A Temporal Configuration Logic for Dynamic Reconfigurable Systems

Antoine El-Hokayem*
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG
Grenoble, France
antoine.el-hokayem@univ-grenoble-alpes.fr

Marius Bozga
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG
Grenoble, France
marius.bozga@univ-grenoble-alpes.fr

Joseph Sifakis
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG
Grenoble, France
joseph.sifakis@univ-grenoble-alpes.fr

ABSTRACT

Configuration logics have been proposed for the specification of architectural styles of component-based systems. We use such a logic for the specification and verification of architectural properties of dynamic reconfigurable systems. In particular, we introduce the Temporal Configuration Logic (TCL), a linear time temporal logic built from atomic formulas characterizing system configurations and temporal modalities. We study an effective model-checking procedure based on SMT techniques for a non-trivial fragment of TCL which has been implemented in a prototype runtime verification tool. We provide preliminary experimental results illustrating the capabilities of the tool on two non-trivial benchmark systems.

CCS CONCEPTS

• **Software and its engineering** → **Architecture description languages**; *Formal software verification*; **Software architectures**; • **Theory of computation** → **Modal and temporal logics**;

KEYWORDS

configuration logic, reconfigurable systems, runtime verification

ACM Reference Format:
Antoine El-Hokayem, Marius Bozga, and Joseph Sifakis. 2021. A Temporal Configuration Logic for Dynamic Reconfigurable Systems. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3442017

1 INTRODUCTION

Dynamic reconfigurable systems (DR-systems) are emerging as dynamism and reconfiguration are essential in many application areas demanding adaptation of system behavior to a changing environment, as well as self-organization to satisfy changing goals and needs. These systems are built from instances of predefined component types. As usual, each component type is specified by its behavior and its interface that defines how it can be composed with other components. The main characteristic of DR-systems is that they involve a changing number of component instances that are dynamically coordinated so that their collective behavior satisfies given global properties. Modeling and verifying such systems involves specific difficulties that are not experienced with static systems involving a fixed number of components and interactions.

DR-systems differ from static systems in that their dynamics cannot be captured as a sequence of global component states. As a rule, knowing a state snapshot of component instances does not allow deciding how the system can evolve. Additional information is needed about the connectivity of the components in the dynamically changing structure. Of course, one can observe that if the internal behavior of each component is adequately modified we can encode information about its position in the system structure. This is in fact possible to some extent, e.g., by adding to each component attributes that are updated with the names of the components directly connected to it. But this approach has its limits. No component has a global view of the system structure neither can easily infer structural parameters that determine system behavior, e.g., the total number of component instances.

Besides these considerations, there is a compelling argument against this idea of component modification. When we are reusing components to build systems, we want that they are "architecture-agnostic". That is, the way they are composed is uniquely defined by their interface without any modification of their internal behavior. Component instances belong to predefined component types and must offer the corresponding basic functionality without any restriction about their context of use. This implies in particular that coordination of components is exogenous: only the "glue" between components fully determines the way they interact.

The above remarks suggest that to capture the dynamic behavior of DR-systems, component state should be adequately enriched with information about system structure. This leads to the concept of configuration which is a tuple (U, γ, σ) where U is the set of component instances, σ is a state, i.e. a valuation of the component variables, and γ is a set of interactions between the components of U. So, a configuration in addition to the component state, defines the system architecture (U, γ) at a moment of its evolution. Any DR-system can be considered as a transition system on configurations.

We study the Temporal Configuration Logic (TCL) for the specification and validation of properties of DR-systems. This is a linear time temporal logic built from atomic formulas characterizing configurations using temporal modalities. The main difference with

*The research performed by these authors was partially funded by H2020-ECSEL grants CPS4EU 2018-IA call - Grant Agreement number 826276.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442017

ordinary temporal logic is that its models are sets of component state sequences while the models of TCL are sequences of configurations (U, γ, σ). Thus TCL allows specification of properties of DR-systems as temporal logic does for static systems.

We use formulas of a Configuration Logic (CL) inspired from [17] as atomic formulas of TCL. CL is a second-order logic with component variables and component set variables. Atomic formulas of CL are either component state predicates or connectors. The latter are predicates modeling interactions as n-ary relations between component instances. For example, $\mathrm{K}(x_1, ..., x_n)$ is a connector modeling n-ary interactions of type K between component instances $x_1, ..., x_n$. The formulas of CL are built from atomic formulas with the usual logical connectives and an associative and commutative coalescing operator + used to build architectures.

The paper is organized as follows. In section 2 we introduce our underlying model of dynamic reconfigurable systems. In section 3 we introduce the configuration logic CL. We provide first the propositional fragment, without quantifiers, then, the complete setting where quantifiers are allowed. In section 4 we provide methods for checking the validity of CL on configurations using SMT-checking techniques. In section 5 we introduce the temporal configuration logic TCL and we present the method for runtime verification. We present experimental results in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 DYNAMIC RECONFIGURABLE SYSTEMS

We consider dynamic reconfigurable systems consisting of a variable number of component instances that are dynamically coordinated so that their collective behavior satisfies given properties.

As an illustrative example, consider a Master/Slave system consisting of instances of masters m_i and slaves s_j. Masters interact in an ring structure: the output of m_i is connected to the input of m_{i+1} where +1 is interpreted modulo the size of the ring. Furthermore, each master m_i can interact with disjoint sets of slave instances s_j's (see Figure 1). We assume that the system evolves dynamically to meet the following two requirements:

Load balancing: If in a configuration the number of slaves associated with master m_i exceeds at least by 2 the number of slaves of its successor in the ring and m_i has a slave s_j that is not busy, then s_j is assigned to the successor.

Reconfigurable fault-tolerance: The system components, masters or slaves, are equipped with self-detection and recovery mechanisms. When a component fails, it disconnects without disturbing the system operation. If the size of the ring is greater than 2, a master can fail and the ring connectivity is preserved after it disconnects; furthermore, when it fails all its slaves are assigned to its successor in the ring. A slave failure is detected by the corresponding master as absence of response to its requests. When a component, master or slave, recovers it is dynamically and seamlessly integrated in the system.

For the considered example, masters interact depending on their current position in the ring as their neighborhood may change dynamically due to master failure or recovery. Slave assignment to masters depends on load balancing criteria. Furthermore, if a component fails, it remains disconnected until it recovers. These remarks show that knowing the internal state of each component does not suffice to determine if an evolution rule is applicable. For instance, we need to know the size of the ring, the set of active components or the set of the slaves assigned to a master.

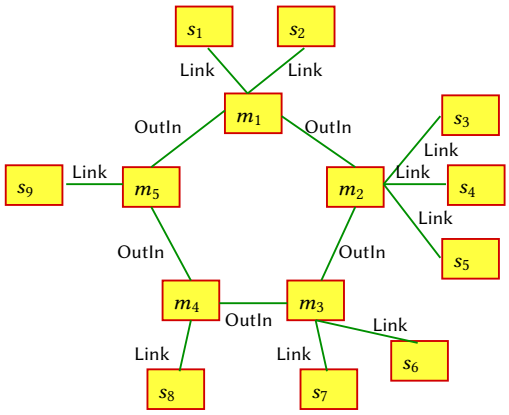

Figure 1: Master/Slave System Architecture

In previous work we have studied the problem of component-based modeling of DR-systems and proposed an expressive modeling framework implemented by the DR-BIP language [2, 6]. This language allows building DR systems from instances of architecture-agnostic component types. Coordination between components is exogenous and their behavior ignores the way they are integrated. We have shown that modeling DR-systems requires languages encompassing the following three key features:

Parametric transition system model where parameters are used to represent the interconnecting structure of component instances. As a rule, two categories of parameters are needed, respectively, parameters denoting instances of components and parameters denoting sets of instances of components. Component parameters are used to express relations between particular component instances, e.g., connection between two masters x_1, x_2 in the ring. Component set parameters are used to represent relations between arbitrarily many instances, e.g., a set parameter X to represent the set of all active masters in the ring, set parameters Y_i to represent the set of slaves connected to master x_i, etc. Both categories of parameters are necessary to describe system behavior. A valuation of the parameters occurring in a system model and a state function σ determine a configuration and the subsequent system behavior.

Multiparty interactions between components which are atomic state changes of a non-empty set of component instances resulting in the synchronous exchange of data between the involved components. In the considered example, there will be interactions between two consecutive masters in the ring and also between a master and its slaves as depicted in Figure 1. Interactions are parameterized by component and component set parameters. The latter are necessary to define groups of interacting components, e.g., the group of slaves related to a given master.

Reconfiguration operations which consist in creating/deleting component instances and updating their connections in the structure. These operations boil down to modifying parameters. For

instance, the failure of a master x results in removing x from X which represents the set of masters connected in the ring; the application of the load balancing rule results in removing a non-busy slave y from the set Y_1 of slaves connected to x_1 and adding y to the set Y_2 of the successor x_2 of x_1.

3 CONFIGURATION LOGIC CL

Configuration logic allows the specification of system configurations characterizing the way system components interact. It can be used for the description of architecture properties usually called architecture styles [12] which are sets of architectures sharing common characteristics. For instance, Master/Slave architectures share a common property characterizing the configurations consisting of sets of Master and Slave instances where each slave interacts with a single master. Similarly, we can talk about Ring, Pipe and Filter or Client/Server architecture styles.

There exists a rich literature on architecture style specification following two different directions. One originates in [9, 18], and studies the use of graph grammars and their transformations. The key idea is that the different configurations are obtained by application of rewriting rules. The other direction is declarative based on relational logic. It is represented by works using Alloy [11] (e.g. ACME [12], Darwin [8]), or OCL [23]. The presented configuration logic has some similarity with these works. Nonetheless, it relies on a minimal set of notions, and emphasizes on conceptual clarity. It adopts a strict separation between component computation and coordination structure by using connectors which are abstract n-ary connectivity predicates. It is more expressive then existing logical formalisms that are often limited to binary connection predicates, and make use of variants of first-order logic.

Configuration logic formulas characterize sets of configurations (U, γ, σ) consisting of a set of components instances U with their states σ and the set of connectors γ for their coordination. Atomic formulas are naturally state predicates and connector predicates with an arbitrary number of arguments that can be either component instances or sets of component instances. In addition to the logical connectors that have the usual set-theoretic interpretation on configurations, the logic is equipped with a coalescing operator +that allows grouping connectors in the same configuration. This combination of the coalescing operator with logical connectives confers configuration logic enhanced expressiveness and elegance in property specification.

3.1 Notations

Let C denote a finite set of component types (names). For every component type C, let $\mathcal{Q}^C$ denote the set of its states. We denote by $\mathcal{Q}$ the set of all component type states, that is, $\mathcal{Q} \triangleq \cup_{C \in \mathcal{C}} \mathcal{Q}^C$.

For every component type C, let $\mathcal{U}^C$ denote the set of instances (names) of type C. These sets are assumed pairwise disjoint, that is, $\mathcal{U}^C \cap \mathcal{U}^{C'} = \emptyset$ whenever $C \neq C'$. Let denote by $\mathcal{U}$ the set of all instances, that is, $\mathcal{U} \triangleq \cup_{C \in \mathcal{C}} \mathcal{U}^C$. For an instance u of $\mathcal{U}$ we denote by $type(u)$ its type, that is, the unique component type C such that $u \in \mathcal{U}^C$. We extend the notation to sets of instances, namely for every $U \subseteq \mathcal{U}$ we denote by $type(U)$ the set of component types occurring in U, that is, $type(U) \triangleq \{type(u) \mid u \in U\}$.

For a subset of instances $U \subseteq \mathcal{U}$, we denote by $\mathcal{Q}^U$ the set of states of U, that is, the set of mappings $\{\sigma : U \to \mathcal{Q} \mid \forall u \in U.\ \sigma(u) \in \mathcal{Q}^{type(u)}\}$ that correctly associate instances to states of their corresponding type.

Example 3.1. For the Master/Slave example, the set C contains two component types, namely Master and Slave. Their sets of states are denoted by respectively $\mathcal{Q}^{\mathsf{Master}}$, $\mathcal{Q}^{\mathsf{Slave}}$. The set $\mathcal{U}^{\mathsf{Master}}$ of instances of type Master is $\{m_1, m_2, ...\}$. The set $\mathcal{U}^{\mathsf{Slave}}$ of instances of type Slave is $\{s_1, s_2, ...\}$. For the set of instances depicted in Figure 1, a state σ is a mapping $\sigma : \{m_1, ...m_5, s_1, ...s_9\} \to \mathcal{Q}^{\mathsf{Master}} \cup \mathcal{Q}^{\mathsf{Slave}}$ such that $\sigma(m_i) \in \mathcal{Q}^{\mathsf{Master}}$, $\sigma(s_j) \in \mathcal{Q}^{\mathsf{Slave}}$.

Let $\mathcal{K}$ denote a finite set of connector types (names). Every connector type K has a signature denoted as $sign(K)$ which is a sequence $C_{i_1}...C_{i_n}2^{C_{j_1}}...2^{C_{j_m}}$ where $m, n \geq 0$ and C_{i_k}, C_{j_ℓ} are component types. Connector types have instances, that is, interactions relating tuples of component instances and/or set of component instances, compatible with their signature. For a connector type K with signature $sign(K) = (C_{i_k})_{k=1,n}(2^{C_{j_\ell}})_{\ell=1,m}$ its set of interactions denoted as Γ^K is the set of terms of the form:

$$\Big\{\ K(u_{i_1}, \cdots u_{i_n}, U_{j_1}, \cdots U_{j_m}) \mid \\ \forall k \in [1,n].\ u_{i_k} \in \mathcal{U}^{C_{i_k}},\ \forall \ell \in [1,m].\ U_{j_\ell} \subseteq \mathcal{U}^{C_{j_\ell}} \\ u_{i_1}, \cdots u_{i_n} \text{ pairwise distinct} \\ \{u_{i_1}, \cdots u_{i_n}\}, U_{j_1}, \cdots U_{j_m} \text{ pairwise disjoint}\ \Big\}$$

Moreover, we denote by Γ the set of all interactions for all connector types, that is, $\Gamma \triangleq \cup_{K \in \mathcal{K}} \Gamma^K$. For a subset of instances $U \subseteq \mathcal{U}$ we denote by $\Gamma_{|U}$ the restriction of Γ to U, that is, the set of interactions connecting only instances from U.

Example 3.2. For the Master/Slave example, the set $\mathcal{K}$ contains two connector types, namely InOut and Link. InOut is used to connect two instances of type Master, that is, the signature $sign(\mathsf{InOut})$ is Master Master. Link is used to connect an instance of type Master to an instance of type Slave, that is, the signature $sign(\mathsf{Link})$ is Master Slave. The system depicted in Figure 1 contains interactions of both types, namely five of type InOut of the form $\mathsf{InOut}(m_i, m_{i+1})$ and nine of type Link of the form $\mathsf{Link}(m_i, s_j)$.

Let $\mathcal{P}$ denote a finite set of state predicates (names). Every predicate P has a signature denoted as $sign(P)$, of the same form as defined earlier for connector types. Predicates are interpreted on tuples of component states and/or sets of component states compatible with their signature. For a predicate P with signature $sign(P) = (C_{i_k})_{k=1,n}(2^{C_{j_\ell}})_{\ell=1,m}$ its domain of interpretation denoted as $domain(P)$ is the Cartesian product $\prod_k^n \mathcal{Q}^{C_{i_k}} \times \prod_{\ell=1}^m 2^{\mathcal{Q}^{C_{j_\ell}}}$. For a predicate P, we denote by $[[P]]$ its interpretation on the domain, that is, a boolean function $[[P]] : domain(P) \to \mathbb{B}$. We tacitly assume True, False belong to $\mathcal{P}$ with the usual meaning.

Example 3.3. For the Master/Slave example, we consider two predicates, namely running and fail. The signature $sign(\mathsf{running})$ is Master, that means, this predicate is interpreted over the states of the component type Master i.e., $[[\mathsf{running}]] : \mathcal{Q}^{\mathsf{Master}} \to \mathbb{B}$ and shall define the "running" status depending on these states. Similarly, the signature $sign(\mathsf{fail})$ is Slave, that is, this predicate is used to determine the "fail" status depending on the states of the component type Slave.

3.2 Propositional Configuration Logic

The propositional configuration logic has the following syntax:

$$\begin{array}{rcl} \phi & ::= & P(u_1, \cdots u_{n_P}, U_1, \cdots U_{m_P}) \\ & | & K(u'_1, \cdots u'_{n_K}, U'_1, \cdots U'_{m_K}) \mid n\sigma\mathcal{K} \\ & | & \phi + \phi \mid \phi \vee \phi \mid \neg\phi \end{array}$$

where $P \in \mathcal{P}$ denotes a state predicate, $K \in \mathcal{K}$ denotes a connector type, $u_i, u'_i \in \mathcal{U}$ denote component instances belonging to the universe $\mathcal{U}$ of instances, $U_i, U'_i \subseteq \mathcal{U}$ denote sets of component instances. We tacitly restrict to formulas which are correctly typed, that is, where the (sets of) instances used as arguments for state predicates and connector types are matching the expected signature, formally, iff $sign(P) = (type(u_i))_{i=1,n_P}(2^{type(U_i)})_{i=1,m_P}$, $sign(K) = (type(u'_i))_{i=1,n_K}(2^{type(U'_i)})_{i=1,m_K}$.

We define boolean operators in the usual way, $\phi_1 \wedge \phi_2 \triangleq \neg(\neg\phi_1 \vee \neg\phi_2)$, $\phi_1 \Rightarrow \phi_2 \triangleq \neg\phi_1 \vee \phi_2$, $\phi_1 \equiv \phi_2 \triangleq (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$.

We denote by $instances(\phi)$ the set of instances occurring explicitly in a propositional configuration formula ϕ as parameters of state predicates and/or connector types. The semantics of propositional configuration logic is defined on *configurations* that is, triples of the form (U, γ, σ) where (i) $U \subseteq \mathcal{U}$ is a set of instances, (ii) $\gamma \subseteq \Gamma_{|U}$ is a set of interactions defined on U and (iii) $\sigma \in \boldsymbol{Q}^U$ is a state of U. For a propositional formula ϕ its semantics is restricted to configurations (U, γ, σ) where $instances(\phi) \subseteq U$, that is, where all support instances are effectively defined in the configuration.

$$\begin{array}{lll} (U,\gamma,\sigma) \models P(u_1, \cdots u_{n_P}, U_1, \cdots U_{m_P}) & & \\ \quad \text{iff } [\![P]\!](\sigma(u_1), \cdots \sigma(u_{n_P}), \sigma(U_1), \cdots \sigma(U_{m_P})) & & \\ (U,\gamma,\sigma) \models K(u'_1, \cdots u'_{n_K}, U'_1, \cdots U'_{m_K}) & & \\ \quad \text{iff } \gamma = \{K(u'_1, \cdots u'_{n_K}, U'_1, \cdots U'_{m_K})\} & & \\ (U,\gamma,\sigma) \models n\sigma\mathcal{K} & \text{iff} & \gamma = \emptyset \\ (U,\gamma,\sigma) \models \phi_1 + \phi_2 & \text{iff} & \exists\gamma_1\exists\gamma_2.\ \gamma = \gamma_1 \cup \gamma_2, \\ & & (U,\gamma_1,\sigma) \models \phi_1, (U,\gamma_2,\sigma) \models \phi_2 \\ (U,\gamma,\sigma) \models \phi_1 \vee \phi_2 & \text{iff} & (U,\gamma,\sigma) \models \phi_1 \text{ or } (U,\gamma,\sigma) \models \phi_2 \\ (U,\gamma,\sigma) \models \neg\phi & \text{iff} & (U,\gamma,\sigma) \not\models \phi \end{array}$$

Intuitively, predicate atoms $P(u_1, \cdots u_{n_P}, U_1, \cdots U_{m_P}))$ are evaluated on the state σ of corresponding instances $u_1, \cdots\ u_{n_P}, U_1, \cdots U_{m_P}$. By abuse of notation, we consider the state $\sigma(U_j)$ of a set of component instances U_j to be defined as $\sigma(U_j) \triangleq \{\sigma(u) \mid u \in U_j\}$, for all U_j. Connector atoms of the form $K(u'_1, \cdots u'_{n_K}, U'_1, \cdots U'_{m_K})$ hold iff the set of interactions γ contains precisely the interaction $K(u'_1, \cdots u'_{n_K}, U'_1, \cdots U'_{m_K})$. The atom $n\sigma\mathcal{K}$ holds if the set of interactions γ is empty. The formula $\phi_1 + \phi_2$ holds for configurations whose architecture (U, γ) is obtained by *coalescing* architectures (U, γ_1) and (U, γ_2) satisfied respectively by the formulas ϕ_1, ϕ_2. The meaning of the boolean connectives is the usual one.

Example 3.4. The formula $\mathsf{Link}(m_1, s_1) + \mathsf{Link}(m_1, s_2)$ denotes the architecture where the master m_1 can interact with two slaves s_1 and s_2. Thus in propositional CL we can specify all the Master/Slave architectures consisting of two masters m_1, m_2 and two slaves s_1, s_2 as the disjunction of the architectures where each slave has a unique master:

$$\begin{array}{l} (\mathsf{Link}(m_1, s_1) + \mathsf{Link}(m_1, s_2)) \vee (\mathsf{Link}(m_1, s_1) + \mathsf{Link}(m_2, s_2)) \vee \\ (\mathsf{Link}(m_1, s_2) + \mathsf{Link}(m_2, s_1)) \vee (\mathsf{Link}(m_2, s_1) + \mathsf{Link}(m_2, s_2)) \end{array}$$

The following lemma provides few elementary identities directly derived from the proposed CL semantics. In particular, they point out how atoms of the $K(...)$ and $P(...)$ can be composed as well as some natural distributivity laws involving logic operators. These are similar to identities proved for the propositional configuration logic defined in [17]. Nonetheless, the underlying models and semantics of the two logics are different.

Lemma 3.5. *The following identities hold*

(i) $K_1((u_i)_i, (U_j)_j) \wedge K_2((v_i)_i, (V_j)_j) \equiv \text{False}$ *whenever* $K_1 \neq K_2$ *or* $((u_i)_i, (U_j)_j) \neq ((v_i)_i, (V_j)_j)$
(ii) $P_1(...) \wedge \phi_1 + P_2(...) \wedge \phi_2 \equiv P_1(...) \wedge P_2(...) \wedge (\phi_1 + \phi_2)$
(iii) $\phi + (\phi_1 \vee \phi_2) \equiv (\phi + \phi_1) \vee (\phi + \phi_2)$

Proof. (i) an atom of the form $K(...)$ holds iff γ contains exactly the corresponding interaction, therefore, a conjunction holds if the two atoms are identical and fails otherwise (ii) as predicate atoms depend only on σ, they are independent of coalescing and can be factorized (iii) follows the distributivity of conjunction (in the semantics of +) over the disjunction □

Example 3.6. As + is distributive with respect to $\vee$, the formula

$$(\mathsf{Link}(m_1, s_1) \vee \mathsf{Link}(m_2, s_1)) + (\mathsf{Link}(m_1, s_2) \vee \mathsf{Link}(m_2, s_2))$$

is equivalent to the formula given in Example 3.4. The formula simply models the architectures obtained by coalescing architectures where each slave is connected to either of the two masters.

We define the closure ~ operator by taking $\sim\phi \triangleq \phi + \text{True}$. Intuitively, $\sim\phi$ denotes architectures obtained by extending architectures modeled by ϕ with any number of interactions. For example, the formula $\sim\mathsf{Link}(m_1, s_2)$ specifies architectures containing at least the interaction Link between m_1 and s_2. The following identities established for propositional configuration logic of [17] hold also for propositional CL.

Lemma 3.7. *The following identities hold*

(i) $\sim\sim\phi \equiv \sim\phi$ *(iii)* $\sim(\phi_1 \vee \phi_2) \equiv \sim\phi_1 \vee \sim\phi_2$
(ii) $\phi \Rightarrow \sim\phi$ *(iv)* $\sim\phi_1 + \sim\phi_2 \equiv \sim(\phi_1 + \phi_2) \equiv \sim\phi_1 \wedge \sim\phi_2$

Proof. (i) immediate as the idempotence True + True ≡ True holds (ii) using the definition of +, knowing that True holds for any architecture (iii) direct consequence of point (iii) in Lemma 3.5 (iv) using associativity and commutativity of + and the idempotence of True for the first equality; using associativity of conjunction (as used in the definition of +) and the idempotence of True for the second equality. □

Note that the closure operator proves to be particularly useful for specifications which are the conjunction of characteristic properties each one expressed by a formula. For the Master/Slave example of Fig. 1, the formula $\sum_{i=1}^{5} \mathsf{OutIn}(m_i, m_{i+1})$ strictly characterizes the connectivity between masters in the ring. However, if it is involved in a conjunction with formulas expressing another connectivity property, e.g., that masters are connected to slaves, the resulting formula will be false. To characterize exactly the Master/Slave architecture the coalescing of this formula should be taken with the formulas specifying the connectors $\mathsf{Link}(m_i, s_j)$. An alternative solution leading to conjunctive specifications, would be

to specify the ring by the formula $\sim\sum_{i=1}^{5}\mathsf{OutIn}(m_i, m_{i+1})$ equivalent to $\bigwedge_{i=1}^{5}\sim\mathsf{OutIn}(m_i, m_{i+1})$ which characterizes configurations containing the connectors of the ring but does not exclude other configurations. This formula can be used in a conjunction with other formulas involving closure operators.

Besides its usefulness for expressing conjunctively configuration properties, the subset of CL involving only closure formulas proves to be tractable for verification using SMT techniques. This *Closure Configuration Logic* (CL) fragment introduced later in section 4 represents a trade-off between expressive power and complexity of verification. In this logic the coalescing operator can be replaced by conjunction thanks to Lemma 3.7 (iv).

3.3 Second-Order Configuration Logic

CL extends its propositional version with component variables and component set variables. The second-order configuration logic has the following syntax:

$$\begin{array}{rcl}\Phi & ::= & P(z_1,\cdots z_{n_P}, Z_1,\cdots Z_{m_P}) \\ & | & K(z'_1,\cdots z'_{n_K}, Z'_1,\cdots Z'_{m_K}) \mid no\mathcal{K} \\ & | & z''_1 = z''_2 \mid z''_3 \in Z''_3 \\ & | & \Phi + \Phi \mid \Phi \vee \Phi \mid \neg\Phi \\ & | & \exists x : C.\ \Phi(x) \mid \exists X : 2^C\ .\Phi(X)\end{array}$$

where $P \in \mathcal{P}$ denotes a state predicate, $K \in \mathcal{K}$ denotes a connector type, $C \in \mathcal{C}$ denotes a component type. Lower case z_i, z'_i, z''_i denote component instances (that is, first-order constants) and/or first order variables x ranging over instances. Similarly, upper case Z_i, Z'_i, Z''_i denote sets of components instances (that is, second-order constants) and/or second-order variables X ranging over sets of instances. All variables x, X occur only in the scope of existential quantifiers and are *typed*. We denote by $type(x)$, $type(X)$ their type, that is, a component type from $\mathcal{C}$. As previously, we tacitly restrict to formulas which are correctly typed, that is, where variables and/or constant (sets of) instances used as arguments for state predicates and connector types are matching to the expected signature.

All the additional operators defined for the propositional case are naturally lifted for the second-order case. Moreover, we define first-order and second-order universal quantifiers as usual $\forall x : C.\ \Phi(x) \triangleq \neg\exists x : C.\ \neg\Phi(x)$, $\forall X : 2^C.\ \Phi(X) \triangleq \neg\exists x : 2^C.\ \neg\Phi(X)$. Furthermore, for every component type C, we define inclusion $\subseteq_C$ and equality $=_C$ on sets of instances by taking $Z_1 \subseteq_C Z_2 \triangleq \forall x : C.(x \in Z_1 \Rightarrow x \in Z_2)$, $Z_1 =_C Z_2 \triangleq Z_1 \subseteq_C Z_2 \wedge Z_2 \subseteq_C Z_1$.

As for the propositional case, we denote by *instances*(Φ) the set of instances occurring explicitly in Φ, as parameters of state predicates and/or connector types. We denote by *freevars*(Φ) the set of free variables occurring in Φ.

Let $\mathcal{X}$ be a set of typed variables, partitioned into respectively first-order $\mathcal{X}^{(1)}$ and second-order $\mathcal{X}^{(2)}$ variables. For a given set of instances U, the set of valid interpretations of $\mathcal{X}$ on U is denoted as $U^{\mathcal{X}}$ and defined as the set of functions

$$\left\{\iota : \mathcal{X} \to U \cup 2^U \;\middle|\; \begin{array}{l}\forall x \in \mathcal{X}^{(1)}.\ \iota(x) \in \mathcal{U}^{type(x)} \cap U \\ \forall X \in \mathcal{X}^{(2)}.\ \iota(X) \subseteq \mathcal{U}^{type(X)} \cap U\end{array}\right\}$$

Interpretation ι is naturally extended to constants by letting $\iota(u) \triangleq u$ for all $u \in \mathcal{U}$, $\iota(U) \triangleq U$, for all $U \subseteq \mathcal{U}$.

The semantics of a second-order formula Φ is defined on configurations (U, γ, σ) with respect to an interpretation ι of *freevars*(Φ) on U. As for the propositional case, we tacitly restrict semantics to configurations (U, γ, σ) where *instances*$(\Phi) \subseteq U$, that is, where all support instances are effectively defined.

$$\begin{array}{lll}(U,\gamma,\sigma) \models_\iota P(z_1,\cdots z_{n_P}, Z_1,\cdots Z_{m_P}) & & \\ \quad \text{iff } [\![P]\!](\sigma(\iota(z_1)),\cdots\sigma(\iota(z_{n_P})),\sigma(\iota(Z_1)),\cdots\sigma(\iota(Z_{m_P}))) & & \\ (U,\gamma,\sigma) \models_\iota K(z'_1,\cdots z'_{n_K}, Z'_1,\cdots Z'_{m_K}) & & \\ \quad \text{iff } \gamma = \{K(\iota(z'_1),\cdots\iota(z'_{n_K}),\iota(Z'_1),\cdots\iota(Z'_{m_K})\} & & \\ (U,\gamma,\sigma) \models_\iota no\mathcal{K} & \text{iff} & \gamma = \emptyset \\ (U,\gamma,\sigma) \models_\iota z''_1 = z''_2 & \text{iff} & \iota(z''_1) = \iota(z''_2) \\ (U,\gamma,\sigma) \models_\iota z''_3 \in Z''_3 & \text{iff} & \iota(z''_3) \in \iota(Z''_3) \\ (U,\gamma,\sigma) \models_\iota \Phi_1 + \Phi_2 & \text{iff} & \exists\gamma_1\exists\gamma_2.\ \gamma = \gamma_1 \cup \gamma_2, \\ & & (U,\gamma_1,\sigma) \models_\iota \Phi_1,\ (U,\gamma_2,\sigma) \models_\iota \Phi_2 \\ (U,\gamma,\sigma) \models_\iota \Phi_1 \vee \Phi_2 & \text{iff} & (U,\gamma,\sigma) \models_\iota \Phi_1 \text{ or } (U,\gamma,\sigma) \models_\iota \Phi_2 \\ (U,\gamma,\sigma) \models_\iota \neg\Phi & \text{iff} & (U,\gamma,\sigma) \not\models_\iota \Phi \\ (U,\gamma,\sigma) \models_\iota \exists x : C.\ \Phi & \text{iff} & (U,\gamma,\sigma) \models_{\iota\cup\{x\mapsto u_0\}} \Phi \\ & & \text{for some } u_0 \in \mathcal{U}^C \cap U \\ (U,\gamma,\sigma) \models_\iota \exists X : 2^C.\ \Phi & \text{iff} & (U,\gamma,\sigma) \models_{\iota\cup\{X\mapsto U_0\}} \Phi \\ & & \text{for some } U_0 \subseteq \mathcal{U}^C \cap U\end{array}$$

The evaluation of predicate and connector atoms is the same as in propositional CL, after taking into account the interpretation of formula variables defined by ι. The meaning of equality and membership test is the usual one. Similarly, the meaning of + is lifted from propositional CL, and the meaning of all boolean connectives is the usual one. Finally, quantifiers are interpreted according to the domain U of instances, by choosing an appropriate instance (respectively set of instances) of corresponding type.

Example 3.8. Using CL we can express configuration properties independent of the identity of components. For example, for any slave y that has not failed, there exists a master x such that x and y can interact via connector Link:

$$\forall y : \mathsf{Slave}.\ \neg\mathsf{fail}(y) \Rightarrow \exists x : \mathsf{Master}.\ \sim\mathsf{Link}(x, y)$$

Moreover, the second-order extension allows us to express complex structural properties such as, for example, all instances of type Master are interconnected into a single ring using the OutIn connectors. This property is obtained actually as the conjunction of two simpler properties, namely:

(i) every instance of type Master interacts with precisely one other instance through the OutIn connector type:

$$\forall x : \mathsf{Master}.\ \exists x' : \mathsf{Master}.\ \sim\mathsf{OutIn}(x, x') \wedge \forall x'' : \mathsf{Master}.\ (x'' \neq x' \Rightarrow \neg\sim\mathsf{OutIn}(x, x''))$$

(ii) all instances of type Master are connected, that is, for any non-empty strict subset X of Master instances there exists an interaction between an instance within X and an instance outside X:

$$\forall X : 2^{\mathsf{Master}}.\ (\exists x, x' : \mathsf{Master}.\ x \in X \wedge x' \notin X) \Rightarrow (\exists x, x' : \mathsf{Master}.\ x \in X \wedge x' \notin X \wedge \sim\mathsf{OutIn}(x, x'))$$

4 VERIFICATION OF CL PROPERTIES

In this section we provide an effective method for checking the validity of a CL formula Φ on finite configurations (U, γ, σ). The method is based on the equivalent encoding of the validity question "$(U, \gamma, \sigma) \models_\emptyset \Phi$" as the satisfiability question of an *instantiated*

formula, expressed in a decidable target logic, and solvable by existing off-the-shelf SMT solvers. The target logic includes at least the theory of finite length bitvectors, for encoding architectural properties evaluated on (U, γ), as well as any additional theories needed to evaluate state predicates on σ.

The method is actually restricted to the *Closure Configuration Logic* (CL) fragment, namely, where (i) all connector atoms $K(\cdots)$ occur under the immediate scope of the ~ operator and (ii) only logical connectives are used. Under this restriction, the *instantiated formula* is obtained by a structural translation of Φ augmented with some additional clauses for encoding architecture and state information about (U, γ, σ). Without this restriction, a more general encoding following the structure of the validity question "$(U, \gamma, \sigma) \models_\emptyset \Phi$" would be needed.

In this section, we first present the principle of encoding instances and sets of instances using finite length bitvectors. Then, we introduce the encoding of connector types and state predicates as (interpreted) predicates and functions on bitvectors. Finally, we present the structural translation of CL formulas as formulas of finite-length bitvector theory.

4.1 Encoding using finite-length bitvectors

Let (U, γ, σ) be a fixed configuration defined on a finite set of component instances U. Let denote by N the cardinality of U.

Let choose $\ll_U$ a total order on U and denote by $idx_{\ll_U}(u)$ the *index* of u in the ordered sequence defined by $\ll_U$ on U, formally

$$idx_{\ll_U}(u) \triangleq |\{u' \in U \mid u' \ll_U u,\ u' \neq u\}|$$

We associate constant bitvectors of length N to component instances, sets of component instances, component types as follows:

$$\begin{array}{rcl} \textit{bvconst-}u & \triangleq & \underline{nat2bv}[N](2^{idx_{\ll_U}(u)}) \\ \textit{bvconst-}\emptyset & \triangleq & \underline{nat2bv}[N](0) \\ \textit{bvconst-}\{u_1, \cdots u_k\} & \triangleq & \underline{bvor}(\textit{bvconst-}u_1, \cdots \textit{bvconst-}u_k) \\ \textit{bvconst-}C & \triangleq & \textit{bvconst-}\{u \in U \mid \textit{type}(u) = C\} \end{array}$$

In the above, the function $\underline{nat2bv}[N]$ takes a non-negative integer n smaller than 2^N and returns a bitvector of length N corresponding to the binary encoding of n using N bits.

Example 4.1. Consider the system configuration (U, γ, σ) depicted in Figure 1. The cardinality of U is 14, that is, target formulas will operate with bitvectors of size 14. Let consider the total order $\ll_U$ defined by the sequence $m_1, m_2, ...m_5, s_1, s_2, ...s_9$. Instances and sets of instances are therefore encoded as follows, for example:

$$\begin{array}{rcl} \textit{bvconst-}m_2 & \triangleq & \langle 01000 \quad 000000000 \rangle \\ \textit{bvconst-}\{s_1, s_2, s_7\} & \triangleq & \langle 00000 \quad 110000100 \rangle \\ \textit{bvconst-}\emptyset & \triangleq & \langle 00000 \quad 000000000 \rangle \\ \textit{bvconst-}\mathsf{Master} & \triangleq & \langle 11111 \quad 000000000 \rangle \end{array}$$

We define few more helper predicates, respectively, *bvpred-one* to check if a bitvector contains precisely a single 1 bit (that is, it represents precisely a unique instance), *bvpred-subset* to check inclusion between two sets and *bvpred-elem* to check membership of an element to a set, represented as bitvectors:

$$\begin{array}{rcl} \textit{bvpred-one}(x) & \triangleq & \vee_{i=0}^{N-1}(x = \underline{nat2bv}[N](2^i)) \\ \textit{bvpred-subset}(X, Y) & \triangleq & \underline{bvor}(X, Y) = Y \\ \textit{bvpred-elem}(x, X) & \triangleq & \textit{bvpred-one}(x) \wedge \textit{bvpred-subset}(x, X) \end{array}$$

4.2 Handling connector types

For every connector type K, we define the predicate *bvpred-K* to encode the set of interactions of type K defined in γ. Assuming the signature of K is $sign(K) = (C_{i_k})_{k=1,n_K}(2^{C_{j_\ell}})_{\ell=1,m_K}$, the predicate *bvpred-K* is defined on tuples of $n_K + m_K$ bitvectors by taking

$$\begin{array}{l} \textit{bvpred-}K(x_1, ..., x_{n_K}, X_1, ..., X_{m_K}) \triangleq \\ \bigvee_{K(u_1, \cdots u_{n_K}, U_1, \cdots U_{m_K}) \in \gamma} \left(\begin{array}{c} \wedge_{i=1}^{n_K} x_i = \textit{bvconst-}u_i \\ \wedge \\ \wedge_{j=1}^{m_k} X_j = \textit{bvconst-}U_j \end{array} \right) \end{array}$$

Example 4.2. In the configuration depicted in Figure 1, the set of interactions γ contains 5 interactions of type InOut of the form $\mathsf{InOut}(m_i, m_{i+1})$. These are encoded by the bitvector predicate $\textit{bvpred-}\mathsf{InOut}(x_1, x_2)$ defined as follows:

$$\textit{bvpred-}\mathsf{InOut}(x_1, x_2) \triangleq \vee_{i=1}^{5}(x_1 = \textit{bvconst-}m_i \wedge x_2 = \textit{bvconst-}m_{i+1})$$

The next lemma characterizes the encoding of connector types.

Lemma 4.3. *For any connector type K*

$$\begin{array}{l} (U, \gamma, \sigma) \models {\sim}K(u_1, \cdots u_{n_K}, U_1, \cdots U_{m_K}) \textit{ iff} \\ \quad \textit{bvpred-}K(\textit{bvconst-}u_1, \cdots \textit{bvconst-}u_{n_K}, \\ \qquad \textit{bvconst-}U_1, \cdots \textit{bvconst-}U_{m_K}) \equiv \mathsf{True} \end{array}$$

Proof. By definition of *bvpred-K* and semantics of ~$K(...)$. □

4.3 Handling state predicates

As for connector types, for every state predicate P, we define the bitvector predicate *bvpred-P* to encode its interpretation $[\![P]\!]$ restricted to the state configuration σ. Assuming that the signature of P is $sign(P) = (C_{i_k})_{k=1,n_P}(2^{C_{j_\ell}})_{\ell=1,m_P}$, the predicate *bvpred-P* is defined on tuples of $n_P + m_P$ bitvectors.

For practical reasons, we restrict to state predicates defined by multi-sorted expressions constructed from state variables and constants, functions and predicates defined for some data sorts T_i's. We consider that, for every component type C, the set of component states Q^C is defined as a Cartesian product $T_{i_1} \times T_{i_2} \times ...$ corresponding to valuations of a tuple $\langle s_1^C, s_2^C, ...\rangle$ of state variables with sorts respectively $T_{i_1}, T_{i_2}, \cdots$. The syntax of predicates terms and expressions is as follows

$$\begin{array}{rcl} t & ::= & c \mid x.s \mid f(t_1, t_2, \cdots) \\ e & ::= & t_1 = t_2 \mid p(t_1, t_2, \cdots) \mid e_1 \vee e_2 \mid \neg e \end{array}$$

In the above, c denotes a predefined constant, f a predefined function and p a predefined predicate of the considered data sorts. x is a typed variable denoting component instances and s a state variable defined for the type of x. We tacitly assume that terms and expressions are correctly typed with respect to data sorts and component types. A state predicate P described by an expression $e_P(x_1, \cdots x_m)$ has signature $sign(P) = (type(x_i))_{i=1,m}$ according to the types of variables used in e_P and the interpretation $[\![P]\!]$ defined through the

Table 1: Translation rules for state expressions (left) and CL formulas (right)

$$
\begin{array}{rcl}
tr(c) & \triangleq & c \\
tr(x.s) & \triangleq & \textit{bvfun-s}(x) \\
tr(f(t_1, t_2, \cdots)) & \triangleq & f(tr(t_1), tr(t_2), \cdots) \\
tr(t_1 = t_2) & \triangleq & tr(t_1) = tr(t_2) \\
tr(p(t_1, t_2, \cdots)) & \triangleq & p(tr(t_1), tr(t_2), \cdots) \\
tr(e_1 \vee e_2) & \triangleq & tr(e_1) \vee tr(e_2) \\
tr(\neg e) & \triangleq & \neg tr(e)
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
tr(u) & \triangleq & \textit{bvconst-u} & tr(U) & \triangleq & \textit{bvconst-U} \\
tr(x) & \triangleq & x & tr(X) & \triangleq & X \\
tr(z_1 = z_2) & \triangleq & tr(z_1) = tr(z_2) & tr(z \in Z) & \triangleq & \textit{bvpred-subset}(tr(z), tr(Z)) \\
tr(\Phi_1 \vee \Phi_2) & \triangleq & tr(\Phi_1) \vee tr(\Phi_2) & tr(\neg\Phi) & \triangleq & \neg tr(\Phi) \\
tr(\exists x : C.\ \Phi) & \triangleq & \exists x : \underline{bv}[N].\ \textit{bvpred-elem}(x, \textit{bvconst-C}) \wedge tr(\Phi) & & & \\
tr(\exists X : C.\ \Phi) & \triangleq & \exists X : \underline{bv}[N].\ \textit{bvpred-subset}(X, \textit{bvconst-C}) \wedge tr(\Phi) & & & \\
tr(\sim K(z_1, \cdots z_n, Z_1, \cdots Z_m)) & \triangleq & \textit{bvpred-K}(tr(z_1), \cdots tr(z_n), tr(Z_1), \cdots tr(Z_m)) & & & \\
tr(P(z_1, \cdots z_n, Z_1, \cdots Z_m)) & \triangleq & \textit{bvpred-P}(tr(z_1), \cdots tr(z_n), tr(Z_1), \cdots tr(Z_m)) & & &
\end{array}
$$

standard evaluation of e_P. The encoding of predicates P relies on the encoding of state variables as functions on bitvectors. Let s be the jth state variable of component type C and let T be the sort of s. We define the bitvector function $\textit{bvfun-s} : \underline{bv}[N] \to T$ to obtain the value of s for instances of type C in the state configuration σ by taking:

$$
\textit{bvfun-s}(x) \triangleq \begin{cases} c_j & \text{if } x = \textit{bvconst-u} \text{ for some } u \in U \text{ such that} \\ & \quad \textit{type}(u) = C \text{ and } \sigma(u) = (c_1, c_2, \cdots c_j, \cdots) \\ c_0 & \text{otherwise, for some fixed } c_0 \in T \end{cases}
$$

Based on the encoding of state variables, we define a translation of state terms and state expressions into bitvector expressions. The translation preserves their structure and rewrites any component variable access $x.s$ with the function call respectively $\textit{bvfun-s}(x)$. Formally, the translation is defined in Table 1 (left). Finally, for any predicate P described by a state expression $e_P(x_1, \cdots x_m)$ we define the bitvector predicate *bvpred-P* by taking

$$
\textit{bvpred-P}(x_1, \cdots x_m) \triangleq tr(e_P)
$$

By construction, $\textit{bvpred-P}(x_1, \cdots x_m)$ evaluates into the value of $e_P(x_1, \cdots x_m)$ depending on the assignment of x_i's to component instances from U and on their state configuration as defined by σ. The following lemma characterizes formally our encoding.

Lemma 4.4. *For any state predicate P*

$$
\begin{array}{c}
(U, \gamma, \sigma) \models P(u_1, \cdots u_m) \textit{ iff} \\
\textit{bvpred-P}(\textit{bvconst-u}_1, \cdots \textit{bvconst-u}_m) \equiv \mathsf{True}
\end{array}
$$

Proof. By structural induction on the expression e_P. □

4.4 Handling CL formulas

The translation of a configuration formula Φ is defined in Table 1 (right). Intuitively, the translation fully preserves the logical structure of Φ and rewrites (1) constants u, U into bitvector constants *bvconst-u*, *bvconst-U*, (2) variables x, X of some component type C into (first-order) bitvector variables x, X plus additional bitvector constraints (to restrict type to instances of C and cardinality for first-order), (3) formulas $\sim K(\cdots)$ into bitvector predicates $\textit{bvpred-K}(\cdots)$ and (4) state predicates $P(\ldots)$ into bitvector predicates $\textit{bvpred-P}(\ldots)$.

Example 4.5. Consider the configuration formula Φ_0 introduced in Example 3.8:

$$
\forall x : \mathsf{Master}.\ \exists x' : \mathsf{Master}.\ \sim \mathsf{OutIn}(x, x')
$$

The translation $tr(\Phi_0)$ according to configuration (U, γ, σ) depicted in Figure 1 is defined as

$$
\begin{array}{l}
\forall x : \underline{bv}[14].\ (\textit{bvpred-elem}(x, \textit{bvconst-}\mathsf{Master})) \Rightarrow \\
\quad (\exists x' : \underline{bv}[14].\ \textit{bvpred-elem}(x', \textit{bvconst-}\mathsf{Master}) \wedge \\
\qquad \textit{bvpred-}\mathsf{InOut}(x, x'))
\end{array}
$$

where bitvector constants *bvconst-*Master, and bitvector predicates *bvpred-elem*, *bvpred-subset* and *bvpred-*InOut were defined earlier.

The next proposition expresses the correctness of the encoding.

Proposition 4.6. *If Φ is a CL formula then*

$$
(U, \gamma, \sigma) \models_\emptyset \Phi \textit{ iff } tr(\Phi) \textit{ is sat.}
$$

Proof. By structural induction on the formula Φ. □

Finally, let us remark that the complexity of satisfiability problems for fixed-size bitvector logics has been thoroughly investigated [13, 14]. In particular, if unary encoding (that is, bit blasting) is used, the complexity is respectively NP-complete for the quantifier-free fragment and PSPACE-complete for the full bit-vector logic. These known results can be used to establish upper bounds for the complexity of the verification problem of CL formulas on finite configurations. Nonetheless, determining precisely this complexity needs further investigation and is not considered in this work.

5 TEMPORAL CONFIGURATION LOGIC TCL

The temporal configuration logic has the following syntax

$$
\Psi ::= \Phi \mid \mathbf{flip} \mid \mathbf{N}\Psi \mid \Psi\mathbf{U}\Psi \mid \Psi \vee \Psi \mid \neg\Psi
$$

where Φ is a second-order configuration formula, **flip** is the reconfiguration status operator, **N** is the next operator, **U** is the until operator.

The semantics of TCL formulas is defined on infinite sequences $w = w_0 w_1 \ldots$ of configurations, that is, where every w_i is a configuration $(U_i, \gamma_i, \sigma_i)$ as defined earlier. For an infinite sequence w, we denote by $w^{(i)}$ the infinite subsequence of w starting at index i.

$$
\begin{array}{lll}
w \models \Phi & \text{iff} & w_0 \models_\emptyset \Phi \\
w \models \mathbf{flip} & \text{iff} & U_0 \neq U_1 \vee \gamma_0 \neq \gamma_1 \text{ where } w = (U_i, \gamma_i, \sigma_i)_{i \geq 0} \\
w \models \mathbf{N}\Psi & \text{iff} & w^{(1)} \models \Psi \\
w \models \Psi_1 \mathbf{U} \Psi_2 & \text{iff} & \exists i \geq 0.\ w^{(i)} \models \Psi_2 \wedge \forall j \in [0, i).\ w^{(j)} \models \Psi_1 \\
w \models \Psi_1 \vee \Psi_2 & \text{iff} & w \models \Psi_1 \text{ or } w \models \Psi_2 \\
w \models \neg\Psi & \text{iff} & w \not\models \Psi
\end{array}
$$

Notice that the formula **flip** is satisfied for sequences with an initial state after which a reconfiguration (change of configuration) occurs. We consider the additional boolean operators defined in the standard way, that is, $\Psi_1 \wedge \Psi_2 \triangleq \neg(\neg\Psi_1 \vee \neg\Psi_2)$, $\Psi_1 \Rightarrow \Psi_2 \triangleq \neg\Psi_1 \vee \Psi_2$. We consider moreover the usual eventually operator $\Diamond\Psi \triangleq \mathrm{True}\mathbf{U}\Psi$, and always operator $\Box\Psi \triangleq \neg\Diamond\neg\Psi$. We define furthermore an additional *next reconfiguration* $\mathbf{N}_R$ operator by taking $\mathbf{N}_R\Psi \triangleq (\neg\mathbf{flip})\mathbf{U}(\mathbf{flip} \wedge \mathbf{N}\Psi)$. Notice that this formula characterizes all the sequences where Ψ holds right after the first reconfiguration.

Example 5.1. We show how TCL allows the expression of non-trivial properties of DR-systems. For example, immediately after reconfiguration, all instances of type Master are running

$$\Box\ (\mathbf{flip} \Rightarrow \mathbf{N}(\forall x : \mathrm{Master}.\ \mathrm{running}(x)))$$

or, whenever an instance of type Master is not running, reconfiguration is eventually executed

$$\Box\ (\exists x : \mathrm{Master}.\ \neg\mathrm{running}(x)) \Rightarrow \Diamond\mathbf{flip}$$

Moreover, we can specify the rule of load balancing as follows:

$$\begin{aligned}\Box\ \big(&\exists x_1, x_2 : \mathrm{Master}.\ \exists Y_1, Y_2 : 2^{\mathrm{Slave}}.\ \sim\mathrm{OutIn}(x_1, x_2) \wedge \\ &(\forall y_1 : \mathrm{Slave}.\ y_1 \in Y_1 \Leftrightarrow \sim\mathrm{Link}(x_1, y_1)) \wedge \\ &(\forall y_2 : \mathrm{Slave}.\ y_2 \in Y_2 \Leftrightarrow \sim\mathrm{Link}(x_2, y_2)) \wedge \\ &|Y_1| \geq |Y_2| + 2\big) \Rightarrow \mathbf{flip}\end{aligned}$$

that is, whenever exist two masters x_1, x_2, such that x_2 is the successor of x_1 in the ring, the sets of connected slaves Y_1, Y_2 respectively satisfy $|Y_1| \geq |Y_2| + 2$, the system must reconfigure at its next step, that is, **flip** hold.

We note that by considering CL formulas (Φ) and **flip** in the TCL as atomic propositions, we obtain a standard linear-time temporal logic (LTL) formula [20]. Runtime Verification (RV) [3, 7, 21] is a lightweight formal approach that consists in checking that a single run of a system complies with a given specification (typically described in LTL). Existing RV approaches provide tools such as LTL3Tools [3], and LamaConv [10] capable of automatically synthesizing monitors as finite-state machines. To check a TCL formula on a sequence of configurations, we first evaluate the validity of each CL formula for each configuration as described in Section 4, and assign the corresponding truth value to its associated atomic proposition (for that particular configuration). Once CL formulas are evaluated, we are able to check the TCL formula using standard RV tools and approaches, as it has been rewritten to LTL. For the atomic proposition **flip**, we compute its value depending on the change of configuration as specified by its semantics.

6 EXPERIMENTS

We implemented our approach in the tool drbip-verify for DR-BIP [2, 6] traces. We check several properties on two dynamic reconfigurable systems: the Master/Slave system described in this paper, and the Platoon system described in [6].

6.1 Tool Integration

The drbip-verify tool integrates a set of modules as shown in Figure 2.

drbip-verify reads a dr-bip trace containing relevant information on the execution of a dynamic reconfigurable system. In addition it reads a specification file that contains temporal formulas defined over a set of atomic propositions, and for each such atomic proposition an associated CL formula. The temporal formulas are passed to LamaConv [10] to generate FSM monitors. A temporal formula can be specified using any of the LamaConv input logics such as LTL, PLTL, regular expressions, and SALT. The CL formulas are passed to the trace processor which gathers information from the trace based on the referenced components and connectors. For our experiments we evaluate properties whenever the system reconfigures as the configuration changes. This allows us to sample uniformly traces of 100 reconfigurations for the experiments.

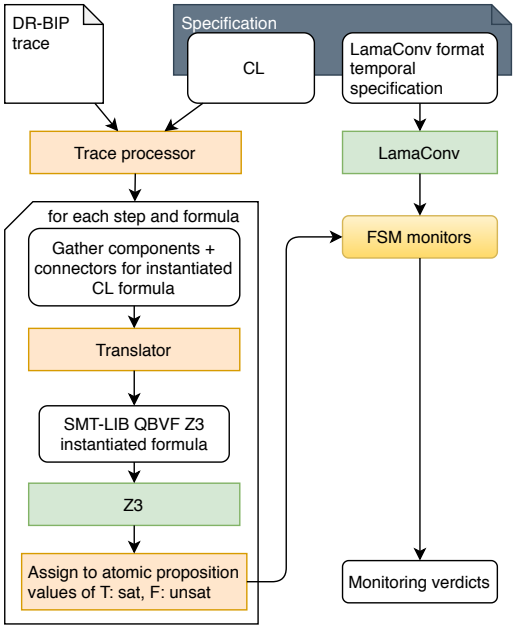

Figure 2: The main components of drbip-verify.

Whenever a reconfiguration occurs, the trace processor generates for each CL formula found in the specification the corresponding instantiated formula using the connector instances and the component instances present during that moment of execution as described in Section 4. The instantiated formula is then passed to Z3 to solve it. Once solved, the associated LTL atomic proposition with the CL formula is set to true iff the Z3 instance was satisfiable. Once all instantiated formulas are solved, the monitors receive the atomic propositions associated with them to update their evaluation of the temporal properties.

dr-bip utilizes the concept of architectural *motifs* gathering sets of components subject to the same coordination rules. For instance in the Master/Slave system a master with its slaves constitute a motif instance. Using motifs we can restrict the universe of each property to the relevant component and connector instances.

6.2 Experimental Setup

For our experiments, we consider two reconfigurable systems: the Master/Slave system, and the Platoon system. The properties considered are shown in Table 2.

Table 2: Properties considered.

Prop	CL Formula
Φ_{link}	$\forall s : \text{Slave}.\ (s.downtime = 0) \Rightarrow$ $(\exists m : \text{Master}.\ \sim\text{Link}(m, s))$
Φ_{inout}	$\forall m_0 : \text{Master}.\ \exists m_1 : \text{Master}.\ \sim\text{OutIn}(m_0, m_1) \wedge$ $(\forall m_2 : \text{Master}.\ m_2 \neq m_1 \Rightarrow \neg\sim\text{OutIn}(m_0, m_2))$
Φ_{ring}	$\forall X : 2^{\text{Master}}:$ $(\exists x, x' : \text{Master}.\ x \in X \wedge x' \notin X) \Rightarrow$ $(\exists x, x' : \text{Master}.\ x \in X \wedge x' \notin X \wedge \sim\text{OutIn}(x, x'))$
Φ_{unbal}	$\exists x_1, x_2 : \text{Master}.\ \exists Y_1, Y_2 : 2^{\text{Slave}}.$ $(\forall y_1 : \text{Slave}.\ y_1 \in Y_1 \Leftrightarrow \sim\text{Link}(x_1, y_1)) \wedge$ $(\forall y_2 : \text{Slave}.\ y_2 \in Y_2 \Leftrightarrow \sim\text{Link}(x_2, y_2)) \wedge$ $\sim\text{OutIn}(x_1, x_2) \wedge \lvert Y_1\rvert \geq \lvert Y_2\rvert + 2$
Φ_{split}	$\forall c_0 : \text{Car}.\ \forall c_1 : \text{Car}.\ \sim\text{Split}(c_0, c_1) \Rightarrow$ $(\forall c_2 : \text{Car}.\ c_2 \neq c_1 \Rightarrow \neg\sim\text{Split}(c_0, c_2))$
Φ_{speed}	$\forall c : \text{Car}.\ \forall X_0 : 2^{\text{Car}}.\ \sim\text{Speed}(X_0, c) \Rightarrow$ $(\forall X_1 : 2^{\text{Car}}.\ X_0 \neq X_1 \Rightarrow \neg\sim\text{Speed}(X_1, c))$
Φ_{nohalt}	$\forall c : \text{Car}.\ c.\text{velocity} > 0$
Φ_{safe}	$\exists c_0 : \text{Car}.\ \exists X_0 : 2^{\text{Car}}.\ \sim\text{Speed}(X_0, c_0) \wedge$ $(\forall c_1 : \text{Car}.\ c_1 \in X_0 \Rightarrow c_1.\text{pos} < c_0.\text{pos})$

Master/Slave System. For the Master/Slave system we consider two parameters: the number of masters and slaves per master. The properties we check are taken from Examples 3.8 and 5.1,

Platoon System. The Platoon system introduced in [6] describes cars moving on a single lane-road, grouped together in platoons. Each platoon has a leader that regulates the speed of the cars. Cars can split to form new platoons. Platoons also merge when close to each other. For the Platoon system we consider a single parameter: the number of cars on the road. We consider the following four properties: 1) a follower car can only split from a single leader car (Φ_{split}); 2) a leader car only regulates the speed of a single platoon (Φ_{speed}); 3) cars are always moving (Φ_{nohalt}); and 4) each platoon consists of a leader car and several followers cars located behind the leader (Φ_{safe}).

6.3 Results

For each system, we execute 10 runs for different values of their parameters to obtain configurations of varying number of connector and component instances.

We stop the execution after 100 reconfigurations, and record the trace. Each trace is then checked using DRBIP-VERIFY. We record the time spent by Z3 to solve the instantiated CL formulas. Table 3 presents the average number of connector instances, component instances, and the average time needed to solve a single instantiated CL formula. The parameters are the following: number of masters followed by number of slaves per master for the Master/Slave system, and number of cars for the Platoon system. When the solving time of a given CL formula does not depend on a parameter, we used its maximum value to simplify presentation. For some formulas, we notice a lower number of components than the sum of all components of the system. This is because they apply to motif instances and therefore benefit from a smaller universe size, even as the system grows (e.g., for Φ_{link} applies to all components of a single master and is evaluated multiple times for each master).

Table 3: Average time spent by Z3 (in ms) to solve instantiated CL formulas in traces of size 100. Cell contains: mean ± 95% confidence interval error.

Prop	Params		$\lvert\Gamma\rvert_{inst}$	$\lvert\mathcal{U}\rvert_{inst}$	t_{inst}	
Φ_{link}	4	50	97.4	49.2	31.80 ±	1.00
		100	190.52	95.76	64.81 ±	2.02
		200	399.99	200.49	195.11 ±	6.16
Φ_{inout}	2	200	1.68	1.84	11.32 ±	0.70
	3		2.98	2.98	12.94 ±	0.80
	4		3.86	3.86	13.67 ±	0.85
Φ_{ring}	2	200	1.68	1.84	15.73 ±	0.98
	3		2.98	2.98	157.48 ±	9.76
	4		3.86	3.86	1,707.21 ±	105.81
Φ_{unbal}	5	5	57.01	30.00	32.44 ±	2.01
	2	51	199.38	104.00	122.23 ±	7.58
	4	25	200.59	104.00	144.75 ±	8.97
	8	12	201.25	104.00	152.41 ±	9.45
	13	7	200.38	104.00	343.04 ±	21.26
Φ_{split}	20		18.55	20.00	18.57 ±	1.15
	50		48.52	50.00	45.84 ±	2.84
	100		98.51	100.00	207.37 ±	12.85
Φ_{speed}	20		18.55	20.00	12.57 ±	0.78
	50		48.52	50.00	16.74 ±	1.04
	100		98.51	100.00	24.29 ±	1.51
Φ_{nohalt}	20		18.55	20.00	13.23 ±	0.82
	50		48.52	50.00	23.02 ±	1.43
	100		98.51	100.00	52.21 ±	3.24
Φ_{safe}	20		11.36	13.33	13.50 ±	0.68
	50		31.34	33.33	19.67 ±	1.00
	100		64.67	66.67	31.87 ±	1.61

We notice generally that the time needed by Z3 to solve an instantiated formula increases with the number of connector and component instances. However, for complex formulas such as Φ_{ring} where we have a universally quantified second-order variable ($\forall X : 2^{\text{Master}}$) and an alternation of quantifiers, we are unable to handle rings of size larger than 4. Nonetheless for most formulas, including ones with second-order variables (Φ_{speed}, Φ_{unbal}, Φ_{safe}), we are able to handle systems containing up to 100 components.

We also notice sensitivity to different parameters, since the exploration of the models for the formula by the solver is affected by its structure, and the universe over which the variables are quantified. For the load (un)balancing property Φ_{unbal}, we observe worse solving times when we have more masters even if the number of component instances is the same. The instantiated formula is solved much faster when we have 2 masters and 51 slaves per master, than when we have 13 masters and 7 slaves per master, where both amount to 104 component instances.

7 RELATED WORK

The use of configuration logics for specifying static architectures and/or architecture styles has been investigated in [17]. A model checking procedure has been proposed which relies on the syntactic transformation to a normal form, followed by the checking of every term of this form against the configuration model. In [19], configuration logics are extended to express precedence constraints on execution of interactions. Yet, the systems considered have a static architecture.

Formal verification of dynamic reconfigurable systems has been considered in [4] where system configurations are expressed as annotated hyper-graphs, reconfiguration is specified using graph transformations, and configuration properties are specified using a variant of first-order logic. In this work, Alloy Analyzer [11] is used as a back-end tool to generate valid configurations and/or to check invariant properties on finite traces. No temporal logic aspects were considered.

The use of temporal logic for expressing properties about the evolution of reconfigurable systems dates back to [1] and has been widely investigated in several contexts, e.g., recently for dynamic product lines [22]. Nonetheless, its use in combination with richer configuration logics for expressing architectural properties remains marginal, with few exceptions such as [16] which consider a specific computational model of reconfigurable systems.

Temporal logic has been studied in [5, 15] in the context of runtime verification for reconfigurable systems. Architectural configurations are expressed in the B language and validated on instantiated models with the ProB[1] model checker. For temporal aspects, an extension of LTL is used. In contrast, our proposed configuration logic aims to maintain the separation between architectural constraints and state constraints, and relies on state-of-the-art LTL runtime checkers in order to provide more effective decision procedures and achieve scalability for larger systems.

8 CONCLUSION

The contribution of the paper is two-fold. The first is on modeling and specification of dynamic reconfigurable systems allowing deeper insight into key characteristics of their behavior and properties. In contrast to static component-based systems, the knowledge of the states of the constituent components is not enough to capture system dynamics; two instances of the same type of component may exhibit very different behavior depending on their position in the system coordination structure. The application of specification and verification techniques based on temporal logic, relies on the concept of configuration which combines state and architecture information. The second contribution is on runtime verification of temporal logic properties. The method uses an effective procedure for checking the validity of configuration logic formulas on finite configurations. Temporal logic formulas are checked using existing runtime verification techniques as their atomic propositions are configuration logic formulas. The preliminary experimental results concerning the verification of two non trivial examples are encouraging. Future work will focus on the investigation of verification techniques for unrestricted TCL and further improvement of the performance of the DRBIP-VERIFY tool.

[1] https://www3.hhu.de/stups/prob

REFERENCES

[1] Nazareno Aguirre and T. S. E. Maibaum. 2002. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002)*. IEEE Computer Society, 271–274.

[2] Rim El Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. 2018. Programming Dynamic Reconfigurable Systems. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Proceedings (LNCS)*, Kyungmin Bae and Peter Csaba Ölveczky (Eds.), Vol. 11222. Springer, 118–136.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 14.

[4] Antonio Bucchiarone and Juan P. Galeotti. 2008. Dynamic Software Architectures Verification using DynAlloy. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 10 (2008).

[5] Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. 2010. Using Temporal Logic for Dynamic Reconfigurations of Components. In *Formal Aspects of Component Software - 7th International Workshop, FACS 2010, Revised Selected Papers (LNCS)*, Luís Soares Barbosa and Markus Lumpe (Eds.), Vol. 6921. Springer, 200–217.

[6] Antoine El-Hokayem, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. 2020. A Layered Implementation of DR-BIP Supporting Run-Time Monitoring and Analysis. In *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Proceedings (LNCS)*, Frank S. de Boer and Antonio Cerone (Eds.), Vol. 12310. Springer, 284–302.

[7] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. 2012. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.* 14, 3 (2012), 349–382.

[8] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. 2002. Self-organising software architectures for distributed systems. In *Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002*, David Garlan, Jeff Kramer, and Alexander L. Wolf (Eds.). ACM, 33–38.

[9] Dan Hirsch, Paola Inverardi, and Ugo Montanari. 1999. Modeling Software Architecutes and Styles with Graph Grammars and Constraint Solving. In *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1) (IFIP Conference Proceedings)*, Patrick Donohoe (Ed.), Vol. 140. Kluwer, 127–144.

[10] Institute for Software Engineering and Programming Languages, University of Lübeck. 2020. LamaConv - Logics and Automata Converter Library. https://www.isp.uni-luebeck.de/lamaconv.

[11] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

[12] Jung Soo Kim and David Garlan. 2010. Analyzing architectural styles. *J. Syst. Softw.* 83, 7 (2010), 1216–1235.

[13] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. 2012. On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012 (EPiC Series in Computing)*, Pascal Fontaine and Amit Goel (Eds.), Vol. 20. EasyChair, 44–56.

[14] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. 2016. Complexity of Fixed-Size Bit-Vector Logics. *Theory Comput. Syst.* 59, 2 (2016), 323–376.

[15] Arnaud Lanoix, Julien Dormoy, and Olga Kouchnarenko. 2011. Combining Proof and Model-checking to Validate Reconfigurable Architectures. *Electron. Notes Theor. Comput. Sci.* 279, 2 (2011), 43–57.

[16] Diego Marmsoler and Mario Gleirscher. 2016. On Activation, Connection, and Behavior in Dynamic Architectures. *Sci. Ann. Comput. Sci.* 26, 2 (2016), 187–248.

[17] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. 2017. Configuration logics: Modeling architecture styles. *J. Log. Algebraic Methods Program.* 86, 1 (2017), 2–29.

[18] Daniel Le Métayer. 1998. Describing Software Architecture Styles Using Graph Grammars. *IEEE Trans. Software Eng.* 24, 7 (1998), 521–533.

[19] Maria Pittou and George Rahonis. 2020. Architecture Modelling of Parametric Component-Based Systems. In *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Proceedings (LNCS)*, Simon Bliudze and Laura Bocchi (Eds.), Vol. 12134. Springer, 281–300.

[20] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, 1977*. IEEE Computer Society, 46–57.

[21] Amir Pnueli and Aleksandr Zaks. 2006. PSL Model Checking and Run-Time Verification Via Testers. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Proceedings (LNCS)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.), Vol. 4085. Springer, 573–586.

[22] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2017. Extending Dynamic Software Product Lines with Temporal Constraints. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017*. IEEE Computer Society, 129–139.

[23] Jos Warmer and Anneke Kleppe. 1998. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley.