Seventh Framework Programme – Theme 3.6: Computing Systems

Grant agreement n°248776

D3.1
"Report on 3D Aware Specification of Parallel Applications"

WP3: "High Level Programming Model, Compilation & Verification"

Editor: Saddek BENSALEM, Marius BOZGA, UJF

Date: Friday 1st July, 2011

1	CEA (Coord.)	Commissariat à l'énergie atomique et aux énergies alternatives, Laboratoire d'électronique et des technologies de l'information
2	UJF	Université Joseph Fourier Grenoble 1 – VERIMAG
3	ETHZ	Eidgenössische Technische Hochschule Zürich
4	UNIBO	Università di Bologna
5	STM	STMicroelectronics
6	EPFL	École polytechnique Fédérale de Lausanne

Version: v1.2 – Status: Public

CEA ref.: LETI/DACLE/11-0434

Versions of the Document

Version	Date	Author	Comment
0.1	Sep. 17st, 2010	Marius BOZGA	Initial draft version
1.0	Dec. 20th, 2010	Ananda BASU	Delivered version
1.1	Feb. 18th, 2011	Marius BOZGA	Public version
1.2	Jun. 17th, 2011	Christian FABRE & Saddek BENSALEM	Added impact of 3D on programming models and runtimes.

Contributors

Ananda BASU (UJF), Saddek BENSALEM (UJF),
Marius BOZGA (UJF), Christian FABRE (CEA).

Contents

1 Introduction

After a discussion on the impact of 3D on programing models, the remaining of this report will provide the syntax and the semantics of the PRO3D programming model, and a detailed description of the translation tool for application software into the intermediate format.

1.1 Impact of 3D on Programming Models & Runtimes

The *programming model* is the set of formalisms provided to the programmer to write his *application*. Once writen, the *compiler* translates an application into binary code executable by the target *architecture*. The compiler also provide a *runtime* for the platform, made of some binary code libraries called by the compiled binary code.

Although there need not be anything in common between programming models and the target architecture, there is a historical trend to have a tight relationship between the programming model and its target architecture: Lisp machines for Lisp programming in the '80, uniprocessors with flat memories for C since the 80', vectors processors at the time of Fortran 77, multiprocessors with NUMA memories for Fortran 95, transputers for CSP [1], etc.

As 3D is an innovation from *microelectronics*, i.e. from the architecture side, one legitimate question can be "*what will be the impact of 3D on programming models?*"

The remaining of this introduction tries to answer this question by walking though 3D opportunities [2, 3], and discussing their impact on programming models – but also, as we will see, on the runtime.

1.1.1 3D Stacked Memory

This 3D stack is made of a generic SMP-based SoC (e.g. dual-core ARM A9 for smartphones) for which its memory would be stacked and accessed through TSVs.

In this case the only thing visible from the programming side is a much higher bandwidth to the RAM, lower latencies, larger cache or a combination thereof [4]. Such evolution can be reduced to a shift of the architecture on the RISC/CISC axis towards better memory performance. This can be addressed by an update of the compilation algorithms with these new memory parameters and some performance analysis. The impact on the programming model as such, is therefore minimal, and von-Neumann programming languages such as C/C++ are still applicable.

1.1.2 3D Manycore with Distributed Memory

NoC-Based Manycore architectures, 2D or 3D, are more efficiently addressed by programming models that do not assume a global shared memory. E.g. Process networks and message-passing programming models are more suitable to such architectures, even though no 3D stacking is involved yet, as they are more amendable to an explicit mapping phase during compilation.

For such architectures, going 3D leads basically to two more architectural options [3]: (1) vertical NoC routing, and again, (2) memory stacking. These options can mostly be addressed in the compilation flow, as their main impact is for (1) vertical NoC to introduce different latencies and bandwidth for horizontal and vertical links; and for (2) stacked memory, again much higher bandwidth, lower latencies, larger cache or a combination thereof.

1.1.3 On Architecture Topologies and Thermal Aspects

The main changes discussed so far are a retrofit of the topological characteristics of the architecture into the compilation flow. Somehow, this is very similar to the issues encountered in

HPC with distributed machines in the early '90. The problem is difficult, but a wide body of literature exists.

The new issues introduced by 3D stacking are mostly related to thermal issues. Theses issues have two main origins:

1. *Thermal cross-coupling of execution units.* The relative position of processing units as a whole, or computing units from therein (operators, instructions decoders, register files, caches, etc.) and memory defines how heat from one element impacts another one. If two processors are too close to each other, we may have to offload both of them in situations where a single one could have run without harm. So not only the topoplogy of the manycore will have to be know from the compilation flow and the runtime, but also the geometry and thermal characteristics of the hardware [5];

2. *Different time scale for thermal propagation and computation forecast.* Manycore architecture are in the GHz range, while the evolution of the temperature is in the Hz range. This means several order of magnitude between the cause of heating –computations– and heating itself [6]. This gap in dynamic magnitude is reinforced by the fact that even at constant frequency, energy consumption increases with temperature. All this makes it difficult to reverse temperature variations. Any decision related to thermal management will probably have to use predictive thermal models [7].

1.1.4 Impact on Runtimes & Programming Models

The consequence of 3D stacking on the runtime and the programming model are twofolds:

- *The fading of pure static compilation.* Due to the huge gap of time scale between computation and thermal effects, it seems very difficult, if doable at all, to build full-static compilation schemes where the compiler will decide of the mapping off-line, before execution, once and for all.

  At least to ensure platform's thermal integrity, some level of responsibility w.r.t. mapping must be left to the runtime [8]. To ensure this integrity the runtime will have to deal with tasks scheduling and resource allocation while taking into account not only the architecture's topology and the computation load [9], but also the actual geometry and thermal characteristics of the material involved in the architecture [10].

  This will require programming models that can provide enough flexibility at execution whereas essential properties can be guaranteed at compile-time [11].

- *The fading of von Neumann as a programming model.* As for programming models, we should move away from von Neumann –only as programming model, not as computing architecture– and consider other kinds of programming models naturally parallel, like process network and message passing already discussed [12, 13, 14].

  Even these parallel programming models must be checked to be amendable to analyses that can predict the amount of computing load, if not to an absolute time reference, at least towards a moving horizon. This is necessary to provide computation forecasts to a runtime scheduler that can efficiently use the stacked architecture while preserving its thermal integrity.

2 PRO3D Programming Model

DOL3D is used as the programming model in PRO3D. This section gives an overview of the DOL3D application specification. For a complete description, refer to deliverable D2.1 [15]. As stated in the deliverable D2.1 [15], we use Synchronous Dataflow (SDF) [16] as the model of computation in PRO3D. In DOL3D, an application specification consists of (a) structural specification which specifies the structure of SDF, and (b) functional specification which specifies the functionality of each actor in the SDF. We now describe each of these specifications.

2.1 Structural specification of an application:

The structure of an application is defined by the structure of the SDF. With respect to the structure, SDF is a directed graph whose nodes represent actors (processes) and whose directed edges represent communication channels between actors.

In DOL3D, SDFs are syntactically represented in an XML file comprising of an interconnection of the following elements: `process`, `sw_channel`, `connection` and `port`. An example of such representation is shown in Figure 1, where a channel S between processes P and C (Figure 1(a)) is depicted with the corresponding ports and connections (Figure 1(b)).

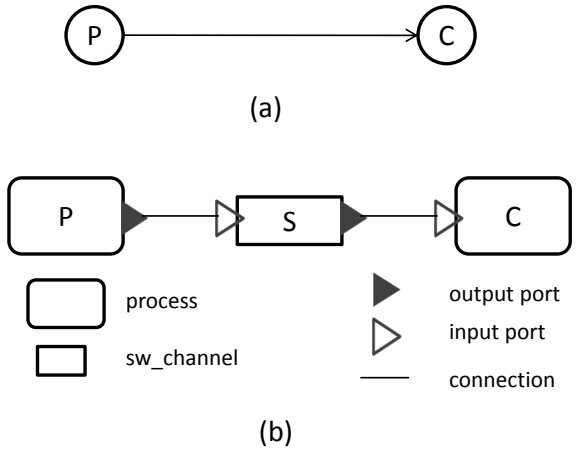

Figure 1: Process-network representation

To simplify specifications of large and regular SDF graphs, DOL3D provides the use of iterators in the XML specification. With the help of iterators, a variable number of instances of processes or channels can be created and interconnected by specifying them only once. For examples using such iterators refer to deliverable D7.1 [17].

Each of the above elements in the XML file has specific parameters. For instance, a `process` has parameters that assign it a unique name, the source code which specifies its functionality, and a list of input/output ports. In addition, other details can be annotated during the toolflow. For instance, after the functional simulation, ports of each process are annotated with the number and the size of tokens through them. Also after calibration, the obtained estimates of execution times can also be annotated in the specification of the process.

For a detailed listing of the XML file and explanation of the different elements refer to deliverable D7.1 [17].

2.2 Functional specification

The behaviour of the DOL3D application is given by the functionality of the processes. In DOL3D, the functionality of the processes is written in separate C/C++ files. These files

contain two methods, namely `DOL3D_init()` and `DOL3D_fire()`. `DOL3D_init()` is executed once at the beginning and `DOL3D_fire()` is executed repeatedly until the application is terminated with the `DOL3D_detach()` command.

The communication between processes is through the communication API which consists of two primitives `DOL3D_read()`, and `DOL3D_write()`. The semantics of these primitives would depend on the platform-specific implementation.

As the model of computation is SDF, the total number of tokens consumed or produced by an actor on each channel must remain the same across all firings. In DOL3D, we interpret consuming (producing) a token as a single call to the primitive `DOL3D_read()` (`DOL3D_write()`). Hence, in the number of `DOL3D_read()` and `DOL3D_write()` calls must be constant for each actor across firings. However, note that the amount of data read (written) by `DOL3D_read()` (`DOL3D_write()`) can vary across firings. Validation if a given application satisfies this property is performed by the DOL3D tool-chain, as discussed in [15].

3 Intermediate Format

The BIP component framework is used as an intermediate format for application software described in DOL3D.

The BIP component framework [12] is entirely supported by the BIP language and its associated toolset. This includes translators from various programming models, verification tools, source-to-source transformers and C/C++-code generators for BIP models.

3.1 Syntax

The BIP language is a notation which allows building complex systems by coordinating the behavior of a set of *atomic* components. BIP uses *connectors*, to specify possible interaction patterns between components, and *priorities*, to select amongst possible interactions. The main syntactic elements are the following:

- *atomic* components, with behavior specified as a set of transitions labeled by *ports* which are action names used for synchronization.
- *connectors* used to specify possible interaction patterns between ports of components.
- *priority* used to select amongst possible interactions according to conditions depending on the state of the integrated atomic components.

Atomic components: An atomic component consists of:

- A set of *ports* $P = \{p_1 \dots p_n\}$. Ports are action names used for synchronization with other components.
- A set of *control locations* $S = \{s_1 \dots s_k\}$. They denote locations at which the components await for synchronization.
- A set of *variables* V used to store (local) data.
- A set of transitions modeling atomic computation steps. A transition is a tuple of the form (s_1, p, g_p, f_p, s_2), representing a step from control location s_1 to s_2. It can be executed if the guard (boolean condition on V) g_p is true and some interaction including port p is offered. Its execution is an atomic sequence of two microsteps: 1) an interaction including p which involves synchronization between components with possible exchange of data, followed by 2) an internal computation specified by the function f_p on V. That is, if v is a valuation of V after the interaction, then $f_p(v)$ is the new valuation when the transition is completed.

In BIP, variables and functions are described in C/C++. Figure 2 shows an atomic component

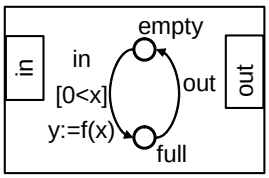

Figure 2: An atomic component

with two ports *in*, *out*, variables x, y, and control loations *empty*, $full$. At control location

empty, the transition labeled *in* is possible if $0 < x$. When an interaction through *in* takes place, the variable x is eventually modified and a new value for y is computed. From control location $full$, the transition labeled *out* can occur. The omission of guard and function for this transition means that the associated guard is *true* and the internal computation microstep is empty.

The description of coordination between atomic components is layered. The first layer describes the interactions between components by using connectors.

Connectors: A connector γ is a set of ports of atomic components which can be involved in an interaction. We assume that connectors contain at most one port from each atomic component. An interaction of γ is any non empty subset of this set. For example, a connector γ with ports p_1, p_2, p_3 has seven interactions: $p_1, p_2, p_3, p_1p_2, p_1p_3, p_2p_3, p_1p_2p_3$. Each non trivial interaction i.e., interaction with more than one port, represents a synchronization between transitions labeled with its ports. Further, connectors are associated with a typing mechanism on its ports to specify the feasible interactions of a connector γ, in particular to express the following two basic modes of synchronization:

- Strong synchronization or rendezvous, when the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ. In this case, each port of the connector is of type *synchron*.
- Weak synchronization or broadcast, when feasible interactions are all those containing a particular port which initiates the broadcast. In this case, the initiator ports have type *trigger* and the rest are typed *synchron*. That is, if γ contains ports p_1, p_2 and p_3 and p_1 is the *trigger*, then the feasible interactions are $p_1, p_1p_2, p_1p_3, p_1p_2p_3$.

Additionally, each interaction α of a connector γ may have associated guard G_α (enabling condition) and data transfer F_α specified by methods $up()$ and $sown()$, in order to implement data exchange between the interacting components. The execution of α is possible if G_α is *true*. It atomically changes the global valuation v of the synchronized components to $F_\alpha(v)$. As for atomic components, guards and functions are C expression and statements respectively. We

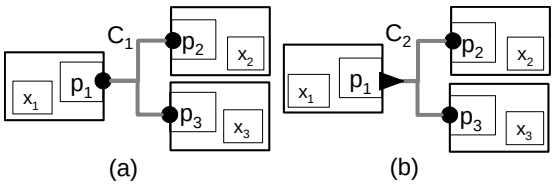

Figure 3: Interaction types

use a graphical notation for connectors, where *synchron* ports are represented by bullets and *trigger* by a triangle. The connector in figure 3(a) models a strong synchronization between the ports p_1, p_2 and p_3, while the one in figure 3(b) models a broadcast initiated by p_1.

Priorities: The second layer describes dynamic priorities that provide a mean to coordinate the execution of interactions within a BIP system. They are used to express scheduling policies between simultaneously enabled interactions. More concretely, priorities are rules, each consisting of an ordered pair of interactions associated with a condition. When the condition holds and both interactions of the corresponding pair are enabled, only the one with higher-priority can be executed.

The combination of interactions and priorities characterizes the overall architecture of a component. It confers BIP strong expressiveness that cannot be matched by other languages [18].

Composite components: Composite components are defined by assembling sub-components (atomic or composite) using connectors and priorities. Figure 4 shows a graphical representation of a BIP model. It consists of atomic components *Sender*, *Receiver1* and *Receiver2*. The behavior of *Sender* is described as an automaton with control locations *Idle* and *Active*. It communicates through port s which exports the variable x. Components *Receiver1* and *Receiver2* are composed by the connector *C1*, which represents a rendezvous interaction between ports *r1* and *r2*, leading to the composite component *Receivers*. The composite component exports *C1* by using port r. As a result of the data transfer in *C1*, the sum of the local variables *y1* and *y2* is exported through the port r by using variable v. The interaction is completed by assigning v to variables *y1* and *y2*. The model is the composition of *Sender* and *Receivers* using the connector *C2* which represents a broadcast from the *Sender* to the *Receivers*. When the broadcast occurs, as a result of the composed data transfer, the *Sender* gets the sum of *y1* and *y2*, and each *Receiver* gets the value x from the *Sender*.

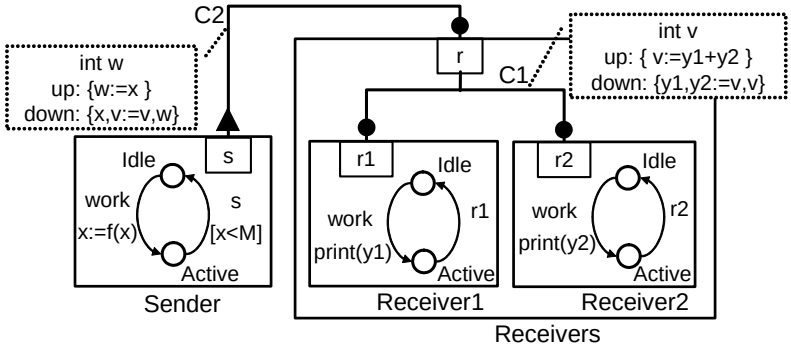

Figure 4: An example of a BIP model

3.2 Semantics

BIP has clean operational semantics that describe the behavior of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

We focus on aspects that are crucial for understanding the underlying mechanism for execution of interactions between a set of components. For the sake of simplification, we consider components without priorities. These are only a filtering mechanism for interactions. Consider a composite component built from n atomic components by using a set of connectors and without priority rules. Its meaning is an extended automaton with:

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

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

- $s = (s_1, \ldots, s_n)$, s_i being a control state of the i^{th} atomic component.

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

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

4 Translation Tool

4.1 Design Decisions

The application specification in DOL3D is translated into an application model in BIP. The specification consists of a structural description of the application, syntactically represented in XML, while the functionality is defined by the function of the processes in C/C++. The translation takes into account the structural and the functional specification of the application. It takes as input the XML description of the structure, where all iterators have already been pre-processed and flattened. The process behavior is accepted as plain C code whereby a set of coding rules needs to be respected. Each element in the application model, namely *process*, *software channel*, and *connection* is translated into a corresponding element in the BIP model. Processes and software channels are translated into BIP atomic components, while connections are translated into BIP connectors.

4.2 Functional Architecture

Figure 5 shows the architecture of the tool. It consists of an XML parser and a C parser. The former reads the structural specification, while the later is used to parse the process behavior. The parsed structure and behavior information are stored as an intermediate data structure, from which the BIP model of the application is generated by the translation step. The translation uses a library of pre-defined BIP elements for generation of the software BIP model. It contains the definition of input and output port types for the processes, connector type definitions and atomic type definition of the software FIFO channel.

The tool has been entirely developed in Java with models represented in emf [19]. For the C parsing, the open source C parser codegen [20] has been used.

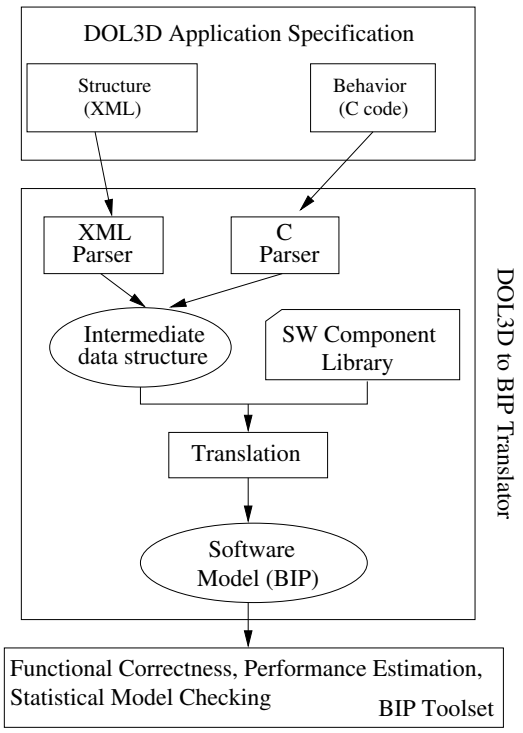

Figure 5: Tool Architecture

4.3 Translation and Intermediate Format Generation

The main structural elements in DOL3D are *process*, *sw_channel*, and *connection*, and each is translated to a corresponding element in the BIP model.

Translation of process: The translation converts each `<process>` to an atomic component in BIP.

```
<process name="pross">
  <port type="input" name="inPort"/>
  <port type="output" name="outPort"/>
<source type="c" location="pross.c"/>
</process>
```

The description of a `<process>` specifies its `<port>`. Each port is defined as a port in the atomic component. In BIP, we have predefined port types, one for type `"input"` and the other for `"output"`. Each port is associated with variables, needed to implement the transfer of data during read() and write(). Two variables are created, one contains the address of the data, while the other contains its size.

The behavior of the process is translated to BIP from the source C file specification.

Translation of process behavior: The behaviour is abstracted into two methods, namely init() and fire(), defining the initialization phase and the execution phase respectively. The init() function is called once at the initialization of the application. The purpose of this function is to initialize a process before running. The fire() function is called repeatedly by the scheduler during runtime, by which the actual behaviour of a process is activated. There can be other methods, which can be called from fire() and init(). The function detach() can be used to terminate the execution of a process. The communication between processes is unified into a set of two functions, i.e. DOL3D_read(), and DOL3D_write().

The interface to the internal state and the behavioural functions of a process is abstracted in the C structure DOL3D_Process, which contains pointer to a structure that stores local information of a process. All the C structures are parsed and defined as local data in the process component. Local variables defined in the init() and fire() functions are directly translated as data in the process component.

The translation starts by parsing the C source code of a process to an itermediate object model. The translation to BIP is done in two steps using the visitor pattern on the object model. In the first step, the interaction points in the code are identified. Each call to a DOL3D_read()/DOL3D_write() primitive is registered as an interaction point.

The second step involves the generation of an automata from the C statements. The fire() routine is taken as the entry point for the generation of the automata. Control locations correspond to DOL3D_read()/DOL3D_write() primitive calls, for which synchronization is required. For every call to a DOL3D_read()/DOL3D_write(), a control location is created. An outgoing transition is added from this location, labeled by the port used in the primitive. This transition models the primitive call, which requires synchronization with a software FIFO component. The port of a transition is associated with data that is read/written by the primitive invocation. Additional assignment statements are added to read/write these data into local variables in the function.

A block statement that contains interaction points is transformed into sequence of control locations and transitions in the automaton. For such statements, e.g., conditional (if-else, switch) or loop (for, while) or control satatement (break, continue, return), additional control

locations are created and internal transitions guarded by the control condition are added to model the control automaton.

For a conditional statement, a new control location is created with an incoming transition where the branch condition evaluation action is added. Outgoing transitions, one for the positive branch and another for the negative branch are created. The branches are finally merged to a new control location.

For a loop statement, a new control location is created with an incoming transition where the loop initialization action and exit condition are added. Outgoing transitions, one for the positive exit condition and the other for the negative exit condition are created. For the negative exit branch, a transition back to the starting location of the loop is added, with the exit condition updation action.

Statements that do not contain interaction points are added as actions to the existing transition. Subroutine calls that contain DOL3D_read()/DOL3D_write() primitive calls (either directly or through nested subroutines) are flattened.

For the termination primitive DOL3D_detach(), a control location with an incoming transition and without any outgoing transition is created.

From the last control location of the fire() method, a transition to its starting contol location is added. This models the repeated call of the fire() method at runtime. The automaton generated from the translation of the init() method is added to the starting location of the fire() method.

Restrictions of the translator: The translation requires analysis of arbitrary C code and hence is non-trivial. For the current implementation, we assume that the behavior of processes are specified in C (the codegen parser is a C parser), with the following coding rules:

- in a switch statement, all case must end with a break statement
- a subroutine must have a single return statement
- DOL3D_read()/DOL3D_write() cannot be invoked in recursive subroutines
- no shared variables can be used between processes. Any global variable associated to a process is translated to a local variable of the process component
- no use of *goto* statement

An example of the translation of the function $pross\_fire()$ into an atomic component in BIP is shown in figure 6. The function defines local integer variables i, j, k and $size$. Its invocation reads data from port $inPort$ of size $size$ into the variable i. It then performs an iteration of local computation and writes to port $outPort$ the value k.

The BIP component generated from $pross\_fire()$ has ports $inPort$, $outPort$ and control locations $L1$, $L2$ and $L3$. i, j, k and $size$ are defined as variables in the component. i is associated with $inPort$ and k is associated with $outPort$. At $L1$, the component awaits synchronization through $inPort$ corresponding to the $DOL3D\_read()$ primitive call. At $L3$ it awaits synchronization through $outPort$ corresponding to the $DOL3D\_write()$ primitive call. At $L2$, the component can perform internal transitions (labeled by τ) guarded by the loop control condition.‘ Computations between successive $DOL3D\_read/DOL3D\_write$ calls in $pross\_fire()$ are added as action statements with the respective transitions in the generated automaton.

```
int pross_fire(DOL3D_process *p) {
  int i, j, k;
  DOL3D_read((void*)inPort, &i, sizeof(int), p);
  i++;
  for (j=0; j<10; j++) {
    i=i+j;
    k = compute(i);
    DOL3D_write((void*)outPort, &k, sizeof(int), p);
    i–;
  }
  return 0;
}
```

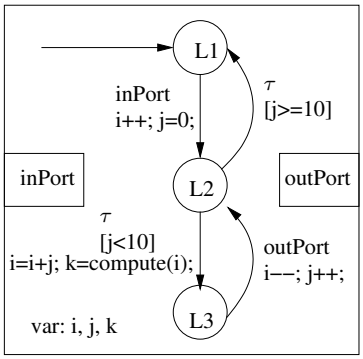

Figure 6: Translating a process into a BIP component

Translation of a sw_channel: The `<sw_channel>` element is used to describe FIFO channels in the SDF. It has three attributes `type`, `size`, and `name`. `type` describes the type of the channel. Currently, the type of a sw_channel must be `fifo`. The attribute size specifies the number of bytes a FIFO must be able to hold. Each `<sw_channel>` element must have exactly two `<port>` elements, one input and the other output. An example (from [17]) is the following:

```
<sw_channel type="fifo" size="100" name="C">
  <port type="input" name="0"/>
  <port type="output" name="1"/>
</sw_channel>
```

For a `<sw_channel>`, the behavior is modeled as a predefined BIP atomic component modeling a FIFO, represented in figure 7. It has ports *recvPort* and *sendPort*, and a single control location. It contains an array variable $buff$ parametrized by size N. The variable x associated with *recvPort* gets the received value which is inserted into $buff$. The variable y associated with *sendPort* contains the value to be read next. The FIFO policy is implemented by using the two indices i and j, for insertion and deletion respectively from $buff$.

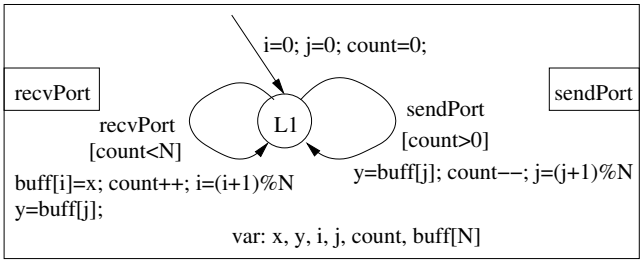

Figure 7: FIFO channel in BIP

Translation of connection: The `<connection>` element is used to establish connections between processes and the FIFO channels. More precisely, a connection is established between two ports of the corresponding elements. Connections may only connect:

- a single output port of a `<process>` with a single input port of a `<sw_channel>`, or

- a single output port of a `<sw_channel>` with a single input port of a `<process>`.

All connection specify an `<origin>` and a `<target>` to denote the two elements of the SDF that should be connected, i.e. one port of a `<process>` and one port of a `<sw_channel>` element. Each connection defines a BIP connector which strongly synchronizes the corresponding source and target ports of the respective atomic components. The connector types are predefined in the library of BIP software components. The connectors are associated with transfer of data, implementing the data transfer for *read* and *write* operations. A connector implementing a *write* transfers data from a process to a channel, whereas the one implementing *read* transfers data from a channel to a process. An example of a DOL3D application is shown in figure 8.

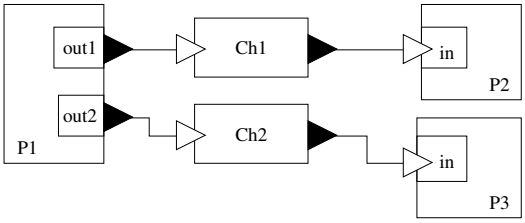

Figure 8: Application model in DOL3D

It consists of a process P1 sending data to processes P2 and P3 by using FIFO channels Ch1 and Ch2 respectively. Each connection of the DOL3D model is translated into a BIP connector in the corresponding BIP model of the application, as shown in the corresponding translated example of figure 9.

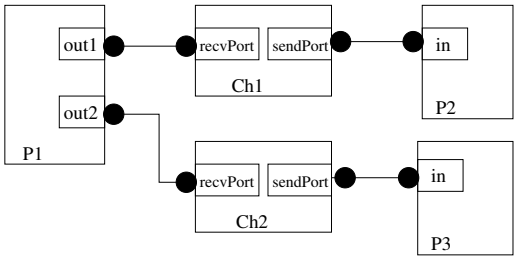

Figure 9: Application model in BIP

References

[1] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[2] Denis Dutoit and Ahmed Jerraya. 3D Integration Opportunities for Memory Interconnect in Mobile Computing Architectures. *Future Fab International*, pages 38–45, July 2010.

[3] Fabien Clermidy, Florian Darve, Denis Dutoit, Walid Lafi, and Pascal Vivet. 3D Embedded Multicore: Some Perspectives. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, Grenoble, France, March 2011.

[4] Jung-Sik Kim, Chi Sung Oh, Hocheol Lee, Donghyuk Lee, Hyong-Ryol Hwang, Sooman Hwang, Byongwook Na, Joungwook Moon, Jin-Guk Kim, Hanna Park, Jang-Woo Ryu, Kiwon Park, Sang-Kyu Kang, So-Young Kim, Hoyoung Kim, Jong-Min Bang, Hyunyoon Cho, Minsoo Jang, Cheolmin Han, Jung-Bae Lee, Kyehyun Kyung, Joo-Sun Choi, and Young-Hyun Jun. A 1.2v 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4×128 I/Os Using TSV-Based Stacking. In *ISSCC 2011*, pages 496–497, February 2011.

[5] Lothar Thiele, Lars Schor, Hoeseok Yang, and Iuliana Bacivarov. Thermal-aware system analysis and software synthesis for embedded multi-processors. In *Proc. Design Automation Conference (DAC)*, San Diego, California, USA, 2011. ACM.

[6] Mahankali Sridhar, Arvind Raj, Alessandro Vincenzi, Martino Ruggiero, Thomas Brunschwiler, and David Atienza Alonso. 3D-ICE: Fast Compact Transient Thermal Modeling for 3D-ICs with Inter-Tier Liquid Cooling. In *Proceedings of the 2010 International Conference on Computer-Aided Design (ICCAD 2010)*, New York, 2010. ACM and IEEE Press.

[7] Sabry Aly, Mohamed Mostafa, Ayse Kivilcim Coskun, and David Atienza Alonso. Fuzzy Control for Enforcing Energy Efficiency in High-Performance 3D Systems. In *Proceedings of the 2010 International Conference on Computer-Aided Design (ICCAD 2010*, New York, 2010.

[8] Francesco Zanini, David Atienza, Luca Benini, and Giovanni de Micheli. Thermal-Aware System-Level Modeling and Management for Multi-Processor Systems-on-Chip. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'11)*, 2011.

[9] Marongiu A., Burgio P., and Benini L. Vertical Stealing: Robust, Locality-Aware Do-All Workload Distribution for 3D MPSoCs . In *Proceedings of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2010)*, 2010.

[10] M. Sabry, D. Atienza, and A. K. Coskun. Thermal Analysis and Active Cooling Management for 3D MPSoCs. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'11)*, 2011.

[11] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.

[12] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *Software Engineering and Formal Methods SEFM'06 Proceedings*, pages 3–12. IEEE Computer Society Press, 2006.

[13] Joven J., Marongiu A., Angiolini F., Benini L., and De Micheli G. Exploring Programming Model-Driven QoS Support for NoC-based Platforms . In *Proceedings of the 2010 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2010)*, 2010.

[14] Marongiu A. and Benini L. An OpenMP Compiler for Efficient Use of Distributed Scratch-pad Memory in MPSoCs. *IEEE Transactions on Computers*, 2010.

[15] PRO3D consortium, D2.1: Report on Process-Level Mapping & Tool Support in DOL3D. 2010. Consortium confidential document.

[16] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[17] PRO3D consortium, D7.1: 3D Requirements Synchronisation for Technical Work Packages. 2010. Consortium confidential document.

[18] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In *Concurrency Theory CONCUR'08 Proceedings*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.

[19] http://www.eclipse.org/modeling/emf.

[20] http://think.ow2.org.