A Methodology and Supporting Tools for the Development of Component-Based Embedded Systems.

Marc Poulhiès, Jacques Pulou, Christophe Rippert, Joseph Sifakis

A Methodology and Supporting Tools for the Development of Component-Based Embedded Systems

Marc Poulhiès [1,2], Jacques Pulou[2], Christophe Rippert[1], Joseph Sifakis[1]
{poulhies, rippert, sifakis}@imag.fr,
jacques.pulou@orange-ftgroup.com

[1]VERIMAG and [2] France Telecom R&D

Abstract. The paper presents a methodology and supporting tools for developing component-based embedded systems running on resource-limited hardware platforms. The methodology combines two complementary component frameworks in an integrated tool chain: BIP and THINK. BIP is a framework for model-based development including a language for the description of heterogeneous systems, as well as associated simulation and verification tools. THINK is a software component framework for the generation of small-footprint embedded systems. The tool chain allows generation, from system models described in BIP, of a set of functionally equivalent THINK components. From these and libraries including OS services for a given hardware platform, a minimal system can be generated. We illustrate the results by modeling and implementing a software MPEG encoder on an iPod.

1 Introduction

Embedded systems development is subject to strong requirements for optimality in the use of resources, and correctness with respect to non-functional properties, as well as requirements for time-to-market and low cost through reuse and easy customization.

We need holistic methodologies and supporting tools for all development activities from application software to implementation. The methodologies should be component-based to ease code reuse, modularity, reconfiguration and allow implementations having a minimal footprint by only including the necessary services. Components allow abstractions for structuring code according to a logical separation of concerns. For early design error detection, application of validation and analysis techniques is essential, especially to guarantee non-functional properties. Finally, the methodologies should rely on automated implementation techniques that, for a given hardware platform, make the best possible use of its characteristics and include only strictly necessary OS services.

Model-based development techniques aim at bridging the gap between application software and its implementation by allowing predictability and guidance through analysis of global models of the system under development. They offer in principle, domain-specific abstractions independent of programming languages and implementation choices. Nevertheless, they rely on system component models, which drastically differ from software component models used for operating systems and middleware [1–3]. For system description, components should encompass real-time behavior, rich interfaces and a notion of composition for natural description of heterogeneous interaction and computation. In contrast, software components allow structuring and reuse of functions and associated data. They use point-to-point interaction (*e.g.* function calls) through binding interface specifications.

We present a fully component-based methodology and supporting tools for the development of real-time embedded systems. The methodology combines two complementary component frameworks in an integrated tool chain: BIP [4, 5] and THINK [6] (see Figure 1). BIP is a framework for model-based development including a language for the description of heterogeneous real-time systems, as well as associated simulation and verification tools. THINK is a software component framework for the generation of small-footprint embedded systems. Our tool chain allows generation, from system models described in BIP, of a set of functionally equivalent THINK components. From these and libraries including OS services, a minimal system is generated for a given hardware platform.

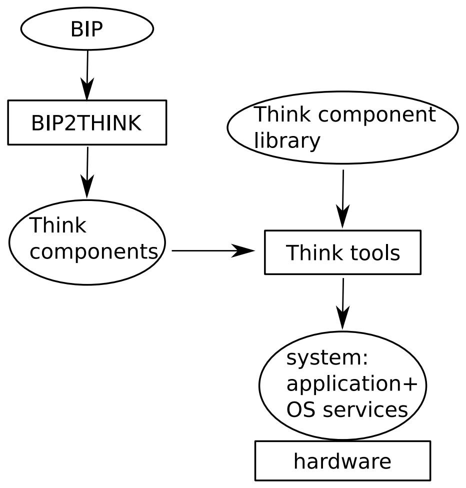

Fig. 1. The BIP to THINK tool-chain

The paper deals with the integration of component-based approaches used at the two ends of the development chain. Usually, model-based techniques focus on system description and analysis while they provide limited or very specific support for component-based implementation. For instance, tools supporting heterogeneous description, such as Ptolemy [7] or Metropolis [8], do not address implementation issues. Others, such as MetaH [9], Giotto [10] or ROOM [11], rely on given models of computation and provide support for specific implementations.

Component-based techniques for operating systems and middleware lack analysis capabilities [1]. This motivates work on modeling middleware and operating systems, for instance to evaluate performance and validate configuration mechanisms as in [12]. However, such works typically use a standard system architecture with all system services located in the kernel, and providing no support for applications to control the behavior of low-level services. This requires modeling the kernel in order to validate its runtime behavior, a very difficult task considering the complexity of standard kernels.

In contrast, Think is based on the exokernel paradigm [13] leading to minimal solutions involving only the strictly necessary services. Using this paradigm permits to move all the critical services (such as scheduling for instance) into the application space, where they can be validated with the applications. This leaves only very basic functionalities into the nano-kernel, which can be tested separately to guarantee their proper runtime behavior. Think is a mature exokernel technology which has been successfully used to generate implementations for very constrained platforms such as smart cards [14], AVR (ATmega 2561, 8bits microprocessor, 8Kb RAM, 256Kb FLASH) as well as ARM platforms (32Mb RAM, 64Mb FLASH).

Our work integrates heterogeneous system modeling and analysis with a general component-based implementation techniques. In this respect, it has similar objectives with the work around nesC/TOSSIM/TinyOS environment for the development of wireless sensor networks [15–17]. This framework has a more narrow application scope and the integration between programming, simulation and implementation tools is much stronger. However, in contrast with TinyOS, Think preserves the components as runtime entities, permitting dynamic reconfiguration or component replacement. Our work has also some similarities with VEST [18]. However, VEST relies upon a thread-based model, whereas neither BIP nor Think adopt any specific behavioral model.

The paper is organized as follows. Section 2 presents the BIP component framework used to model the behavior and structure of systems. Section 3 describes the Think framework which provides the library and tool chain used to generate system implementations. The generation tool used to translate a BIP description to a Think system is presented in Section 4. A quantitative evaluation of the results on a software MPEG encoder is presented in Section 5.

2 The BIP component model

BIP[5, 19] (*Behavior, Interaction, Priority*) is a framework for modeling heterogeneous real-time components. BIP 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.

The application of this methodology leads to layered components (see fig 2). The lower layer describes the behavior of a component as a set of atomic components; the intermediate layer includes connectors describing interactions between transitions of the layer underneath; the upper layer consists of a set of priority rules used to describe scheduling policies for interactions.

This methodology allows a clear separation between behavior and structure of a system (interactions and priorities).

The implementation of the BIP component framework includes a language for hierarchical component modeling, and a code generator for an execution platform on Linux. The execution platform allows simulation as well as exhaustive state space enumeration. The generated models can be validated by using techniques available in Verimag's IF toolset [20, 21].

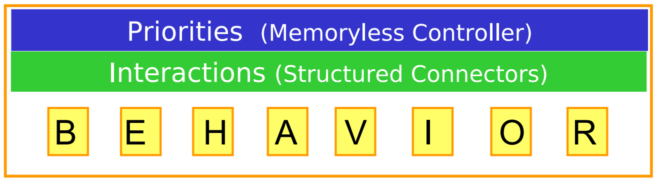
Priorities (Memoryless Controller)
Interactions (Structured Connectors)
B
E
H
A
V
I
O
R

Fig. 2. Layered component model

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

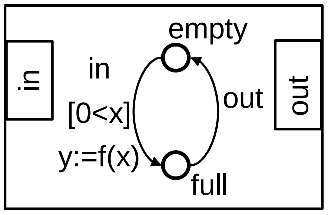

Fig. 3. An atomic component.

Figure 3 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 $0 < x$. Interactions through *in* may modify the variable x . They are immediately followed by the computation of a new value for y. 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 [complete | incomplete] 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 have attributes complete and incomplete whose meaning will be explained in 2.2. 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 3 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
```

The following example shows an atomic component modeling the control of a simple preemptable task:

```
component Task
    port complete awake,begin,finish
    port incomplete preempt, resume
    behavior
        state IDLE
            on awake to WAIT
        state WAIT
            on begin to EXECUTE
        state EXECUTE
            on finish to IDLE
            on preempt to SUSPEND
        state SUSPEND
            on resume to EXECUTE
    end
end
```

The component has four control states called `IDLE`, `WAIT`, `EXECUTE` and `SUSPEND`, five ports called `awake`, `begin`, `finish`, `preempt` and `resume`. The ports have attributes *complete* and *incomplete* which characterize the way they synchronize with other ports to form *interactions* (see next section).

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 [22], 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 $p_1, p_1|p_2, p_1|p_3, p_1|p_2|p_3$.

A system run is a sequence of feasible interactions.

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: the only feasible interaction involves all the ports of the connector (this is the maximal incomplete interaction of the connector), see figure 4(a). 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 expressions 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 $F_\alpha(v)$.

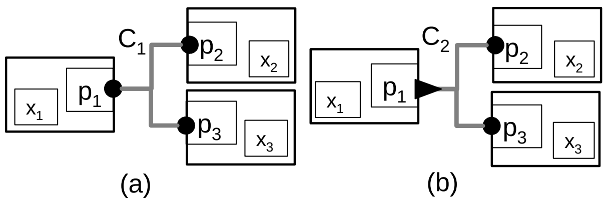

Fig. 4. Interaction types.

We use a graphical notation for connectors in the form of trees.

We denote an incomplete singleton interaction by a bullet on the corresponding port and a complete singleton interaction by a triangle. A generalisation of this notation is possible to describe hierarchical connectors [23]. 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 $x_1, x_2, x_3 := \text{MAX}(x_1, x_2, x_3)$
end

It represents a strong synchronization between p_1, p_2 and p_3 which is graphically represented in figure 4(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 4(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 relations, 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.

Notation: We simplify the notation for repetitive rules. It is possible to have a set of interactions instead of a single interaction in the previous pair. All the interactions in the left hand side set have a lower priority than all the interactions in the right hand side set.

The *System* example given in section 2.4 illustrates the use of priorities.

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 5. It is the serial connection of three reactive components, defined as:

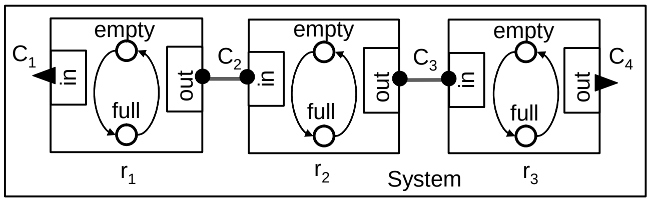

Fig. 5. A compound component.

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

The following example shows a compound component obtained by composition of three instances `task1`, `task2` and `task3` of the atomic component `Task`, given in the previous example in section 2.1.

```
component $FPPS$
    contains $Task\ task1, task2, task3$
    connector $beg1 = task1.begin,\ task2.preempt, task3.preempt$
    connector $beg2 = task2.begin,\ task1.preempt, task3.preempt$
    connector $beg3 = task3.begin,\ task1.preempt, task2.preempt$
    connector $fin1\_res2 = task1.finish,\ task2.resume$
    connector $fin1\_res3 = task1.finish,\ task3.resume$
    connector $fin2\_res1 = task2.finish,\ task1.resume$
    connector $fin2\_res3 = task2.finish,\ task3.resume$
```

connector $fin3\_res1 = task3.finish,\ task1.resume$
connector $fin3\_res2 = task3.finish,\ task2.resume$
priority
// Priorities are shown below
end

The connectors are used to enforce mutual exclusion, that is, at most one task can be in state `EXECUTE`. For example, the connector `beg`i is used to force preemption by `task`i of the other tasks when they are at state `EXECUTE`. The connector `fin`i`_res`$j (i \neq j)$ is used to resume preempted tasks when `task`i finishes. It is easy to check mutual exclusion between tasks in the compound component.

We show below the three types of rules used to enforce decreasing priorities between the tasks.Rules `beg`$i\_j (i \neq j)$ for beginning tasks of higher priority. Rules `beg`i`pre`j to avoid preemption of tasks of higher priority. Finally, the rule `fin1_2_3` ensures that when both `task2` and `task3` are suspended, `task2` will resume.

priority // resume task2 if both task3 and
// task2 are suspended
priority `fin1_2_3` $task1.finish|task3.resume < task1.finish|task2.resume$
// do not start 3 if 1 is ready
priority `beg1_3` $task3.begin, task3.begin|task1.preempt, task3.begin|task2.preempt$
$< task1.begin$
// do not start 2 if 1 is ready
priority `beg1_2` $task2.begin, task2.begin|task1.preempt, task2.begin|task3.preempt$
$< task1.begin$
// do not start 3 if 2 is ready
priority `beg2_3` $task3.begin, task3.begin|task2.preempt, task3.begin|task1.preempt$
$< task2.begin$
// do not start 2 if 1 is executing
priority `beg2pre1` $task2.begin|task1.preempt < task1.preempt$
// do not start 3 if 1 is executing
priority `beg3pre1` $task3.begin|task1.preempt < task1.preempt$
// do not start 3 if 2 is executing
priority `beg3pre2` $task3.begin|task2.preempt < task2.preempt$

2.5 Implementation

The implementation of the BIP framework (see figure 6) includes a frontend for editing and parsing BIP programs, and a dedicated platform for the validation of models. The execution platform (BIP/Linux Platform on fig. 6) consists of an Engine and software infrastructure for executing the models. It directly implements BIP's operational semantics in the following manner:

At a given control state, an atomic component waits for interactions through the ports of the transitions enabled at that state. The Engine has access to the connectors and the priority rules of the compound components. When all the atomic components are waiting for interaction, the Engine:

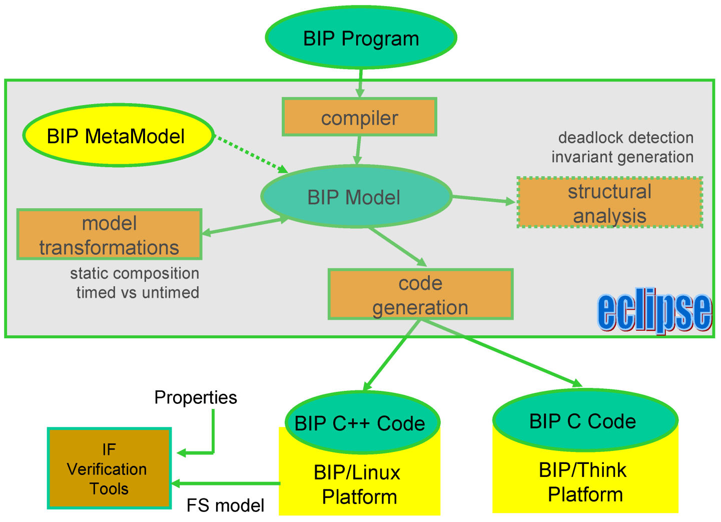

Fig. 6. BIP Framework

1. computes the possible interactions;
2. filters by using priority rules the possible interactions by considering only the maximal ones according to the priority orders;
3. chooses and executes one maximal interaction. The execution may involve transfer of data between the interacting components. These are notified at the end of the transfer to continue the execution of their interacting transitions.

The platform allows state space exploration and provides access to the model-checking tools of the IF toolset [20, 21]. It generates a finite state model that is fed to the IF tools permitting to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability.

For instance, it is easy to check that the `FPPS` example is deadlock-free. For schedulability analysis, a timed BIP model is needed. It can be obtained by adding timing constraints (*e.g.* enforcing periodicity of the `awake` port and worst case execution times on the `finish` port) to the system model. Timing constraints are expressed in BIP by using variables modeling clocks following the hybrid automata paradigm [24]. More information about modeling real-time systems in BIP can be found at [19].

3 The Think framework

THINK[6, 25] (*THink Is Not a Kernel*) is a software framework for the development of small-footprint embedded systems. It includes a programming model, a library of operating system abstractions, and a set of tools dedicated to automatize the configuration and building processes.

Think is an implementation of the Fractal component model [26]. Fractal has been implemented on various software platforms (Java [27], .Net, C++ [28], etc.) and for various uses (middlewares, multimedia applications, aspect-oriented programming, etc.). Think is a C/assembly implementation of Fractal aiming as easing the development of low-footprint embedded systems. Fractal is a hierarchical component model which advocates design patterns, such as separation of concerns for instance, to reduce development and maintenance costs of complex software systems. A component in Fractal consists of two parts: a functional core which implements the service provided by the component, and a control layer used to manage the component itself and implement non-functional properties. Fractal programming model is based on the *export-bind* design pattern, which guarantees the flexibility of the composition and permits to develop modular implementations of system services. Components in Fractal can be dynamically reconfigured or replaced due to the separation between the functional part of a component and its control interface [29]. This programming model is a major asset compared to similar system-building tools such as the OSKit [30] for instance, as it permits to manage components as runtime entities which can be easily reconfigurated or replaced.

Think includes a library of standard system abstractions optimized for various embedded platforms (ARM, PowerPC, Xscale, AVR, etc) that can be used to build minimal systems suitable for execution on severely constrained hardware platforms. Think includes both platform-independent services (*i.e.* memory manager, TCP/IP stack, file-systems, etc.) and services depending on the characteristics of the underlying hardware (*i.e.* MMU manager, NIC and IDE drivers, etc.). The modularity of the Fractal component model permits to link only the required services, without having to manage cross-dependencies between modules as this is often the case in monolithic kernels. This flexibility is a major asset with respect to traditional embedded operating systems which typically include all the system services that could possibly be used by applications, resulting in a major waste of memory for embedded applications. Think is thus especially well suited for resource-limited embedded systems as it permits to build dedicated runtime environments including only the system abstractions needed by the embedded applications.

Think offers various tools easing the configuration and building of the system. The structure of the system is described using an Architecture Description Language (ADL) that permits to specify which component must be included in the system and the static links between the components. An Interface Description Language is used to describe the services implemented by the components and how they can interact. A generation chain takes these descriptions and automatically generates a minimal system composed of the applications, the selected system services and the software framework needed to make them interact.

4 The BIP to Think compiler

We developed a compiler which generates THINK components from BIP source code, as well as the glue code needed to bind them (C and ADL source files). Figure 7 depicts the translation process, which preserves the structure of BIP models. Atomic components are translated into THINK components. For each connector a THINK component is generated. Priorities are implemented by using a specific THINK component. Finally, an Engine component is used to implement the operational semantics of BIP. Checking the correctness of the translation can be decomposed into two steps:

– checking the correctness of each individual translation for atomic components, connectors and priorities. These translations are simple expansions of the BIP code which are easy to check.
– checking correctness of the Engine which is the only active component. The Engine implements the semantics in the form of a simple automaton (see figure 8).

Bellow, we illustrate the method by using the `FPPS` example.

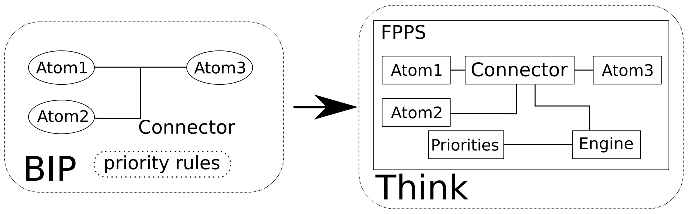

Fig. 7. The BIP to THINK translation process

4.1 Atomic components

Each BIP atomic component is mapped into a THINK component. Each data variable of an atomic BIP component is translated into an interface (`Data`) exported by the corresponding primitive THINK component. The interface consists in two simple methods `get` and `set`.

Similarly, each port of an atomic BIP component is translated into an interface (`Port`) with 2 main methods: `isSynced`, to evaluate the guard of the transition associated with this port, and `execute`, to compute the function of this transition. We provide below the code generated for the port `begin` and the transition from state `WAIT` to `EXECUTE`:

```
#define TASK_EXECUTE 3
```

```
// self variable is the Task component
//   instance reference used to access
//   bound interfaces and component's
//   variables
_enter_state_EXECUTE (Taskdata *self) {
  reset_ports(self);
  self->finish_port = 1;
  self->preempt_port = 1;
  self->state = TASK_EXECUTE;
}

_begin_execute (Taskdata *self){
  if (self->state == TASK_WAIT){
    _enter_state_EXECUTE(self);
  }
}
```

Notice that when a state is entered (method `_enter_state_EXECUTE` here), the variables associated with ports (`finish_port` and `preempt_port`, ...) are reset (*i.e.* set to 0) and only the variables associated with the ports that can be enabled (*ie.* the port labels an outgoing transition from present state and its guard (if any) evaluates to true) are toggled. Interactions eligible for execution must have their port variables set to 1.

4.2 Connector components

Each connector of a BIP description is translated into a Think component. This component is bound to the `Port` interface of each port of the connector and exports a `Connector` interface. If the connector has guarded commands, it is also bound to the `Data` interface of each involved variable.

The connector component computes the feasible interactions of the connector and triggers the execution of a maximal one, when it is needed. This is implemented by using 2 methods:

- `execute` which executes one maximal feasible interaction of the connector and returns false if there is no feasible interaction;
- `isLegal` which tests whether there is at least one feasible interaction to execute.

A connector component can also inhibit some interaction so as to respect priorities (see 4.3). This is implemented by 2 methods:

- `inhibit(id)` method which marks the `id` interaction as not eligible
- the `isInteractionLegal(id)` that tests whether or not the `id` interaction is feasible.

We provide below the `isLegal` and `execute` methods for a connector connecting port `begin` of the component `task1`, and ports `preempt` of components `task2` and `task3`. The compiler is able to statically compute the maximum interaction and stores it in the `MAX_INT` macro. The example below shows a *broadcast* synchronization used when the first task begins (connector `beg1`):

```
// max interaction code
#define MAX_INT 0x0007
// connector local port coding
#define TASK1_BEGIN   0x1
#define TASK2_PREEMPT 0x2
#define TASK3_PREEMPT 0x4

// self variable is the connector component
//   instance reference
bool isLegal(beg1data *self) {
// this mask has 1 bit for each port
  self->port_mask = 0;
//ask if task1.begin is synced or not
  if(CALL(self->task1_begin, isSynced)){
    self->port_mask |= TASK1_BEGIN;
  }
  if(CALL(self->task2_preempt, isSynced)){
    self->port_mask |= TASK2_PREEMPT;
  }
  if(CALL(self->task3_preempt, isSynced)){
    self->port_mask |= TASK3_PREEMPT;
  }
// legal iff begin port synced
  return (self->port_mask & TASK1_BEGIN);
}

bool execute(beg1data *self){
// if begin not synced, nothing to execute
  if (!(self->port_mask & TASK1_BEGIN)) {
    return 1;
  }
//notify all synced ports
  if (self->port_mask & TASK1_BEGIN){
    CALL(self->task1_begin, execute);
  }
  if (self->port_mask & TASK2_PREEMPT){
    CALL(self->task2_preempt, execute);
  }
  if (self->port_mask & TASK3_PREEMPT){
    CALL(self->task3_preempt, execute);
  }
  return 0;
}
```

4.3 The priority component

All BIP priority rules are implemented into a single THINK component. This component is bound to the `Connector` interface of each involved connector and to the `Data` interface of each variable used in the guards. It exports a simple `Priority` interface which includes the method `apply`. This method sequentially applies all the priority rules in the system. For example, one relation of the BIP priority rule `beg1_3` from the previous example is translated into:

```
// beg3 : task3.begin, task2.preempt
//       < beg1 : task1.begin

// self variable is the priority component
// instance reference

// guard is empty
guard = 1;
// low prio connector : beg2
cn_low = self->beg2;
// high prio connector : beg1
cn_high = self->beg1;
iid_low = BEG2_TASK2_BEGIN |
  BEG2_TASK3_PREEMPT ;
// high prio inter. id (iid)
iid_high = BEG1_TASK1_BEGIN ;

// ask high prio connector if it is legal
// and inhibit inter on lower prio connector
// if needed
if (guard && CALL(cn_high,
 isInteractionLegal, iid_high))
  CALL(cn_low, inhibit, iid_low);
```

4.4 The Engine

The `Engine` component implements the BIP Engine using a THINK component. It contains the entry point of the system generated by THINK and is responsible for scheduling the computation. It runs an infinite loop (see figure 8) choosing one maximal feasible interaction out of all possible ones and executing it. The Engine first builds a list of connectors for which at least one interaction is feasible (using the `isLegal` method for connectors), then it asks the priority component to apply priorities (using the `apply` method). Finally, it chooses a connector from the previous list and executes it (using the `execute` method of the connector).

4.5 Deployment

Figure 9 shows the architecture generated for `FPPS` after it has been deployed (running). For the sake of clarity, only one connector component is represented.

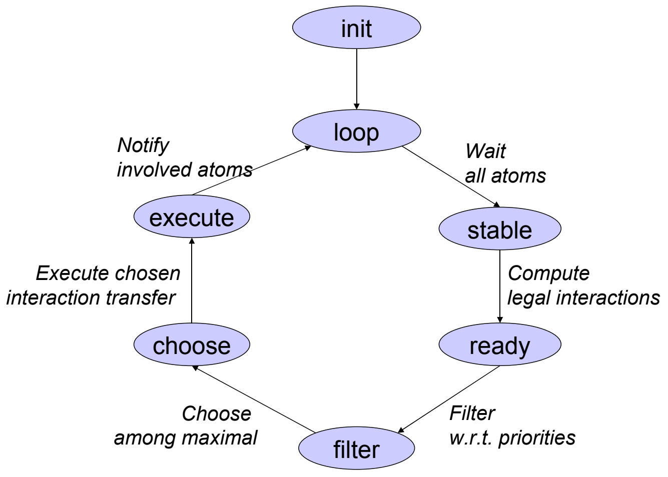

Fig. 8. Engine's loop

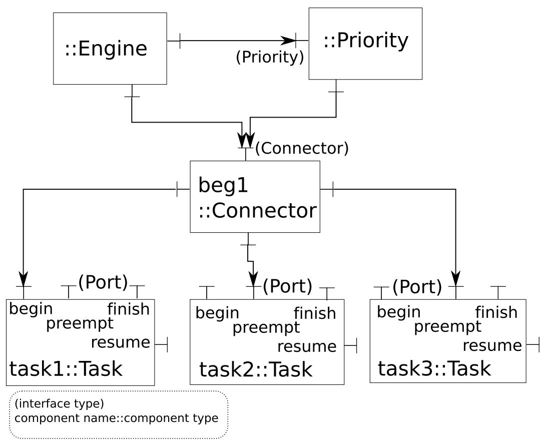

Fig. 9. The generated system after deployment

The FPPS example described in BIP in the previous section includes three tasks running in mutual exclusion. This property has been validated on the BIP code using model-checking tools. The correctness of the translation process ensures that mutual exclusion between the three tasks is respected in the code generated by the BIP2Think compiler.

5 Evaluation

To illustrate our methodology and evaluate the performances and memory-footprint of the generated system, we considered a software MPEG encoder. We started from monolithic legacy C code (approx. 7000 lines of code). We used BIP as a programming model for componentizing the C code so as to reveal causality dependencies between functions. This led to a BIP model consisting of 20 components and 34 connectors. A high-level decomposition of the BIP encoder model in shown below. Bullets represent incomplete ports and thick lines represent buffered connections (*i.e* 2 connectors with a buffer component in the middle).

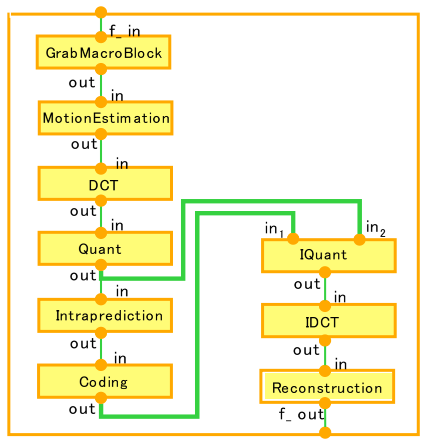

Fig. 10. BIP encoder model

We used scheduling policies proposed in [31] to control the execution of the model so as to respect given deadlines and optimize quality. The BIP to THINK compiler produces 56 components. The resulting code is 6300 lines of C code and 1000 lines of ADL code.

We generated an implementation of the encoder for an Apple iPod Video on which we only used one of the two ARM cores running at 80Mhz. The test consists in the encoding of 2 different videos.

To estimate the overhead added by the BIP componentization, we compare the generated encoder system against a monolithic implementation derived from the original encoder source code (*i.e.* without BIP or THINK). Results are given on Figures 11 and 12.

Resolution	Length	Encoding time	Speed
(in pixels)	*(in frames)*	*(in seconds)*	*(in fps)*
320×240	40	500	0.08
64×48	161	77	2

Fig. 11. Performances on the iPod Video for the BIP+Think encoder.

Resolution	Length	Encoding time	Speed
(in pixels)	*(in frames)*	*(in seconds)*	*(in fps)*
320×240	40	200	0.203
64×48	161	37	4.3

Fig. 12. Performances on the iPod Video for the monolithic encoder.

The overall encoding frame rate seems reasonable given the low CPU frequency and memory bandwidth of the iPod. Figure 12 shows an overhead in performance of roughly 100% for the BIP+Think version. This is reasonably good considering that our compiler is still in an early stage of development and has no optimization features. A more detailed analysis of the overheads by profiling, shows that this is due for approximately 66% to execution of connectors and for 33% to execution of priority rules. These can be reduced by code optimization. One possible optimization is to replace the priority component and take into account priorities at compile time by restricting the guards of the atomic components. This solution is more efficient but less modular as it is not possible to incrementally modify priorities. A similar optimization can be applied for connectors. It is possible, by using BIP's operational semantics, to replace two components by a single product component. The execution of connectors between composed components becomes an internal transition of the product component. This avoids communication overheard but also leads to a less modular solution.

The system size (including the video encoder) is 300Kb for the BIP+Think and 216Kb for the monolithic version which results in a 38% overhead in size. For comparison, a regular iPod linux kernel weights more than 1Mb without any application code.

6 Conclusion and future work

We presented a methodology and tools for the design, validation and implementation of component-based systems. The methodology integrates two existing component frameworks: one for high-level system description and analysis, the other for component-based execution and reconfiguration. BIP allows high-level system descriptions which can be simulated and validated on a workstation before being deployed on the target embedded platform. Think offers a large library of system services, already optimized for embedded hardware platforms, to produce a minimal system, based on the modularity of the library and compilation chain.

Integration is through a transformation preserving not only the semantics but also the structure of the system model. The implementation includes a set of components for scheduling, interconnect and functionality, which can be dynamically reconfigured. Rigorous operational semantics of BIP allows application of state-of-the-art validation techniques on system models for checking properties such as deadlock-freedom, state invariants and schedulability. Model validation implies validation of the implementation, provided that the tool chain and the low-level services included in the system are correct. This is ensured by the systematic approach used in the translator which maps BIP concepts directly into THINK components, and by the simplicity of the nano-kernel which includes only basic services, due to the exokernel architecture.

Bare-machine implementation is particularly appropriate for embedded systems. It allows tailored, lightweight, low-overhead solutions and precise control of execution timing. Another advantage is validation of the implementation which may be problematic when legacy operating systems are used. Faithfully modeling the underlying execution mechanisms for a given operating system is non-trivial.

Future work concerning the BIP to THINK translator includes support for external events, such as interrupts or I/O. This type of events are currently supported by THINK and can be modeled in BIP as external ports. Another work direction is the optimization of bindings in the THINK framework, in order to keep the flexibility of the export-bind design pattern without enduring the cost of going through a proxy each time an inter-component method invocation is executed.

7 Acknowledgments

We would like to thank Ananda Basu, Jacques Combaz and Loïc Strus for contributing to the work presented in this paper.

References

1. Friedrich, L., Stankovic, J., Humphrey, M., Marley, M., Haskins, J.: A Survey of Configurable, Component-Based Operating Systems for Embedded Applications. IEEE Micro **21**(3) (June 2001) 54–68
2. Group, O.M.: The CORBA Component Model Specification v4.0 (April 2006)
3. Corporation, M.: COM: Component Object Model Technologies `http://www.microsoft.com/com/`.
4. Sifakis, J.: A Framework for Component-based Construction. In: Proceedings of the International Conference on Software Engineering and Formal Methods. (September 2005)
5. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-Time Components in BIP. In: 4th IEEE International Conference International Conference on Software Engineering and Formal Methods. (September 2006)

6. Fassino, J.P., Stefani, J.B., Lawall, J., Muller, G.: THINK: A Software Framework for Component-based Operating System Kernels. In: Proceedings of the Usenix Annual Technical Conference. (June 2002)
7. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming Heterogeneity: The Ptolemy Approach. Proceedings of the IEEE **91**(1) (January 2003) 127–144
8. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An Integrated Electronic System Design Environment. IEEE Computer **36**(4) (April 2003) 45–52
9. Vestal, S.: Formal Verification of the MetaH Executive Using Linear Hybrid Automata. In: IEEE Real Time Technology and Applications Symposium. (June 2000) 134–144
10. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A., Pree, W.: From Control Models to Real-Time Code using Giotto. IEEE Control Systems Magazine **23**(1) (2003) 50–64
11. Selic, B.: Real-Time Object-Oriented Modeling (ROOM). In: IEEE Real Time Technology and Applications Symposium. (June 1996)
12. Subramonian, V., Gill, C.D., Sanchez, C., Sipma, H.B.: Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time Embedded Systems. In: Proceedings of the 6th Conference on Embedded Software. (October 2006)
13. Engler, D.R., Kaashoek, M.F., O'Toole, J.: Exokernel: An Operating System Architecture for Application-Level Resource Management. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles. (December 1995)
14. Deville, D., Galland, A., Grimaud, G., Jean, S.: Smart Card operating systems: Past, Present and Future. In: Proceedings of the 5th NORDU/USENIX Conference. (February 2003)
15. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System Architecture Directions for Network Sensors. In: Proceedings of the 9th ACM Conference on Architectural Support for Programming Languages and Operating Systems. (November 2000)
16. Levis, P., Lee, N., Welsh, M., Culler, D.E.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: ACM SenSys. (November 2003) 126–137
17. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: Proceedings of the ACM Conference on Programming Language Design and Implementation. (June 2003)
18. Stankovic, J.A., Zhu, R., Poornalingam, R., Lu, C., Yu, Z., Humphrey, M., Ellis, B.: VEST: An Aspect-Based Composition Tool for Real-Time Systems. In: Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium. (May 2003)
19. BIP: `http://www-verimag.imag.fr/~async/BIP/bip.html`
20. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF Toolset. In: School on Formal Methods for the Design of Computer, Communication and Software Systems. (September 2004)
21. Bozga, M., Graf, S., Mounier, L.: IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In: Proceedings of the International Conference on Computer Aided Verification. (July 2002)
22. Gößler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. **55**(1-3) (2005) 161–183

23. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in bip. In: EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software, New York, NY, USA, ACM Press (2007) 11–20
24. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The Algorithmic Analysis of Hybrid Systems. Theor. Comput. Sci. **138**(1) (1995) 3–34
25. Think: `http://think.objectweb.org/`
26. Fractal: `http://fractal.objectweb.org/`
27. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and Its Support in Java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems **36**(11–12) (September 2006) 1257–1284
28. Layaïda, O., Hagimont, D.: Plasma: A component-based framework for building self-adaptive applications. In: Proceedings of the Conference on Embedded Multimedia Processing and Communications. (January 2005)
29. Polakovic, J., Özcan, A.E., Stefani, J.B.: Building Reconfigurable Component-Based OS with THINK. In: Euromicro Conference on Software Engineering and Advanced Applicati ons. (September 2006)
30. Ford, B., Back, G., G., B., Lepreau, J., Lin, A., Shivers, O.: The Flux OSKit: A Substrate for OS and Language Research. In: Proceedings of the 16^{th} ACM Symposium on Operating Systems Principles. (October 1997)
31. Combaz, J., Fernandez, J.C., Lepley, T., Sifakis, J.: QoS Control for Optimality and Safety. In: Proceedings of the 5^{th} Conference on Embedded Software. (September 2005)