UNIVERSITÀ DEGLI STUDI DI UDINE

Finanziato dall'Unione europea NextGenerationEU

Ministero dell'Università e della Ricerca

Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA

Compositional Bigraphical Models for Container-Based Systems Security

Marino Miculan

DMIF, University of Udine - SERICS Spoke 4
marino.miculan@uniud.it

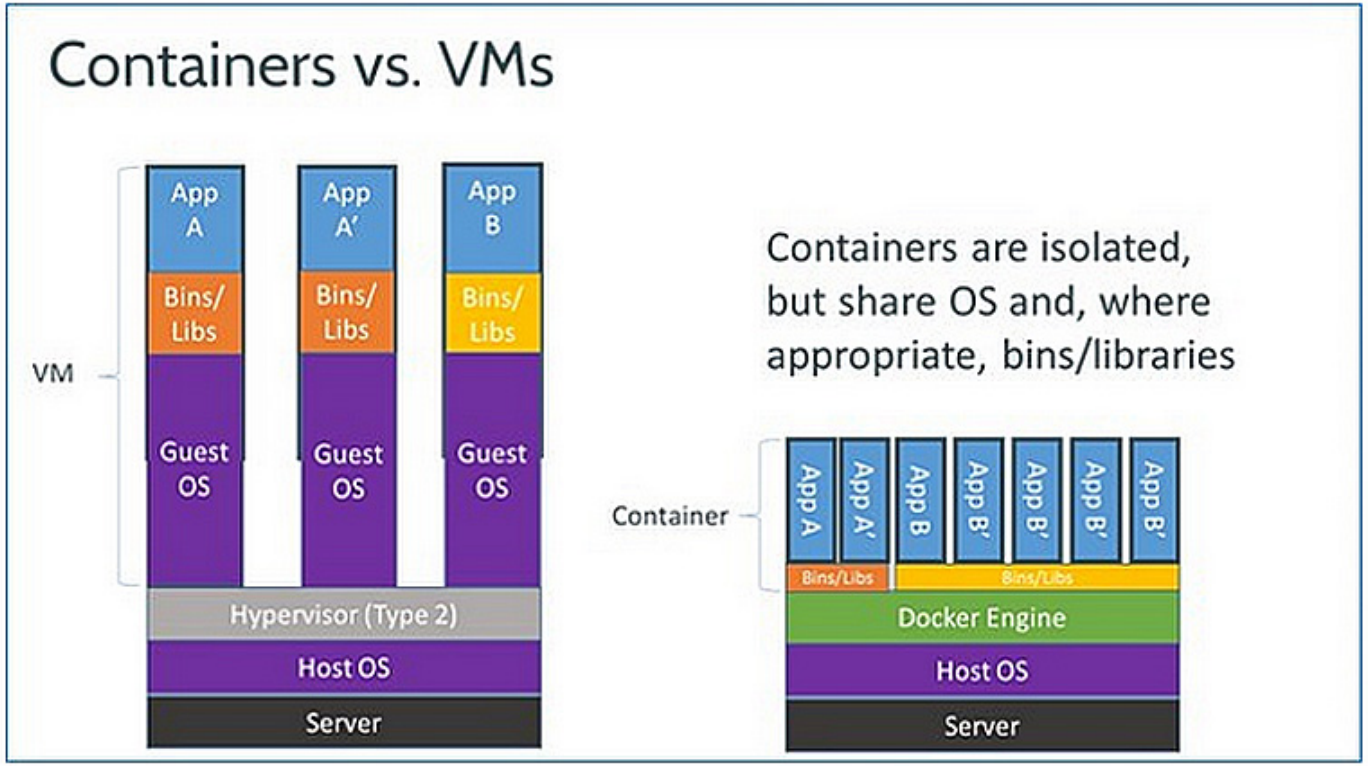
NeCS Winter School, Cortina d'Ampezzo
January 21, 2025

Microservice-oriented architectures…

- **Microservice-oriented architecture**
  - Modern applications are built by composing **microservices** through **interfaces**
  - Distributed, component-based
  - Flexible, scalable, supporting dynamic deployment and reconfiguration, agile programming, etc.

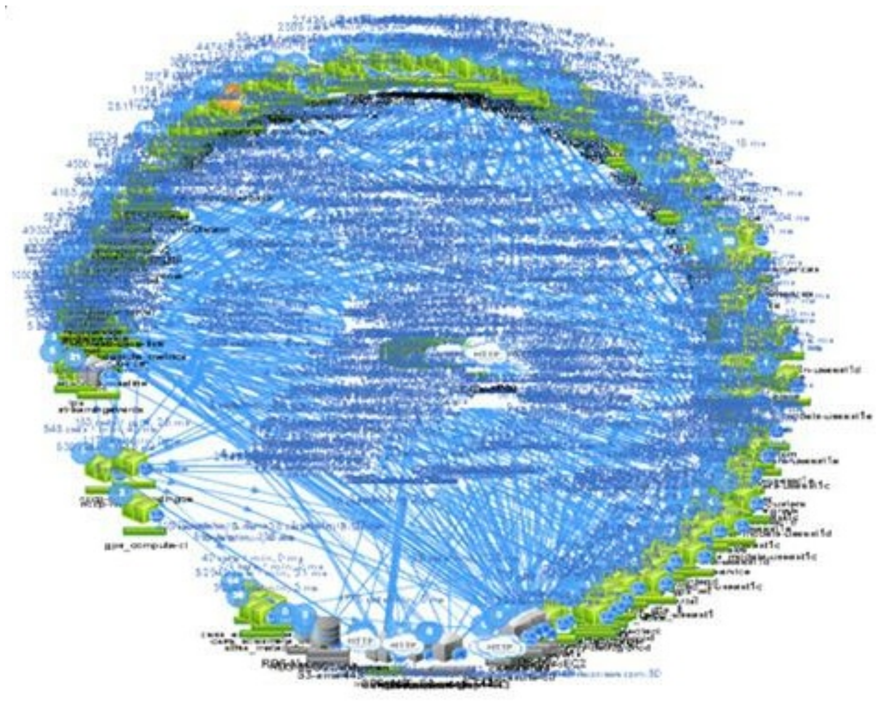
… and containers

- **Containers** are a lighter, more efficient alternative to Virtual Machines
- Ensure execution separation leveraging kernel namespaces and cgroups in the host OS
- Containers offer:
    - Fine granularity services and components
    - Clear definition of **interfaces**
    - Support for service and component **composition**
    - Simpler horizontal and vertical scalability
- Widely used for Microservice-oriented Architectures, especially in the Cloud



Containers enforce weaker separation than VMs

- Applications can be composed by hundreds or thousands of containers
- A cloud provider often runs many applications (possibly from different clients) on the same infrastructure
- Connecting and coordinating containers into a complete working system is not trivial
- Violating security goals and policies through misconfigurations is easy

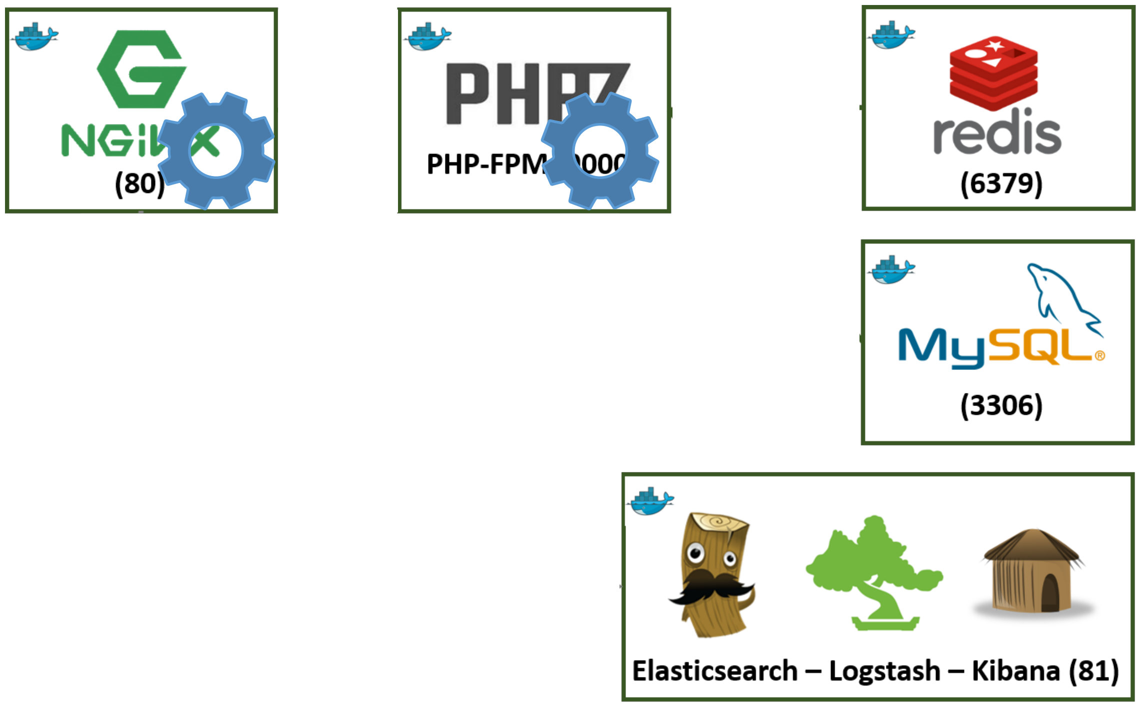


NETFLIX

Vertical vs Horizontal Composition

- Containers can be composed to form larger systems
- Two different compositions:
  - **Vertical***: containers can be filled with application specific code, processes… and containers can be put inside *pods*
  - **Horizontal***: containers are on a par, and communicate through channels (sockets, API), volumes, networks

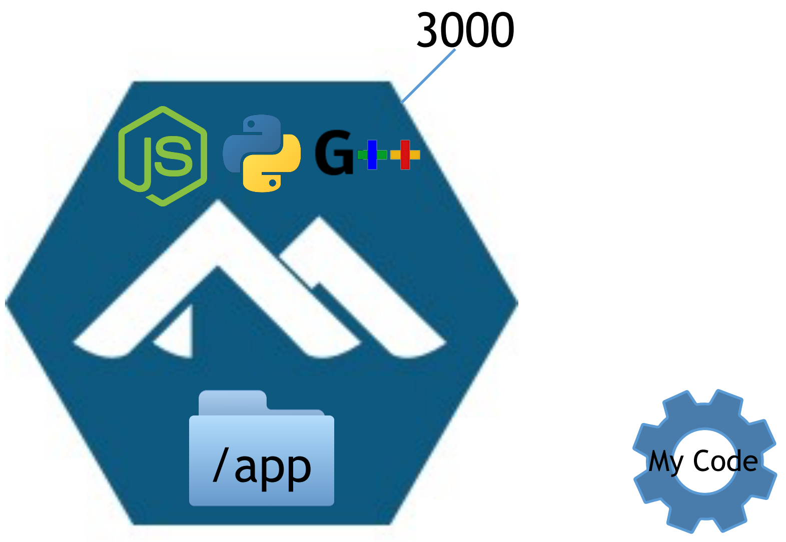
* = my naming, not official



Containers can be filled with libraries, code, data…

- **Dockerfiles**: recipes to build *images*. Example:
  - Start from an existing image
  - Run any command, e.g. to extend the image with any needed package
  - Install programmer's specific code
  - Define the entry point command (what to execute when the container is launched)
  - Declare exposed ports (interfaces)
- These recipes are fed to `docker build`
- Result: **a new image**, which can be run in a container, or used as basis for further builds
- (We will not discuss dockerfiles in this talk; see other work from SERICS Spoke 4)

```
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

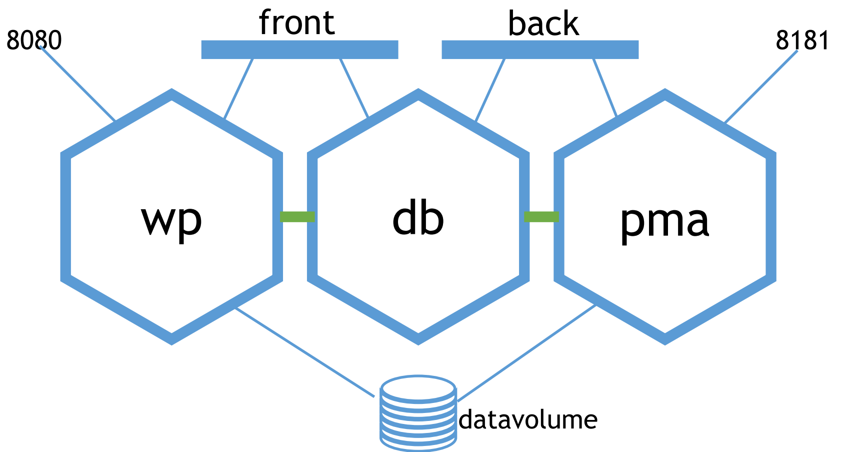


(Horizontal) Composition of containers

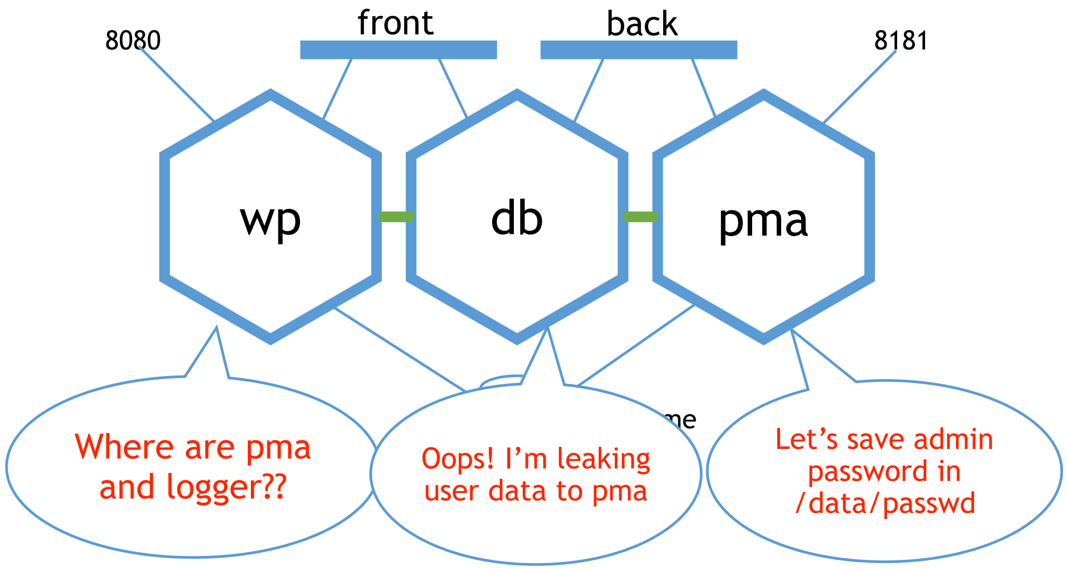
- Composition is defined by YAML files declaring
  - (Virtual) Networks
  - Volumes (possibly shared)
  - For each container
    - Name
    - Images
    - Networks which are connected to
    - Port remapping for exposed services
    - Volumes
    - Links between services
- Configuration file is fed to a tool (e.g., `docker compose`) which downloads images, creates containers, networks, connections, etc. and launches the system

```
services:
  wp:
    image: wordpress
    links:
      - db
    ports:
      - "8080:80"
    networks:
      - front
    volumes:
      - datavolume:/var/www/data:ro
  db:
    image: mariadb
    expose:
      - "3306"
    networks:
      - front
      - back
```

```
  pma:
    image: phpmyadmin/phpmyadmin
    links:
      - db:mysql
    ports:
      - "8181:80"
    volumes:
      - datavolume:/data
    networks:
      - back
networks:
  front:
    driver: bridge
  back:
    driver: bridge
volumes:
  datavolume:
    external: true
```



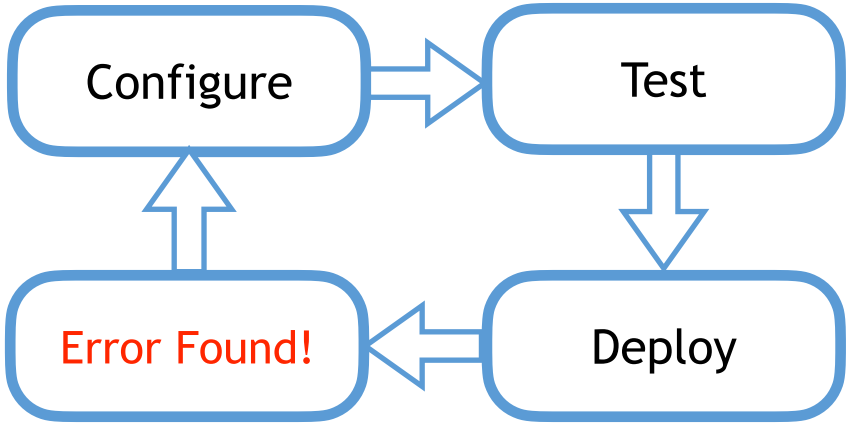
What if a composition configuration is not *correct*?



- A configuration may contain several errors, which may lead to problems during **composition**, or (worse) at **runtime**. E.g.:
  - A container may try to access a **missing services**, or a service which is not connected to by a network
  - **Security policies** violations, e.g. sharing networks or volumes which should not (or only in a controlled way) leading to information leaks
- **Dynamic reconfiguration** can break properties previously valid
  - Container's images can be updated at runtime (e.g. for bug fixing)
  - Adding or removing containers to an existing and running system

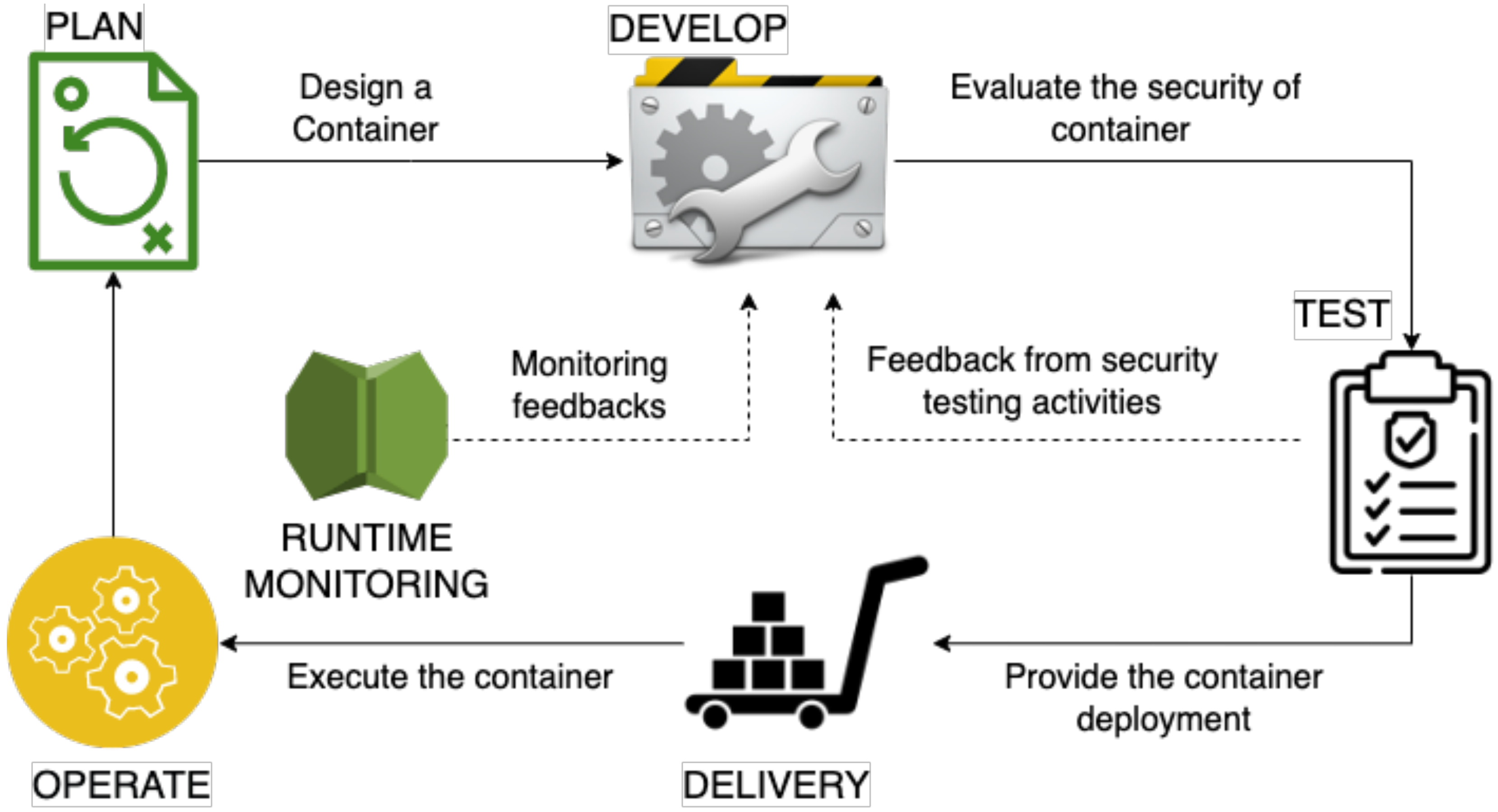
What if a composition configuration is not *correct*?

- Actual composition tools check only very basic aspects
- Common approach: *try-and-error*
    - Expensive
    - Slow
    - Not scalable
    - Not safe enough
    - Not acceptable in critical situations



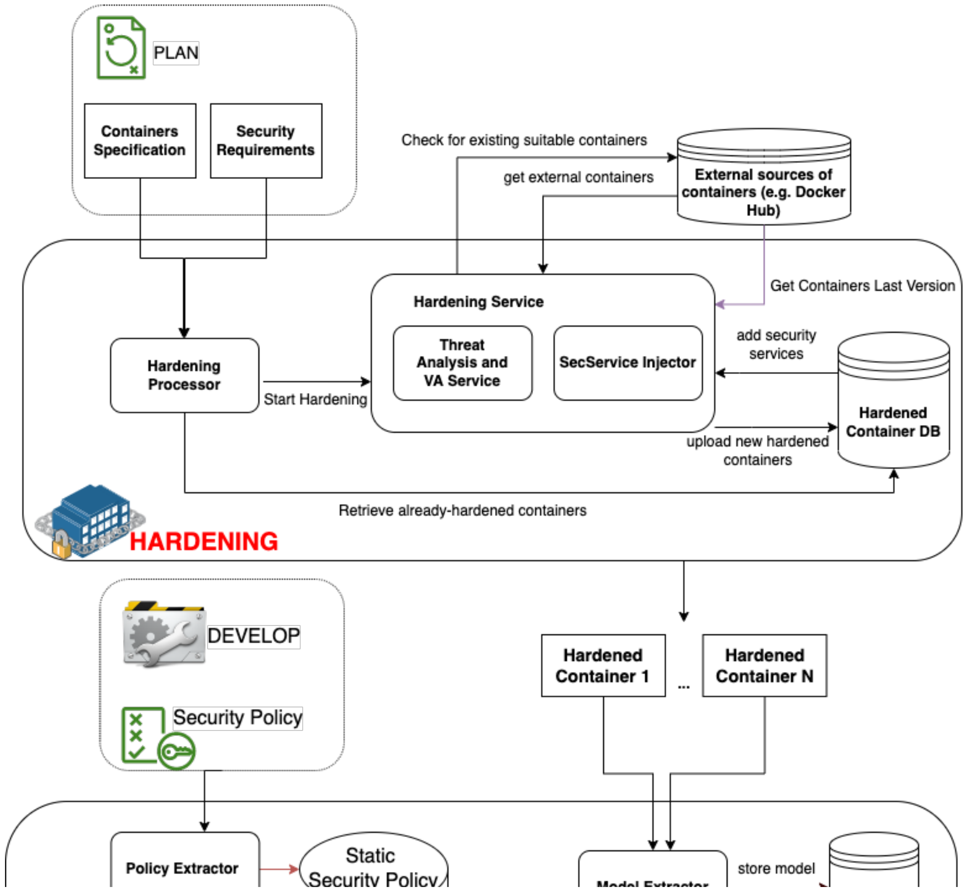
- We aim to analyze, verify (and possibly manipulate) container configurations **before** executing the system (static analysis) and/or at **runtime**

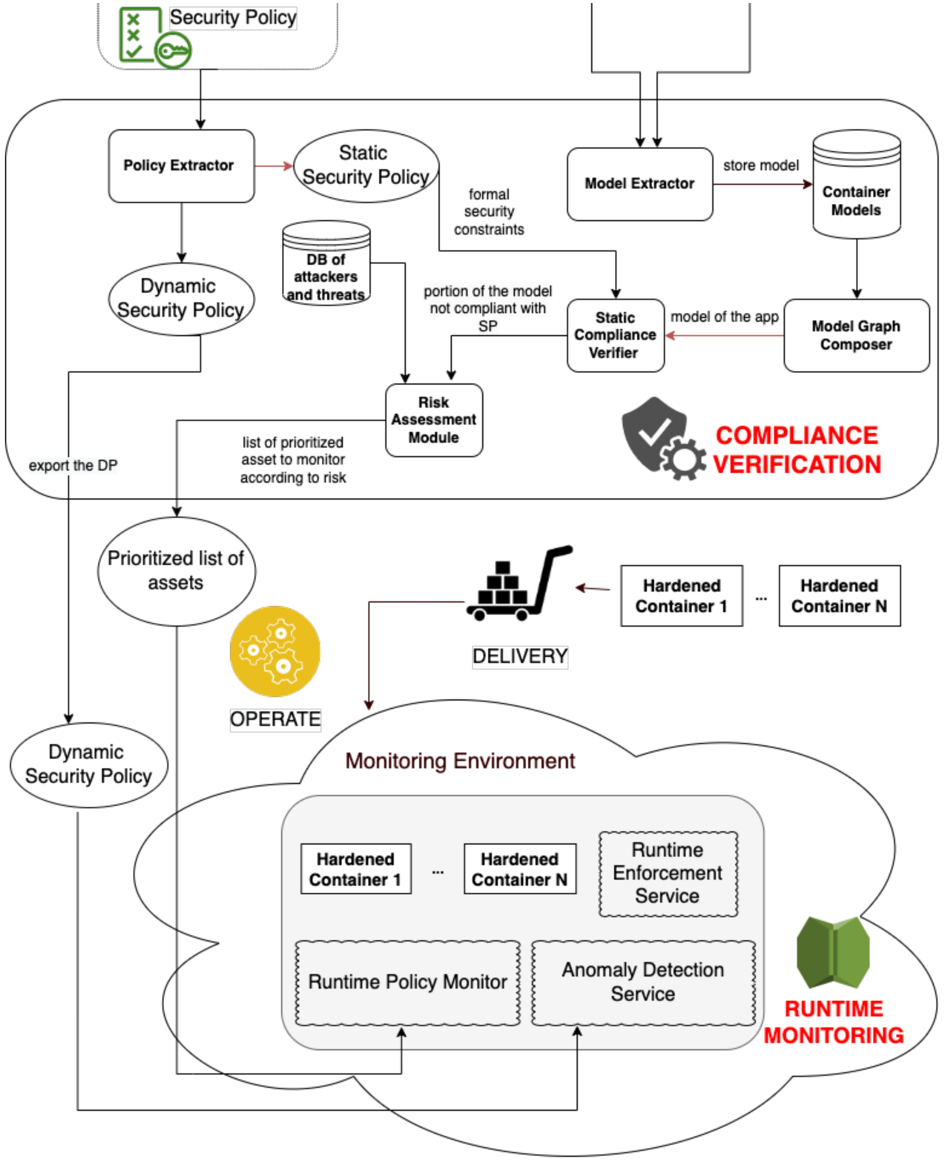
SECCO's DevSecOps scenario for cloud-native applications



Picture from (Verderame et al., 2023)

The SECCO project





Solid tools need solid theoretical foundations

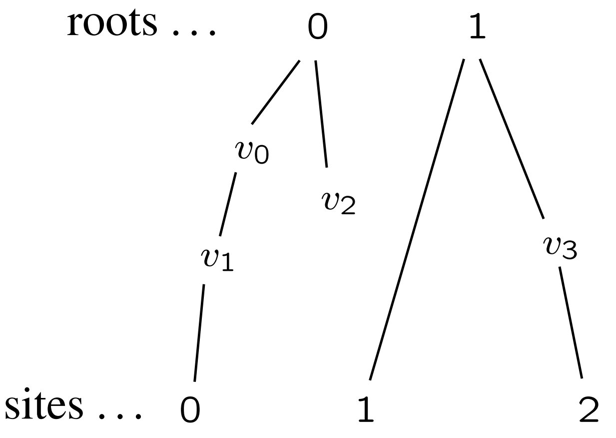
- We need **tools** for analyzing, verifying (and possibly manipulate) container configurations, before executing the system (static analysis), or at runtime
- We need a ***formal model of containers and services composition***
- This model should support:
  - Composition and nesting of components
  - Dynamic reconfiguration
  - Different granularities of representation
  - Flexibility (can be adapted to various aspects)
  - Openness (we may need to add more details afterwards)
  - …

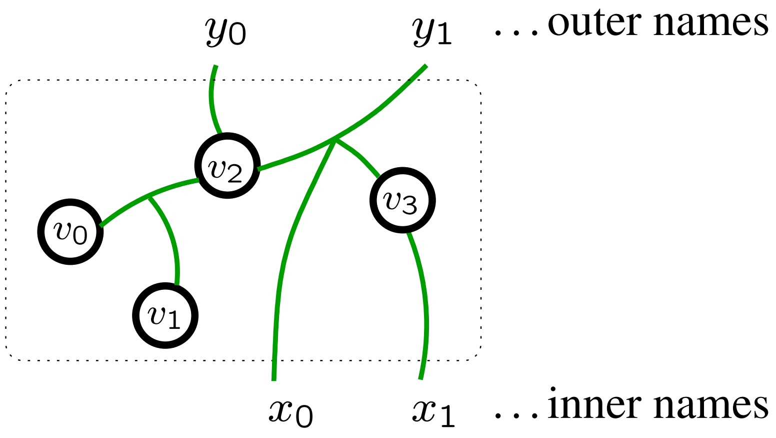
Bigraphs (Milner, 2003): "a general (meta)model for distributed communicating systems, supporting **composition** and **nesting**."

Quick intro to bigraphs

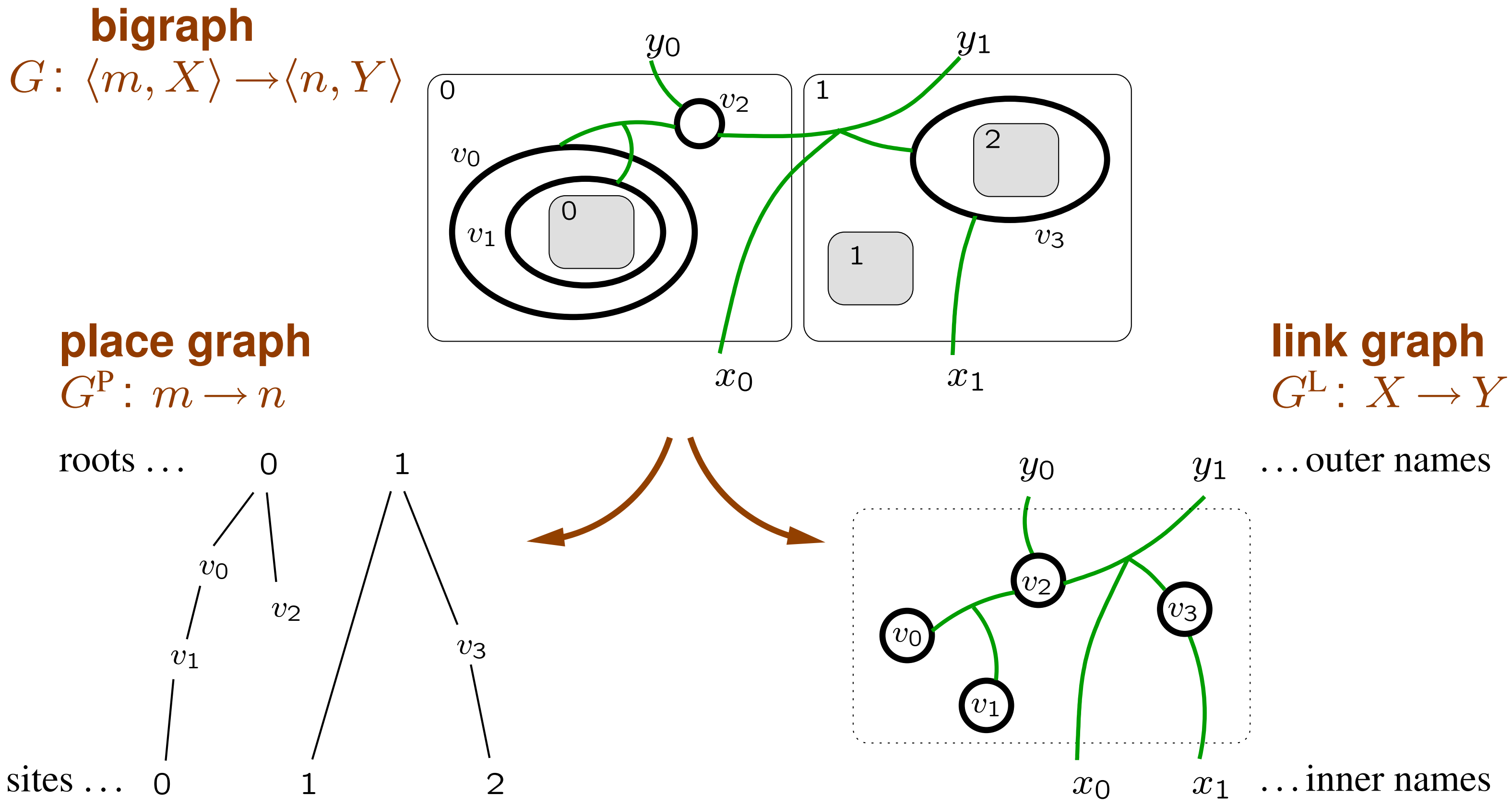
A bigraph combines two graph structures based on the same node set:

- **Place graph**: a *forest* describing the nesting of the nodes (the *mereology* of the system). Roots are *regions*, leaves can be nodes or *holes* (sites), where other bigraphs can be *grafted*
- **Link graph**: a *hypergraph* describing the *connectivity* of nodes. *Outer names* and *inner names*, represented as open links.
- Each node has a fixed number of connections (*ports*), according to a given *signature*. Node shapes are visually useful, but not formally meaningful.





Quick intro to bigraphs



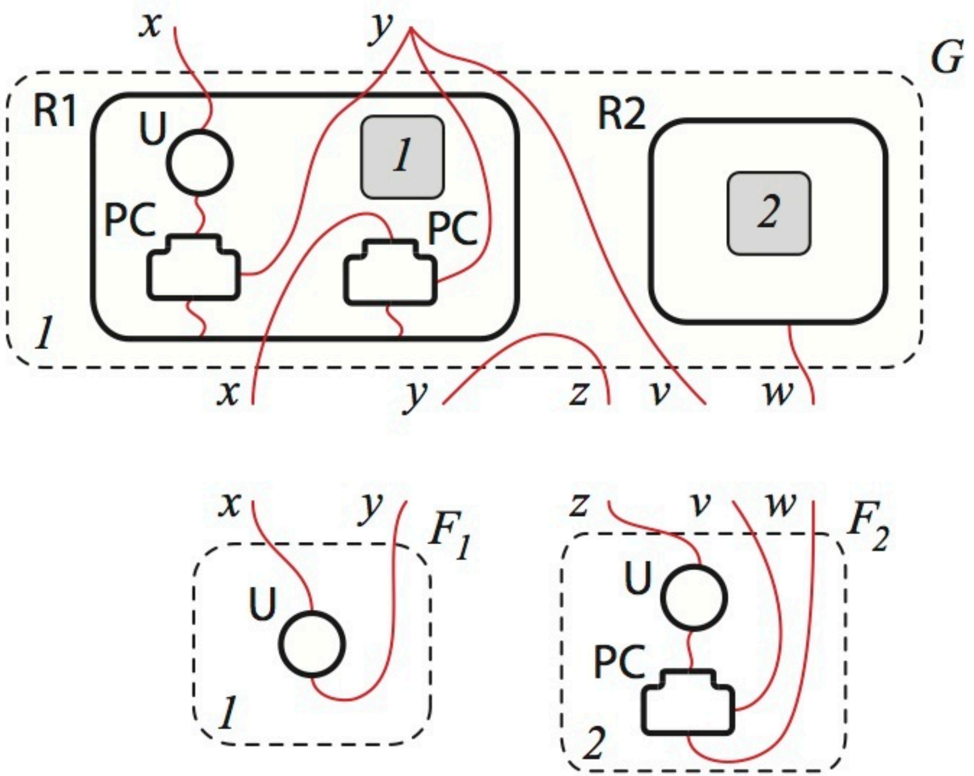
Each bigraph has

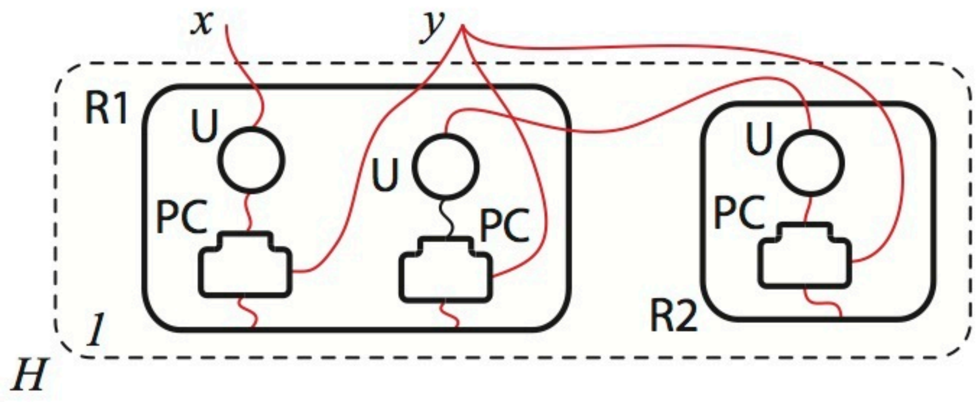
- *outer interfaces*: roots with exposed names, to be connected
- *inner interface*: sites where other components can be connected

Bigraphs can be composed - vertically and horizontally

Horizontal composition: “putting things along”

Vertical composition: If H : X→Y and G : Y→Z, then G∘H : X→Z is defined and obtained by *grafting* place graphs and connecting links. Example:





$$H \equiv G \circ (F_1 \otimes F_2)$$

Tools and libraries for bigraphs

BigraphER

- **BigraphER** (https://uog-bigraph.bitbucket.io/): a modelling and reasoning environment for bigraphs providing an efficient implementation of rewriting, simulation, and visualisation
- **Bigraph Framework** (https://bigraphs.org/): a framework written in Java for the manipulation and simulation of bigraphical reactive systems
- **jLibBig** (https://bigraphs.github.io/jlibbig/): a Java library providing efficient and extensible implementation of bigraphical reactive systems for (directed) bigraphs
- And some others

Multi-agent Systems Design and Prototyping with Bigraphical Reactive Systems*

Alessio Mansutti, Marino Miculan, and Marco Peressotti

Bigraphical models for protein and membrane interactions

Giorgio Bacci Davide Grohmann Marino Miculan

A Strategy-Based Formal Approach for Fog Systems Analysis

Souad Marir [1,2,*], Faiza Belala [1] and Nabil Hameurlain [2]

Modeling Self-Adaptive Fog Systems Using Bigraphs

Hamza Sahli[1], Thomas Ledoux[2], and Éric Rutten[3]

Modeling and Verification of Evolving Cyber-Physical Spaces

Christos Tsigkanos, Timo Kehrer, and Carlo Ghezzi
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy

Bigraph Theory for Distributed and Autonomous Cyber-Physical System Design

Vincenzo Di Lecce, Alberto Amato, Alessandro Quarto *Member IAENG,* Marco Minoia

UAV Swarms Behavior Modeling Using Tracking Bigraphical Reactive Systems

Piotr Cybulski * and Zbigniew Zieliński

Controlling resource access in Directed Bigraphs

Davide Grohmann[1], Marino Miculan[2]

BigraphTalk: Verified Design of IoT Applications

Blair Archibald, Min-Zheng Shieh, Yu-Hsuan Hu, Michele Sevegnani, and Yi-Bing Lin, *Fellow, IEEE*

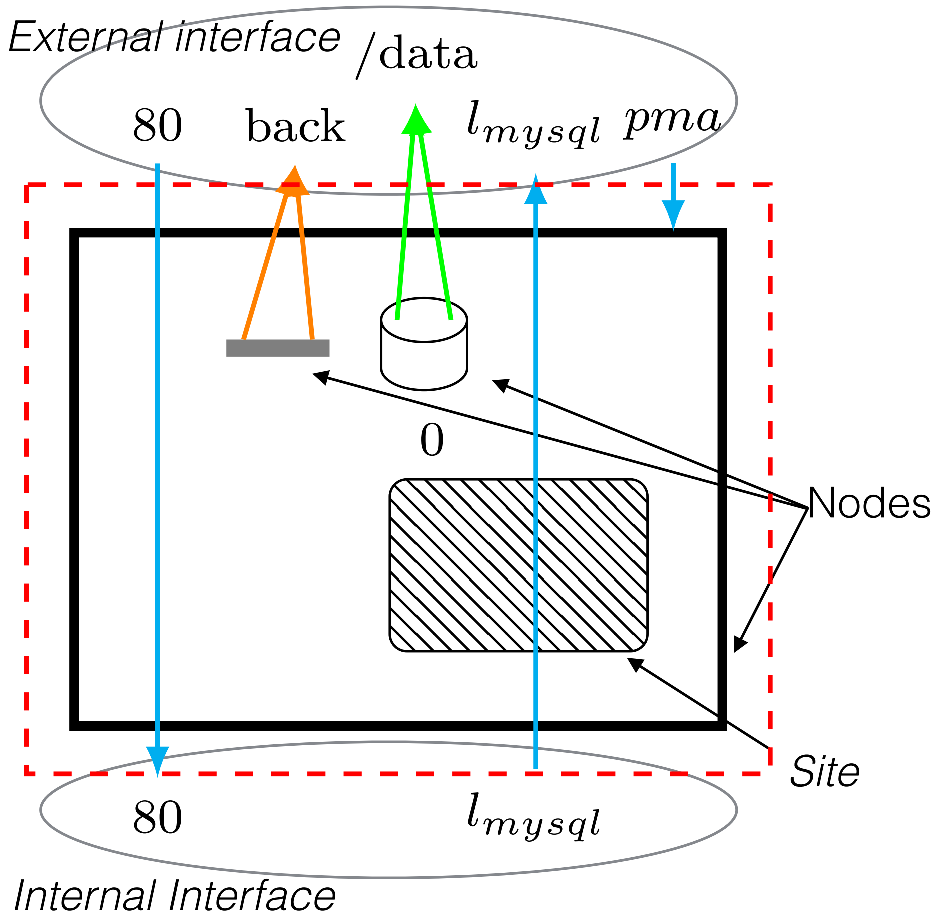
Security, cryptography and directed bigraphs

Davide Grohmann

Local direct bigraphs

[Burco, Peressotti, M., ACM SAC 2020]

- For containers, we have introduced **local directed bigraphs**, where
  - Nodes have assigned a type, specifying arity and polarity (represented by different shapes) and can be nested
  - *Sites* represent “holes” which can be filled with other bigraphs
  - Arcs can connect nodes to nodes (respecting polarities) or to names in *internal* and *external interfaces* (with locality)



Local directed bigraphs — more formally

- A *(polarized) interface (with localities)* is a list of pairs of finite sets of names

Global names

Local names (a pair for each locality)

$$X : \langle (X_0^+, X_0^-), (X_1^+, X_1^-), \ldots, (X_n^+, X_n^-) \rangle$$

$$X^+ \triangleq \biguplus_{i=1}^{n} X_i^+ \qquad X^- \triangleq \biguplus_{i=1}^{n} X_i^- \qquad width(X) \triangleq n$$

Ascending names

Descending names

- Interfaces can be juxtaposed:

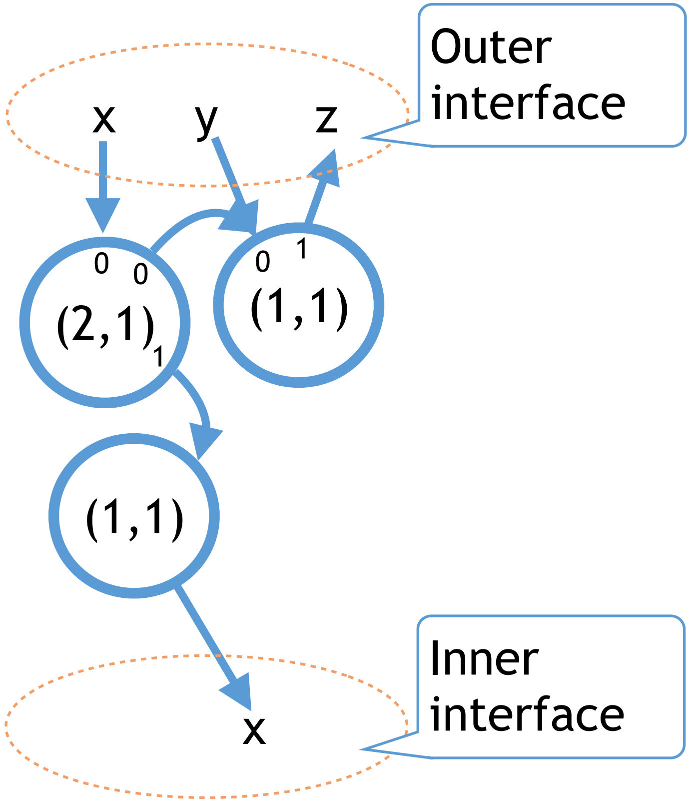
$$X \otimes Y \triangleq \langle (X_0^+ \uplus Y_0^+, X_0^- \uplus Y_0^-), (X_1^+, X_1^-), \ldots, (X_n^+, X_n^-), (Y_1^+, Y_1^-), \ldots, (Y_m^+, Y_m^-) \rangle$$

Local interfaces are everywhere

- This system has an interface (on this side) of width=24
- Each locality (i.e. each socket) has many wires, that is, *names*
  - Ascending names = wires accessing resources outside the PC
  - Descending names = wires giving access to resources inside the PC
- Each locality is for accessing external resources (e.g. energy, mike, network, keyboard, mouse…), or to provide access to internal resources (e.g. PCIe), or both

Local directed bigraphs — more formally

- A **signature** $K = \{c_1, c_2, \dots\}$ is a set of controls, i.e. pairs $c_i = (n_i^+, n_i^-)$
- Each *control* is the type of basic components, specifying inputs (positive part) and outputs (negative part)
- Notice: direction of arrows represents “access” or “usage”, not “information flow” (somehow dual to string diagrams for monoidal cats)
- Figure aside: a graph representing a system that accesses to some internal service over x, some external service over z, and provides services over x,y



Local directed bigraphs — more formally

- A **signature** $K = \{c_1, c_2, \ldots\}$ is a set of controls, i.e. pairs $c_i = (n_i^+, n_i^-)$
- Given two interfaces *I*, *O*, a local directed bigraph $B : I \to O$ is a tuple

$$B = (V, E, ctrl, prnt, link)$$

  where

  - V = finite set of *nodes*
  - E = finite set of *edges*
  - $ctrl : V \to K$ = *control map*: assigns each node a type, that is a number of *inward* and *outward ports*
  - *prnt*: tree-like structure between nodes
  - *link*: directed graph connecting nodes' ports and names in interfaces (respecting polarity)

Local directed bigraphs — more formally

- Let K be a fixed signature, and *X*, *Y*, *Z* three interfaces.
- Given two bigraphs $B_1 : X \to Y, B_2 : Y \to Z$, their composition is

$$B_2 \circ B_1 = (V, E, ctrl, prnt, link) : X \to Z$$

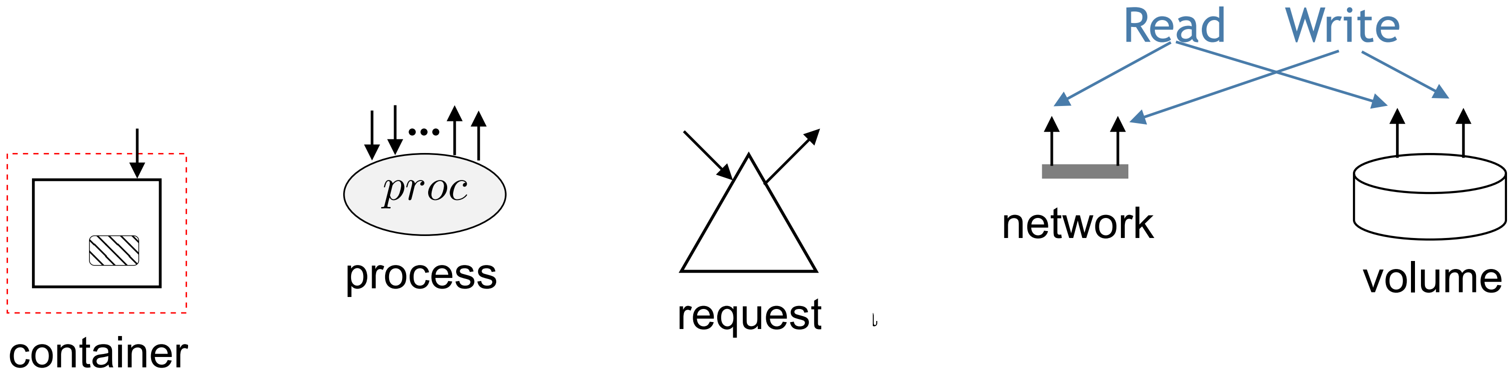
  defined by "filling the holes and connecting the wires" as expected

- Yields a **monoidal category** (Ldb(K),⊗,0)
  - Objects: local directed interfaces
  - Arrows: local directed bigraphs
  - Tensor: juxtaposition
- Enjoys nice properties of bigraphs (RPOs, IPOs, etc.)

A signature for containers

- Controls to represent main elements of a container

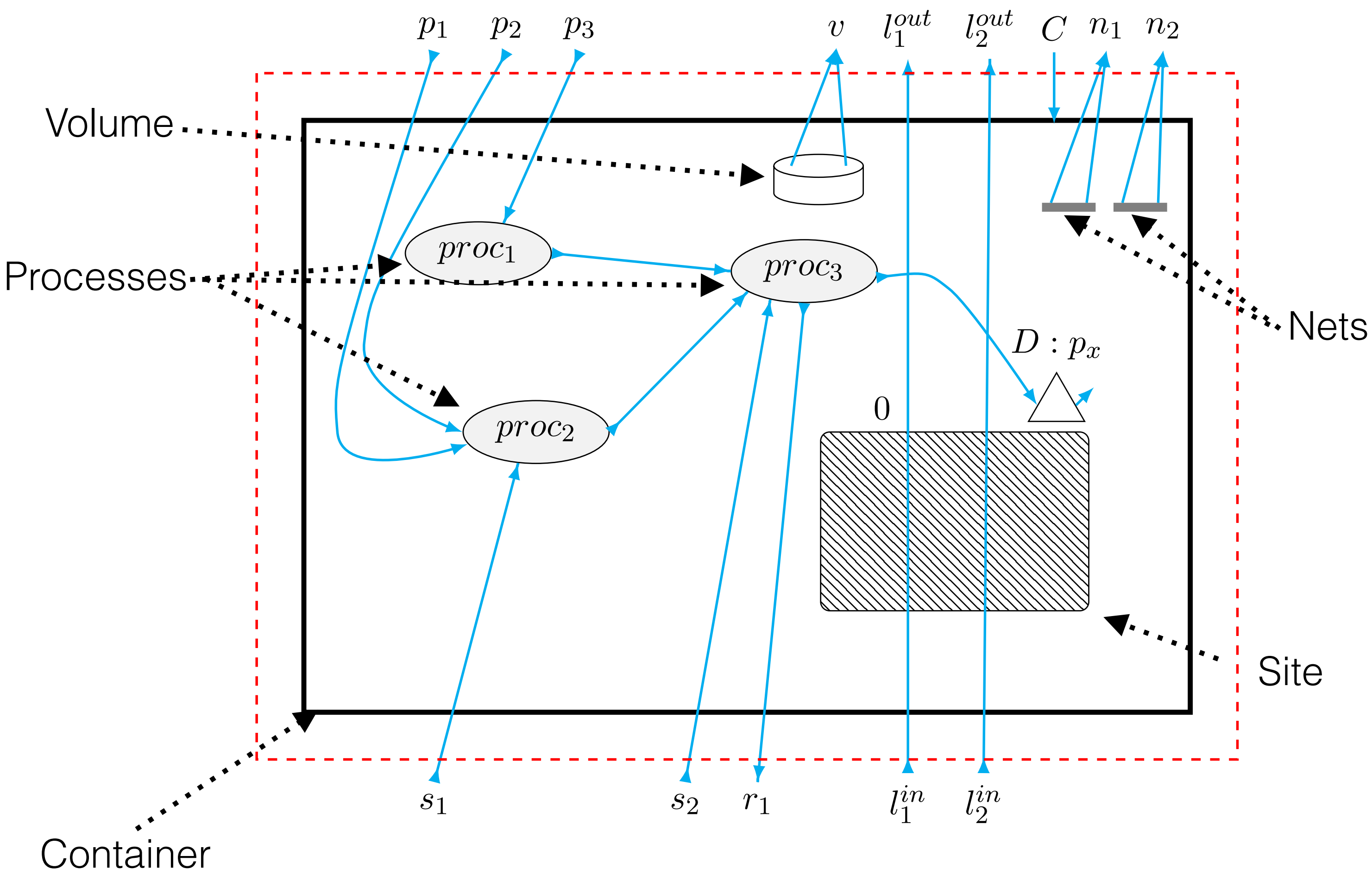
K_{cont} =



- shapes are only for graphical rendering
  - (nodes are subject to some sorting conditions)
- Can be extended with other controls as needed (achieving *flexibility* and *openness*)
  - Changing signature = change of base in fibred category

Containers are modeled as local directed bigraphs

- Container = local directed bigraph whose interfaces contain the name of the container, the exposed ports, required volumes and networks, etc.
- This is not only a picture, but the graphical representation of two interfaces and a morphism in the category Ldb(K_{cont})



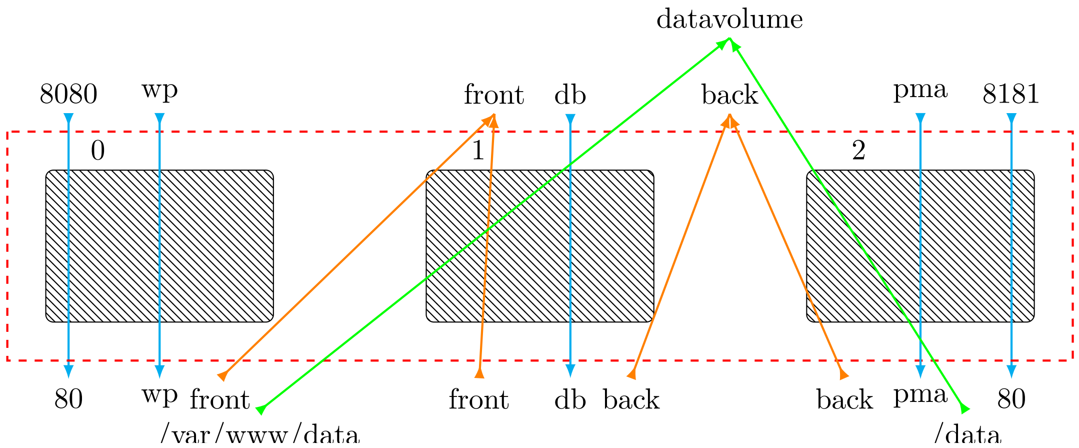
$$B : \langle(\{\},\{\}),(\{s_1,s_2,l_1^{in},l_2^{in}\},\{r_1\})\rangle \to \langle(\{\},\{\},(\{n_1,n_2,v,l_1^{out},l_2^{out}\},\{p_1,p_2,p_3,C\}))\rangle$$

And composition is another bigraph

- The YAML configuration file for `docker compose` corresponds to a *deployment bigraph* specifying volumes, networks, name and port remapping, etc.

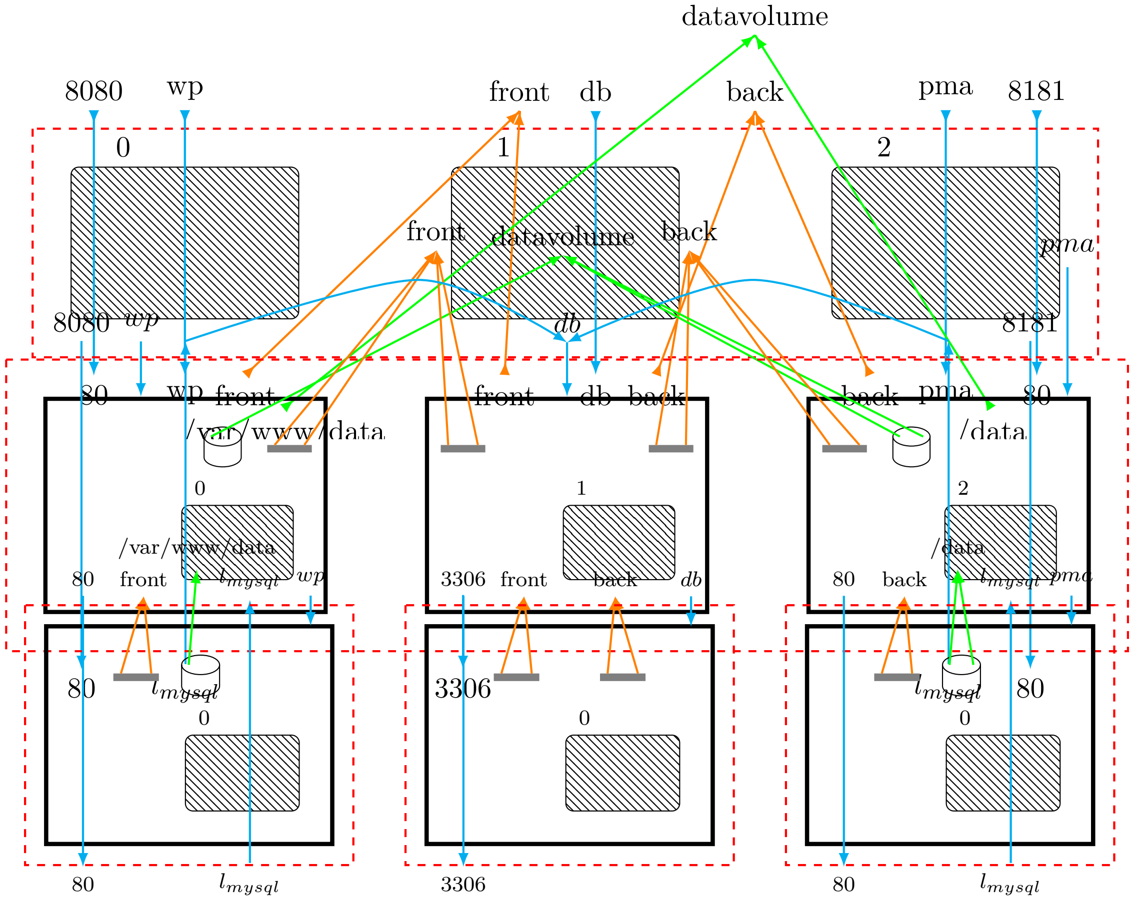
```
services:
  wp:
    image: wordpress
    links:
      - db
    ports:
      - "8080:80"
    networks:
      - front
    volumes:
      - datavolume:/var/www/data:ro
  db:
    image: mariadb
    expose:
      - "3306"
    networks:
      - front
      - back
```

```
  pma:
    image: phpmyadmin/phpmyadmin
    links:
      - db:mysql
    ports:
      - "8181:80"
    volumes:
      - datavolume:/data
    networks:
      - back
networks:
  front:
    driver: bridge
  back:
    driver: bridge
volumes:
  datavolume:
    external: true
```



And composition is another bigraph

- Composition of containers (as done by `docker compose`)
  =
  composition of corresponding bigraphs inside the deployment bigraph
  - Encoding is "functorial"
- The model of a running application is a bigraph obtained by composing the bigraphs of the components

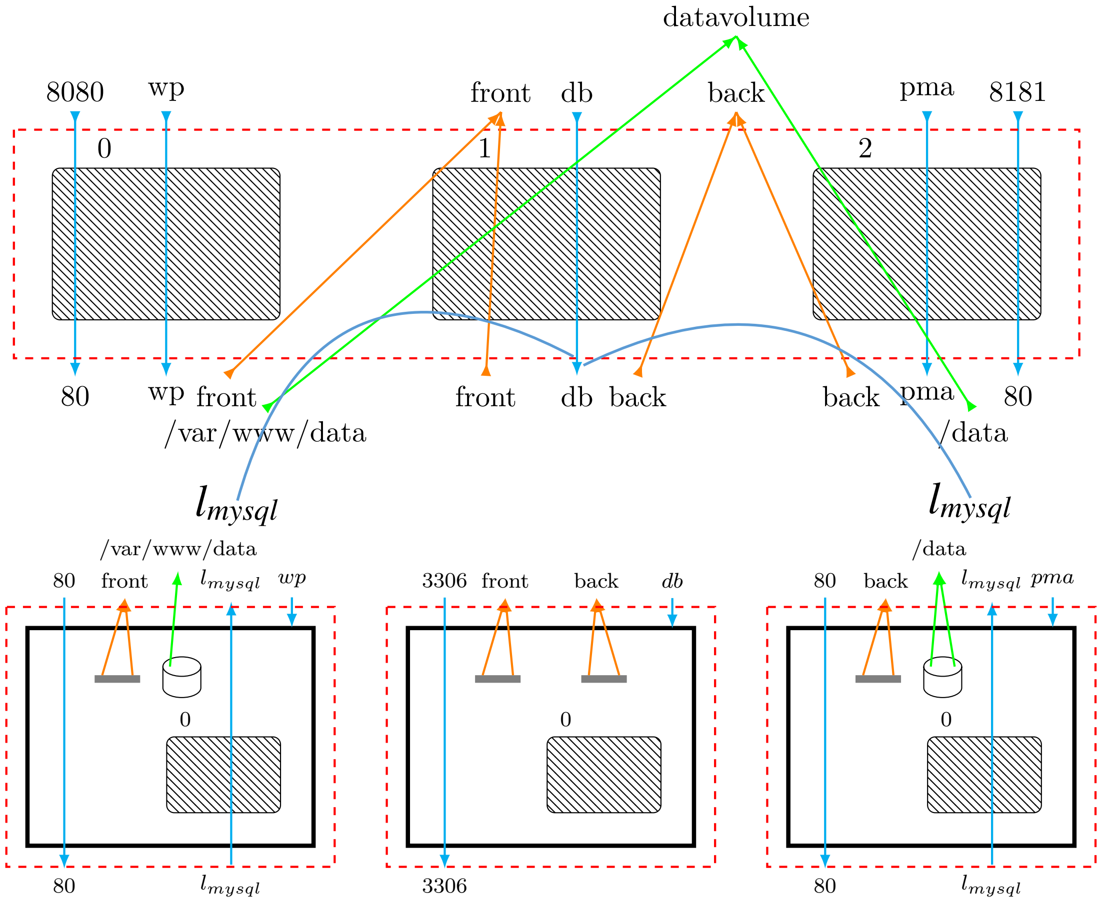


Application: safety checks on the configuration

When represented as bigraphs, systems can be analysed using tools and techniques from graph theory

Simple example:

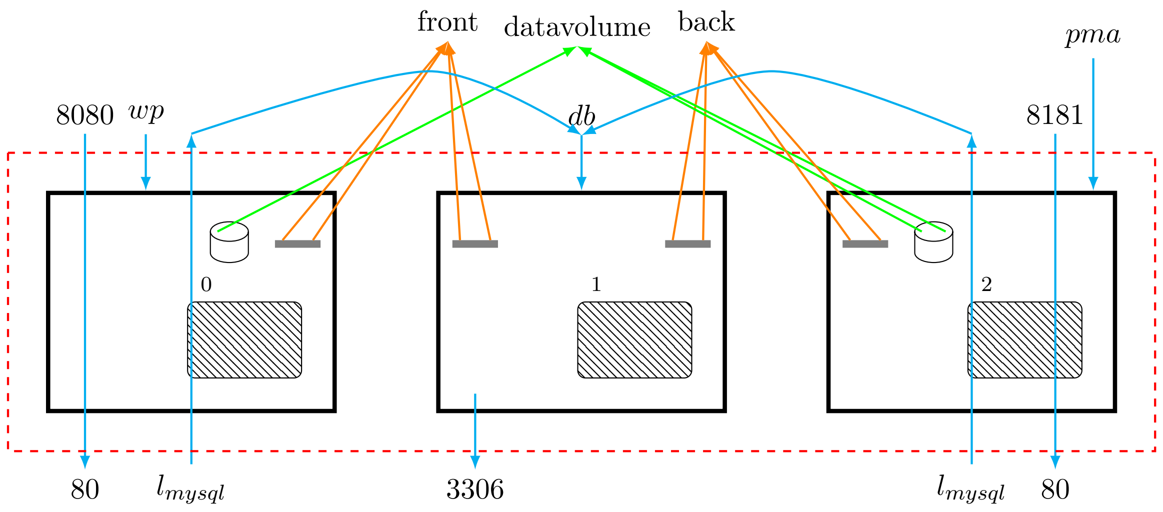
- **Valid links**: "if a container has a link to another one, then the two containers must be connected by at least one network"
  - Corresponds to a simple constraint on the deployment bigraph



Application: Network separation (no information leakage)

- assume that networks (or volumes) have assigned different security levels (e.g "public < guests < admin", "back < front").
- Security policy we aim to guarantee (akin Bell-LaPadula):
  - "Information from a higher security network cannot leak into a lower security network, even going through different containers"

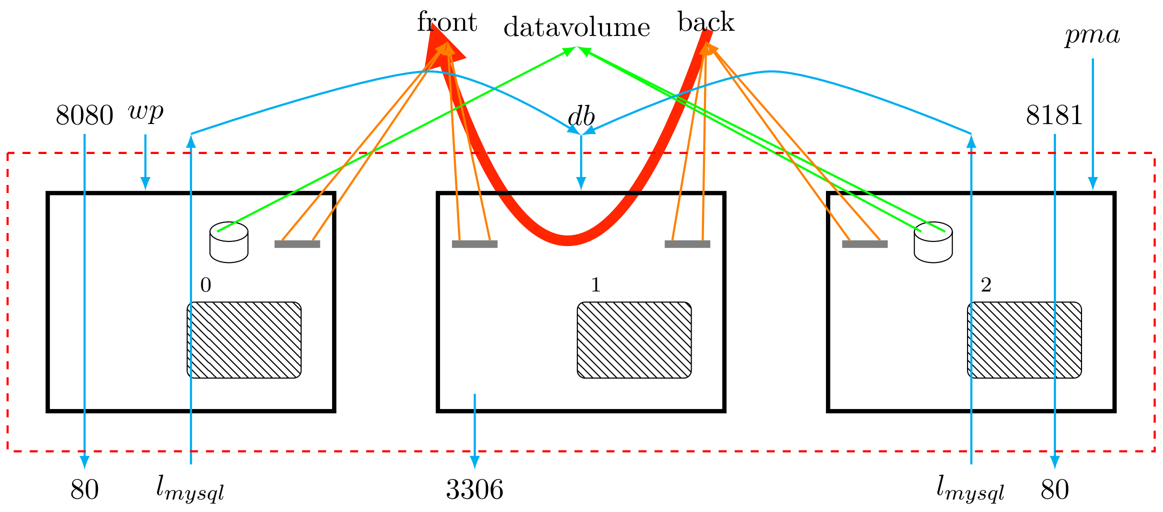
front > back

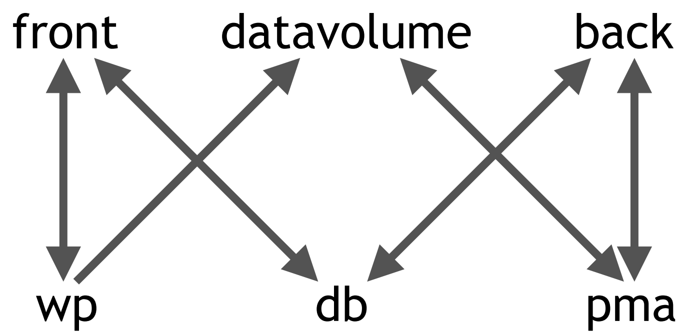


Application: Safe network separation

- Can be reduced to a *reachability problem* on an auxiliary graph representing *read-write accessibility* of containers to resources
  - The r/w accessibility graph is easily derived from the bigraph of the system
- Security policy is reduced to the property: “For each pair of resources m,n such that n < m, there is no path from n to m” (i.e., n cannot access m)
  - If this is the case, the configuration respects the security policy. Otherwise, an information leakage is possible

front > back



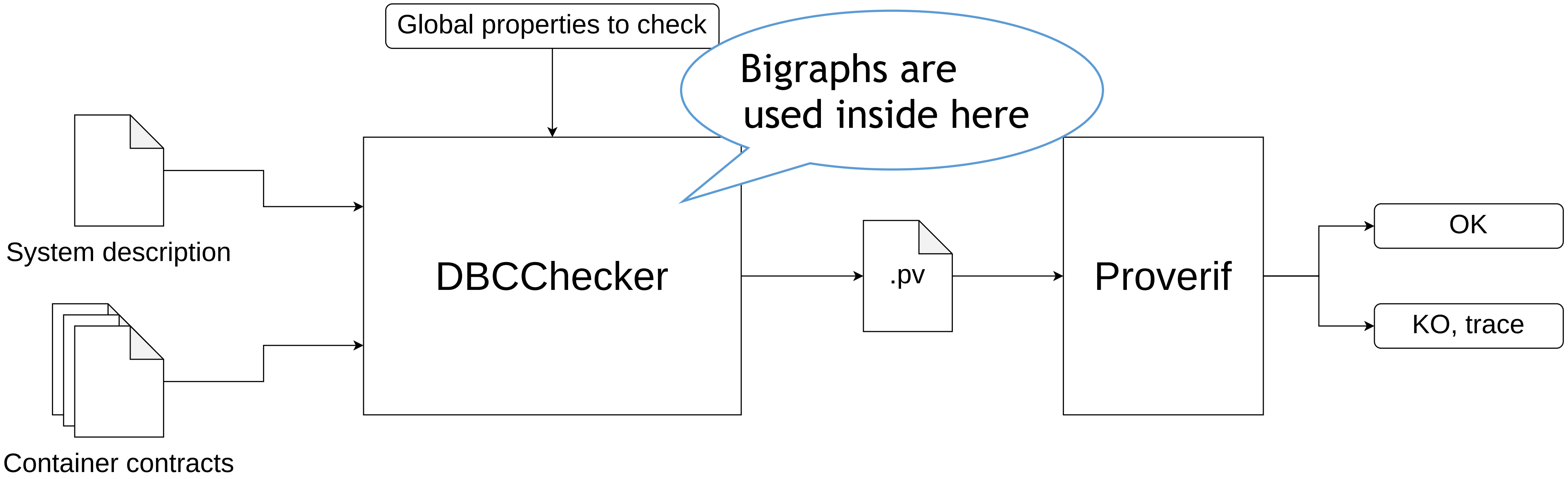


DBCChecker [Altarui, M., Paier, ITASEC 2023]

A tool aiming to verify security properties of systems obtained by composition of containers

D B C
CHECKER

DBCChecker



- Input:
  - a configuration of a container-based system (in JBF - *JSON Bigraph Format*)
  - for each container, an abstract description of the interaction on its interface ("contract")
  - Global properties to be checked
- Output: a model for the global system, verifiable in some backend (here, ProVerif)

JSON Bigraph Format (JBF)

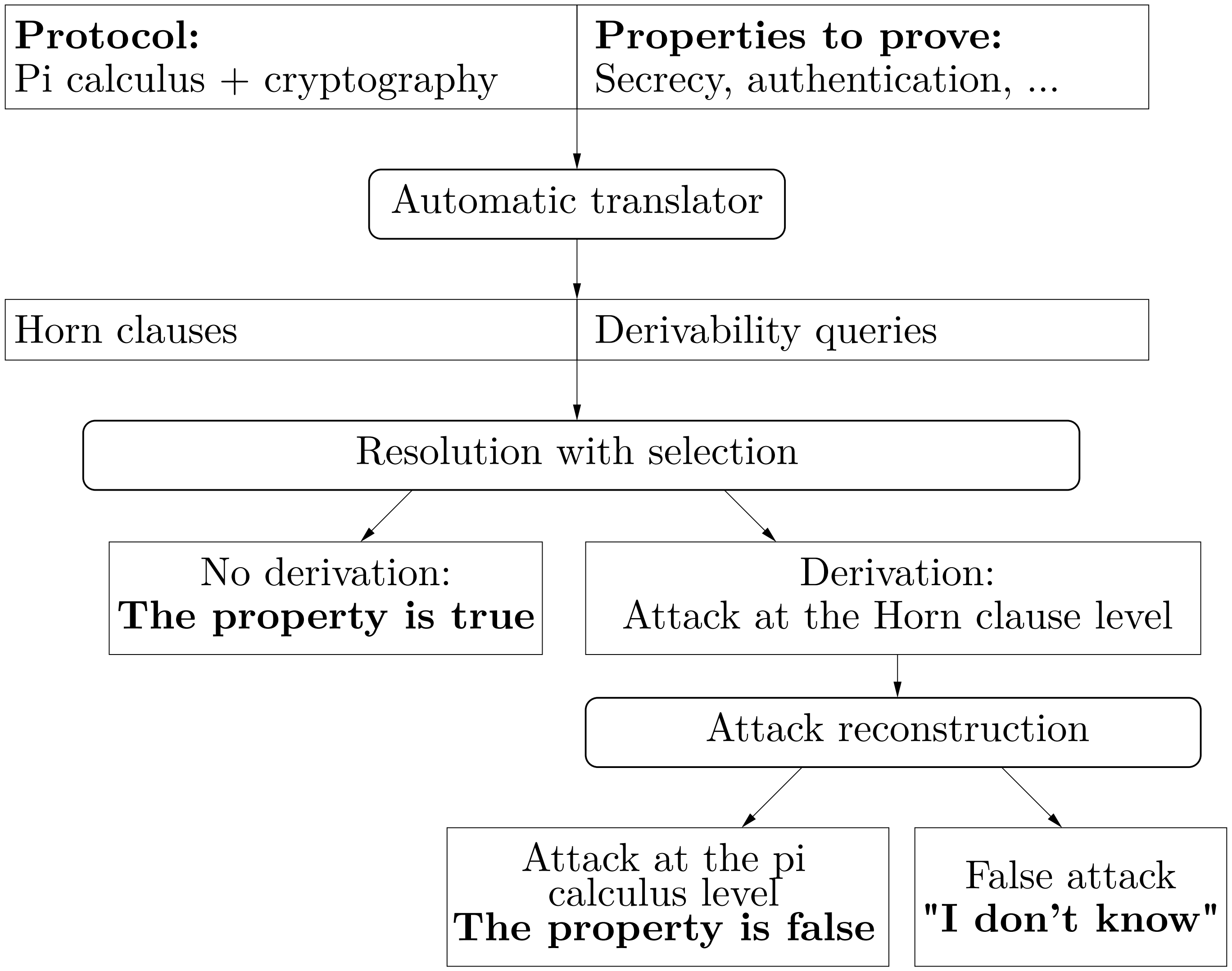
- Based upon the standard JSON Graph Format (JGF).
- Uses metadata objects to describe the signature and other specific information of directed bigraphs.
    - This allows us to describe the properties that do not fit in JGF without modifying the format

```
{
  "graph": {
    "nodes": {
      "NodeName": {
        "metadata": {
          "type": "type"
        },
        "label": "label"
      }
    },
    "edges": [
    {
      "source": "sourceNode",
      "relation": "relation",
      "target": "targetNode",
      "metadata": {
        "portFrom": "portFrom",
        "portTo": "portTo"
      }
    },
    {
      "source": "sourceNode",
      "relation": "relation",
      "target": "targetNode",
      "metadata": {
        "portFrom": "portFrom",
        "portTo": "portTo"
      }
    }
    ],
    "type": "type",
    "metadata": {
      "signature": [
      {
        "name": "name",
        "arityOut": 1,
        "arityIn": 1
      }
      ]
    }
  }
}
```

ProVerif [Blanchet, 2016]

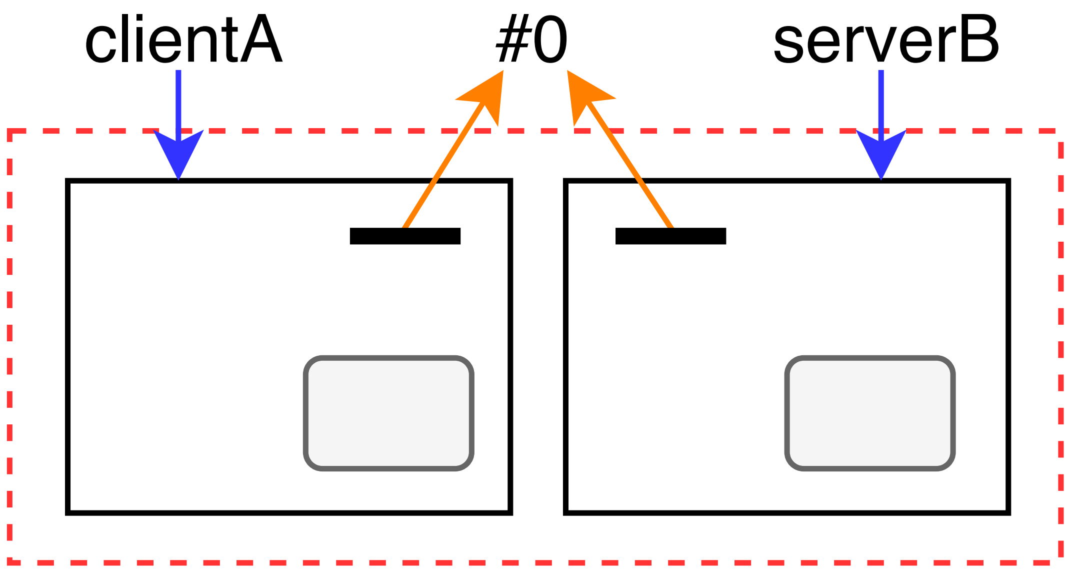
- ProVerif is a verifier for cryptographic protocols that may prove that a protocol is secure or exhibit attacks in the Dolev-Yao model
- Advantages
  - fully automatic, and quite efficient
  - a rich process algebra (based on applied π-calculus)
  - handles many cryptographic primitives
  - various security properties: secrecy, correspondences, equivalences
- Cons:
  - the tool can say "can not be proved"
  - termination is not guaranteed
- Available at http://proverif.inria.fr

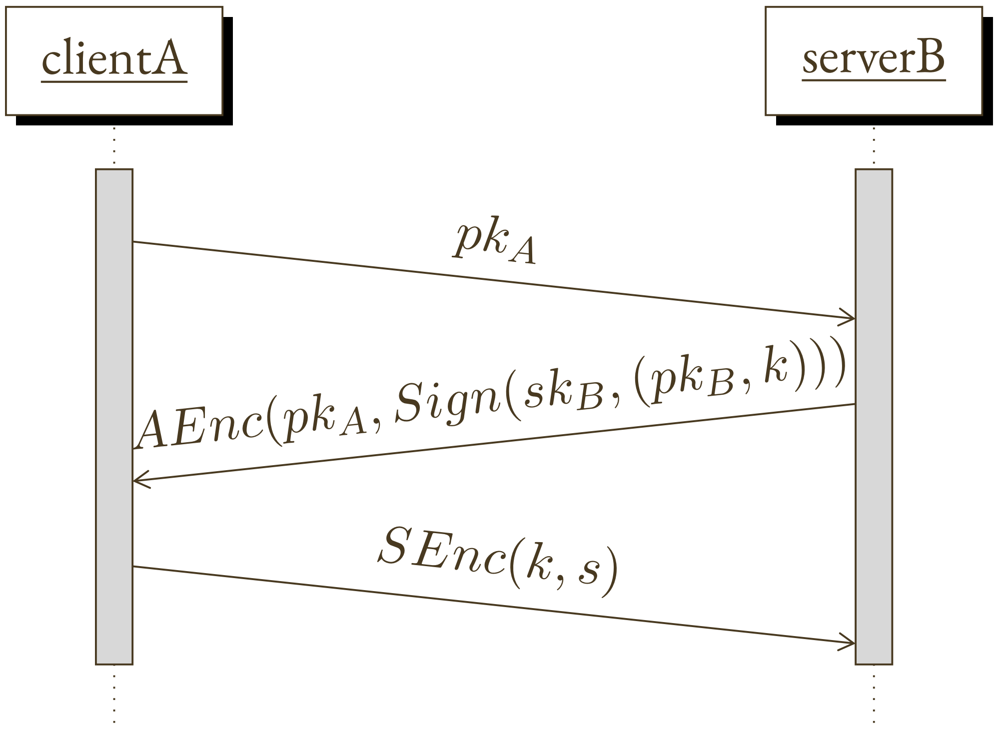
ProVerif architecture [Blanchet, 2016]



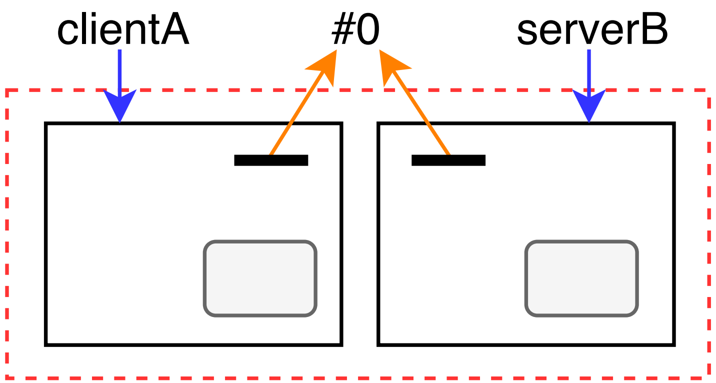
A basic example: secure handshake

- Two containers, “client” and “server”
- Global property to check: **confidentiality** of message s





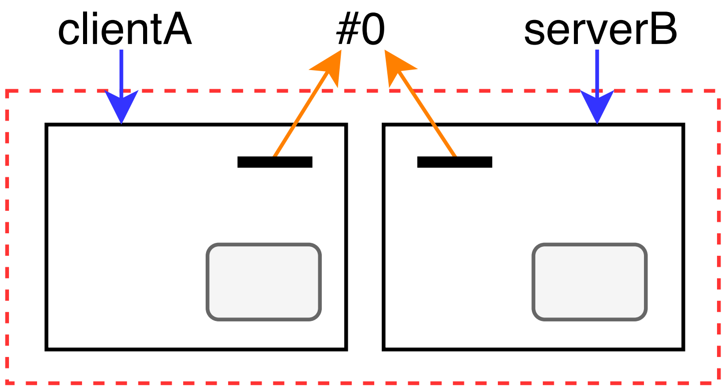
A basic example: secure handshake: contracts

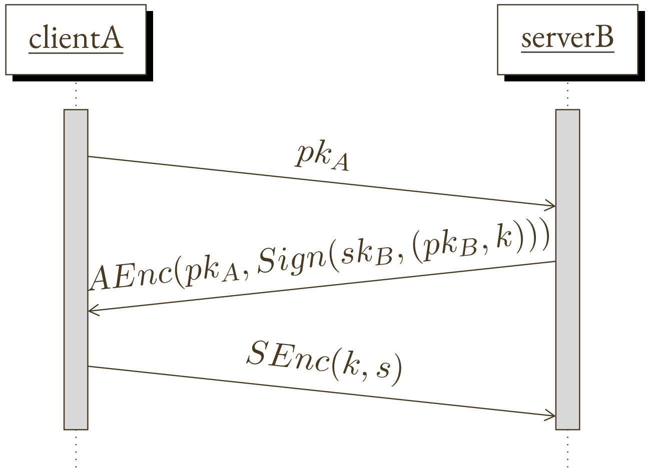


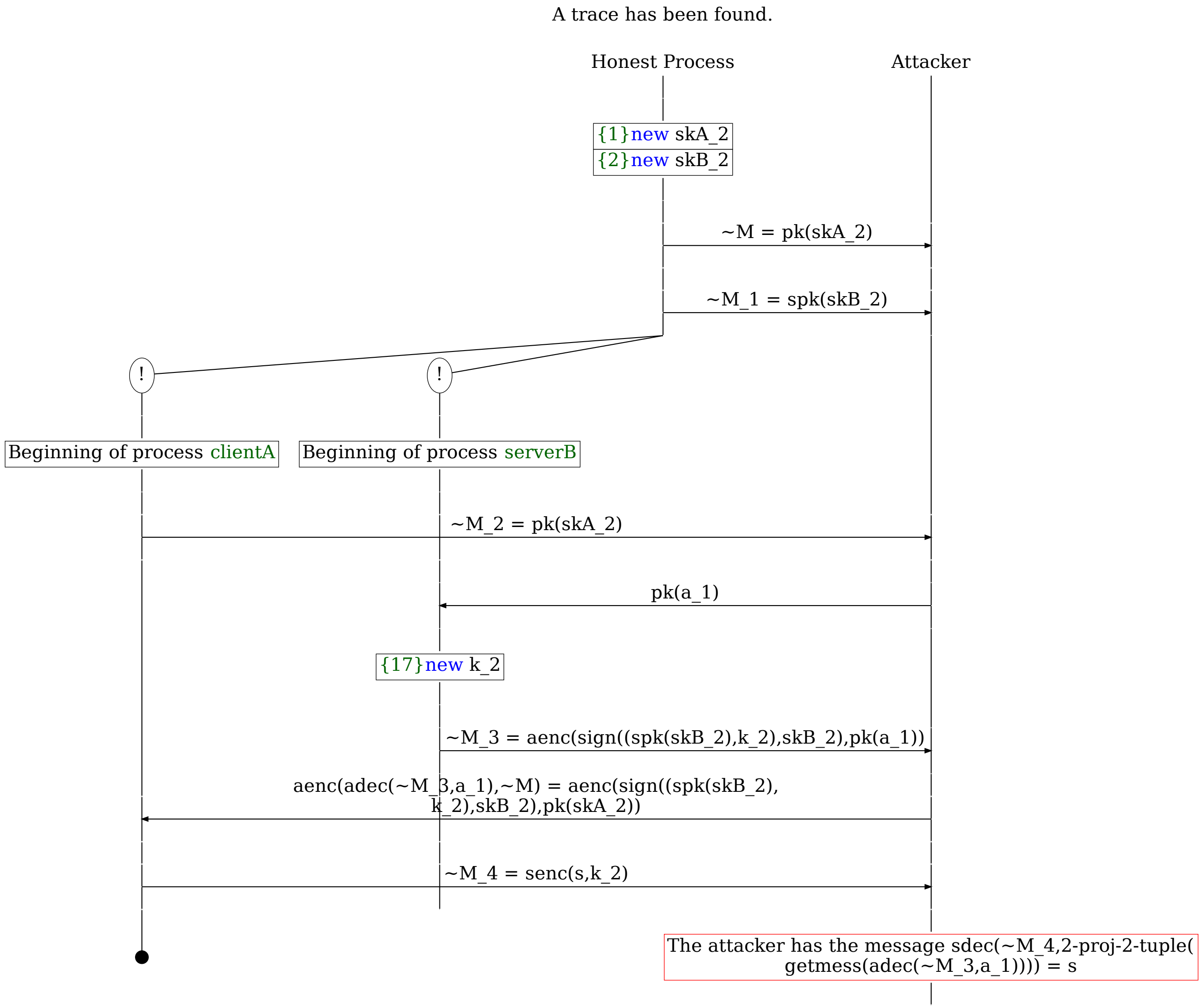

```
"clientA": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "params": ["pkA:pkey", "skA:skey",
           "pkB:spkey"],
    "behaviour": "!(out (#0+, pkA);
           in (#0+, x : bitstring);
           let y = adec(x, skA) in
           let (=pkB, k : key) = checksign(y,
           pkB) in
           out (#0+, senc(s, k)))."，
    "attribute": ""
  },
  "label": "clientA"
}
```

```
"serverB": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "params": ["pkB:spkey", "skB:sskey"],
    "behaviour": "!(in(#0+, pkX : pkey);
           new k : key;
           out(#0+, aenc(sign((pkB, k), skB),
           pkX));
           in(#0+, x : bitstring);
           let z = sdec(x, k) in 0 ).",
    "attribute": ""
  },
  "label": "serverB"
}
```

A basic example: secure handshake: analysis result

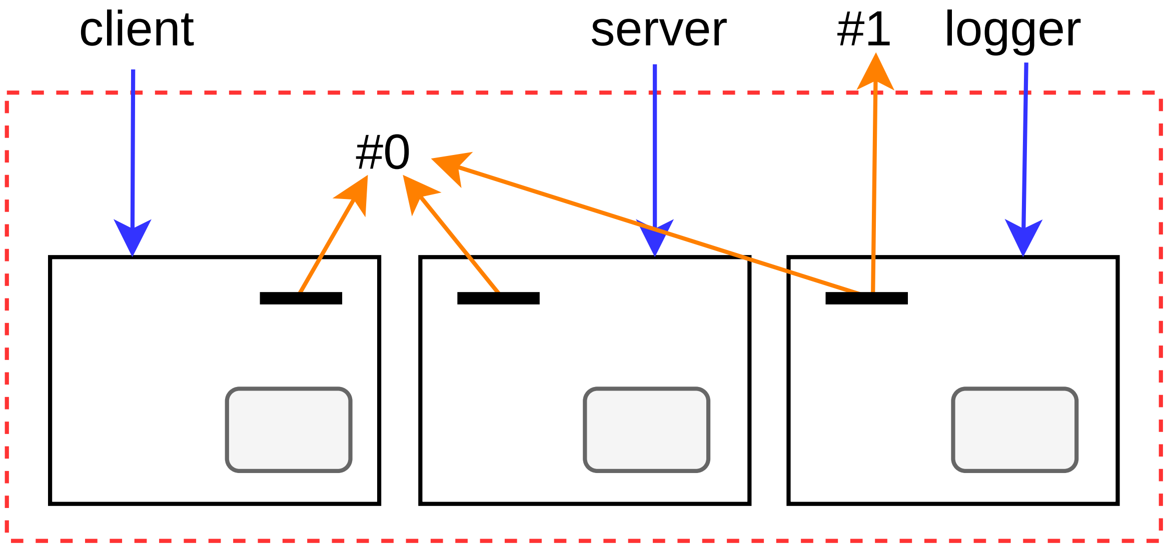




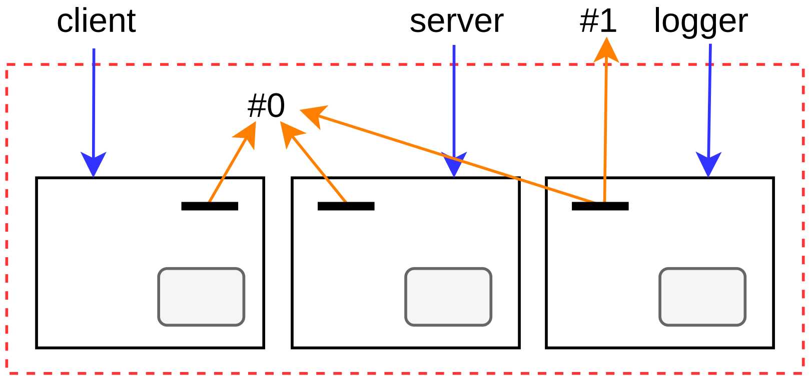


A slightly more advanced example: reconfiguration

- Two containers are communicating over a private channel.
- Global property to check: **confidentiality** of data.
- The system is secure (because the network is internal).
- But if we add another container, the property may not be preserved



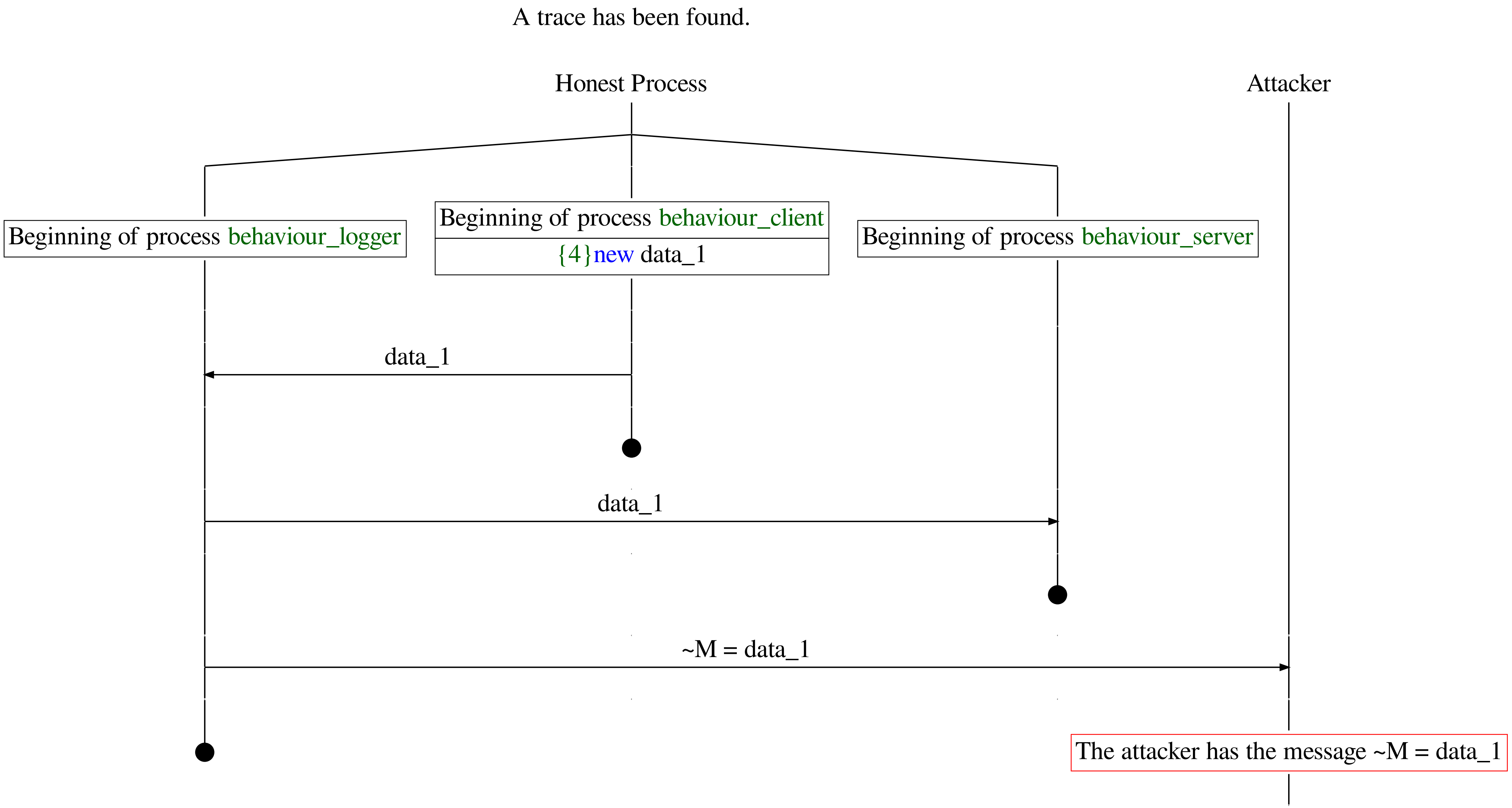
Reconfiguration: contracts




```
"client": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "properties": {
      "params": [],
      "behaviour": "new
             data:bitstring;
             out(#0-, data).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "client"
},
```

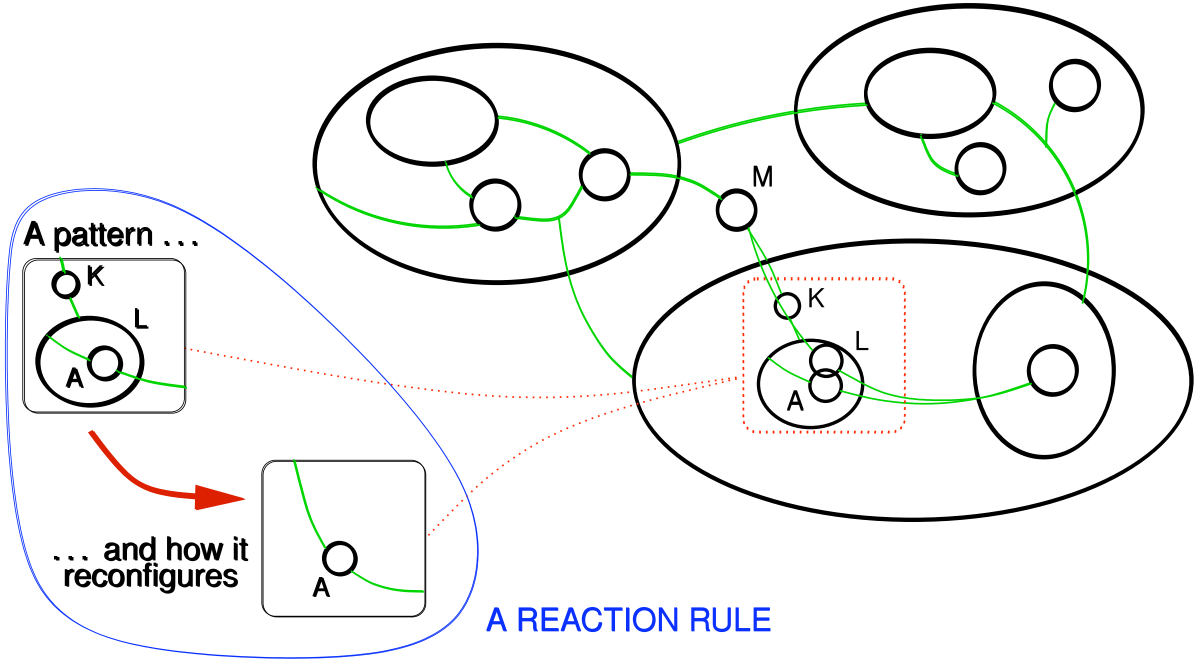
```
"server": {
  "metadata": {
    "type": "node",
    "control": "1on0",
    "properties": {
      "params": [],
      "behaviour": "in(#0-,
             data_received:bitstring).",
      "events": [],
      "attribute": ""
    }
  },
  "label": "server"
},
```

Reconfiguration: analysis result



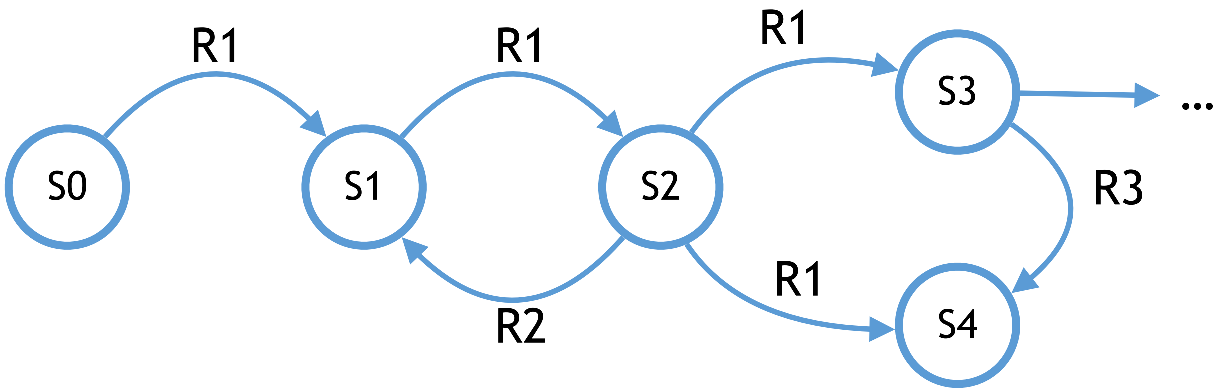
System modification = Bigraphic rewriting

- So far, bigraphs have been used to represent the connection configuration of a containerized system
- Connections and positions of elements of a system can change at run-time (connections, services requests between processes…)
- Bigraphic models represent these dynamics by means of **rewriting rules**
- A rule can replace/move nodes, change connections, etc…

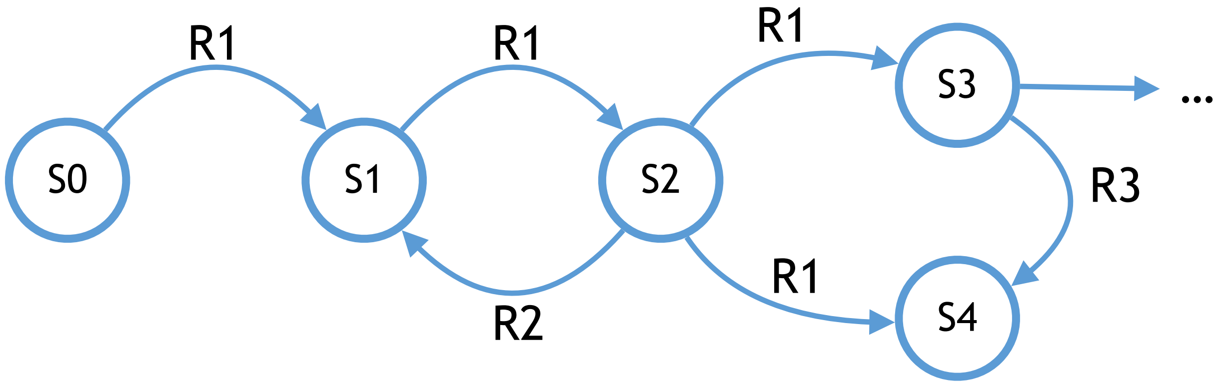


Container system evolution: by means of rewriting rules

- A *LDB Reactive System (LDBRS)* is defined by a set of rules
- Given a starting configuration (= a ground bigraph), a LDBRS induces a *labelled transition system* (LTS), where
  - States = reachable *configurations* by means of rewritings
  - Labels = rules applied in the rewritings (= actions)



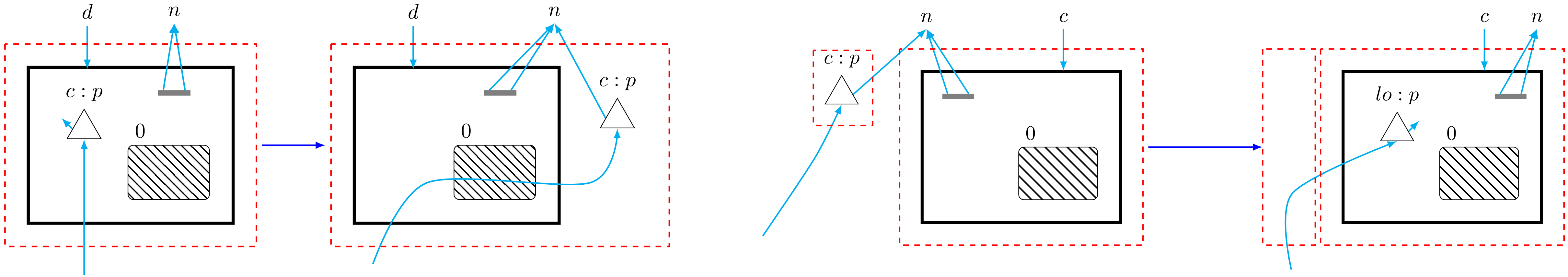
Container system evolution: by means of rewriting rules



- Over this LTS we can verify many properties by *model checking*, e.g.:
    - *reachability* and *planning*
    - *safety properties* ("bad things don't happen")
    - *liveness properties* ("good things do happen")
- We can verify these properties *before* actually applying the changes, or to plan the correct sequence of changes

Dynamic properties: System's runtime

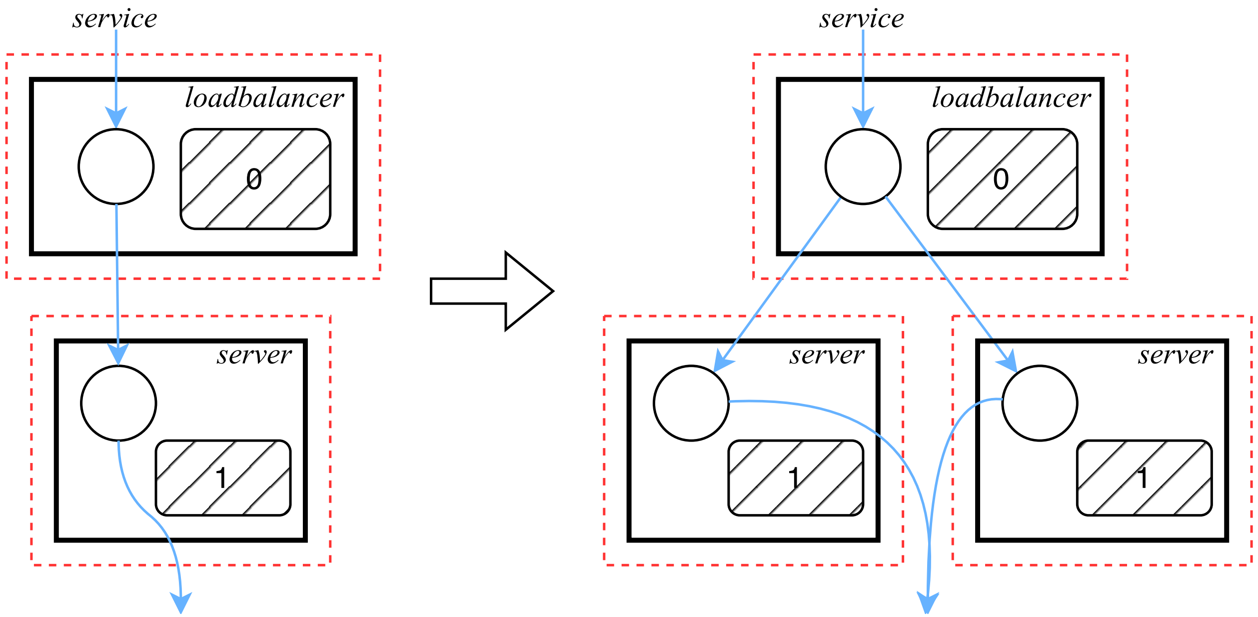
- Rules can represent runtime dynamics
- Example: connection request / connection accepted



- The induced LTS represent different states that the system can reach at runtime
- Over this LTS we can verify usual temporal properties (liveness, fairness), e.g.
  - Eventual success of service request
  - Temporal security guarantees, eg: "if a process reads from X then it cannot write on any Y whose security level is less than X's"

Dynamic properties: System's reconfiguration

- Rules can represent *system reconfiguration* (static or dynamic), such as:
  - Container replacement / update (e.g. library/code upgrade)
  - Horizontal scaling:



- The induced LTS represent different configurations of the system
- "Temporal" safety invariants = stability under reconfiguration

Conclusions: what we have done…

- Proposed a bigraph-based formal model for container-based systems
- Captures logical connections of components and processes, nesting of components, composition of containers
- Basis for tools and for theoretical results
- Applicable for, e.g., static analysis of container systems
- Implemented prototype checker tool

Conclusions: some current and future work

- Formalisation of other static properties (Spatial logics?)
- Integrate with runtime monitoring
  - Generate rules for runtime monitors (see Baldo's work)
  - If we observe something unexpected, is it an error, or reconfiguration?
- Quantitative aspects (e.g. fault probability estimation)
- Configuration synthesis or refinement (e.g. by rewriting rules which fix security policy violation)
- Session types for specifying contracts
- Improve tools, UI/UX
- …

Thanks for your attention! Questions?

marino.miculan@uniud.it

References

- [Verderame et al, 2023] L Verderame, L Caviglione, R Carbone, A Merlo. “SecCo: Automated Services to Secure Containers in the DevOps Paradigm” Proceedings of the 2023 International Conference on Research in Adaptive and Convergent Systems
- [Milner, 2006] R Milner. "Pure bigraphs: Structure and dynamics." Information and computation 204.1 (2006): 60-122.
- [Archibald et al, 2024] B Archibald, M Calder, M Sevegnani. “Practical Modelling with Bigraphs.” arXiv preprint arXiv:2405.20745, 2024
- [Burco et al., 2020] F Burco, M Miculan, M Peressotti. "Towards a formal model for composable container systems." Proceedings of the 35th Annual ACM Symposium on Applied Computing. 2020.
- [Altarui et al., 2023] A Altarui, M Miculan, and M Paier. "DBCChecker: A Bigraph-Based Tool for Checking Security Properties of Container Compositions." CEUR WORKSHOP PROCEEDINGS. Vol. 3488. CEUR-WS, 2023.
- [Blanchet, 2016] B Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif." Foundations and Trends in Privacy and Security 1.1-2 (2016): 1-135.