Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

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

Marius Bozga, Mohamad Jaber, Joseph Sifakis

Verimag Research Report n^o TR-2009-3

March 25, 2009

Reports are downloadable at the following address
http://www-verimag.imag.fr

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

Marius Bozga, Mohamad Jaber, Joseph Sifakis

March 25, 2009

Abstract

BIP (Behavior, Interaction, Priorities) 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 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 the paper.

Keywords: Source-to-source transformation, component-based systems, optimization for performance, correctness preservation

How to cite this report:

@techreport { ,
title = { Source-to-Source Architecture Transformation for Performance Optimization in BIP},
authors = { Marius Bozga, Mohamad Jaber, Joseph Sifakis},
institution = { Verimag Research Report },
number = {TR-2009-3},
year = { 2009},
note = { }
}

1 Introduction

Component-based systems are desirable because they allow reuse of sub-systems 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 [13].

Source-to-source transformations have been considered as a powerful means for optimizing programs [12, 5]. 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.

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 BIP (Behavior, Interaction, Priority) language [3, 8] where an architecture is characterized as a hierarchically structured set of components obtained by composition from a set *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.

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 Figure 1).

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

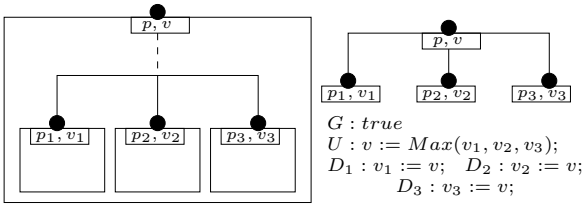


Figure 1: Example 1

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; 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 the 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. *Component 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 non trivial examples the benefits of this approach.

The paper is structured as follows. In section 2 we define the syntax for the description of structured components in BIP. In section 3, we define the semantics by successive application of the three source-to-source transformations. In section 4, we provide benchmarks for two examples: a MPEG encoder and a concurrent sorting program. In Section 5, we discuss other applications and future developments.

2 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 labelled relation on the 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 $\tau \in T$ respectively g_τ is a guard, an action $X := f_\tau(X)$.*

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

Figure 2 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 labelled t_1 is possible. 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 labelled r_1 can occur (the guard of the transition is true by default), the variable a is eventually modified and the value of a is printed.

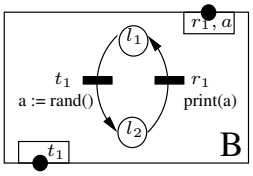


Figure 2: An example of an atomic component in BIP

Definition 3 (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,*

- $\delta = (G, U, D)$ *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,* $\cup_{p_i} x_i := F^d_{x_i}(x)$*.*

Figure 3 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.

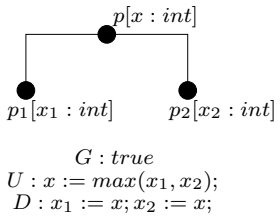


Figure 3: An example of a connector in BIP

For a set of connectors $\Gamma = \{\gamma_j\}_{j\in J}$, we define the domination 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 Figure 4).

Γ is a set of *flat connectors*, iff no connector dominates another, that is, $\forall \gamma_i, \gamma_j \in \Gamma$ we have $\gamma_i \not\rightarrow \gamma_j$.

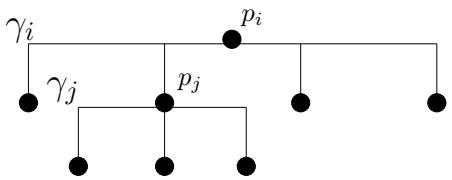


Figure 4: γ_i dominates γ_j

Definition 4 (component) *Composite components are defined from existing components (atomic or composite) by the following grammar:*

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

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,*
  - $(\Gamma, \rightarrow)$ *has no cycle,*

- $\cup_{j\in J}\gamma_j.P \subseteq P$ *(P is defined above),*
- *Each* $\gamma \in \Gamma$ *uses at most one port of every constituent component, that is,* $\forall\gamma \in \Gamma, \forall i \in I$, $|C_i.P \cap \gamma.P| \leq 1$.

That is, 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 $\Gamma = \{\gamma_j\}_{j\in J}$.

For example, consider the BIP component composed of two composite components shown in Figure 5. Each constituent component consists of three identical atomic components described in Figure 2, connected by using the connector described in Figure 3. 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.

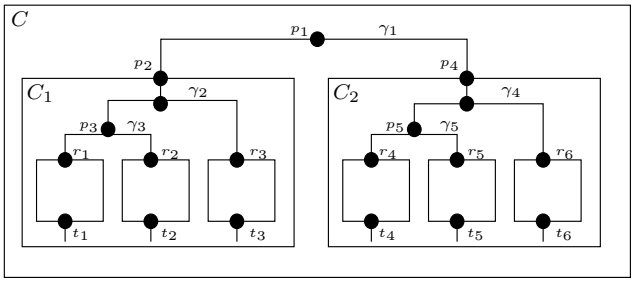


Figure 5: Example 2

3 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 a atomic one is fully automated and implemented through three steps:

1. *Component 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 a set of atomic components to get one atomic component.

In this section, we describe the three transformations, and we illustrate them on Example 2 shown in Figure 5.

3.1 Component flattening

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 modelled as the set of its atomic components and their hierarchically structured connectors (see Figure 6).

Definition 5 (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* $C = (\{C_i\}_{i\in I} \cup \{C_{jk}\}_{k\in K} \setminus \{C_j\}, \Gamma \cup \Gamma_j, P)$.

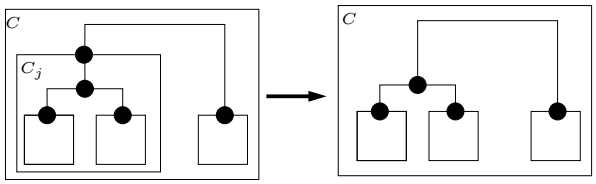


Figure 6: Component flattening

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 Figure 7.

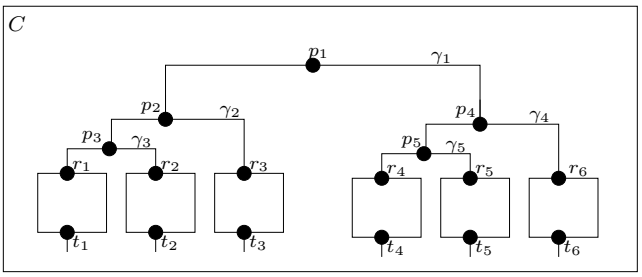


Figure 7: Component flattening for Example 2

Proposition 1 *Component flattening is confluent and terminates.*

Proof 1 *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.

3.2 Connector flattening

This transformation flattens hierarchical connectors. It takes two connectors γ_i and γ_j such that $\gamma_i \rightarrow \gamma_j$ and produces an equivalent connector.
We show in Figure 8 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 6 (Connector flattening) *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

- $p = p_i$,
- $P = P_j \cup P_i \setminus \{p_j\}$,
- $\delta = (G, U, D)$ *is defined as follows:*

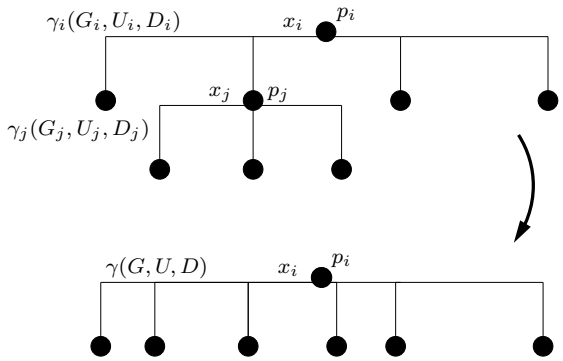


Figure 8: Connector flattening

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

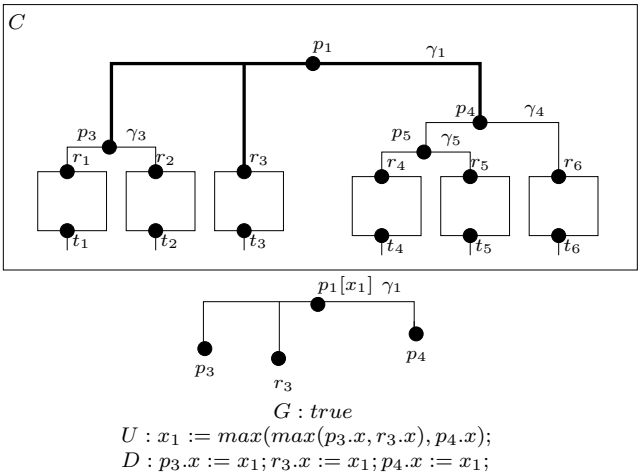


Figure 9: Connector flattening for Example 2

By application of the transformation $\gamma_1[p_2 \mapsto \gamma_2]$ to Example 2 in Figure 7, we obtain the new composite component presented in Figure 9. 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 presented in Figure 10.

Proposition 2 *Connector flattening is confluent and terminates.*

Proof 2 *Let us first introduce some notations.*

Let $C = (\{C_i\}_{i \in I}, \Gamma, P)$ be a composite component. 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 flattening.*

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:

$\Gamma[p_j \mapsto \gamma_j] = \{\gamma \mid \gamma \in \Gamma, p_j \notin \gamma.ports, \gamma \neq \gamma_j\} \cup \{\gamma[p_j \mapsto \gamma_j] \mid \gamma \in \Gamma, p_j \in \gamma.ports\}$.

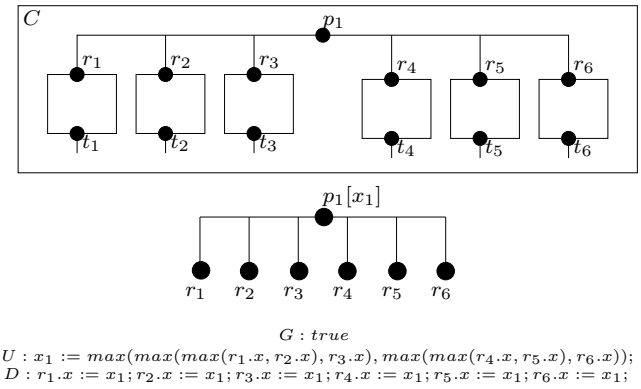


Figure 10: Connector flattening for Example 2

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 6.

Now, it can be shown that, if p_j and p_k are two transient ports of Γ defined respectively by connectors γ_j and γ_k, then flattening can be done in any order, formally

$\Gamma[p_j \mapsto \gamma_j][p_k \mapsto \gamma_k] = \Gamma[p_k \mapsto \gamma_k][p_j \mapsto \gamma_j]$.

The equality amounts 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 is easily shown, case by case, depending on the occurrence of ports p_j and p_k on the support of γ, γ_j and γ_k following the definition 6.

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, the flattening eventually terminates when no more transient ports exist, that is, Γ is a set of flat connectors.

The following proposition ensures that our semantics is well-defined.

Proposition 3 *The combination of component flattening and connector flattening is confluent and terminates.*

3.2.1 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 Figure 11, 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 7 (Component composition) *Consider a component $C = (\{B_i\}_{i\in I}, \Gamma, P)$ such that $\forall i \in I$ B_i 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 places $L = \cup_{i\in I}B_i.L$,*
- *the set of variables $X = (\cup_{i\in I}B_i.X) \cup (\cup_{\gamma\in\Gamma}\gamma.p.x)$,*

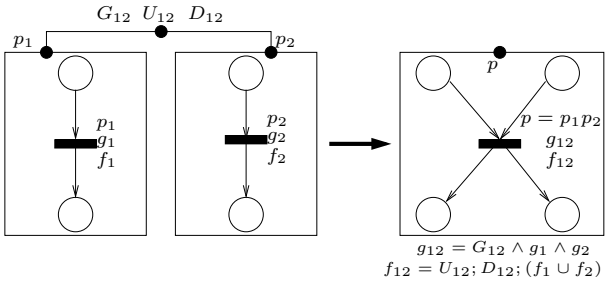


Figure 11: Component composition

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

Figure 12 shows the Petri net obtained by composition of the atomic components of Figure 10 through the interaction $r_1 r_2 r_3 r_4 r_5 r_6$.

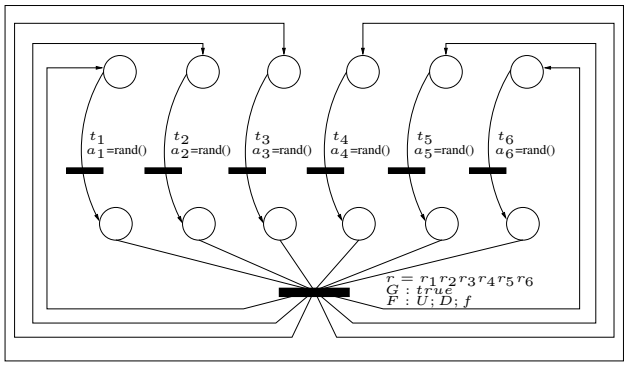


Figure 12: Component composition for Example 2

4 Experimental Results

4.1 The BIP2BIP tool

These transformations have been implemented in the BIP2BIP tool, which is currently integrated in the BIP toolset [7] as shown in Figure 13.

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. The 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 perform execution and enumerative state-space exploration. The

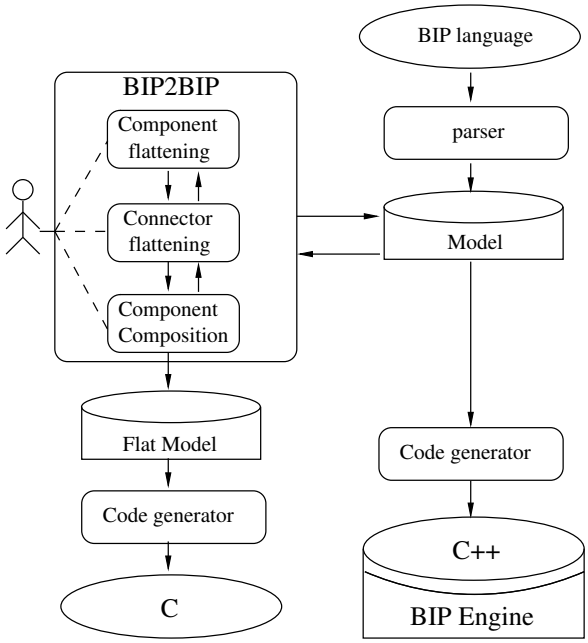


Figure 13: BIP toolset: General architecture

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 replaces a composite component by its description. By successive applications of this module, we obtain an equivalent model consisting only of atomic components of the initial model and a set of connectors.
- Connector flattening : this module transforms an hierarchically structured connector to an equivalent flat one. By successive applications of this module, we obtain a new model with flat connectors.
- Component composition : this module transforms a set of atomic components and a set of flat connectors into an equivalent atomic component. By successive applications of this module, we obtain a new model consisting of a single 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 non-determinism 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 seconds to generate the corresponding C program.

4.2 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 C code from component-based descriptions in BIP.

4.2.1 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 to evaluate gains in scheduling and quality control of the componentized program. The results were quite positive regarding quality control [11, 10] but the componentized program was almost two times slower than the handwritten C program. We have used BIP2BIP to generate automatically C code from the BIP program as explained below (see Figure 14).

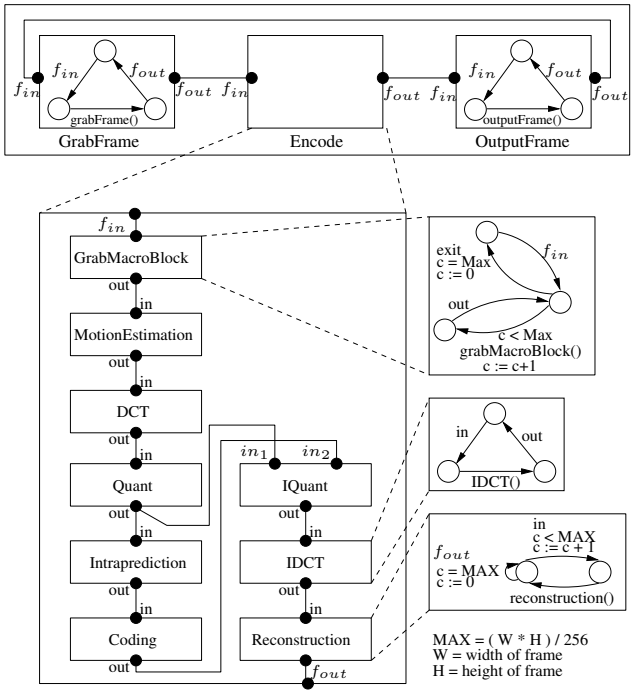


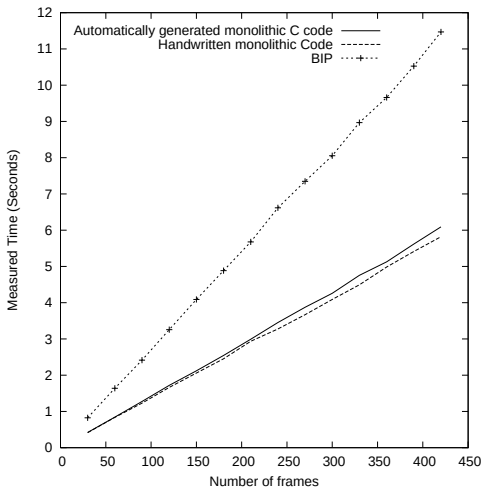
Figure 14: MPEG4 encoder

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 9 atomic components and the corresponding connectors. It encodes macroblocks produced by the component **GrabFrame**.

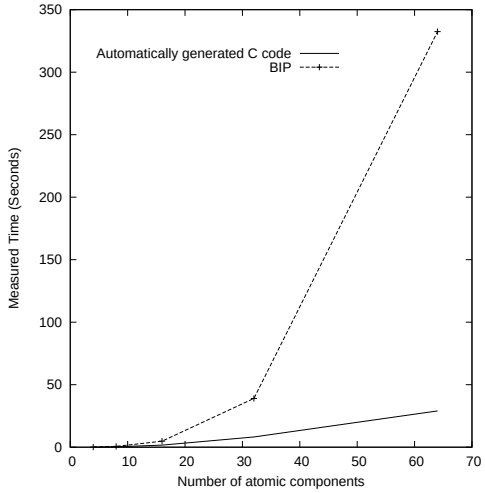
Figure 15 shows the execution times for the initial handwritten C code, for the BIP program and the corresponding C code generated automatically. Notice that the automatically generated C code and the handwritten C code have almost the same execution times. The advantages from the componentization of the handwritten code are multiple. The BIP program has been rescheduled as shown in [11, 10] so as to meet given timing requirements.
Table 16 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 1sec.

4.2.2 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.



(a)



(b)

Figure 15: Execution time for the MPEG4 Encoder (a) and Execution time for Concurrent Sorting (b)

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

(a)

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

(b)

Figure 16: Code size in loc for MPEG4 Encoder (a) and Code size in loc for Concurrent Sorting (b)

In Figure 17, 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 the 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 the 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 Figure 18).

Figure 15 shows the execution times for the hierarchically structured BIP program and for the corresponding C code generated automatically. Notice, the exponentially increasing difference between the execution time of the component-based BIP program and the corresponding C code. Table 16 shows the size in lines of code of the BIP program, the component-based C++ corresponding program and the C code for 4, 8, 16, 32 and 64 atomic components. The size of the BIP model changes only linearly with n.

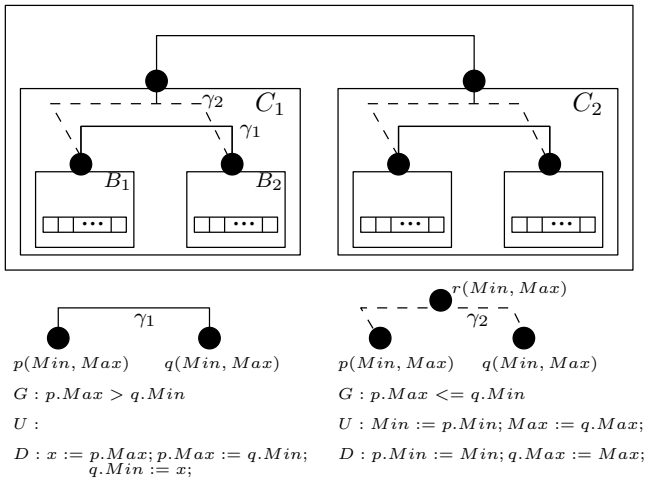


Figure 17: Concurrent Sorting

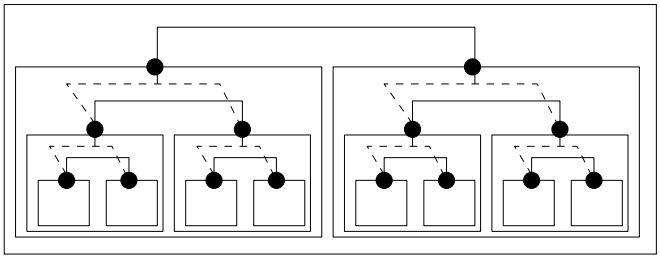


Figure 18: Concurrent Sorting

5 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 modelling notation based on well-defined operational semantics and supporting powerful mechanisms for expressing structured coordination between components; 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 [4]. This allowed building component-based models for which enhanced analysis and verification is possible by using tools such as D-Finder [6] 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 expressing any kind of coordination by using only architectural constraints [9].

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 sub-systems 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 [2], 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*, 3(1):1–19, 1983. 4.2.2

[2] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *FORTE*, pages 116–133, 2008. 5

[3] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In 4^{th} *IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM06)*, pages 3–12, September 2006. Invited talk. 1

[4] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. Incremental component-based construction and verification of a robotic system. In *ECAI*, pages 631–635, 2008. 5

[5] Richard Vincent Bennett, Alastair Colin Murray, Björn Franke, and Nigel P. Topham. Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. In *LCTES*, pages 83–92, 2007. 1

[6] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *ATVA*, pages 64–79, 2008. 5

[7] BIP. http://www-verimag.imag.fr/∼async/bip.php. 4.1

[8] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. In *FMCO*, pages 179–199, 2007. 1

[9] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, pages 508–522, 2008. 5

[10] Jacques Combaz, Jean-Claude Fernandez, Thierry Lepley, and Joseph Sifakis. Fine grain qos control for multimedia application software. In *DATE*, pages 1038–1043, 2005. 4.2.1, 4.2.1

[11] Jacques Combaz, Jean-Claude Fernandez, Joseph Sifakis, and Loïc Strus. Symbolic quality control for multimedia applications. *Real-Time Systems*, 40(1):1–43, 2008. 4.2.1, 4.2.1

[12] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Memory System Performance and Correctness*, pages 82–91, 2006. 1

[13] David Loveman. Program improvement by source-to-source transformation. In *JACM, 24(1)*, pages 121–145, 1977. 1