Modeling Heterogeneous Real-Time Components in BIP

Ananda Basu, Marius Bozga, Joseph Sifakis

Modeling Heterogeneous Real-time Components in BIP

Ananda Basu, Marius Bozga and Joseph Sifakis
Verimag, 38610 Gieres, France
{basu, bozga, sifakis}@imag.fr

Abstract

We present a methodology for modeling heterogeneous real-time components. Components are obtained as the superposition of three layers : Behavior, specified as a set of transitions; Interactions between transitions of the behavior; Priorities, used to choose amongst possible interactions. A parameterized binary composition operator is used to compose components layer by layer.

We present the BIP language for the description and composition of layered components as well as associated tools for executing and analyzing components on a dedicated platform. The language provides a powerful mechanism for structuring interactions involving rendezvous and broadcast. We show that synchronous and timed systems are particular classes of components. Finally, we provide examples and compare the BIP framework to existing ones for heterogeneous component-based modeling.

1. Introduction

A central idea in systems engineering is that complex systems are built by assembling components. System designers deal with a large variety of components, each having different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system. A central problem is the meaningful composition of heterogeneous components to ensure their correct inter-operation [12].

One fundamental source of heterogeneity is the composition of subsystems with different execution and interaction semantics. At one extreme of the semantic spectrum are fully synchronized components, which proceed in lockstep with a global clock and interact in atomic transactions. Such a tight coupling of components is the standard model for most synthesizable hardware and for synchronous real-time software. At the other extreme are completely asynchronous components, which proceed at independent speeds and interact non-atomically. Such a loose coupling of components is the standard model for most multithreaded software. Between the two extremes, a variety of intermediate and hybrid models exist (e.g., globally-asynchronous locally-synchronous models).

Another fundamental source of heterogeneity is the use of models that represent a system at varying degrees of detail and are related to each other in an abstraction (or equivalently, refinement) hierarchy. A key abstraction in system design is the one relating application software to its implementation on a given platform. Application software is largely untimed, in the sense that it abstracts away from physical time. The application code running on a given platform, however, is a dynamic system that can be modeled as a timed or hybrid automaton [3]. The run-time state includes not only the variables of the application software, but also all variables that are needed to characterize its dynamic behavior, such as time variables and other quantities used to model resources. We need tractable theories to relate component-based models at application and implementation levels. In particular, such theories must provide means for preserving, in the implementation, all essential properties of the application software.

Unified frameworks encompassing heterogeneity in systems design have been proposed in [5, 8, 6, 4]. Nevertheless, in these works unification is achieved by reduction to a common low-level semantic model. We need a framework which is not just a disjoint union of submodels, but one which preserves properties during model composition and supports meaningful analyses and transformations across heterogeneous model boundaries.

We present the BIP (Behavior, Interaction, Priority) framework for modeling heterogeneous real-time components. BIP integrates results developed at Verimag over the past five years. It is characterized by the following:

- It supports a component construction methodology based on the thesis that components are obtained as the superposition of three layers. The lower layer describes *behavior*. The intermediate layer includes a set of *connectors* describing the *interactions* between transitions of the behavior. The upper layer is a set of *priority* rules describing scheduling policies for interactions. Layering implies a clear separation between

behavior and *structure* (connectors and priority rules).

- It uses a parameterized binary *composition* operator on components. The product of two components consists in composing their corresponding layers separately. Parameters are used to define new interactions as well as new priority rules between the composed components [11, 13]. The use of such a composition operator allows *incremental* construction. That is, any compound component can be obtained by successive composition of its constituents. This is a generalization of the associativity/commutativity property for composition operators whose parameters depend on the order of composition.

- It encompasses heterogeneity. It provides a powerful mechanism for structuring interactions involving strong synchronization (rendezvous) or weak synchronization (broadcast). Synchronous execution is characterized as a combination of properties of the three layers. Finally, timed components can be obtained from untimed components by applying a structure preserving transformation of the three layers.

- It allows considering the *system construction* process as a sequence of transformations in a three-dimensional space: $Behavior \times Interaction \times Priority$. A transformation is the result of the superposition of elementary transformations for each dimension. This provides a basis for the study of property preserving transformations or transformations between subclasses of systems such as untimed/timed, asynchronous/synchronous and event-triggered/data-triggered.

The paper is structured as follows. Section 2 presents the BIP language and the underlying concepts for component composition. Section 3 presents the operational semantics of the language and associated tools for execution and analysis. Section 4 shows that timed and synchronous systems correspond to particular classes of BIP components. This is illustrated by examples provided in Section 5. Finally, Section 6 discusses some more fundamental issues about the component construction space and its properties.

2. The BIP language — Basics for Component Composition

The BIP language supports a methodology for building components from:

- *atomic* components, a class of components with behavior specified as a set of transitions and having empty interaction and priority layers. Triggers of transitions include *ports* which are action names used for synchronization.

- *connectors* used to specify possible interaction patterns between ports of atomic components.

- *priority relations* used to select amongst possible interactions according to conditions depending on the state of the integrated atomic components.

We provide a description of the main features of the language.

2.1. Atomic Components

An atomic component consists of:

- A set of *ports* $P = \{p_1 \dots p_n\}$. Ports are action names used for synchronization with other components.

- A set of *control states* $S = \{s_1 \dots s_k\}$. Control states denote locations at which the components await for synchronization.

- A set of *variables* V used to store (local) data.

- A set of transitions modeling atomic computation steps. A transition is a tuple of the form (s_1, p, g_p, f_p, s_2), representing a step from control state s_1 to s_2. It can be executed if the guard (boolean condition on V) g_p is true and some interaction including port p is offered. Its execution is an atomic sequence of two microsteps: 1) an interaction including p which involves synchronization between components with possible exchange of data, followed by 2) an internal computation specified by the function f_p on V. That is, if v is a valuation of V after the interaction, then $f_p(v)$ is the new valuation when the transition is completed.

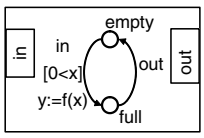

Figure 1. An atomic component.

Figure 1 shows an atomic reactive component with two ports in, out, variables x, y, and control states $empty$, $full$. At control state $empty$, the transition labeled in is possible if $0 < x$. When an interaction through in takes place, the variable x is eventually modified and a new value for y is computed. From control state $full$, the transition labeled out can occur. The omission of guard and function for this

transition means that the associated guard is $true$ and the internal computation microstep is empty. The syntax for atomic components in BIP is the following:

atom::=
component *component_id*
port *port_id*$^+$
[**data** *type_id data_id*$^+$]
behavior
{**state** *state_id*
{**on** *port_id* [**provided** *guard*]
[**do** *statement*] **to** *state_id*}$^+$}$^+$
end
end

That is, an atomic component consists of a declaration followed by the definition of its behavior. Declaration consists of ports and data. Ports are identifiers and for data, basic C types can be used. In the behavior, $guard$ and $statement$ are C expressions and statements respectively. We assume that these are adequately restricted to respect the atomicity assumption for transitions e.g. no side effects, guaranteed termination.

Behavior is defined by a set of transitions. The keyword **state** is followed by a control state and the list of outgoing transitions from this state. Each transition is labelled by a port identifier followed by its guard, function and a target state.

The BIP description of the reactive component of figure 1 is:

component $Reactive$
port in, out
data int x, y
behavior
state $empty$
on in **provided** $0 < x$ **do** y:=f(x) **to** $full$
state $full$
on out **to** $empty$
end
end

2.2. Connectors and Interactions

Components are built from a set of atomic components with disjoint sets of names for ports, control states, variables and transitions.

Notation: We simplify the notation for sets of ports in the following manner. We write $p_1|p_2|p_3|p_4$ for the set $\{p_1, p_2, p_3, p_4\}$ by considering that singletons are composed by using the associative and commutative operation $|$.

A connector γ is a set of ports of atomic components which can be involved in an interaction. We assume that connectors contain at most one port from each atomic component. An interaction of γ is any non empty subset of this set. For example, if p_1, p_2, p_3 are ports of distinct atomic components, then the connector $\gamma = p_1|p_2|p_3$ has seven interactions: $p_1, p_2, p_3, p_1|p_2, p_1|p_3, p_2|p_3, p_1|p_2|p_3$. Each non trivial interaction i.e., interaction with more than one port, represents a synchronization between transitions labeled with its ports.

Following results in [11], we introduce a typing mechanism to specify the *feasible* interactions of a connector γ, in particular to express the following two basic modes of synchronization:

- Strong synchronization or rendezvous, when the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ.
- Weak synchronization or broadcast, when feasible interactions are all those containing a particular port which initiates the broadcast. That is, if $\gamma = p_1|p_2|p_3$ and the broadcast is initiated by p_1, then the feasible interactions are $p_1, p_1|p_2, p_1|p_3, p_1|p_2|p_3$.

The typing mechanism distinguishes between *complete* and *incomplete* interactions with the following restriction: All the interactions containing some *complete* interaction are complete; dually, all the interactions contained in incomplete interactions are incomplete. An interaction of a connector is *feasible* if it is complete or if it is maximal.

Preservation of completeness by inclusion of interactions allows a simple characterization of interaction types. It is sufficient, for a connector γ to give the set of its minimal complete interactions. For example, if $\gamma = p_1|p_2|p_3|p_4$ and the minimal complete interactions are p_1 and $p_2|p_3$, then the set of the feasible interactions are $p_1, p_2|p_3, p_1|p_4, p_2|p_3|p_4$, $p_1|p_2|p_3, p_1|p_2|p_3|p_4$.

If the set of the complete interactions of a connector is empty, that is all its interactions are incomplete, then synchronization is by rendezvous. Broadcast through a port p_1 triggering transitions labeled by ports $p_2, \ldots, p_n$ can be specified by taking p_1 as the only minimal complete interaction.

The syntax for connectors is the following:

interaction ::= *port_id*$^+$
connector::=
connector *conn_id* = *port_id*$^+$
[**complete** = interaction$^+$]
[**behavior**
{**on** interaction [**provided** *guard*] [**do** *statement*]}$^+$
end]

That is, a connector description includes its set of ports followed by the optional list of its minimal complete interactions and its behavior. If the list of the minimal complete

interactions is omitted, then this is considered to be empty. Connectors may have behavior specified as for transitions, by a set of guarded commands associated with feasible interactions. If $\alpha = p_1|p_2|...|p_n$ is a feasible interaction then its behavior is described by a statement of the form: **on** α **provided** G_α **do** F_α, where G_α and F_α are respectively a guard and a statement representing a function on the variables of the components involved in the interaction. As for atomic components, guards and statements are C expression and statements respectively.

The execution of α is possible if G_α is $true$. It atomically changes the global valuation v of the synchronized components to $F_\alpha(v)$.

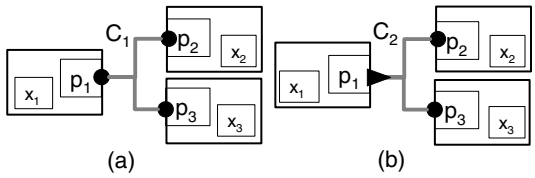

Figure 2. Interaction types.

We use a graphical notation for connectors in the form of trees. The leaves are singleton interactions and the higher level nodes are synchronization types. We denote an incomplete interaction by a bullet and a complete interaction by a triangle. For example, consider the connector C_1 described below:

connector $C_1 = p_1|p_2|p_3$
behavior
  on $p_1|p_2|p_3$ **provided** $\neg(x_1 = x_2 = x_3)$
  do $x_1, x_2, x_3 :=$ MAX(x_1, x_2, x_3)
end

It represents a strong synchronization between p_1, p_2 and p_3 which is graphically represented in figure 2(a), where the singleton incomplete interactions p_1, p_2, p_3 are marked by bullets. The behavior for the interaction $p_1|p_2|p_3$ involves a data transfer between the interacting components: the variables x_i are assigned the maximum of their values if they are not equal.

The following connector describes a broadcast initiated by p_1. The corresponding graphical representation is shown in fig 2(b).

connector $C_2 = p_1|p_2|p_3$
complete = p_1
behavior
  on p_1 **do** skip
  on $p_1|p_2$ **do** $x_2 := x_1$
  on $p_1|p_3$ **do** $x_3 := x_1$
  on $p_1|p_2|p_3$ **do** $x_2, x_3 := x_1$
end

This connector describes transfer of value from x_1 to x_2 and x_3.

Notice that contrary to other formalisms, BIP does not allow explicit distinction between inputs and outputs. For simple data flow relation, variables can be interpreted as inputs or outputs. For instance, x_1 is an output and x_2, x_3 are inputs in C_2.

2.3. Priorities

Given a system of interacting components, priorities are used to filter interactions amongst the feasible ones depending on given conditions. The syntax for priorities is the following:

priority::=
  priority *priority_id* [**if** *cond*] interaction $<$ interaction

That is, priorities are a set of rules, each consisting of an ordered pair of interactions associated with a condition (*cond*). The condition is a boolean expression in C on the variables of the components involved in the interactions. When the condition holds and both interactions are enabled, only the higher one is possible. Conditions can be omitted for static priorities. The rules are extended for composition of interactions. That is, the rule $begin_1 < begin_2$ means that any interaction of the form $begin_2|q$ has higher priority over interactions of the form $begin_1|q$ where q is an interaction.

2.4. Compound Components

A compound component allows defining new components from existing sub-components (atoms or compounds) by creating their instances, specifying the connectors between them and the priorities. The syntax of a compound component is defined by:

compound::=
  component *component_id*
    {**contains** *type_id* {*instance_id*[*parameters*]}$^+$}$^+$
    [connector$^+$]
    [priority$^+$]
  end

The instances can have parameters providing initial values to their variables through a named association.

An example of a compound component named $System$ is shown in figure 3. It is the serial connection of three reactive components, defined as:

component $System$
  contains $Reactive\ r_1, r_2, r_3$
  connector $C_1 = r_1.in$

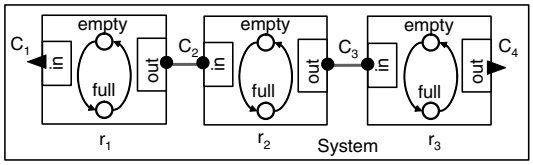

Figure 3. A compound component.

complete = $r_1.in$
connector C_2 = $r_1.out|r_2.in$
behavior
on $r_1.out|r_2.in$ **do** $r_2.x := r_1.y$
end
connector C_3 = $r_2.out|r_3.in$
behavior
on $r_2.out|r_3.in$ **do** $r_3.x := r_2.y$
end
connector C_4 = $r_3.out$
complete = $r3.out$
priority P_1 $r_1.in < r_2.out|r_3.in$
priority P_2 $r_1.in < r_3.out$
priority P_3 $r_1.out|r_2.in < r_3.out$
end

We use priorities to enforce a causal order of execution as follows: once there is an in through C_1, the data are processed and propagated sequentially, finally producing an out through C_4 before a new in occurs through C_1. This is achieved by a priority order which is the inverse of the causal order.

3. Implementation

In this section, we describe briefly the operational semantics and the execution platform for BIP.

3.1. Operational semantics

A detailed and fully formalized operational semantics is beyond the scope of this paper. We focus on aspects that are crucial for understanding the underlying mechanism for execution of interactions between a set of components. For the sake of simplification, we consider components without priorities. These are only a filtering mechanism for interactions.

Consider a compound component built from w atomic components by using a set of connectors and without priority rules. Its meaning is an extended automaton with:

- A set of variables, which is the union of the sets of variables of the atomic components.
- A set of states, which is the cartesian product of the sets of control states of the atomic components.

The extended automaton has a set of transitions of the form (s, α, g, f, s'), where:

- $s = (s_1, \ldots, s_w)$, s_i being a control state of the i^{th} atomic component.
- α is a feasible interaction associated with a guarded command (G_α, F_α), such that there exists a subset $J \subseteq \{1, \ldots, w\}$ of atomic components with transitions $\{(s_j, p_j, g_j, f_j, s'_j)\}_{j \in J}$ and $\alpha = \{p_j\}_{j \in J}$.
- $g = (\bigwedge_{j \in J} g_j) \wedge G_\alpha$.
- $f = F_\alpha; [f_j]_{j \in J}$. That is, the computation starts with the execution of F_α followed by the execution of all the functions f_j in some arbitrary order. The result is independent of this order as components have disjoint sets of variables.
- $s'(j) = s'_j$ if $j \in J$; otherwise $s'(j) = s_j$. That is, the states from which there are no transitions labeled with ports in α, remain unchanged.

The extended automaton is a machine with moves of the form $(s, v) \xrightarrow{\alpha} (s', v')$, where s is a state of the automaton and v is a valuation of its variables. The move $(s, v) \xrightarrow{\alpha} (s', v')$ is possible if there exists a transition (s, α, g, f, s'), such that $g(v) = true$ and $v' = f(v)$.

3.2. The BIP Execution Platform

The BIP framework has been partially implemented in the IF toolset [7] and the PROMETHEUS tool [10]. We have developed a full implementation of the toolset that includes:

- A frontend for editing and parsing BIP and generating C++ code to be executed and analyzed on a backend platform.
- A backend platform consisting of an engine and the infrastructure for executing the generated C++ code.
- Connections to analysis tools of the IF tool-set.

The execution engine directly implements the operational semantics. At control states, atomic components post the port names of enabled transitions. The execution engine monitors the state of atomic components and finds all the enabled interactions by evaluating the guards on the connectors. Then, between the enabled interactions, priority rules are used to eliminate the ones with low priority. Amongst the maximal enabled interactions, it executes one and notifies the atomic components involved in this interaction. The notified components continue their local computation independently and eventually reach new control states.

Two options, one for multithreaded execution (each atomic component having a thread) and the other for a single threaded execution (the execution engine being the only thread) are implemented.

The engine has a state space exploration mode, which under some restrictions on the code for guards and functions, generates state graphs that can be analyzed by using model-checking tools.

The backend, which is the BIP exploration platform, has been entirely implemented in C++ on Linux and uses POSIX threads. This choice has been made mainly for efficiency reasons but also to allow a smooth integration of components with behavior expressed using plain C/C++ code. The BIP frontend which includes the C++ code generator and some editing facilities, is implemented in Java. In particular, the frontend uses Eclipse EMF technology and associated tools for model representation and model transformations.

4. Subclasses of Components

In this section, we show that timed and synchronous systems can be represented as BIP components.

4.1. Timed Components

We define *timed* components and provide a structure preserving mapping from timed components to BIP components defined in section 2.

Timed components are built from atomic timed components. The definition of atomic timed components for discrete time is inspired from [2]. They have:

- A set of ports $P = \{p_1 \dots p_n\}$.
- A set of control states $S = \{s_1 \dots s_k\}$.
- A set of *variables* V partitioned into two sets U and X, respectively the set of *untimed* and *timed* variables.
- A set of transitions of the form $(s_1, p, g_p^u \wedge g_p^t, f_p, s_2)^{\tau_p}$, representing a step from control state s_1 to s_2. The *urgency type* τ_p which can be either *eager* or *lazy* is used to characterize the urgency of the transition. As for ordinary transitions, $g_p = g_p^u \wedge g_p^t$ is a guard, conjunction of conditions on untimed and timed variables respectively and f_p is a function on V.
- A set of *evolution functions* $\{\phi_i\}_{1 \leq i \leq k}$ in bijection with control states. The function ϕ_i gives for a given valuation x of X at control state s_i and a given value of a discrete time parameter t, the valuation $\phi_i(x,t)$ reached when time progresses by t. Further, it satisfies the following properties:
  - $\phi_i(x,0) = x$, and
  - $\phi_i(x, t_1 + t_2) = \phi_i(\phi_i(x,t_1), t_2)$.

An atomic timed component *C* represents a transition system [2] in the following manner.

Let s_i be a control state of *C* and (u,x) be a valuation of (U,X). From the state (s_i, u, x),

- Either an enabled transition can be executed independently of its urgency type - the semantics is the same as for transitions of BIP components.
- Or time can progress by one unit, leading to state $(s_i, u, \phi_i(x,1))$, if all eager transitions are disabled.

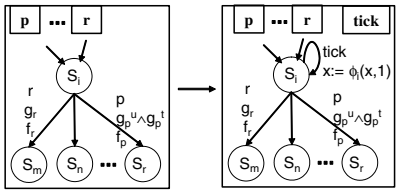

Figure 4. Transformation from Timed Component to BIP Component.

We provide a translation from timed to BIP atomic components. The principle of the translation is explained in figure 4. It consists in implementing for each atomic component, time progress from s_i and subsequent state changes, by a loop transition with guard $true$ and function $\phi_i(x,1)$. This transition has a port $tick$ for synchronization with other timed components.

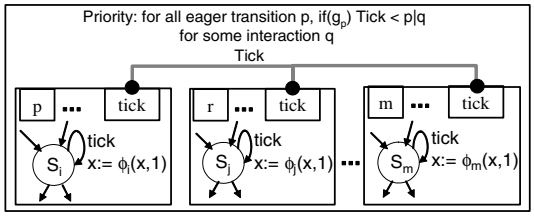

Figure 5. Composition of Timed Components.

In the composition of the resulting BIP components from the timed components, strong synchronization is necessary for the $tick$ ports as shown in the architecture of figure 5. That is, a connector $Tick$ relating all the $tick$ ports is used. $Tick$ has an empty set of complete interactions.

Furthermore, to take into account urgency of the transitions, we use priorities. For each eager transition with guard g_p, a rule of the form: **priority** P **if** $g_p\ Tick < p|q$ is used

where q is an interaction. This means that strong synchronization through $Tick$ can occur only if there are no enabled eager transitions.

4.2. Synchronous Components

Synchronous components are a subclass of components built from synchronous atomic components which have:

- A set of control states S, partitioned into three sets $S^{st} \neq \emptyset$, S^{un}, $S^{syn} \neq \emptyset$, respectively the sets of *stable*, *unstable* and *synchronization* states.
- A particular class of transitions, *synchronization* transitions. These have a special port syn, their guards are true and functions are empty. From synchronization states only synchronization transitions are possible and lead to stable states.
- Transitions from unstable states leading either to unstable states or to synchronization states.
- Stable states with synchronization transition loops. All other transitions from these states lead to unstable or synchronization states. There is a finite number of transitions in a path between two successive distinct stable states in the transition graph. This guarantees that for dealockfree execution, stable states are visited infinitely often.

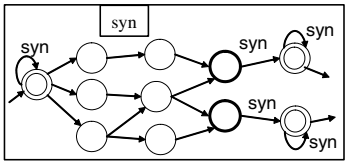

Figure 6. Synchronous Atomic Component.

The general form of a synchronous atomic component is shown in figure 6. Composition of synchronous components is characterized by the following requirements:

- Ports of transitions from unstable states must belong to complete interactions, that is, their execution cannot be blocked due to synchronization constraints.
- All the syn ports are strongly synchronized via a connector Syn with empty set of complete interactions.
- Any interaction has higher priority than the syn interaction. This enforces progress from stable states and prevents livelock by looping on syn transitions.

The general form of a synchronous architecture is shown in figure 7. The above requirements ensure lockstep execution. For such an execution, a run is a sequence of steps

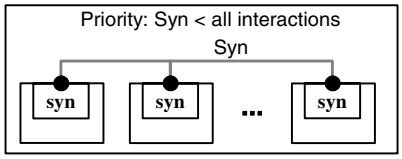

Figure 7. Composition of Synchronous Components.

leading from one global stable state to another. If all the components are at stable states, then each component can execute a sequence of transitions leading to a synchronization state. From synchronization states, strong synchronization through syn is enforced. This interaction cannot occur as long as other interactions are possible, as it has the lowest priority. It may happen that during a synchronous step some components stay at the same stable state, if the transitions leaving this state are not enabled. The synchronization transition loops from stable states allow transitions from synchronization states of the other components to be executed.

5. Applications and Case Studies

In this section we provide examples that were modeled and executed in BIP. The first example, taken from [1], is a performance evaluation problem with timed tasks processing events from a bursty event generator. The second example is a synchronous modulo-8 counter.

5.1. Tasks with Bursty Event Generator

We model $System$ which is the serial connection of a bursty event generator with three tasks t_1, t_2 and t_3, running on independent CPU's. A task can execute as soon as its predecessor has finished. The block diagram for $System$ is shown in figure 8(a).

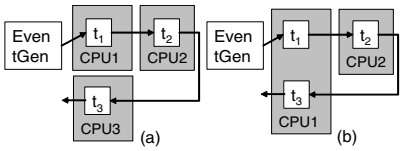

Figure 8. End to end delay measurement for tasks.

The atomic components of $System$ are $Task$ and $EventGenerator$. For each atomic component, we specify its ports, variables and behavior.

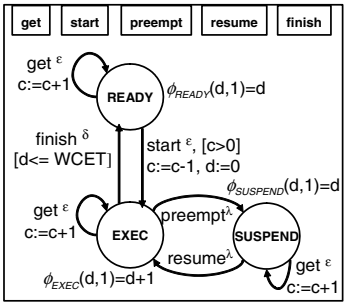

Figure 9. The Task component.

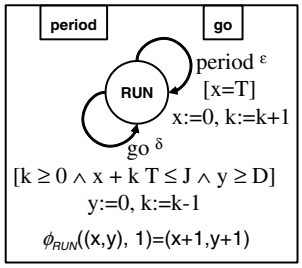

Figure 10. The EventGenerator component.

A generic timed model of $Task$ is shown in figure 9, which can be used either as a simple task, or as a preemptable task. It has a timed variable d to enforce worst case execution time $WCET$. In the graphical notation, urgency types are associated with ports. We use the notation p^{τ}, where τ can be either ϵ (eager), λ (lazy), or δ (delayable). The latter is a composite type very useful in practice. It means that the transition labeled with p is considered to be lazy at some state if it remains enabled at the next time unit; otherwise, it is eager. For instance, the notation $finish^{\delta}$ in figure 9 means that the transition is lazy when $d < WCET$ and eager when $d = WCET$.

The ports and behavior of $EventGenerator$ are shown in figure 10. It has a period T and a jitter J with $J > T$ and has a minimum inter-arrival time D between successive events being generated.

The compound component $System$ is a serial connection of the $EventGenerator$ and three instances of $Task$, t_1, t_2 and t_3, as shown in figure 11. In this model, the tasks are non-preemptable, so the ports *preempt* and *resume* are not associated to connectors. Component instances are parameterized: *EventGenerator* instance by the time-period *T*, jitter *J* and minimum inter-arrival time *D*; *Task* instances by their worst case execution time *WCET*. The BIP description of $System$ without the priorities enforcing urgency is shown below:

component $System$
  contains $EventGenerator\ e(T = 10, J = 40, D = 1)$
  contains $Task\ t_1(WCET = 8), t_2(WCET = 4),$
    $t_3(WCET = 1)$
  connector C_0 = $e.tick|t_1.tick|t_2.tick|t_3.tick$
  connector C_1 = $e.period$
  complete = $e.period$
  connector C_2 = $e.go|t_1.get$
  connector C_3 = $t_1.finish|t_2.get$
  connector C_4 = $t_2.finish|t_3.get$
  connector C_5 = $t_3.finish$
  complete = $t_3.finish$
  connector C_6 = $t_1.start$
  complete = $t_1.start$
  connector C_7 = $t_2.start$
  complete = $t_2.start$
  connector C_8 = $t_3.start$
  complete = $t_3.start$
  priority ...
end

The maximum end-to-end delay was calculated by exhaustive state space exploration of *System* composed with an observer component measuring the end to end delay of the events. With the values of the parameters as used in the BIP description above, the maximum delay obtained was 43 time units ($tick$ transitions).

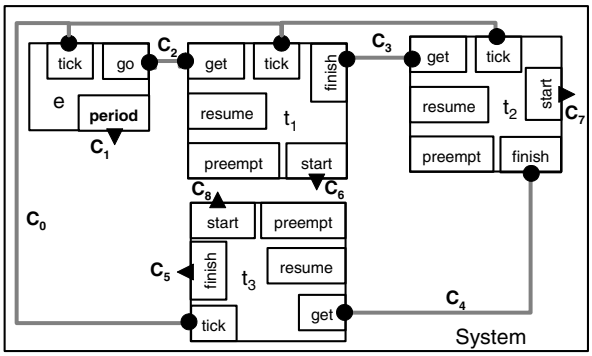

Figure 11. Architecture of System.

Consider a modification of *System* where tasks t_1 and t_3 share CPU1 and t_1 can preempt t_3, whereas t_2 runs on CPU2, as shown in figure 8(b). Its BIP description without the priorities enforcing urgency is given below:

component $System$
  contains $EventGenerator\ e(T = 10, J = 40, D = 1)$
  contains $Task\ t_1(WCET = 8), t_2(WCET = 4),$
    $t_3(WCET = 1)$
  connector C_0 $e.tick|t_1.tick|t_2.tick|t_3.tick$
  connector C_1 = $e.period$
  complete = $e.period$
  connector C_2 = $e.go|t_1.get$
  connector C_3 = $t_1.finish|t_2.get|t_3.resume$
  complete = $t_1.finish|t_2.get$

connector $C_4 = t_3.finish|t_1.resume$
complete = $t_3.finish$
connector $C_5 = t_1.start|t_3.preempt$
complete = $t_1.start$
connector $C_6 = t_3.start|t_1.preempt$
complete = $t_3.start$
connector $C_7 = t_2.start$
complete = $t_2.start$
connector $C_8 = t_2.finish|t_3.get$
priority P_1 $t_3.start < t_1.start$
priority P_2 $t_3.start|t_1.preempt < t_1.finish|t_2.get$
priority ...
end

Mutual exclusion between t_1 and t_3 is enforced by the connectors C_3, C_4, C_5 and C_6. They guarantee that a task will preempt if the other has to start, and similarly a task can resume only when the executing task finishes, as depicted in the architecture of figure 12. Priority P_1 enforces the static priority between t_1, t_3 and priority P_2 ensures non preemption of t_1 by t_3. The maximum end to end delay recorded by the observer component in this example was 194 time units.

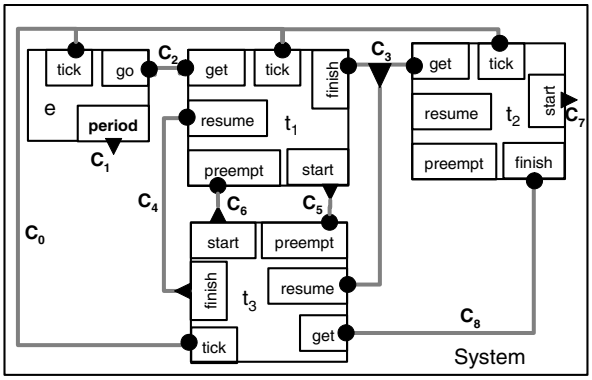

Figure 12. Architecture of System with mutual exclusion.

5.2. Synchronous Modulo-8 Counter

We model a synchronous modulo-8 counter producing a 3 bit count value. A bit is modeled as an atomic component representing a modulo-2 counter. The modulo-8 counter is obtained by composing three modulo-2 counters.

The behavior of the atomic component $Modulo2Counter$ is depicted in figure 13. *Zero'* and *One'* are *stable* states whereas *Zero* and *One* are *synchronization* states respectively. The port $flip$ corresponds to a complete interaction. The variable x is the input and y is the output.

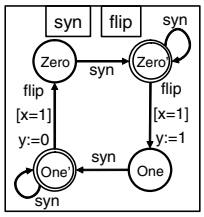

Figure 13. The Modulo2Counter component.

The compound component $Counter$ consists of 3 instances of $Modulo2Counter$ namely b_0, b_1, and b_2, where b_0 is the least significant bit. The C_0 connector consists of all the syn ports requiring strong synchronization. All the $flip$ interactions are complete.

component $Counter$
contains $Modulo2Counter$ b_0, b_1, b_2
connector $C_0 = b_0.syn|b_1.syn|b_2.syn$
behavior
on $b_0.syn|b_1.syn|b_2.syn$
do b_0.x:=1, b_1.x:=b_0.y, b_2.x:=b_0.y$\wedge$$b_1$.y
end
connector $C_1 = b_0.flip$
complete = $b_0.flip$
connector $C_2 = b_1.flip$
complete = $b_1.flip$
connector $C_3 = b_2.flip$
complete = $b_2.flip$
priority P_0 $b_0.syn|b_1.syn|b_2.syn < b_0.flip$
priority P_1 $b_0.syn|b_1.syn|b_2.syn < b_1.flip$
priority P_2 $b_0.syn|b_1.syn|b_2.syn < b_2.flip$
end

The global data flow is encoded in the behavior of the *syn* connector C_0. The input (x) of b_0 representing the least significant bit of the counter, is always set to 1, for b_2, the input is the conjunction of the outputs(y) of the previous two bits(b_0, b_1). The y values form the output of the *Counter* on execution generates the count sequence 000 to 111. Low priority of the *syn* interaction favours internal computations of the atomic components.

6. Discussion

The BIP framework shares features with existing ones for heterogeneous components, such as [5, 8, 6, 4]. A common key idea is to encompass high-level structuring concepts and mechanisms. Ptolemy was the first tool to support this by distinguishing between behavior, channels, and directors. Similar distinctions are also adopted in Metropolis and BIP, which offer interaction-based and control-based mechanisms for component integration. The two types of

mechanisms correspond to *cooperation* and *competition*, two complementary fundamental concepts for system organization.

There is evidence through numerous examples treated in BIP, that the combination of interaction and control mechanism allows enhanced modularity and direct modeling of schedulers, quality controllers and quantity managers.

BIP characterizes components as points in a three-dimensional space: $Behavior \times Interaction \times Priority$, as represented in figure 14. Elements of the $Interaction \times Priority$ space characterize the overall *architecture*. Each dimension, can be equipped with an adequate partial order, e.g., refinement for behavior, inclusion of interactions, inclusion of priorities. Some interesting features of this representation are the following:

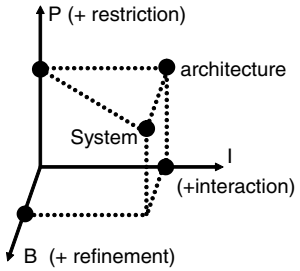

Figure 14. The Construction Space.

Separation of concerns: Any combination of behavior, interaction and priority models meaningfully defines a component. This is not the case for other formalisms e.g., in Ptolemy [8], for a given model of computation, only particular types of channels can be used. Separation of concerns is essential for defining a component's construction process as the superposition of elementary transformations along each dimension.

Unification: Different subclasses of components e.g.,untimed/timed, asynchronous/synchronous, event-triggered/data-triggered, can be unified through transformations in the construction space. These often involve displacement along the three coordinates. They allow a deeper understanding of the relations between existing semantic frameworks in terms of elementary behavioral and architectural transformations.

Correctness by construction: The component construction space provides a basis for the study of architecture transformations allowing preservation of properties of the underlying behavior. The characterization of such transformations can provide (sufficient) conditions for correctness by constructions such as *compositionality* and *composability* results for deadlock-freedom [11].

Current work on BIP deals with both theoretical issues and applications. Theoretical work directions include the study of a notion of glue for components and its properties as well as a notion of expressiveness for component-based description languages. Applications aim at "componentizing" existing real-time software written in C++ including an MPEG encoder and an adaptive robotic application. Finally, we are developing a tool for translating a subset of the BIP language into THINK [9], from which implementations on bare machines can be generated.

References

[1] Workshop on distributed embedded systems, lorentz center, leiden, 2005. http://www.tik.ee.ethz.ch/ leiden05.

[2] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1-2):55–84, 2002.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[4] F. Arbab. Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52, 2005.

[5] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

[6] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *IEEE Computer*, 39(2):33–40, 2006.

[7] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *SFM*, pages 237–267, 2004.

[8] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity: The Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[9] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *Proceedings of the Usenix Annual Technical Conference*, June 2002.

[10] G. Gößler. PROMETHEUS — a compositional modeling tool for real-time systems. In P. Pettersson and S. Yovine, editors, *Proc. Workshop RT-TOOLS 2001*. Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.

[11] G. Gößler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[12] T. Henzinger and J. Sifakis. The embedded systems design challenge. *Proceedings FM 2006, LNCS*, 2006.

[13] J. Sifakis. A framework for component-based construction. In *Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM)*, pages 293–300. IEEE Computer Society, 2005.