Formal Specification and Verification of Dynamic Parametrized Architectures

Alessandro Cimatti, Ivan Stojic, and Stefano Tonetta

FBK-irst
{cimatti,stojic,tonettas}@fbk.eu

Abstract. We propose a novel approach to the formal specification and verification of dynamic architectures that are at the core of adaptive systems such as critical infrastructure protection. Key features include run-time reconfiguration based on adding and removing components and connections, resulting in systems with unbounded number of components. We provide a logic-based specification of a Dynamic Parametrized Architecture (DPA), where parameters represent the infinite-state space of possible configurations, and first-order formulas represent the sets of initial configurations and reconfiguration transitions. We encode information flow properties as reachability problems of such DPAs, define a translation into an array-based transition system, and use a Satisfiability Modulo Theories (SMT)-based model checker to tackle a number of case studies.

1 Introduction

In many applications, safety-critical systems are becoming more and more networked and open. For example, many critical infrastructures such as energy distribution, air traffic management, transport infrastructures, and industrial control nowadays employ remote communication and control. Critical infrastructure protection is becoming of paramount importance as witnessed for example by related European and US frameworks [1,2] which promote actions to make critical infrastructures more resilient.

In order to be resilient, a system must be adaptive, changing its architectural configuration at run-time, due to new requirements, component failures or attacks. A reconfiguration means adding and removing components and connections, so that the resulting system has an infinite state space where each state is an architectural configuration. In this context, simple reachability properties such as the existence of information flow paths become very challenging due to the interplay between communications and reconfigurations. The design, implementation, and certification of a system with such properties are the challenges of the European project CITADEL [3].

While the literature about the formal specification of dynamic software architectures is abundant [24,5,26,10,32,21,25,8,7,9,31,33], very few works consider their formal verification and none of them provided a concrete evaluation showing the feasibility of the proposed analysis.

If the number of components is bounded, formal verification can be reduced to static verification by encoding in the state if the component is active or not with possibly an additional component to control the (de)activation (see, for example, [9]). If, instead,

new components can be added, the encoding is less trivial. In principle, parametrized verification, by verifying a system considering any number of replicas of components, seems a good candidate, but we need the capability to encode in the state the activation of an unbounded number of components.

In this paper, we propose Dynamic Parametrized Architectures (DPAs), which extend a standard architecture description of components and connections with 1) parameters and symbolic constraints to define a set of configurations, 2) symbolic constraints to define the sets of initial configurations and reconfigurations. In particular, the architectural topology is represented by indexed sets of components and symbolic variables that can enable/disable the connections, while the constraints are specified with first-order formulas with quantifiers over the set of indices.

We propose to use Satisfiability Modulo Theories (SMT)-based model checking for array-based transition systems [19,13], a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes. We define carefully the fragment of first-order logic used in the architecture description so that we can provide a translation into array-based transition systems.

In this paper, we focus on simple information "can-flow" properties over DPAs: we check if information can pass from one component to another one through a sequence of communications between connected components and reconfigurations. We automatically translate the DPA and the properties into an array-based transition system and we verify the properties with Model Checker Modulo Theories (MCMT) [19].

Summarizing, the contributions of the paper are: 1) to define a new formal specification of dynamic architectures; 2) to translate the reachability problem of dynamic architectures into reachability problems for infinite-state array-based transition systems; 3) to provide a prototype implementation and an experimental evaluation of the feasibility of the approach.

The rest of the paper is structured as follows: Sec. 2 gives an account of related work; Sec. 3 exemplifies the problem using a concrete language; Sec. 4 defines the abstract syntax and semantics of DPAs; Sec. 5 describes array-based transition systems and the translation from DPAs; Sec. 6 presents some benchmarks and experimental results; and, finally, in Sec. 7, we draw some conclusions and directions for future works.

2 Related Work

The approach closest to the one presented below is proposed in [31] as extension of the BIP (Behavior, Interaction, Priority) framework [6]. BIP has been extended for dynamic and parametrized connections in [7,23], and to allow spawning new components and interactions in [9]. The work in [31] proposes a second-order logic to specify constraints on the evolution of the architecture including creation/removal of components. However, no model checking procedure is provided to verify such dynamic architectures. In this paper, we restrict the logic, for example avoiding second-order quantification, so that the language is still suitable to describe interesting dynamics but can be translated into array-based transition systems for model checking.

Since a system architecture can be seen as a graph of connections, graph grammars [29] are a good candidate for specification of dynamic architectures. In fact, in

[26,32,21,33], the authors propose to model dynamic architectures in terms of graph transformation systems: the initial configuration is represented by a graph (or hypergraph) and reconfigurations as graph production rules, which are based on subgraph matching. In [26,32,21], there is no attempt at formal verification, while in [33] it is limited to finite-state model checking. Moreover, compared to our language, reconfigurations are limited to matching a finite subgraph which does not allow to express transition guards based on negation or updates that change sets of components. These limitations can be partly lifted considering infinite-state attributed graph grammars and related verification techniques [22]. After a first attempt to use these techniques as backends for DPAs, we concluded that in practice they do not scale very well on our benchmarks.

π-calculus [27] is another clear candidate to represent the semantics of dynamic architectures, since it has the ability to describe the creation of processes and their dynamic synchronizations. As is, it does not clearly define the topology of the network, but works such as [24,10] use it as underlying semantics for dynamic extensions of architecture specification languages. Also in this context, no previous work provided a concrete proposal for model checking showing the feasibility of the approach.

The analysis of how information can flow from one component to another is addressed in many contexts such as program analysis, process modeling, access control, and flow latency analysis. The novel challenge addressed by this paper is posed by the complexity of the adaptive systems' architectures, for which design and verification is an open problem. We propose a very rich system specification and we provide verification techniques for simple information flow properties formalized as reachability problems.

In this paper, we consider information flow as reachability, which is well studied for standard state machine models. More complex information flow properties extensively studied in the literature on security are related to the notion of *non-interference*. In the seminal work of Goguen and Meseguer [20], the simple information flow property is extended to make sure that components at different levels of security do not interfere. The verification of non-interference on DPAs is an open problem left for future work.

3 An Example of a Dynamic Parametrized Architecture

In this section, we describe an example of a dynamic architecture in an extended version of the Architecture Analysis and Design Language (AADL) [16], which is an industrial language standardized by SAE International [30]. The concrete language extension is under definition [12] within the CITADEL project, while in this paper we focus on the underlying formal specification and semantics of DPAs.

In AADL, the system is specified in terms of component types, defining the interfaces and thus the input/output ports, and component implementations, defining how composite components are built from other components, called subcomponents, composed by connecting their ports. An example of an AADL component implementation is shown in Figure 1. The system represents a network of computers, in which one is a database server that contains sensitive data; three are application servers that provide services to the clients, with two of them connected to the database server; and the others

are clients. Each client is connected to one server and may be connected to other clients. As can be seen, the number of components and that of their connections are finite and static. The specification represents a single static architecture.

We now extend the example to consider an arbitrary number of servers and clients and to consider changes in the configurations so that computers and connections can be added/removed and computers can be compromised becoming untrusted (due to a failure or attack). We consider the system to be *safe* if no information can flow from the database to the untrusted clients; otherwise the system is *unsafe*. This dynamic version of the system is specified in Figure 2 (for the moment, ignore the highlighted parts of the code), in an extended version of AADL. In particular, the extension has two layers:

1. *Parametrized Architecture*: the subcomponents can now be indexed sets (e.g., clients is a set indexed by C) and connections are specified iterating over the indices (e.g., there is a connection from servers[s] to clients[c] for each s in S, c in C); besides the sets of indices, the architecture is parametrized by other parameters that can be used in expressions that guard the connections (e.g., there exists a connection from servers[s] to clients[c] only if s = connectedTo[c]); notice that also parameters may be indexed sets.
2. *Dynamic Parametrized Architecture*: an initial formula defines the set of initial configurations; a set of reconfigurations defines the possible changes in the values of the parameters.

```
system implementation sys.impl
    subcomponents
        d: system databaseServer;
        s1: system applicationServer;
        s2: system applicationServer;
        s3: system applicationServer;
        c1: system Client;
        c2: system Client;
        c3: system Client;
        c4: system Client;
        c5: system Client;

    connections
        con1: port d.output -> s1.input;
        con1: port d.output -> s2.input;
        con1: port s1.output -> c1.input;
        con1: port s1.output -> c2.input;
        con1: port s2.output -> c3.input;
        con1: port s3.output -> c4.input;
        con1: port s3.output -> c5.input;
        con1: port c4.output -> c1.input;
        con1: port c4.output -> c2.input;
        con1: port c4.output -> c3.input;
        con1: port c4.output -> c5.input;
        con1: port c5.output -> c1.input;
        con1: port c5.output -> c2.input;
        con1: port c5.output -> c3.input;
        con1: port c5.output -> c4.input;
end sys.impl;
```

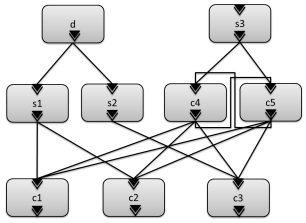

Fig. 1: Example of a component implementation in AADL

Analyzing the reconfigurations of the example (still ignoring the highlighted parts), we can see that every time an untrusted client is connected to a server, the server becomes untrusted as well. Since the connection with the database is disabled when the server is not trusted, one may erroneously think that no information can flow from the database to an untrusted client. In fact, the information can flow to the server while it is trusted, a reconfiguration may happen making the server untrusted, and the information can then flow to an untrusted client, making the system unsafe.

The version of the example with the highlighted parts is safe because it introduces two phases (represented by the protected parameter) and it allows connection to the database only in the protected mode, while reconfigurations downgrading the servers are allowed only in the unprotected mode. Proving automatically that this system is safe is quite challenging.

```
system implementation sys.impl
  parameters
    C: set of indices;                          S: set of indices;
    trustedClients: set indexed by C of bool;   trustedServers: set indexed by S of bool;
    connectedTo: set indexed by C of index;     protected: bool;
  subcomponents
    d: system databaseServer;
    servers: set indexed by S of system applicationServer;
    clients: set indexed by C of system Client;
  connections
    con1: port d.output -> servers[s].input if protected and trustedServers[s] for s in S;
    con2: port servers[s].output -> clients[c].input if s = connectedTo[c] for s in S, c in C;
    con3: port clients[c].output -> clients[r].input if not trustedClients[c] for c in C, r in C;
  initial
    not protected and
    forall(c in C, forall (s in S, (not trustedClients[c] and s = connectedTo[c]) -> (not trustedServers[s]))) and
    forall(c in C, forall (s not in S, s != connectedTo[c]));
  reconfigurations
    next(protected) = true;
    exists(s not in S, next(S) = add(S, s) and next(trustedServers[s]) = true);
    exists(s in S, not protected and next(trustedServers[s]) = false);
    exists(s in S, exists (c not in C, not trustedServers[s] and next(C) = add(C, c)
                                       and next(connectedTo[c]) = s and next(trustedClients[c]) = false));
    exists(s in S, exists (c not in C, trustedServers[s] and next(C) = add(C, c)
                                       and next(connectedTo[c]) = s and next(trustedClients[c]) = true));
    exists(s in S, exists (c in C, not trustedClients[c] and not protected
                                   and next(connectedTo[c]) = s and next(trustedServers[s]) = false));
    exists(s in S, exists (c in C, trustedClients[c] and s = connectedTo[c] and not protected
                                   and next(trustedClients[c]) = false and next(trustedServers[s]) = false));
end sys.impl;
```

Fig. 2: Example of a DPA specified in an extension of AADL

4 Formal Specification of Dynamic Parametrized Architectures

4.1 Definitions

In the following, let N be a countable set of indexes (in practice, we set $N = \mathbb{Z}$). An *index set* is a finite subset of N. Given a set S and an index set I, S is indexed by I iff there exists a bijective mapping from I to S. If S is indexed by I, we write $S = \{s_i\}_{i \in I}$. An index set parameter is a variable whose domain is the set of finite subsets of N.

Definition 1. *An architectural* configuration *is a pair (C, E), where C is a set of* components *and $E \subseteq C \times C$ is a set of* connections *between components.*

We now define a more structured version of architecture, still flat but in which components are grouped into sets. We use indexed sets of components. For example, $I = \{1, 2, 3\}$ is a set of indexes and $C = \{c_1, c_2, c_3\}$ and $C' = \{c'_1, c'_2, c'_3\}$ are two sets of components indexed by I.

Definition 2. *An architectural* structured configuration *is a pair $(\mathcal{C}, E)$, where:*

- *$\mathcal{C}$ is a finite set of disjoint sets of* components *indexed by some index sets;*
- *$E \subseteq \bigcup_{C \in \mathcal{C}} \times \bigcup_{C \in \mathcal{C}}$ is a set of* connections *between components.*

If $c \in C$ and $C \in \mathcal{C}$ we write simply (abusing notation) $c \in \mathcal{C}$.

For example, consider index sets I_1, I_2, where $I_1 = \{1, 2, 3\}$, $I_2 = \{2\}$, $\mathcal{C} = \{C_1, C_2, C_3\}$, $C_1 = \{c_{1i}\}_{i \in I_1}$, $C_2 = \{c_{2i}\}_{i \in I_1}$, $C_3 = \{c_{3i}\}_{i \in I_2} = \{c_{32}\}$, $E = \{\langle c_{1i}, c_{2i} \rangle \mid i \in I_1\}$.

Definition 3. *A* system of parameters *is a pair $(\mathcal{I}, \mathcal{V})$ where $\mathcal{I}$ is a finite set of symbols for index set parameters and $\mathcal{V}$ is a finite set of symbols for indexed sets of parameters, where each $V \in \mathcal{V}$ is associated with an index set parameter symbol $I_V \in \mathcal{I}$ and with a sort $\mathtt{sort}_V$ (in practice, $\mathtt{sort}_V \in \{\mathtt{bool}, \mathtt{int}\}$).*

Definition 4. *An* assignment *to a system of parameters $(\mathcal{I}, \mathcal{V})$ is a tuple $\mu = (\mu_{\mathcal{I}}, \{\mu_V\}_{V \in \mathcal{V}})$, where*

- $\mu_{\mathcal{I}} : \mathcal{I} \to \{S \subset N : S \text{ finite}\}$;
- *For $V \in \mathcal{V}$, $\mu_V : \mu_{\mathcal{I}}(I_V) \to R(sort_V)$ (in practice, $R(\texttt{bool}) = \mathbb{B} = \{\top, \bot\}$ and $R(\texttt{int}) = \mathbb{Z}$).*

The following definitions refer to formulas of the *logic for systems of parameters* and the evaluation (under an assignment μ) $[\![\cdot]\!]_\mu$ of its expressions and formulas. These are defined later in Sec. 4.2.

Definition 5. *A* parametrized architecture *is a tuple $A = (\mathcal{I}, \mathcal{V}, \mathcal{P}, \Psi, \Phi)$ where*

- *$(\mathcal{I}, \mathcal{V})$ is a system of parameters;*
- *$\mathcal{P}$ is a finite set of parametrized indexed sets of components; each set $P \in \mathcal{P}$ is associated with an index set $I_P \in \mathcal{I}$;*
- *$\Psi = \{\psi_P(x)\}_{P \in \mathcal{P}}$ is a set of formulas (*component guards*) over $(\mathcal{I}, \mathcal{V})$ and a free variable x;*
- *$\Phi = \{\phi_{PQ}(x, y)\}_{P,Q \in \mathcal{P}}$ is a set of formulas (*connection guards*) over $(\mathcal{I}, \mathcal{V})$ and free variables x, y.*

Given an assignment μ to the system of parameters $(\mathcal{I}, \mathcal{V})$, the instantiated (structured architectural) configuration defined by the assignment *μ is given by $\mu(A) := (\mathcal{C}, E)$ (we also write $(\mathcal{C}_\mu, E_\mu)$) where*

- *$\mathcal{C} = \{C_{\mu P} : P \in \mathcal{P}\}$. For all $C_{\mu P} \in \mathcal{C}$, for all indexes i, $c_{Pi} \in C_{\mu P}$ iff $i \in \mu_{\mathcal{I}}(I_P)$ and $[\![\psi_P(i/x)]\!]_\mu = \top$.*
- *for all $C_{\mu P}, C_{\mu Q} \in \mathcal{C}$, for all component instances $c_{Pi} \in C_{\mu P}$, $c_{Qj} \in C_{\mu Q}$, $(c_{Pi}, c_{Qj}) \in E$ iff $[\![\phi_{PQ}(i/x, j/y)]\!]_\mu = \top$.*

Syntactic restrictions: formulas in Ψ and Φ are quantifier-free and do not contain index set predicates $=, \subseteq$.

Example: For $\mathcal{I} = \{I_1, I_2\}$, $\mathcal{V} = \{V\}$, V a set of Boolean variables, with $I_V = I_1$, $\mathcal{P} = \{P_1, P_2, P_3\}$, $I_{P_1} = I_1$, $I_{P_2} = I_1$, $I_{P_3} = I_2$, $\psi_{P_1}(x) = \psi_{P_2}(x) = \psi_{P_3}(x) := \top$, $\phi_{P_1P_2}(x, y) := x = y \wedge x \in I_V \wedge V[x]$, $\phi_{P_1P_3}(x, y) := x \in I_V \wedge \neg V[x]$, $\phi_{P_2P_3}(x, y) = \phi_{P_2P_1}(x, y) = \phi_{P_3P_1}(x, y) = \phi_{P_3P_2}(x, y) := \bot$. By assigning $\mu_{\mathcal{I}}(I_1) = \{1, 2, 3\}$, $\mu_{\mathcal{I}}(I_2) = \{2\}$, $\mu_V(1) = \mu_V(2) = \mu_V(3) = \top$, we get the configuration in the previous example.

Definition 6. *A* dynamic parametrized architecture *is a tuple (A, ι, κ, τ), where*

- *$A = (\mathcal{I}, \mathcal{V}, \mathcal{P}, \Psi, \Phi)$ is a parametrized architecture;*
- *ι is a formula over $(\mathcal{I}, \mathcal{V})$, specifying the set of initial assignments;*
- *κ is a formula over $(\mathcal{I}, \mathcal{V})$, specifying the invariant;*
- *τ is a transition formula over $(\mathcal{I}, \mathcal{V})$, specifying the reconfiguration transitions.*

The dynamic parametrized architecture defines a dynamically changing architecture as a transition system over (structured architectural) configurations obtained by instantiation from A. The set of initial configurations is given by

$$\{\mu(A) : \mu \text{ is an assignment to } (\mathcal{I}, \mathcal{V}) \text{ such that } [\![\iota]\!]_\mu = \top \text{ and } [\![\kappa]\!]_\mu = \top\}.$$

A configuration $\mu'(A)$ is directly reachable from a configuration $\mu(A)$ iff $[\![\tau]\!]_{\mu\mu'} = \top$ and $[\![\kappa]\!]_{\mu'} = \top$.

Syntactic restrictions:

- *ι is of the form $\forall_{\underline{I}_1} \underline{i}_1 \forall_{\underline{I}_2^C} \underline{i}_2\ \alpha$, where α is a quantifier-free formula in which index set predicates $=, \subseteq$ do not appear under negation.*
- *κ is a quantifier-free formula without index set predicates $=, \subseteq$.*
- *τ is a disjunction of transition formulas of the form $\exists_{\underline{I}_1} \underline{i}_1 \exists_{\underline{I}_2^C} \underline{i}_2\ (\alpha \wedge \beta \wedge \gamma_\beta)$ where $\underline{I}_1, \underline{I}_2 \subseteq \mathcal{I}$, α is a quantifier-free formula, β is a conjunction of transition formulas of the forms 1) $I' = I \cup \{t_k\}_k$, 2) $I' = I \setminus \{t_k\}_k$, 3) $t \in I'$, 4) $V'[t] = e$, 5) $\forall_{I'_{V'}} j\ V'[j] = e$, where $I \in \mathcal{I}, I' \in \mathcal{I}'$ are variables of sort* `is` *(each variable $I' \in \mathcal{I}'$ may appear at most once), t, t_k are terms over $(\mathcal{I}, \mathcal{V})$ of sort* `idx`*, $V' \in \mathcal{V}'$ is a variable of one of the sorts $\texttt{vs}_k$ (for each V', either atoms of the form 4 appear in β, or at most a single atom of the form 5 appears: atoms of forms 4, 5 never appear together), e are terms over $(\mathcal{I}, \mathcal{V})$ of sorts $\texttt{el}_k$, and j is a variable of sort* `idx`*; furthermore, value $V'[t_k]$ of every introduced (by an atom of form 1,with $I' = I'_{V'}$ and $I = I_V$) parameter must be set in β with an atom of form 4 or 5; finally, the* frame condition *γ_β is a conjunction of*

  - *transition formulas $\forall_{I'_{V'}} j \left(\left(\bigwedge_{t \in t_{V'}} (j \neq t) \right) \rightarrow V'[j] = V[j] \right)$ for all $V' \in \mathcal{V}$ which do not appear in a conjunct of form 5 in β, where $t_{V'}$ is the (possibly empty) set of all terms which appear as indexes of V' in β, and*
  - *transition formulas $I' = I$ for all $I' \in \mathcal{I}'$ which do not appear in conjuncts of forms 1, 2 in β.*

  (In practice, when specifying the transition formulas, the frame condition γ_β is generated automatically from β, instead of being specified directly by the user.)

Example: Consider the parametrized architecture from the previous example, with $\iota := I_1 = \{1,2,3\} \wedge I_2 = \{2\} \wedge V[1] \wedge V[2] \wedge V[3]$; , $\kappa := \top$ and $\tau := \tau_1 \vee \tau_2$, where $\tau_1 := \exists_{I_1^C} i\, (I'_1 = I_1 \cup \{i\} \wedge V'[i])$ and $\tau_2 := \exists_{I_1} i\, (\neg V'[i])$. This defines the set of initial architectures which contains only the single architecture from the previous example and two transitions. The transition τ_1 adds a new index $i \in I_1^C$ into the index set I_1, adding two new components C_{1i} and C_{2i} and a new parameter $V[i]$ and sets the value of the newly added parameter $V[i]$ to $\top$, adding a connection between the two new components C_{1i} and C_{2i}. The transition τ_2 changes the value of some $V[i]$ to $\bot$, removing the connection between components C_{1i} and C_{2i} and adding connections between component C_{1i} and each of the components $C_{3j}, j \in I_2$.

Definition 7. *Given a dynamic parametrized architecture (A, ι, κ, τ), where $A = (\mathcal{I}, \mathcal{V}, \mathcal{P}, \Psi, \Phi)$, a* configuration *is an assignment to the system of parameters $(\mathcal{I}, \mathcal{V})$. For a configuration $\mu(A) = (\mathcal{C}, E)$, a* communication event *is a connection $e \in E$.*

Trace of the dynamic parametrized architecture *is a sequence $e_1, e_2, \ldots$ of configurations and communication events, such that:*

- *$e_1 = \mu_1$ is a configuration such that $\mu_1(A)$ is in the set of initial configurations.*
- *The subsequence $e_{k_1}, e_{k_2}, \ldots$ of all configurations in the trace is such that for all k_i, $e_{k_{i+1}}(A)$ (if it exists) is directly reachable from $e_{k_i}(A)$.*

– *For all communication events* $e_k = (c_k, c'_k)$ *in the trace,* $(c_k, c'_k) \in E_{e_{r(k)}}$*, where* $e_{r(k)}$*,* $r(k) := \max\{n \in \mathbb{N} : n < k, e_n$ *is a configuration*$\}$*, is the last configuration prior to* e_k*.*

Definition 8. *An instance of the* dynamic information flow problem *is a tuple* $(D, P_{src}, \rho_{src}, P_{dst}, \rho_{dst})$ *where*

– $D = (A, \iota, \kappa, \tau)$ *is a dynamic parametrized architecture;*
– $P_{src} \in \mathcal{P}_A$ *and* $P_{dst} \in \mathcal{P}_A$ *are (*source *and* destination*) parametrized indexed sets of components;*
– $\rho_{src}(x)$ *and* $\rho_{dst}(x)$ *(*source *and* destination guard*) are formulas over* $(\mathcal{I}_A, \mathcal{V}_A)$*.*

The problem is to determine whether there exists a finite trace (called information flow witness trace*)* $e_1, e_2, \ldots, e_n$ *of* D *with a subtrace* $e_{k_1} = (c_{k_1}, c'_{k_1}), \ldots, e_{k_m} = (c_{k_m}, c'_{k_m})$ *of communication events such that:*

– $c_{k_1} = c_{P_{src} i_{src}} \in C_{e_{r(k_1)} P_{src}}$ *for some* i_{src}*, and* $[\![\rho_{src}(i_{src}/x)]\!]_{e_{r(k_1)}} = \top$ *(the information originates in a source component);*
– $c'_{k_m} = c_{P_{dst} i_{dst}} \in C_{e_{r(k_m)} P_{dst}}$ *for some* i_{dst}*, and* $[\![\rho_{dst}(i_{dst}/x)]\!]_{e_{r(k_m)}} = \top$ *(the information is received by a destination component);*
– *for all* n *such that* $1 \leq n < m$*,* $c'_{k_n} = c_{k_{n+1}}$ *(the intermediate components form a chain over which the information propagates);*
– *for all* n*,* $1 \leq n < m$*, for all configurations* $e_{k'}$ *such that* $k_n < k' < k_{n+1}$*,* $c_{k_{n+1}} \in \mathcal{C}_{e_{k'}}$ *(after an intermediate component receives the information and before it passes it on, it is not replaced by a fresh component with the same index).*

If such a trace exists, we say that information may flow *from a source component which satisfies the source condition given by* P_{src}, ρ_{src} *to a destination component which satisfies the destination condition given by* P_{dst}, ρ_{dst}*.*

Syntactic restrictions: ρ_{src} *and* ρ_{dst} *are quantifier-free formulas, without index set predicates* $=, \subseteq$*.*

4.2 Logic for Systems of Parameters

In the following, we define a many-sorted first-order logic [15]. Signatures contain no quantifier symbols except those explicitly mentioned.

Syntax Theory $T_{IDX} = (\Sigma_{IDX}, \mathcal{C}_{IDX})$ of indexes with a single sort `idx` (in practice, we are using the theory of integers with sort `int` and with standard operators).

A finite number K of theories $T_{EL_k} = (\Sigma_{EL_k}, \mathcal{C}_{EL_k})$ of elements, each with a single sort $\mathtt{el}_k$ with a distinguished constant symbol $d_{\mathtt{el}_k}$ (a *default value*) (in practice, we consider the theory of booleans with sort `bool` and the theory of integers, the same as the theory T_{IDX}).

The theory $SP_{IDX}^{EL_1,\ldots,EL_K}$ (or simply SP_{IDX}^{EL}) of systems of parameters with indexes in T_{IDX} and elements in $EL_1, \ldots, EL_K$ is a combination of the above theories. Its sort symbols are `idx`, `is`, $\mathtt{el}_1, \ldots, \mathtt{el}_K$, $\mathtt{vs}_1, \ldots, \mathtt{vs}_K$, where `is` is a sort for index sets and $\mathtt{vs}_k$ is a sort for indexed sets of parameters of sort $\mathtt{el}_k$. The set of variable symbols for each sort $\mathtt{vs}_k$ contains a countable set of variables $\{V^I_{k,n}\}_{n \in \mathbb{N}}$ for each variable symbol I of sort `is` (we omit the superscript and subscripts when they

are clear from the context). The signature is the union of the signatures of the above theories, $\Sigma_{IDX} \cup \bigcup_{k=1}^{K} \Sigma_{EL_k}$, with the addition of: for sort $\mathtt{idx}$, quantifier symbols $\forall$, $\exists$, and $\forall_I$, $\forall_{I^C}$, $\exists_I$, $\exists_{I^C}$ for all variables I of the sort $\mathtt{is}$; predicate symbol $\in$ of sort $(\mathtt{idx}, \mathtt{is})$; predicate symbols $=, \subseteq$ of sort $(\mathtt{is}, \mathtt{is})$; function symbols $\cup, \cap, \setminus$ of sort $(\mathtt{is}, \mathtt{is}, \mathtt{is})$; for every $n \in \mathbb{N}$, n-ary function symbol $\{\cdot, \ldots, \cdot\}^{(n)}$ (we write simply $\{\cdot, \ldots, \cdot\}$) of sort $(\mathtt{idx}, \ldots, \mathtt{idx}, \mathtt{is})$; function symbols $\cdot[\cdot]_k, k = 1, \ldots, K$ (we write simply $\cdot[\cdot]$) of sorts $(\mathtt{vs}_k, \mathtt{idx}, \mathtt{el}_k)$.

Semantics A structure $\mathcal{M} = (\mathtt{idx}^{\mathcal{M}}, \mathtt{is}^{\mathcal{M}}, \mathtt{el}_1^{\mathcal{M}}, \ldots, \mathtt{el}_k^{\mathcal{M}}, \mathtt{vs}_1^{\mathcal{M}}, \ldots, \mathtt{vs}_k^{\mathcal{M}}, \mathcal{I}_{\mathcal{M}})$ for SP_{IDX}^{EL} is restricted in the following manner:

- $\mathtt{is}^{\mathcal{M}}$ is the power set of $\mathtt{idx}^{\mathcal{M}}$;
- each $\mathtt{vs}_k^{\mathcal{M}}$ is the set of all (total and partial) functions from $\mathtt{idx}^{\mathcal{M}}$ to $\mathtt{el}_k^{\mathcal{M}}$;
- $\in$ is interpreted as the standard set membership predicate;
- $=, \subseteq$ are interpreted as the standard set equality and subset predicates;
- $\cup$, $\cap$, $\setminus$ are interpreted as the standard set union, intersection and difference on $\mathtt{is}^{\mathcal{M}}$, respectively;
- for every $n \in \mathbb{N}$, $\mathcal{I}_{\mathcal{M}}(\{\cdot, \ldots, \cdot\}^{(n)})$ is the function that maps the n-tuple of its arguments to the set of indexes containing exactly the arguments, i.e. it maps every $(x_1, \ldots, x_n) \in (\mathtt{idx}^{\mathcal{M}})^n$ to $\{x_1, \ldots, x_n\} \in \mathtt{is}^{\mathcal{M}}$;
- $\cdot[\cdot]_k, k = 1, \ldots, K$ are interpreted as function applications: $(V[i])^{\mathcal{M}} := V^{\mathcal{M}}(i^{\mathcal{M}})$.

The structure $\mathcal{M}$ is a model of SP_{IDX}^{EL} iff it satisfies the above restrictions and $(\mathtt{idx}^{\mathcal{M}}, \mathcal{I}_{\mathcal{M}}{\restriction_{\Sigma_{IDX}}})$, $(\mathtt{el}_1^{\mathcal{M}}, \mathcal{I}_{\mathcal{M}}{\restriction_{\Sigma_{EL_1}}}), \ldots, (\mathtt{el}_K^{\mathcal{M}}, \mathcal{I}_{\mathcal{M}}{\restriction_{\Sigma_{EL_K}}})$ are models of T_{IDX}, T_{EL_1}, $\ldots, T_{EL_K}$, respectively.

Definition 9. *A* formula (resp. term) over a system of parameters $(\mathcal{I}, \mathcal{V})$ *is a formula (resp. term) of the logic SP_{IDX}^{EL} in which the only occuring symbols of sort* $\mathtt{idx}$ *are from $\mathcal{I}$ and the only occuring symbols of sort* $\mathtt{vs}_k$ *are from the set $\{V \in \mathcal{V} : \mathtt{sort}_V = \mathtt{el}_k\}$, for $k = 1, \ldots, K$. Furthermore, in the formula all accesses $V[\cdot]$ to parameters $V \in \mathcal{V}$ are* guarded parameter accesses*, i.e. each atom $\alpha(V[t])$ that contains a term of the form $V[t]$ must occur in conjunction with a guard which ensures that index term t is present in the corresponding index set: $t \in I_V \wedge \alpha(V[t])$.*

Definition 10. *A* transition formula over the system of parameters $(\mathcal{I}, \mathcal{V})$ *is a formula of the logic SP_{IDX}^{EL} in which the only occuring symbols of sort* $\mathtt{idx}$ *are from $\mathcal{I} \cup \mathcal{I}'$ and the only occuring symbols of sort* $\mathtt{vs}_k$ *are from the set $\{W \in \mathcal{V} \cup \mathcal{V}' : \mathtt{sort}_W = \mathtt{el}_k\}$, for $k = 1, \ldots, K$. Furthermore, all accesses $W[\cdot]$ to parameters $W \in \mathcal{V} \cup \mathcal{V}'$ are guarded parameter accesses (as defined in Def. 9).*

The subscripted quantifier symbols are a syntactic sugar for quantification over index sets and their complements: all occurences of the quantifiers $\forall_I i\ \phi$, $\forall_{I^C} i\ \phi$, $\exists_I i\ \phi$, $\exists_{I^C} i\ \phi$—where I is a variable of sort $\mathtt{is}$, i is a variable of sort $\mathtt{idx}$, and ϕ is a formula—are rewritten to $\forall i\, (i \in I \rightarrow \phi)$, $\forall i\, (i \notin I \rightarrow \phi)$, $\exists i\, (i \in I \wedge \phi)$, $\exists i\, (i \notin I \wedge \phi)$, respectively, after which the formula is evaluated in the standard manner.

Definition 11. Evaluation $[\![\phi]\!]_\mu$ with respect to an assignment μ to a system of parameters $(\mathcal{I}, \mathcal{V})$*, of a formula (or a term) ϕ over $(\mathcal{I}, \mathcal{V})$ is defined by interpreting I with $I^{\mathcal{M}} = \mu_{\mathcal{I}}(I)$ for every $I \in \mathcal{I}$ and interpreting $V[x]$ as follows for every $V \in \mathcal{V}$:*

$(V[x])^{\mathcal{M}} = \mu_V(x^{\mathcal{M}})$ *if* $x^{\mathcal{M}} \in \mu(I_V)$*, and* $(V[x])^{\mathcal{M}} = d^{\mathcal{M}}_{\mathtt{sort}_V}$ *otherwise. The evaluation* $[\![\phi]\!]_{\mu\mu'}$ *of a transition formula with respect to two assignments* μ*,* μ' *is defined by interpreting, in the above manner,* $(\mathcal{I}, \mathcal{V})$ *with* μ *and* $(\mathcal{I}', \mathcal{V}')$ *with* μ'*.*

5 Analysis with SMT-Based Model Checking

5.1 Background Notions on SMT-Based Model Checking

Many-sorted first-order logic of arrays The target logic for the translation is the many-sorted first-order logic [15] with theories for indexes, elements and arrays as defined in [18]. Following that paper, we fix a theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes whose only sort symbol is $\mathtt{index}$ and we fix theories $T_{E_k} = (\Sigma_{E_k}, \mathcal{C}_{E_k}), k = 1, \ldots, K$ whose only sort symbols are $\mathtt{elem}_k$, respectively. The theory $A_I^{E_1,\ldots,E_K}$ (or simply A_I^E) of arrays with indexes in T_I and elements in $E_1, \ldots, E_K$ is defined as the combination of theories $T_I, T_{E_1}, \ldots, T_{E_K}$ as follows. The sort symbols of A_I^E are $\mathtt{index}$, $\mathtt{elem}_1, \ldots, \mathtt{elem}_K, \mathtt{array}_1, \ldots, \mathtt{array}_K$, the signature is $\Sigma := \Sigma_I \cup \bigcup_{k=1}^K \Sigma_{E_i} \cup \bigcup_{k=1}^K \{\cdot[\cdot]_k\}$ where $\cdot[\cdot]_k$ are function symbols of sorts $(\mathtt{array}_k, \mathtt{index}, \mathtt{elem}_k)$. A structure $\mathcal{M} = (\mathtt{index}^{\mathcal{M}}, \mathtt{elem}_1^{\mathcal{M}}, \ldots, \mathtt{elem}_K^{\mathcal{M}}, \mathtt{array}_1^{\mathcal{M}}, \ldots, \mathtt{array}_K^{\mathcal{M}}, \mathcal{I})$ is a model of A_I^E iff $\mathtt{array}_k^{\mathcal{M}}$ are sets of all functions from $\mathtt{index}^{\mathcal{M}}$ to $\mathtt{elem}_k^{\mathcal{M}}$, respectively, the function symbols $\cdot[\cdot]_k$ are interpreted as function applications, and $\mathcal{M}_I = (\mathtt{index}^{\mathcal{M}}, \mathcal{I}\restriction_{\Sigma_I})$, $\mathcal{M}_{E_k} = (\mathtt{elem}_k^{\mathcal{M}}, \mathcal{I}\restriction_{\Sigma_{E_k}})$ are models of T_I, $T_{E_k}, k = 1, \ldots, K$, respectively.

Array-based transition systems In the following, i, j denote variables of the sort $\mathtt{index}$, $\underline{i}$ denotes a set of such variables, a denotes a variable of one of the $\mathtt{array}$ sorts, $\underline{a}$ denotes a set of such variables, notation $\underline{a}[\underline{i}]$ denotes the set of terms $\{a[i] : a \in \underline{a}, i \in \underline{i}\}$, and $\phi(x), \psi(x)$ denote quantifier free $\Sigma(x)$ formulas.

As in [18], an *array-based (transition) system (for* $(T_I, T_{E_1}, \ldots, T_{E_K})$*)* is a triple $\mathcal{S} = (\underline{a}, Init, Tr)$ where

- $\underline{a} = \{a_1, \ldots, a_n\}$ is a set of state variables of the sorts $\mathtt{array}_1, \ldots, \mathtt{array}_K$.
- $Init(\underline{a})$ is the *initial* $\Sigma(\underline{a})$-formula of the form

$$\forall \underline{i}.\phi(\underline{i}, \underline{a}[\underline{i}]). \tag{1}$$

- $Tr(\underline{a}, \underline{a}')$ is the *transition* $\Sigma(\underline{a}, \underline{a}')$-formula and is a disjunction of formulas of the form

$$\exists \underline{i} \left(\psi(\underline{i}, \underline{a}[\underline{i}]) \wedge \bigwedge_{k=1}^{n} \forall j \; a'_k[j] = t_k(\underline{i}, \underline{a}[\underline{i}], j, \underline{a}[j]) \right) \tag{2}$$

  where each t_k is a $\Sigma(\underline{a})$-term which may contain nested if-then-else operators.

Given an array-based system $\mathcal{S} = (\underline{a}, Init, Tr)$ and a $\Sigma(\underline{a})$-formula U (*unsafe* formula) of the form

$$\exists \underline{i}.\phi(\underline{i}, \underline{a}[\underline{i}]) \tag{3}$$

an instance of the *array-based safety problem* is to decide whether there exists $n \in \mathbb{N}$ such that the formula $Init(\underline{a}_0) \wedge Tr(\underline{a}_0, \underline{a}_1) \wedge \cdots \wedge Tr(\underline{a}_{n-1}, \underline{a}_n) \wedge U(\underline{a}_n)$ is A_I^E-satisfiable.

Decidability of the array-based safety problem The array-based safety problem is in general undecidable (Thm. 4.1. in [18]), but it becomes decidable under 1) the following assumptions on the theory T_I of indexes: local finiteness, closedness under substructures, decidability of SMT(T_I), 2) assumptions of local finiteness of T_E and of decidability of SMT(T_E), and 3) further assumptions on the array-based transition system under analysis (for details see Thm. 3.3. and Thm. 4.6. in [18]).

5.2 Encoding into SMT-Based Model Checking

Translation of formulas We recursively define the translation $\cdot^{\mathcal{A}}$ of formulas and transition formulas of SP^{EL}_{IDX} to formulas of A^E_I. We set the index and element sorts to correspond, i.e. $\texttt{index} := \texttt{idx}$ and $\texttt{elem}_k := \texttt{el}_k, k = 1, \ldots, K$. In practice, we set T_I to be the theory of integers (with sort $\texttt{index} = \texttt{int}$), number of element theories to $K = 2$, and we set $E_1 = T_I$ and E_2 to be the theory of Booleans (with sort $\texttt{elem}_2 = \texttt{bool}$).

- Symbols of the sorts $\texttt{idx}$ and $\texttt{el}_k, k = 1, \ldots, K$ are treated as symbols of the sorts $\texttt{index}, \texttt{elem}_k, k = 1, \ldots, K$, respectively.
- For a variable I of sort $\texttt{is}$, $I^{\mathcal{A}} := a_I$, where a_I is of the sort $\texttt{array}_{\texttt{bool}}$.
- For a variable V of sort $\texttt{vs}_k$, $V^{\mathcal{A}} := a_V$, where a_V is of the sort $\texttt{array}_{\texttt{elem}_k}$.
- For a term t of sort $\texttt{idx}$ and term T of sort $\texttt{is}$, $(t \in T)^{\mathcal{A}} := T^{\mathcal{A}_t}$.
- For terms T_1, T_2 of sort $\texttt{is}$, $(T_1 \cap T_2)^{\mathcal{A}_t} := T_1^{\mathcal{A}_t} \wedge T_2^{\mathcal{A}_t}$; analogously for $\cup$ and $\setminus$.
- For a variable I of sort $\texttt{is}$, $I^{\mathcal{A}_t} := I^{\mathcal{A}}[t^{\mathcal{A}}]$.
- $(\{e_1, \ldots, e_n\})^{\mathcal{A}_t} := \bigvee_{k=1}^{n} \left(t^{\mathcal{A}} = e_k^{\mathcal{A}}\right)$.
- For terms T_1, T_2 of sort $\texttt{is}$, $(T_1 = T_2)^{\mathcal{A}} := \forall i \; (T_1^{\mathcal{A}_i} = T_2^{\mathcal{A}_i})$, where i is a fresh variable of sort $\texttt{idx}$; analogously for $\subseteq$ which is translated using $\rightarrow$.
- For a variable V of sort $\texttt{vs}_k$ and a term t of sort $\texttt{idx}$, $(V[t])^{\mathcal{A}} := V^{\mathcal{A}}[t^{\mathcal{A}}]$.
- Other logical connectives, quantifiers and operators are present in both logics and are translated directly, e.g. $(e_1 \leq e_2)^{\mathcal{A}} := e_1^{\mathcal{A}} \leq e_2^{\mathcal{A}}$ and $(\phi_1 \wedge \phi_2)^{\mathcal{A}} := \phi_1^{\mathcal{A}} \wedge \phi_2^{\mathcal{A}}$.

Translation of a dynamic information flow problem to an array-based safety problem Given a dynamic parametrized architecture $D = (A, \iota, \kappa, \tau)$ where $A = (\mathcal{I}, \mathcal{V}, \mathcal{P}, \Psi, \Phi)$, and given an information flow problem instance $(D, P_{src}, \rho_{src}, P_{dst}, \rho_{dst})$, we generate a safety problem $(\mathcal{S}, U)$ where $\mathcal{S} = (\underline{a}, Init, Tr)$, as follows.

Given a system of parameters $(\mathcal{I}, \mathcal{V})$, we set $\underline{a}$ to be the (disjoint) union:

$$\begin{aligned} \underline{a} := &\{a_I : I \in \mathcal{I}, sort(a_I) = \texttt{array}_{\texttt{bool}}\} \\ \cup &\{a_V : V \in \mathcal{V}, sort(a_V) = \texttt{array}_{\texttt{sort}_V}\} \\ \cup &\{a_P : P \in \mathcal{P}, sort(a_P) = \texttt{array}_{\texttt{bool}}\}, \end{aligned} \tag{4}$$

of the set of boolean array symbols a_I which model index sets, the set of array symbols a_V which model sets of parameters, and the set of boolean array symbols a_P which model information taint of the component instances.

The initial formula $Init$ is set to

$$\begin{aligned} \iota^{\mathcal{A}} \wedge \kappa^{\mathcal{A}} \wedge \forall j \left(a_{P_{src}}[j] = \left(a_{I_{P_{src}}}[j] \wedge \psi_{P_{src}}(j/x)^{\mathcal{A}} \wedge \rho_{src}(j/x)^{\mathcal{A}}\right)\right) \\ \wedge \textstyle\bigwedge_{P \in \mathcal{P} \setminus \{P_{src}\}} \forall j \; a_P[j] = \bot. \end{aligned} \tag{5}$$

Here the third conjunct models the initial taint of the source components, by specifying that a source component with index j is tainted iff it is present in the system and satisfies the constraint ρ_{src}, and the last conjunct models the fact that initially all non-source components are not tainted.

Recall that $\tau = \bigvee_k \tau_k$, where τ_k are of the form $\exists_{\underline{I}_1} \underline{i}_1 \exists_{\underline{I}_2^C} \underline{i}_2 \, (\alpha_k \wedge \beta_k \wedge \gamma_{\beta_k})$ (see Def. 6). The transition formula Tr is set to

$$\bigvee_{P,Q \in \mathcal{P}} Taint(P,Q) \vee \bigvee_k Reconf_k. \tag{6}$$

Here $Taint(P,Q)$ is the following formula that models taint propagation between two connected component instances of which the first one is tainted:

$$\begin{aligned} &\exists i_1 \exists i_2 \, \big(\phi_{PQ}(i_1/x, i_2/y)^{\mathcal{A}} \wedge a_{I_P}[i_1] \wedge \psi_P(i_1/x)^{\mathcal{A}} \wedge a_{I_Q}[i_2] \wedge \psi_Q(i_2/x)^{\mathcal{A}} \\ &\wedge a_P[i_1] \wedge \forall j \, (a'_Q[j] = (j = i_2 \, ? \, \top : a_Q[j])) \wedge \textstyle\bigwedge_{a \neq a_Q} \forall j \, (a'[j] = a[j])\big). \end{aligned} \tag{7}$$

$Reconf_k$ is obtained from τ_k by the following steps.

Differentiation of primed parameter accesses We say that accesses to a primed parameter $V' \in \mathcal{V}'$ in τ_k are *differentiated* if for all pairs of conjuncts of form 4 in β_k as defined in Def. 6, $V'[t_1] = e_1$ and $V'[t_2] = e_2$, α_k contains a top-level conjunct $(t_1 \neq t_2)$, i.e., α_k is of the form $\alpha'_k \wedge (t_1 \neq t_2)$. We may assume that in τ_k, accesses to all primed parameters $V' \in \mathcal{V}'$ are differentiated. Note that if the accesses to some primed parameter $V' \in \mathcal{V}'$ in τ_k are not differentiated, then for a pair of undifferentated accesses $V'[t_1] = e_1$ and $V'[t_2] = e_2$ formula τ_k can be rewritten as a disjunction of two formulas $\tau_k^{=}$ and $\tau_k^{\neq}$ which are of the same general form as τ_k and are defined by

- $\tau_k^{=} := \exists_{\underline{I}_1} \underline{i}_1 \exists_{\underline{I}_2^C} \underline{i}_2 \left(\alpha_k^{=} \wedge \beta_k^{=} \wedge \gamma_{\beta_k^{=}}\right)$ where $\alpha_k^{=} := \alpha_k \wedge (t_1 = t_2) \wedge (e_1 = e_2)$, and $\beta_k^{=}$ is obtained from β_k by removing the conjunct $V'[t_2] = e_2$;
- $\tau_k^{\neq} := \exists_{\underline{I}_1} \underline{i}_1 \exists_{\underline{I}_2^C} \underline{i}_2 \left(\alpha_k^{\neq} \wedge \beta_k \wedge \gamma_{\beta_k}\right)$ where $\alpha_k^{\neq} := \alpha_k \wedge (t_1 \neq t_2)$.

It is easy to verify that the formulas τ_k and $\tau_k^{=} \vee \tau_k^{\neq}$ are equivalent. By continuing the rewriting recursively, τ can be transformed into a disjunction of formulas with differentiated accesses to primed parameters.

For a symbol $I' \in \mathcal{I}'$, there is exactly one conjunct in τ_k in which I' appears in the equality, and it is one of $I' = I \cup \{t_k\}_k$, $I' = I \setminus \{t_k\}_k$, or $I' = I$. In all three cases, value of I' is a function of the value of I and some terms over $(\mathcal{I}, \mathcal{V})$, and therefore the conjunct can be rewritten as $\forall j \, (j \in I' \leftrightarrow Update_I(j))$ where $Update_I$ is a term of sort `bool` over $(\mathcal{I}, \mathcal{V})$ and a free variable. For example, for the first case we have $Update_I(j) = (j \in I \vee \bigvee_k (j = t_k))$. The conjuncts in τ_k of the form $t \in I'$ can be rewritten as $Update_I(t)$. From τ_k we obtain τ'_k by performing the above rewriting of conjuncts which contain I', for all $I' \in \mathcal{I}'$. It is easy to verify that τ'_k and τ_k are equivalent formulas.

For a symbol $V' \in \mathcal{V}'$, the set of conjuncts in the τ'_k in which V' occurs is either equal to $\{V'[t_k] = e_k : k = 1, \ldots, n\} \cup \{\forall_{I'_{V'}} j \, ((\bigwedge_{k=1}^{n} (j \neq t_k)) \rightarrow V'[j] = V[j])\}$ where t_k are differentiated, or to $\{\forall_{I'_{V'}} j \; V'[j] = e_j\}$. In both cases, the set of conjuncts can be rewritten as $\forall j \, (V'[j] = Update_V(j))$, where $Update_V$ is a term of sort

$\mathtt{sort}_{V'}$; in the first case,

$$Update_V(j) := \text{if } j = t_1 \text{ then } e_1 \text{ else if } \ldots \text{ else if } j = t_n \text{ then } e_n \\ \text{else if } Update_{I_V}(j) \text{ then } V[j] \text{ else } d_{\mathtt{sort}_{V'}}$$

and in the second case $Update_V(j) := \text{if } Update_{I_V}(j) \text{ then } V[j] \text{ else } d_{\mathtt{sort}_{V'}}$. Formula τ''_k is obtained from τ'_k by performing the above rewrites for every $V' \in \mathcal{V}'$. It is easy to verify that τ''_k and τ'_k are equivalent.

From the invariant formula κ we obtain the next-state invariant κ' by first distributing set membership operator over the set operations (e.g. transforming $t \in I \cup J$ to $t \in I \vee t \in J$), and then replacing, for all $I \in \mathcal{I}$, each term of the form $t \in I$ with the term $Update_I(t)$, and replacing, for all $V \in \mathcal{V}$, each term of the form $V[t]$ with the term $Update_V(t)$. Analogously, from the formula ρ_{src} and component guards $\psi_P, P \in \mathcal{P}$ we obtain their next-state versions ρ'_{src} and $\psi'_P, P \in \mathcal{P}$ by performing the same transformations. $Reconf_k$ is set to

$$\begin{gathered} {\tau''_k}^{\mathcal{A}} \wedge {\kappa'}^{\mathcal{A}} \wedge \\ \forall j \left(a'_{P_{src}}[j] = \left(Update_{I_{P_{src}}}(j)^{\mathcal{A}} \wedge \psi'_{P_{src}}(j/x)^{\mathcal{A}} \wedge (a_{P_{src}}[j] \vee \rho'_{src}(j/x)^{\mathcal{A}})\right)\right) \\ \wedge \textstyle\bigwedge_{P \in \mathcal{P} \setminus \{P_{src}\}} \forall j \left(a'_P[j] = (Update_{I_P}(j)^{\mathcal{A}} \wedge \psi'_P(j/x)^{\mathcal{A}} \wedge a_P[j])\right). \end{gathered} \tag{8}$$

Here the last conjunct updates the information taint for all components, by setting it to true iff the component is present in the next state and it is currently tainted. The third conjunct performs the same update for source components, taking care to also taint the source components which satisfy the next-state source condition ρ'_{src}.

Finally, the unsafe formula U is set to

$$\exists i \left(a_{I_{P_{dst}}}[i] \wedge \psi_{P_{dst}}(i/x)^{\mathcal{A}} \wedge \rho_{dst}(i/x)^{\mathcal{A}} \wedge a_{P_{dst}}[i]\right), \tag{9}$$

modeling the set of states in which there exists a destination component with index i which satisfies the destination condition ρ_{dst} and is tainted.

The following theorems state that the information flow problem can be reduced to the array-based safety problem, using the above translation. The detailed proofs can be found in the extended version of the paper at `https://es.fbk.eu/people/stojic/papers/fm18`.

Theorem 1. *Problem $(\mathcal{S}, U)$, $\mathcal{S} = (\underline{a}, Init, Tr)$, which is obtained by translation from an arbitrary information flow problem, where $\underline{a}$ is given by* (4)*, $Init$ is given by* (5)*, Tr is given by* (6)*,* (7)*,* (8)*, and U is given by* (9)*, is an array-based safety problem.*

The proof amounts to the inspection of the obtained formulas, to confirm that they are indeed in the required fragment.

Theorem 2. *Let $DIFP = (D, P_{src}, \rho_{src}, P_{dst}, \rho_{dst})$ be an arbitrary instance of the dynamic information flow problem, and $ASP = (\mathcal{S}, U)$ the array-based safety problem obtained by translation from $DIFP$. There is an information flow witness trace for $DIFP$ if and only if ASP is unsafe.*

The proof involves constructing, for an information flow witness trace for $DIFP$, a counterexample (unsafe) trace of the problem ASP, and vice-versa.

Decidability The dynamic information flow problem is undecidable in general (it is straightforward to model Minsky 2-counter machines [28]), but it is decidable under certain assumptions inherited from the array-based transition systems (see the remark on decidability at the end of Sec. 5.1).

6 Experimental Evaluation

6.1 Setup

Back-end solver We use MCMT [4] version 2.5.2 to solve array-based safety problems. MCMT is a model checker for array-based systems, based on the SMT solver Yices[1]. We run MCMT with the default settings. The time-out for testing is set to 1000 seconds.

Translation implementation We have implemented in C the translation from the extended version of AADL to the input language for MCMT using the parser generator GNU Bison. The input language of MCMT is low level and as such is not suitable for manual modeling of anything but the simplest examples, being instead intended as a target language for automatic generation from specifications written in a higher level language [17]. The translation follows the same outline as its theoretical description in Sec. 5.2, but is more complicated due to the specific features, limitations and idiosyncracies of the target MCMT input language. In particular, the more constraining limitations of MCMT, in addition to the theoretical restrictions on formulas from Sec. 5.1, are:

- The initial formula can contain at most two universally quantified variables.
- The transitions can contain at most two existentially quantified variables.
- The maximum number of transitions (the disjuncts in the transition formula) is 50.
- The unsafe formula can contain at most four existentially quantified variables.
- A term can contain at most ten index variables.

Our translator inherits the above restrictions on the formulas specified in the extended AADL model. While these restrictions do not severely limit the expressivity of the language, the limitation on the maximum number of transitions limits the size of the examples that can be handled by the present version of the tool.

Hardware We have used a desktop PC based on an Intel® Core™ i7 CPU 870 clocked at 2.93GHz, with 8 GB of main memory and running Ubuntu 14.04.5 LTS.

Distribution Tarball The translator, tested models, scripts which automatically perform the translation from extended AADL to MCMT input languge and run MCMT, as well as setup and usage instructions can be found at `https://es.fbk.eu/people/stojic/papers/fm18/`.

6.2 Benchmarks and Results

In the following diagrams, arrows between (sets of) components represent connections from all components in the source set to all components in the destination set, unless

[1] `http://yices.csl.sri.com/`

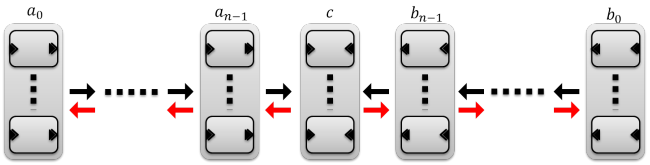

Fig. 3: Converging model

further restricted in the model description. All sets of components are dynamic, allowing addition/removal of components.

Converging Model This model contains $2n + 1$ sets of components $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c$, with the connections shown as black arrows in Fig. 3. There are also connections between all pairs of components in the same set. We test for information flow from the set $a_{\lfloor n/2 \rfloor}$ to the set b_0. The unsafe model in addition contains the connections shown as red arrows. Results for the model are in Fig. 7. We hit the MCMT limitation on the number of transitions (see Sec. 6.1) for $n = 7$ for the safe model, and for $n = 5$ for the unsafe model. Number of calls made by MCMT to the underlying SMT solver ranges from 211 (safe, $n = 1$) to 29907 (safe, $n = 6$).

Messenger Model In this model, for $n = 1$ there are two sets of components, a and b, and a singleton component m_0. m_0 models a messenger which is intended to allow components within the same set to communicate; m_0 can connect in turn to any single component in a or in b, but not at the same time. We test for information flow from set a to set b. The system as described is unsafe because m_0 can connect to some $a[i]$, disconnect, and then connect to some $b[j]$, therefore establishing a path for flow of information. The safe model removes such paths by using Boolean parameters to record whether m_0 has previously connected to components in a and b. If it has previously connected to one of these sets, then it is not allowed to connect to the other set before it is scrubbed (which is modeled by removing and re-adding it). For $n = 2$ (Fig. 4), the system is extended with another set of components c and another messenger m_1 which is shared by b and c, and we check for information flow between a and c. Results are shown in Fig. 5.

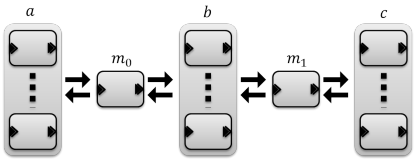

Fig. 4: Messenger model

Model	n	Time (s)	SMT calls
Messenger (safe)	1	1.894	9370
Messenger (safe)	2	TO	-
Messenger (unsafe)	1	0.240	1563
Messenger (unsafe)	2	24.639	82648
Messenger (unsafe)	3	TO	-
Network (safe)	-	0.875	3327
Network (unsafe)	-	0.171	1063

Fig. 5: Messenger and Network models results

Network Model This is the model whose safe version is specified in Fig. 2, while the highlighted parts are omitted in the unsafe version. Results are shown in Fig. 5.

Sequence Model This is a scalable example which models a sequence of n sets of components $a_0, \ldots, a_{n-1}$ (see Fig. 6 ignoring the dashed loop-back arrow). There is a connection from $a_x[i]$ to $a_y[j]$ iff $(x = y-1 \vee x = y) \wedge i < j$. We check for information flow from $a_0[0]$ to $a_{n-1}[n-2]$ in the safe version and from $a_0[0]$ to $a_{n-1}[n-1]$ in the unsafe version. The results are shown in Fig. 8. The verification of this model times out for $n = 6$ (safe) and $n = 7$ (unsafe). Number of calls to the SMT solver ranges from 116 (unsafe, $n = 1$) to 60799 (unsafe, $n = 6$).

Ring Model This model is the same as the Sequence model, but with additional connections from $a_{n-1}[i]$ to $a_0[j]$ (dashed loop-back arrow in Fig. 6) which are present only when $i < j + n$ in the safe version ($i < j + n + 1$ in the unsafe version), and we check for information flow from $a_0[0]$ to $a_{n-1}[n-2]$ in both the safe and unsafe versions. The results are shown in Fig. 9. The verification of this model times out for $n = 6$ (safe) and $n = 5$ (unsafe). Number of calls to the SMT solver ranges from 188 (unsafe, $n = 1$) to 130068 (unsafe, $n = 4$).

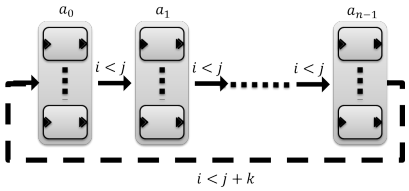

Fig. 6: Sequence and Ring models

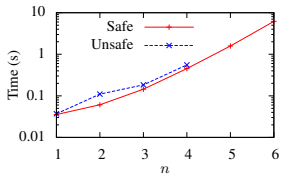

Fig. 7: Converging model results

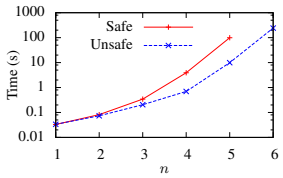

Fig. 8: Sequence model results

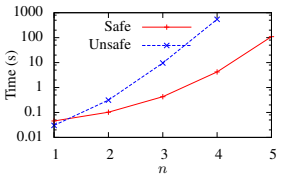

Fig. 9: Ring model results

7 Conclusions and Future Work

We propose a new logic-based specification of dynamic architectures where the architectural topology is represented by a set of parameters, while first-order formulas over such parameters define the sets of initial configurations and reconfigurations. The Dynamic Parametrized Architectures so defined can be translated into array-based transition systems, which are amenable to SMT-based model checking. We provide an initial experimental evaluation of various DPAs proving safe and unsafe cases with the MCMT model checker. The results show that the approach is feasible and promising.

As future work, we aim at trying other SMT-based model checkers such as Cubicle [13] and nuXmv [11]. We will investigate new algorithms that directly exploit the topology of the architecture. We will extend the specification to incorporate component behavior and more complex interactions, as well as more general properties. Finally, we are interested in generating certifying proofs for the safe DPAs, possibly exploiting the existing automatic generation of proofs for array-based transition systems [14].

References

1. European Programme for Critical Infrastructure Protection (EPCIP). http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2006:0786:FIN:EN:PDF. Accessed 15 Jan 2018.
2. NIST Cybersecurity Framework. http://www.nist.gov/cyberframework. Accessed 15 Jan 2018.
3. The CITADEL Project (Critical Infrastructure Protection using Adaptive MILS). http://www.citadel-project.org/. Accessed 15 Jan 2018.
4. ALBERTI, F., GHILARDI, S., AND SHARYGINA, N. A Framework for the Verification of Parameterized Infinite-state Systems. In *CEUR Workshop Proceedings* (2014), vol. 1195, pp. 302–308.
5. ALLEN, R., DOUENCE, R., AND GARLAN, D. Specifying and Analyzing Dynamic Software Architectures. In *FASE* (1998), pp. 21–37.
6. BASU, A., BOZGA, M., AND SIFAKIS, J. Modeling Heterogeneous Real-time Components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India* (2006), pp. 3–12.
7. BOZGA, M., JABER, M., MARIS, N., AND SIFAKIS, J. Modeling Dynamic Architectures Using Dy-BIP. In *Software Composition - 11th International Conference, SC 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings* (2012), pp. 1–16.
8. BRADBURY, J. S., CORDY, J. R., DINGEL, J., AND WERMELINGER, M. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, Newport Beach, California, USA, October 31 - November 1, 2004* (2004), pp. 28–33.
9. BRUNI, R., MELGRATTI, H. C., AND MONTANARI, U. Behaviour, Interaction and Dynamics. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi* (2014), pp. 382–401.
10. CANAL, C., PIMENTEL, E., AND TROYA, J. M. Specification and Refinement of Dynamic Software Architectures. In *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA* (1999), pp. 107–126.
11. CAVADA, R., CIMATTI, A., DORIGATTI, M., GRIGGIO, A., MARIOTTI, A., MICHELI, A., MOVER, S., ROVERI, M., AND TONETTA, S. The nuXmv Symbolic Model Checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), pp. 334–342.
12. CITADEL Modeling and Specification Languages. Tech. Rep. D3.1, Version 2.2, CITADEL Project, Apr. 2018.
13. CONCHON, S., GOEL, A., KRSTIC, S., MEBSOUT, A., AND ZAÏDI, F. 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* (2012), pp. 718–724.
14. CONCHON, S., MEBSOUT, A., AND ZAÏDI, F. Certificates for Parameterized Model Checking. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings* (2015), pp. 126–142.
15. ENDERTON, H. B. *A Mathematical Introduction to Logic*, second ed. Academic Press, Boston, Jan 2001.
16. FEILER, P. H., AND GLUCH, D. P. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012.

17. GHILARDI, S. MCMT v2.5 - User Manual. http://users.mat.unimi.it/users/ghilardi/mcmt/UM_MCMT_2.5.pdf, 2014. Accessed 15 Jan 2018.
18. GHILARDI, S., AND RANISE, S. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. *Logical Methods in Computer Science Volume 6, Issue 4* (Dec. 2010).
19. GHILARDI, S., AND RANISE, S. MCMT: A Model Checker Modulo Theories. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings* (2010), pp. 22–29.
20. GOGUEN, J. A., AND MESEGUER, J. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28 1982* (1982), pp. 11–20.
21. HIRSCH, D., INVERARDI, P., AND MONTANARI, U. Reconfiguration of Software Architecture Styles with Name Mobility. In *Coordination Languages and Models, 4th International Conference, COORDINATION 2000, Limassol, Cyprus, September 11-13, 2000, Proceedings* (2000), pp. 148–163.
22. KÖNIG, B., AND KOZIOURA, V. Towards the Verification of Attributed Graph Transformation Systems. In *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings* (2008), pp. 305–320.
23. KONNOV, I. V., KOTEK, T., WANG, Q., VEITH, H., BLIUDZE, S., AND SIFAKIS, J. Parameterized Systems in BIP: Design and Model Checking. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada* (2016), pp. 30:1–30:16.
24. MAGEE, J., AND KRAMER, J. Dynamic Structure in Software Architectures. In *SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996* (1996), pp. 3–14.
25. MEDVIDOVIC, N., AND TAYLOR, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Eng. 26*, 1 (2000), 70–93.
26. MÉTAYER, D. L. Describing Software Architecture Styles Using Graph Grammars. *IEEE Trans. Software Eng. 24*, 7 (1998), 521–533.
27. MILNER, R., PARROW, J., AND WALKER, D. A Calculus of Mobile Processes, I and II. *Inf. Comput. 100*, 1 (1992), 1–77.
28. MINSKY, M. L. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
29. ROZENBERG, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
30. Architecture Analysis & Design Language (AADL) (rev. B). SAE Standard AS5506B, International Society of Automotive Engineers, Sept. 2012.
31. SIFAKIS, J., BENSALEM, S., BLIUDZE, S., AND BOZGA, M. A Theory Agenda for Component-Based Design. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering* (2015), pp. 409–439.
32. WERMELINGER, M., AND FIADEIRO, J. L. Algebraic Software Architecture Reconfiguration. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Toulouse, France, September 1999, Proceedings* (1999), pp. 393–409.
33. XU, H., ZENG, G., AND CHEN, B. Description and Verification of Dynamic Software Architectures for Distributed Systems. *JSW 5*, 7 (2010), 721–728.