

DBCChecker: A Bigraph-Based Tool for Checking Security Properties of Container Compositions^{*}

Andrea Altarui¹, Marino Miculan^{2,3,*} and Matteo Paier^{2,4}

¹Department of Computer Science, University of Milan, Italy

²Department of Mathematics, Computer Science and Physics, University of Udine, Italy

³Department of Environmental Sciences, Informatics and Statistics, University of Venice, Italy

⁴IMT School for Advanced Studies, Lucca, Italy

Abstract

Despite their widespread application in modern systems, container composition is often complex and error-prone. In this work, we present DBCChecker, a tool aiming to verify security properties of systems obtained by composition of containers. From the configuration of a container-based system and an abstract description of the interface behaviour of each container, the tool builds a formal model of the overall system, which can be verified in ProVerif (an automatic symbolic protocol verifier), to check that the overall system satisfies the required properties. The system can be described in a specification language capable to express at once the interfaces and connections of containers and the relevant behavioural aspects of their interfaces, called *JSON Bigraph Format* (JBF), and inspired by previous formal models, based on *bigraphs*, for containerized architectures.

Keywords

Containers, Contract-based design, Formal methods, Verification, System Security, Bigraphs

1. Introduction

Nowadays, *containers* are increasingly adopted in the design and implementation of complex software systems. The flexibility of containerized architectures support easier integration, scalability, dynamic deployment and reconfiguration. However, connecting and coordinating containers into a complete working system, using tools like `docker compose`, is not an easy task: a misconfiguration can easily yield unexpected behaviours or allow for dangerous communications between containers, violating security goals and policies. Sometimes these problems may be caught at the system's start-up, but often they arise suddenly at any time during the system lifetime, causing service malfunctions or interruptions, information leakages, vulnerabilities, etc. Clearly, this situation can benefit from methodologies and tools for checking and verifying systems configuration before they are put in production.


ITASEC 2023: The Italian Conference on CyberSecurity, May 03–05, 2023, Bari, Italy


^{*} Work partially supported by the Italian MIUR project PRIN 2017FTXR7S *IT MATTERS*, the Department Strategic Project of the University of Udine within the InterDepartment Project on Artificial Intelligence (2020-25), and the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

^{*}Corresponding author.

✉ andrea.altarui@studenti.unimi.it (A. Altarui); marino.miculan@uniud.it (M. Miculan); matteo.paier@imtlucca.it (M. Paier)

ORCID 0000-0003-0755-3444 (M. Miculan); 0009-0000-7588-7169 (M. Paier)

 © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

In this work, we present the design and prototype implementation of such a tool, called DBCChecker, whose aim is to verify security properties of systems obtained by the composition of containers. More precisely, given a configuration of a container-based system and for each container an abstract description of the interaction on its interface (i.e., a *contract*), the tool assembles these information into a formal model of the overall system. Then, this model is fed to a verification backend (in our case, ProVerif [1], a state-of-the-art automatic symbolic protocol verifier), to check that the overall system satisfies the required properties.

This work leverages previous results on formal models for containerized architectures based on *bigraphs* [2, 3, 4]. Bigraphs are graph-like data structures capable to describe at once both the locations and the logical connections of (possibly nested) components; as such, they have successfully been applied to the formalization of a broad variety of domain-specific models, including context-aware systems and web-service orchestration languages [5, 6, 7, 8]. Moreover, several tools and libraries have been implemented for bigraph manipulation, such as [9, 10, 11] and in particular JLibBig [12], which is the basis of the current work.

However, interfacing these libraries with containers configurations on one hand, and with protocol verifiers on the other, requires some ingenuity; in particular, we have to introduce a specification language capable to express at once the interfaces and connections of containers (like in bigraphical model) and their contracts; from these specifications DBCChecker can build the input model to be fed to the backend verifier (i.e., ProVerif). This language is called *JJSON Bigraph Format* (JBF), and it is based on JGF, the standard JSON Graph Format.

This paper is structured as follows. In Section 2 we recall the bigraphical models of containers. Based on it, in Section 3 we introduce the specification language JBF. These specifications are the inputs of DBCChecker, which is presented in Section 4. In Section 5 we provide two examples of verification of security properties of systems obtained by container compositions. Finally, in Section 6 we draw some conclusions and sketch directions for further work.

DBCChecker is open source and available at <https://github.com/cysecud/DBCChecker>.

2. Bigraphical models of containers

The formal model for container-based systems is based on *local directed bigraphs* [2], a variant of directed bigraphs [4, 13] which allows us to deal with *localized* resources. Here we briefly recall the basic definitions, referring to [2, 12, 13, 14] for more details.

A *(local) interface* $X = \langle (X_0^+, X_0^-), (X_1^+, X_1^-), \dots, (X_n^+, X_n^-) \rangle$ is a list of *polarized interfaces*, each of which is defined as a pair of disjoint finite sets of names (X_i^+, X_i^-) . The pair (X_0^+, X_0^-) contains the global names. We define $X^+ \triangleq \biguplus_{i=1}^n X_i^+$ and $X^- \triangleq \biguplus_{i=1}^n X_i^-$, $width(X) \triangleq n$.

Two interfaces X, Y can be juxtaposed yielding a new interface, as follows:

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

A *signature* defines the basic building blocks of a bigraph; in our case, these will represent processes, containers, networks, etc.. Formally, a signature \mathcal{K} is a set of elements $c \in \mathcal{K}$, called *controls*, each with a *polarized arity* $c = \langle n^+, m^- \rangle$; intuitively, these represent the input and output ports of the control, respectively.



Figure 1: Signature for container bigraphs.

Given a signature, we can consider bigraphs built using these basic blocks. A *local directed bigraph* (ldb) $B : I \rightarrow O$ is a tuple $B = (V, E, ctrl, prnt, link)$, where

- I and O are the *inner* and *outer* interfaces respectively;
- V and E are the sets of *nodes* and *edges* respectively;
- $ctrl : V \rightarrow \mathcal{K}$ is the *control* map, assigning a control type to each node of the bigraph;
- $prnt : width(I) \uplus V \rightarrow V \uplus width(O)$ is the *parent* map, representing the hierarchy between nodes and their position in the external interface;
- $link : Pnt(B) \rightarrow Lnk(B)$ is the *link* map, connecting *points* (i.e., positive ports on nodes and inward names on interfaces) to *links* (i.e., negative ports and outward names).

Let $B_1 : I_1 \rightarrow O_1, B_2 : I_2 \rightarrow O_2$ be two ldb. Their *juxtaposition* $B_1 \otimes B_2 : I_1 \otimes I_2 \rightarrow O_1 \otimes O_2$ is defined as $B_1 \otimes B_2 = (V_1 \uplus V_2, E_1 \uplus E_2, ctrl_1 \uplus ctrl_2, prnt_1 \uplus prnt_2, link_1 \uplus link_2)$.

Let $B_1 : X \rightarrow Y, B_2 : Y \rightarrow Z$ be two ldb. Their *composition* $B_2 \circ B_1 : X \rightarrow Z$ is defined as the bigraph $(V_1 \uplus V_2, E_1 \uplus E_2, ctrl_1 \uplus ctrl_2, prnt, link)$, where $prnt : width(X) \uplus V \rightarrow V \uplus width(Z)$ and $link : Pnt(B_2 \circ B_1) \rightarrow Lnk(B_2 \circ B_1)$ are defined as expected.

Thus, (directed) bigraphs can be seen as particular data structures, parametric in the given signature. Several tools and libraries have been implemented for representing and manipulating these data structures, see e.g. [9, 11, 10], and in particular JLibBig (<https://bigraphs.github.io/jlibbig/>), a Java library for bigraphs which is the basis of the current work [12].

Now that we have defined the algebraic framework of our model, we can introduce a signature for containers. The signature we consider in this paper is the one presented in [2]:

$$\mathcal{K} = \{container : (0, 1), process_{r,s} : (r, s), request : (1, 1), network : (2, 0), volume : (2, 0)\}$$

This signature is graphically depicted in Figure 1. These basic elements can be nested and connected, as defined above, yielding bigraphs such as the one in Figure 2. This bigraph has one *root*, represented by the red dotted rectangle. Under this root there is one container node, which contains three process nodes, one volume node, two network nodes, a request node, and one *site* (the gray area). Arrows connect node ports and names, respecting their polarity. The intended meaning of arrows is that of “resource accesses”, or dependencies. In this example, the container offers services to (i.e., accepts requests from) the surrounding environment on ports p_1, p_2, p_3 , and needs to access a volume v and two networks n_1, n_2 . The site is a “hole” that can be filled by another bigraph which can access to services offered inside the container through s_1, s_2 and provide resources which $proc_3$ can access through r_1 . Filling a hole with another bigraph corresponds to composition, and as such it is subject to precise formal conditions, similar to composition of typed functions; in particular, a name of one interface can be connected to that of another interface if their polarity is the same.

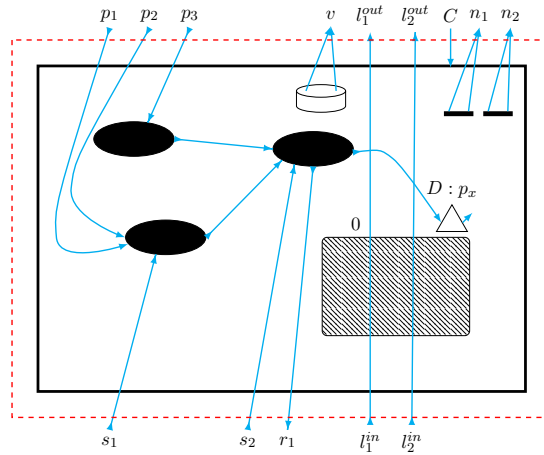


Figure 2: A bigraph representing a container. Inner interface is $\langle\langle\emptyset, \emptyset\rangle, (\{s_1, s_2, l_1^{in}, l_2^{in}\}, \{r_1\})\rangle$, and outer interface is $\langle\langle\emptyset, \emptyset\rangle, (\{v, l_1^{out}, l_2^{out}, n_1, n_2\}, \{p_1, p_2, p_3, C\})\rangle$.

```

1  version: "2"
2  services:
3    wp:
4      image: wordpress
5      links:
6        - "db"
7      ports:
8        - "8080:80"
9      networks:
10     - front
11     volumes:
12     - datavolume:/var/www/data:ro
13   db:
14     image: mariadb
15     expose:
16     - "3306"
17     networks:
18     - front
19     - back
20   pma:
21     image: phpmyadmin/phpmyadmin
22     links:
23     - "db:mysql"
24     ports:
25     - "8181:80"
26     volumes:
27     - datavolume:/data
28     networks:
29     - back
30   networks:
31   front:
32     driver: bridge
33   back:
34     driver: bridge
35   volumes:
36   datavolume:
37     external: true

```

Figure 3: Example of docker-compose.yml configuration file.

A relevant example of bigraph composition is given by the composition of containers, as performed by, e.g., `docker compose`. In this case, the context bigraph can be obtained automatically from the `docker-compose.yml` file. As an example (taken from [2]), let us consider the `docker-compose.yml` in Figure 3. Its corresponding “context” bigraph is shown in Figure 4(a). This bigraph has one root (representing the whole resulting system), as many holes as components (“services”) to be assembled, the (possibly shared) networks and volumes that each container requires, and exposes the (possibly renamed) ports to the external environment. Three bigraphs with the correct interfaces (Figure 4(b)) can be composed into the environment, yielding the system in Figure 4(c). This resulting system can be seen as a “pod”, and which can be composed into the site (of the right interface) of other bigraphs, in a modular fashion.

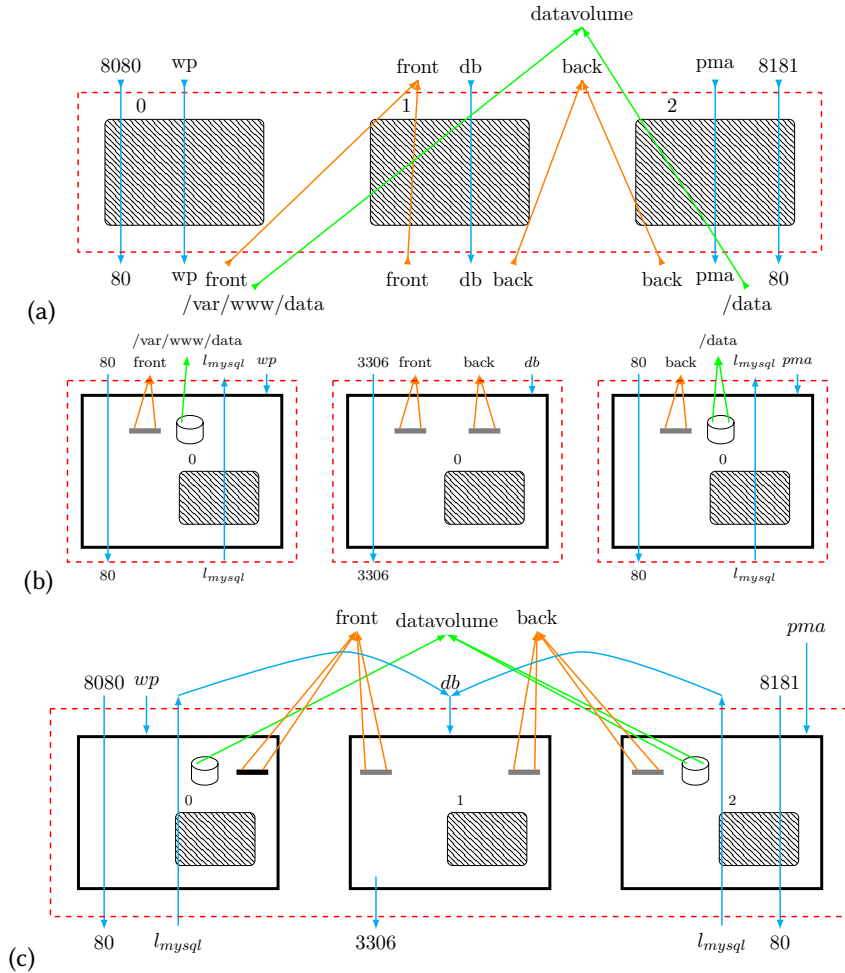


Figure 4: Example of composition: (a) the composition environment (corresponding to the YAML file in Figure 3); (b) the containers to be assembled; (c) the result of the composition.

3. JBF Specification language

In order to make the whole system scalable, modular and to allow for the conservation of the single modules, we need a specification language that allows us to keep all the necessary informations, including those not representable in a bigraph, i.e., the behaviour of a node in a communication. We want this language to be based on a standard specification language, in order to improve compatibility. Furthermore we want this language to be expressive enough to be able to describe a broad class of behaviours.

We thus introduce the *J*JSON *B*igraph *F*ormat (JBF). This language is based upon the JSON Graph Format (JGF) [15], which is a standard format for graph structures: it allows us to describe nodes, edges and the metadata of graphs. JBF leverages on these metadata objects to describe the signature and other specific informations of directed bigraphs, while the *bare structure* of JBF (Fig. 5a) is the same as JGF's for compatibility with the standard.

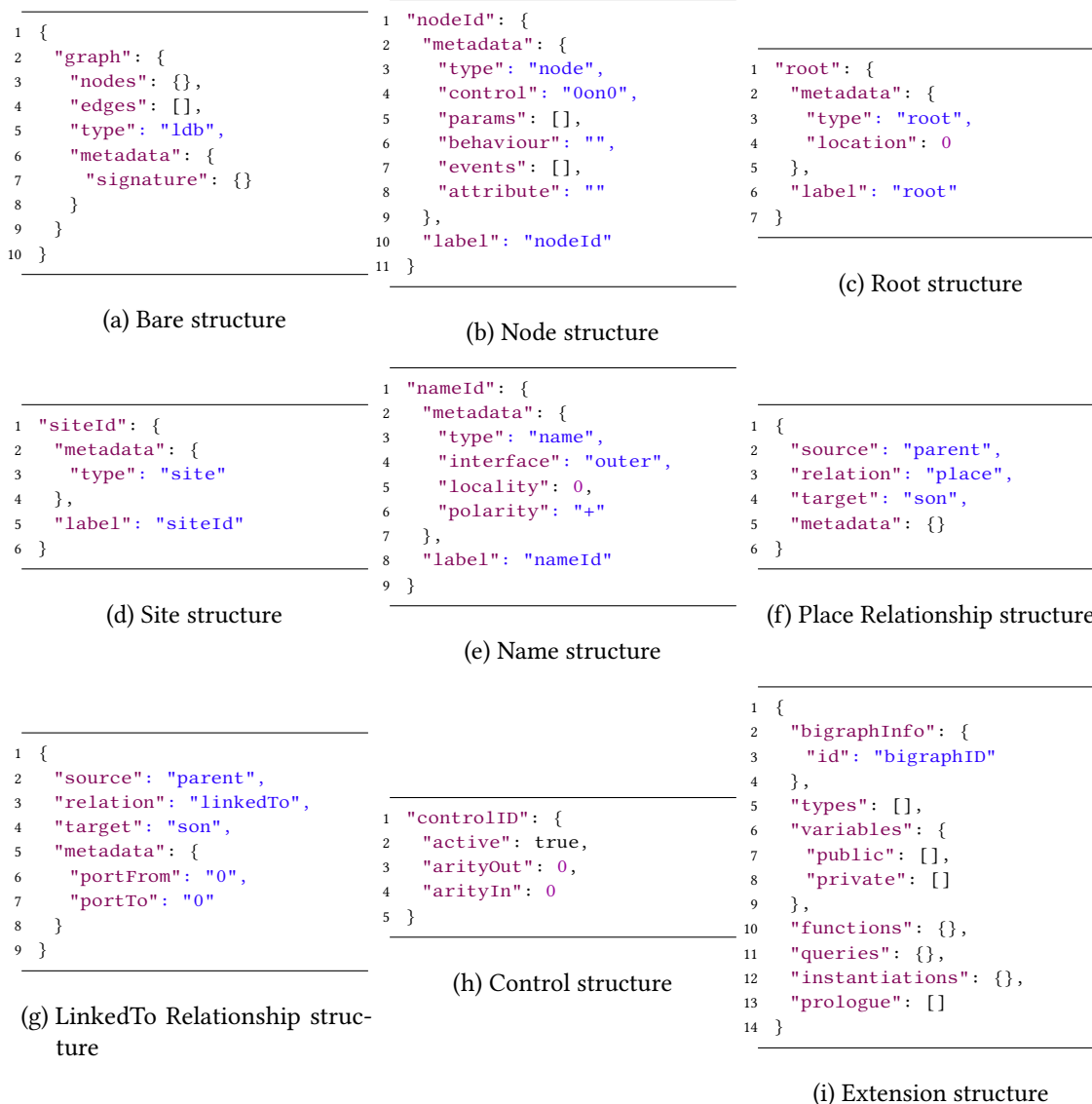


Figure 5: Structures of JBF

In Fig. 5b it is shown the *node structure*: we use metadata objects to describe the node properties that are not representable in JGF (e.g. the type of the node, its control and the data needed for security checks). The label object is used to give a meaningful name to the node.

The *root structure* is provided in Fig. 5c. The location object must be a progressive integer from 0 to N , where 0 is the global location. Likewise, we describe sites (Fig. 5d).

In Fig. 5e we focus on the *name structure*. The interface object must be either “outer” or “inner”. The locality object has the same role it has in the root structure. The polarity object must be “+” or “-”: a “+” indicates an outgoing edge in an outer interface and an ingoing edge in an inner interface, while a minus means the opposite.

There are two types of edges in JBF: *Place Relationships* and *LinkedTo Relationships*. For the *Place Relationship structure* (Fig. 5f) we reuse the JGF default fields. The source and target objects identify the parent and son IDs. The relation object must be set to “place” to indicate that the edge we are describing is a place relationship. The metadata object must be empty.

In Fig. 5g we show the *LinkedTo Relationship structure*: we have to use the metadata object to describe the information that can not be represented in standard JGF. The relation object must be of value “linkedTo”. The portFrom and portTo objects must be progressive integers and they represent the ports used by the source and the target for the connection. If a *name* is involved in the connection the port must not be specified since *names* do not have ports.

The *Control structure* is shown in Fig. 5h: it is part of the signature object inside the metadata of the bigraph. This allows us to deviate from the standard JGF specification in order to describe the control of the bigraph. The active object is a boolean that indicates whether the control is active or not. The arityOut and arityIn objects must be progressive integers and they represent the cardinality of outgoing and incoming edges of the control.

Protocol specification Since we want to keep the standard provided by JGF unchanged, we could not express all the properties necessary for the security checks in a single JSON file. Instead, we create an extension to the JBF specification which allows us to describe the remaining properties. This extension takes the form of a separate JSON file. If we want to verify the security properties of a directed bigraph, we need to provide to the system both its JBF specification and its extension.

The bare structure of this extension is provided in Fig. 5i. The bigraphInfo object is used to set the unique ID of the bigraph so as to simplify the identification of the bigraph and its storing in a database. The types object is used to declare the types used in the verification process. The variables object is used to declare the variables which could be both public and private. The functions object is used to declare the functions while the queries object is used to declare the queries which can be of three types: “query”, “attacker” and “equation”. The instantiations object is used to declare the instantiations of the variables, in order to relate the abstract variables to the concrete ones. Finally, the prologue object can be used to declare code that will be executed before the main process in the verification backend.

4. DBCChecker: Tool architecture

As shown in the previous sections, when it comes to representing a container system, bigraphs are a natural choice. The JLibBig library is the ideal candidate for representing and manipulating bigraphs due to its flexibility and extensibility. However, to fulfil our specific needs, we have extended JLibBig by adding new features, including an Import/Export feature and a Verification feature, which are divided into five modules, as depicted in Fig. 6.

The Import/Export functionality is facilitated by the I/O module, which is one of the most important components of our system. It is responsible for importing and exporting Bigraphs in a standard format, specifically the JSON Bigraph Format (JBF) which we have defined in Section 3. The I/O module is designed to be used independently of the verification functionality. Furthermore, to make sure that it remains as general as possible and not limited to our specific

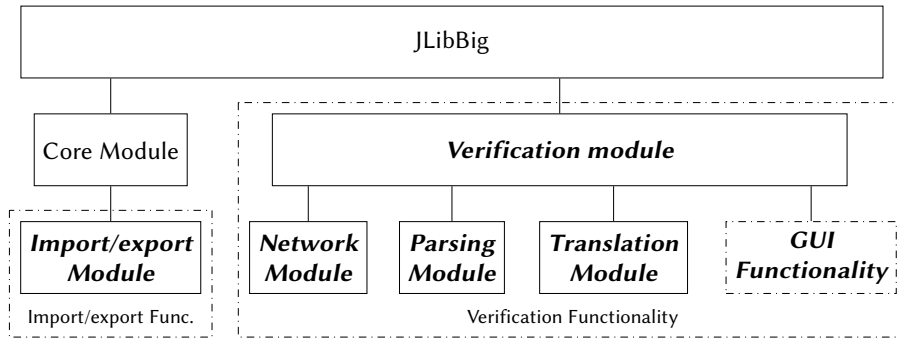


Figure 6: JLibBig module structure. Lines represent inclusions; bold modules have been developed in this work. GUI Functionality is not described in this paper.

purposes, we have set up extension points, often represented by interfaces, which can be used to extend the Import/Export functionality to bare bigraphs.

The Networking module is a service component that manages all communications between the library and external components, including the user. To ensure portability and isolation, all communications are performed through REST methods.

The Parsing module extends the Import/Export module and is responsible for parsing additional information from the directed Bigraph necessary for security checks, and represented in the JBF extension described in Section 3.

The Translation module is responsible for translating the parsed information into a file that can be processed by ProVerif. This step is critical as it is the connection point between Bigraphs and ProVerif. Furthermore, our system is modular and can be adapted to different verification backends. This can be achieved by suitably modifying the Translation module, in order to produce models for alternative protocol verifiers. Given the relevance of this module, in Fig. 7, we provide an UML describing the flow of the parsing and translation process. When the User wants to verify a system, he provides the two JBF to the system. The core parsing module extracts the standard properties of the system and creates a Bigraph Object, while the parsing module specific for security properties parses the behaviour of every node and all the other related properties and save them in records. Lastly the Translation module use these records to create the input file in ProVerif language (.pv). When the file is created the translation is completed and the tool can call ProVerif and check the security properties of the system.

The Verification module is the main actor and controller of our system, responsible for the entire verification process. It uses all the previous modules except the I/O module which is unrelated to the verification process. In Fig. 8, we provide a sequence diagram of a verification request. When the User ask for a verification, if the data contain errors, then the system terminates the operations returning an error. Otherwise, it will process the data, constructs a complete model of the container system, and sends it to the backend (ProVerif) to start the security checks. Once the verification is complete, ProVerif sends the result to the system, which could be either a success or an error. Then the system forward the result to the user.

The verification module calls the Parsing Module to parse the directed bigraph security properties. If the parsing succeeds, the verification module calls the Translation Module to

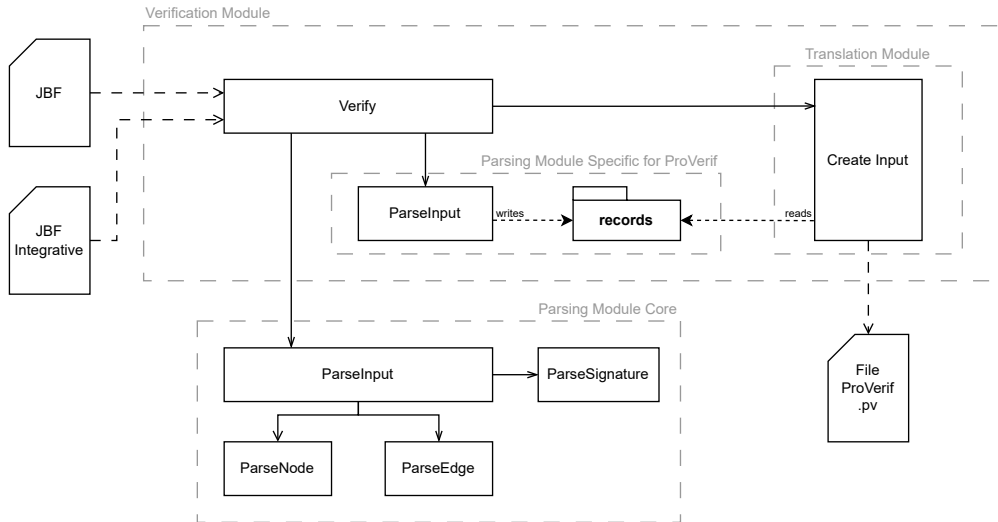


Figure 7: Parsing and Translation Flow.

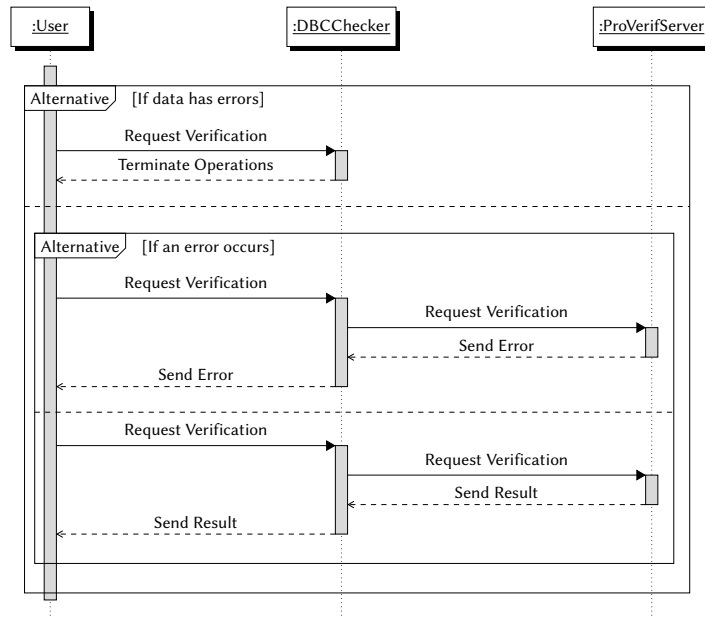


Figure 8: Verification sequence diagram

create a file for ProVerif, which is then sent to the backend for verification. Then, once the Networking Module has received the result of the verification, it sends it back to the Verification Module, which returns the result to the user.

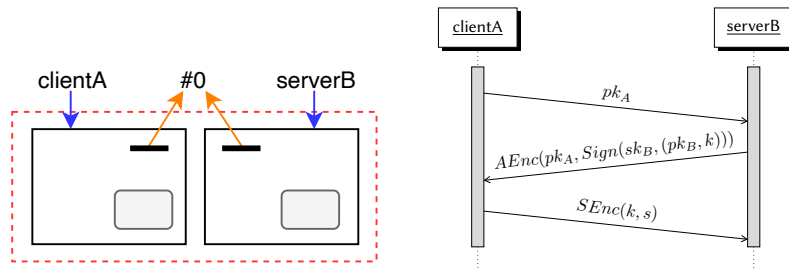


Figure 9: Secure handshake: bigraphical model of the client and server containers (left); sequence diagram of the handshake protocol (right).

5. Examples

To demonstrate the use of the DBCChecker and JBF, in this section we provide two simple example applications.

5.1. Verification of a cryptographic handshake

In this first example, we deal with a simple handshake protocol between two containers, a client A and a server B , over a shared channel. The bigraphical model of this setting, and a sequence diagram of the protocol, are shown in Fig. 9. Informally, the protocol proceeds as follows:

1. A sends its public key pk_A to B .
2. B generates a random key k , and sends it back to A signed with his private key sk_B and encrypted using the public key received pk_A received at step 1.
3. A decrypts the key k using her secret key sk_A , and checks the signature using the (known) public key pk_B ; if this succeeds, A uses k to encrypt and send a secret message s to B .

The security property we want to check is that an attacker (e.g., the host or another container), observing the traffic between the two containers, cannot obtain s .

Fig. 10 depicts a portion of the JBF structure. Most of the information conveyed by JBF is contained within the “metadata” object, as the node is responsible for its own behaviour and information. In this case, we have a short fragment in applied π -calculus (the specification language of ProVerif) abstractly describing the steps that each container will perform.

Fig. 11 shows the complete structure of the JBF Integrative file, which contains all the information required by ProVerif for verification that is not specific by any node. This separation facilitates the combination of multiple nodes by simply modifying the JBF Integrating file. In particular, this file declares also the security property(ies) to be proved (lines 13–15).

To verify this system of containers, the process is straightforward. The user creates two JSON files needed for the verification: one containing the description of the system (JBF) and one with the information necessary for the verification with ProVerif (JBF Integrative). These files can be created from scratch or composed and built upon existing files, since it is the container itself, within the “behaviour” field, which contains the logic of its behaviour. Subsequently, the user can submit the files to DBCChecker, which verifies the whole system obtained by the composition, and provides the user with the result of the ProVerif verification.

```

1 "clientA": {
2   "metadata": {
3     "type": "node",
4     "control": "1on0",
5     "params": ["pkA:pkey", "skA:skey",
6               "pkB:spkey"],
7     "behaviour": "!(out (#0+, pkA);
8                   in (#0+, x : bitstring);
9                   let y = adec(x, skA) in
10                  let (=pkB, k : key) = checksign(y,
11                pkB) in
12                  out (#0+, senc(s, k))).",
13   "attribute": ""
14 },
15 "label": "clientA"
16 }

```

```

1 "serverB": {
2   "metadata": {
3     "type": "node",
4     "control": "1on0",
5     "params": ["pkB:spkey", "skB:sskey"],
6     "behaviour": "!(in(#0+, pkX : pkey);
7                   new k : key;
8                   out(#0+, aenc(sign((pkB, k), skB),
9                   pkX));
10                  in(#0+, x : bitstring);
11                  let z = sdec(x, k) in 0 ).",
12   "attribute": ""
13 },
14 "label": "serverB"
15 }

```

Figure 10: JBF specifications of containers clientA and serverB.

```

1 {
2   "bigraphInfo": { "id": "handshake" },
3   "types": [ "skey", "pkey", "sskey", "spkey", "key" ],
4   "variables": {
5     "public": [],
6     "private": ["s:bitstring"]
7   },
8   "functions": {
9     "senc": "fun senc(bitstring, key) : bitstring.
10            reduc forall m : bitstring, k : key; sdec(senc(m, k), k) = m.",
11     "aenc": "fun pk(skey) : pkey.
12            fun aenc(bitstring, pkey) : bitstring.
13            reduc forall m : bitstring, k : skey; adec(aenc(m, pk(k)), k) = m.",
14     "sign": "fun spk(sskey) : spkey.
15            fun sign(bitstring, sskey) : bitstring.
16            reduc forall m : bitstring, k : sskey; getmess(sign(m, k)) = m.
17            reduc forall m : bitstring, k : sskey; checksign(sign(m, k), spk(k)) = m."
18   },
19   "queries": {
20     "query1": { "attacker": "s" }
21   },
22   "instantiations": {
23     "clientA": {"pkA": "pkA", "skA": "skA", "pkB": "pkB" },
24     "serverB": {"pkB": "pkB", "skB": "skB" }
25   },
26   "prologue": [
27     "new skA : skey;",
28     "new skB : sskey;",
29     "let pkA = pk(skA) in out(c, pkA);",
30     "let pkB = spk(skB) in out(c, pkB);"
31   ]
32 }

```

Figure 11: JBF - Integrative configuration.

In this case, ProVerif finds an attack on s , providing also the execution trace that an attacker can follow to obtain s , as shown in Fig. 12. In this attack trace ProVerif shows that the protocol examined is vulnerable to Man in the Middle Attacks. The Attacker pretends to be clientA towards serverB and vice versa, so that, when the serverB sends the session key the attacker obtain it and, in the next step decrypts the secret message sent by clientA.

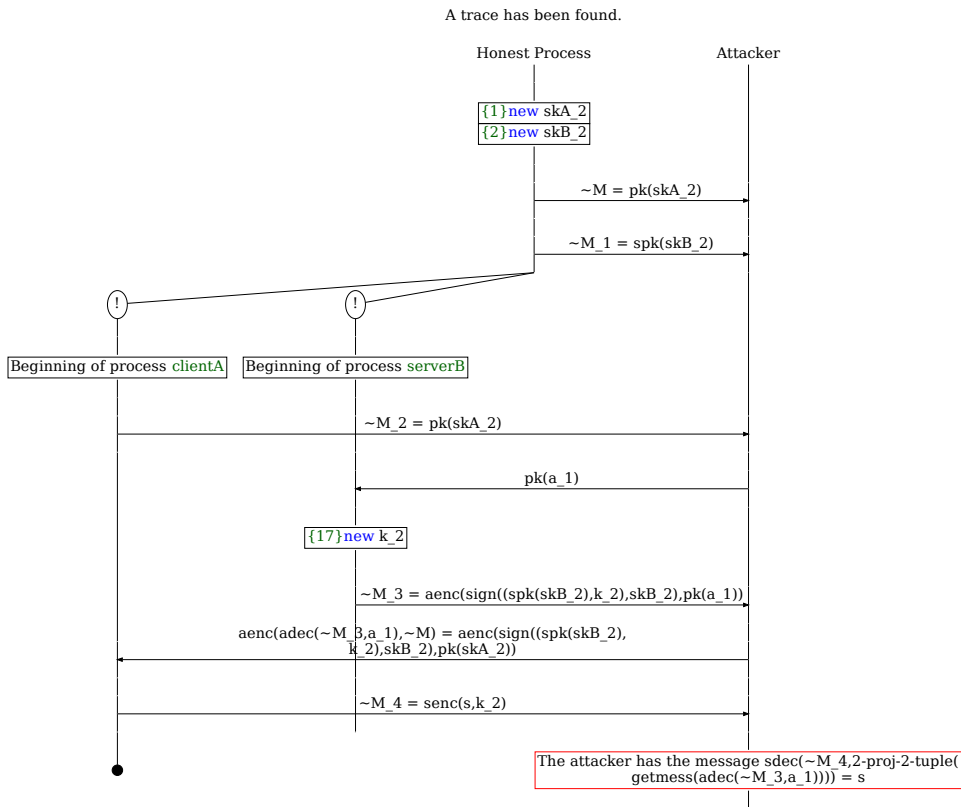


Figure 12: Attack trace for the handshake protocol of Fig. 9.

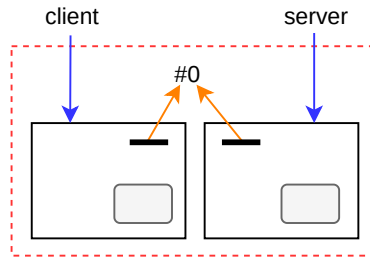


Figure 13: The bigraphical model of the client and server containers with the private (i.e. not exposed on the interface) communication channel.

5.2. Verification of information leakage between containers

In this example, we consider two containers that communicate over a *private* shared channel. The exchange is trivial: the client sends to the server a secret piece of information. The bigraphical model of this scenario is shown in Fig. 13. We define the containers contracts using JBF, a partial view of which is given in Fig. 14.

The global property we want to check is that no attacker can obtain the secret *data*.

```

1 "client": {
2   "metadata": {
3     "type": "node",
4     "control": "1on0",
5     "properties": {
6       "params": [],
7       "behaviour": "new data:bitstring;
          out(#0-, data).",
8     "events": [],
9     "attribute": ""
10  }
11 },
12 "label": "client"
13 },

```

```

1 "server": {
2   "metadata": {
3     "type": "node",
4     "control": "1on0",
5     "properties": {
6       "params": [],
7       "behaviour": "in(#0-,
          data_received:bitstring).",
8     "events": [],
9     "attribute": ""
10  }
11 },
12 "label": "server"
13 },

```

Figure 14: JBF specifications of the client and server containers.

```

1 Process 0(that is, the initial process):
2 (
3   {1}in(privateNetwork, data_received: bitstring)
4 ) | (
5   {2}new data: bitstring;
6   {3}out(privateNetwork, data)
7 )
8
9 -- Query not attacker(data[]) in process 0.
10 Translating the process into Horn clauses...
11 Completing...
12 Starting query not attacker(data[])
13 RESULT not attacker(data[]) is true.
14 -----
15 Verification summary:
16 Query not attacker(data[]) is true.
17 -----

```

Figure 15: Proverif Verification Results for the system of Fig. 13.

Since the channel is private (i.e. not exposed to any attacker), the property is trivially verified, as confirmed also by DBCChecker: the query is verified by Proverif, as shown in Fig. 15.

Let us suppose we want to add another functionality to the system, e.g., we want to log the requests from the client to the server for legal compliance reasons. This can be accomplished by adding a new container that listens on the same network where the client-server exchanges are made and publishes, on another network, a log entry for each request. The new bigraphical model is shown in Fig. 16. The actual logger implementation is not fixed: different contracts can lead to different system behaviour, some of which may prove problematic.

Let us suppose the contract specification of the logger we want to introduce is the one in Fig. 17. Since this implementation writes the entire request to the public channel, the secret *data* is leaked. This can be seen by the attack trace in Fig. 18. To solve this problem it is necessary to choose another logger that, e.g., does not leak the entire request. This is well-known in the industry; for example the standard logger for GitHub Actions scans the log message body for GitHub Secrets and removes them from the final output.

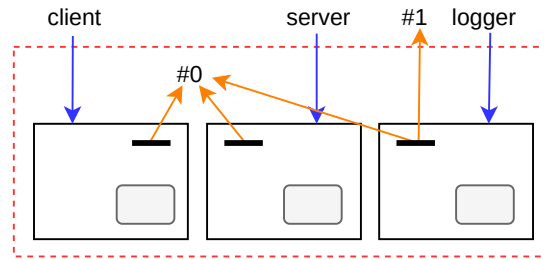


Figure 16: The bigraphical model of the client-server system, extended with the logger container.

```

1  "logger": {
2    "metadata": {
3      "type": "node",
4      "control": "2on0",
5      "properties": {
6        "params": [],
7        "behaviour": "in(#0-,data_toLog:bitstring); out(#0-,data_toLog); out(#1+,data_toLog).",
8        "events": [],
9        "attribute": ""
10     }
11   },
12   "label": "logger"
13 }

```

Figure 17: JBF specification of the logger container.

This example shows that composing containers is a complex and dangerous operation: we started with a secure container system and we added another container, which is not malicious *per se*. While apparently the composition is safe, DBCChecker proves that it is not: the extended system violates the security property that the original one satisfies.

6. Conclusions

In this work, we have presented DBCChecker, a prototype tool for verifying security properties of systems obtained by composition of containers. To this end, we have introduced a specification language, called *JSON Bigraph Format* (JBF), inspired by formal models for containerized architectures based on *bigraphs*. This formalism is capable to express at once the interfaces and connections of containers and the relevant behavioural aspects of their interfaces. From these specifications, the DBCChecker tool builds a model of the overall system, which can be verified in ProVerif, to check that the overall system satisfies the required properties.

Future work. We are currently working on adding support for other kinds of bigraphs using the extension point and the interfaces we have prepared in this work. Another possibility is to add support for quantitative aspects, taking advantage of stochastic semantics and the algorithm for computing optimal embeddings implemented in jLibBig [12, 16, 17].

Another important goal is simplifying the user interaction with the tool and help the modelisation of the container network. At the moment DBCChecker does not offer an easy way to

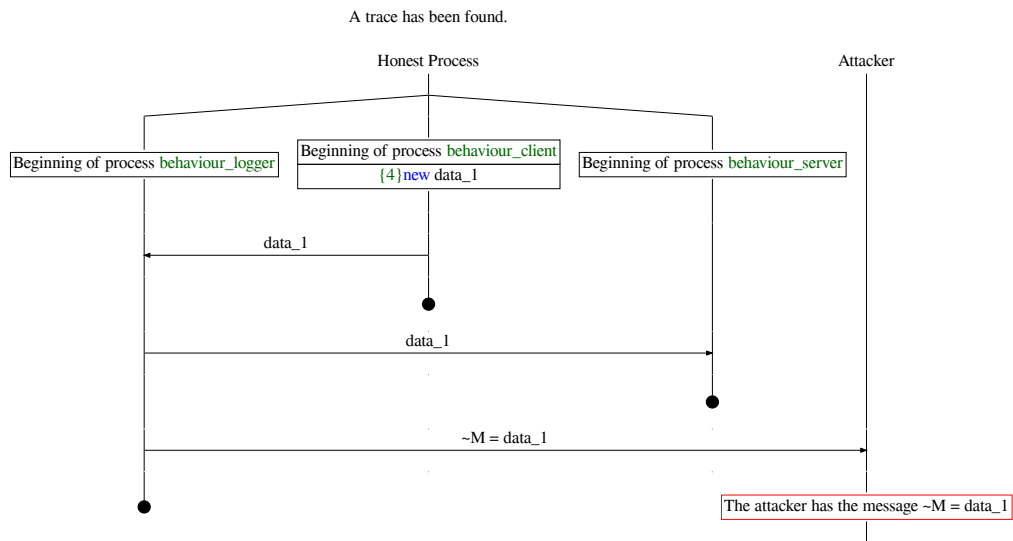


Figure 18: Attack trace for the leaking container systems.

model complex systems: the implementation of a GUI could be a major step towards a complete and user-friendly toolkit. Along this line, another improvement would be to integrate the system with a network discovery tool, in order to simplify the process of modelling and verifying large and dynamic containers based systems. This would result in a semi-automatic bigraph creation, allowing the user to concentrate upon formalizing the security properties to be verified.

References

- [1] B. Blanchet, Modeling and verifying security protocols with the applied pi calculus and ProVerif, *Found. Trends Priv. Secur.* 1 (2016) 1–135.
- [2] F. Burco, M. Miculan, M. Peressotti, Towards a formal model for composable container systems, in: *SAC, ACM*, 2020, pp. 173–175.
- [3] R. Milner, *The Space and Motion of Communicating Agents*, CUP, 2009.
- [4] D. Grohmann, M. Miculan, Directed bigraphs, in: *MFPS*, volume 173 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2007, pp. 121–137.
- [5] G. Bacci, D. Grohmann, M. Miculan, Bigraphical models for protein and membrane interactions, in: *MeCBIC*, volume 11 of *EPTCS*, 2009, pp. 3–18.
- [6] A. Mansutti, M. Miculan, M. Peressotti, Multi-agent systems design and prototyping with bigraphical reactive systems, in: K. Magoutis, P. R. Pietzuch (Eds.), *Proc. DAIS 2014*, volume 8460 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 201–208. doi:10.1007/978-3-662-43352-2_16.
- [7] H. Sahli, T. Ledoux, É. Rutten, Modeling Self-Adaptive Fog Systems Using Bigraphs, in: *FOCLASA 2019 - 17th International Workshop on coordination and Self-Adaptiveness of Software applications*, Oslo, Norway, 2019, pp. 1–16. URL: <https://hal.inria.fr/hal-02271394>.

- [8] R. Moudjari, Z. Sahnoun, F. Belala, Towards a fuzzy bigraphical multi agent system for cloud of clouds elasticity management, *Int. J. Approx. Reasoning* 102 (2018) 86–107. doi:10.1016/j.ijar.2018.07.012.
- [9] G. Bacci, D. Grohmann, M. Miculan, Dbtk: A toolkit for directed bigraphs, in: CALCO, volume 5728 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 413–422.
- [10] A. J. Faithfull, G. Perrone, T. T. Hildebrandt, Big red: A development environment for bigraphs, *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 61 (2013).
- [11] M. Sevegnani, M. Calder, BigraphER: Rewriting and analysis engine for bigraphs, in: CAV (2), volume 9780 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 494–501.
- [12] A. Chiapperini, M. Miculan, M. Peressotti, Computing (optimal) embeddings of directed bigraphs, *Sci. Comput. Program.* 221 (2022) 102842.
- [13] D. Grohmann, M. Miculan, Reactive systems over directed bigraphs, in: Proc. CONCUR 2007, volume 4703 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 380–394.
- [14] D. Grohmann, M. Miculan, Controlling resource access in directed bigraphs, *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 10 (2008).
- [15] JGF, JSON graph format (JGF), 2022. [Http://jsongraphformat.info](http://jsongraphformat.info).
- [16] G. Bacci, M. Miculan, Structural operational semantics for continuous state stochastic transition systems, *Journal of Computer and System Sciences* 81 (2015) 834–858. doi:10.1016/j.jcss.2014.12.003.
- [17] T. Brengos, M. Miculan, M. Peressotti, Behavioural equivalences for coalgebras with unobservable moves, *J. Log. Algebraic Methods Program.* 84 (2015) 826–852. doi:10.1016/j.jlamp.2015.09.002.