Verimag

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

A formal approach to derivation of concurrent implementations in software product lines

S. Yovine, I. Assayad, F.-X. Defaut, M. Zanconi, A. Basu

Verimag Research Report n^o TR-2007-12

December 2008

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

UNIVERSITE JOSEPH FOURIER
CNRS
INPG

A formal approach to derivation of concurrent implementations in software product lines

S. Yovine, I. Assayad, F.-X. Defaut, M. Zanconi, A. Basu

VERIMAG, Centre Equation, 2 Ave. Vignate, 38610 Gières, France. E-mail: `Sergio.Yovine@imag.fr`

December 2008

Abstract

We present a formal tool support for software production lines focused on high-performance real-time applications, where dealing with concurrency at both software and hardware levels is needed. The framework consists in (1) a formal language which provides platform-independent constructs to specify the behavior of an application using an abstract execution model, and (2) a compilation chain for refining the application abstract model into its concrete implementation on a target platform. The prototype JAHUEL is currently being used for developing experimental industrial applications.

Keywords: Software product lines, high-performance applications, concurrency, coordination languages

Reviewers:

Notes: Published as Chapter 11 in *Process Algebra for Parallel and Distributed Processing*, Michael Alexander and William Gardner (Editors), Chapman & Hall/CRC Computational Science Series, 1st edition (December 22, 2008), ISBN-13: 978-1420064865.

How to cite this report:

@techreport { ,
title = { A formal approach to derivation of concurrent implementations in software product lines},
authors = { S. Yovine, I. Assayad, F.-X. Defaut, M. Zanconi, A. Basu},
institution = { Verimag Research Report },
number = {TR-2007-12},
year = { },
note = { }
}

1 Introduction

High-performance real-time embedded applications, such us HDTV, video streaming and packet routing, motivate the use of multicore and multiprocessor hardware platforms offering multiple processing units (e.g., VIPER [15], Philips Wasabi/Cake [35], Intel IXP family of network processors [21]. These architectures provide significant price, performance and flexibility advantages. Besides, such applications are subject to mass customization, as many variations of the same product are delivered to the market with different price, performance, and functionality. The key to mass customization is to capitalize on the commonality and to effectively manage the variation in a software product line [13]. However, in current industrial practices, application requirements and design constraints are spread out and do not easily integrate and propagate through the development process. Moreover, the increasing complexity of applications tends to enlarge the abstraction gap between application description and hardware. Therefore, customization becomes a burdensome and error-prone task. In summary, the complexity of both software and hardware, together with the stringent performance requirements (e.g., timing, power consumption, etc.), makes design, deployment, and customization extremely difficult, leading to costly development cycles which result in products with sub-optimal performances.

During the development cycle of applications for multiprocessors, two models of execution should be distinguished. The first one is the abstract model inherent to the specification of the application, which typically corresponds to logically concurrent activities, with data and control dependencies. The second one is the concrete execution model provided by a particular platform (run-time system and hardware architecture). The customization problem consists in exploiting platform capabilities (e.g., multithreading, pipelinening, dedicated devices, multiprocessors, etc.) to implement the abstract model, or eventually restricting the latter because of constraints imposed by the concrete model (e.g., synchronous communication, shared memory, single processor, bus contention, etc.). In any case, the programmer must handle both types of execution models during the development cycle.

Therefore, there is a need for design flows for software product lines (1) based on formalisms providing appropriate mechanisms for expressing these models, and (2) supported by tools for formally relating them, in order to produce executable code which (a) is correct with respect to application's logic, and (b) ensures non-functional requirements are met on the concrete execution platform.

In the context of high-performance real-time applications, two questions are particularly important: (2) how to map software logical concurrency onto hardware physical parallelism, and (2) how to meet application-level timing requirements with architecture-level resources and constraints. This chapter presents a design flow that provides formal means for coping with concurrency and timing properties from the abstract model all the way down to the concrete one. Current practices to handle these two issues are summarized below.

Run-time libraries and compiler directives. A very common practice consists in using a language with no support for concurrency or time (e.g., C), together with specific libraries or system calls (e.g., POSIX threads or MPI [19]) provided by the underlying run-time system or using compiler directives (e.g., OpenMP [27]).[1] This approach has several inconveniences. First, there is no way to distinguish between abstract and concrete execution models at program level, and therefore, the reason that motivated the programmer's choice (i.e., application design or platform capability) is irrecoverable from program code. This gives rise to a messy development cycle, where application design and system deployment are not handled separately, and application code is customized too early for a specific target, therefore impeding reusability and portability. Second, correctness verification is almost impossible due to system calls (e.g., for threading and resource management [9, 31]).

Domain-specific programming languages. Another practice consists in using a language with a (more or less formal) abstract execution model where time and concurrency are syntactic and semantic concepts (e.g., Lustre [20], Ada [11].) It is entirely the role of the compiler to implement the abstract execution model on the target platform. This approach enhances formal analysis. Nevertheless, these languages rely on a fully automatic implementation phase that makes essential customization issues such as targeting,

[1] Java provides some mechanisms, but they are typically implemented using platform libraries.

platform exploration, and optimization, very hard to achieve. For instance, a typical industrial practice for exploiting multiprocessor architectures for synchronous programs consists in manually cutting the code into pieces, and adding hand-written wrappers. This practice breaks down formal analysis and suffers from the same inconveniences of the library/directives approach. Although there is ongoing work to solve this problem for specific execution platforms (e.g., [12]), there is no attempt neither to provide language support nor to develop a general framework.

Modelling frameworks and architecture description languages. To some extent, some of the above-mentioned problems could be avoided using domain-specific architecture description languages that provide means to integrate software and hardware models (e.g., [8].) Still, in all ADL-based approaches we are aware of, description of the application execution model is tied up to a platform-dependent execution model, which, consequently, is implemented using platform primitives by direct translation of the application code. Model-integrated development [22] also handles requirements composed horizontally at the same level of abstraction. However, it is not well adapted to take care of a primary concern in software product lines which is the vertical propagation of concurrency and timing requirements across different abstraction layers. Platform-based design [32] is a methodology that supports vertical integration, but it is mainly focused on composing functionality while abstracting away non-functional properties. PTOLEMY II [29] is a design framework that supports composition of heterogeneous models of concurrent computation, but it is oriented towards modeling and simulation rather than to application-code synthesis.

Aspect-oriented software development. Aspects could help in bridging the gap between application's specification and the actual platform-specific implementation. However, to our knowledge, current AOP-based approaches require an important programming effort, do not handle timing constraints, and are not specifically focused on code synthesis for different platforms, but are typically used for monitoring and optimization [23].

To overcome the aforementioned problems, we think code-generation tools based on formal languages and models must play the central role of mapping platform independent software into target execution platforms (operating system and hardware), while ensuring at compile time that non-functional requirements provided by system's engineers will be met at run-time. Integrating in a design flow, formal analysis and synthesis techniques for handling non-functional constraints and heterogeneous architectures, is an innovative way to provide correct-by-construction code. This enables code generation for specific platforms (including software-to-processor mapping and scheduling), and platform-independent functional analysis, to be linked together in the same tool-chain without semantic gap.

Such a framework will considerably increase the overall quality of industrial systems designed with these tools, guaranteeing the correctness of the resulting solution. This approach enhances the applicability of formal verification and analysis techniques in industrial design flows, leading to a significant reduction in overall system's validation time. Nevertheless, building representative models that adequately relate functional and non-functional behavior, of both application software and execution platforms, is challenging [34]. Multithreaded software and multicore, multiprocessor architectures bring in additional complexity.

To circumvent this complexity, we propose a design flow consisting of a formal language and its associated compilation chain. The purpose of the language, called FXML [1], is threefold. First, it provides simple and platform-independent constructs to specify the behavior of the application using an abstract execution model. Second, it provides semantic and syntactic support for correctly refining the abstract execution model into the concrete one. Third, the language and the compilation chain are extensible to easily support new concrete execution models, without semantic break-downs. Besides, the language can be used by the programmer to express program structure, functionality, requirements and constraints, as well as by the compiler as a representation to be directly manipulated to perform program analyses and program transformations to generate executable code which achieves application requirements and complies to platform constraints.

On one hand, FXML can be regarded as an algebraic language which provides constructs for expressing concurrency and timing constraints, and means for proving whether a term in the algebra is an "implementation" of another, by term rewriting. On the other, FXML can be seen as a formal coordination language

with general-purpose constructs for expressing concurrency (e.g, par, forall), where coordination is thought as managing dependencies between activities. The main difference with other coordination languages and process algebras (see [28] and [6] for comprehensive surveys) is that FXML (1) can express rich control and data precedence constraints, and (2) can be gradually extended with more concrete constructs in order to provide synchronization, communication, and scheduling mechanisms for implementing the abstract behavior. Moreover, by design, FXML and its code-generation tool-suite JAHUEL [1], provide an extensible and customizable software production line oriented towards generating code for multiple platforms via domain-specific semantics-preserving syntactic transformations.

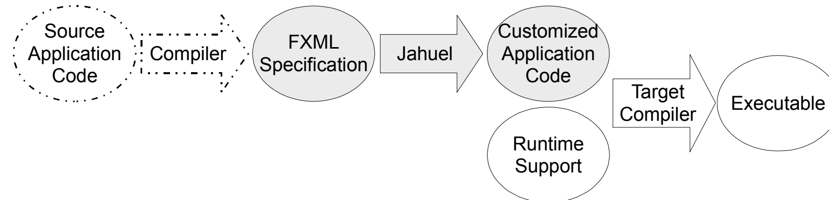


Figure 1: Design flow.

Chapter outline. This chapter presents the components of the design flow shown in Fig. 1. Gray-colored components constitute the kernel of the design flow. Components in dotted lines are not mandatory, i.e., they may or may not exist in a specific tool based on FXML/JAHUEL. Non-colored elements correspond to platform-dependent components. Sec. 2 and Sec. 3 present the gray-colored components. Sec. 2 gives the syntax and semantics of the basic language. FXML can be used as a front-end specification language or obtained from an application source code in some other language. The role of FXML as formal specification language is illustrated in Sec. 2 with a simple Writer-Reader program and the Smith-Waterman local sequence matching algorithm [16]. Sec. 3 overviews the code-generation approach for FXML implemented in the compilation chain JAHUEL. The Writer-Reader case-study is used to exemplify how C code is generated from FXML for several target run-time platforms, such as *pthreads* and OpenMP [27]. Sec. 4 discusses two applications of FXML in the role of "intermediate" representation formalism, rather than top-level specification language. The first one presents the integration of FXML and JAHUEL in a C-based compilation tool-suite, where the input language is C extended with pragmas in FXML. The second application consists in using FXML as a formalism for giving semantics to the programming language StreamIt [36]. As an outcome, JAHUEL can be used to generate code for multiple run-time targets from the FXML-based representation of annotated C or StreamIt programs. Sec. 5 is about another use of JAHUEL as tool for generating customized code. It explains how to produce component-structured code from FXML, by providing a translation into BIP [4]. This transformation enables, for instance, formal verification via the IF framework [10] and execution on sensor networks [5].

2 The language: FXML

FXML [1] is a language for expressing concurrency, together with control and data dependencies which can be annotated with properties to restrict parallelism because of timing or precedence constraints.

2.1 Abstract syntax

This section overviews the abstract syntax of FXML used hereinafter. It is out of the scope of this chapter to explain the full concrete syntax of FXML *pnodes* used by JAHUEL, which is defined as an XML schema.

The *body* of an FXML specification is composed of blocks called *pnodes*.[2]

[2]The term *pnode* stands for "presentation node". This notion comes from model theory: a *pnode* "presents" an abstract execution.

Basic pnodes:

- `nil` denotes an *empty* set of executions.
- Let X be the set of variables. Variables store values from a set V. An *assignment* α has the form $x_0 = \zeta(x_1, \ldots, x_n)$, where $x_i \in X, i \in [0, n]$, and $\zeta : V^n \to V$ is a computable function. We write α_i for x_i.

  Variables are assumed to be assigned in static single assignment (SSA) form, that is, there is only one assignment statement for each variable.
- `legacy` B declares a block B of *legacy code* written in, e.g., C, C++, Java, etc.

Conditional pnodes: `if` $\zeta(x_1, \ldots, x_n)$ `then` p `else` q, where p and q are *pnodes*, and $\zeta : V^n \to \mathbb{B}$ is a boolean function.

Sequential composition: `seq` $p_1 \ldots p_n$.

For/while loops:
`for(i` $= init(x_1, \ldots, x_n)$`;` $test$`(i);i` $= inc$`(i))`$\langle$`per=P`$\rangle$ p, and `while(`$test(x_1, \ldots, x_n)$`)`$\langle$`per=P`$\rangle$ p, express iterations: `i` is the iteration variable, $init : V^n \to \mathbb{N}$ is a computable function that gives the initial value of `i`, $inc : \mathbb{N} \to \mathbb{N}$ is the increment function, and $test : \mathbb{N} \to \mathbb{B}$ is a boolean function that defines the looping condition. Variable `i` is assumed not to be modified in p.

The optional declaration `per=P` states that the loop is *periodic* with period P, that is, there is a loop iteration every P. The execution time of the body of the loop has to be attached to p. Different loop iterations may have different execution times, as long as they are consistent with the loop period P and the execution time interval attached to p (see below).

Parallel composition: `par` $p_1 \ldots p_n$.

Forall loops: `forall(i` $= init(x_1, \ldots, x_n)$`;` $test$`(i);i` $= inc$`(i))` p where `i`, $init$, inc and $test$ are as for `for`-loops specifies parallel executions of p.

Labeling: `L:` p is a *pnode*.

Dependencies: `dep`$\{\langle$`[`a`,`b`]`$\rangle\langle type\rangle$`L`$_1 \to$ `L`$_2\}p$, with $a, b \in \mathbb{N}$, specifies a *dependency* between occurrences of two descendants `L`$_i : p_i, i \in [1, 2]$, of p. The optional declaration $type$ annotates the dependency with a type:

- The default type is `weak` and means that *at least* one occurrence of p_1 should precede *every* occurrence of p_2.
- The type `strong` means that *every* occurrence of p_1 should precede *at least* an occurrence of p_2.
- The type $(k, f(k))$ means that the $f(k)$-th occurrence of p_2 should be preceded by the k-th occurrence of p_1.

The optional declaration `[`a`,`b`]` specifies that the timing distance between the corresponding occurrences of p_1 and p_2 falls in the interval $[a, b]$.

Execution times: `p[a,b]`, $a, b \in \mathbb{N}$, means that the execution time of p is in the interval $[a, b]$.

Example 2.1 (Writer/Reader) *The FXML specification of a simple program where a reader reads and prints out a value written by a writer is as follows:*

```
dep [0,15] W -> R
par
  Writer:
    seq
      p = 0 [0,1]
      while(true) per=10
        W: seq {
          x = p
          p = p + 1
        } [0,1]
  Reader:
    while(true)
      R: seq {
        y = x
        legacy{ printf("%d\n", y); }
      } [0,1]
```

The declaration `dep W → R` *declares the dependency between occurrences of* pnodes *labelled* `W`*, in* `Writer`*, and* `R`*, in* `Reader`*. This dependency comes from the fact that variable* `x` *must have some value by the time* `Reader` *uses it. Since no type is attached to the dependency, it follows that it is of the default type* `weak`*. This means that written values may not be read or read more than once, but* `x` must *have been written* at least once *by* `Writer` *before it is first read by* `Reader`*.*

This default behavior can be strengthen with a `strong` *type declaration to require that every written value* must *be read* at least once*. To specify that the value written in the* i*-th iteration of the* `Writer`*'s loop must be used in the* i*-th iteration of the* `Reader`*'s loop, the declaration* `(i,i)` *has to be added to the dependency* `dep W → R`*.*

The period declaration attached to the `while` *loop of the* `Writer` *states that the body of the loop is executed periodically every* 10 *time units. The interval* `[0,15]` *in* `dep [0,15] W→R` *serves for specifying a* freshness constraint*: the value of* `x` *cannot be read if the time distance between the write and read operations is greater than 15 time units. The execution times of* `p = 0`*,* `W` *and* `R` *are specified to be in the interval* $[0, 1]$[3]*.*

Example 2.2 (Smith-Waterman) *The Smith-Waterman [16] local sequence matching algorithm consists of computing the elements of a* $N + 1$ *by* $M + 1$ *matrix* `A`*, from two strings* `S1` *and* `S2` *of lengths* $N + 1$ *and* $M + 1$*, respectively.*

In FXML, it can be expressed as follows:

```
dep ((i,j), (i+1,j)) LA -> LX
dep ((i,j), (i,j+1)) LA -> LY
dep ((i,j), (i+1,j+1)) LA -> LZ
seq
  forall(j = 0; j <= M; j+1)
    forall(i = 0; i <= N; i+1)
      LI: A[i][j] = 0
  forall(j = 1; j <= M; j+1)
    forall(i = 1; i <= N; i+1)
      seq
```

[3]When the execution time of a *pnode* is not given, it means it can take an arbitrary amount of time to execute which is consistent with all timing constraints.

```
par {
  LX: X = A[i-1][j] + 2
  LY: Y = A[i][j-1] + 2
  LZ: Z = A[i-1][j-1] + (S1[i]==S2[j]?-1:1)
}
LA: A[i][j] = MIN(0, X, Y, Z)
```

The dependencies state that the computation of each element (i, j) *is a function of its "North"* $(i-1, j)$*, "West"* $(i, j-1)$*, and "NorthWest"* $(i-1, j-1)$ *neighbors* `A`*.*

Hereinafter, the keyword `seq` will be omitted in the examples.

2.2 Semantics

2.2.1 Definitions

Before giving semantics to FXML specifications, let us introduce some definitions.

Indexed assignments: An index is a list I of natural numbers and labels. $\langle \ell_1, \ldots \rangle$ denotes the list consisting of elements $\ell_1, \ldots$, and $\circ$ denotes concatenation of lists. An *indexed* assignment is denoted α^I. A set of indexed assignments is denoted $\mathcal{A}$.

Timing: Time is modeled with a timing function $\tau : \mathcal{A} \to \mathbb{R}^+ \times \mathbb{R}^+$. We write, $\tau^b(\alpha^I) = \pi_1(\tau(\alpha^I))$, and $\tau^e(\alpha^I) = \pi_2(\tau(\alpha^I))$, which denote respectively the beginning and ending times of assignment α^I.

τ satisfies $\mathcal{A}$, denoted $\tau \models \mathcal{A}$, iff for each $\alpha^I \in \mathcal{A}$, $\tau^b(\alpha^I) \leq \tau^e(\alpha^I)$.

Dependencies: Let $Out = \{x \in X \mid \exists \alpha^I \in \mathcal{A} : \ x = \alpha_0\}$ be the set of variables assigned in $\mathcal{A}$.

- The relation $\xrightarrow{d} \subseteq \mathcal{A} \times \mathcal{A}$ models data dependencies: for all $\beta^J \in \mathcal{A}$, for all β_j, if $\beta_j \in Out$, then there exists a *unique* $\alpha^I \in \mathcal{A}$ s.t. $\alpha_0 = \beta_j$ and $\alpha^I \xrightarrow{d} \beta^J$. We write $\alpha^I \xrightarrow{\beta_j} \beta^J$ as a shorthand for $\alpha^I \xrightarrow{d} \beta^J \wedge \ \alpha_0 = \beta_j$.

  $\tau \models \xrightarrow{d}$ iff $\forall \alpha^I, \beta^J \in \mathcal{A} : \ \alpha^I \xrightarrow{d} \beta^J \implies \tau^e(\alpha^I) \leq \tau^b(\beta^J)$.

- The relation $\xrightarrow{;} \subseteq \mathcal{A} \times \mathcal{A}$ gives an order between indexed assignments in $\mathcal{A}$, thus modeling dependencies derived from the sequential composition.

  $\tau \models \xrightarrow{;}$ iff $\forall \alpha^I, \beta^J \in \mathcal{A} : \ \alpha^I \xrightarrow{;} \beta^J \implies \tau^e(\alpha^I) \leq \tau^b(\beta^J)$.

We define $\longrightarrow = \xrightarrow{d} \cup \xrightarrow{;}$. $\tau \models \longrightarrow$ iff $\tau \models \xrightarrow{d}$ and $\tau \models \xrightarrow{;}$.

Valuations: $\mathcal{A}$ can be seen as a family $\{\mathcal{A}^n\}_{n \in \mathbb{N}}$ of sets of indexed assignments, where $\mathcal{A}^n$ contains only indexed assignments α^I of the form $x_0 = \zeta(x_1, \ldots, x_n)$. Let $\Upsilon = \{\nu^n\}_{n \in \mathbb{N}}$ be a family of $\mathbb{N}$-indexed functions, with $\nu^n : \mathcal{A}^n \to V^{n+1}$, $\nu^n(\alpha^I) = (v_0, v_1, v_2, \ldots, v_n)$, $v_0 = \zeta(v_1, v_2, \ldots, v_n)$. We write $\nu^n(\alpha^I)_i$ for v_i.

$\Upsilon \models \xrightarrow{d}$ iff $\forall (\alpha^I, \beta^J) \in \mathcal{A}^n \times \mathcal{A}^m$: $\alpha^I \xrightarrow{\beta_j} \implies \nu^m(\beta^J)_j = \nu^n(\alpha^I)_0$.

Executions: An execution e is a tuple $(X, \mathcal{A}, V, \xrightarrow{d}, \xrightarrow{;}, \tau, \Upsilon)$, such that $\tau \models \mathcal{A}$, $\tau \models \longrightarrow$, $\Upsilon \models \xrightarrow{d}$.

Timing constraints: The starting and ending time of e are, respectively: $\tau^b(e) = \min_{\alpha^I \in A_e} \tau^b(\alpha^I)$, and $\tau^e(e) = \max_{\alpha^I \in A_e} \tau^e(\alpha^I)$.

Subexecutions: f is a subexecution of e, denoted $f \subseteq e$, iff $\mathcal{A}_f \subseteq \mathcal{A}_e$, $\tau_f = \tau_e \restriction_{\mathcal{A}_f}$, $\nu^n_f = \nu^n_e \restriction_{\mathcal{A}^n_f}$, $\xrightarrow{d}_f = \xrightarrow{d}_e \restriction_{\mathcal{A}_f \times \mathcal{A}_f}$, $\xrightarrow{;}_f = \xrightarrow{;}_e \restriction_{\mathcal{A}_f \times \mathcal{A}_f}$, where $\restriction$ is "restricted to".

Partitions: A partition of e, denoted $\&_{i\in I}e_i$, is such that for all $i \in I$, e_i is a non-trivial subexecution of e, and for all $i \neq j$, $\mathcal{A}_{e_i} \cap \mathcal{A}_{e_j} = \emptyset$, and $\bigcup_{i\in I} \mathcal{A}_{e_i} = \mathcal{A}$.

A *sequential* partition of e, denoted $;_{i\in I}e_i$, is a partition such that $\forall \alpha^I \in \mathcal{A}_{e_i}, \beta^J \in \mathcal{A}_{e_j} : i < j \implies \alpha^I \xrightarrow{;}_e \beta^J$.

Dependencies: For $e_1, e_2 \subseteq e$, $e_1 \longrightarrow_e e_2$ iff $\forall \alpha^I \in \mathcal{A}_{e_1}, \beta^J \in \mathcal{A}_{e_2} : \alpha^I \longrightarrow_e \beta^J$.

Indexed executions: The indexing of e with K is the execution e^K where $\mathcal{A}_{e^K}$ is defined such that for all $\alpha^I \in \mathcal{A}_e : \alpha^{K\circ I} \in \mathcal{A}_{e^K}$. We write $e \cong^K e^K$ to denote that e is the same execution as e^K modulo the indexing with K.

2.2.2 Semantic rules

The semantics of an FXML specification is a set of executions. We use an algebraic definition of the semantics [17]. If p is a *pnode* and e is an execution, $e \models p$ means that e is an execution for p. The semantics of p is $[\![p]\!] = \{e \mid e \models p\}$.

Nil: The semantics of `nil` is the empty execution $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

Assignments: $e \models \alpha$ iff $\mathcal{A}_e = \{\alpha^I\}$ for some index I.

Conditional statements: $e \models \texttt{if}\, \zeta(x_1, \ldots, x_n)\, \texttt{then}\, p\, \texttt{else}\, q$ iff $e = e_1; e_2$ s.t. $e_1 \models \zeta(x_1, \ldots, x_n)$, with $\mathcal{A}_{e_1} = \{\alpha^I\}$, and $e_2 \models p$ if $\nu^n_{e_1}(\alpha^I) =$ *true*, else $e_2 \models q$.

Sequential composition: $e \models \texttt{seq}\, p_1 \ldots p_n$ iff $e = ;_{i\in[1,n]}e_i$, such that $e_i \models p_i$.

Iterations: Let $\mathcal{K} = \{k_j\}_{j\in\mathcal{J}}$ (with $\mathcal{J}$ a finite or infinite interval of $\mathbb{N}$) be the indexed set of the values taken by the iteration variable `i`. $\mathcal{K}$ is defined by *inc*, which is increasing, that is: $i < j \implies k_i \leq k_j$, for all $k_i, k_j \in \mathcal{K}$.

- The semantics of `for`-loops is the set of executions defined as follows:
  $e \models \texttt{for()}\; p$ iff $e = ;_{j\in\mathcal{J}} {f_j}^{\langle j\rangle}$, where $f_j \models p$, $j \in \mathcal{J}$, and for every $\alpha^I \in \mathcal{A}^m_{f_j}$ (for any $m \in \mathbb{N}$): $\alpha^I_l = \texttt{i} \implies \nu^m_{f_j}(\alpha^I)_l = k_j$ (that is, the value of the iteration variable `i` is equal to k_j in f_j).

  If the optional declaration `[per=P]` is present, e is such that: for all $j \in \mathcal{J}$, $[\,\tau^b({f_j}^{\langle j\rangle}), \tau^e({f_j}^{\langle j\rangle})\,] \subseteq [(j-1)P, jP\,)$.

- For `while`, assignments are indexed using a hidden variable j, whose values are $0, \ldots, N-1$, when the loop stops after N turns:
  $e \models \texttt{while}(test(x_1, \ldots, x_n))\, p$ iff $e = ;_{j\in[0,N-1]} \left(c_j\, ;\, {f_j}^{\langle j\rangle}\right)\, ;\, c_N$, where $c_j \models test(x_1, \ldots, x_n)$, $j \in [0, N]$, $f_j \models p$, $j \in [0, N-1]$, and the conditions evaluate to *true* in c_j, $j \in [0, N-1]$, and to *false* in c_N. The semantics of a non-terminating loop is an infinite execution where conditions evaluate to *true* for all c_j.

  If the period declaration is given, the semantics is similar to `for`-loop periods.

Parallel composition: $e \models \texttt{par}\, p_1 \ldots p_n$ iff $e = \&_{i\in[1,n]}e_i$, such that $e_i \models p_i$.

Proposition 2.1 *Parallel composition is commutative and associative.*

Proposition 2.2 $[\![\texttt{seq}\, p_1 \ldots p_n]\!] \subseteq [\![\texttt{par}\, p_1 \ldots p_n]\!]$.

Forall loops: Let $\mathcal{K} = \{k_j\}_{j\in\mathcal{J}}$ be the indexed set of indices defined by inc. $e \models \texttt{forall()}\ p$ iff $e = \&_{j\in\mathcal{J}} f_j{}^{\langle j\rangle}$, where $f_j \models p$, $j \in \mathcal{J}$, and for every $\alpha^I \in \mathcal{A}^m_{f_j}$ (for any $m \in \mathbb{N}$): $\alpha^I_l = \texttt{i} \implies \nu^m_{f_j}(\alpha^I)_l = k_j$.

Proposition 2.3 $[\![\mathit{for}(i = init(x_1,\ldots,x_n); test(i); i = inc(i))\langle per{=}P\rangle\ p]\!] \subseteq$
$[\![\mathit{forall}(i = init(x_1,\ldots,x_n); test(i); i = inc(i))\langle per{=}P\rangle\ p]\!]$.

Proposition 2.4 $[\![\mathit{par}\, p_1 \ldots p_n]\!] \overset{\simeq\langle i\rangle}{=} [\![\mathit{forall\,(i{=}1;i{<=}n;i{+}1)}p]\!]$, *where* $\overset{\simeq\langle i\rangle}{=}$ *means equal modulo the indexing given by the iteration variable* $\texttt{i}$.

Dependencies: $e \models \texttt{dep}\{\langle\texttt{[a,b]}\rangle\langle type\rangle \texttt{L}_1 \to \texttt{L}_2\}p$, iff $e \models p$, and

- $type$ = `weak`: for every $e_2 \subseteq e$, s.t. $e_2 \models \texttt{L}_2 : p_2$, there exists $e_1 \subseteq e$, s.t. $e_1 \models \texttt{L}_1 : p_1$ and $e_1 \longrightarrow_e e_2$.
- $type$ = `strong`: e satisfies the condition above and for every $e_1 \subseteq e$, s.t. $e_1 \models \texttt{L}_1 : p_1$, there exists $e_2 \subseteq e$, s.t. $e_2 \models \texttt{L}_2 : p_2$ and $e_1 \longrightarrow_e e_2$.
- $type$ = $(j, f(j))$: for all $j \in \mathcal{J}$, if $e_1^{\langle j\rangle} \subseteq e$ is s.t. $e_1^{\langle j\rangle} \models \texttt{L}_1 : p_1$, and $e_2^{\langle f(j)\rangle} \subseteq e$ is s.t. $e_2^{\langle f(j)\rangle} \models \texttt{L}_2 : p_2$, then $e_1^{\langle j\rangle} \longrightarrow_e e_2^{\langle f(j)\rangle}$.

If the optional timing interval `[a,b]` is present, e is such that: for all $e_i \subseteq e$, $e_i \models p_i$, $i = 1, 2$, $e_1 \longrightarrow_e e_2 \implies \tau^b(e_2) - \tau^e(e_1) \in [a, b]$.

Proposition 2.5 *Let* q *be* $\mathit{dep}\{\langle\texttt{type}\rangle L_1 \to L_2\}p$. *We have that:*
$[\![q[\texttt{type} := (i,i)]]\!] \subseteq [\![q[\texttt{type} := \mathit{strong}]]\!] \subseteq [\![q[\texttt{type} := \mathit{weak}]]\!]$.

Proposition 2.6 $\forall [a', b'] \subseteq [a, b]$:
$[\![\mathit{dep}\{[a',b']L_1 \to L_2\}p]\!] \subseteq [\![\mathit{dep}\{[a,b]L_1 \to L_2\}p]\!] \subseteq [\![\mathit{dep}\{L_1 \to L_2\}p]\!]$.

Execution times: $e \models p\,[a,b]$ iff $e \models p$ and $\tau^e(e) - \tau^b(e) \in [a, b]$.

Proposition 2.7 $\forall [a', b'] \subseteq [a, b]$: $[\![p\,[a',b']]\!] \subseteq [\![p\,[a,b]]\!] \subseteq [\![p]\!]$.

Example 2.3 (Writer/Reader) *Fig. 2 shows examples of executions of Ex. 2.1 for different types of dependencies between* pnodes `W` *and* `R`*: (a)* `weak`*, (b)* `strong`*, and (c)* (i, i)*. Non-labelled assignments are not shown. The vertical placement of* `W`*'s and* `R`*'s corresponds to their occurrence in global time, which proceeds from top to bottom. Recall that, by Prop. 2.5, any execution of type* (i, i) *is also* `strong`*, and any* `strong` *is also* `weak`*.*

The executions of pnodes `Writer` *and* `Reader` *are total orders of the form* $\texttt{W0} \xrightarrow{;} \texttt{W1} \cdots$ *and* $\texttt{R0} \xrightarrow{;} \texttt{R1} \cdots$*, respectively, which are consistent with the timing constraints (*`Writer`*'s loop period and execution times). Each execution of the composed system contains the union of the executions of* pnodes `Writer` *and* `Reader` *which are consistent with the dependency declaration* `dep [0,15] W` $\to$ `R`*, together with precedences added by it. For instance, in the execution shown in (a)-left, the value written by* `W0` *is read by* `R0` *and* `R1`*. This means that* `R0` *and* `R1` started *at most 15 time units after* `W0` terminated.

However, the occurrence of `W1` *between* `R0` *and* `R1` *does not prevent the value written by* `W0` *to be read twice. This execution models a behavior that may occur in a concrete implementation of this program where values are buffered. We will see later in Sec. 3 how such implementation can be derived from this FXML specification.*

Example 2.4 (Smith-Waterman) *Fig. 3 shows a part of the model of the Smith-Waterman program (Ex. 2.2).*

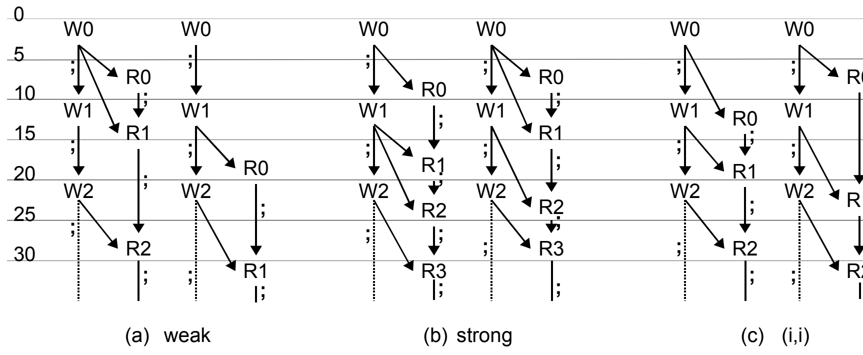


Figure 2: Examples of executions of Writer-Reader.

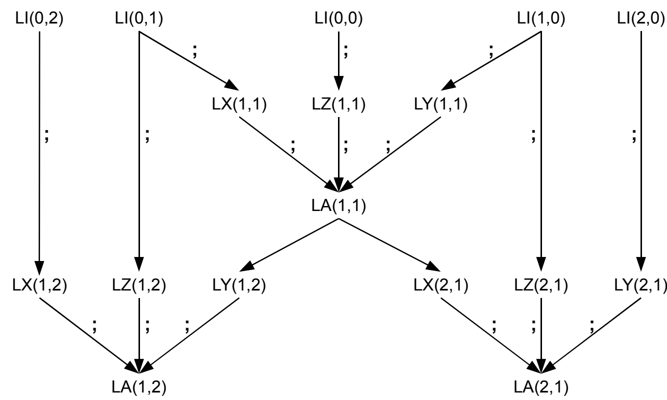


Figure 3: Examples of executions of Smith-Waterman.

3 The code generation chain

3.1 Compilation approach

Compiling an FXML specification consists in transforming it until actual executable code for a specific platform could be generated. Let $\mathcal{L}$ denote a language. Concretely, $\mathcal{L}$ is given by an XML schema, where each element definition has an associated type.

A *transformation* from $\mathcal{L}$ to $\mathcal{L}'$ is an injective map $\phi : \mathcal{L} \rightarrow \mathcal{L}'$, that is, every element of the XML schema $\mathcal{L}$ is in the set of elements $\mathcal{L}'$. Let $E_{\mathcal{L}}$ be the set of executions of type $\mathcal{L}$, and $F_\phi : E_{\mathcal{L}'} \rightarrow E_{\mathcal{L}}$ be the "forgetting" function that forgets any information specific to executions of type $\mathcal{L}'$. $\phi : \mathcal{L} \rightarrow \mathcal{L}'$ satisfies that for all executions $e' \models_{\mathcal{L}'} \phi(p)$ it follows that $F_\phi(e') \models_{\mathcal{L}} p$.

The compilation process is a sequence of transformations $\mathcal{L}_0 \mapsto^* \mathcal{L}_0 \mapsto \mathcal{L}_1 \mapsto^* \ldots \mathcal{L}_n$, where $\mathcal{L}_0$ is basic FXML. $\mathcal{L}_i \mapsto^* \mathcal{L}_i$ is a sequence of transformations from $\mathcal{L}_i$ to $\mathcal{L}_i$, resulting in a sequence of programs $p_i^1 \ldots p_i^n$, such that $[\![p_i^{k+1}]\!] \subseteq [\![p_i^k]\!]$. Examples of transformations from $\mathcal{L}_0$ to $\mathcal{L}_0$ are: replacing `weak` dependencies by `strong` or (i,i) ones; `par` and `forall` by `seq` and `for`, resp., etc.

Transformations from $\mathcal{L}_0$ to $\mathcal{L}_0$: Let us define the following transformations:

- $\phi_;$ s.t. $\phi_;(p) = \texttt{seq}\, p_1 \ldots p_n$, if $p = \texttt{par}\, p_1 \ldots p_n$, else $\phi_;(p) = p$.
- $\phi_{\texttt{for}}$ s.t. $\phi_{\texttt{for}}(p) = \texttt{for(...)}q$, if $p = \texttt{forall(...)}q$, else $\phi_{\texttt{for}}(p) = p$.
- $\phi_{\texttt{strong}}$ s.t. $\phi_{\texttt{strong}}(p) = p[^{\texttt{weak}}/_{\texttt{strong}}]$, if $p = \texttt{dep}\{\texttt{weak L}_1 \rightarrow \texttt{L}_2\}\ q$, else $\phi_{\texttt{strong}}(p) = p$.
- $\phi_{(i,i)}$ s.t. $\phi_{(i,i)}(p) = p[^{type}/_{(i,i)}]$, with $type = \texttt{weak}$ or $type = \texttt{strong}$, if $p = \texttt{dep}\{type\ \texttt{L}_1 \rightarrow \texttt{L}_2\}\ q$, else $\phi_{(i,i)}(p) = p$.

Proposition 3.1 *For all pnode p, $[\![\phi_*(p)]\!] \subseteq [\![p]\!]$, with $\phi_* \in \{\phi_;, \phi_{for}, \phi_{strong}, \phi_{(i,i)}\}$.*

Transformations of the form $\mathcal{L}_i \mapsto \mathcal{L}_{i+1}$ *add* information not expressible in $\mathcal{L}_i$. An example consists in inserting communication and synchronization mechanisms (e.g., semaphores, queues, etc.) to ensure dependencies are met.

3.2 Tool: JAHUEL

A sequence of transformations *defines* the steps to be carried out to perform a specific *customization* of the product decided by the designer. The goal is to have a tool which (1) provides the appropriate transformations, and (2) *automatically* performs such a specified sequence of them. Moreover, the tool must be extensible, in the sense that it should be able to *add* new transformations to it.

For this purpose, we have developed JAHUEL, an FXML-based code-generation chain, constructed to be easily extended to cope with new execution models, by extending the basic FXML XML-schema, and by adding transformations.

JAHUEL is implemented in Java, using the Java Architecture for XML Binding (JAXB) API [4], to manipulate XML documents. FXML and its extensions are defined by XML schemes. Using JAXB, each language is bound to a Java class which provides the appropriate data representation and manipulation methods. Transformations are implemented on top of these Java classes.

The architecture and flow of the implementation of a transformation in JAHUEL is shown in Fig. 4. The flow of a transformation $\phi : \mathcal{L} \rightarrow \mathcal{L}'$ is as follows. The input specification in $\mathcal{L}$ is given as an XML file according to the $\mathcal{L}$ schema. The XML input file is *unmarshalled* to obtain its internal representation as a Java object, to which the method implementing the transformation is applied. The result is an object which is then *marshalled* into the XML output file according to $\mathcal{L}'$ schema, which can be used by a subsequent transformation. This strategy ensures traceability of implementation choices. The ultimate code generation phase for the target platform is done via a stylesheet. A *configuration* of JAHUEL consists in applying a sequence of transformations. This is done through a configuration file.

[4]http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

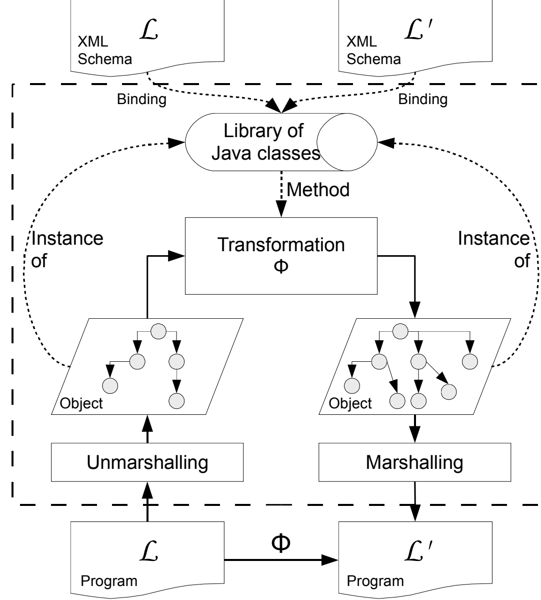


Figure 4: Schematic architecture and flow of the implementation of a transformation in JAHUEL.

Currently, JAHUEL provides some general transformations which can be customized for different execution and simulation platforms. We have instantiated them to generate code for, e.g., Java, C with *pthreads*, SystemC, and P-Ware [3]. The compilation chain is indeed to be instantiated with the sequence of transformations to be applied.

3.3 Examples of code generation

We illustrate here the use of JAHUEL in the Writer-Reader (Ex. 2.1) and Smith-Waterman (Ex. 2.2) examples.

3.3.1 Generic transformations

In order to generate executable code, one customization decision that needs to be made is to determine the active *components* of the system, which will become processes, threads, etc., depending on the target programming language and execution platform. For instance, in the Writer-Reader example, it is natural to consider *pnodes* `Writer` and `Reader` as components.

Components: Let ϕ_c such that $\phi_c(p, \mathtt{L}) = \mathtt{L:component}\ q$, if $p = \mathtt{L:}q$, else $\phi_c(p, \mathtt{L}) = p$. We define: $[\![\mathtt{component}\ q]\!] = [\![q]\!]$. Trivially, $F_{\phi_c}([\![\phi_c(p)]\!]) = [\![p]\!]$.

Let w and r be *pnodes* `Writer` and `Reader`, respectively. Then, ϕ_c allows transforming w and r as follows:

$$\begin{aligned}\phi_c(w, \mathtt{Writer}) &= \mathtt{Writer:component}\ w\\ \phi_c(r, \mathtt{Reader}) &= \mathtt{Reader:component}\ r\end{aligned}$$

Besides, most synchronization mechanisms have the same kind of behavior: a component implementing a *pnode* will *wait for* some condition to hold before executing a piece of code involved in a dependency, and it will *signal* the other activities concerned by the dependency that something has happened after executing it.

Synchronization: Let ϕ_{wn} such that $\phi_{wn}(\texttt{L}:q) = \texttt{seq}\{\texttt{waitfor L}:q\ \texttt{signal}\}$ if q is a descendant of some p with $\texttt{dep}\{\texttt{W} \rightarrow \texttt{L}\}p$ or $\texttt{dep}\{\texttt{L} \rightarrow \texttt{R}\}p$, else $\phi_{wn}(p) = p$. We define: $[\![\phi_{wn}(p)]\!] = [\![p]\!]$. Trivially, $F_{\phi_{wn}}([\![\phi_{wn}(p)]\!]) = [\![p]\!]$.

Let us first consider the Writer-Reader specification without timing constraints. We will take care of timing constraints later. The transformed specification obtained by applying $\phi_{\texttt{c}}$ and ϕ_{wn}, is as follows:

```
dep W -> R
par
  Writer: component
    p = 0
    while(true)
      waitfor
      W: { x = p
           p = p + 1 }
      signal
  Reader: component
    while(true)
      waitfor
      R: { y = x
           legacy{ printf("%d\n", y); } }
      signal
```

These “generic” transformations have no effect on the semantics, but only annotate the specification with useful information for easing further ones.

3.3.2 Threads with lock, unlock, wait and notify primitives

JAHUEL provides a transformation of an FXML specification into a C program where concurrency is implemented using the *pthreads* library. Roughly speaking, it works as follows. Suppose now we would like to generate code for an execution platform providing *threads*, *mutexes*, and *condition variables*, such as the *pthreads* library. The generated code will consist of two threads, sharing variable `x`. Concurrent accesses to `x` must be ensured to be mutually exclusive, and for a `weak` dependency, `x` must be written at least once by `Writer`, before `Reader` could read it. In order to do this, basic FXML is extended with the appropriate constructs to handle theses notions, independently of the actual API provided by the run-time. The transformed specification looks as follows:

```
dep W -> R
par
  Writer: thread
    p = 0
    while(true)
      mcx.lock
      W: { x = p
           p = p + 1 }
      mcx.notify(1)
  Reader: thread
    while(true)
      mcx.wait(1)
      R: { y = x
           legacy{ printf("%d\n", y); } }
      mcx.unlock
```

The statement `thread` specifies that component `Writer` will later become a *thread*. The translation of this statement into actual C code with *pthreads* requires a rather involved transformation which is out of the scope of this chapter.

`mcx` is a structure composed of a *mutex* `mx` and a *condition variable* `cx` to protect accesses to the shared variable `x`. In *pnode* `Writer`, `waitfor` is implemented by `mcx.lock`, since a `weak` dependency does not require `Writer` to wait, but the implementation of FXML variable `x` as a shared C variable imposes mutual exclusion. `mcx.lock` can be directly translated into the corresponding *pthreads* operation, e.g., `pthread_mutex_lock(&(mcx.mx))`.

The code generated for the notification is `mcx.notify(1)`, which consists in setting the value of a flag attached to the condition variable to 1. The implementation of this statement in *pthreads* looks like:

```
/* The mutex has already been acquired */
mcx.b=1;
pthread_cond_signal(&(mcx.cx));
pthread_mutex_unlock(&(mcx.mx));
```

In `Reader`, the `waitfor` statement is translated into `mcx.wait(1)`, which consists in waiting for the condition variable to be equal to 1. It can be implemented in *pthreads* as follows:

```
pthread_mutex_lock(&(mcx.mx));
while(mcx.b==0)  pthread_cond_wait(&(mcx.cx), &(mcx.mx));
```

The `signal` statement is translated into `mcx.unlock`, since no notification is required, and implemented as `pthread_mutex_unlock(&(mcx.mx))`.

Proposition 3.2 *Let ϕ_{luwn} be the transformation that translates* `waitfor` *and* `signal` *into* `lock`, `unlock`, `wait`, *and* `notify`*. For all p, $F_{\phi_{luwn}}([\![\phi_{luwn}(p)]\!]) \subseteq [\![p]\!]$.*

3.3.3 Threads communicating through buffers

JAHUEL provides another transformation that allows implementing FXML variables as buffers. This leads to another extension of FXML. For instance, the transformed Writer-Reader specification is as follows:

```
dep W -> R
par
  Writer: thread
    p = 0
    while(true)
      W: { xbuf.put(p)
           p = p + 1 }
  Reader: thread
    while(true)
      R: { y = xbuf.get()
           legacy{ printf("%d\n", y); } }
```

The FXML variable `x` is implemented by a shared buffer `xbuf`. Writing and reading `x` become `xbuf.put(`e`)` and `xbuf.get()`, respectively.

The actual implementation (e.g., array, queue, socket, etc.) and size are to be determined later, by a subsequent transformation. The abstract behavior depends on the type of the dependency. This is captured in the specification by attaching the buffer to the `dep` declaration:

- For a `weak` dependency, the buffer is only requested to produce values in a way consistent with the order of writes and reads, that is, the value returned by the $i+1$-th call to `get()` *must not* have been put *before* the value returned the i-th time.
- For `strong`, `get()` is required to deliver *all* written values. This imposes a fairness constraint, which can be realized, for instance, by implementing `get()` so as to return the value inserted *right after* the one delivered in the previous call, if it exists, otherwise the last returned one.
- The (i,i) case can be implemented with a blocking FIFO buffer.

Proposition 3.3 *Let ϕ_{buf} be the transformation that translates* `waitfor` *and* `signal` *using operations on buffers. For all p, $F_{\phi_{buf}}([\![\phi_{buf}(p)]\!]) \subseteq [\![p]\!]$.*

3.3.4 Translation into OpenMP

The definition of FXML has been actually inspired by OpenMP [27], and therefore, there is a natural translation into it. The basic idea consists in encapsulating sequential *pnodes* in sections, and compiling `par` and `forall` into *parallel* sections and for-loops respectively. However, since intra-forall dependencies are not allowed, it is necessary to perform the adequate transformations before to rule them out.

Example 3.1 (Smith-Waterman) *The dependencies in the Smith-Waterman program characterize the well-known* wavefront-*like scheduling where the matrix elements on a diagonal are computed in parallel, using elements on matrix diagonals previously computed: all elements (i, j) such that $i + j = d$, for all $d \in [2, M + N]$, can be simultaneously computed, as they only depend on elements (i', j'), with $i' + j' = d' < d$ (Fig. 5). This behavior can be expressed in FXML* without *intra-forall dependencies:*

```
for(d = 2; d <= M+N; d+1)
  forall(k = 1; cond(k, d, M, N); k+1) {
    i = indexi(k, d, M, N);
    j = indexj(k, d, M, N);
    LX: X = A[i-1][j] + 2;
    LY: Y = A[i][j-1] + 2;
    LZ: Z = A[i-1][j-1] + (S1[i]==S2[j]?-1:1;
    LA: A[i][j] = MIN(0, X, Y, Z);
  }
```

where `cond()`, `indexi()` *and* `indexj()` *are appropriately defined functions. The resulting C+OpenMP code looks as follows:*

```
#pragma omp section
for(int d = 2; d <= M+N; d=d+1)
  #pragma omp parallel for
  for(k = 1; cond(k, d, M, N); k+1) {
    i = indexi(k, d, M, N);
    j = indexj(k, d, M, N);
    LX: X = A[i-1][j] + 2;
    LY: Y = A[i][j-1] + 2;
    LZ: Z = A[i-1][j-1] + (S1[i]==S2[j]?-1:1;
    LA: A[i][j] = MIN(0, X, Y, Z);
  }
```

Indeed, this and other code transformations issued from research on loop parallelization [14] could be specified in terms of FXML transformations.

3.3.5 General code-generation flow

So far, we have left aside several important issues, such as, whether the FXML semantics is ensured by the target platform, how do we cope with target language limitations, and how timing constraints are handled by JAHUEL.

Concerning semantics, our approach relies on the existence of an *abstract formal model* of the target concrete execution platform onto which basic FXML can be translated to, by performing successive transformations whose correction is proved formally inside the FXML semantic world. The underlying assumption is that the concrete platform is indeed an *implementation* of the model, and that this relationship can be proved by some other means such as theorem proving or model checking.

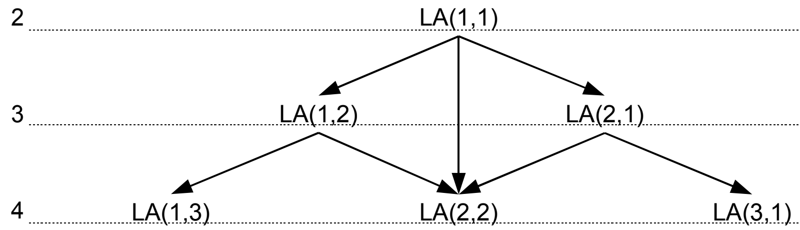


Figure 5: Smith-Waterman.

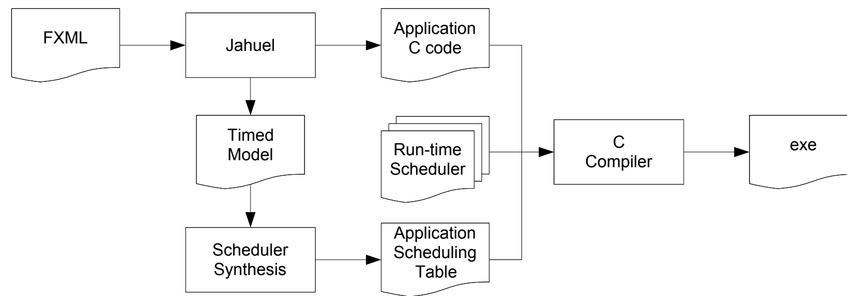


Figure 6: General code-generation flow.

Every transformation from a language $\mathcal{L}$ to $\mathcal{L}'$ has to deal with the question of how $\mathcal{L}$ statements are implemented in terms of $\mathcal{L}'$. If this cannot be done in general, then the transformation is typically defined only for a translatable subset of $\mathcal{L}$. Like any other compiler, JAHUEL performs static sanity checks to the input specification and rejects those which do not conform to the restrictions imposed by the transformation.

Nevertheless, our approach handles semantics, language limitations and non-functional constraints (which are sometimes not directly supported by the target execution platform) homogeneously following the general code-generation flow shown in Fig. 6. The basic approach has been first proposed in [25], and implemented for Java in [24], in the context of a Java-to-native code-generation tool-chain.

The idea is as follows. JAHUEL generates two outputs, namely, the application code and a timed model of it, both generated from the FXML specification. The former, rather than calling platform primitives directly, it calls a generic primitive `jahuel_call()`, implemented on top of the target platform, which is responsible for ensuring the correct behavior. For a *thread*-based implementation, the pseudo-code of `jahuel_call()` looks like this:

```
void jahuel_call(th_id tid, state curr, call_op cop, params p)
{
  system_lock(scheduler_mutex);
  jahuel_scheduler(tid, curr, cop, p);
  system_unlock(scheduler_mutex);
}
```

where `system_lock()` and `system_unlock()` are the corresponding platform lock and unlock functions, `scheduler_mutex` is a platform mutex used to insure mutual exclusion access to the scheduler by concurrent application threads, and `jahuel_scheduler()` is the platform-dependent function that performs the appropriate scheduling in order to preserve the semantics. This includes, in particular, ensuring that timing constraints are met. For this, the application provides some reflexive information, such as its *thread id* (`tid`) and its *current state* (`curr`), as well as the primitive to be called (`cop`) together with its parameters (`p`).

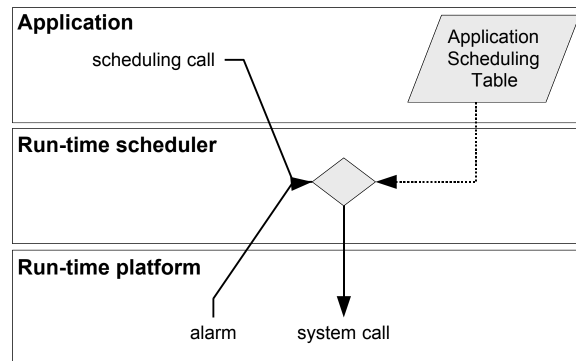


Figure 7: Schematic architecture and flow of the run-time.

The timed model is fed into a *scheduler synthesis* algorithm, based on the *controller synthesis* approach presented in [26], which generates an *application scheduling table*, if a scheduler exists. This table determines for each application's state with which the scheduler is called, which action has to be taken to preserve the semantics. Typically, such action consists in choosing a thread to execute and updating its state according to the scheduling table. The schematic view of the run-time is shown in Fig. 7. If a scheduling table cannot be computed, the algorithm returns useful information which can be used by the designer to understand the reasons of this and to modify its design accordingly.

In this setting, the generated code for the Writer-Reader looks as follows:

```
dep [0,15] W -> R
par
  Writer: thread
    state(wr0)
    jahuel_execute(Writer, wr0, {p = 0}, [0,1])
    jahuel_setclock(WPER)
    while(true)
      state(wr1)
      jahuel_call(Writer, wr1, lock)
      state(wr2)
      W: jahuel_execute(Writer, wr2,
             { x = p; p = p + 1; }, [0,1])
      jahuel_setclock(CLK)
      state(wr3)
      jahuel_call(Writer, wr3, notify)
      state(wr4)
      jahuel_waitforperiod(Writer, wr4, WPER, 10)
  Reader: thread
    while(true)
      state(rd0)
      jahuel_call(Reader, rd0, wait, CLK, [0,15])
      state(rd1)
      R: jahuel_execute(Reader, rd0,
             { y = x; legacy{ printf("%d\n", y); } }, [0,1])
      state(rd2)
      jahuel_call(Reader, rd2, notify)
```

Roughly speaking, calls with arguments `lock` and `notify` will behave like for *pthreads*. The call `jahuel_setclock(CLK)` sets clock `CLK` to 0. Thus, `CLK` counts the time elapsed since the last oc-

currence of `W. jahuel_call(Reader, rd0, wait, CLK, [0,15])` will block `Reader` if the value of `CLK` is not in the interval `[0,15]`. In this case, `Reader` will be awaken next time `Writer` executes `jahuel_setclock(CLK)`. The call `jahuel_waitforperiod()` makes `Writer` to wait until the value of clock `WPER` reaches its loop's period (10), and resets `WPER` to 0 when it returns, to start counting another period. If execution times are satisfied, the scheduler synthesis algorithm ensures that `Writer` never misses its period.

The function `jahuel_execute()` executes a block of code, updates the thread state, and checks whether specified execution times are respected. In principle, execution times should be checked to hold using worst- and best-case execution-time analysis techniques (e.g., [30]). In this case, run-time monitoring of execution times could be disabled.

In this example, the scheduling table generated by the synthesis algorithm depends on the type of the dependency, either `weak`, `strong`, or `(i,i)`, as well as on the timing constraints. The reader is referred to [24, 25] for more detailed information about this technique.

4 From code to FXML

FXML can be used as an *intermediate* format to represent program behavior. Besides providing formal semantics, translating a language into FXML enables performing program transformations and compiling programs to different target platforms, as explained in Section 3, and also doing, for instance, performance-driven design-space exploration [3], application-oriented scheduler synthesis [25], etc.

In this section, we study two examples of this approach. The first one consists in extending C with FXML-driven pragmas. The second one is about giving semantics to StreamIt [36].

4.1 C with pragmas

A common embedded-software programming practice in industry consists in using pragmas to annotate the program with extra-functional information about the behavior of the program, the target execution platform (run-time system and hardware). Such annotations are used to produce optimized code by the industrial C-compiler FlexCC2 [7] developed by STMicroelectronics. This approach has two major drawbacks. First, pragmas typically do not have formally defined meaning. Second, they are compiler-dependent. Using FXML allows to overcome these two issues.

To illustrate the idea, let us use again the Writer-Reader application. The following `writer-reader.c` C program is given in the actual syntax used by FlexCC2:

```
int x = 0;

#pragma code_block
#pragma dependency
main.writer.write -> (x)
  main.reader.read [0,15] us
#pragma parallel writer reader
main()
{
  writer(); /* /&/ */ reader();
}

#pragma code_block
void writer()
{
  int p;
  write_init();

  #pragma period 10 us
  while (1) { write(); }
}
```

```
#pragma code_block
#pragma execution_time [0,1] us
void write_init()
{
    p = 0;
}

#pragma code_block
#pragma execution_time [0,1] us
void write()
{
    x = p++;
}

#pragma code_block
void reader()
{
  while (1) { read(); }
}
```

```
#pragma code_block
#pragma execution_time [0,1] us
void read()
{
    int y = x;
    printf("%d\n", y);
}
```

FlexCC2 analyzes the program (pragmas and C code) and extracts a description of it in the concrete XML syntax of FXML. The FXML specification obtained is similar to the one used in Sec. 2. The main difference is that assignments and calls to C functions are considered to be legacy code:

```
dep [0,15]
main.writer.write -> main.reader.read
main:
  legacy{ #include writer-reader.h }
  par
    main.writer:
      legacy{ int p; write_init(); } [0,1]
      while(true) per=10
        main.writer.write: legacy{ write(); } [0,1]
    main.reader:
      while(true)
        main.reader.read: legacy{ read(); } [0,1]
```

Then, JAHUEL can be used to generate code for different execution platforms as explained in Section 3. An industrial application of the tool-chain composed by FlexCC2 and JAHUEL is presented in [2].

4.2 StreamIt to FXML

StreamIt [36] [5] is a language designed for programming streaming applications. StreamIt semantics is defined by its compiler infrastructure which provides native compilation for the MIT Raw machine [18] and code-generation into a C++ run-time library for execution on general-purpose processors.

Here, we provide a translation of a subset of StreamIt into FXML.

4.2.1 StreamIt syntax

StreamIt is built around the notion of *stream*. A stream is an ordered (unbounded) sequence of data. Streams are implicit, that is, they do not have a name and can only be accessed through specific built-in functions. A StreamIt process S is defined as follows.

Filters: The basic StreamIt process is the filter. Filters are (endless) loops with (at most) one input stream and (at most) one output stream. The syntax is as follows: **filter init** C_{init} **work pop** k_1 **peek** k_2 **push** k_3 C_{work}. C is a *block* of sequential code. A filter executes the **init** block C_{init} once and the **work** block C_{work} at every iteration.

Filters manipulate the input stream via the function `pop()`, that returns and removes the first element of the stream, and the function `peak(`i`)`, that returns the i-th element. Filters manipulate the output stream via the function `push(data)` that appends data to the output stream.

The number of inserted, peeked and deleted elements at each iteration, that is the pop, peek and push *rates*, are specified by the **pop**, **peek**, and **push** declarations. The peek rate specifies the *maximum* index that is allowed to be peeked in any iteration. It is required to be greater than or equal to the pop rate.

Pipelines: Processes can be grouped in a *pipeline* connected through input/output streams: **pipeline** $S_1 \ldots S_n$. Connections are made sequentially following the declaration order.

Example 4.1 (Writer/Reader) *In StreamIt, the writer-reader application of Example 2.1 is as follows:*

[5] http://www.cag.csail.mit.edu/streamit/index.shtml

```
void -> void pipeline PC() {
  add Writer();
  add Reader();
}
void -> int filter Writer() {
  init int p = 0;
  work push 1 {
     push(p);
     p++;
  }
}
int -> void filter Reader() {
  work pop 1 {
     print(pop());
  }
}
```

`Writer` and `Reader` are implemented as filters *connected in a* pipeline. *A **filter** repeatedly executes the **work** function: `Writer` pushes a value of `p` at a time, expressed by a* push *rate of 1, while `Reader` pops one element at each iteration, expressed by a* pop *rate of 1, and in the same order. Clearly, this StreamIt program behaves like the FXML one in Example 2.1, with a dependency of type (i, i).*

Example 4.2 (Writer/Reader with peek) *Now consider the following StreamIt program where `Reader` peeks two values from the input stream and sums them up:*

```
void -> void pipeline PC2() {
  add Writer();
  add Reader();
}
void -> int filter Writer() {
  int p = 0;
  work push 1 {
     push(p);
     p++;
  }
}
int -> void filter Reader() {
  work pop 1 peek 2 {
     print(peek(0) + peek(1));
     pop();
  }
}
```

In this case, the translation to FXML is not as simple as before. A compositional *and* systematic *way of doing it consists in adding a buffer in-between* [6]*:*

```
var int soutP
var int sinC[2]
dep (0,0) push -> init
dep (i+1,i) push -> get
dep (i,i) put -> peek
par
  Writer:
    var int p = 0
```

[6]To enhance readability, we explicitly declare variables through the `var` statement.

```
    while(true)
      push: soutP = p
      p++
  Buffer:
    init: sinC[1] = soutP
    while(true)
      put: {
        shift: sinC[0] = sinC[1]
        get: sinC[1] = soutP
      }
  Reader:
    var int x[]
    while(true)
      peek: for(k=0; k<2; k++) x[k] = sinC[k]
      print(x[0]+x[1])
```

This example provides the basis for a systematic translation of StreamIt into FXML.

StreamIt provides other constructs, such as **splitjoin**, which allows an input stream to be split in several copies to be handled by multiple filters simultaneously, and **feedbackloop**, that allows making the output stream available as input stream. For simplicity, we do not consider here these operators since their translation into FXML is more involved, but can be done following the same ideas.

4.2.2 StreamIt semantics in FXML

Let S be a syntactically correct StreamIt program. We assume that for every **work** construct in a filter F pop rate $popr(F)$ and peek rate $peekr(F)$:

- Pops are grouped at the end in a loop of the form:

  $$\texttt{for(}k=0\texttt{;}\;\; k<popr(F)\texttt{;}\;\; k\texttt{++)}\,\textbf{pop}().$$

  We denote this block **pop**$(popr(F))$.

- Peeks appear in a loop of the form:

  $$\texttt{for(}k=0\texttt{;}\;\; k<peekr(F)\texttt{;}\;\; k\texttt{++)}\,\{x[k]=\textbf{peek}(k)\},$$

  where $x[]$ is a local array of dimension $peekr(F)$. We denote this block **peek**$(peekr(F))$.

The translation from StreamIt to FXML is as follows.

Filter: A filter F is a sequential *pnode*, with two associated variables s_F^{in} and s_F^{out}:

$$\Gamma(\textbf{filter init } C_{init} \textbf{ work } \textit{rates } C_{work}) \;\triangleq\; \texttt{seq}\,\Gamma(\textbf{init } C_{init})\;\; \Gamma(\textbf{work } \textit{rates } C_{work})$$

Work: The translation of **work** functions is independent of the push, peek and pop rates:

$$\Gamma(\textbf{work } \textit{rates } C) \;\triangleq\; \texttt{while(true)}\{\Gamma(C)\}.$$

Push: Pushing a value in the output stream of a filter F is translated into storing the value in the variable s_F^{out}:

$$\Gamma(\textbf{push}(f(\ldots))) \;\triangleq\; \texttt{push}_F\texttt{:}\; s_F^{out}=\Gamma(f(\ldots)).$$

Pop: Popping values is done in the corresponding buffer. Then:

$$\Gamma(\textbf{pop}(n)) \triangleq \texttt{nil}.$$

Peek: $\Gamma(\textbf{peek}(n)) \triangleq \texttt{peek}_F\texttt{: forall(}k=0\texttt{;}\ \ k<n\texttt{;}\ \ k\texttt{++)}\{\,x[k] = s_F^{in}[k]\,\}$.

Pipeline: The *pnode* of a pipeline consists in composing its processes with a `par` and connecting them with intermediate buffers: $\Gamma(\textbf{pipeline}\ S_1 \ldots S_n)$ is the *pnode*

$$\texttt{dep}\ D_{1,\ldots,n}\ \texttt{par}\ \Gamma(S_1)\ B_{S_1,S_2} \ldots B_{S_{n-1},S_n}\ \Gamma(S_n)$$

with

$$B_{S_i,S_{i+1}} \triangleq B(s_{S_i}^{out}, s_{S_{i+1}}^{in}, peekr(S_{i+1}), popr(S_{i+1}))$$

where `B(bin, bout, peekr, popr)` is the *pnode*:

```
B:
  for(k = 0; k < peekr - popr; k++)
    init: bout[k + popr] = bin
  while(true)
    put: {
      forall(k = 0; k < peekr - popr; k++)
         shift: bout[k] = bout[k + popr]
      for(k = 0; k < popr; k++)
         get: bout[k + peekr - popr] = bin
    }
```

such that `bin` is instantiated with the output variable $s_{S_i}^{out}$ of the process S_i pushing values into the stream, `bout[]` is instantiated with the corresponding vector $s_{S_{i+1}}^{in}[]$ to store the pushed values, and `peekr` and `popr` are the peek $peekr(S_{i+1})$ and pop $popr(S_{i+1})$ rates of S_{i+1}, respectively.

`B` starts by initializing the vector `bout[]` with `peekr-popr` elements, from index `popr`. Afterwards, it keeps forever shifting left the contents of the vector, which corresponds to popping `popr` items, and inserting `popr` new ones.

$D_{1,\ldots,n} \triangleq \bigcup_{1\leq i\leq n-1} D_{i,i+1}$ with $D_{i,i+1}$ the set of dependencies from $\Gamma(S_i)$ to $B_{i,i+1}$ and from $B_{i,i+1}$ to $\Gamma(S_{i+1})$, defined as follows:

- The first k pushed values, $k \in [0, peekr(S_{i+1}) - popr(S_{i+1}))$ serve to initialize the buffer, that is $\texttt{init}^{\langle k\rangle}$ in $B_{i,i+1}$ depends on $\texttt{push}^{\langle k\rangle}$ in $\Gamma(S_i)$:

$$\{(k,k) \mid k \in [0, peekr(S_i) - popr(S_i)]\}\ \texttt{push}_{S_i} \rightarrow \texttt{init}_{B_{i,i+1}}$$

- Every occurrence $\texttt{peek}^{\langle j\rangle}$ in $\Gamma(S_{i+1})$ is required to be preceded by $\texttt{put}^{\langle j\rangle}$ in $B_{i,i+1}$ in order to ensure that $peekr(S_{i+1})$ values have been pushed:

$$(j,j)\ \texttt{put}_{B_{i,i+1}} \rightarrow \texttt{peek}_{S_{i+1}}$$

- Every occurrence $\texttt{get}^{\langle j,k\rangle}$ in $B_{i,i+1}$, $k \in [0, peekr(S_{i+1}) - popr(S_{i+1}))$, gets the value pushed by $\Gamma(S_i)$ in the assignment $\texttt{push}^{\langle h\rangle}$, with index $h = peekr(S_{i+1}) - popr(S_{i+1}) + j \cdot popr(S_{i+1})$:

$$\begin{array}{lll}\{(h,(j,k)) & \mid & h = peekr(S_{i+1}) - popr(S_{i+1}) + j \cdot popr(S_{i+1}) \\ & \wedge & k \in [0, peekr(S_{i+1}) - popr(S_{i+1}))\ \}\ \texttt{push}_{S_i} \rightarrow \texttt{get}_{B_{i,i+1}}\end{array}$$

The translation of StreamIt into FXML gives a *formal* semantics to StreamIt and enables verification and scheduler synthesis. Besides, it allows using JAHUEL to generate code for target platforms other than those supported by the StreamIt compiler infrastructure.

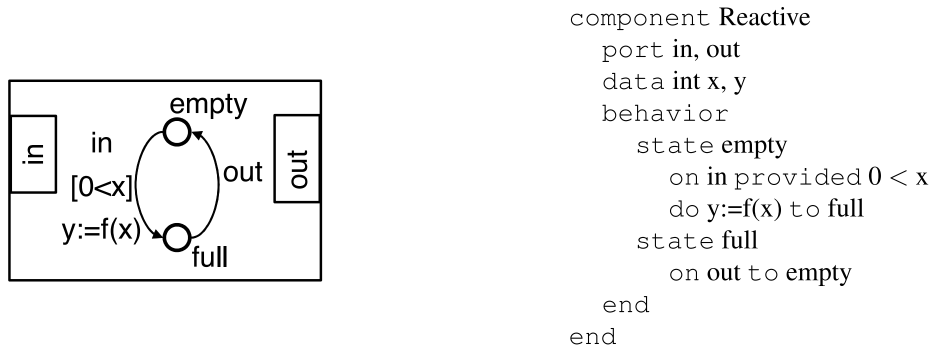


Figure 8: An atomic component.

5 FXML to BIP

For embedded software product lines, where new functionalities and services are continuously developed, the main challenge is to provide design frameworks capable of supporting software componentization to ease integration and evolution. BIP (Behavior, Interaction, Priority) [4] has been designed to overcome the difficulties of state-of-the-art component-based approaches [33]. BIP provides a language and a theory for incremental composition of heterogeneous components, ensuring correctness-by-construction for essential system properties such as mutual exclusion, deadlock-freedom and progress. Besides, it enables verification through model-checking via the IF tool-suite [10].

Nevertheless, many high-performance embedded applications, such as video compression (e.g., MPEG-4), are not programmed following a component-based approach, but most likely a data-flow one. These applications are better described using languages such as StreamIt and FXML. Here, we provide an automated method for generating componentized implementations in BIP of data-driven applications specified in FXML. We illustrate the concept with an industrial MPEG-4 video encoder [2].

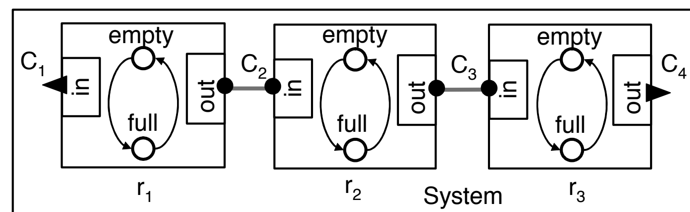
5.1 The BIP language

BIP is formally defined in [33]. It supports a methodology for building components from *atomic* ones, using *connectors*, to specify *interaction* patterns between *ports* of atomic components, and *priority relations*, to select amongst possible interactions. Here, we review the basic concepts through illustrative examples.

Fig. 8 shows an *atomic* component with two *ports* in, out, local *variables* x, y, and *control states* $empty$, $full$. Ports are action names used for synchronization with other components. Control states denote locations at which the components await for synchronization. Variables are used to store local data. Transitions model atomic computation steps. In general, 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 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. In the example, component *Reactive* can take the transition labeled in at $empty$ 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 guard is $true$ and the internal computation microstep is empty.

A *compound* component is a component consisting of atomic or compound sub-components. An example of a compound component named *System* is shown in Fig. 9. It is the connection of three instances of *Reactive*.

Components are connected through *connectors*, which are sets of ports that contain at most one port from each atomic component. An *interaction* is any non-empty subset of a connector. In *System* there are four connectors: C_1, consisting of port $r_1.in$ alone, C_2 consisting of ports $r_1.out$ and $r_2.in$, and so forth. There are two types of interactions, namely *complete* and *incomplete*. An interaction of a connector




```
component System
   contains Reactive r1, r2, r3
   connector C1 = r1.in
   complete = r1.in
   connector C2 = r1.out|r2.in
   behavior
      on r1.out|r2.in do r2.x := r1.y
   end
   connector C3 = r2.out|r3.in
   behavior
      on r2.out|r3.in do r3.x := r2.y
   end
   connector C4 = r3.out
   complete = r3.out
   priority P1 r1.in < r2.out|r3.in
   priority P2 r1.in < r3.out
   priority P3 r1.out|r2.in < r3.out
end
```

Figure 9: A compound component.

is *feasible* if it is complete or if it is maximal. We denote graphically an incomplete interaction by a bullet and a complete one by a triangle. For instance: C_1 is complete, meaning that *System* can engage in an interaction containing port in if r_1 can; C_2 is maximal, meaning that components r_1 and r_2 must *synchronize* on $r_1.out$ and $r_2.in$, that is, neither one can proceed alone on transitions labelled out and in, respectively. Connectors may have behavior specified as for transitions, by a set of guarded commands associated with feasible interactions. For instance, whenever the interaction $r_2.out|r_3.in$ takes place, $r_2.x$ receives the value of $r_1.y$. In general, guards and statements are C expressions and statements, respectively.

Priorities are used to choose amongst simultaneously enabled interactions. They are a set of rules, each consisting of an ordered pair of interactions associated with a condition. When the condition holds and both interactions are enabled, only the higher-priority one is possible. Conditions can be omitted for static priorities. The rules are extended for composition of interactions, e.g., $b_1 < b_2$ means that any interaction of the form $b_2|\alpha$ has higher priority than all interactions of the form $b_1|\alpha$, for all interactions α. In our example, $r_1.in < r_2.out|r_3.in$ means that *System* will not take any transition where $r_1.in$ is involved, whenever the synchronization between $r_2.out$ and $r_3.in$ is enabled. Indeed, the priorities specified in *System* enforce a causal order of execution as follows: once there is an in through C_1, data are processed and propagated sequentially through sub-components r_1, r_2, and r_3, 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.

5.2 Translation scheme

To illustrate the idea of the FXML-to-BIP translation scheme, let us start with the writer-reader FXML specification of Ex. 2.1, without timing constraints. *Pnodes* `Writer` and `Reader` become BIP components:

```
component Writer
  port out
  data int x, p
  behavior initial do p = 0; to S
    state S
      on out do x = p; p = p + 1;  to S
  end
end

component Reader
  port in
  data int y
  behavior initial to S
    state S
      on in do y = x; {# printf("%d\n", y); #} to S
  end
end
```

Communication between the two is done through a component `Buffer` which is added for several reasons: (1) to encapsulate `x`, because BIP does not allow shared variables; (2) to realize the synchronization protocol ensuring the dependency `W` $\rightarrow$ `R`, to comply with FXML semantics; and (3) to implement the buffering scheme. In BIP, legacy code is written between “`{#`” and “`#}`”.

The composed system in BIP is:

```
component System
  contains Buffer B
  contains Writer P
  contains Reader C
  connector C1 = P.out | B.put
```

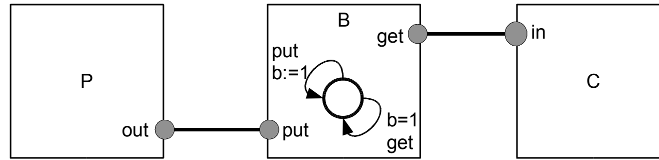


Figure 10: BIP model for *weak* dependency

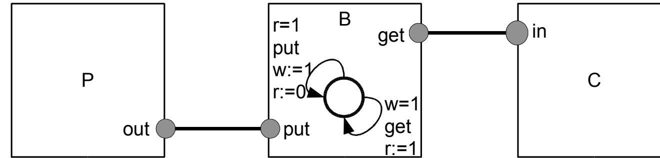


Figure 11: BIP model for *strong* dependency

```
    behavior do B.x = P.x; end
  complete P.out | B.put
  connector C2 = C.in | B.get
    behavior do C.x = B.x; end
  complete C.in | B.get
end
```

Connectors `C1` and `C2` implement the data transfer. The behavior of `Buffer` depends on the dependency type and the storage policy. For a *weak* dependency with single-storage (Fig. 10), `Buffer` is:

```
component Buffer
  port put, get
  data int x, b
  behavior initial do b=0; to S
    state S
      on put do b=1; to S
      on get provided (b==1) to S
  end
end
```

For a *strong* dependency with single-storage (Fig. 11), `Buffer` uses variable `r` to notify whether the latest written value of `x` has been read, and therefore whether a new `put` can be accepted, and `w`, to notify whether `x` has been written (at least once), to condition interactions on `get`. The BIP model of `Buffer` (Fig. 11) is as follows:

```
component Buffer
  port put, get
  data int x, b
  behavior initial do r=1; w=0; to S
    state S
      on put provided (r==1) do w=1; r=0; to S
      on get provided (w==1) do r=1; to S
  end
end
```

For a (i, i) dependency with single-storage, `Buffer` (Fig. 12):

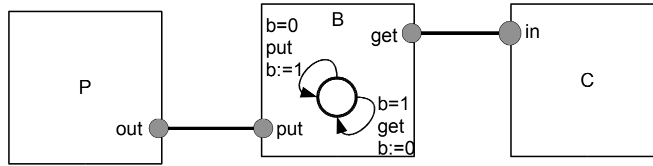


Figure 12: BIP model for (i, i) dependency

```
component Buffer
  port put, get
  data int x, b
  behavior initial do b=0; to S
    state S
      on put provided (b==0) do b=1; to S
      on get provided (b==1) do b=0; to S
  end
end
```

The writer-reader example provides a basis for a general translation scheme.

- Consider the case of multiple dependencies incoming into an assignment $\ell : y = f(x_1, \ldots, x_n)$ of a *pnode* C from assignments $\ell_i : x_i = \ldots$ in *pnodes* P_i, $i \in [1, n]$. The FXML semantics is the *conjunction* of constraints imposed by the dependencies. In BIP, this can be modeled by setting up buffer components B_i, one for each dependency $\ell_i \to \ell$, $i \in [1, n]$, whose role is to realize the corresponding control policy depending on the dependency type, as well as to implement the desired buffering policy (if any). W.l.o.g., we assume B_i is a single storage buffer, with a local variable $B_i.x$, $i \in [1, n]$. Other buffering policies only require changing the behavior of connectors. The conjunctive semantics is ensured in BIP by the *maximal interaction* $in|get_1| \ldots |get_n$, where buffered values $B_i.x$ are copied to C-local variables $C.x_i$ (Fig. 13).
- The other paradigmatic case consists of multiple dependencies $\ell \to \ell_i$ outgoing from a (writer-like) *pnode* P, executing the assignment $\ell : x = \ldots$, to many (consumer-like) *pnodes* C_i, computing $y_i = f_i(x)$, $i \in [1, n]$. The translation is similar to the previous case where a buffer B_i is used for each dependency $\ell \to \ell_i$, $i \in [1, n]$. The conjunctive semantics is ensured by the *maximal interaction* $out|put_1| \ldots |put_n$, whose behavior is to set $B_i.x = P.x$ for all $i \in [1, n]$. (Fig. 14).

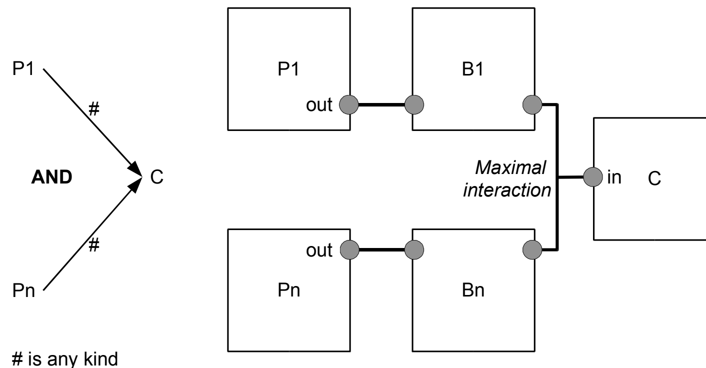


Figure 13: Multiple incoming dependencies

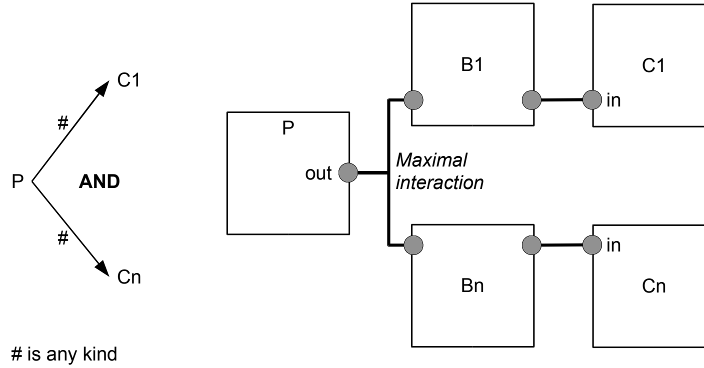


Figure 14: Multiple outgoing dependencies

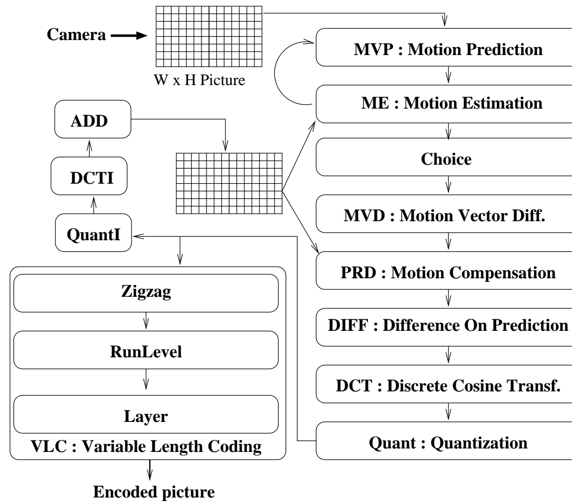


Figure 15: MPEG block diagram view

5.3 Case study: MPEG-4 encoder

In this section we apply the translation scheme presented in previously to a MPEG-4 video encoder. For lack of space, we only present here a (significant) part of the FXML and BIP models. The full FXML specification is given in [2]. This model describes all the existing concurrency in the compression algorithm at the macroblock level. Such concurrency does not appear in the simplified MPEG-4 block diagram shown in Fig. 15.

The specification is composed of `forall` nodes, C-code blocks of the corresponding MPEG-4 computations, and dependencies of the MPEG-4 phases. The basic data structure is a matrix of $W \times H$ macroblocks. FXML specification uses x and y as the iteration variables, i.e., $x \in [0, W)$ is the row, and $y \in [O, H)$ the column, of the macroblock. A *pnode* specifying the behavior of an MPEG-4 computation step `s` (e.g., MVP, ME, etc.) has the following structure:

```
forall(x = 0; x < W; x + 1)
  forall(y = 0; y < H; y + 1)
    legacy{ M_s[x][y] = F_s( ... ); }
```

where `M_s` is the *output* matrix of step `s`, and `F_s` is the computation applied at step `s`. `F_s` depends on a matrix computed in a preceding step, e.g., `M_MVP[x][y]` depends on `M_ME[x][y]`, `M_ME[x-1,y]`,

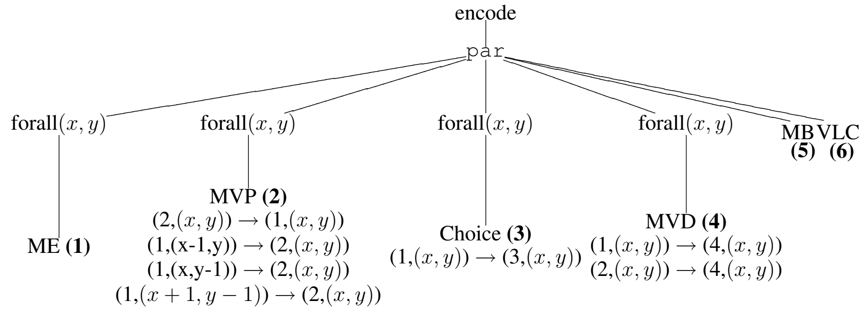


Figure 16: FXML specification of encode

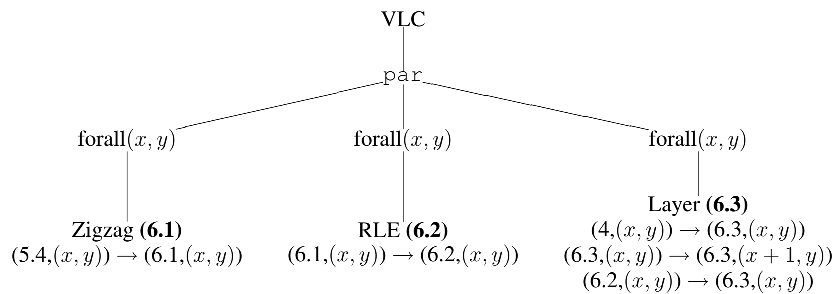


Figure 17: FXML specification of VLC

`M_ME[x,y-1]`, and `M_ME[x-1,y-1]`.

For readability, we use a tree-like representation, instead of a textual pseudo-code, where *pnodes* are labeled with numbers in brackets (Fig. 16). We note (1,(x,y)) the computation corresponding to the execution of the ME (motion estimation) phase on the frame macroblock at position (x,y). The arrows indicate dependencies between these computations. There are three types of dependencies : (1) data dependencies resulting from the MPEG-4 standard specification (e.g., in Fig. 16, (1,(x,y)) → (3,(x,y)) is a data dependency expressing that the ME phase on macroblock (x,y) must finish before starting the Choice phase on the same macroblock), (2) functional dependencies necessary for the correct functioning of the application (e.g., there is a functional dependency from macroblock (x,y) to macroblock $(x+1,y)$ in the specification of VLC (Fig. 17) because generated headers and blocks are sequentially written in the output bitstream), and (3) dependencies resulting from implementation decisions (e.g., using input and output buffers with one-frame capacity) of encoding frames one after another.

The overall specification is 7650 lines of FXML, including 7500 lines of C code corresponding to encoding computations. Indeed, the FXML specification can be obtained from the sequential C code annotated with special purpose pragmas [1] using FlexCC2 [7].

Before applying the translation scheme to obtain a BIP model, we perform several FXML-to-FXML transformations. The main problem to face is to determine the granularity of the componentization. Here, we take each MPEG-4 computation step to be implemented inside a single atomic component. To achieve this, the parallelism inside each step is eliminated, by making each `forall` statement to become a `for`. To code the nested `for` in BIP, we need to add two complete ports `tau` and `stop`, together with two singleton connectors, to model internal component transitions. Therefore, the component `C_s` is as follows:

```
component C_s()
  port in, out
  port complete tau, stop
  data int x, y
  behavior initial x = 0; y = 0; to WAIT
    state WAIT
```

```
      on in provided ((x < W) && (y < H)) to WORK
      on tau provided ((x < W) && (y == H))
         do x = x + 1; y = 0; to WAIT
      on stop provided (x == W) do x = 0; to WAIT
    state WORK
      on out do {# M_s[x][y] = F_s(...); #}; y = y + 1; to WAIT
  end
  connector conn_tau = tau
    behavior
    end
  connector conn_stop = stop
    behavior
    end
end
```

`C_s` has a local matrix variable `M_s[][]` to store the computed value at each iteration. For each component `C_s`, there is a single output buffer `B_s` which encapsulates the output matrix of macro-blocks. This is because there is only one outgoing dependency out of each component. The transfer of each macro-block `M_s[x][y]` from the (writer) component `C_s` to the buffer `B_s` is done in the connector connecting ports `C_s.out` and `B_s.put`. From `B_s` to the (consumer) component `C_s'`, corresponding to the step `s'` following `s`, the transfer occurs in the connector connecting ports `B_s.get` and `C_s'.in`. Fig. 18 depicts the schematic view of the BIP components for VLC, where diamonds represent buffers.

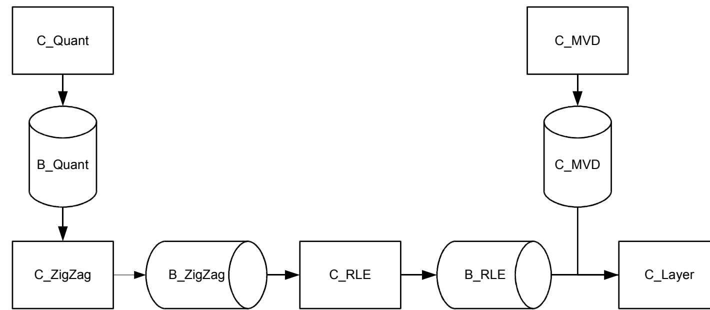


Figure 18: VLC

The connector `T_4_62_63` defining the interaction between `B4`, `B62`, and `C63`, is as follows:

```
connector T_4_62_63 = B4.get,B62.get,C63.in
  behavior
    do C63.M_4[x][y] = B4.M[x][y], C63.M_62[x][y] = B62.M[x][y]
  end
```

The other connectors are obtained similarly.

The resulting BIP model consists of 37 components, 21 of which are buffers, and 32 connectors. The BIP source code has 8150 lines, including 650 lines of pure BIP and the 7500 lines of C code of the (original sequential) encoder.

It is worth noticing that the BIP model characterizes the set of *all* possible schedulings of the computations. This non-deterministic behavior can be constrained by adding priorities, eventually resulting on a sequential execution of the encoder. To start with, we have considered single-storage buffers, so producers are prevented from re-writing values until consumer(s) have read them. Clearly, by changing the size of buffers and their policy, we can obtain a higher degree of parallelism. Moreover, this change does not affect the FXML specification, but only the last phase of the code generation chain. The latter just implies

re-generating the behavior of the buffer component in BIP, but not its ports and connectors, thus achieving modular code generation.

We have also compared the BIP model obtained from the FXML description with a BIP model of the encoder written by hand. The latter consists of 15% fewer lines of BIP (548 instead of 650), almost 50% less components (11 MPEG components and 9 buffers), and 33 connectors. The larger number of components is mainly due to the fact that the hand-written model encapsulates VLC and QuantI/DCI in single components. Besides, most dependencies are between two successive components and hence all connectors are of arity two (creating a chain of dependencies). In the FXML specification, dependencies allow more parallelism and relate more than two components, resulting in more than one incoming buffers and n-ary connectors in some cases (with $n > 2$) (see Fig. 18, for instance).

6 Conclusions

FXML is a formal language for specifying concurrent real-time applications. It has a simple abstract execution model based on the notion of assignments and dependencies. It can be incrementally extended with information related to refinements of the abstract model into more concrete ones. FXML can be used as a modelling language by itself, that is, FXML specifications can be directly written by designers, or as semantic framework for other languages, such as StreamIt, where FXML specifications are obtained by a compiler.

An FXML-compiler is a sequence of transformations going from a language (or model) to another (more concrete one). Based on this idea, we have developed the compilation chain JAHUEL which implements several translation phases which can be easily customized for different platforms. Hence, JAHUEL provides tool support for handling concurrency and timing constraints in software product lines. In particular, we have shown how to generate code for several execution platforms, such as *pthreads* and OpenMP. JAHUEL provides an FXML-to-BIP transformation which enables formal verification via the IF framework.

The FXML and JAHUEL are grounded on well established notions from process algebras, program analysis and transformation, refinement and scheduler synthesis. The main contribution of this work is to have shown that these techniques could be put together into a pre-industrial, extensible, and customizable code-generation chain for software product lines, without semantic break-downs from an abstract specification all the way down to executable code.

Ongoing work includes applying FXML and JAHUEL in other industrial applications, strengthening the integration into an end-to-end industrial design flow, and generating code for other platforms. Special effort is being put on generating code for the simulation infrastructure P-WARE [3]. The main motivation for this is early prototyping, verification and testing of embedded applications on simulated hardware platforms, since automated generation of both executable and simulation code from the same formal model ensures simulation results are trustworthy.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We are indebted to Valérie Bertin, Philippe Gerner, Christos Kloukinas and Olivier Quévreux for their contribution to the early stages of the definition and development of FXML and JAHUEL, and to J. Sifakis who motivated the FXML-to-BIP transformation. The work reported in this chapter has been partially supported by projects MEDEA+ NEVA, Minalogic SCEPTRE, Crolles-II ANACONDA, STIC-AmSud TAPIOCA, IST SPEEDS and RNTL OpenEmbeDD. This chapter has been written while S. Yovine was Visiting Professor at DC/FCEyN/UBA and UADE, Argentina, partially funded by BID-ANPCyT PICTO-CRUP 31352 and MINCyT "César Milstein" grant.

References

[1] I. Assayad, V. Bertin, F. Defaut, Ph. Gerner, O. Quévreux, S. Yovine. JAHUEL: A formal framework for software synthesis. In *7th International Conference on Formal Engineering Methods*, ICFEM 2005: 204-218. Lecture Notes in Computer Science 3785. Springer 2005. 1, 2, 5.3

[2] I. Assayad, Ph. Gerner, S. Yovine, V. Bertin. Modelling, analysis, implementation of an on-line video encoder. In *1st International Conference on Distributed Frameworks for Multimedia Applications*, DFMA 2005: 295-302. IEEE Computer Society 2005. 4.1, 5, 5.3

[3] I. Assayad, S. Yovine. P-Ware: A precise, scalable component-based simulation tool for embedded multiprocessor industrial applications. In *10th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD'07: 181-188 . IEEE Computer Society 2007. 3.2, 4, 6

[4] A. Basu, M. Bozga, J. Sifakis. Modeling heterogeneous real-time components in BIP. In *4th IEEE International Conference on Software Engineering and Formal Methods*, SEFM 2006: 3-12. IEEE Computer Society 2006. 1, 5

[5] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, J. Sifakis. Using BIP for Modeling, Verification of Networked Systems: A Case Study on TinyOS-based Networks. In *6th IEEE International Symposium on Network Computing and Applications*, NCA 2007: 257-260. IEEE Computer Society 2007. 1

[6] J.A. Bergstra, A. Ponse, S.A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001. 1

[7] V. Bertin, J.M. Daveau, P. Guillaume, T. Lepley, D. Pilat, C. Richard, M. Santana, T. Thery. FlexCC2: An optimizing retargetable C compiler for DSP processors. In *2nd International Conference on Embedded Software*, EMSOFT 2002: 382-398. Lecture Notes in Computer Science 2491, Springer 2002. 4.1, 5.3

[8] P. Binns, S. Vestal. Formalizing software architectures for embedded systems. In *1st International Workshop on Embedded Software*, EMSOFT 2001: 451-468. Lecture Notes in Computer Science 2211, Springer 2001. 1

[9] H. Boehm. Threads cannot be implemented as a library. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI 2005: 261-268. ACM 2005. 1

[10] M. Bozga, S. Graf, Il. Ober, Iul. Ober, J. Sifakis. The IF Toolset. In *Formal Methods for the Design of Real-Time Systems*, SFM 2004: 237-267. Lecture Notes in Computer Science 3185, Springer 2004. 1, 5

[11] A. Burns, A. Wellings. *Concurrency in Ada*. Cambridge University Press, 1998. 1

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2003: 153-162. ACM 2003. 1

[13] P. C. Clements, L. Northrop. *Software product lines: Practices, patterns*. Addison-Wesley, 2001. 1

[14] A. Darte, Y. Robert, F. Vivien. *Scheduling, Automatic Parallelization*. Birkhäuser, Boston, 2000. 3.1

[15] S. Dutta, R. Jensen, A. Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box, Digital TV Systems. *IEEE Design, Test of Computers*, 18(5):21-31, Sep./Oct. 2001. IEEE Computer Society. 1

[16] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *3rd Workshop on Productivity and Performance in High-End Computing*, PPHEC 2006: 30-36. IEEE Computer Society 2006. 1, 2.2

[17] J. Goguen, G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996. 2.2.2

[18] M. I. Gordon, W. Thies, S. P. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2006: 151-162. ACM 2006. 4.2

[19] W. Groppa, E. Lusk, A. Skjellum. *Using MPI*. Scientific, Engineering Computation. MIT Press, 2nd edition, November 1999. 1

[20] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), Sept. 1991. IEEE Computer Society. 1

[21] www.intel.com/design/network/products/npfamily/index.htm 1

[22] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1), 2003. IEEE Computer Society. 1

[23] M. Kersten. Comparison of the leading AOP tools. In *4th International Conference on Aspect-Oriented Software Development*, AOSD 2005. Industry track. Invited talk. ACM 2005. 1

[24] Ch. Kloukinas, Ch. Nakhli, S. Yovine. A Methodology and Tool Support for Generating Scheduled Native Code for Real-Time Java Applications. In *3rd International Conference on Embedded Software*, EMSOFT 2003: 274-289.Lecture Notes in Computer Science 2855, Springer 2003. 3.3.5, 3.3.5

[25] Ch. Kloukinas, S. Yovine. Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems, In *15th Euromicro Conference on Real-Time Systems*, ECRTS 2003: 287-294, IEEE Computer Society 2003. 3.3.5, 3.3.5, 4

[26] O. Maler, A. Pnueli, J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, STACS 1995: 229-242, Lecture Notes in Computer Science 900, Springer 1995. 3.3.5

[27] www.openmp.org 1, 1, 3.3.4

[28] G. A. Papadopoulos, F. Arbab. Coordination models and languages. *Advances in Computers*, 46, 1998. Elsevier. 1

[29] http://ptolemy.eecs.berkeley.edu/ptolemyII 1

[30] P. Puschner, A. Burns. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems* 18(2/3):115-128, 2000. Springer. 3.3.5

[31] M. C. Rinard. Analysis of Multithreaded Programs. In *8th International Symposium Static Analysis*, SAS 2001: 1-19. Lecture Notes in Computer Science 2126, Springer 2001. 1

[32] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, February 5, 2002. http://www.gigascale.org/pubs/141.html 1

[33] J. Sifakis. A framework for component-based construction. In *3rd IEEE International Conference on Software Engineering and Formal Methods* , SEFM 2005: 293-300. IEEE Computer Society 2005. 5, 5.1

[34] J. Sifakis, S. Tripakis, S. Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100–111, January 2003. IEEE Computer Society. 1

[35] P. Stravers. Homogeneous multiprocessing for the masses. In *2nd Workshop on Embedded Systems for Real-Time Multimedia*, ESTImedia 2004: 3. IEEE Computer Society 2004. 1

[36] W. Thies, M. Karczmarek, S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *11th International Conference on Compiler Construction*, CC 2002: 179-196. Lecture Notes in Computer Science 2304, Springer 2002. 1, 4, 4.2