PAPER

A Model of Computation for Bit-Level Concurrent Computing and Programming: APEC*

Takashi AJIRO^{†a)}, Student Member and Kensei TSUCHIDA^{†b)}, Member

SUMMARY A concurrent model of computation and a language based on the model for bit-level operation are useful for developing asynchronous and concurrent programs compositionally, which frequently use bit-level operations. Some examples are programs for video games, hardware emulation (including virtual machines), and signal processing. However, few models and languages are optimized and oriented to bit-level concurrent computation. We previously developed a visual programming language called A-BITS for bit-level concurrent programming. The language is based on a dataflow-like model that computes using processes that provide serial bit-level operations and FIFO buffers connected to them. It can express bit-level computation naturally and develop compositionally. We then devised a concurrent computation model called APEC (Asynchronous Program Elements Connection) for bit-level concurrent computation. This model enables precise and formal expression of the process of computation, and a notion of primitive program elements for controlling and operating can be expressed synthetically. Specifically, the model is based on a notion of uniform primitive processes, called *primitives*, that have three terminals and four ordered rules at most, as well as on bidirectional communication using vehicles called *carriers*. A new notion is that a carrier moving between two terminals can briefly express some kinds of computation such as synchronization and bidirectional communication. The model's properties make it most applicable to bit-level computation compositionally, since the uniform computation elements are enough to develop components that have practical functionality. Through future application of the model, our research may enable further research on a base model of fine-grain parallel computer architecture, since the model is suitable for expressing massive concurrency by a network of primitives.

key words: model of computation, visual programming, bidirectional communication, concurrent, bit-level

1. Introduction

A concurrent model of computation for bit-level operation is useful for developing asynchronous and concurrent programs, which frequently use bit-level operations. Some examples are programs for video games, hardware emulation (including virtual machines), and signal processing. However, few models and languages are optimized and oriented to bit-level concurrent computation. Such a model and language, though, allow easier development and briefer expression of bit-level concurrent programs than with languages based on general operation-level (integer, string, etc.) computation.

Demands for concurrent programming have increased since recent computer systems and programs require asynchronous and fine-grain concurrency. Especially, the use of applications such as those mentioned above has increased as the performance of computer systems has improved. For instance, the implementation of emulation programs that correspond to the architectures of legacy computer systems is desired. We consider such applications are suitable for development through the composition of atomic bit-level operations, since the many parts of these programs require fine-grain concurrency and the treatment of bit-level data of variable size.

In the first stage of our research, we developed a bit-level concurrent visual programming language (VPL), called A-BITS [1], [2], for developing programs based on bit-level concurrent computation. It provides only binary data-types and functions for composing programs, since it is based on the premise that a program consists of smaller components based on bit-level computation. Therefore, the language specification is simpler and learning is easier than with general operation-level languages. The language is based on a dataflow-like model that computes using processes which provide serial bit-level operations and FIFO buffers connected to them. The properties of the model make it possible to express bit-level computation naturally and to develop it compositionally.

However, the base model of computation for A-BITS makes it difficult to express the process of computation precisely and formally, since the base model of computation for the language does not incorporate uniform primitive computation elements for controlling, operating, and synchronizing. In addition, the model makes it difficult to briefly and simply express synchronizations and bidirectional communications as well as other dataflow-like models. This is not a serious problem for many general programs based on dataflow-like models, since their main applications are stream-oriented computations and data-driven computations. However, bit-level concurrent programs generally need many operations for synchronization and bidirectional communication. Thus, it is important to solve the problem since bit-level concurrent programs can be expressed more briefly and simply.

We devised the APC (Ajiro Program Circuit) model [2] as a concurrent model of bit-level computation to solve the problems, and to be the base model of the next version of A-BITS. In addition, we developed an APC simulator [2] that edits and simulates APC circuits (networks) visually with a

Manuscript received March 15, 2007.

Manuscript revised August 7, 2007.

[†]The authors are with the Department of Open Information Systems, Toyo University, Kawagoe-shi, 350-8585 Japan.

*This work was partially presented at "The IEEE Symposium on Visual Languages and Human-Centric Computing [VL/HCC '05]" in Dallas, Texas, USA, 20-24 September 2005 [2].

a) E-mail: t.ajiro@bitprog.info

b) E-mail: kensei@toyonet.toyo.ac.jp

DOI: 10.1093/ietisy/e91-d.1.1

GUI to make it easier to understand the model of behavior. We slightly modified the APC model, and changed its name to "APEC" (Asynchronous Program Elements Connection). The essential difference between the APC model and the APEC model is in the treatment of an unconnected terminal. The former treats it as an isolated connection, while the latter treats it as a looped connection. The APEC's foremost advantage over other models is that it makes it possible to synthetically express communication and synchronization using a notion of a carrier. In addition, this notion reduces the number of interfaces needed for connections since half-duplex bidirectional communication can be naturally expressed. The benefit of this is that it allows the development of composed components from primitive elements, and easy connection between components through bit-level programming.

Regarding future application of the model, our research may lead to further research on a base model of fine-grain parallel computer architecture [23]–[26]. A notion of primitives can enable implementation of simple processing elements of a fixed size, so the model is suitable for expressing massive concurrency in a network of primitives.

The remainder of the paper proceeds as follows. In Sect. 2, we introduce the basic concept of the APEC model and the diagrammatic notation. Section 3 explains the formal definition that corresponds to the notation. Section 4 presents some APEC primitives to compute basic operations, and some important compositional networks using these elements. In Sect. 5, we present the features and advantages of the APEC model. Section 6 explains an example of an APEC system that acts as a simple 4-bit computer, and describes a simulator of the model. In Sect. 7, we discuss related work concerning other models. Section 8 concludes the paper and looks at future work.

2. Basic APEC Concept

Primitive processes of APEC are called *primitives*, and they behave asynchronously and independently of each other. Each primitive has three ports at most, called *terminals* for sending and receiving data. Data are sent or received using vehicles called *carriers*, and the carriers move between two terminals or remain on them to be operated by primitives. A terminal can be connected to another terminal by only one carrier, and more than one carrier cannot share one terminal.

Each primitive has four ordered rules at most, which are used to operate the remaining carriers on terminals. Each carrier must be in the *remaining* or the *leaving* state. The former is the state in which a carrier remains on a terminal, and the latter is the state in which a carrier is leaving a terminal. In other words, the leaving state means that a carrier is moving between two terminals. A definition of primitives and their connections is called a *network*. The network state is determined by the states of all carriers. In other words, primitives have no internal states.

The maximum number of terminals is limited to three, and the maximum number of rules is limited to four at most.

These limitations are needed to define primitives as processing elements that have fixed size in an actual implementation and that provide only atomic functions. Primitives without any limitations can be considered, but such elements have unfixed size and are non-uniform in complexity. The limitations do not reduce the needed capability as a model of bit-level computation, though, and we will show by an application in a later section that composed networks using the limited primitives are capable of expressing most general computations.

Detailed concepts are explained individually in the following subsections.

2.1 States of Carriers

In general, an APEC network is represented and discussed using the diagrammatic notation. The diagrammatic notation uses a diagram and rule tables to express the network structure, carrier states, and the behavior of primitives. Carrier states and a corresponding representation using diagrammatic notation are shown in Fig. 1. Rectangles are primitives, a pipe between rectangles is a connection, a circle on a pipe is a carrier, and the letters beside the primitives identify each primitive.

Terminal "x" of primitive "A" is represented as "A.x". A representation enclosed in a dashed rectangle is a diagrammatic notation of the picture above it. A carrier remains on "A.x", and "1" is the value the carrier has in (a). In (b), a carrier leaves "A.x" and is moving. "A" has no carrier in (e), and terminals without carriers do not need to be displayed. A carrier remains on "A.x" which connects to itself (looped connection) in (f).

2.2 Behavior of Primitives

The behavior of each primitive is defined by rule tables. A rule table contains a number of rules used to operate terminals owned by a primitive. Rules have a certain form, and this is shown in Fig. 2. In (a), the "JOIN" rule table is illustrated. Values on the left side of the table are called *condition-values*, and they are used as conditions to operate actions represented on the right side. These values for actions are called *action-values*.

Each action-value of the terminals is applied to each carrier if each condition-value of the terminals is equal to

Fig. 1 States of carriers and corresponding diagrammatic notations.

Fig. 2 Rule table of "JOIN" and a primitive that has the rule.

the corresponding values of the remaining carriers. Action-value "X" means the value of a carrier is changed to "X", and action-value "X" means the carrier is changed to "X" and sent, where X is 0 or 1. Furthermore, a condition-value of "*" means the terminal is ignored, so the corresponding action-value must also be "*". The value of a leaving carrier does not satisfy the condition, since a condition-value is compared to only a remaining carrier. Only one applied rule is determined on a rule table, since a rule that has a higher position is given higher priority. For instance, in the "JOIN" table, the first rule means that carriers on "x" and "z" are sent when the carriers arrive with a value of "0". The third rule uses only terminal "y" and "z". However, the first rule is applied due to priority if arrival carriers are remaining and have "0" values on all terminals.

The states of the primitives with "JOIN" are represented in (b). The right image is the state after the second rule of "JOIN" is applied. As indicated, the value on "x" is copied onto "z" and is reset (become "0"), and the carriers enter the leaving-state.

2.3 Transition

A change in the network state is called a *transition*. There are two types of transition: *primitive-transition* and *carrier-transition*. In the former, a primitive operates carriers that are in the remaining state according to the rules. In the latter, a carrier enters the remaining state as a result of its arrival at the destination. Only one transition occurs concurrently, but the notion is not related to concrete time. A state in which no transitions can occur on the network is called a *finished* state. The meaning of a finished state is not defined in the model since the APEC model only concerns the concept of atomic bit-level operations. In general, this state is defined as indicating completion of a computation in a meaningful network.

A primitive-transition can occur whenever the terminals of the primitive satisfy at least one rule on the rule table, and it operates the carriers according to the applied rule. A carrier-transition can occur whenever the carrier is in the leaving state, and it sends the carrier to the destination terminal and enters it into the remaining state.

Here, $S \rightarrow p_1$ represents a network state in which the primitive-transition of "p1" occurred, where S is the previous state of the network. Similarly, $S \rightarrow p_1.x$ represents a network state in which carrier-transition "p1.x" occurred. Here, $S \rightarrow p \rightarrow p.x$ is interpreted as $(S \rightarrow p) \rightarrow p.x$, so the

Fig. 3 Simple APEC network.

carrier-transition $p.x$ changes the state of $(S \rightarrow p)$. Such a representation is called a *transition-sequence*.

A simple network and its transitions are shown in Fig. 3. The rule table name is displayed in a rectangle as a primitive, which has those rules. For instance, "c1" has the rule "CST" (ConSTant generator) since "CST" is displayed in the rectangle of primitive "c1". This network indicates that a constant value of "1" is sent, and that primitive "c1" sends a value "1" to "d1" using a carrier on "c1.z". The behavior of "d1" is defined by the "DMY" (DuMmY) rule table, but the primitive does nothing in the network. The first state is the initial state, the second state is the state after the primitive-transition of "c1", and the final state is the state after the carrier-transition of "c1.z". The transition-sequence is represented as $S \rightarrow c_1 \rightarrow c_1.z$.

There is no notion of transitions that occur concurrently in the model. However, such transitions can be expressed by a sequence of the same kind of transitions, since a transition does not affect the continuation of other transitions of the same kind. This is because a primitive operates only attached carriers and a moving carrier does not interfere with other carriers. This property is needed to express the independence of the behavior of primitives, since the model is designed to express a fixed network of simple processing elements connected only by carriers. A sequence of the same kind of transitions means either concurrent occurrence or independent timely sequential occurrence in the actual world. Also, a sequence of different kinds of transitions means either independent occurrence (only if primitives or carriers do not relate to each other) or true sequential occurrence. For instance, the result of $S \rightarrow p_1 \rightarrow p_2 \rightarrow p_2.x \rightarrow p_1.y$ is equal to $S \rightarrow p_2 \rightarrow p_1 \rightarrow p_1.y \rightarrow p_2.x$. Changing transitions p_1 and p_2 or $p_2.x$ and $p_1.y$ produce the same result, but changing transitions p_2 and $p_2.x$ might not give the same result (or such changed transitions might not exist). In general, the arrival of carriers may change the applied rule of primitives because of rule priority.

2.4 Relation between APEC and Our Previous Work

We previously developed bit-level VPL A-BITS [1], [2] as the first stage of research for bit-level concurrent programming. A process network composed of bit-streams and simple state machines can be developed using the A-BITS language. The property of the base model is similar to existing dataflow-like models [12]–[17]. In addition, A-BITS is similar in appearance to dataflow VPLs [3]–[5] and a language using streams [6], [7]. However, our language differs from them with regard to specifying functions for only

bit-level operations and developing programs by composing. Although there are visual languages based on binary operation [8], [9], they were developed for digital circuit design. Our language is unrelated to practical hardware and is oriented toward developing software designed on a bit-level.

As the next stage, we devised APC model [2] since it is difficult to design a synchronization mechanism in programming with the base model of A-BITS. For instance, an individual route with some operations for sending an acknowledgement signal is needed if starting an operation indirectly depends on completion of previous operations. The APC model solved the above problem by bidirectional communications using carriers and rule-based processing elements without internal states. Such a route is not generally needed since the operations are weaved with in the data operation process.

The APEC model is a refined version of the APC model, and two main modifications are as follows. First, in the APC model, carriers on non-connected terminals have special connections which do not have destination terminals, and such carriers are called *isolated carriers*. However, such terminals are instead defined as looped terminals in the APEC model. This modification allows a briefer definition of the model by excluding the anomalistic notion. Second, the formalization and the description of the model have been refined. As a consequence, the names for some notions have been changed: a transition is called an *application*, and a network is called a *circuit*. Changing the state of a circuit through an application is represented by a form such as $p_1 + p_2$. According to the modifications, the APEC simulator we will explain later is a modified version of the APC simulator [2] which reflects the APEC definition.

3. Formal Definition of APEC

We do not assume to use of the formal definition of the APEC model to explain the meanings of program (computation) components — that is, the behaviors of primitives and components — since it corresponds to the diagrammatic notation and the mathematical expression is complex. However, formal definition is needed to precisely express the behavior of bit-level computation on the APEC model. For instance, although the behavior of a network is often expressed by a description of a transition-sequence, which is easy to understand intuitively as in the previous section, we provide a formal definition of it in the APEC model to eliminate any ambiguity regarding its meaning. Hence, we introduce and explain a formal definition in this section.

We first introduce and define the gAPEC (general APEC) model as it is the general model based on communication using carriers and asynchronous concurrent operations by primitives with ordered rules. The APEC model is defined as a restricted model of gAPEC since the notion of communicating by carriers can be applied to other applications (for instance, computation using multiple values).

3.1 gAPEC Model

As a preliminary, we premise that an operator $\times$ implicitly concatenates two tuples or elements, and that nested tuples are also concatenated. For instance, an element of $\{(p, t)\} \times \{n\}$ is (p, t, n) , and $((p, t), n)$ is (p, t, n) .

A network M of gAPEC is defined by the tuple $M = (P, T, V, R, S_I)$. The sets of the tuple are defined as follows:

P : a finite set of primitives,

T : a finite set of terminals,

V : a finite set of carrier values,

$R \in R_{all}$, where R is a set of rules of primitives.

$R_{all} = \{X \subseteq P_T \times \mathbb{N} \times V \times V_A : \forall \alpha \in P_T \times \mathbb{N} [|(\{\alpha\} \times V \times V_A) \cap X| \leq 1] \}$,

An element of R corresponds to a part of one rule of a primitive. The meanings of the elements of R are as follows. The term P_T is defined as $P \times T$ and means a set of pairs of a primitive and a terminal. $\mathbb{N}$ is defined as $\{0, 1, 2, \dots\}$ and means a set of ordering numbers of a rule, where a smaller ordering number has a higher priority. The term V is a set of values called *condition-values*. The term V_A is a set of pairs, each pair consisting of a value and a behavioral state that can change the state of a carrier if the condition permits. Pairs of V_A are called an *action-values*, where $V_A = V \times A$ and $A = \{\tau_{rm}, \tau_{lv}\}$. Here, τ_{rm} means *remaining*, and τ_{lv} means *leaving*.

$S_I \in S_{all}$, where S_I is an initial state of M .

$S_{all} = \{X \subseteq P_T^2 \times V_A : \forall \alpha \in P_T \forall \beta \in P_T [|(\{\alpha, \beta\} \times V_A) \cap X| \leq 1 \wedge |(\{\alpha\} \times P_T \times V_A) \cap X| \leq 1 \wedge |P_T \times \{\alpha\} \times V_A \cap X| \leq 1] \}$.

A set $S \in S_{all}$ is called the network *state*, and an element of the set is called a *carrier*. The meanings of the elements are as follows. The first pair of P_T means the destination terminal, the second pair of P_T means the current terminal, and the pair of V_A means a value and the current state of a carrier. The conditions for the set mean that a connection between two terminals must consist of one carrier at most, and a terminal must have at most one connection.

3.2 Transition of gAPEC

If we let $M = (P, T, V, R, C, S_I)$, and operating state $S \in S_{all}$, making $S' \in S_{all}$ is called a *transition*. There are two kinds of transition, *primitive-transition* and *carrier-transition*. The former is expressed by $f_c : S_{all} \times P_T \rightarrow S_{all}$. As $S \in S_{all}$, $p \in P$, $t \in T$, the results of $f_c(S, p, t)$ are defined as

$$f_c(S, p, t) = \{x : (q, u, p, t, v, \tau_{lv}) \in S, x \in ((S - \{(q, u, p, t, v, \tau_{lv})\}) \cup \{(p, t, q, u, v, \tau_{rm})\})\}.$$

Similarly, the latter is expressed by $f_p : S_{all} \times P \rightarrow S_{all}$. As $S \in S_{all}$, $p \in P$, the result of $f_p(S, p)$ is defined as

$$f_p(S, p) = \{x : (p, t, n, v_c, v_a, a) \in R, (q, u, p, t, v_c, \tau_{rm}) \in S, \alpha = \{y : \forall (p', t', n, v'_c, v'_a, a') \in R [y \in ((P_T \times \{(p, t', v'_c, \tau_{rm})\}) \cap S)] \}, \alpha \neq \emptyset,$$

$$\neg \exists n' (\in \mathbb{N}) [\forall (p, t', n', v'_c, v'_a, a') (\in R) [\\ (P_T \times \{(p, t', v'_c, \tau_{rm})\}) \cap S \neq \emptyset] \wedge n' < n], \\ x \in ((S - \alpha) \cup \{(q, u, p, t, v_a, a)\})]$$

where variables marked with a prime “'” are bound variables. For instance, in $\forall (p, t', n, v'_c, v'_a, a')$, variables bound by $\forall$ are $t' \in T$, $v'_c, v'_a \in V$, and $a' \in A$. The previous state is called *transitable* if a state mapped by f_c or f_p is not $\emptyset$. A transition that maps $\emptyset$ is called an *invalid-transition*.

Here, we introduce another representation of a state mapped by the above functions for a brief representation, which is defined as $f_c(S, p, t) \equiv S \rightarrow p.t$, and $f_p(S, p) \equiv S \rightarrow p$, where $S \in S_{all}$, $p \in P$, and $t \in T$. The representation means “a state” mapped by the functions, thus $S \rightarrow p \rightarrow p.t$ is interpreted as $(S \rightarrow p) \rightarrow p.t$ and is equivalent to the state mapped by $f_c(f_p(S, p), p, t)$. Such a sequence of transitions is called a *transition-sequence*. The result of the sequence is $\emptyset$ if a transition-sequence includes at least one invalid-transition. The state is called *finished* if the result of a transition-sequence is not $\emptyset$ and the state is not transitable.

3.3 APEC Model

APEC is a restricted model of gAPEC for bit-level concurrent computation using simple primitive processes communicated by carriers. The restrictions to gAPEC are $|T| = 3, |V| = 2, \forall (p', t', n', v'_c, v'_a, a') (\in R) [n' < 4]$. Thus, an APEC network is a gAPEC network. In general, an APEC network is defined by $T = \{x, y, z\}$ and $V = \{0, 1\}$.

As a sample expression using this definition, we here represent the simple network shown in Fig. 3. The network M is defined by the tuple $M = (P, T, V, R, S_I)$, $P = \{c_1, d_1\}$, $T = \{x, y, z\}$, $V = \{0, 1\}$, $G_{cst} = \{(y, 0, 0, 0, \tau_{rm}), (z, 0, 0, 0, \tau_{lv}), (y, 1, 1, 1, \tau_{rm}), (z, 1, 0, 1, \tau_{lv})\}$, $G_{dmy} = \{(x, 0, 0, 0, \tau_{lv}), (y, 0, 0, 0, 1, \tau_{rm}), (x, 1, 1, 1, \tau_{lv}), (y, 1, 0, 1, \tau_{rm})\}$, $R = \{(\{c_1\} \times G_{cst}) \cup (\{d_1\} \times G_{dmy})\}$, $S_I = \{(c_1, y, c_1, y, 1, \tau_{rm}), (d_1, x, c_1, z, 0, \tau_{rm})\}$ where G_{cst} and G_{dmy} are groups of rules and correspond to the rule tables. The finished state is $\{(c_1, y, c_1, y, 1, \tau_{rm}), (c_1, z, d_1, x, 1, \tau_{rm})\}$.

4. Basic Primitives and Compositional Networks

In this section, we will present some basic primitives and important compositional networks. These primitives and networks will be referred to later when we present the design of an example system in Sect. 6.

4.1 Basic Primitives

In the APEC model, an enormous number of basic primitives can be defined through the combination of rules. However, there are not many valuable primitives. Basic primitives provide functions for operations, controlling flows of data, and synchronizations. The functions can be expressed by a combination of rules synthetically.

Sending and returning a carrier is one step of a computation of a primitive in the APEC model. A carrier that has a value not used for computation is set to a default value. In general, the value “1” is set to a carrier for sending, and the value “0” is set to a carrier for returning as the default value. The protocol can emulate unidirectional communication in dataflow-like models, setting the default value to a carrier for returning. The unidirectional primitives we will explain later are designed like this.

["AND" and "DUP" primitives]

In Fig. 4, (a1) shows a network including an “AND” primitive, (a2) shows the result of transitions, and (a3) shows the finished state of the network. The dark-gray rectangles are primitives that have generated transitions lasting from the previous snapshot. The images at the left of the snapshots are intuitive representations that indicate data flows by arrows. The bold rectangles in the image correspond to the dashed rectangle indicating the function provided by the network. A “DMY” primitive sends a carrier on terminal “x”, and then uses a carrier on terminal “y” as a state variable. No transition of d_3 occurs, and the primitive is only used as a place for the carrier that has the result of the operation by a_1 . The “AND” rule table can be modified so that other primitives that behave as logic operators can be made since such operations can be defined by truth tables that relate one or two inputs to one output. For instance, “OR”, “NOT”, “XOR”, etc. A transition-sequence up to the finished state is $S_{(a1)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_1.x \rightarrow d_2.x = S_{(a2)}, S_{(a2)} \rightarrow a_1 \rightarrow a_1.x \rightarrow a_1.y \rightarrow a_1.z = S_{(a3)}$.

A network that includes a “DUP” primitive is shown in (b1). A “DUP” primitive duplicates the value of a carrier on terminal “x” to carriers on “y” and “z”. A transition-sequence up to the finished state is $S_{(b1)} \rightarrow d_1 \rightarrow d_1.x = S_{(b2)}, S_{(b2)} \rightarrow u_1 \rightarrow u_1.y \rightarrow u_1.z = S_{(b3)}$.

["JOIN" and "GT" primitives]

In Fig. 5, snapshots of the state of a network using the “JOIN” and “GT” primitives are shown. The “JOIN” primitive means that one datum is sent from either terminal “x” or “y” to terminal “z”. If there are data on both terminals, then data on “x” has priority. The matched rule of the primitive depends on a carrier-transition either $d_1.x$ or $d_2.x$.

Fig. 4 Transition snapshots of networks for bit-level operation.

Fig. 5 Transition snapshots of JOIN and GT primitives.

Transition-sequences of the snapshots are $S_{(ab)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_2.x \rightarrow S_{(a1)}, S_{(a1)} \rightarrow j_1 \rightarrow d_1.x \rightarrow j_1.y \rightarrow j_1.z \rightarrow S_{(a2)}$, and $S_{(ab)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_1.x \rightarrow d_2.x \rightarrow S_{(b1)}, S_{(b1)} \rightarrow j_1 \rightarrow j_1.x \rightarrow j_1.z \rightarrow S_{(b2)}$. The former sequence is the case in which the third rule is matched, and the latter is the case in which the first rule is matched.

The “GT” primitive means that one datum passes only if the value of an arrival carrier is “1”. The function is known as T-Gate in a dataflow-like models, and is used to select whether the data passes. Operation of a consuming token in a dataflow-like model is emulated by clearing and returning the value on “x” without copying the value onto “z”.

Transition-sequences of the snapshots are $S_{(c1)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_1.x \rightarrow d_2.x \rightarrow S_{(c2)}, S_{(c2)} \rightarrow g_1 \rightarrow g_1.x \rightarrow g_1.y \rightarrow g_1.z \rightarrow S_{(c3)}$, and $S_{(d1)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_1.x \rightarrow d_2.x \rightarrow S_{(d2)}, S_{(d2)} \rightarrow g_1 \rightarrow g_1.x \rightarrow g_1.y \rightarrow S_{(d3)}$.

["RET" primitive]

In Fig. 6, snapshots of the state of a network using the “RET” primitive are shown. The dashed arrow indicates a carrier returning with the value “0” (as the default value). A “RET” primitive is used to input a value from a route and output a value to one route, thus it is needed to compose bidirectional interfaces. Terminal “x” of the primitive is used to receive a value and then to send a value, terminal “y” is used to send a received value from the value on “x”, and terminal “z” is used to receive a sent value. A carrier on “y” is used not only as a sender, but also as a state variable to control the phase. Concretely, the state of the primitive

Fig. 6 Transition snapshots of the RET primitive.

is the first phase (waiting to receive from a carrier on “x”) if the value on “y” is “1”, and is the second phase (waiting to send to a carrier on “x”) if the value on “y” is “0”. The “TRSH” (TRaSH) primitive receives from a carrier on “x” and returns the carrier cleared to “0”. This function is generally used to trash a datum.

A transition-sequence up to the finished state is $S_{(a1)} \rightarrow d_1 \rightarrow d_1.x \rightarrow d_2 = S_{(a2)}, S_{(a2)} \rightarrow d_2.x \rightarrow r_1 \rightarrow r_1.y = S_{(a3)}, S_{(a3)} \rightarrow t_1 \rightarrow t_1.x = S_{(a4)}, S_{(a4)} \rightarrow r_1 \rightarrow r_1.z \rightarrow r_1.x = S_{(a5)}$.

4.2 Compositional Networks for Unidirectional Communication

Compositional networks are used as components in a network. Networks for unidirectional communication correspond to computations for controlling flows of data in dataflow-like models.

[Selector]

Transition snapshots of a network acting as a “Selector” are shown in Fig. 7. Image (A) in the figure is an intuitive representation that indicates the flows of data. The text in the bold rectangle indicates the component name which will be referred to later in this paper. The text enclosed by brackets “[]” show interface names that will also be referred to later.

The “Selector” component sends one datum from either of two input routes to one output route. The “WAIT” primitive waits to return a carrier after sending the carrier from “x”. A carrier on “y” is returned when a sent carrier from “z” is returned. The “RSEL” primitive sends a carrier from either “y” or “z”, depending on the value of a carrier on “x”, and waits for the return of the sent carrier.

A transition-sequence up to the finished state is $S_{(a1)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_3 \rightarrow d_1.x \rightarrow d_2.x \rightarrow d_3.x \rightarrow r_1 \rightarrow r_1.y \rightarrow w_1 \rightarrow w_1.z \rightarrow j_1 \rightarrow j_1.z \rightarrow j_1.x \rightarrow w_1 \rightarrow w_1.y \rightarrow w_1.x \rightarrow r_1 \rightarrow r_1.x = S_{(a2)}$.

[Switch]

Transition snapshots of a network acting as a “Switch” are shown in Fig. 8. The “Switch” component sends one datum

Fig. 7 Transition snapshots of a 1-bit Selector.

Fig. 8 Transition snapshots of a 1-bit Switch.

to either of two output routes. The “NT” (Negative gaTe) or “GT” primitive works as a gate that passes or clears a data. The “NT” primitive differs from the “GT” primitive only in that it passes a value if the value on “y” is “0”. “RDUP” (Responsible DUPLICator) is a special version of “DUP”, and waits for sent carriers returning sent carriers from “y” and “z” in addition to the “DUP” function. This function is useful for synchronizing the completion of distributing data to two routes.

A transition-sequence until the finished state is $S_{(a1)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_1.x \rightarrow d_2.x \rightarrow r_1 \rightarrow r_2 \rightarrow r_1.y \rightarrow r_1.z \rightarrow r_2.y \rightarrow r_2.z \rightarrow n_1 \rightarrow g_1 \rightarrow n_1.x \rightarrow n_1.y \rightarrow g_1.x \rightarrow g_1.y \rightarrow g_1.z \rightarrow r_1 \rightarrow r_2 \rightarrow r_1.x \rightarrow r_2.x = S_{(a2)}$

[Detector]

Transition snapshots of a network acting as a “Detector” are shown in Fig. 9. “Detector” component detects a carrier arriving from either routes. The value on “d3.x” is “0” when the carrier from “p1.y” has arrived on “x” and “a1” occurred. Otherwise, the value on “d3.x” is “1”. This function is used

Fig. 9 Transition snapshots of a Detector.

as part of the Arbiter component which we will later discuss.

The “ABT” primitive detects a carrier from either “x” or “y”, and sends a signal that indicates the detected route. The carrier of the detected route is locked after the signal is sent from “z”. The “PRB” primitive probes a carrier coming on “x”, and sends a signal and the value of the carrier. The value is sent from “z” and a signal that has “1” is sent from “y”. A route probed by the “PRB” primitive is locked when the primitive has probed a carrier on “x”.

A transition-sequence until the finished state is $S_{(a1)} \rightarrow d_1 \rightarrow d_2 \rightarrow d_1.x \rightarrow d_2.x \rightarrow p_1 \rightarrow p_2 \rightarrow p_2.y \rightarrow p_2.z \rightarrow a_1 \rightarrow p_1.y \rightarrow p_1.z \rightarrow a_1.z = S_{(a2)}$

4.3 Compositional Networks for Bidirectional Communication

Networks for bidirectional communication make it possible to express some kinds of operation simply and briefly. Connections for notification and data transfer are especially needed, at least in unidirectional models. Thus, the number of interfaces is more than double that of the bidirectional network in APEC.

Bidirectional components can be designed by combining the unidirectional components previously introduced.

[Bidirectional Selector and Switch]

The design of networks that act as “BiSel” (Bidirectional Selector) or “BiSw” (Bidirectional Switch) components are shown in Fig. 10. Lines attached to the bound rectangles represent interfaces to external connections, and the text close to the lines shows the interface names.

The “BiSel” component differs from “Selector” in that it waits to receive a returned carrier with a meaningful value. Similarly, the “BiSw” component differs from “Switch” in the same way. These networks are designed simply using unidirectional selectors or switches and “RET” primitives. The “RET” primitives work not only as interfaces of bidirectional communication, but also as synchronization points

Fig. 10 Bidirectional Selector and Bidirectional Switch.

Fig. 11 1-bit Arbiter and multiple-bit extension of the Arbiter.

for sending and receiving a datum.

[Arbiter]

The design of a networks acting as an “Arbiter” (1-bit Arbiter) is shown in Fig. 11. The “Arbiter” component first occupies one of two routes (“f” or “t”), sends a value of the route to the “dt” route, and then returns a value from the “dt” route to the occupied route. In other words, the component works as a bidirectional selector using the order of carrier arrival. Actually, the component is designed by connecting of one “Detector” and one “BiSel” in the figure. In addition, extension to a multiple-bit arbiter can be implemented by connecting one “Detector” and a number of “BiSel” components as shown in (B).

5. Features and Advantages

In this section, we summarize the features of the APEC model and explain the benefits of each feature. We also explain the advantages of the APEC model by comparing it to other models based on an analogous concept.

The main features of the APEC model are as follows.

- (1) Bit-level concurrent computation is expressed diagrammatically and naturally
- (2) Synchronization and operation are expressed briefly and synthetically through a notion of carriers
- (3) Bidirectional communication is expressed simply
- (4) Meaningful computations can generally be expressed through the connection of uniform elements where there are three terminals and at most four ordered rules

The advantages of the APEC model are related to the features. Since the description capability and representation

Fig. 12 APEC primitives for synchronization and the corresponding unidirectional element connections.

of the APEC model are similar to those of dataflow models [12]–[14]. Here, we discuss its advantages compared with these dataflow models.

The first feature is the commonality with the dataflow models, with both being represented diagrammatically. However, our model has an advantage over these models in that it is capable of representing precise snapshots of a network state as the positions and bit-values of carriers.

The APEC model gains significant advantages from the second and third features. Processing elements of dataflow models have unidirectional ports, and operation processes progress by producing and consuming data tokens. Thus, notification signals are expressed by dedicated data tokens and routes for the signals. In our model, such notification signals can be briefly expressed through returned carriers after data is sent. In addition, bidirectional communication can be expressed similarly since it is a special method of the synchronization. As a consequence, systems designed by APEC are less complex than systems designed by dataflow models when dealing with synchronization and bidirectional communication. In addition, bit-level programs can be developed more easily and intuitively by using APEC.

We here explain the features using an example for comparison. Intuitive images of unidirectional element connections corresponding to the APEC primitives are shown in the upper parts of Fig. 12. The unidirectional connection corresponding to a “WAIT” primitive that uses three terminals needs six input or output interfaces, since all the terminals are used for input and output. The connection corresponding to “RSEL” needs the same number of interfaces. In general, unidirectional connections corresponding to primitives for synchronization are more complex and increase the number of interfaces, since carriers are used for both data transfer and notification. Although the number of interfaces can be reduced by using bundling functions provided by languages, the total complexity does not decrease since base components must handle unbundled interfaces to implement the behavior.

The last feature of APEC is concerned with implementation mainly. Uniform elements are easier and simpler to implement than non-uniform elements, since differences in the behavior of each element only depend on defined rules. For instance, the form can define even duplication and joint elements that are defined individually on the models. In addition, a network composed of uniform elements implicitly has fine-grained and homogeneous concurrency. Therefore,

such a network makes it easier to reduce concurrency on an actual system to execute it than is the case in a network of dataflow models.

The computation and description capability of APEC is almost equal to that of static dataflow models [12], [14], which compose a fixed network of non-uniform elements with fixed functions, since the functions of APEC primitives can be implemented by defining unidirectional elements individually. From the another aspect, the computation capability of these models is equal to finite state machine model, since whole states of an static network are finite number.

6. Applications

In this section, as a sample application of the APEC model we present a network that is a simple system. It demonstrates how complicated systems can be neatly designed using the APEC model. We also explain the advantages of the model through this example. Moreover, we describe a simulator of the model, which provides functions to design APEC networks and visually simulate their behavior.

6.1 A Simple 4-bit Computer as a Sample Network

As a sample application, we designed a simple 4-bit computer using the APEC model; the design is shown in Fig. 13 and Fig. 14. The specification of the system is as follows: The system has an accumulator, a flag register, an instruction pointer, and a 16-word (4-bit in 1-word) memory. The address port and data port have a 4-bit width. Supported arithmetic operations are addition (ADD), subtraction (SUB), and comparison (CMP), which write to only the accumulator. Conditional branching (JPL, JMI, JZE, JMP), loading (LD), storing (ST), and halting the system (HALT) are supported.

6.1.1 Notation for Compositional Network

We here introduce some notation to represent the compositional networks and their connections. An arrow means a bundled connection, and the arrow direction indicates the flow of meaningful data (without a default value). A line connecting terminals or interfaces represents a single connection. A bold bar attached to arrows or lines bundles connections and represents a connection array, and a number enclosed by brackets “[]” close to the bar indicates the number of connections and the head side of the connection array. The arrangement in the array depends on the arrangement of attached positions.

Figure 13 shows the APEC network of the simple 4-bit computer. Descriptions such as “<\$0>” at the “Mem” and “Mem4” components are the initial values set to carriers. A letter string beginning with “\$” is a parameter that is replaced by a value, and the replacement value is propagated to parameters of the contained components. For instance, “Mem4<0011>” replaces “\$0” in each “Mem<\$0>” finally. The data in the memory array mean a program that

totalizes values between 5 and 1. Connections for control signals are represented by dark-gray and unwired arrows for ease of understanding the network. For instance, the “adda” interface is connected to the no-name interface of an “Rd4” component.

Figure 14 shows the network of the “Controller” component in the computer network, and describes the mnemonic codes for the instructions that the computer has. The “control signal” descriptions are sequences of control signals in the table. “<>” is one combination of control signals sent concurrently when the operation code is decoded. A letter string separated by “,” is an interface name used to send a signal “1”, and a name with an overbar means a negative signal (a carrier with “0”). The “mnemonic” descriptions are mnemonic codes, and an operand “[addr]” means a value in the address that is “addr”.

6.1.2 Behavior of Primitives and Components

The most important parts of the system are the “Rd4” and “Wr4” components. These components work as accessors for data storage in “Mem4”. “Rd4” sends 5-bit request carriers to read 4-bit data from “req”, and sends the data from “dt” when each carrier is returned. “Wr4” sends 5-bit request carriers including 4-bit data to write from “req”, and waits for the returned carriers. Both components return a carrier as a trigger (from the no-name interface) when the operation is completed.

“NOT”, “XOR”, “OR” and “EQ” (Equal) provide logic operations that correspond to each name. Intuitive representations of the behavior of some primitives are shown in Fig. 15. “FILT” (FILTer) sends a value that is equal to a carrier on “y”. “RD” and “WR” provide functions to access the “Mem” component. “SEQ” (SEQuencer) is a function for sequential operation, which sends a value using a carrier on “z” when a carrier arrives on “x”, sends a value “1” using a carrier on “y” when the carrier on “z” is returned, and returns the carrier on “x” when the carrier on “y” is returned. “RC1” (Responsible Constant “1” generator) sends a value of “1” and waits for the return of the carrier.

6.1.3 System Behavior

The system is controlled by carriers sent as control signals from the “Controller”, and one computation step is expressed by a process of sending and returning the carriers. The next step is not started until all sent carriers are returned. In other words, the mechanism emulates synchronization by clock cycle. Connections for sending and for notification are needed if such a mechanism is implemented in unidirectional models. In addition, individual elements for operating the notification signals are required. As above, this shows that the APEC model can express complex behavior more simply than with dataflow-like models. The interface to the memory array is also designed simply since the connections for requesting and sending are shared.

In the “Controller” component, the first operation starts

Fig. 13 APEC network of a simple 4-bit computer.

Fig. 14 "Controller" component of a simple 4-bit computer.

Fig. 15 Behavior of primitives used in the simple 4-bit computer.

from the primitive near "Fetch", which sends a carrier with "1" that is received by "Rd4". Carriers for requesting are sent from "req", and then return with data. The data is sent to be decoded, and the results are then sent as corresponding control signals to external components. The next step is started when one step is completed and all control signals are returned. Note that internal components which send carriers are locked until the carriers are returned. This mechanism makes it possible to do both operation and synchronization synthetically. Figure 16 shows sent carriers and locked internal components in the case of decoding "1010" ("ADD val"). All carriers acting as control signals corre-

Fig. 16 Sent carriers and locked components at "1010".

sponding to the operation code were sent, and routes used by the operations are locked until the carriers return.

6.1.4 Benefits of Designing with APEC

The benefits of designing a system using the APEC model correspond to the features and advantages described in Sect. 5. Here, we discuss them through a comparison with similar models or environments, since there are few other diagrammatical models for bit-level programming.

Regarding the first feature, bit-level operations on the system are expressed diagrammatically and naturally in APEC. The design of the system is similar to a design using logic gates since the connection of primitives expresses implicit concurrency. This implies that the model is suitable for designing bit-level computation, including hardware emulation programs. In general, the behavior of bit-level programs is difficult to understand and these programs are hard to develop since they are developed through textual languages; for instance, HDLs, and C language. CAD systems for hardware design can express digital circuits diagrammatically, but these systems are seldom applied to bit-level programming since the logic gate model is difficult to design intuitively. Some VPLs [5], [8], [9] aimed at the design of digital systems are implemented through dataflow models. Our model is more natural than the models of these languages regarding the precise representation of bit-level snapshots, through, since the APEC model is optimized to bit-level representation.

Regarding the second feature, the "Controller" component in the system uses a synchronization mechanism in

most parts that also include data transfer functions. Such a design is not easy to achieve with other models, since the carrier notion synthesizes both functions. Such a component that has complex behavior is difficult to design directly through that style of programming. In general, such a part is translated from a design using a popular model such as the state machine model. However, our model makes it easy to directly design and develop such a part by imagining the movement of carriers.

The third feature is related to the second, and the carrier notion synthesizes two streams of unidirectional communication. The mechanism is used by memory components and for route arbitration in the system, and the design of reading from or writing to a memory can be briefly represented by sharing sending and receiving ports. The mechanism also works as an alternative clock cycle method, since the arrival of carriers triggers the operation of a component. The design of a mechanism through general dataflow VPLs [3]–[5] is more complex than that with our model, since the process requires at least double unidirectional routes (for input and output, see Fig. 12).

With respect to the fourth feature, the system can be expressed by only connecting primitives that are uniform elements. This feature can express the behavior of the system precisely in the regular form that is the rule table. In addition, fine-grain and massive concurrency of the network is expressed implicitly through the independent behavior of uniform primitives. The implementation of an actual system using our models is simpler than when using similar models, since the system can be implemented using only the behavior of primitives and carriers. For instance, all individual elements of dataflow models must be implemented by individual procedures (or circuits).

Although, in this paper, it is not proven precisely that the combination of elements with these forms can express any nonuniform bidirectional component, the design of the system indicates the combination of primitives and the networks introduced here are capable of expressing practical functions. Especially, the system design indicates that our model has sufficient capability to describe at least non-interpreted (static) dataflow operations.

6.2 Simulator of the APEC Model

Figure 17 shows the appearance of the APEC simulator[†] we developed, which can compose and simulate any APEC network. A simulation proceeds using a transition-sequence generated by random numbers. A transition-sequence can proceed by interval timer or by the operator pushing a button. The networks introduced in this paper can be simulated using this system (the one in the figure is the “Detector”).

The system was developed using Borland C++ Builder 6, and the program size of the system is about 5000 lines. The most of the code is used to process graph representation and mouse operation. We actually composed and simulated all of the uncompositional networks presented in this paper using this system.

Fig. 17 Appearance of APEC simulator.

7. Related Works

The APEC model is a diagrammatic and concurrent model of computation oriented and optimized to bit-level computation. Other concurrent and diagrammatic models are Petri nets [10], [11] and dataflow-like models [12]–[17]. Petri nets can be used to express the behavior of a concurrent system by removing and adding a number of tokens from or to certain places. The model is suitable for expressing synchronization between concurrent processes, but some extensions are needed to express operations by data values. Thus, derivational models based on Petri nets have been proposed for concurrent computations (Colored Petri nets, etc). However, many extension and limitations are needed to express bit-level processing elements that behave independently of each other.

Dataflow-like models can express operation-level concurrent computation naturally. Many VPLs based on such models have been developed and studied [3]–[5]. In our previous research, we also developed a VPL called A-BITS [1], [2] based on a dataflow-like model. A programming language based on a similar model has also been developed [6], [7]. The language is more optimized for developing stream-oriented programs compared to other dataflow VPLs.

These dataflow-like models are suitable for expressing flows of data and implicit concurrency, and thus are frequently applied to programs developed by connecting function-level components using data streams. However, with these models it is difficult to frequently synchronize concurrent processes not needed for stream-oriented programs, since individual unidirectional connections for synchronization make the design of programs more complex. Therefore, some VPLs based on the models provide special synchronization methods. For instance, Prograph provides a mechanism called synchro-link [5] that emulates sequential execution in a control-flow program. However, such meth-

[†]The software can be downloaded at <http://www.bitprog.info/>

ods are not premised on the synchronization of many parts used in bit-level computation, and the behaviors are highly dependent on each specification of languages.

The APEC model is most similar to a model called the “uninterpreted dataflow graph” [14], which expresses a static (fixed) network of processes for unidirectional computation. However, the APEC model differs from this model from the point of bidirectional communication using the carrier notion and uniform processing elements. Computation by rule-based elements called primitives and carriers can express synchronization and communication synthetically. This feature is more suitable for developing bit-level programs since the “clock cycle” notion that is frequently used on hardware systems can be emulated through a process of sending and returning carriers. Actually, the controller of the simple computer is designed using this method. In addition, many kinds of bit-level program can be expressed simply and briefly since bidirectional communication is useful for reducing the number of interfaces.

A concurrent model called the Actor model is a non-diagrammatic model [18] which computes by communicating component-level processes using messages. This model is suitable for and applied to object-oriented programming and to design large scale distributed systems. The notion of carriers differs from the notion of messages, though, in that carriers exist persistently and become state variables of primitives. The concurrent models called Process Algebras are also non-diagrammatic models [19]–[22], and are more suitable for formal specification description and analyzing concurrent processes. These models differ from the APEC model by expressing explicit concurrency through operating function-level processes using algebraic operators. This feature is not suitable for bit-level computation since many bit-level programs such as digital circuit emulators have implicit concurrency expressed by logic gates.

Recently, research on fine-grain computer architecture [23]–[26] has advanced, although this is not directly related to our research. A dataflow model has been implemented onto the Plastic Cell Architecture (PCA) [27], and a dedicated architecture for a dataflow model also has also been designed [28]. Similarly, the APEC model may lead to research on such a domain, since primitives that are uniform elements of a fixed size are more suitable for optimizing fine-grain massive concurrency than dataflow-like models. A rule table of a primitive can be coded to 48-bit data since one of the rules can be coded as 12-bit data, and the carrier connections can be implemented and emulated by a packet transfer network.

8. Conclusion

We have proposed a concurrent model of computation, called APEC, for bit-level concurrent computation. We introduced the basic concept of the APEC model, the diagrammatic notation, and the formal definition that corresponds to the notation. We then presented the basic primitives and some important compositional networks, and presented an

example network and a simulator of APEC as applications of the APEC model. We also showed that the APEC model has the potential to sufficiently express asynchronous concurrent bit-level computation, and is simpler to use than unidirectional models to express synchronization and bidirectional communication.

Our future work will be to develop a VPL oriented to bit-level computation based on the model, and to refine the model for more powerful expression, for instance, by sharing operation resources such as subroutines in sequential programs.

References

- [1] T. Ajiro and K. Tsuchida, “A visual programming language A-BITS,” IEICE Technical Report, SS2003-13, Sept. 2003.
- [2] T. Ajiro and K. Tsuchida, “A bit-level concurrent visual programming language (A-BITS) and a base computation model (APC) for its development,” IEEE Symposium on Visual Languages & Human-Centric Computing, pp.269–271, Sept. 2005.
- [3] N.C. Shu, *Visual Programming*, International Thomson Publishing, ISBN 0-442-28014-9, 1988.
- [4] M. Burnett, “The visual language research bibliography,” <http://web.engr.oregonstate.edu/burnett/vpl.html>
- [5] D.D. Hils, “Visual languages and computing survey: Data flow visual programming languages,” *J. Visual Languages and Computing*, vol.3, no.1, pp.69–101, March 1992.
- [6] K. Kuse, M. Sassa, and I. Nakata, “Design and implementation of a stream programming language,” *IPSI MAGAZINE*, vol.31, no.5, pp.673–685, May 1990.
- [7] K. Kuse, M. Sassa, and I. Nakata, “On stream programming environment using diagram representations,” *IPSI Journal*, vol.27, no.12, pp.1249–1253, Dec. 1986.
- [8] T.J. Smedley, “A high-level visual language for the graphical description of digital circuits,” 11th International IEEE Symposium on Visual Languages, pp.77–82, 1995.
- [9] D.L. Miller-Karlow and E.J. Golin, “vVHDL: A visual hardware description language,” *Proc. 1992 IEEE Workshop on Visual Languages*, pp.133–139, 1992.
- [10] J.L. Peterson, “Petri nets,” *ACM Comput. Surv.*, vol.9, no.3, pp.223–252, Sept. 1977.
- [11] T. Murata, “Analysis and application of Petri nets,” *Kindai kagaku sya*, ISBN 4-7649-0204-4, 1992.
- [12] J.A. Sharp, *DATA FLOW COMPUTING*, Ellis Horwood Limited, ISBN 0-470-20167-3, 1985.
- [13] R. Jagannathan, “Dataflow models,” in *Parallel and Distributed Computing Handbook*, chapter 8, McGraw-Hill, ISBN 0-07-073020-2, 1996.
- [14] K.M. Kavi and B.P. Buckles, “A formal definition of data flow graph models,” *IEEE Trans. Comput.*, vol.C-35, no.11, pp.940–948, Nov. 1986.
- [15] G. Kahn, “The semantics of a simple language for parallel programming,” *Inf. Process.*, vol.74, pp.471–475, 1974.
- [16] E.A. Lee and T.M. Parks, “Dataflow process networks,” *Proc. IEEE*, vol.83, no.5, pp.773–801, May 1995.
- [17] M. Geilen and T. Basten, “Reactive process networks,” *Proc. 4th ACM International Conference on Embedded Software*, pp.137–146, 2004.
- [18] C. Hewitt, “Viewing control structures as patterns of passing messages,” *J. Artificial Intelligence*, vol.8, no.3, pp.323–364, 1977.
- [19] J. Baeten, “A brief history of process algebra,” *Theor. Comput. Sci.*, vol.335, no.2-3, pp.131–146, May 2005.
- [20] R. Milner, *Communicating and Mobile Systems: the Pi-calculus*, Cambridge University Press, ISBN 0-521-65869-1, 1999.
- [21] C.A.R. Hoare, “Communicating sequential processes,” *Commun.*

- ACM, vol.21, pp.666–677, 1978.
- [22] L. Cardelli and A.D. Gordon, “Mobile ambients,” *Theor. Comput. Sci.*, vol.240, no.1, pp.177–213, June 2000.
 - [23] K. Compton and S. Hauck, “Reconfigurable computing: A survey of systems and software,” *ACM Comput. Surv.*, vol.34, no.2, pp.171–210, June 2002.
 - [24] H. Ito, R. Konishi, H. Nakada, H. Tsuboi, Y. Okuyama, and A. Nagoya, “Dynamically reconfigurable logic LSI: PCA-2,” *IEICE Trans. Inf. & Syst.*, vol.E87-D, no.8, pp.2011–2020, Aug. 2004.
 - [25] T. Sugawara, K. Ide, and T. Sato, “Dynamically reconfigurable processor implemented with IPFlex’s DAPDNA technology,” *IEICE Trans. Inf. & Syst.*, vol.E87-D, no.8, pp.1997–2003, Aug. 2004.
 - [26] B. Tan, R. Yoshimura, T. Matsuoka, and K. Taniguchi, “Dynamically programmable parallel processor (DPPP): A novel reconfigurable architecture with simple program interface,” *IEICE Trans. Inf. & Syst.*, vol.E84-D, no.11, pp.1521–1527, Nov. 2001.
 - [27] N. Imlig, T. Shiozawa, R. Konishi, K. Oguri, K. Nagami, H. Ito, M. Inamori, and H. Nakada, “Programmable dataflow computing on PCA,” *IEICE Trans. Inf. & Syst.*, vol.E83-A, no.12, pp.2409–2416, Dec. 2000.
 - [28] J. Teifel and R. Manohar, “Asynchronous dataflow FPGA architecture,” *IEEE Trans. Comput.*, vol.53, no.11, pp.1376–1392, Nov. 2004.

Takashi Ajiro received his B.E. and M.E. degrees from Toyo University in 2003 and 2005 respectively. He is currently in the doctor course of the Department of Open Information Systems, Toyo University. His main research interests are concurrent models of computation, programming languages, and visual languages.

Kensei Tsuchida received his M.S. and D.S. degrees in mathematics from Waseda University in 1984 and 1994 respectively. He was a member of the Software Engineering Development Laboratory, NEC Corporation in 1984–1990. From 1990 to 1992 he was a Research Associate in the Department of Industrial Engineering and Management at Kanagawa University. In 1992 he joined Toyo University, where he was an Instructor until 1995 and an Associate Professor from 1995 to 2002, and since 2002 he

has been a Professor of the Department of Information and Computer Sciences. He was a visiting associate professor in the Department of Computer Science at Oregon State University from 1997 to 1998.