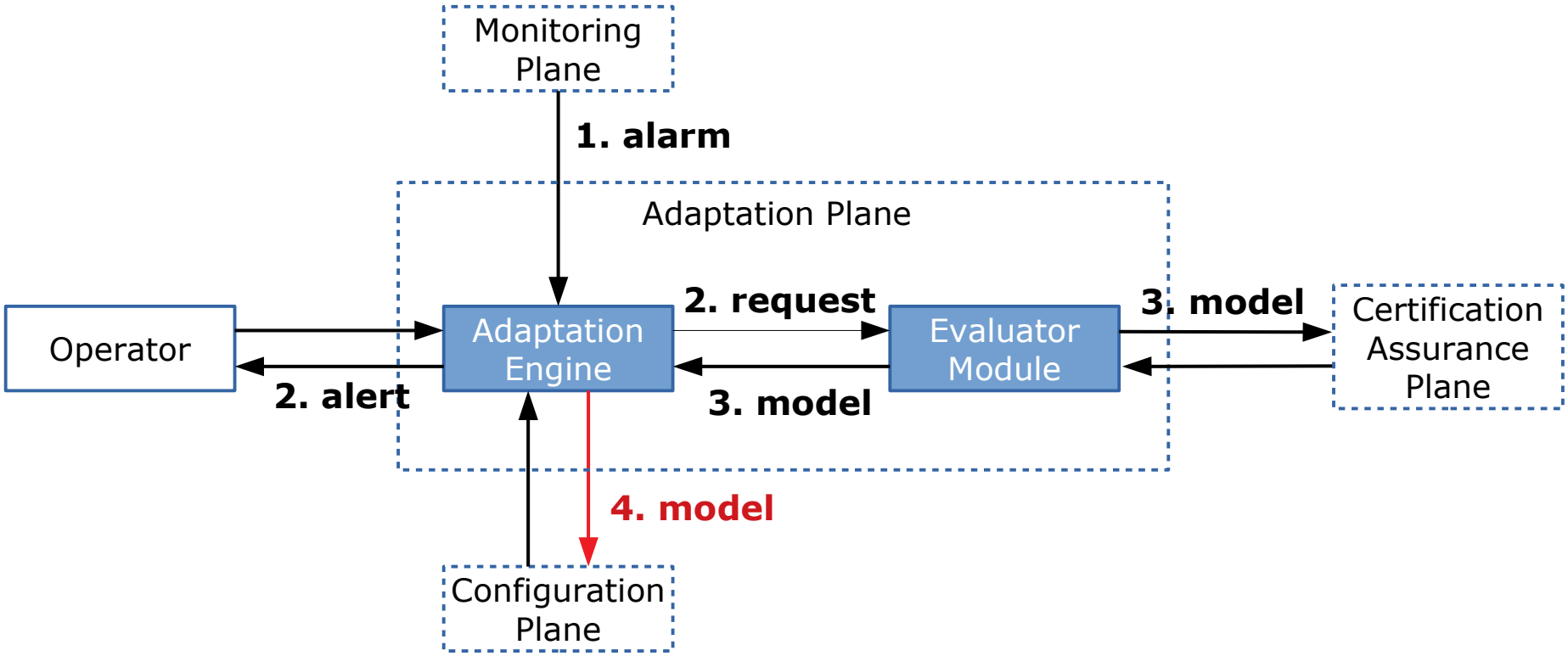
Nominal behaviour

- Evaluator Module computes the model of the next architectural configuration and sends it to the Adaptation Engine and the Certification Assurance Plane



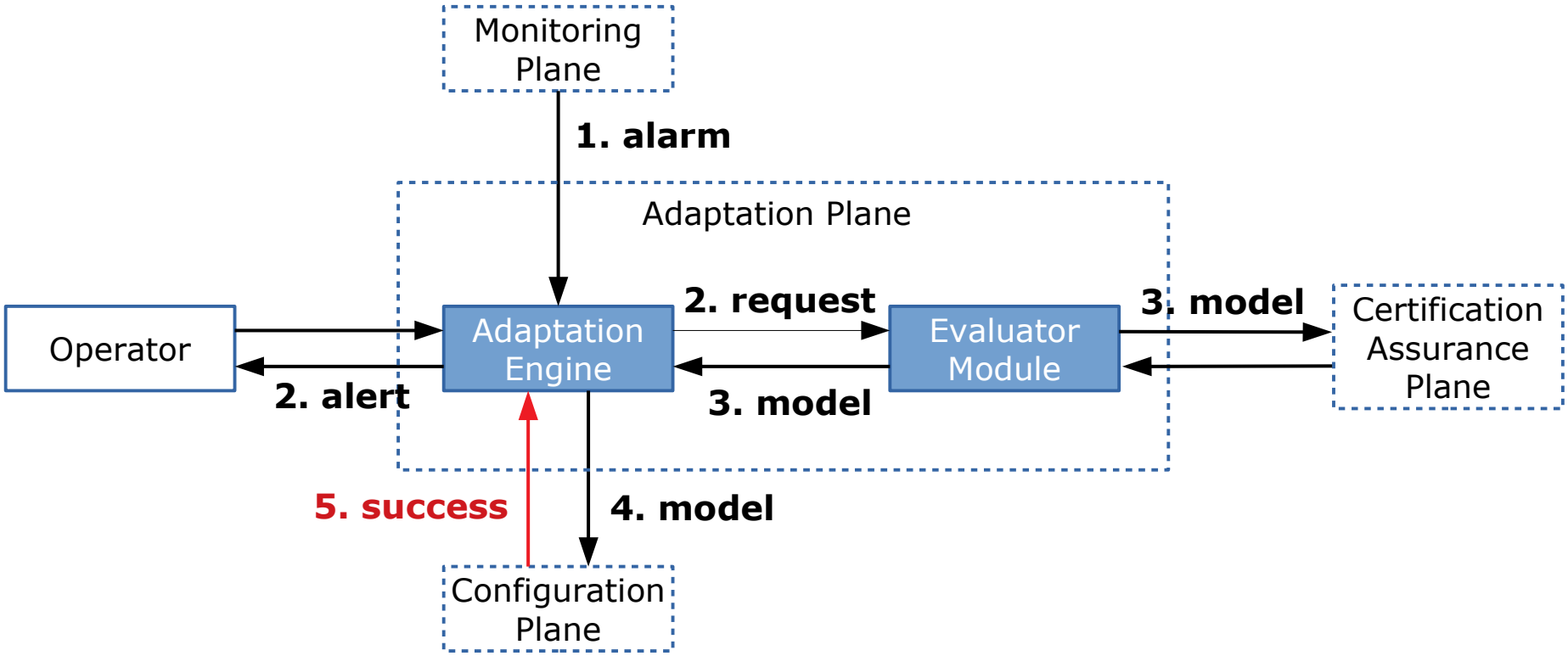
Nominal behaviour

- Adaptation Engine sends the model of the next architectural configuration to the Configuration Plane



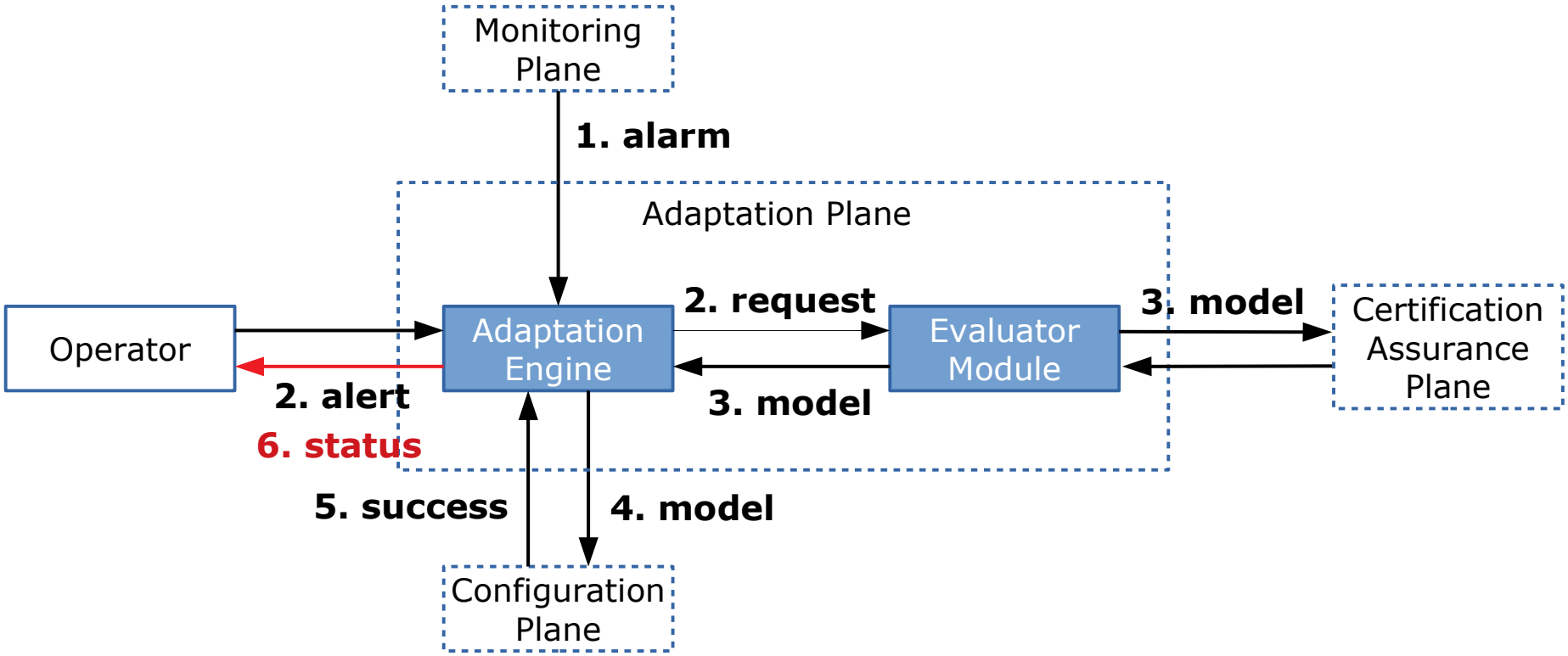
Nominal behaviour

- Configuration Plane reconfigures the system and sends status “success” to the Adaptation Engine



Nominal behaviour

- Operator is notified of the successful system reconfiguration



Reconfiguration rule table

Id	Alarm pattern	Action	Priority	Time limit
r1	database_failed	ask	10	300
r2	-	halt	11	-
r3	malicious_client(c)	make_client_untrusted[c]	0	0
r4	-	reason	1	15
r5	-	ask	2	-
r6	-	halt	3	-

- Rules are triggered by matching the incoming alarm with alarm patterns
  - Alarm malicious_client(1) triggers rule r3, yielding action make_client_untrusted[1]
- Rules without the alarm pattern are fallback rules for the rule above
  - Triggered on Evaluator Module evaluation failure

Priority, time limit, reconfiguration

- Priorities
  - While a rule is being processed
    - triggered rules of lower or equal priority are ignored
    - triggered rules of higher priority (or actions requested by the Operator actions) preempt the processing of the current rule
- Time limits
  - Specify the maximum amount of time within which the Evaluator Module must respond with a model or a failure
- Reconfiguration by the Configuration Plane
  - During reconfiguration, alarms are ignored
    - In this phase, the actual architecture is “outside the model” and the alarms cannot be interpreted
  - Reconfiguration failure is considered fatal; Adaptation Engine halts and dumps its state

Evaluator Module

- EM synthesizes the next architectural configuration so that it satisfies all assumptions on the parameters, and all safety and security properties specified in the model
- Synthesis modes:
  - Simple evaluation (automatic)
    - For a deterministic transition (e.g. make_client_untrusted[1])
    - EM checks the transition guard and computes the next values of parameters by evaluating the transition step expression
  - Parameter synthesis (automatic)
    - For a non-deterministic transition (e.g. add_untrusted_client[1][*], specifying addition of client 1 and its connection to any untrusted server)
    - EM utilises SMT-based techniques to synthesize the values of the unspecified indexes, and then performs the simple evaluation of the resulting deterministic transition

Evaluator Module

- Synthesis modes (cont.):
  - Reasoning (automatic)
    - For action “reason”
    - EM automatically selects a reconfiguration transition and synthesizes its indexes
    - EM attempts to minimize the difference between the current and next architectural configurations
  - Querying an engineer
    - For action “ask”
    - EM interactively queries an engineer who provides the next architectural configuration (i.e. the next values of parameters)

CITADEL

CRITICAL INFRASTRUCTURE PROTECTION
USING ADAPTIVE MILS
www.citadel-project.org

Thank you!

References

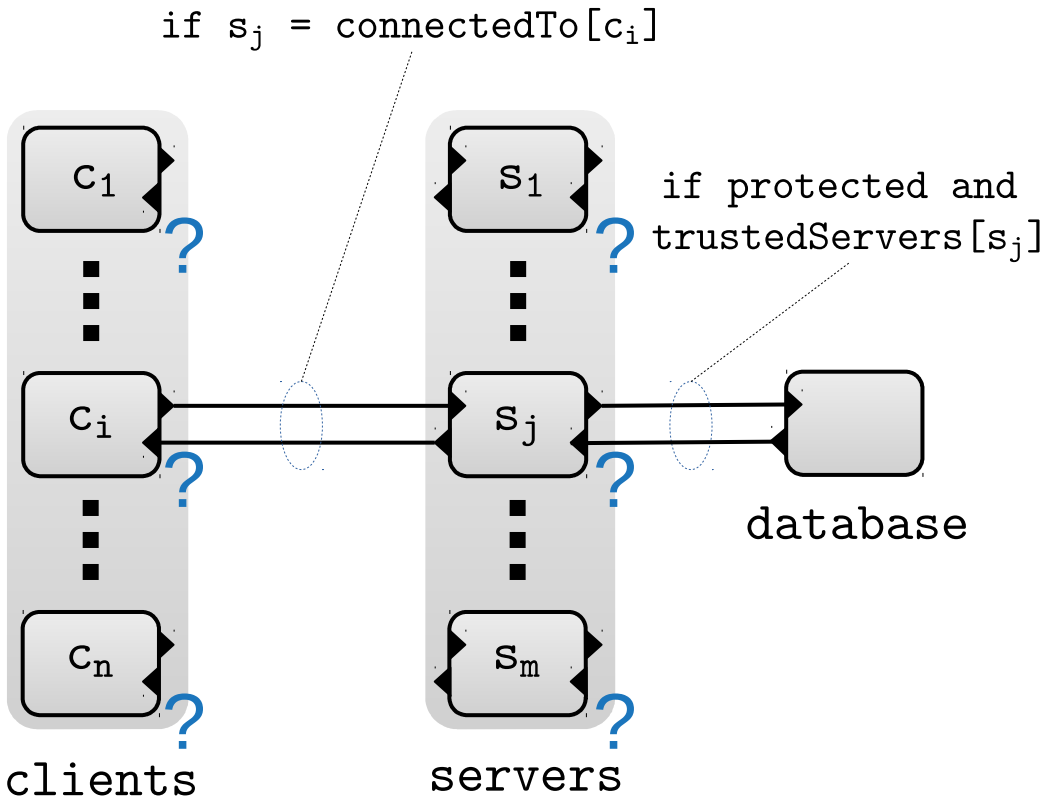
- *Architecture Analysis & Design Language (AADL) (rev. B)*. SAE Standard AS5506B, International Society of Automotive Engineers, Sept. 2012.
- P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012.
- *SLIM 3.0 - Syntax and Semantics*. Technical Note D1-2, Issue 4.7, COMPASS Project, June 2016.
- *CITADEL Modeling and Specification Languages*. Technical Report D3.1, Version 2.2, CITADEL Project, Apr. 2018.
- A. Cimatti, I. Stojic, and S. Tonetta. *Formal Specification and Verification of Dynamic Parametrized Architectures*. In FM 2018. Springer International Publishing, 2018, forthcoming.
- S. Ghilardi and S. Ranise. *MCMT: A Model Checker Modulo Theories*. In Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, pages 22–29, 2010.
- S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. *Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper*. In Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, pages 718–724, 2012.
- R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. *The nuXmv Symbolic Model Checker*. In CAV 2014. Springer International Publishing, 2014.

Example Model

- The system represents a network of computers, in which
  - there is a database that contains sensitive data,
  - there are servers which can connect to the database,
  - there are clients which connect to servers.
- The numbers of servers and clients are arbitrary, and more clients and servers can be added.
- Servers and clients are either *trusted* or *untrusted* to access the sensitive data which is stored in the database.
  - Trusted servers and clients can be compromised, becoming untrusted.

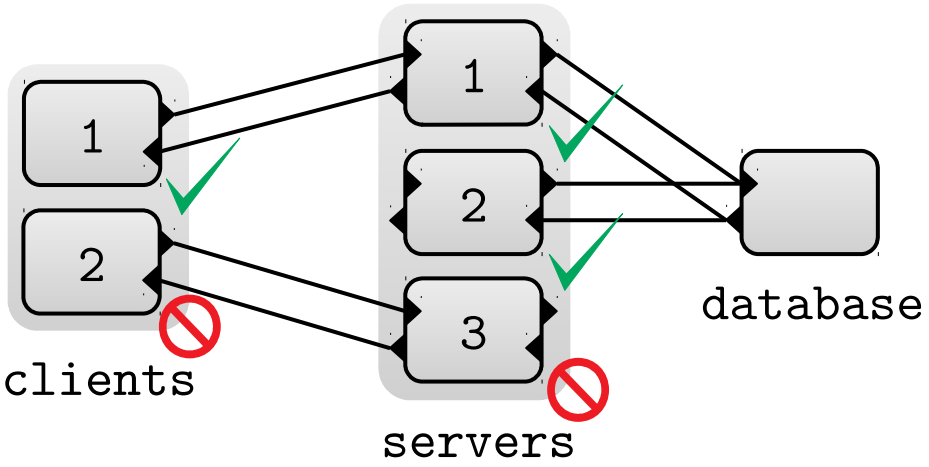
Example Model Parametrized Architecture

- Diagram of the Example Model parametrized architecture
  - monitors are not shown



Example Model instance

- Diagram of an instance of the Example Model, instantiated from the shown assignment to parameters
  - monitors are not shown




```
C = {1, 2}
S = {1, 2, 3}
trustedClients = {1:true, 2:false}
trustedServers = {1:true, 2:true, 3:false}
connectedTo = {1:1, 2:3}
protected = true
```

Example Model

- Required property is to prevent any leak of sensitive data from the database to the untrusted clients.
  - Verification of the model *without* the highlighted parts produces a counterexample, showing that this model is unsafe.
    - The counterexample: the sensitive data can 1) flow from the database to a server while it is trusted, then 2) a reconfiguration can happen making the server untrusted, after which 3) the data can flow to an untrusted client.
  - Verification proves that the model *with* the highlighted parts included is safe.
    - Highlighted parts introduce two phases (represented by the Boolean parameter "protected"): connections to the database are allowed only in the protected mode, while reconfigurations downgrading the servers are allowed only in the unprotected mode.

Example model listing (1/8)

```
package networkExampleModel

data sqlRequest
end sqlRequest;

data implementation sqlRequest.Data
end sqlRequest.Data;

data sqlResponse
end sqlResponse;

data implementation sqlResponse.Data
end sqlResponse.Data;

data message
end message;

data implementation message.Data
end message.Data;
```

Example model listing (2/8)

```
subject databaseServer
    features
        input: in event data port sqlRequest.Data;
        output: out event data port sqlResponse.Data;
        heartbeat: out event port;
end databaseServer;

subject implementation databaseServer.impl
end databaseServer.impl;

system heartbeatMonitor
    features
        heartbeat_in: in event port;
        database_failed: out event port {
            Alarm => true;
            MonitoringProperty =>
                "always (time_until(heartbeat_in) msec < HeartbeatTimeout)";
        };
    properties
        FDIR => true;
end heartbeatMonitor;
```

Example model listing (3/8)

```
system implementation heartbeatMonitor.impl
end heartbeatMonitor.impl;

subject applicationServer
   features
      db_input: in event data port sqlResponse.Data;
      db_output: out event data port sqlRequest.Data;
      input: in event data port message.Data;
      output: out event data port message.Data;
end applicationServer;

subject implementation applicationServer.impl
end applicationServer.impl;

subject client
   features
      input: in event data port message.Data;
      output: out event data port message.Data;
end client;
```

Example model listing (4/8)

```
subject implementation client.impl
end client.impl;

system clientMonitor
    parameters
        client_id: index;
    features
        client_out: in event data port message.Data;
        malicious_client: out event data port index {
            Alarm => true;
            MonitoringProperty => "never Malicious(last_data(client_out))";
            AlarmArguments => "client_id";
        };
    properties
        FDIR => true;
end clientMonitor;

system implementation clientMonitor.impl
end clientMonitor.impl;
```

Example model listing (5/8)

```
system sys
end sys;

system implementation sys.impl
    parameters
        C: set of index;
        S: set of index;
        trustedClients: set indexed by C of bool;
        trustedServers: set indexed by S of bool;
        connectedTo: set indexed by C of index;
        protected: bool;
    assumptions
        size(S) > 0;
    subcomponents
        database: subject databaseServer.impl;
        database_monitor: system heartbeatMonitor.impl;
        servers: set indexed by S of subject applicationServer.impl;
        clients: set indexed by C of subject client.impl;
        client_monitors: set indexed by C of system clientMonitor.impl
                        where forall(c in C, client_monitors[c].client_id = c);
```

Example model listing (6/8)

```
    connections
        port database.output -> servers[s].db_input if protected and trustedServers[s]
            for s in S;
        port servers[s].db_output -> database.input if protected and trustedServers[s]
            for s in S;
        port database.heartbeat -> database_monitor.heartbeat_in;
        port servers[s].output -> clients[c].input if s = connectedTo[c] for s in S, c in C;
        port clients[c].output -> servers[s].input if s = connectedTo[c] for s in S, c in C;
        port clients[c].output -> client_monitors[c].client_out for c in C;
end sys.impl;

CTS sys_cts
    architecture
        a: sys.impl;
    initial
        not a.protected and
        forall(c in a.C, forall (s in a.S, (not a.trustedClients[c] and s = a.connectedTo[c])
                                            implies (not a.trustedServers[s])))
        and forall(c in a.C, forall (s not in a.S, s != a.connectedTo[c]));
```

Example model listing (7/8)

```
transitions
    protect: step(next(a.protected) = true);

    add_trusted_server[s]: step(next(a.S) = add(a.S, {s})
                                and next(a.trustedServers[s]) = true)
                           for s not in a.S;

    make_server_untrusted[s]: step(next(a.trustedServers[s]) = false)
                              when (not a.protected)
                              for s in a.S;

    add_untrusted_client[c][s]: step(next(a.C) = add(a.C, {c})
                                    and next(a.connectedTo[c]) = s
                                    and next(a.trustedClients[c]) = false)
                                when (not a.trustedServers[s])
                                for c not in a.C, s in a.S;

    add_trusted_client[c][s]: step(next(a.C) = add(a.C, {c})
                                   and next(a.connectedTo[c]) = s
                                   and next(a.trustedClients[c]) = true)
                              when (a.trustedServers[s])
                              for c not in a.C, s in a.S;
```

Example model listing (8/8)

```
        make_client_untrusted[c][s]: step(next(a.trustedClients[c]) = false
                                          and next(a.trustedServers[s]) = false)
                                     when (a.trustedClients[c]
                                           and s = a.connectedTo[c]
                                           and not a.protected)
                                     for c in C, s in S;
end sys_cts;

properties
    Constants => "Malicious: function message.Data -> bool;
                  HeartbeatTimeout: clock msec := 10 msec;";

end networkExampleModel;
```

End