BEHAVIOR EXPRESSION AND OMDD

Yunming WANG
INRIA - IRISA,
Campus de Beaulieu,
35042 Rennes, France
ywang@irisa.fr

Abstract We propose a new synchronous language called Behavior Expression, its semantics and compilation mechanism. We also present OMDD as intermediate code for its compilation. Dependency cycle, determinism and composability can be checked directly by analyzing OMDDs. Consequently, it allows partial compilation and automatic distribution. Based on these benefits, we propose a new methodology for the development of real-time distributed systems by integrating behavior expression into UML.

Keywords: Behavior expression, OMDD, real-time system, distributed system

Introduction

The Unified Modeling Language (UML) [8] has rapidly become a hot topic of the software design community. It is composed of different kinds of diagrams which describe different views. These views represent our complementary and orthogonal cognitions of the desired system. By specifying one cognition in one diagram, UML eases system modeling, and induces less misunderstandings. Having a set of benefits, it becomes a standard framework for object-oriented methodologies.

However, when it comes to consistency check for UML, or formal verification, or code generation etc., it is somehow hard to grasp a uniform and mathematically well-founded semantics from these various different diagrams. And without a formal semantics, formal verification becomes a hard work.

Meanwhile, the concept of synchronous programming [2] has been proposed and widely accepted in the development of real-time systems, circuits, and embedded systems. Based on their mathematical foundation, synchronous languages have strict semantics and efficient approaches for their compilation and optimization [5, 1, 3]. Formal techniques for verification and validation have also been proposed.

Our aim in this paper is to take advantage of the rich background of synchronous model and UML by providing a new synchronous language called BE (Behavior Expression) and a new methodology for the development of real-time distributed systems. Thanks to the flexibility of BE, we can easily integrate it into UML. And with this integration, we have the benefits of easy system modeling (from UML), automatic code generation and system distribution (from BE) at the same time.

We will present the syntax and semantics of BE in section 1 and 2. Then we will provide a mechanism of compilation in section 3. In section 5, we will present OMDD as intermediate code for compilation. Partial compilation and automatic system distribution are concisely sketched in section 6 and 7. At last, we discuss the integration of BE into UML in section 8.

1. SYNTAX OF BEHAVIOR EXPRESSION

1.1. PRINCIPLES OF SYNCHRONOUS PROGRAMMING

In the concept of synchronous programming, we assume a real-time system reacts according to its environment step by step. Suppose the system has data elements $x_1, x_2, \cdots, x_n$, then its behavior may be described as Figure 1. At instant 0, the values represent the initialization of the system. At each instant $i > 0$, the system generates new values for these data elements according to the environment and previous state. A data element may have no value at some instants (e.g. x_3 at instant 2).

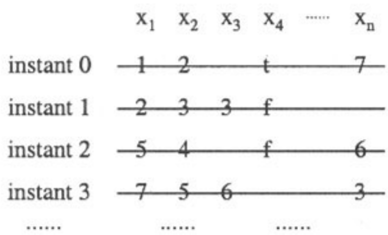


Figure 1 Principles of synchronous programming

For a certain instant $i > 0$, denote the current value of x with $x.set$, and its previous value with $x.get$. Then, the principles of synchronous languages are to describe how $x_i.set$ can be calculated from $x_i.get$ and inputs for every instant. In the next sections we will propose a new language called Behavior Expression (BE).

1.2. GRAMMAR

The behavior of an object is specified by a *Behavior Expression* E, whose grammar is:

$$
\begin{array}{lll}
e ::= & & \text{//event} \\
| & integer & \text{//integer constant} \\
| & true|false & \text{//boolean constant} \\
| & x.get & \\
| & x.set & \\
| & f_a(e,\cdots,e) & \text{// } f_a \text{ is an arithmetic function} \\
| & f_b(e,\cdots,e) & \text{// } f_b \text{ is a boolean function}
\end{array}
\qquad
\begin{array}{ll}
C ::= & x.get(e) \\
| & x.set(e) \\
| & \neg x.get(e) \\
| & \neg x.set(e) \\
| & C \wedge C
\end{array}
\qquad
\begin{array}{ll}
E ::= & x.set(e) \\
| & C \vdash E \\
| & E \parallel E \\
| & E \vee E
\end{array}
$$

Figure 2 Grammar of behavior expression

In this figure, e is an expression of a certain type (integer, boolean, event etc.). A data element x has two channels: $x.get$ and $x.set$, they can be used in e (like a variable). C is a condition[1] used to trigger a BE E.

1.3. COMMON SENSE

Each BE E describes a behavior, or, vaguely speaking, a "task" we have to do. We have four ways to specify it:

- Assign a set-channel with a value ($x.set(e)$).
- Divide the "task" into several parts, each corresponds to a "sub-task". Specify each sub-task with a BE, and then *compose* them ($E \parallel E'$).
- The "task" can be done in some different ways. Each is specified by a BE, and then make a choice between them ($E \vee E'$).
- The "task" will be performed only on some condition C, it is specified as $C \vdash E$.

1.4. EXTENSIONS

- Inputs

  Input events or values are necessary to control a system interactively. They are represented by *parameters*. A parameter p has only a channel $p.get$ whose value comes from the environment or user input.

- Guards

[1]It resembles boolean expression. $x.get(e)$ can be regarded as "if the value of $x.get$ equals e". A careful reader may ask why we do not use disjunction in its definition, and why we use $\neg x.set(e)$ and $\neg x.get(e)$ instead of $\neg C$. The reasons are: 1/ The restriction of C in this form avoids transforming a boolean expression in Disjunctive Normal Form (which is NP-complete) while constructing OMDDs. 2/ This is already enough for specification since pure boolean expression can be used as a guard (cf. section 1.4).

Suppose b is a boolean expression. $b \vdash E$ is introduced for convenience:
$b \vdash E = (\xi.set(b) \parallel \xi.set(true) \vdash E)$

- Initial state

  We can also provide the initial state for a BE as shown in the following syntax.

$$\begin{array}{lll} E ::= & \cdots & \text{/* defined in 1.2 */} \\ \mid & b \vdash E & \text{/* guard extension */} \\ \mid & E \text{ "initial" } I & \text{/* initialization */} \end{array}$$

$$\begin{array}{ll} I ::= & x.set(e) \\ \mid & I \text{ ',' } x.set(e) \end{array}$$

- Name

  We can give a BE a name, and then use the name for clearer representation.

$$\begin{array}{lll} E ::= & \cdots & \text{/* defined above */} \\ \mid & name \text{ ":=" } E & \text{/* give } E \text{ a name */} \\ \mid & name & \text{/* recall a named BE */} \end{array}$$

- Priority

  We uses $E_1 \overset{p}{\vee} E_2$ to describe a "choice with priority". When the choice is not exclusive, we choose E_1 as it has higher priority than E_2 (See 3.3 for more detail).

1.5. EXAMPLES

Example 1 *This is a simple example mimicking the function of a clock. It is divided into three parts: second, minute and hour. For every instant, the behavior of second is to increase by one.*

$$E_s := S.set((S.get + 1)\ mod\ 60)\ initial\ S.set(0)$$

The behavior of minute is to increase by one whenever a new minute passes (S is set to 0), or to keep the same value otherwise.

$$E_m := \left(\begin{array}{ll} & S.set(0) \vdash M.set((M.get + 1)\ mod\ 60) \\ \overset{p}{\vee} & M.set(M.get) \end{array} \right) initial\ M.set(0)$$

Similarly, the behavior of hour is to increase by one whenever a new hour passes (S and M are set to 0 at the same time), or to keep the same value otherwise.

$$E_h := \left(\begin{array}{ll} & S.set(0) \wedge M.set(0) \vdash H.set((H.get + 1)\ mod\ 24) \\ \overset{p}{\vee} & H.set(H.get) \end{array} \right) initial\ H.set(0)$$

As a result, the behavior of the total system is the composition of these three: $E_{clock} = E_s \parallel E_m \parallel E_h$.

Example 2 *Suppose we have a virtual system with a boolean input C. When C is true, we let X increase by 1 and Y be X * 2. Otherwise, we let X decrease by 1 and Y be X/2. Initially, X and Y are 0. The behavior expression of this system is:*

$$\left(\begin{array}{ll} & C.get(true) \vdash \left(\begin{array}{ll} & X.set(X.get+1) \\ \| & Y.set(X.set*2) \end{array}\right) \\ \vee & C.get(false) \vdash \left(\begin{array}{ll} & X.set(X.get-1) \\ \| & Y.set(X.set/2) \end{array}\right) \end{array}\right) \textit{initial } X.set(0), Y.set(0)$$

2. SEMANTICS OF BE

- A *pre-assignment* P of a BE is to associate each get-channel with a value in its corresponding domain.
- An *assignment* A of a BE is to associate each set-channel with a value in its extended domain[2].

2.1. MAP

Given a pre-assignment P and an assignment A, we define a map $f : E \cup C \rightarrow \{T, F, -\}$ corresponding to grammar items illustrated in Figure 2.

$$\begin{aligned} f(x.get(e)) &= (x.get == e) \\ f(x.set(e)) &= (x.set == e) \\ f(\neg C) &= \neg f(C) \\ f(C_1 \wedge C_2) &= f(C_1) \wedge f(C_2) \\ f(C \vdash E) &= f(C) \vdash f(E) \\ f(E_1 \parallel E_2) &= f(E_1) \parallel f(E_2) \\ f(E_1 \vee E_2) &= f(E_1) \vee f(E_2) \end{aligned}$$

In addition, $\neg T = F$; $\neg F = T$; $\wedge, \vdash, \parallel$ and $\vee$ are defined in Figure 3.

$\wedge$	T	F
T	T	F
F	F	F

$\vdash$	T	F	$-$
T	T	F	$-$
F	$-$	$-$	$-$

$\parallel$	T	F	$-$
T	T	F	T
F	F	F	F
$-$	T	F	$-$

$\vee$	T	F	$-$
T	T	T	T
F	T	F	F
$-$	T	F	$-$

Figure 3 Map definition for $\wedge$, $\vdash$, $\parallel$, $\vee$

[2]As stated in section 1.1, a data element may have no value at some instants, this is called absence. We extend the domain with an "absent" value "$\perp$" denoting its absence.

Example 3 *As a continuation of Example 2, $P = \{C.get = true, X.get = 3, Y.get = 6\}$ is a pre-assignment, and $A = \{X.set = 4, Y.set = 8\}$ is an assignment. Let's demonstrate the definition of map step by step:*

$$f(X.set(X.get + 1)) = (X.set == X.get + 1) = T \tag{1}$$

$$f(Y.set(X.set * 2)) = (Y.set == X.set * 2) = T \tag{2}$$

$$f(C.get(true)) = (C.get == true) = T \tag{3}$$

$$f\begin{pmatrix} & X.set(X.get + 1) \\ \| & Y.set(X.set * 2) \end{pmatrix} = T \,\|\, T = T \tag{4}$$

$$f\left(C.get(true) \vdash \begin{pmatrix} & X.set(X.get + 1) \\ \| & Y.set(X.set * 2) \end{pmatrix}\right) = T \vdash T = T \tag{5}$$

$$f(C.get(false)) = (C.get == false) = F \tag{6}$$

$$f\left(C.get(false) \vdash \begin{pmatrix} & X.set(X.get - 1) \\ \| & Y.set(X.set/2) \end{pmatrix}\right) = F \vdash\ ? = - \tag{7}$$

$$f\begin{pmatrix} & C.get(true) \vdash \begin{pmatrix} & X.set(X.get + 1) \\ \| & Y.set(X.set * 2) \end{pmatrix} \\ \vee & C.get(false) \vdash \begin{pmatrix} & X.set(X.get - 1) \\ \| & Y.set(X.set/2) \end{pmatrix} \end{pmatrix} = T \,\|\, - = T \tag{8}$$

In (7), we used a trick: for any value "?", $F \vdash\ ? = -$.

2.2. SEMANTICS

Given a BE E, a pre-assignment P and an assignment A, A is called a *solution* of E with respect to (w.r.t) P iff:

- $f(E) = T$
- for all $x.set$, if $\forall C \vdash x.set(e)$ appear in E, $f(C) = F$, then $x.set$ is assigned with $\perp$ in A.

In Example 3, A is a solution w.r.t P, while $A' = \{X.set = 4, Y.set = 7\}$ is not.

Given an object whose behavior is described by E with initialization A_0, its behavior is the trace:

$$A_0, A_1, A_2, \cdots$$

where A_i is a solution of E w.r.t A_{i-1}, $i = 1, 2, 3, \cdots$. Note that when shifting from instant $(i - 1)$ to i, a data element moves the value in $x.set$ to $x.get$. Which means that A_{i-1} (together with the input values) is actually the pre-assignment for A_i.

2.3. DETERMINISM

Given an expression E and an initial state, it is deterministic iff for all $i = 1, 2, 3, \cdots$, there is only one solution A_i of E w.r.t A_{i-1}.

2.4. COMMENTS

According to the syntax and semantics, we can see that BE is simply an expression that must be satisfied at every instant. It is flexible in the sense of "choice" and "activation". Suppose we have already described two behaviors in BE E_1 and E_2. Now we have a new system which behaves as E_1 on condition C_1 and as E_2 on conditions C_2. Then we can easily write the BE $(C_1 \vdash E_1) \vee (C_2 \vdash E_2)$ for the new system. Some earlier synchronous languages [2] do not share these flexibilities however.

3. COMPILATION OF BE

3.1. SIMPLIFYING BE

Theorem 1 *According to the map defined in section 2.1, we can easily prove:*

$$\begin{array}{rcl} C \vdash (E_1 || E_2) & = & (C \vdash E_1) || (C \vdash E_2) \\ C \vdash (E_1 \vee E_2) & = & (C \vdash E_1) \vee (C \vdash E_2) \\ C \vdash (C' \vdash E_2) & = & (C \wedge C') \vdash E_2 \end{array}$$

We simplify a BE by repeatedly substituting the left part of these equations by the right part, until for any $C \vdash E$, E is a set-channel. Such $C \vdash E$ is called a *primitive* BE. A *simplified* BE is then composed of a set of primitive BEs with operations $\vee$ and $||$.

Example 4 *Let us consider the* BE *in example 2. After simplification it becomes:*

$$\left(\begin{array}{cc} & \left(\begin{array}{ll} & C.get(true) \vdash X.set(X.get + 1) \\ || & C.get(true) \vdash Y.set(X.set * 2) \end{array} \right) \\ \vee & \left(\begin{array}{ll} & C.get(false) \vdash X.set(X.get - 1) \\ || & C.get(false) \vdash Y.set(X.set / 2) \end{array} \right) \end{array} \right) \textit{initial } X.set(0), Y.set(0)$$

3.2. WELL-FORMED BE

A simplified BE E is *well-formed* iff: for all $E_1 \mid\mid E_2$ appears in E, there doesn't exist $C_1 \vdash x_1.set(e_1)$ in E_1 and $C_2 \vdash x_2.set(e_2)$ in E_2 such that $x_1.set$ and $x_2.set$ are actually the same set-channel[3].

In this paper, we consider only well-formed BE. A BE not well-formed is something like a C program with syntax error.

Theorem 2 *A well-formed* BE E *is deterministic iff for all* $E_1 \vee E_2$ *appears in* E*,* $C_1 \vdash x_1.set(e_1)$ *in* E_1 *and* $C_2 \vdash x_2.set(e_2)$ *in* E_2*,* $f(C_1 \wedge C_2) \equiv F$*. (proof omitted)*

[3]This is similar with SIGNAL, we don't accept $(|X := 1 \textit{ when } C_1 | X := 2 \textit{ when } C_2|)$ in SIGNAL. This is to say, if a system S is a composition of two sub-systems $S1$ and $S2$, then $S1$ and $S2$ are supposed to do different things.

3.3. NON-DETERMINISM

If a well-formed BE is not deterministic, we do not know which one of E_1 or E_2 should be chosen when C_1 and C_2 are both true. We will give two remedies for this problem.

- Using priority

  $E_1 \overset{p}{\bigvee} E_2$ is introduced as a "choice with priority". Whenever we have $C_1 \vdash x_1.set(e_1)$ in E_1, $C_2 \vdash x_2.set(e_2)$ in E_2, C_1 and C_2 are not exclusive, suppose $C_1 = \bigwedge_i C_{1i}$ (cf. its grammar in Figure 2), then $C_2 \vdash x_2.set(e_2)$ is substituted by: (where $\neg(\neg C) = C$)

$$\begin{array}{ll} & C_2 \wedge \neg C_{11} \vdash x_2.set(e_2) \\ \vee & C_2 \wedge C_{11} \wedge \neg C_{12} \vdash x_2.set(e_2) \\ \vee & \cdots \\ \vee & C_2 \wedge C_{11} \wedge \cdots \wedge C_{1,n-1} \wedge \neg C_{1n} \vdash x_2.set(e_2) \end{array}$$

- User indication

  Introduce a new parameter ξ to control the choice between E_1 and E_2: $(\xi.get(true) \vdash E_1) \vee (\xi.get(false) \vdash E_2)$.

Example 5 *After this process, the* BE *in Example 1 is changed to:*

$$\begin{array}{ll} & S.set((S.get+1)\,mod\,60)\ initial\ S.set(0) \\ \| & \left(\begin{array}{ll} & S.set(0) \vdash M.set((M.get+1)\,mod\,60) \\ \vee & \neg S.set(0) \vdash M.set(M.get) \end{array}\right) initial\ M.set(0) \\ \| & \left(\begin{array}{ll} & S.set(0) \wedge M.set(0) \vdash H.set((H.get+1)\,mod\,24) \\ \vee & \neg S.set(0) \vdash H.set(H.get) \\ \vee & S.set(0) \wedge \neg M.set(0) \vdash H.set(H.get) \end{array}\right) initial\ H.set(0) \end{array}$$

3.4. ORGANIZING BE

Both $C \vdash x.set(e)$ and $\bigvee_i C_i \vdash x.set(e_i)$ are called a *single choices on* x, denoted by $\bigvee_x$. We will re-organize a BE to "compositions of single choices", that is, of the form $\|_i \bigvee_{xi}$.

For any minimal occurrence of:

$$(\|_i \vee_{xi}) \vee (\|_j \vee_{yj}) \tag{9}$$

we can re-write it as:

$$\begin{array}{ll} \| (\vee_{xi}) \vee (\vee_{yj}) & \text{where } xi = yj \\ \| \vee_{xi} & \nexists yj : xi = yj \\ \| \vee_{yj} & \nexists xi : xi = yj \end{array} \tag{10}$$

Theorem 3 *On condition of well-formed deterministic* BE, *(9)=(10).*

By repeatedly applying this rule, we can gain our aim.

Example 6 *The* BE *in example 4 is a well-formed deterministic* BE. *After re-organization, it becomes:*

$$\left(\begin{array}{c} \left(\begin{array}{ll} & C.get(true) \vdash X.set(X.get + 1) \\ \vee & C.get(false) \vdash X.set(X.get - 1) \end{array} \right) \\ \| \\ \left(\begin{array}{ll} & C.get(true) \vdash Y.set(X.set * 2) \\ \vee & C.get(false) \vdash Y.set(X.set/2) \end{array} \right) \end{array} \right) \text{initial } X.set(0), Y.set(0)$$

3.5. CODE GENERATION

- Generating SIGNAL code

  SIGNAL was proposed in [2] and its compilation has already been implemented. An advantage of generating SIGNAL code is to re-use the existing clock calculus and causality analysis procedures of SIGNAL compiler.

  After the re-organization, a BE looks like a SIGNAL process. A "single choice" is an equation in SIGNAL (probably) with "default", and composition of "single choices" is just like composition of equations in SIGNAL. However, if we re-write a BE in SIGNAL directly like this, we will usually get clock constraints. The main reasons are:

  - X and $X\$$ (correspond to $X.set$ and $X.get$ in this paper) have the same clock in SIGNAL.
  - for an equation $X := f(X_1, \cdots, X_n)$, X, X_1, $\cdots$, X_n have the same clock.

  However, in BE, we do not have these constraints. A solution is to create a set of new variables in SIGNAL carrying the wanted values with the "most frequent" clock. As these variables have the same clock, they can be used in every expression as we like. More details are omitted in this paper, please refer to appendix for examples.

  A prototype of translating a BE into SIGNAL is already implemented. The generated signal processes of Example 1 and 2 are provided in the appendix.

- Using OMDD as intermediate code

  This will be presented in section 5.

4. DEPENDENCY AND VIRTUAL ORDER

The compilation of synchronous languages usually requires dependency (causality) analysis [1, 5]. Let's consider the BE in example 2, we must know the value of $X.set$ and $C.get$ before the calculation of $Y.set$, so we have dependency $X.set \rightarrow Y.set$ and $C.get \rightarrow Y.set$. Surely, dependencies cannot have cycle. So there exists a total order which covers all the dependencies. We call it *virtual* order. Figure 4 gives an algorithm to calculate a virtual order from a re-organized BE $E = \ \|_i \vee_{xi}$.

```
Let S_i be the set of channels other than x_i.set appearing in ∨_xi
Let B be the set of all get-channels of E
Let N = ∅ //channels already ordered
Let V = ⟨⟩ //empty virtual order
while (E is not empty) {
    if ∃i such that S_i ⊆ B then {
        append a random order of (S_i \ N) to V
        append x_i.set to V
        N = N ∪ S_i ∪ {x_i.set}
        B = B ∪ {x_i.set}
        remove ∨_xi from E
        }
    else report error // due to dependency cycle
}
return V
```

Figure 4 Algorithm of virtual order

Example 7 *The virtual orders of the* BE *in Example 5 is:*

$$< S.get, S.set, M.get, M.set, H.get, H.set >$$

and that of Example 6 is:

$$< C.get, X.get, X.set, Y.set >$$

5. OMDD

Binary Decision Diagram (BDD) [6, 9] has been proposed as a date structure to represent boolean functions. Reduced ordered BDD (ROBDD) [7] introduces further restrictions on the order of decision variables in the graph.

In this section, we will present a date structure called Ordered Multiple Decision Diagram (OMDD) on a similar principle as an intermediate code for the compilation of behavior expressions.

5.1. DEFINITION

Given the channels totally ordered, we define OMDD as:

1 A node $(x.set, x.set(e))$ is an OMDD

2 Given a set of OMDDs $M_1, M_2, \cdots, M_n$, the structure of figure 5 is also an OMDD, where:

(a) C is a channel

(b) if C is a set-channel, A is an (possibly empty) action $C(e)$. (A must be empty if C is not a set-channel.)

(c) L_i are mutually exclusive restrictions of the value of channel C

(d) for any channel C' in $M_1, M_2, \cdots, M_n$, $C < C'$

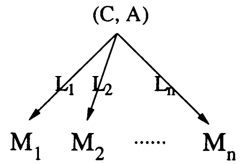


Figure 5 Base structure of OMDD

We are somehow indifferent to the strict form of the "restriction" of values. We can use "c", "$\neq c$" (c is a constant like 0, true, etc.), "*any*" or even subranges such as $[3, +\infty)$. What's important is to be able to perform the union, intersection, subtraction of "restrictions".

The structure in figure 5 means, we will do action A (if there is one) when arriving at node (C, A). Then we check the value of C, if it satisfies L_k, then we leave for M_k. So, code generation from OMDD is very simple.

5.2. CODE GENERATION FROM OMDD

1) For a leaf node $N = (x.set, x.set(e))$, it is translated to a procedure:

$$\text{P\_}N()\{x.set = e;\}$$

2) For an OMDD M as Figure 5, it is translated to a procedure:

```
P_M(){
    A;   // if A is empty, we do nothing here.
    if (C is in L1) then P_M1();
    else if (C is in L2) then P_M2();
    ...
    else if (C is in Ln) then P_Mn();
}
```

5.3. CONSTRUCTING OMDD

In this subsection, we will discuss the construction of OMDD for a BE. After the simplification, determinism check, and re-organization, a BE can be described as:

$$E = \, \|_i E_i \tag{11}$$
$$E_i = \vee_j E_{ij} \tag{12}$$
$$E_{ij} = \wedge_k C_{ijk}(e_{ijk}) \vdash x_i.set(e'_{ij}) \tag{13}$$

- E_{ij}

  Without loss of generality, we suppose $C_{ij1} < C_{ij2} < \cdots < C_{ijn}$. Then the OMDD of E_{ij} is[4]:

$$M_{ij} = \begin{array}{c} (C_{ij1}, \emptyset) \xrightarrow{e_{ij1}} (C_{ij2}, \emptyset) \xrightarrow{e_{ij2}} \cdots \xrightarrow{e_{ijn-1}} \\ (C_{ijn}, \emptyset) \xrightarrow{e_{ijn}} (x_i.set, x_i.set(e'_{ij})) \end{array}$$

- $E_i = \vee_j E_{ij}$

  The OMDD M_i of E_i can be obtained from M_{ij} by *choice* function given in Figure 6.

- $E = \, \|_i E_i$

  The OMDD M of E is obtained from M_i by *comp* function illustrated in Figure 7.

5.4. EXAMPLES

Example 8 *Let us consider the expression in Example 6, and use the virtual order in example 7, the four base* OMDDs *are:*

$$(C.get, \emptyset) \xrightarrow{t} (X.set, X.set(X.get + 1))$$
$$(C.get, \emptyset) \xrightarrow{f} (X.set, X.set(X.get - 1))$$
$$(C.get, \emptyset) \xrightarrow{t} (Y.set, Y.set(X.set * 2))$$
$$(C.get, \emptyset) \xrightarrow{f} (Y.set, Y.set(X.set/2))$$

When they are bound together, the result OMDD *and its corresponding code are presented in Figure 8.*

[4] A careful reader may ask how about if e_{ijk} is not a constant. This is not a fatal problem, however. We can simply substitute $C_{ijk}(e_{ijk})$ by $\xi.set(true)$, and compose E_{ij} with $\xi.set(C_{ijk} == e_{ijk})$.

```
OMDD choice(OMDD m1, OMDD m2)
{
  if (m1.c==m2.c)
  {
    if (m1.a is not empty && m2.a is not empty) then
         raise error; // nondeterministic
    else
    {
      if (m1 has no subtree) raise error; // nondeterministic
      if (m2 has no subtree) raise error; // nondeterministic
      suppose the subtrees of m1 are m11, m12, ... m1n with label 111, ... 11n;
      suppose the subtrees of m2 are m21, m22, ... m2m with label 121, ... 12m;
      NODE tmp;
      tmp.c=m1.c;
      tmp.a=union of m1.a and m2.a // the non-empty action if there is one
      11=union of 111,112,...,11n;
      12=union of 121,122,...,12m;
      for(i=1;i<=n;i++)add m1i as a subtree of tmp with label 11i\12;
      for(j=1;j<=m;j++)add m2j as a subtree of tmp with label 12j\11;
      for(i=1;i<=n;i++)for(j=1;j<=m;j++)
          add choice(m1i,m2j) as a subtree of tmp with label 12j^11i;
      return tmp;
    }
  }
  else if (m1.c<m2.c)
  {
    NODE tmp;
    tmp.c=m1.c;tmp.a=empty;
    add subtree m2 to tmp with lable "any";
    return choice(m1,tmp);
  }
  else
  {
    NODE tmp;
    tmp.c=m2.c;tmp.a=empty;
    add subtree m1 to tmp with lable "any";
    return choice(tmp,m2);
  }
}
```

Figure 6 Choice operation for two OMDDs

```
OMDD comp(OMDD m1, OMDD m2)
{
  if (m1.c==m2.c)
  {
    if (m1.a is not empty && m2.a is not empty) then
        raise error; //OMDDs can not be composed
    else
    { if (m1 has no subtree) {
         m2.a=m1.a; //m1.a is not empty because it is a leaf node
         return m2;
      }
      if (m2 has no subtree) {
         m1.a=m2.a; //m2.a is not empty because it is a leaf node
         return m1;
      }
      suppose the subtrees of m1 are m11, m12, ... m1n with label l11, ... l1n;
      suppose the subtrees of m2 are m21, m22, ... m2m with label l21, ... l2m;
      NODE tmp;
      tmp.c=m1.c;
      tmp.a=union of m1.a and m2.a // the non-empty action if there is one
      l1=union of l11,l12,...,l1n;
      l2=union of l21,l22,...,l2m;
      for(i=1;i<=n;i++)add m1i as a subtree of tmp with label l1i\l2;
      for(j=1;j<=m;j++)add m2j as a subtree of tmp with label l2j\l1;
      for(i=1;i<=n;i++)for(j=1;j<=m;j++)
          add comp(m1i,m2j) as a subtree of tmp with label l2j^l1i;
      return tmp;
    }
  }
  else if (m1.c<m2.c)
  {
        NODE tmp;
        tmp.c=m1.c;tmp.a=empty;
        add subtree m2 to tmp with lable "any";
        return comp(m1,tmp);
  }
  else // m2.c < m1.c
  {
        NODE tmp;
        tmp.c=m2.c;tmp.a=empty;
        add subtree m1 to tmp with lable "any";
        return comp(tmp,m2);
  }
}
```

Figure 7 Composition operation for two OMDDs

```
if C.get==true then
  {X.set=X.get+1;Y.set=X.set*2;}
else if C.get==false then
  {X.set=X.get-1;Y.set=X.set/2;}
```

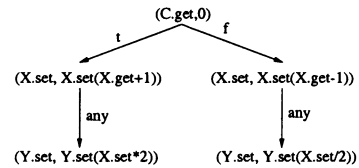
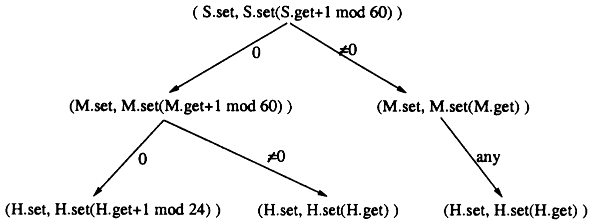


Figure 8 Bound OMDD and generated code of Example 8

Example 9 *Let us consider the* BE *in Example 5, and virtual order in example 7, the base* OMDDs, *the bound* OMDD, *and the generated code are:*

$$(S.set, S.set(S.get + 1 \; mod \; 60))$$
$$(S.set, \emptyset) \xrightarrow{0} (M.set, M.set(M.get + 1 \; mod \; 60))$$
$$(S.set, \emptyset) \xrightarrow{\neq 0} (M.set, M.set(M.get))$$
$$(S.set, \emptyset) \xrightarrow{0} (M.set, \emptyset) \xrightarrow{0} (H.set, H.set(H.get + 1 \; mod \; 24))$$
$$(S.set, \emptyset) \xrightarrow{\neq 0} (H.set, H.set(H.get))$$
$$(S.set, \emptyset) \xrightarrow{0} (M.set, \emptyset) \xrightarrow{\neq 0} (H.set, H.set(H.get))$$




```
S.set=S.get+1 mod 60;
if S.set==0 then
  { M.set=M.get+1 mod 60;
    if M.set==0 then H.set=H.get+1 mod 24;
    else if M.set!=0 then H.set=H.get;
  }
else if S.set!=0 then
  { M.set=M.get;
    H.set=H.get;
  }
```

6. PARTIAL COMPILATION

In section 3, we presented the compilation of BE by way of SIGNAL. One benefit of this compilation is the re-use of SIGNAL compiler. In section 5, we presented OMDD as intermediate code. Besides the efficiency of generated code, this has two other advantages: partial compilation and automatic system distribution. We discuss partial compilation in this section.

Suppose we have a system E consisting of sub-systems E_i with $||$ and $\vee$ operations. In section 3 and 5, we compile E directly. So, if there is a sub-system reused several times, it will be integrated and compiled several times. This is not satisfying when re-use is highly demanded. Can we compile sub-systems E_i so that the compilation of E is simply an integration of pre-compiled codes? Unfortunately, it is shown in [4] that, brute-force pre-compilation and their simple combination have some problems.

Actually, in order for partial compilation, we must be able to do following things *from the pre-compiled codes* for system integration:

1 Check if the integration of sub-systems will introduce dependency cycles.

2 Check the composability of subsystems. For example, if two sub-systems assign different values to the same channel on the same condition, they can not be composed.

3 Check the determinism of the integrated system.

These are easy when OMDD is used as intermediate codes. For 1, we need only to check if their virtual orders are conflict. And the algorithms given in Figure 6 and 7 contain already check 2 and 3.

Actually, in the total compilation presented in section 3 and 5, we have already checked these properties from BE: virtual order is generated, composability and determinism are already assured. So, when constructing OMDD, they do not need the checks in "choice" and "comp" operation any more.

This result is satisfying: Partial compilation ensures re-usability. Sub-systems and classes can be designed and compiled into OMDDs and stored in a library. We are able to reuse the compiled OMDDs as well as existing classes. This is helpful for large-scale systems.

7. SYSTEM DISTRIBUTION

Although the concept of synchronous model has been widely accepted, it is also argued that, very frequently, real-life architectures do not obey the ideal model of perfect synchrony. Consequently, when a synchronous system is distributed to several sites with an asynchronous communication, its behavior will probably change.

Fortunately, some technical results on this issue have been presented in [4, 3]. As an inference of these results, we can safely distribute a synchronous system without changing its behavior if the OMDDs of sub-systems are all sub-trees of the OMDD of the total system.

Suppose, for instance, the total system E is composed of sub-systems E_i. If for all i, the OMDD of E_i is a subtree of that of E, then we can safely distribute E_i into different sites. However, suppose the OMDD of E_1 is not a sub-tree of that of E, we can find a small E_1' (by analyzing OMDDs) such that the OMDD of $(E_1' \parallel E_1)$ is a sub-tree and E has the same behavior when E_1 is substituted by $(E_1' \parallel E_1)$. Then we can use $(E_1' \parallel E_1)$ instead of E_1 in the distribution. This is not a magic, essentially, adding E_1' to E_1 actually means adding a protocol of communication.

8. METHODOLOGY OF DEVELOPMENT

In this section, we aim to integrate BE into UML in order to take both the advantages of synchronous concept and that of object oriented concept, and to present a new methodology of the development for real-time distributed systems.

In this methodology, a class of reactive objects will be represented by a graph as :

Class name
Interface
Behavior expression

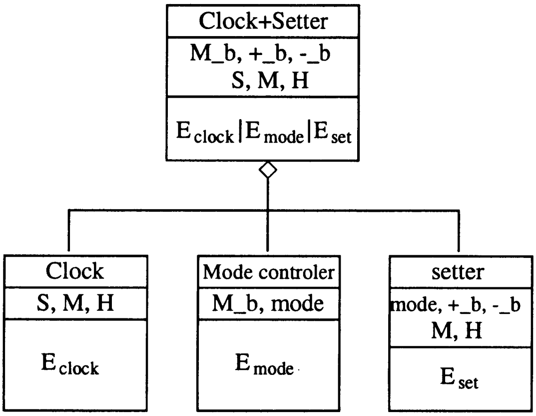
So, we will use UML class diagrams and deployment diagrams to describe the architecture of the desired system; and use state machines, MSC, and BE to define its reactive behavior.

In the beginning, we may have only vague ideas and the system is highly abstracted. Thanks to the flexibility of BE, it allows us to specify premature systems when we have only vague ideas in early stages. As development proceeds through the life cycle, these high-level abstract elements are expanded into low-level concrete elements; and the maturity level of element increases as it is corrected, polished, and optimized.

The choice of using state-machine, MSC or BE depends on the characteristic of the described object. Although BE fits the specification for lots of objects, state-machine may fit better for some other objects. But, by the end of the design process, other diagrams are translated to behavior expressions automatically. For example, we have already developed a prototype to *translate state machines into* BE. At the last stage, we will use the techniques stated in previous sections to simplify the BE, check and enforce determinism, re-organize it, and at last, OMDDs are constructed, codes are generated and distributed correctly over different sites.

Let us consider a simple example: a clock with time-set operation. It has three buttons. "Mode" button (denoted with M_b) is used to change the mode of the clock to vision-mode, set-minute-mode, set-hour-mode and again vision-mode. An "Add" button and a "Sub" button (denoted with +_b and -_b respectively) are used to increase or decrease the value when the clock is in set-minute-mode or set-hour-mode. Pressing "Add" button and "Sub" button when the clock is in vision-mode will do nothing.

After analysis of this requirement, we can decompose the system into three parts. One is to run the time normally in vision-mode as stated in the example 1, one is to manipulate the mode, and the other is to increase or decrease the value of minute or hour when necessary. So this system can be drawn as a class diagram in the following figure.



9. CONCLUSION

In this paper, we proposed a new synchronous language called BE to describe behavior of reactive objects. Essentially, it is simply an expression that must be satisfied at every instant. It is a declarative language and it is flexible in the sense of activation and choice. As a result, it is suitable even in early stages of development when we have only vague cognition of the desired system.

Based on its mathematical semantics and mathematical form, we presented a set of mechanisms to simplify BE; to remove non-determinism by priority or user indication; to re-organize well-formed deterministic BE; and to compile BE by way of SIGNAL. We provided also an approach to build a total order of channels covering all the dependencies.

We also proposed OMDD as an inter-mediate structure for the compilation of BE. Constructing OMDD from BE and generating code from OMDD are presented. The "choice" and "comp" operations on OMDDs allow partial compilation and facilitate automatic system distribution.

By integrating BE into UML, we get an ideal approach for the development of real-time distributed systems. We have at the same time the advantages of easy system modeling and formal techniques. We use UML class diagram and deployment diagram to describe the architecture; use state machine, MSC, and BE to define the behavior. State-machines and MSCs can be translated into BE to take advantages of partial compilation and system distribution mechanism. Till now, a prototype of translating from state-machines into BE, and another from BE into SIGNAL have been implemented with satisfying results.

Appendix: SIGNAL processes for Example 1 and 2

```
process result =
( ?
  !     integer S;
        integer M;
        integer H
)
   (| sys := 1
    | S_val := S default (S_val$ init 0) when ^sys
    | S_pre :=  (S_val$ init 0) when ^sys
    | S_pre ^= sys ^= S_val
    | M_val := M default (M_val$ init 0) when ^sys
    | M_pre :=  (M_val$ init 0) when ^sys
    | M_pre ^= sys ^= M_val
    | H_val := H default (H_val$ init 0) when ^sys
    | H_pre :=  (H_val$ init 0) when ^sys
    | H_pre ^= sys ^= H_val
    | S := ((S_pre+1) modulo 60) when ^sys
    | M := ((M_pre+1) modulo 60) when S_val=0 when ^S
          default M_pre when ^sys
    | H := ((H_pre+1) modulo 24) when S_val=0 when M_val=0 when ^M when ^S
          default H_pre when ^sys
    |)
```

Figure A.1 The SIGNAL process of Example 1

```
process result =
( ?     boolean C
  !     integer X;
        integer Y
)
   (| sys := 1
    | sys ^= C
    | X_val := X default (X_val$ init 0) when ^sys
    | X_pre :=  (X_val$ init 0) when ^sys
    | X_pre ^= sys ^= X_val
    | Y_val := Y default (Y_val$ init 0) when ^sys
    | sys ^= Y_val
    | C_val := C default C_val$ when ^sys
    | sys ^= C_val
    | X := (X_pre+1) when C_val=true when ^C
          default (X_pre-1) when C_val=false when ^C
    | Y := (X_val*2) when C_val=true when ^X when ^C
          default (X_val/2) when C_val=false when ^X when ^C
    |)
```

Figure A.2 The SIGNAL process of Example 2

Acknowledgments

The author wishes to thank Guy Leduc sincerely for his valuable comments.

References

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. Arborescent canonical form of boolean expressions. Technical Report 2290, Inria, June 1994.

[2] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *In* Science of Computer Programming, 16, 1991.

[3] A. Benveniste, B. Caillaud and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, CONCUR'99, Concurrency Theory, 10th International Conference, vol. 1664 of Lecture Notes in Computer Science, 162-177. Springer V., 1999.

[4] Albert Benveniste, Paul Le Guernic, and Benoît Caillaud. Compositionality in dataflow synchronous languages: specification & code generation. *Information and computation*, 1999.

[5] Loïc Besnard. *Compilation de* SIGNAL*:horloges, dépendances, environment.* PhD thesis, l'Université de Rennes I, IFSIC, Jan. 1993.

[6] C.Y.Lee. Representation of switching circuits by binary decision programs. *Bell. Syst. Thch. J.*, 38:985–999, July, 1959.

[7] R.E.Bryant. Graph-based algorithm for boolean function manipulation. *IEEE trans. Comput.*, C-35(8), Aug. 1986.

[8] J. Rumbaugh, I. Jacobson, and G. Booch. The unified modeling language reference manual. *Addison-Wesley object technology series*, 1999.

[9] S.B.Akers. Binary decision diagrams. *IEEE Trans. Comput.*, C-27:509–516, June 1978.