Reasoning about Safety and Progress Using Contracts

Imene Ben-Hafaiedh, Susanne Graf, and Sophie Quinton

Université Joseph Fourier, VERIMAG

Abstract. Designing concurrent or distributed systems with complex architectures while preserving a set of high-level requirements through all design steps is not a trivial task. Building upon a generic notion of *contract framework* which relies on a *component framework* and two refinement relations: *conformance* and *refinement under context*, we provide a condition under which circular reasoning can be used for checking *dominance*, i.e. refinement between contracts. We then propose an instantiation of such a contract framework for safety and progress requirements in component systems with data exchange. This allows us to prove non-trivial properties of a protocol for tree-like networks.

1 Introduction

We aim at a scalable methodology for design and verification of distributed component systems of arbitrary size with complex architectures which preserves a set of high-level requirements through all design steps. Like in contract-based design [1], we use contracts to constrain, reuse and replace implementations.

In this paper we formalize and extend the verification methodology introduced in [2] to distributed component systems of arbitrary size and we show its usefulness for proving safety and progress properties in networked systems. This methodology consists in two phases: (1) define a general notion of contract framework stating the necessary ingredients — a component framework, notions of conformance (for ensuring global properties φ), satisfaction (of contracts by implementations), and dominance (refinement between contracts). Rules for establishing dominance and validity conditions for them are provided. (2) for any particular application, one only has to define instantiations of these generic notions and check the validity conditions. Once the concrete framework has been defined, the rules for dominance can be applied without any further proofs.

For expressing the rich, yet abstract specifications required by our example, we propose a formalism similar to symbolic transition systems as introduced in [3], which we extend in several ways. We define progress constraints generalizing the usual strong and weak fairness and we decorate control states with invariants on state variables. We also consider an explicit composition model represented by sets of *connectors*. Each connector defines a set of interactions and a transformation on (non persistent) port variables, where *ports* name transitions of the local components involved in the interaction. For achieving scalability, we base

© Springer-Verlag Berlin Heidelberg 2010

verification on an abstract semantics in which explicit values of state variables are abstracted by the defined state invariants. Given the complexity of the specifications, not having to prove the correctness of the proof rules in this concrete setting is very helpful.

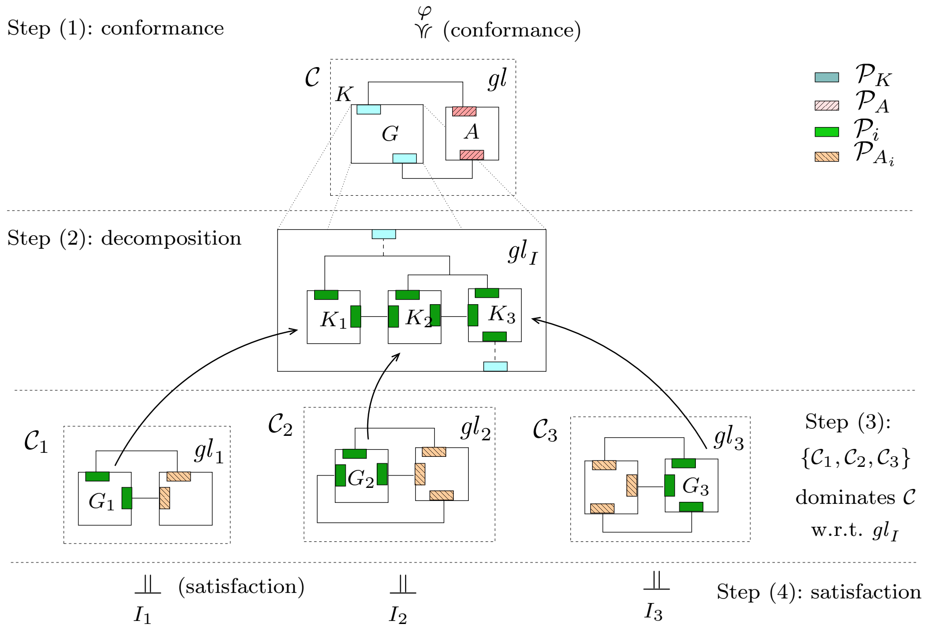


Fig. 1. Methodology steps ensuring that $gl\{A, gl_I\{I_1, I_2, I_3\}\} \preccurlyeq \varphi$

Methodology. Figure 1 illustrates our design and verification methodology. It is represented in a top-down fashion in which high-level properties are pushed progressively from the overall system into atomic components — which we call implementations. As usual, this is just a convenient representation; in real life, we will always achieve the final picture in several iterations alternatively going up and down. We are interested in systems with a complex architecture which are potentially of arbitrary size.

We suppose given a global property φ which the system K under construction has to realize together with an environment on which we may have some knowledge, expressed by a property A. φ and A are expressed w.r.t the interface $\mathcal{P}_K$ of K. We proceed as follows: (1) define a *contract* $\mathcal{C}$ for $\mathcal{P}_K$ which *conforms* to φ; (2) define K as a composition of subcomponents K_i and a contract $\mathcal{C}_i$ for each of them; possibly iterate this step if needed. (3) prove that any set of implementations (components) for K_i *satisfying* the contracts $\mathcal{C}_i$, when composed, satisfies the top-level contract $\mathcal{C}$ (*dominance*) — and thus guarantees φ; (4) provide such implementations.

The global property φ appears at the top of Figure 1, while the implementations I_i are at the bottom.

The correctness proof for a particular system is split into 3 phases: *conformance* of the top-level contract $\mathcal{C}$ to φ, *dominance* between the contracts $\mathcal{C}_i$ and $\mathcal{C}$, *satisfaction* of the $\mathcal{C}_i$ by the implementation I_i.

To be more precise, we use the notion of *context* for an interface $\mathcal{P}$ to describe how a component with interface $\mathcal{P}$ is intended to be connected to its environment and provides a property A expected from this environment. In the sequel, we denote composition operators by gl — standing for *"glue"* [4]. A context is then of the form (gl, A). A *contract* for an interface $\mathcal{P}$ consists of a context (gl, A) and a property G on $\mathcal{P}$ that the component under design must ensure in the given context in order to *satisfy* this contract. *Conformance* relates properties of closed systems and *dominance* relates contracts.

Related work. We propose here 3 improvements with respect to [2]: (a) we do not suppose a fixed composition operator: we encompass any composition satisfying some basic properties; (b) we extend the definition of contract framework to take into account port hiding which is a key ingredient for proving refinement between specifications at different levels of granularity; (c) we provide a complex application using an instantiation with variables and data transfer and allowing expression of liveness properties. The proof steps are performed automatically.

Interfaces [5] have been proposed for a purpose similar to ours. However, we are interested here in rich exogenous composition operators which allow to represent abstractions of protocols, middleware components and orchestrations whereas assumptions and guarantees should constrain peers at the same or at an upper layer. These composition operators cannot be encoded into interface automata, which are I/O based.

Other formalisms for describing such rich connectors abstractly have been proposed, e.g., the Kell calculus [6] or the connector calculus Reo [7]. Kell is, however, mainly concerned with obtaining correctly typed connectors, and Reo supposes independence amongst connectors and does not take into account constraints imposed by components. The composition operators used in our application are defined using a subset of the rich connectors of the BIP component framework [8] because these connectors have the required expressiveness, define interaction with component behaviors and handle conflicting connectors.

Organization. Section 2 introduces and extends the notions from [2] of contract framework and properties that *conformance*, *dominance* and *satisfaction* must ensure in order to support this methodology. In section 3, we give an instantiation of this framework based on symbolic transition systems and rich connectors, which is expressive enough for the safety and progress properties we want to prove. Finally, Section 4 applies the methodology to a resource sharing algorithm in a networked system of arbitrary size: the actual conformance, dominance and satisfaction proofs are automated in a tool developed for this purpose.

2 A Contract-Based Design Framework and Methodology

We develop our methodology on a generic framework that supports hierarchical components and mechanisms to reason about composition. The following notions and properties form the basis of this framework. Here, we use glue operators [4] to generalize the operation of parallel composition found in most traditional frameworks. The notion of component is intentionally kept very abstract to encompass various frameworks. It can be e.g. a labeled transition system, but it can also have a structural part, e.g. it can be a BIP component.

Definition 1 (Component framework). *A* component framework *is a structure of the form* $(\mathcal{K}, GL, \circ, \cong)$ *where:*

– $\mathcal{K}$ *is a set of* components*. Each component* $K \in \mathcal{K}$ *has as its* interface *a set of* ports*, denoted* $\mathcal{P}_K$*.*
– GL *is a set of* glue (composition) operators*. A glue is a partial function* $2^{\mathcal{K}} \longrightarrow \mathcal{K}$ *transforming a set of components into a new component. Each* $gl \in GL$ *is associated with a set of ports* S_{gl} *from the original set of components — called its* support set *— and a new interface* $\mathcal{P}_{gl}$ *for the new component — called its* exported interface*. A composition* $K = gl(\{K_1, \ldots, K_n\})$ *is defined if* $K_1, \ldots, K_n \in \mathcal{K}$ *have disjoint interfaces,* $S_{gl} = \bigcup_{i=1}^{n} \mathcal{P}_{K_i}$ *and the interface of* K *is* $\mathcal{P}_{gl}$*, the exported interface of* gl*.*
– $\cong \subseteq \mathcal{K} \times \mathcal{K}$ *is an equivalence relation. In general, this equivalence is derived from equality or equivalence of semantic sets.*
– $\circ$ *is a partial operation on* GL *to hierarchically compose glues.* $gl \circ gl'$ *is defined if* $\mathcal{P}_{gl'} \subseteq S_{gl}$*. Then, its support set is* $S_{gl} \setminus \mathcal{P}_{gl'} \cup S_{gl'}$ *and its interface is* $\mathcal{P}_{gl}$ *(cf. Figure 2). Furthermore,* $\circ$ *must be coherent with* $\cong$ *in the sense that* $gl\{gl'\{\mathcal{K}_1\}, \mathcal{K}_2\} \cong (gl \circ gl')\{\mathcal{K}_1 \cup \mathcal{K}_2\}$ *for any sets of components* $\mathcal{K}_i$ *such that all terms are defined.*

To simplify the notation, we write $gl\{K_1, \ldots, K_n\}$ instead of $gl(\{K_1, \ldots, K_n\})$. Figure 2 shows how hierarchical components and connectors are built from atomic ones. Note that exported ports of internal connectors (which are not connected) are not represented in this figure.

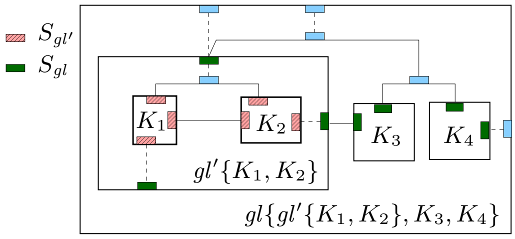


Fig. 2. Hierarchical components and connectors

We use the notion of *context* to restrict how a component may be further composed. The set of contexts is denoted Γ.

Definition 2 (Context). *A* context *for an interface* $\mathcal{P}$ *is a pair* (E, gl) *where* $E \in \mathcal{K}$ *is such that* $\mathcal{P} \cap \mathcal{P}_E = \emptyset$ *and* $gl \in GL$ *is defined on* $\mathcal{P} \cup \mathcal{P}_E$*.*

We introduce two refinement relations to reason about contracts: *conformance*, which we informally introduced when discussing our methodology; and *refinement under context*, used to define *satisfaction* and *dominance*. Refinement under context is usually considered as a derived relation and chosen as the weakest relation implying conformance and ensuring *compositionality*, i.e., preservation by composition. We loosen the coupling between these two refinements to obtain stronger reasoning schemata for dominance.

Definition 3 (Contract framework). *A* contract framework *is a tuple* $(\mathcal{K}, GL, \circ, \cong, \{\sqsubseteq_c\}_{c \in \Gamma}, \preccurlyeq)$ *where:*

- $(\mathcal{K}, GL, \circ, \cong)$ *is a component framework.*
- $\{\sqsubseteq_c\}_{c \in \Gamma}$ *is a set of* refinement under context *relations, one for each context in Γ. Given a context (E, gl) for an interface $\mathcal{P}$, $\sqsubseteq_{E,gl}$ is a preorder over the set of components on $\mathcal{P}$.*
- $\preccurlyeq \subseteq \mathcal{K} \times \mathcal{K}$ *is a* conformance *relation between components with the same interface. It is a preorder such that for any K_1, K_2 on the same interface $\mathcal{P}$ and for any context (E, gl) for $\mathcal{P}$, $K_1 \sqsubseteq_{E,gl} K_2 \implies gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$.*

Example 1. Typical notions of conformance $\preccurlyeq$ are *trace inclusion* and *simulation.*

For these notions of conformance, refinement under context (denoted $\sqsubseteq^{\preccurlyeq}$) is usually defined as the weakest preorder included in $\preccurlyeq$ that is compositional:

$$K_1 \sqsubseteq^{\preccurlyeq}_{E,gl} K_2 \triangleq gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$$

Note that there are cases where a stronger notion of refinement under context allows more powerful reasoning, e.g. circular reasoning as used later in this paper.

Definition 4 (Contract). *A* contract $\mathcal{C}$ *for an interface* $\mathcal{P}$ *consists of:*

- *a context $\mathcal{E} = (A, gl)$ for $\mathcal{P}$; A is called the* assumption
- *a component G on $\mathcal{P}$ called the* guarantee

We write $\mathcal{C} = (A, gl, G)$ rather than $\mathcal{C} = ((A, gl), G)$. The interface of the environment is implicitly defined by gl while A expresses a constraint on it and G a constraint on the refinements of K. The "mirror" contract $\mathcal{C}^{-1}$ of $\mathcal{C}$ is (G, gl, A), i.e. a contract for the environment.

Definition 5 (Satisfaction of contract). *A component K* satisfies *a contract $\mathcal{C} = (A, gl, G)$, denoted $K \models \mathcal{C}$, if and only if $K \sqsubseteq_{A,gl} G$.*

In interface theories [5], a single automaton is used to represent both A and G (gl is predefined), namely $gl\{A, G\}$. Only one pair (A, G) corresponds to an interface, because each transition is controlled either by the component or the environment. However, in frameworks with rendez-vous interaction, several pairs (A, G) can correspond to the same interface, as both the component and its environment may prevent a rendez-vous from taking place. This is why we keep assumptions and guarantees separate.

Our notion of contract has a structural part, which makes this definition very general by encompassing any composition framework. A more practical advantage is related to system design: it allows us to separate the architecture and the requirements of the system under construction, which evolve independently during the development process. In particular, in frameworks where interaction is rich, refinement can be ensured by relying heavily on the structure of the system and less importantly on the behavioral properties of the environment.

Dominance is the key notion that distinguishes reasoning in a contract or interface framework from theories based on refinement between components. Contract $\mathcal{C}$ is said to dominate contract $\mathcal{C}'$ if every implementation of $\mathcal{C}$ — i.e., every component satisfying $\mathcal{C}$ — is also an implementation of $\mathcal{C}'$. Intuitively, this is achieved by a $\mathcal{C}'$ that has a stronger promise or a weaker assumption than $\mathcal{C}$.

In our general setting — which does not refer to any particular composition or component model — it is not sufficient to define dominance just on a pair of contracts. A typical situation that we have to handle is that of a hierarchical component depicted in Figure 1, where a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ is defined for the inner components (on disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$) and a contract $\mathcal{C}$ for the hierarchical component whose interface is the exported interface of a composition operator gl_I defined on $P = \bigcup_{i=1}^n \mathcal{P}_i$. It looks attractive to solve such a problem by defining a contract algebra as in [9], as checking dominance boils then down to checking whether $\widetilde{gl}\{\mathcal{C}_1, \dots, \mathcal{C}_n\}$ dominates $\mathcal{C}$ for some operator $\widetilde{gl}$ on contracts. This is, however, not possible for arbitrary component frameworks. We thus provide a broader dominance defined directly for a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ and a contract $\mathcal{C}$ to be dominated w.r.t a composition operator gl_I.

In order to allow hiding ports of the lower-level contracts which do not appear at the interface of the top-level contract, we relax the constraints on the composition operators by only requiring that they agree on their common ports. For this, we need a notion of *projection* of a component K onto a subset P' of its interface, which defines a component denoted $\Pi_{P'}(K)$ with interface P'. This notion is quite natural and must preserve some properties detailed in [10]. Hence the following semantic definition of dominance (notations are those of Figure 1).

Definition 6 (Dominance). *$\{\mathcal{C}_i\}_{i=1}^n$ dominates $\mathcal{C}$ w.r.t. gl_I iff:*

- *for every i, there exists a glue gl_{E_i} s.t. $gl \circ gl_I = gl_i \circ gl_{E_i}$*
- *for any components $\{K_i\}_{i=1}^n$, $(\forall i, K_i \models \mathcal{C}_i) \implies \Pi_P(gl_I\{K_1, \dots, K_n\}) \models \mathcal{C}$*

We present a generalization of the sufficient condition for dominance proposed in [2] that handles port hiding. The proof is similar to that of [2] and requires a specific property called soundness of circular reasoning. Circular reasoning is sound if for any K, G, A, E, gl such that the terms are defined, the following holds: $K \sqsubseteq_{A,gl} G \wedge E \sqsubseteq_{G,gl} A \implies K \sqsubseteq_{E,gl} G$. More details are given in [10].

Theorem 1. *If circular reasoning is sound and $\forall i.\ \exists gl_{E_i}.\ gl \circ gl_I = gl_i \circ gl_{E_i}$, then to prove that $\{\mathcal{C}_i\}_{i=1..n}$ dominates $\mathcal{C}$ w.r.t. gl_I, it is sufficient to prove that:*

$$\begin{cases} \Pi_P(gl_I\{G_1, \dots, G_n\}) \models \mathcal{C} \\ \forall i, \Pi_{\mathcal{P}_{A_i}}(gl_{E_i}\{A, G_1, \dots, G_{i-1}, G_{i+1}, \dots, G_n\}) \models \mathcal{C}_i^{-1} \end{cases}$$

This shows that the proof of a dominance relation boils down to a set of refinement checks, one for proving refinement between the guarantees, the second for discharging individual assumptions. A proof is given in [10].

Methodology. We now extend our design and verification methodology to recursively defined systems so that we can handle systems representing component networks of arbitrary size defined by a component grammar as follows:

- a set of terminal symbols $\{A, I_1, \dots, I_k\}$ representing implementations;
- a set of nonterminal symbols $\{S, K_0, K_1, \dots, K_n\}$ representing hierarchical components; S, which defines the top-level closed system, is the axiom;
- a set of rules corresponding to design steps which define each non-terminal either as a composition of subsystems or as an implementation:
  - $S \longrightarrow gl\{A, K_0\}$.
  - For $i \in [0, n]$, at least one rule either of the form $K_i \longrightarrow I_j$ $(j \in [1, k])$ or $K_i \longrightarrow gl_{\Sigma_i}\{K_j\}_{j \in \Sigma_i}$, where Σ_i a set of indices and gl_{Σ_i} a composition operator on the union of the interfaces of the K_j.

Unlike classical network grammars, we use “rich” composition operators and are not limited to flat regular networks, as for example in [11]. We now instantiate the methodology of Figure 1 for such component networks. The same four steps are presented, namely conformance, decomposition, dominance and satisfaction.

1. formulate a top-level requirement φ characterizing the closed system defined by the system and its environment, define a contract $\mathcal{C} = (A, gl, G)$ associated with K_0 and prove that $gl\{A, G\} \preccurlyeq \varphi$
2. define for every non terminal K_i a contract $\mathcal{C}_{K_i} = (A_{K_i}, gl_{K_i}, G_{K_i})$ such that for every rule $K_l \longrightarrow gl_{\Sigma_l}\{K_j\}_{j \in \Sigma_l}$ having an occurrence of K_i on the right hand side, there exists gl_{E_i} such that $gl_{K_l} \circ gl_{\Sigma_l} = gl_{K_i} \circ gl_{E_i}$
3. for each $K_i \longrightarrow gl_{\Sigma_i}\{K_j\}_{j \in \Sigma_i}$, show that $\{\mathcal{C}_{K_j}\}_{j \in \Sigma_i}$ dominates $\mathcal{C}_{K_i}$ w.r.t gl_{Σ_i}
4. prove that implementations satisfy their contract: $K_i \longrightarrow I_j \implies I_j \models \mathcal{C}_{K_i}$

Theorem 2. *Let $\mathcal{G}$ be a grammar such that all methodology steps have been completed to guarantee a requirement φ. Any component system corresponding to a word accepted by $\mathcal{G}$ satisfies φ.*

The proof is a simple induction on the number of steps required for deriving the component from S, showing that conformance is preserved from the left-hand side to the right-hand side of a rule. If φ can express progress and if dominance preserves progress, this methodology is sufficient for systems with a unique requirement but also for multiple requirements decomposed according to the same network grammar.

3 A Contract Framework with Data for Safety and Progress

In this section, we define a contract framework in order to prove safety and progress properties of distributed systems. We choose to use composition operators based on the BIP interaction model [4,12] because of their expressiveness and their properties making them suitable for structural verification. Our framework handles variables, guards and data transfer — which are supported by the BIP interaction model [13] — and furthermore is adequate for loose specifications.

Components. A component is defined by a labeled transition system enriched with variables. In order to allow abstract descriptions of components, we handle predicates on variables rather than concrete values. Except for τ, which denotes internal actions, labels are ports of the component interface. For example, a transition t labeled by port p denotes that the component will perform the action associated with t only if an interaction in which p is involved happens. Our components also provide some progress properties which are described below.

A port p is sometimes represented along with its associated variables $x_1, \ldots, x_n$, which is denoted $p[x_1, \ldots, x_n]$. Without loss of generality, we suppose in the following that a port is associated with exactly one variable. We suppose given a set of predicates that is closed by $\wedge$ and $\vee$.

Definition 7 (Component). *A* component *is a tuple* $(TS, X, Inv, g, f, Prog)$*:*

- $TS = (Q, q^0, P \cup \{\tau\}, \longrightarrow)$ *is a labeled transition system:* Q *is a set of* states*,* $q^0 \in Q$ *is the* initial state*,* $P \cup \{\tau\}$ *is a set of labels.* $\longrightarrow \subseteq Q \times P \cup \{\tau\} \times Q$ *is a* transition *relation. Elements of* P *are* ports *and* τ *labels* internal *transitions. As usual, a transition* $(q, p, q') \in \longrightarrow$ *is denoted* $q \xrightarrow{p} q'$*;*
- X *is a set of* variables*. Some variables are associated with a (unique) port;* $X^{st} \subseteq X$ *contains* state variables *which are denoted* $st_1, \ldots, st_s$*. Relation* R *relates*[1] *variables in* X *to variables in* X^{st}*;*
- Inv *associates with every* $q \in Q$ *a* state invariant Inv_q *that is a predicate on* X^{st}*;*
- g *associates with every transition* t *a* guard g_t*, i.e. a predicate on* X^{st}*;*
- f *associates with every transition* t *an* action f_t *defined as a predicate on* $X^{st} \cup \{x_\gamma\} \cup X^{st}_{new}$ *where* x_γ *is the variable associated with the port labeling* t[2] *and* $X^{st}_{new} = \{st^{new}_1, \ldots, st^{new}_s\}$ *represents the "updated" variables;*
- $Prog$ *a set of* progress properties *(see below).*

Progress properties. When considering abstract specifications, progress properties are useful to exclude behaviors staying forever in some particular states or loops. We adapt usual weak and strong fairness conditions to component systems: a *progress property* $pr \in Prog$ for a component K is a pair of transition sets (T_c, T_p), where T_c is called the *condition* and T_p the *promise.*

We define the set of *progress states* of T_p, denoted $start(T_p)$, as the set of initial states of transitions of T_p.

(T_c, T_p) is a valid progress property iff: considering an execution σ of K in some context containing infinitely many T_c-transitions, in every state of $start(T_p)$ occurring infinitely often, at least one transition of T_p appears infinitely often in σ, unless the environment forbids it. $(\top, T_p)$ denotes *unconditional progress*, which means that σ cannot stay forever in $start(T_p)$ without firing infinitely often a transition of T_p.

Note that (T_c, T_p) is trivially satisfied if no T_c-transition can be fired infinitely often. When T_p is empty or not reachable from any "T_c-loop", (T_c, T_p) is

[1] Non-state variables are transient. R produces their value whenever it is necessary.

[2] If there is no associated variable (t is labeled by p_γ with $\gamma \in \mathcal{I}_{obs}$ or by τ), f_t is a predicate on $X^{st} \cup X^{st}_{new}$.

a progress property only if no T_c-transition can be fired infinitely often. Monotonicity properties w.r.t. progress which allow inferring new progress properties from existing ones are given in [10].

Semantics. The concrete semantics of a component is the usual SOS semantics for labeled transition systems. We do not need it in the following because we only work with an abstract semantics of components: the latter is a labeled transition system in which there exists a transition iff there exists a concrete valuation of the variables for which the transition can be fired. Our semantics is a *closed* semantics, because we suppose that the environment of the component does not affect the values of the variables attached to ports labeling transitions. This strongly motivates a design approach based on contracts, that is, on closed systems.

Definition 8 (Abstract semantics). *Let $K = (TS, X, g, f, Inv, Prog)$ be a component. The abstract semantics of K is the transition system $(Q, q^0, P, \hookrightarrow)$ where $q \overset{p[x]}{\hookrightarrow} q'$ iff there exist a transition $t = (q \xrightarrow{p[x]} q')$ such that the predicate $(st_1, \ldots, st_s)\, R\, x \wedge Sem_t$ is satisfiable, where Sem_t denotes $Inv_q \wedge g_t \wedge f_t \wedge Inv_{q'}$.*

Note that a transition $t = (q \xrightarrow{p} q')$ is not preserved in the semantics if f_t is not consistent with $Inv_{q'}$ — meaning that firing t leads to a state in which $Inv_{q'}$ cannot hold. Thus, in order to avoid deadlocks, the state invariants must respect some consistency and completeness conditions.

Composition. We now define the composition operators that allow us to build complex components based on atomic ones. These composition operators are called *interaction models* and they are made of *connectors*.

From the possible synchronizations offered by the BIP framework (see [12]), we keep only two basic types of connectors: *rendez-vous* connectors require *all* ports to be activated in order for the interaction to take place and involve data transfers; interactions in an *observation* connector can take place as soon as *any* port is activated, and no data is exchanged. Adding observation connectors does not modify the set of interactions which can be fired in a given state, so this does not change the behavior of the system, hence their name. Two (or more) connectors of the same type can be composed to build a *hierarchical* connector simply by using the exported port of one connector as an element of the support set of the other.

Definition 9 (Rendez-vous connector). *A* rendez-vous *connector $\gamma = (p[x], P, \delta)$ is defined by:*

- *$p[x]$, the* exported port *and $P = \{p_1[x_1], \ldots, p_k[x_k]\}$, the* support set *of ports*
- *$\delta = (G, \mathcal{U}, \mathcal{D})$ where:*
  - *G is the* guard*, that is, a predicate on $X = \{x_1, \ldots, x_k\}$*
  - *$\mathcal{U}$ is the* upward update function *defined as a predicate on $X \cup \{x\}$*
  - *For $x_i \in X, \mathcal{D}_{x_i}$ is a* downward update function*, i.e. a predicate on $\{x\} \cup \{x_i\}$*

where $\mathcal{D}_{x_i}$ is the function that returns the projection of $\mathcal{D}$ corresponding to x_i.

As *observation* connectors do not involve data transfer, they have neither guard nor $\mathcal{U}$ nor $\mathcal{D}$ predicates. The variables attached to ports are useless and thus hidden. Hence the following definition.

Definition 10 (Observation connector). *An* observation *connector* $\gamma = (p, P)$ *is defined by an* exported port p *and a* support set $P = \{p_1, \dots, p_k\}$.

To avoid cyclic connectors, we require also that $p \notin P$. Two connectors γ_1 and γ_2 are *disjoint* if $p_1 \neq p_2$, $p_1 \notin P_2$ and $p_2 \notin P_1$. Note that P_1 and P_2 may have ports in common, as a port may be connected to several connectors.

We can now define our composition operators as sets of connectors.

Definition 11 (Interaction model). *An interaction model $\mathcal{I}$ defined on a set of ports P is a set of disjoint connectors such that P is the union of the support sets of the rendez-vous connectors of $\mathcal{I}$. We denote by $\mathcal{I}_{rdv}$ the set of its rendez-vous connectors and $\mathcal{I}_{obs}$ the set of its observation connectors.*

We associate with an interaction model $\mathcal{I}$ an interface $\mathcal{P}_{\mathcal{I}}$ consisting of the set of the exported ports of its connectors. This means that the interface of the component resulting from a composition using $\mathcal{I}$ has only these exported ports as labels. $X_{\mathcal{I}}$ denotes the set of variables associated with the ports of $\mathcal{P}_{\mathcal{I}}$.

Merge of connectors is the operation that takes two connectors defining together a hierarchical connector and returns a connector of a basic type. Merge is defined for rendez-vous connectors in [13] (where it is called flattening). We restrict this definition so as to preserve associativity of the upward and downward functions. Merge of observation connectors has been described in [12]. These definitions extend naturally to our interaction models, where rendez-vous and observation connectors are merged separately (see [10]).

We now define composition: given a set of components $K_1, \dots, K_n$ and an interaction model $\mathcal{I}$, we build a compound component denoted $\mathcal{I}\{K_1, \dots, K_n\}$, with $\mathcal{P}_{\mathcal{I}}$ as interface. As we do not allow sets of ports as labels of transitions, we require that connectors of $\mathcal{I}$ have at most one port of the same component in their support set. Composition is rather technical but not surprising. It does not involve hiding of ports. Besides, a variable of $\mathcal{I}\{K_1, \dots, K_n\}$ is a variable of some K_i or a variable associated with the exported port of some $p_\gamma \in \mathcal{I}$.

Definition 12 (Composition of components). *Let $\{P_i\}_{i=1}^n$ be a family of pairwise disjoint interfaces and $P = \bigcup_{i=1}^n P_i$. Let $\mathcal{I}$ be an interaction model on P. For $i \in [1, n]$, let $K_i = (TS_i, X_i, g_i, f_i, Inv_i, Prog_i)$ be a component on P_i. The* composition *of $K_1, \dots, K_n$* with *$\mathcal{I}$ is a component $(TS, X, g, f, Inv, Prog)$ such that:*

- *$TS = (Q, q^0, \mathcal{P}_{\mathcal{I}} \cup \{\tau\}, \longrightarrow)$ with $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \dots, q_n^0)$ and where $\longrightarrow$ is the least set of transitions satisfying the following rules*[3]*:*

$$\frac{(p_\gamma, P_\gamma, \delta_\gamma) \in \mathcal{I}_{rdv} \quad \forall i \in [1,n].\ q_i \xrightarrow{P_i \cap P_\gamma}_i q_i'}{(q_1, \dots, q_n) \xrightarrow{p_\gamma} (q_1', \dots, q_n')} \qquad \frac{\exists i \in [1,n].\ q_i \xrightarrow{\tau}_i q_i'}{(q_1, \dots, q_n) \xrightarrow{\tau} (q_1, \dots, q_i', \dots, q_n)}$$

with the convention that $q_i \xrightarrow{\emptyset}_i q_i'$ iff $q_i = q_i'$. Note that $|P_i \cap P_\gamma| \leq 1$.

[3] The rule for connectors in $\mathcal{I}_{obs}$ is similar to the one for rendez-vous connectors except that any subset of the support set P_γ may participate in the interaction.

- $X^{st} = \bigcup_{i=1}^{n} X_i^{st}$ *and* $X = \bigcup_{i=1}^{n} X_i \cup X_{\mathcal{I}}$
  The relation R between variables in X and state variables is defined as:
  Case 1: $x \in X_i$ *for some* $i \in [1,n]$. $x\, R\, (st_1, ..., st_s)$ *iff* $x\, R_i\, (st_1^i, ..., st_{s_i}^i)$, *where* $\{st_1^i, ..., st_{s_i}^i\} = X_i^{st} \subseteq X^{st}$.
  Case 2: $x \in X_{\mathcal{I}}$. *Then x is associated with the exported port p_γ of a rendezvous connector* $\gamma = (p_\gamma, P_\gamma, \delta) \in \mathcal{I}_{rdv}$. *Let* $k = |P_\gamma|$. $\mathcal{U}_\gamma$ *is a predicate on* $\{x_1, ..., x_k\} \cup \{x\}$, *where every x_i is associated with a port of P_γ. Without loss of generality, we suppose each x_i is a variable of component K_i.*
  Then $x\, R\, (st_1, ..., st_s)$ *is defined iff:*
  $$\exists v_1, ..., v_k.\ (\forall i \in [1,k].\ v_i\, R_i\, (st_1^i, ..., st_{s_i}^i)) \wedge \mathcal{U}_\gamma[x_1/v_1, ..., x_k/v_k]$$
  where $\mathcal{U}_\gamma[x_1/v_1, ..., x_k/v_k]$ *is the predicate on x obtained by replacing the variables $x_1, ..., x_k$ by values $v_1, ..., v_k$ compatible with the local relations R_i between the x_i and the local state variables.*
- *For each* $q \in Q$, $Inv_q = \bigwedge_{\{i=1\}}^{n} Inv_{q_i}$
- *Consider* $t = (q_1, ..., q_n) \xrightarrow{p_\gamma} (q'_1, ..., q'_n)$ *for* $\gamma \in \mathcal{I}_{rdv}$[4]. *W.l.o.g., we suppose* $P_\gamma = \{x_1, ..., x_k\}$ *with* $x_i \in P_i$ *for every i in* $[1,k]$. *For* $i \in [1,k]$, *the local transition* $(q_i \xrightarrow{p_i[x_i]}_i q'_i)$ *corresponding to t is denoted* $\pi_i(t)$. *Again,* $\{st_1^i, ..., st_{s_i}^i\} = X_i^{st} \subseteq X^{st}$.
  - $g_t(st_1, ..., st_s)$ *holds iff the following holds:*
    - $\forall i \in [1,k].\ g_{t_i}(st_1^i, ..., st_{s_i}^i)$
    - $\exists v_1, ..., v_k.\ (\forall i \in [1,k].\ v_i\, R_i\, (st_1^i, ..., st_{s_i}^i)) \wedge G[x_1/v_1, ..., x_k/v_k]$
  - $f_t(st_1, ..., st_s, x_\gamma, st_1^{new}, ..., st_s^{new})$ *holds iff* $\exists v_1, ..., v_k$ *s.t. it holds that:*
    - $\mathcal{D}[x_1/v_1, ..., x_k/v_k]$, *which is a predicate on* x_γ
    - $\forall i \in [1,k].\ f_{\pi_i(t)}[x_i/v_i]$, *which is a predicate on* $X_i^{st} \cup X_{i,new}^{st}$
- *The set Prog of progress properties is defined below (see definition 13)*

We never explicitly construct all (strongest) progress properties for a composition: compositions are only built as far as needed to prove dominance. Thus, we only give below a condition for checking that a pair of sets (T_c, T_p) is a progress property of a composition by checking that the projections of (T_c, T_p) onto individual components are local progress properties.

Definition 13 (Progress property in a composition). (T_c, T_p) *is a progress property of* $\mathcal{I}\{K_1,\ ...,\ K_n\}$ *if* $\forall i \in [1,n]$:

- *either* $\pi_i(T_p)$ *never contains more than one joint transition of $\mathcal{I}$ from the same state and then* $(\pi_i(T_c), \pi_i(T_p))$ *is a local progress property.*
- *or it does, and then we split* $\pi_i(T_p)$ *into a set of promises* $T_p^{i,1} ..., T_p^{i,k}$ *containing exactly one joint transition for each state before checking that all pairs in* $\{(T_c^i, T_p^{i,1}), ..., (T_c^i, T_p^{i,k})\}$ *are local progress properties*[5].

[4] For a transition labeled by p_γ with $\gamma \in \mathcal{I}_{obs}$, only the conditions on the local guard and function of the components involved in the interaction are kept. The guard and function of a τ-transition are the corresponding local guard and function.

[5] This is necessary to avoid that different processes choose a different joint transition in a given initial state.

Refinement. Refinement under context ensures that in the given context $(E, \mathcal{I})$ — and in any context refining it — safety and progress properties are preserved from the abstract component K_{abs} to the refined component K_{conc}. Moreover, refinement under context allows circular reasoning for the considered composition operators (provided that the assumptions are deterministic), because enabledness of transitions must be preserved from K_{conc} to K_{abs}. But only states reachable in the considered context must be related. To simplify the definition, we suppose that (a) K_{abs} has no internal transitions, (b) E has no transitions that it may do alone and (c) progress is refined without taking into account the context. The first two steps imply no loss of generality. The last simplification is sufficient for the considered application. It could be refined by requiring from K_{conc} only (part of) the progress properties of K_{conc} which are meaningful in $(E, \mathcal{I})$.

Refinement is defined by means of two relations (1) α relating variables of K_{conc} and K_{abs}, and (2) $\mathcal{R}$ relating concrete and abstract states. For preserving progress, we project transition sets of K_{abs} onto K_{conc} — for this purpose, we define the following auxiliary notations.

Definition 14 (Projection). *Let $\mathcal{R}$ be a relation on $(Q_{conc} \times Q_E) \times Q_{abs}$. We define the projection $\overline{\mathcal{R}}$ of $\mathcal{R}$ onto $Q_{conc} \times Q_{abs}$ by $q_c \overline{\mathcal{R}} q_a$ iff $\exists q_E$ s.t. $(q_c, q_E) \mathcal{R} q_a$. For $Q_a \subseteq Q_{abs}$, we denote $\overline{\mathcal{R}}^{-1}(Q_a) \subseteq Q_{conc}$ the inverse image of Q_a under $\overline{\mathcal{R}}$. $\overline{\mathcal{R}}^{-1}(\{q_a \overset{p}{\hookrightarrow}_{abs} q'_a\})$ denotes the set of p-transitions of K_{conc} between states in $\overline{\mathcal{R}}^{-1}(\{q_a\})$ and in $\overline{\mathcal{R}}^{-1}(\{q'_a\})$. This notation extends naturally to transition sets.*

Definition 15 (Refinement under context). *Given a relation α on $X_{conc} \cup X_{abs}$, K_{conc} refines K_{abs} in the context of $(E, \mathcal{I})$, denoted $K_{conc} \sqsubseteq_{E,\mathcal{I}} K_{abs}$, iff:*

(a) $\exists \mathcal{R} \subseteq (Q_{conc} \times Q_E) \times Q_{abs}$ s.t. $(q_c^0, q_E^0) \mathcal{R} q_a^0$ and s.t. $(q_c, q_E) \mathcal{R} q_a$ implies:

1. *$Inv_{q_c} \wedge \alpha(X_{conc}, X_{abs}) \implies Inv_{q_a}$*
2. *$\forall p[x] \in P$, the following holds ($\mathcal{V}_i$ denotes a valuation of X_i):*
   - *for any value v of x: $\exists t_c = q_c \xrightarrow{p}_c q'_c$ and $\mathcal{V}_c, \mathcal{V}_c^{new}$ satisfying Sem_{t_c} implies $\exists q'_a, t_a = q_a \xrightarrow{p}_a q'_a$ and $\mathcal{V}_a, \mathcal{V}_a^{new}$ consistent with α and satisfying Sem_{t_a}.*
   - *$\exists \gamma.\, P_\gamma = \{p, e\} \wedge (q_c, q_E) \overset{p_\gamma}{\hookrightarrow} (q'_c, q'_E) \implies (q'_c, q'_E) \mathcal{R} q'_a$ with q'_a as above*[6]*.*
3. *$q_c \overset{\tau}{\hookrightarrow}_c q'_c \implies (q'_c, q_E) \mathcal{R} q_a$: states related by τ-transitions refine the same state*

(b) The inverse image under $\overline{\mathcal{R}}$ of any progress property $pr = (T_c, T_p)$ of K_{abs}, which is $(\overline{\mathcal{R}}^{-1}(T_c), \overline{\mathcal{R}}^{-1}(T_p))$, is a progress property of K_{conc}.

Condition (a) ensures that refining an abstract component preserves safety properties. Condition (b) ensures preservation of progress properties. The last step to obtain a contract framework is to define conformance.

Definition 16 (Conformance). *Let $K_\perp = (TS, X, Inv, Prog)$ be defined as: $TS = (\{q_0\}, q_0, \emptyset, \emptyset)$, $X = \emptyset$, $I = \top$ and $Prog = \emptyset$. We define conformance as refinement in the context of $(K_\perp, \emptyset)$ — i.e., an "empty" with no connectors.*

Theorem 3. *We have defined a contract framework. Furthermore, if assumptions are deterministic, then circular reasoning is sound. See [10] for a proof.*

[6] If t is independent of the context, i.e., if $P_\gamma = \{p\}$, we use the convention $q_E \overset{\emptyset}{\hookrightarrow}_E q_E$.

4 An Application to Resource Sharing in a Network

We apply the proposed methodology to an algorithm for sharing resources in a network presented in [14]. The starting point is both a high-level property and an abstract description of the behavior of an individual node. We represent networks of arbitrary size by a grammar and associating a contract with each node, such that the correctness proof boils down to a set of small verification steps. We consider networks structured as binary trees defining a token ring[7].

Resources shared between nodes are represented by *tokens* circulating in packets containing one or more tokens along the token ring (see figure 3). The *value* of a packet is the number of tokens it contains. A particular token is the *privilege* — denoted P — which allows nodes to accumulate tokens.

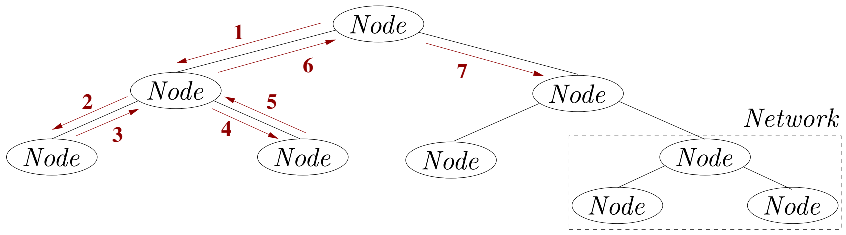


Fig. 3. The overall structure of the application

A node may *request* tokens (Req indicates the numbers of tokens requested). When it has enough tokens for satisfying its request, it is expected to *use* them, and relax the privilege if it has it; when it has resources (tokens) in use, it cannot request additional ones; it may later *free* them or keep them forever. A node can rise a request only when it has no resource in use and no pending request.

Tokens (and the privilege) circulate through ports called get_T (get_P) and $give_T$ ($give_P$), whereas the request, usage or freeing of tokens is indicated through observation ports req, use, $free$. Moreover, a $Node$ has state variables indicating whether it has the privilege (P), its number of tokens (Tk), requests (Req) and some port variables used during interactions.

The network is defined by the grammar $\mathcal{G}$, where $\{E_\perp, Node\}$ are terminals and $\{Sys, Net\}$ nonterminals with axiom Sys. The rules are:

$$Sys \longrightarrow \mathcal{I}_{Net}\{E_\perp, Net), Net \longrightarrow Node, Net \longrightarrow \mathcal{I}\{Node, Net, Net\}$$

The connectors of the composition operators $\mathcal{I}$ and $\mathcal{I}_{Net}$ are indicated in Figures 4 and 5. They handle exchange of tokens and privileges and the observation of requested, used, respectively freed tokens.

We assume that connectivity of the network is guaranteed and tokens are never lost. Here, this assumption is encoded in the composition operator. This allows separating completely design and correctness proofs from the resource sharing algorithm and the algorithm guaranteeing connectivity, which is typically implemented in a lower layer of the overall network protocol.

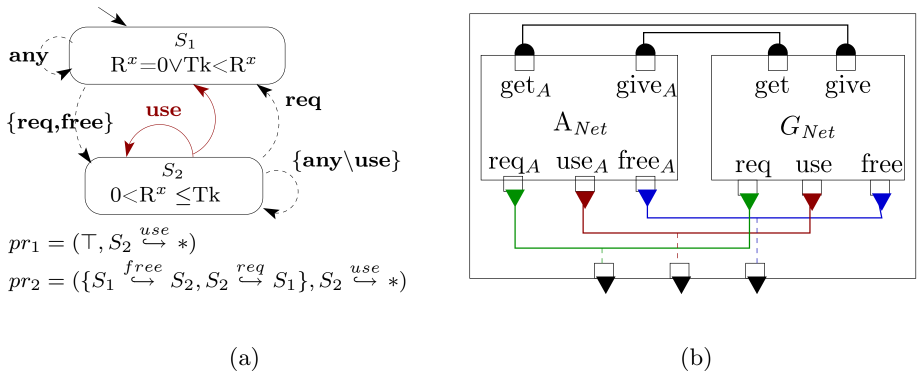
[7] We restrict ourselves to binary branching for simplifying the presentation.

A top-level requirement φ. We consider here one of the top-level requirements of the algorithm, a progress requirement φ stating that "as long as the requests are reasonable, some of the nodes will be served" — *use* will occur — from time to time. φ is represented in our formalism as depicted in Figure 4, where the second progress property pr_2 says that "it is not possible to switch infinitely often between states S_1 and S_2 (that is, *free* occurs infinitely often) without that a *use* occurs infinitely often as well". "Reasonable" requests means that $0 < R^x \leq Tk$ where R^x is the maximal request and Tk the number of available tokens in the system.

Methodology. Our goal is to prove that every network built according to grammar $\mathcal{G}$, together with an environment $E_\perp$ giving back tokens and privilege immediately, conforms to φ. For this purpose, we instantiate the methodology of Section 2:

1. We define $\mathcal{C}_{\text{Node}} = (A_{\text{Node}}, \mathcal{I}_{\text{Node}}, G_{\text{Node}})$ and $\mathcal{C}_{\text{Net}} = (A_{\text{Net}}, \mathcal{I}_{\text{Net}}, G_{\text{Net}})$ that are contracts for component types *Net* and *Node*.
2. We show that $\mathcal{I}_{\text{Net}}\{A_{\text{Net}}, G_{\text{Net}}\} \preccurlyeq \varphi$.
3. We show that $\{\mathcal{C}_{\text{Node}},\ \mathcal{C}_{\text{Net}},\ \mathcal{C}_{\text{Net}}\}$ dominates $\mathcal{C}_{\text{Net}}$ w.r.t. $\mathcal{I}$.
4. We prove that $E_\perp$ satisfies $\mathcal{C}_{\text{Net}}^{-1}$ and that *Node* satisfies $\mathcal{C}_{\text{Node}}$ and $\mathcal{C}_{\text{Net}}$.

Note that if we want to further refine the *Node* component, we may start by a contract $\mathcal{C}_{\text{Node}} = (A_{\text{Net}},\ \mathcal{I}_{Net},\ Node)$. Now, let us give some details.



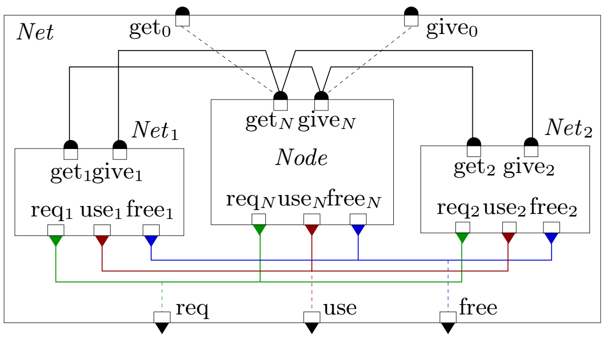
any
S_1
$R^x{=}0 \vee Tk{<}R^x$
{req,free}
use
req
{any\use}
S_2
$0{<}R^x \leq Tk$
$pr_1 = (\top, S_2 \overset{use}{\hookrightarrow} *)$
$pr_2 = (\{S_1 \overset{free}{\hookrightarrow} S_2, S_2 \overset{req}{\hookrightarrow} S_1\}, S_2 \overset{use}{\hookrightarrow} *)$
get$_A$
give$_A$
A$_{Net}$
req$_A$
use$_A$
free$_A$
get
give
G_{Net}
req
use
free
(a)
(b)

Fig. 4. (a) Top-level requirement φ; (b) Composition $\mathcal{I}_{\text{Net}}$ for contract $\mathcal{C}_{\text{Net}}$

Interaction models. Figure 4(b) shows the interaction model $\mathcal{I}_{\text{Net}}$ relating a network — and therefore also a leaf node — to the rest of the system. We represent by *get* and *give* respectively port sets $\{get_T, get_P\}$ and $\{give_T, give_P\}$ for token and privilege exchange. $\mathcal{I}$ consists of 3 observation connectors which export a *use* interaction of either the Net or its environment as a global *use* interaction, and analogously for the others. There are also 4 internal connectors for exchanging tokens and privilege. For example, connector

$\{give_T[tk] \mid get_{TA}[tk_A], \delta_G : [tk > 0], tk_A := tk\}$ pushes a positive number of tokens from the Network to the environment.

Due to lack of space, we do not present the assumptions and guarantees of the node and network contracts. They are detailed in [10].



(a)

1.
$\Pi_{\mathcal{P}_{\mathrm{Net}}}(\mathcal{I}\{G_{\mathrm{Node}}, G_{\mathrm{Net}}, G_{\mathrm{Net}}\}) \models \mathcal{C}_{\mathrm{Net}}$

2.
$\Pi_{\mathcal{P}_{\mathrm{Node}}}(\mathcal{I}_1\{A_{\mathrm{Net}}, G_{\mathrm{Net}}, G_{\mathrm{Net}}\}) \models \mathcal{C}_{\mathrm{Node}}^{-1}$

3.
$\Pi_{\mathcal{P}_{\mathrm{Net}}}(\mathcal{I}_2\{A_{\mathrm{Net}}, G_{\mathrm{Node}}, G_{\mathrm{Net}}\}) \models \mathcal{C}_{\mathrm{Net}}^{-1}$

(b)

Fig. 5. (a) Structure of a network component; (b) Sufficient conditions for dominance

Figure 5(a) shows the inner structure of a network component Net. The interaction model $\mathcal{I}$ builds a tree from a (root) node[8] and two networks Net_1, Net_2. Interactions performed by the connectors depicted here are similar to those of Figure 4(b), except that they also ensure that tokens circulate in the correct order.

Experimental results. To show that $\{\mathcal{C}_{\mathrm{Node}}, \mathcal{C}_{\mathrm{Net}}, \mathcal{C}_{\mathrm{Net}}\}$ dominates $\mathcal{C}_{\mathrm{Net}}$ w.r.t. $\mathcal{I}$, it is sufficient, according to the sufficient condition of section 2, to prove the conditions given in Figure 5(b). Dominance, conformance and satisfaction problems are reduced to refinement under context checked and discharged automatically by a *Java* tool returning either yes or a trace leading to the violation of refinement.

5 Discussion and Future Work

We proposed a design and verification methodology which allows design and verification of safety and progress properties of distributed systems of arbitrary size. This methodology has been successfully applied to an algorithm for sharing resources in a tree-shaped network by automatically discharging the required conformance, dominance and satisfaction checks with a prototype tool.

There are several interesting directions to be explored: (a) We have excluded the use of contracts for assume/guarantee reasoning: we use contracts as design constraints for implementations which are maintained throughout the development and life cycle of the system. On the other hand, in assume/guarantee

[8] Which is connected in a slightly more complex manner than the leaf node.

based compositional verification, assumptions are used to deduce global properties (see [15]). We could integrate this into our methodology: as an example, in our network application, it would be enough to ensure that assumptions express sufficient progress to show conformance of a node contract to “node progress”. (b) We would like to extend the methodology to multiple requirements, possibly by using a different decomposition of the system — i.e. a different grammar. (c) We also intend to extend the component framework to more general connectors and behaviors to express non functional properties. (d) We are currently considering building an efficient checker for different refinement relations, and then, implement tool support for the methodology. We also consider integration into a system design framework — such as SySML promoted by OMG.

References

1. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
2. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Proc. of SEFM 2008, pp. 377–381. IEEE Computer Society, Los Alamitos (2008)
3. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Specification, vol. 1. Springer, Heidelberg (1991)
4. Sifakis, J.: A framework for component-based construction. In: Proc. of SEFM 2005, pp. 293–300. IEEE Computer Society, Los Alamitos (2005)
5. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. of ESEC/SIGSOFT FSE 2001, pp. 109–120. ACM Press, New York (2001)
6. Bidinger, P., Stefani, J.B.: The Kell calculus: operational semantics and type systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 109–123. Springer, Heidelberg (2003)
7. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Strucutres in Computer Science 14(3) (2004)
8. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. of SEFM 2006, pp. 3–12. IEEE Computer Society, Los Alamitos (2006)
9. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
10. Ben-Hafaiedh, I., Graf, S., Quinton, S.: A contract framework for reasoning about safety and progress. Technical Report TR-2010-11, Verimag (2010)
11. Stadler, Z., Grumberg, O.: Network grammars, communication behaviours and automatic verification. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)
12. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in BIP. In: Proc. of EMSOFT 2007, pp. 11–20. ACM Press, New York (2007)
13. Bozga, M., Jaber, M., Sifakis, J.: Source-to-source architecture transformation for performance optimization in BIP. In: Proc. of SIES 2009, pp. 152–160 (2009)
14. Datta, A.K., Devismes, S., Horn, F., Larmore, L.L.: Self-stabilizing k-out-of-l exclusion on tree network. CoRR abs/0812.1093 (2008)
15. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods, vol. 54. Cambridge University Press, Cambridge (2001)