Source-to-Source Architecture Transformation for Performance Optimization in BIP

Marius Bozga, Mohamad Jaber, and Joseph Sifakis

Invited Paper

***Abstract*—Behavior, Interaction, Priorities (BIP) is a component framework for constructing systems from a set of atomic components by using two kinds of composition operators: interactions and priorities.**

In this paper, we present a method that transforms the interactions of a component-based program in BIP and generates a functionally equivalent program. The method is based on the successive application of three types of source-to-source transformations: flattening of components, flattening of connectors, and composition of atomic components. We show that the system of the transformations is confluent and terminates. By exhaustive application of the transformations, any BIP component can be transformed into an equivalent monolithic component. From this component, efficient standalone C++ code can be generated.

The method combines advantages of component-based description such as clarity, incremental construction, and reasoning with the possibility to generate efficient monolithic code. It has been integrated in the design methodology for BIP and it has been successfully applied to two non trivial examples described in this paper.

I. Introduction

COMPONENT-BASED systems are desirable because they allow subsystems to be reused as well as their incremental modification without requiring global changes. Their development requires methods and tools supporting a concept of architecture which characterizes the coordination between components. An architecture is the structure of a system, which involves components and relationships between the externally visible properties of those components. The global behavior of a system can in principle be inferred from the behavior of its components and its architecture.

An advantage of component-based systems is that they have logically clear descriptions. Nonetheless, clarity may be at the detriment of efficiency. Naive compilation of component-based systems results in great inefficiency as a consequence of the interconnection of components [17].

Source-to-source transformations have been considered as a powerful means for optimizing programs [6], [15]. In contrast to conventional optimization techniques, these can be applied for deeper semantics-preserving transformations which are visible to the programmer and subject to his direction and guidance.

Manuscript received October 31, 2009; revised April 02, 2010; accepted August 02, 2010. Paper no. TII-09-10-0289.

The authors are with the Verimag Laboratory, Gieres 38610, France (e-mail: Marius.Bozga@imag.fr; Mohamad.Jaber@imag.fr; joseph.sifakis@imag.fr).

Digital Object Identifier 10.1109/TII.2010.2069102

Source-to-source architecture transformations transform a component-based system into a functionally equivalent system, by changing the structure of its architecture. They may affect performance and quality attributes. They are useful for finding functionally equivalent systems that meet different extra-functional (platform dependent) requirements.

We study transformations for a subset of the Behavior, Interaction, Priority (BIP) language [4], [9] where an architecture is characterized as a hierarchically structured set of components obtained by composition from a set of *atomic components*. In BIP, composition is parameterized by interactions and priorities between the composed components. In this paper, we consider only composition by interactions. Composite components can be hierarchically structured. BIP has been used to model complex heterogeneous systems. It can be considered as an extension of C with powerful primitives for multiparty interaction between components. It has a compilation chain allowing the generation of C++ code from BIP models. The generated code is modular and can be executed on a dedicated platform consisting of an Engine which orchestrates the computation of atomic components by executing their interactions. Hierarchical description allows incremental reasoning and progressive design of complex systems. Nonetheless, it may lead to inefficient programs if structure is preserved at run time. Compared to functionally equivalent monolithic C programs, BIP programs may be more than two times slower. This overhead is due to the computation of interactions between components by the Engine, as illustrated later in this paper on some examples.

The aim of the work is to show that it is possible to synthesize efficient monolithic code from component-based software described incrementally. We study source-to-source transformations for BIP allowing the composition of components and thus leading to more efficient code. These are based on the operational semantics of BIP which allows to compute the meaning of a composite component as a behaviorally equivalent atomic component.

A BIP component is characterized by its *interface* and its *behavior*. An interface consists of a set of ports used to specify interactions. Each port p_i has an associated variable v_{p_i} which is visible when an interaction involving p_i is executed. We assume that the sets of ports and variables of components are disjoint. The behavior of a composite component is obtained by composing the behavior of its atomic components (see Fig. 1).

The behavior of atomic components is described as a Petri net extended with data and functions given in C. A transition of

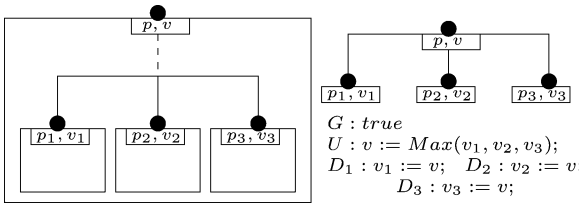


Fig. 1. Example 1.

the Petri net is labeled with a trigger and a function f describing a local computation. A trigger consists of a guard g on (local) data and a port p through which synchronization is sought. For a given marking and data state, a transition can be executed if it is enabled for this marking, its guard g is true and an interaction involving p is possible. Its execution is atomic. It is initiated by the interaction and followed by the execution of f.

Composition consists in applying a set of *connectors* to a set of components. A connector is defined by:

1) its port p and the associated variable v_p;
2) its interaction defined by a set of ports $p_1, \ldots, p_n$ of the composed components ;
3) functions U and $D_1, \ldots, D_n$, specifying the flow of data upstream and downstream, respectively (see Fig. 1).

The global behavior resulting from the application of a connector to a set of components is defined as follows.

An interaction $p_1, \ldots, p_n$ of the connector is possible only if for each one of its ports p_i, there exists an enabled transition in some component labeled by p_i. Its execution involves two steps:

1) the variable v is assigned the value $U(v_{p_1}, \ldots, v_{p_n})$;
2) the variables v_i associated with the ports p_i are assigned values $D_i(v)$.

The execution of an interaction is followed by the execution of the local computations of the synchronized transitions. In Fig. 1, we provide a simple BIP model. It is composed of three atomic components, which compute integers exported through the variables v_1, v_2 and v_3. The connector defines the interaction (strong synchronization) between p_1, p_2 and p_3. As a result of this interaction, each component receives the maximum of the exported values.

A composite component is obtained by successive application of connectors from a set of atomic components. It is a finite set of components equipped with an acyclic *containment relation* and a set of connectors such that: 1) minimal elements are atomic components and 2) if p is the port of a connector then its interaction consists only of ports of components contained in the component with port p. The containment relation defines for each component a level in the hierarchy. A component of level n is obtained by composing a set of components of lower level among which there is at least one component of level $n-1$. The semantics of a composite component is defined from the semantics of atomic components (components at level 0) and the semantics of composition by using connectors. It allows computing for a composite component, an atomic component with an equivalent global behavior.

The main contributions of this paper are the following. We define composite components in BIP and their semantics. We show how by incremental composition of the components contained in a composite component, a behaviorally equivalent component can be computed. This composition operation has been implemented in the BIP2BIP tool, by using three types of source-to-source transformations. A set of interacting components is replaced by a functionally equivalent component. By successive application of compositions, an atomic component can be obtained, that is a component with no interactions.

The transformation from a composite component to an atomic one is fully automated and implemented through three steps.

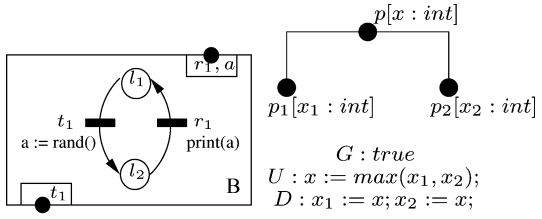
1) *Connector flattening* which replaces the hierarchy on components by a set of hierarchically structured connectors applied on atomic components.
2) *Connector flattening* which computes for each hierarchically structured connector an equivalent flat connector.
3) *Component composition* which composes atomic components to get an atomic component.

Using such a transformation allows to combine advantages of component-based descriptions such as clarity and reuse with efficient implementation. The generated code is readable and by-construction functionally equivalent to the component-based model. We show through nontrivial examples the benefits of this approach.

To the best of our knowledge, we have not seen major work on source-to-source transformations for component-based frameworks. In contrast to other frameworks, component composition in BIP is based on operational semantics. Furthermore, composition can be expressed not only at execution level but also at source level. Similar component frameworks such as [2] and [14] have well-defined denotational semantics. Nonetheless, it is not clear how to define component composition at source level from these semantics. There also exist many component frameworks without rigorous semantics. In this case, using ad hoc transformations, may easily lead to i consistences, e.g., transformations may not be confluent.

Finally, there are strong similarities between our source-to-source transformations for BIP and code optimization techniques applied by language compilers. For example, compilation of synchronous languages [16], or optimizing communications in periodic reactive systems [12], [13] use very similar techniques for flattening structure and composing Petri net behavior. Nonetheless, their underlying models are completely much simpler than BIP. The model considered in [12] and [13] is an extension of Kahn process networks allowing nondeterministic waiting on multiple input channels. The communication is binary, through point-to-point message passing on FIFO channels. In contrast, BIP allows for general n-ary (multiparty) synchronous interactions, allowing to simultaneously transfer data between any number of components. Kahn process networks and their extensions can be structurally represented in BIP, whereas the reverse is not feasible, in general.

Additionally, the intermediate models used for compilation of Esterel [16] are various forms of control flow graphs (sequential, concurrent, etc.). Although we are using a similar model for representation of behavior in atomic components, we believe that such models are not equally appropriate for system or application-level description. Contrarily to BIP, these models do not enforce any separation between computation, coordination and communication. All these issues get mixed due to the arbitrary use of primitives such as explicit fork/join, or communication



(a) (b)

Fig. 2. (a) Example of an atomic component in BIP. (b) A connector with two ports p_1, p_2 and exported port$p$.

through shared variables. Clearly, control-flow graphs models can be efficiently implemented, most likely when they are produced from high-level structural descriptions, however, they are in general more difficult to understand and analyze.

This paper is structured as follows. In Section II, we define the syntax for the description of structured components in BIP. In Section III, we define the semantics by successive application of the three source-to-source transformations. In Section IV, we provide benchmarks for two examples: a MPEG encoder and a concurrent sorting program. In Section V, we discuss other applications and future developments.

II. Component-Based Construction

We define atomic components and their composition in BIP.

Definition 1 (Port): A port $p[x]$ is defined by:

- p—the port identifier.
- x—the data variable associated with the port.

An atomic component is a Petri net extended with data. It consists of a set of ports P used for the synchronization with other components, a set of transitions T and a set of local variables X. Transitions describe the behavior of the component. They are represented as a labeled relation on a set of control locations L.

Definition 2 (Atomic Component): An atomic component B is defined by: $B = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$, where

- (P, L, T) is a Petri net, that is:
  — P is a set of ports;
  — $L = \{l_1, l_2, \ldots, l_k\}$ is a set of control locations;
  — $T \subseteq 2^L \times P \times 2^L$ is a set of transitions.
- $X = \{x_1, \ldots, x_n\}$ is a set of variables and for each transition $\tau \in T$, g_τ is a guard and f_τ is an action $X := f_\tau(X)$.

Fig. 2(a) shows an example of an atomic component with two ports r_1, t_1, a variable a, and two control locations l_1, l_2. At control location l_1, the transition labeled t_1 is enabled. When an interaction through t_1 takes place, a random value is assigned for the variable a. This value is exported through the port r_1. From the control location l_2, the transition labeled r_1 can occur (the guard is true by default), the variable a is eventually modified and the value of a is printed.

We will use the following notations. For a transition $\tau \in T$, we define its preset ${}^\bullet\tau$ (resp. postset $\tau^\bullet$) as the set of the control locations which are direct predecessors (resp. successors) of this transition. Moreover, we use the dot notation to denote the parameters of atomic components. For example, $B.P$ means the set of ports of the atomic component B.

Given a Petri net $N = (P, L, T)$, we define the set of one-safe markings $\mathcal{M}$ as the set of functions $m : L \to \{0, 1\}$. Given two markings m_1, m_2, we define inclusion $m_1 \leq m_2$ iff for all $l \in L$, $m_1(l) \leq m_2(l)$. Also, we define addition $m_1 + m_2$ as the marking m_{12} such that, for all $l \in L$, $m_{12}(l) = m_1(l) + m_2(l)$. Given a set of places $K \subset L$, we define its characteristic marking m_K by $m_K(l) = 1$ for all $l \in K$ and $m_K(l) = 0$ for all $l \in L \setminus K$. Moreover, when no confusion is possible from the context, we will simply use K to denote its characteristic marking m_K.

Definition 3 (Atomic Component Semantics): The semantics of an atomic component $B = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ is defined as the labeled transition system $\mathcal{S}_B = (Q_B, \Sigma_B, \underset{B}{\longrightarrow})$, where

- $Q_B = \mathcal{M} \times \mathcal{V}$ is the set of states defined by:
  — $\mathcal{M} = \{m : L \to \{0, 1\}\}$ the set of one-safe markings;
  — $\mathcal{V} = \{v : X \mapsto \mathcal{D}\}$ is the set of valuations of variables X.
- $\Sigma_B = P \times \mathcal{D}^2$ is the set of labels.
- $\underset{B}{\longrightarrow} = Q_B \times \Sigma_B \times Q_B$ is the set of transitions defined by the following rule:

$$\frac{\tau \in T \quad p \in P \quad {}^\bullet\tau \leq m \quad g_\tau(v) = true \quad v^{up} = v(x_p) \quad m' = m - {}^\bullet\tau + \tau^\bullet \quad v' = f_\tau(v[x_p \mapsto v^{dn}])}{(m, v) \overset{p(v^{up}/v^{dn})}{\underset{B}{\longrightarrow}} (m', v').}$$

Definition 4 (Connector): A connector $\gamma = (p[x], P, \delta)$ is defined as follows:

- p is the exported port of the connector γ.
- $P = \{p_i[x_i]\}_{i\in I}$ is the support set of γ, that is, the set of ports that γ synchronizes.
- $\delta = (G, U, D)$, where
  — G is the guard of γ, an arbitrary predicate $G(\{x_i\}_{i\in I})$;
  — U is the upward update function of γ of the form, $x := F^u(\{x_i\}_{i\in I})$;
  — D is the downward update function of γ of the form, $\cup_{p_i}\{x_i := F^d_{x_i}(x)\}$.

Fig. 2(b) shows a connector with two ports p_1, p_2, and exported port p. Synchronization through this connector involves two steps: 1) The computation of the upward update function U by assigning to x the maximum of the values of x_1 and x_2 associated with p_1 and p_2. 2) The computation of the downward update function D by assigning the value of x to both x_1 and x_2.

For a set of connectors $\Gamma = \{\gamma_j\}_{j\in J}$, we define the dominance relation $\rightarrow$ on Γ as follows:

$$\gamma_i \rightarrow \gamma_j \equiv \gamma_j.p \in \gamma_i.P.$$

That is, γ_i dominates γ_j means that the exported port of γ_j belongs to the support set of γ_i [see Fig. 3(a)].

Definition 5 (Flat Connectors): Γ is a set of *flat connectors*, iff no connector dominates another, that is, $\forall\gamma_i, \gamma_j \in \Gamma$, we have $\gamma_i \not\rightarrow \gamma_j$.

Let Γ be a set of connectors such that $(\Gamma, \rightarrow)$ has no cycle and let $\gamma \in \Gamma$. We denote by:

- $top(\Gamma) = \{\gamma_i \in \Gamma \mid \forall\gamma_k \in \Gamma, \gamma_k \not\rightarrow \gamma_i\}$, the set of maximal connectors in Γ according to $\rightarrow$ relation.
- $tree(\gamma) = \{\gamma_i \in \Gamma \mid \gamma \overset{*}{\rightarrow} \gamma_i\} \cup \{\gamma\}$, the tree of connectors where the root node, at the top, is γ.
- Let $a^t = tree(\gamma)$, $bottom(a^t) = \{\gamma_i \in a^t \mid \forall\gamma_k \in a^t, \gamma_i \not\rightarrow \gamma_k\}$, the leaf connectors of the tree a^t, $support(a^t) = \{p \mid p \in \gamma_i.P, \gamma_i \in bottom(a^t)\}$, the support set of ports for the leaf connectors of the tree a^t.

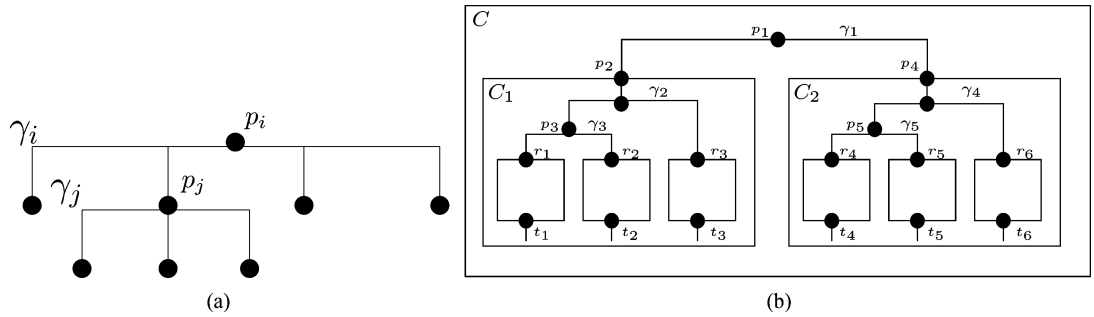


Fig. 3. (a) γ_i dominates γ_j. (b) Example 2.

Definition 6 (Hierarchical Connectors Semantics): Let Γ a set of connectors, and let $a^t = tree(\gamma)$, where $\gamma \in top(\Gamma)$. Let σ_0 be a partial valuation of the variables of the ports in the bottom connectors of a^t, $\sigma_0 = \{p.x \rightarrow v_p \mid p \in support(a^t)\}$. The upward valuation $\mathcal{U}_{a^t}(\sigma_0)$ is obtained by propagating values from ports in the bottom connectors into the tree a^t according to upward update functions $\gamma.F^u$, as long as the guard conditions $\mathcal{G}_{a^t}$ allow them. In a dual manner, we define the downward valuation $\mathcal{D}_{a^t}(\sigma)$ obtained by transforming a given valuation σ on ports of a^t connector according to downward update functions $\gamma.F^d$. More precisely, guards and update functions, $\mathcal{G}_{a^t}, \mathcal{U}_{a^t}$ and $\mathcal{D}_{a^t}$ are defined as follows:

$$\mathcal{U}_{a^t} = \begin{cases} \gamma.F^u, & a^t = \{\gamma\} \\ \mathcal{U}_{a^{t'}} \circ \gamma.F^u, & a^{t'} = a^t \setminus \{\gamma\},\ \gamma \in bottom(a^t). \end{cases}$$

$$\mathcal{G}_{a^t} = \begin{cases} \gamma.G, & a^t = \{\gamma\} \\ \mathcal{G}_{a^{t'}} \wedge \gamma.G, & a^{t'} = a^t \setminus \{\gamma\},\ \gamma \in bottom(a^t). \end{cases}$$

$$\mathcal{D}_{a_t} = \begin{cases} \gamma.F^d, & a^t = \{\gamma\} \\ \gamma.F^d \circ \mathcal{D}_{a^{t'}}, & a^{t'} = a^t \setminus \{\gamma\},\ \gamma \in bottom(a^t). \end{cases}$$

Definition 7 (Component): Composite components are defined from existing components (atomic or composite) by the following grammar:

$$C ::= B \mid (\{C_i\}_{i\in I}, \Gamma, P)$$

where

- B is an atomic component.
- $\{C_i\}_{i\in I}$ is a set of constituent components.
- $P = (\cup_{i\in I} C_i.P) \cup (\cup_{j\in J}\{\gamma_j.p\})$, is the set of ports of the component, that is P contains the ports of the constituent components and the exported ports of the connectors.
- $\Gamma = \{\gamma_j\}_{j\in J}$ is a set of connectors, such that:
  1) $(\Gamma, \rightarrow)$ has no cycle;
  2) $\cup_{j\in J}\gamma_j.P \subseteq P$ (P is defined above);
  3) Each $\gamma \in \Gamma$ uses at most one port of every constituent component, that is, $\forall \gamma \in \Gamma,\ \forall i \in I, |C_i.P \cap \gamma.P| \leq 1$.

Notice that a component is either an atomic component B or a composite component obtained as the composition of a set of constituent components $\{C_i\}_{i\in I}$ by using a set of connectors $\Gamma = \{\gamma_j\}_{j\in J}$. The restriction 3) is needed to prevent simultaneous firing of two or more transitions in the same atomic component, because they may affect the same variables.

For example, consider the component consisting of two composite components shown in Fig. 3(b). Each one of these composite components consists of three identical atomic components described in Fig. 2(a), connected by using the connector described in Fig. 2(b). Each atomic component generates an integer. Then, it synchronizes with all the other atomic components. During synchronization the global maximal value is computed and each atomic component receives the maximum of the values generated.

Definition 8 (Flat Component): Composite component C is flat, iff the set of constituent component $\{C_i\}_{i\in I}$ are atomic components.

The operational semantics of composite components is recursively defined on the component structure. For atomic components, their semantics coincides with the semantics of the underlying behavior. For composition, the semantics is obtained by restricting the parallel behavior according to the interactions.

Definition 9 (Flat Component Semantics): The semantics of component C is a labeled transition system $\mathcal{S}_C = (Q_C, \Sigma_C, \xrightarrow[C]{})$ defined inductively on the structure of C as follows.

1) C is an atomic component, defined by an atomic behavior $B = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$. Then, $\mathcal{S}_C = \mathcal{S}_B$ (see Definition 3).
2) C is a composite component defined as the $(\{C_i\}_{i\in I}, \Gamma, P)$. Let $\mathcal{S}_{C_i} = (Q_{C_i}, \Sigma_{C_i}, \xrightarrow[C_i]{})$ be the semantics of its atomic components. The labeled transition system $\mathcal{S}_C = (Q_C, \Sigma_C, \xrightarrow[C]{})$ is defined as:
   - $Q_C = \bigotimes_{i\in I} Q_{C_i}$ is the of states, the Cartesian product of set of states of subcomponents;
   - $\Sigma_C = \{a^t = tree(\gamma) \mid \gamma \in top(\Gamma)\}$ is the set of labels;
   - $\xrightarrow[C]{} \subseteq Q_C \times \Sigma_C \times Q_C$ is the transition relation, defined by the following rule shown in the equation at the bottom of the page.

$$\frac{\gamma \in top(\Gamma) \quad a^t = tree(\gamma) \quad support(a^t) = \{p_j\}_{j\in J} \quad \forall\, j \in J \quad q_j \xrightarrow[C_j]{p_j(v_j/v'_j)}_j q'_j \quad \{v'_j\}_{j\in J} = \mathcal{D}_{a^t} \circ \mathcal{U}_{a^t}(\{v_j\}_{j\in J}) \quad \mathcal{G}_{a^t}(\{v_j\}_{j\in J}) \quad \forall\, k \notin J.\ \ q_k = q'_k}{q \xrightarrow[C]{a^t} q'.}$$

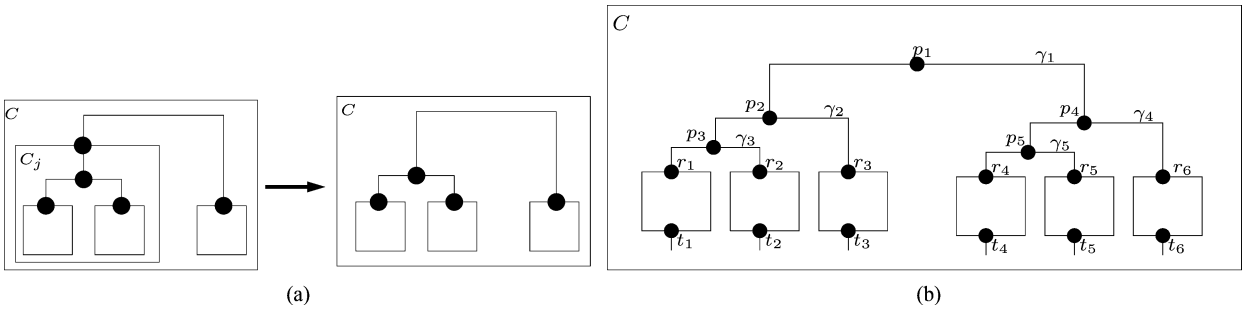


Fig. 4. (a) Component flattening. (b) Component flattening for Example 2.

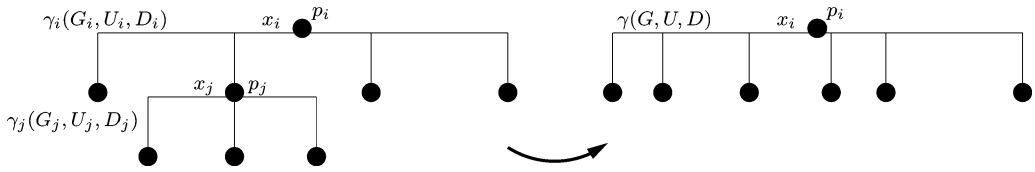


Fig. 5. Connector glueing.

III. Semantics (Transformations)

We define the semantics of composite components by a set of transformations which successively transform them into atomic components. That is, they eliminate component hierarchy and the hierarchical connectors by computing the product behavior.

The transformation from a composite component to an atomic one involves three steps: *Component flattening*, *Connector flattening*, and *Component composition*.

In this section, we describe the three transformations, and we illustrate them in Example 2 shown in Fig. 3(b).

A. Component Flattening

This transformation replaces the hierarchy on components by a set of hierarchically structured connectors applied on atomic components. Consider a composite component C, obtained as the composition of a set of components $\{C_i\}_{i\in I}$. The purpose of this transformation is to replace each non atomic component C_j of C by its description. By successive applications of this transformation, the component C can be modeled as the set of its atomic components and their hierarchically structured connectors [see Fig. 4(a)].

Definition 10 (Component Flattening): Consider a non atomic component $C = (\{C_i\}_{i\in I}, \Gamma, P)$ such that there exists a non atomic component $C_j \in \{C_i\}_{i\in I}$ with $C_j = (\{C_{jk}\}_{k\in K}, \Gamma_j, P_j)$. We define $C[C_j \mapsto \Gamma_j]$ as the component $C = (\{C_i\}_{i\in I} \cup \{C_{jk}\}_{k\in K} \setminus \{C_j\}, \Gamma \cup \Gamma_j, P)$. Component flattening is defined by the following function:

$$\mathcal{F}_c(C) = \begin{cases} C, & \text{if } C \text{ is flat} \\ \mathcal{F}_c(C[C_j \mapsto \Gamma_j]), & \text{if } C \text{ is not flat.} \end{cases}$$

Proposition 1: Component flattening is well-defined, i.e., $\mathcal{F}_c$ is a function which produces a unique result on every input component, and terminates in a finite number of steps.

Proof: Regarding unicity of the result, we can show that, if two constituent components, respectively, C_j and C_k can be replaced inside the composite component C, then the replacement can be done in any order and the final result is the same. That is, formally we have $C[C_j \mapsto \Gamma_j][C_k \mapsto \Gamma_k] = C[C_k \mapsto \Gamma_k][C_j \mapsto \Gamma_j]$. The result follows immediately from the definition and elementary properties of union on sets.

Regarding termination, every transformation step decreases the overall number of composite components by one, so component flattening eventually terminates when all the components are atomic. ■

By applying to Example 2 the transformation $C[C_1 \mapsto \{\gamma_2, \gamma_3\}]$ then $C[C_2 \mapsto \{\gamma_4, \gamma_5\}]$, we obtain the new component in Fig. 4(b).

Finally, notice that this transformation never increases the structural complexity of the transformed component. The transformation does not change the set of atomic components as well as the set of the hierarchical connectors. Hence, it preserves the operational semantics of the original model.

B. Connector Flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j with $\gamma_i \rightarrow \gamma_j$ and produces an equivalent connector.

We show in Fig. 5 the composition of two connectors γ_i and γ_j. It consists in "glueing" them together on the exported port p_j. For the composite connector, the update functions are, respectively, the bottom-up composition of the upward update functions, and the top-down composition of the downward update functions. This implements a general two-phase protocol for executing hierarchical connectors. First, data is synthesized in a bottom up fashion by executing upward update functions, as long as guards are true. Second, data is propagated downwards through downward update functions, from the top to the support set of the connector.

Definition 11 (Connector Glueing): Given connectors $\gamma_i = (p_i[x_i], P_i, \delta_i = (G_i, U_i, D_i))$ and $\gamma_j = (p_j[x_j], P_j, \delta_j = (G_j, U_j, D_j))$ such that $\gamma_i \rightarrow \gamma_j$ we define the composition $\gamma_i[p_j \mapsto \gamma_j]$ as a connector $\gamma = (p, P, \delta)$, where

- $p = p_i$.
- $P = P_j \cup P_i \setminus \{p_j\}$.

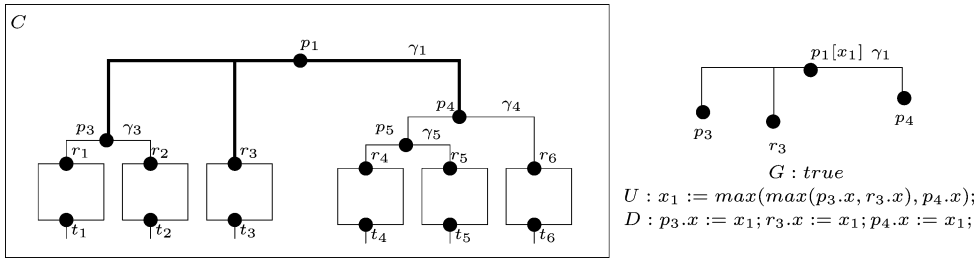


Fig. 6. Connector glueing for Example 2.

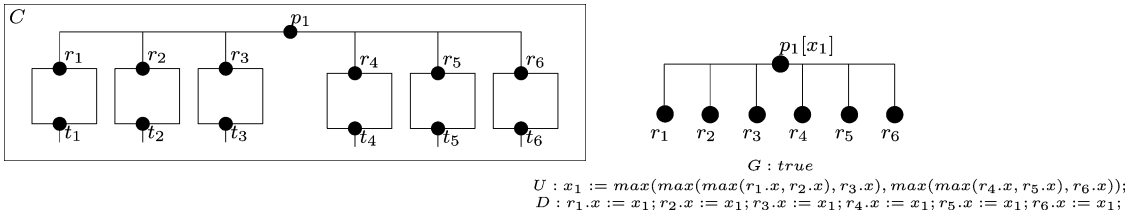


Fig. 7. Connector flattening for Example 2.

- $\delta = (G, U, D)$ is defined as follows:
  - $G = G_j \wedge G_i[F^u_j/x_j]$;
  - $U = x_i := F^u_i[F^u_j/x_j]$;
  - $D = (\cup_{p_k \in P_j} x_k := F^d_{j,x_k}[F^d_{i,x_i}/x_i]) \cup (\cup_{p_k \in P_i \setminus \{p_j\}} x_k := F^d_{i,x_k})$.

Let us introduce some notations. Let $\Gamma = \{\gamma_i = (p_i[x], P_i, \delta_i)\}_{i \in I}$ a set of connectors, and let $P = \{\{p_i\} \cup P_i\}_{i \in I}$ the set of all used ports. We call a port $p_j \in P$ *transient* in Γ if it is both exported by some connector γ_j from Γ and used by another connector γ_i from Γ. Obviously, transient ports can be eliminated through connector glueing.

For a transient port p_j exported by a connector γ_j, we will use the notation $\Gamma[p_j \mapsto \gamma_j]$ to denote the new set of connectors obtained by replacing thoroughly p_j by its exporting connector γ_j, formally: $\Gamma[p_j \mapsto \gamma_j] = \{\gamma | \gamma \in \Gamma, p_j \notin \gamma.ports, \gamma \neq \gamma_j\} \cup \{\gamma[p_j \mapsto \gamma_j] | \gamma \in \Gamma, p_j \in \gamma.ports\}$. That is, all connectors (except γ_j) without p_j in their support set are kept unchanged, while the others are transformed according to Definition 11.

Definition 12 (Connector Flattening): Connector flattening is defined by the following function:

$$\mathcal{F}_\gamma(\Gamma) = \begin{cases} \Gamma, & \text{if } \Gamma \text{ is a set of flat connectors} \\ \mathcal{F}_\gamma(\Gamma[p_j \mapsto \gamma_j]), & \text{if } \Gamma \text{ is not a set of flat connectors, } p_j \text{ is a transient port of } \Gamma. \end{cases}$$

Proposition 2: Connector flattening is well-defined, i.e., $\mathcal{F}_\gamma$ produces a unique result for any set of connectors, and terminates in a finite number of steps.

Proof: Regarding unicity of the result, if p_j and p_k are two transient ports of Γ defined, respectively, by connectors γ_j and γ_k, then flattening in any order gives the same result, formally $\Gamma[p_j \mapsto \gamma_j][p_k \mapsto \gamma_k] = \Gamma[p_k \mapsto \gamma_k][p_j \mapsto \gamma_j]$.

To show this result it is sufficient to show that any connector γ of Γ, different from γ_j and γ_k gets transformed in the same way, independently of the order of application of the two transformations. This can be shown, case by case, depending on the occurrence of ports p_j and p_k in the supports of γ, γ_j, and γ_k following Definition 11.

Regarding termination, flattening of connectors is applicable as long as there are transient ports. Moreover, it can be shown that, every flattening step reduces the number of transient ports by one—the one that is replaced by its definition. Hence, flattening eventually terminates when no more transient ports exist, that is, Γ is a set of flat connectors. ■

By application of the transformation $\gamma_1[p_2 \mapsto \gamma_2]$ to Example 2 in Fig. 4(b), we obtain the new composite component given in Fig. 6. If we apply successively, $\gamma_1[p_3 \mapsto \gamma_3], \gamma_1[p_4 \mapsto \gamma_4], \gamma_1[p_5 \mapsto \gamma_5]$, we obtain the new composite component given in Fig. 7.

In a similar way to component flattening, this second transformation does not increase the structural complexity of the transformed components. The set of atomic components is preserved as such, whereas, the overall set of connectors is decreasing. However, the remaining connectors have an increased computational complexity, because they integrate the guards and the data transfer of the eliminated ones. The operational semantics is also preserved. The effect of the eliminated connectors is "in-lined" in the remaining ones according to Definition 6.

C. Component Composition

We present the third transformation which allows to obtain a single atomic component from a set of atomic components and a set of flat connectors. This transformation defines the composition of behaviors. Intuitively, as shown in Fig. 8(a), the composition operation consists in "glueing" together transitions from atomic components that are synchronized through the interaction of some connector (interaction p_1p_2 for this example). Guards of synchronized transitions are obtained by conjuncting individual guards and the guard of the connector. Similarly, actions of synchronized transitions are obtained as the sequential composition of the upward update function followed by the downward update function of the connector, followed by the actions of the components in an arbitrary order.

Definition 13 (Component Composition): Consider a component $C = (\{B_i\}_{i \in I}, \Gamma, P)$ such that $\forall i \in I$ B_i

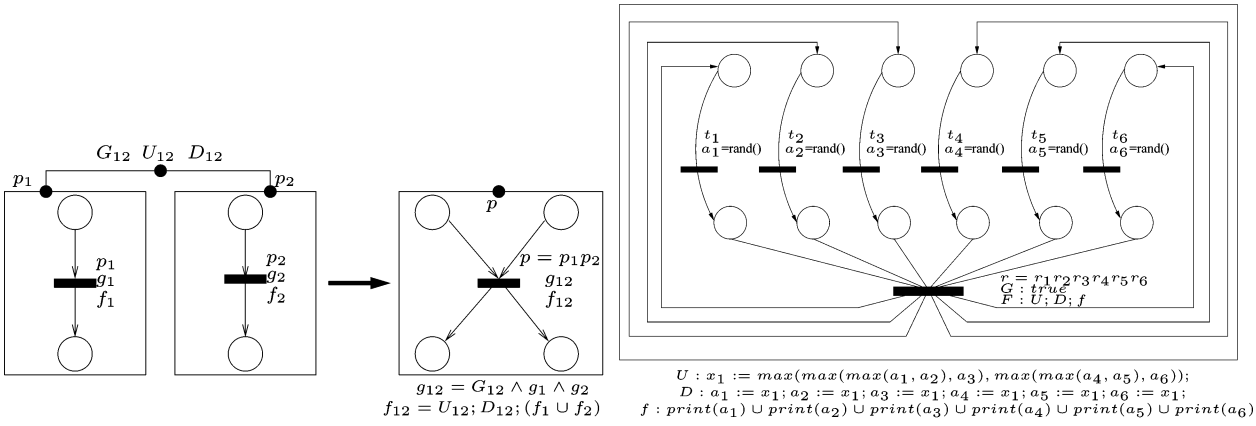


Fig. 8. (a) Component composition. (b) Component composition for Example 2.

is an atomic component and Γ is a set of flat connectors. We define the composition $\Gamma(\{B_i\}_{i\in I})$ as component $B = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ defined as follows:

- the set of ports $P = \cup_{\gamma\in\Gamma}\{\gamma.p\}$;
- the set of control locations $L = \cup_{i\in I} B_i.L$;
- the set of variables $X = (\cup_{i\in I} B_i.X) \cup (\cup_{\gamma\in\Gamma}\gamma.p.x)$;
- each transition in T corresponds to a set of interacting transitions $\{\tau_1, \ldots, \tau_k\} \subseteq \cup_{i\in I} T_i$ such that $\cup_{i=1}^k \tau_i.p = \gamma.P$ $(\gamma \in \Gamma)$. We define the transition $\tau = (l, \gamma.p, l')$, where
  - $l = ^\bullet \tau_1 \cup \cdots \cup ^\bullet \tau_k$;
  - $l' = \tau_1^\bullet \cup \cdots \cup \tau_k^\bullet$;
  - the guard $g_\tau = \wedge_{i=1}^k g_{\tau_i} \wedge \gamma.\delta.G$;
  - the action $X := f_\tau(X)$ with $f_\tau = \gamma.\delta.U;\ \gamma.\delta.D;\ (\cup_{i=1}^k f_{\tau_i})$.

Fig. 8(b) shows the Petri net obtained by composition of the atomic components of Fig. 7 through the interaction $r_1r_2r_3r_4r_5r_6$.

In contrast to previous transformations, component composition may lead to an exponential blowup of the number of transitions in the resulting Petri net. This situation may happen if the same interaction can be realized by combining different transitions from each one of the involved components. For instance, the interaction p_1p_2 can give rise to four transitions in the resulting Petri net if there are two transitions labeled by p_1 and p_2 in the synchronizing components. Nevertheless, in practice, exponential explosion seldom occurs, as in atomic components each port labels at most one transition (as in the examples shown hereafter). In this case, the resulting Petri net has as many transitions as connectors in Γ.

IV. Experimental Results

A. The BIP2BIP Tool

These transformations have been implemented in the BIP2BIP tool, which is currently integrated in the BIP toolset [8], as shown in Fig. 9(a).

The frontend of the BIP toolset is a *parser* that generates a model from a system described in the BIP language. The BIP language allows the description of hierarchically structured components as described in the previous sections. Functions and data are written in C. The language supports description of atomic components as extended Petri nets. It also allows the description of composite components by using connectors.

From the generated model, the code-generator generates C++ code, executable on a dedicated middleware, the *BIP Engine*. The BIP Engine can orchestrate execution of the generated code as well as enumerative state-space exploration. The generated state graphs can be analyzed by using model-checking tools. The BIP2BIP tool is written in *Java* ($\sim$ 4000 loc). It allows transformation of parsed models. It contains the following modules implementing the presented transformations.

- Component flattening: This module transforms a composite component into an equivalent one consisting only of atomic components of the initial model and a set of connectors.
- Connector flattening: This module transforms an hierarchically structured connector into an equivalent flat one.
- Component composition: This module transforms a set of atomic components and a set of flat connectors into an equivalent atomic component.

By exhaustive application of these transformations, an atomic component can be obtained. From the latter, the *code-generator* can generate standalone C++ code, which can be run directly without the Engine. In particular, all the remaining nondeterminism in the final atomic component is eliminated at code generation by applying an implicit priority between transitions.

It should be noted that the transformations also can be applied independently, to obtain models that respond to a particular user needs. For example, one may decide to eliminate only partially the hierarchy of components, or to compose only some components.

The performance of BIP2BIP is quite satisfactory. For example, when applied to an artificially complex BIP model, consisting of 256 atomic components, composed by using 509 connectors with 7 levels of hierarchy, it takes less than 15 s to generate the corresponding C++ program.

B. Examples of Transformation

For two examples, we compare the execution times of BIP programs before and after flattening. These examples show that it is possible to generate efficient standalone C++ code from component-based descriptions in BIP.

1) MPEG Video Encoder: In the framework of an industrial project, we have componentized in BIP an MPEG4 encoder written in C by an industrial partner. The aim of this work was

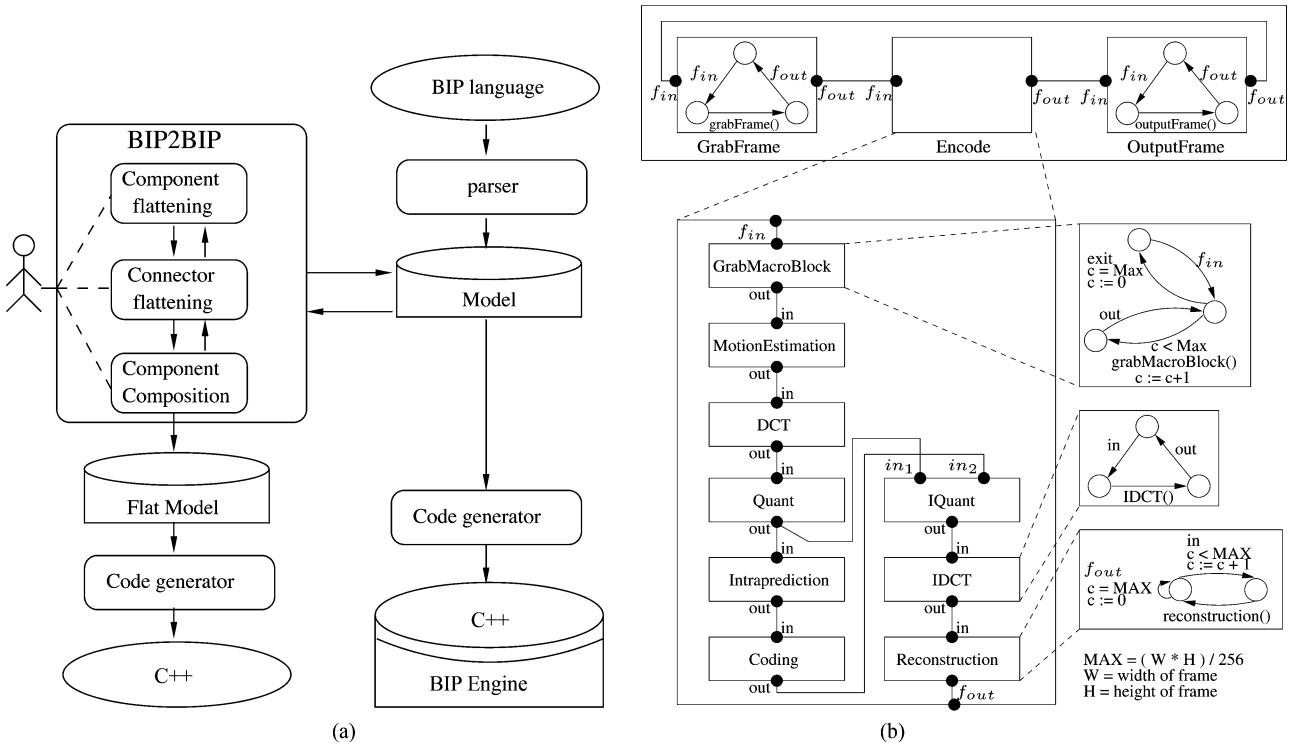
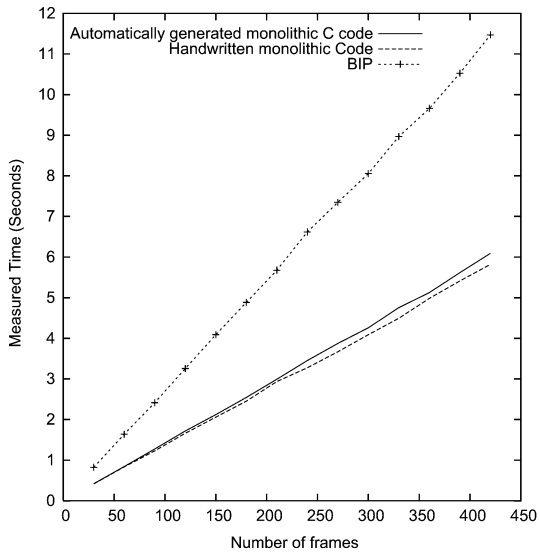


Fig. 9. (a) BIP toolset: General architecture. (b) MPEG4 encoder.



	Handwritten	BIP	$C^{(1)}$	$C^{(2)}$
loc	600	350	1800	800

(b)

Fig. 10. (a) Execution time for the MPEG4 encoder. (b) Code size in loc for MPEG4 encoder.

to evaluate gains in scheduling and quality control of the componentized program. The results were quite positive regarding quality control [11] but the componentized program was almost two times slower than the handwritten C program. We have used BIP2BIP to generate automatically standalone C++ code from the BIP program as explained below [see Fig. 9(b)].

The BIP program consists of 11 atomic components, and 14 connectors. It uses the data and the functions of the initial handwritten C program. It is composed of two atomic components and one composite component. The atomic component **GrabFrame** gets a frame and produces macroblocks (each frame is split into N macroblocks of 256 pixels). The atomic component **OutputFrame** produces an encoded frame. The composite component **Encode** consists of nine atomic components and the corresponding connectors. It encodes macroblocks produced by the component **GrabFrame**.

Fig. 10(a) shows execution times for the initial handwritten C code, for the BIP program and the corresponding standalone C++ code generated automatically by using the presented technique. Notice that the automatically generated C++ code and the handwritten C code have almost the same execution times. The advantages from the componentization of the handwritten code are multiple. The BIP program has been rescheduled as shown in [11] so as to meet given timing requirements.

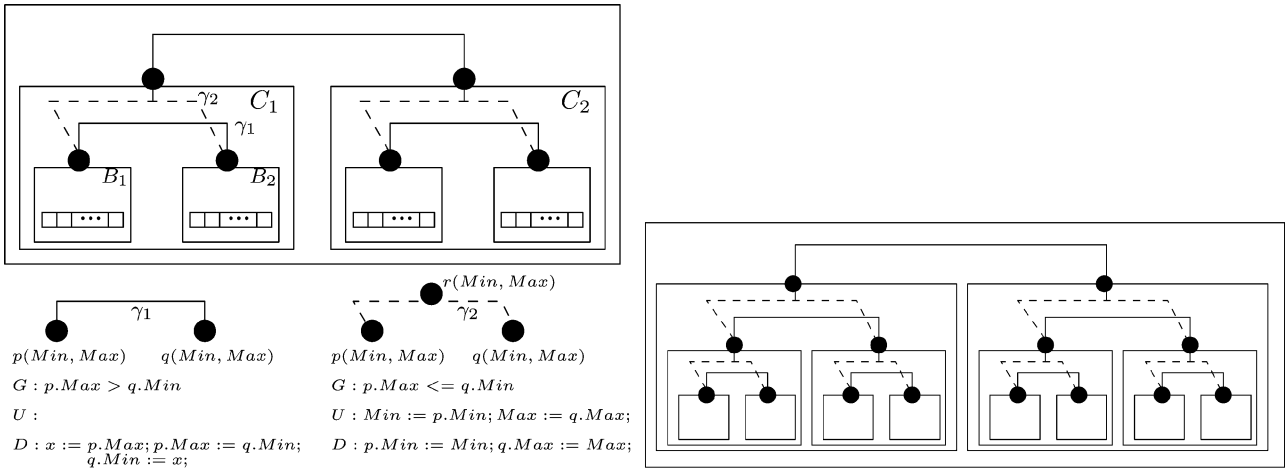
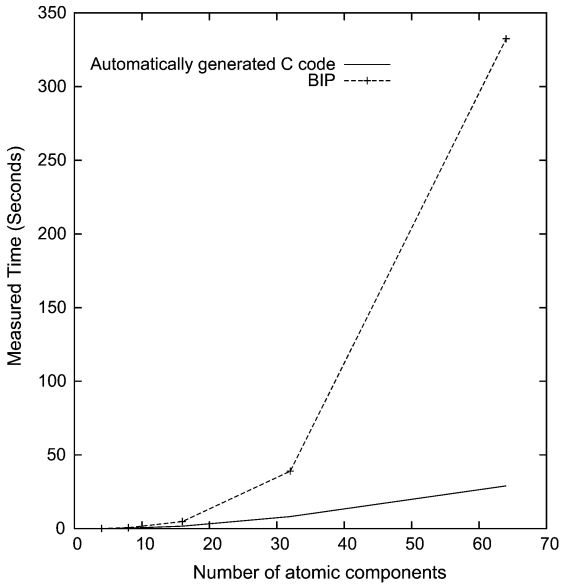


Fig. 11. (a) Concurrent sorting $n = 2$. (b) Concurrent sorting $n = 4$.



n		BIP	$C^{(1)}$	$C^{(2)}$
2	loc	112	360	400
3	loc	120	400	620
4	loc	128	440	1100
5	loc	136	480	1850
6	loc	144	520	2850

(a) (b)

Fig. 12. (a) Execution time for concurrent sorting. (b) Code size in loc for concurrent sorting.

Fig. 10(b) gives the size of the handwritten C code, the BIP model, as well of the generated C++ code from the BIP model $C^{(1)}$ and the generated C++ code from the BIP model after flattening $C^{(2)}$. The time taken by the BIP2BIP tool to generate automatically $C^{(2)}$ is less than 1 s.

2) Concurrent Sorting: This example is inspired from a network sorting algorithm [1]. We consider 2^n atomic components, each of them containing an array of N values. We want to sort all the values, so that the elements of the first component are smaller than those of the second component and so on. We solve the problem by using incremental hierarchical composition of components with particular connectors.

In Fig. 11(a), we give a model for sorting the elements of 4 atomic components. The components C_1 and C_2 are identical. The pair (B_1, B_2) is composed by using two connectors γ_1 and γ_2 to form the composite component C_1. Each atomic component computes the minimum and the maximum of the values in its array. These values are then exported on port p. The connector γ_1 is used to compare the maximum value of B_1 with the minimum value of B_2, and to permute them if the maximum is bigger than the minimum value.

When the maximum value of B_1 is smaller than the minimum value of B_2, that is the components are correctly sorted, then the second connector γ_2 is triggered. It is used to export the minimum value of B_1 and the maximum value of B_2 to the upper level. At this level, the same principle is applied to sort the values of the composite components C_1 and C_2. This pattern can be repeated to obtain arbitrary higher hierarchies [see Fig. 11(b)].

Fig. 12(a) shows the execution times for the hierarchically structured BIP program and for the corresponding standalone C++ code generated automatically by using the presented technique. Notice the exponentially increasing difference between the execution time of the component-based BIP program and the corresponding C++ code. In particular, component flattening and connector flattening do not provide much better performance, because the hierarchical structure is actually exploited by the BIP engine to compute enabled interactions

in an efficient manner. However, these transformations are mandatory for applying the static composition. Notice that the overhead is due to many reasons when using the BIP engine. First, each atomic components sends to the engine its current state and the list of enabled ports. Second, the engine enumerates on the list of interactions in the model, identifies *all* enabled ones based on the current state of the atomic components, then among them it selects one for execution and, finally, notifies atoms to take the corresponding transition. This overhead is partially eliminated in the standalone C++ code generated automatically. Indeed, the call function between components and the engine is omitted. The time needed to select an enabled interaction is drastically reduced. Moreover, control and code optimization such as guard combination, removal of unnecessary assignments, etc., are applied.

Fig. 12(b) shows the size in lines of code of the BIP model, as well of the generated C++ generated from the BIP model C++ and the generated C++ code from the BIP model after flattening $C^{(2)}$, for 4, 8, 16, 3, 2 and 64 atomic components. The size of the BIP model changes only linearly with n. However, we notice that for this example, the size of the generated C++ code from the BIP model is much smaller than the generated C++ code from the BIP model after flattening. This is due to the use of component types and component types instantiation. In particular, for this example, the initial BIP model contains just one component type instantiated, respectively, 4, 8, 16, 32, 64 times for $n = 2, 3, 4, 5, 6$. However, the BIP model after flattening, contains one component type with one instance each. The size of the generated code is directly dependent on the number of component types and not on the number of component types instance.

V. Conclusion

The paper shows that it is possible to reconcile component-based incremental design and efficient code generation by applying a paradigm based on the combined use of: 1) a high-level modeling notation based on well-defined operational semantics and supporting powerful mechanisms for expressing structured coordination between components and 2) semantics-preserving source-to-source transformations that progressively transform architectural constraints between components into internal computation of product components.

BIP has already successfully been used for the componentization of non trivial systems such as the controller of the DALA robot [5]. This allowed building component-based models for which enhanced analysis and verification is possible by using tools such as D-Finder [7] for compositional verification. The use of the BIP2BIP tool allows to reduce overheads in execution time by reducing modularity introduced by the designer when it is not necessary at implementation level.

This paradigm opens the way to the synthesis of efficient monolithic software which is correct-by-construction by using the design methodology supported by BIP. The methodology is currently under study, and involves the following steps.

1) The system (software) to be designed is decomposed into components. The decomposition can be represented as a tree which shows how the system can be obtained as the incremental composition of components. Its root is the system and its leaves correspond to atomic components.
2) Description of the behavior of the atomic components.
3) Description of composite components as the composition of atomic components by using only connectors and priorities.

This is possible because BIP is expressive enough for describing any kind of coordination by using only architectural constraints [10].

Along steps 2) and 3) it is possible by using the D-Finder tool, to generate and/or check invariants of the components and validate their properties. The methodology provides sufficient conditions for preserving the already established properties of the subsystems along the construction.

The BIP2BIP tool is an essential feature of the BIP toolset. Further developments will focus on source-to-source transformations for BIP programs with priorities by following a similar flattening principle. In fact, priority rules can be compiled in the form of restrictions of the guards of components. We plan to use BIP2BIP, for optimizing distributed implementations [3], in particular, to generate monolithic C code for subsystems implemented on the same site.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in c log n parallel steps," *Combinatorica*, vol. 3, no. 1, pp. 1–19, 1983.
[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, 2003.
[3] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis, "Distributed semantics and implementation for systems with interaction and priority," in *FORTE*, K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, Eds. New York: Springer, 2008, vol. 5048, Lecture Notes in Computer Science, pp. 116–133.
[4] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proc. 4th IEEE Int. Conf. Softw. Eng. Formal Methods (SEFM'06)*, Sep. 2006, pp. 3–12.
[5] A. Basu, M. Gallien, C. Lesire, T.-H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis, "Incremental component-based construction and verification of a robotic system," in *ECAI*, M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, Eds. Amsterdam, The Netherlands: IOS Press, 2008, vol. 178, Frontiers in Artificial Intelligence and Applications, pp. 631–635.
[6] R. V. Bennett, A. C. Murray, B. Franke, and N. P. Topham, "Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems," in *LCTES*, S. Pande and Z. Li, Eds. New York: ACM, 2007, pp. 83–92.
[7] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *ATVA*, S. D. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds. New York: Springer, 2008, vol. 5311, Lecture Notes in Computer Science, pp. 64–79.
[8] BIP [Online]. Available: http://www-verimag.imag.fr/~async/bip.php
[9] S. Bliudze and J. Sifakis, "Causal semantics for the algebra of connectors," in *FMCO*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds. New York: Springer, 2007, vol. 5382, Lecture Notes in Computer Science, pp. 179–199.
[10] S. Bliudze and J. Sifakis, "A notion of glue expressiveness for component-based systems," in *CONCUR*, F. van Breugel and M. Chechik, Eds. New York: Springer, 2008, vol. 5201, Lecture Notes in Computer Science, pp. 508–522.
[11] J. Combaz, J.-C. Fernandez, J. Sifakis, and L. Strus, "Symbolic quality control for multimedia applications," *Real-Time Systems*, vol. 40, no. 1, pp. 1–43, 2008.
[12] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe, "Quasi-static scheduling of independent tasksfor reactive systems," in *ICATPN*, J. Esparza and C. Lakos, Eds. New York: Springer, 2002, vol. 2360, Lecture Notes in Computer Science, pp. 80–100.
[13] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe, "Quasi-static scheduling of independent tasks for reactive systems," *IEEE Trans. CAD of Integr. Circuits and Syst.*, vol. 24, no. 10, pp. 1492–1514, 2005.

[14] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the ptolemy approach," *Proc. IEEE*, vol. 91, pp. 127–144, 2003.
[15] B. Hindman and D. Grossman, "Atomicity via source-to-source translation," in *Memory System Performance and Correctness*, A. L. Hosking and A.-R. Adl-Tabatabai, Eds. New York: ACM, 2006, pp. 82–91.
[16] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty, "Performance debugging of esterel specifications," in *CODES+ISSS '08: Proc. 6th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, New York, pp. 173–178.
[17] D. B. Loveman, "Program improvement by source-to-source transformation," *J. ACM*, vol. 24, no. 1, pp. 121–145, 1977.

Marius Bozga graduated from the Faculty of Mathematics and Computer Science, Babes-Bolyai University of Cluj-Napoca, Romania, and received the Ph.D. degree in computer science from the University of Grenoble, Grenoble, France.

He is currently a CNRS Research Engineer and member of the VERIMAG Laboratory. His research interests are focused on component-based design for distributed real-time systems and include formal models for components, model-based design and implementation, automatic validation methods and tools.

Mohamad Jaber received the B.S. degree in computer science from Lebanese University, Beirut, in 2006, the Ms.C. degree in computer science from the University of Grenoble, Grenoble, France, in 2007. He is currently a third year Ph.D. student at VERIMAG Laboratory, Grenoble, France, where he is working in the domain of component-based constructions of distributed systems.

Joseph Sifakis studied electrical engineering at the Technical University of Athens, Athens, Greece, and Computer Science at the University of Grenoble, Grenoble, France.

He is a CNRS researcher and the founder of the Verimag Laboratory, Grenoble, France. He is recognized for his pioneering work on both theoretical and practical aspects of concurrent systems specification and verification. He contributed to emergence of the area of model-checking, currently the most widely used method for the verification of industrial applications. His current research activities include component-based design, modeling, and analysis of real-time systems with focus on correct-by-construction techniques.

Dr. Sifakis received with E. Clarke and A. Emerson the Turing Award in 2007 for their contribution to model checking.