Extracting threads from concurrent objects for the design of embedded systems

Mitsutaka Okazaki, Toshiaki Aoki and Takuya Katayama

2002/08/05

IS-RR-2002-019

Extracting threads from concurrent objects for the design of embedded systems

Mitsutaka 0kazaki
m-okaza@jaist.ac.jp
Japan Advanced Institute of Science and Technology

Toshiaki Aoki
toshiaki@jaist.ac.jp
Japan Advanced Institute of Science and Technology / PRESTO Japan Science and Technology Corporation

Takuya Katayama
katayama@jaist.ac.jp
Japan Advanced Institute of Science and Technology

Abstract

As a result of the increasing size and complexity of embedded systems, object-oriented techniques are going to be adopted in the embedded software development. In embedded software developments, we have to consider non-functional requirements such as real-time properties and resource requirements. To deal with these requirements, some methodologies design the system using a thread-based approach. In such approach, we need to extract threads from the concurrent objects defined in the analysis model. However, current methodologies do not provide enough support to do so. In this paper, we propose a formal approach to extract threads from concurrent objects. We also present an experimental application of the proposed approach to the development of a device driver.

1 Introduction

Due to the increase in size and complexity of embedded systems, object-oriented techniques are going to be adopted in embedded software developments. However, it is still difficult to solve some domain specific problems in embedded systems. The main problems are to satisfy non-functional requirements such as real-time and resource constraints. In the existing object-oriented design, it is hard to deal with these constraints.

In object-oriented developments, we analyze the target system to specify the logical behavior of the system as a set of concurrent objects. If we consider a system which do not have severe non-functional constraints, like an enterprise system, we can directly design and implement objects as software modules which are performed concurrently. However, this is not possible in embedded system developments because of the existence of such constraints. Thus, we need a design model which is suitable for analyzing severe non-functional constraints.

Functions of a system are realized through the cooperation of objects. Objects communicate with each other. The execution process of a function can be represented as a sequence of communication between objects as shown in Figure 1. We call such a sequence a *thread*. Real-time constraints are given to threads as deadlines.

Figure 1 shows the relationship between objects, threads and real-time constraints.

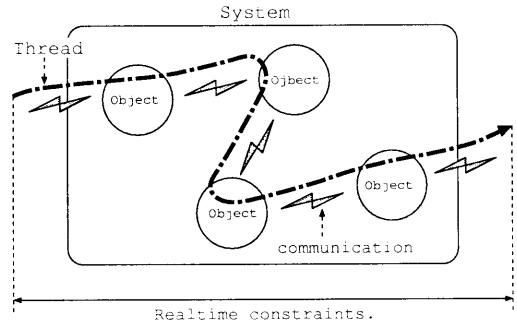

Figure 1: Objects, threads and real-time constraints.

In the design of embedded systems, threads are modified frequently to satisfy severe real-time constraints. If we use an object-based model, the behavior of objects must be changed to satisfy the constraints. In the worst case, we have to divide, unify or remove the objects. It causes the destruction of the logical structure of objects which are defined in the analysis phase.

There are some development methodologies for embedded systems which design the target system using a thread-based approach. OCTOPUS[9] and SES approach[2] have adopted a thread-based design approach. In these meth-

ods, real-time properties are considered after threads are extracted from an analysis model. However, they do not provide enough support on how to derive a design model.

To obtain such design models, we propose in this paper a transformation method from analysis model to design model. In our approach, both analysis and design models are defined by Concurrent Regular Expressions[1]. The design model is derived from the analysis model using an axiomatic system for equivalent transformation using concurrent regular expressions. The equivalent transformation is a transformation that changes the representation of the models but never changes their behavior.

We also introduce an experimental application of our method in the development of the PCM[1] device driver. This driver supplies 3 channels PCM audio playbacks. Our main objectives in this experiment are:

- to show that we can derive a design model with our method in a real application development, and
- to show that we can implement code based on our design model.

The remainder of this paper is organized as follows. In Section 2 we introduce concurrent regular expressions. Section 3 describes how to formalize analysis and design models on concurrent regular expressions. In Section 4, we introduce an axiomatic system for equivalent transformation on concurrent regular expressions. Section 5 describes an experiment in PCM device driver development. Section 6 gives some conclusions and the directions of our future work.

2 Concurrent Regular Expressions

In this section, we introduce Concurrent Regular Expressions(CREs) which are used to formalize our analysis and design models. CREs are used to model concurrent systems. Vijay K. Garg and M.T. Ragunath proposed it as algebraic descriptions of the language of Petri nets[1].

There are many existing algebraic specification methods for concurrent systems such as process algebra CCS[7], CSP[6] and ACP[8]. These methods model the states of a system explicitly. A process is described as actions which are performed at a state in a system. The system is defined as a set of such processes and transition relationships between them. On the other hand, CREs directly model the behavior of a whole system. States in a system are ignored.

In our approach, we do not consider the states of a system. Therefore, we choose simple CREs as our modeling method.

CREs are extension of regular expressions with four operators: interleaving, interleaving-closure, synchronous composition and renaming. In this paper, we use only interleaving and synchronous composition operators. We omit to explain the other operators.

We model a system with the interleaving operator if the system consists of threads which are executed concurrently. If the system consists of objects which communicate with each other, we use the synchronous composition operator.

Let Σ be a finite set of symbols. Concurrent regular expressions on Σ consist of symbols in $\Sigma \cup \{\perp, \epsilon\}$ and operators: choice(+), sequence(.), closure(*), interleaving($\|$) and composition($[S]$). $\perp$ and ϵ are the special symbols. $\perp$ means an empty set and ϵ means an empty sequence. For any expression P, $P.\perp = \perp.P = \perp$ and $P.\epsilon = \epsilon.P = P$ hold.

Expressions which contain no interleaving and synchronous composition operators are the same as *regular expressions* as known so far. The syntax of CREs on Σ is defined as follows.

Definition 2.1 (Concurrent Regular Expressions)

1. c *is a CRE if* $c \in \Sigma \cup \{\perp, \epsilon\}$
2. $P + Q$, $P.Q$ *and* P^* *are CREs if* P *and* Q *are CREs.*
3. $P\|Q$ *is a CRE if* P *and* Q *are CREs.*
4. $P[S]Q$ *is a CRE if* P *and* Q *are CREs and* $S \subseteq \Sigma$.
5. (P) *is a CRE if* P *is a CRE.*

We define the priority of the operators as $* > . > + > \| > [S]$. We can omit the parentheses in a CRE if it does not become ambiguous. For example, we can simply write $a^*.b + c$ instead of $((a^*).b) + c$.

The meaning of a concurrent regular expression P is defined as a set of sequences of symbols denoted by $L(P)$. We also call $L(P)$ the language of P. The sequences of symbols on Σ are defined as follows.

Definition 2.2 (Sequences)

1. c *is a sequence if* $c \in \Sigma \cup \{\epsilon\}$
2. $x \cdot y$ *is a sequence if* x *and* y *are sequences.*

For example, $a \cdot b \cdot c$ is a sequences on $\Sigma = \{a, b, c\}$.

ϵ in a sequence means the zero length sequence. For any sequence x, $\epsilon \cdot x = x \cdot \epsilon = x$ holds.

The definition of L is as follows.

Definition 2.3 (The language of CREs)

Let a,b,c be a symbol in Σ. *Let* P, Q *be CREs on* Σ *and* w, x, y *be sequences on* Σ

1. $L(\perp) = \phi$
2. $L(\epsilon) = \{\epsilon\}$

[1] An abbreviation of Plus Code Modulation

3. $L(c) = \{c\}$ *if* $c \in \Sigma$

4. $L(P.Q) = \{x \cdot y | x \in L(P), y \in L(Q)\}$

5. $L(P + Q) = L(P) \cup L(Q)$

6. $L(P^*) = \bigcup_{i=0,1,...} L(P^i)$

7. $L(P||\epsilon) = L(\epsilon||P) = L(P)$

8. $L(a||b.Q) = L(a.\epsilon||b.Q), L(a.P||b) = L(a.P||b.\epsilon)$

9. $L(a.P||b.Q) = L(a.(P||b.Q)) \cup L(b.(a.P||Q))$

10. $L(P||Q) = \{w | x \in L(P), y \in L(Q), w \in L(\hat{x}||\hat{y})\}$

11. $L(P[S]Q) = \{w | w \in (\Sigma_P \cup \Sigma_Q)^*, w/(\Sigma_P \cup S) \in L(P), w/(\Sigma_Q \cup S) \in L(Q)\}$

12. $L((P)) = L(P)$

$\hat{x}$ is a regular expression which corresponds to a sequence x. For example, $\hat{x} = a.b.c$ for $x = a \cdot b \cdot c$. Σ_P is a set of all symbols which appear on expression P. P^i is defined as the *ith* sequence of P. For example, $P^1 = P, P^2 = P.P$ and $P^3 = P.P.P, ...$ P^0 is defined as ϵ for any P.

w/Σ means a restriction of w over Σ. For instance, $a \cdot c \cdot a \cdot d \cdot a \cdot b/\{a, c\} = a \cdot c \cdot a \cdot a$

The operator [S] is different from that of original CREs. We extend the operator [ ] in the original CREs to [S]. The original [ ] is the same as our [S] if $S = \phi$. In the remainder of this paper, we use [ ] for $[\phi]$.

We say that an operator ω has the distributive property if $L((A + B)\omega C) = L((A\omega C) + (B\omega C))$ always holds. [ ] does not have this property. For instance, $L((a + c)[\,]c) = \{c\}$ but $L((a[\,]c) + (c[\,]c)) = \{a.c, c.a, c\}$

As a result of extending [ ] with S, we can give [S] the distributive property. See the distributive axiom in Section 4.

3 Formalizing Analysis and Design Models

In this section, we formalize both analysis and design models with CREs. We define the analysis model as concurrent objects and the design model as concurrent threads. We describe the behavior of a system with a set of *action sequences*. Let us begin with the description of actions.

3.1 Action

An *action* means the atomic behavior of the system. We define the action as an event, a method invocation or a fragment of program code. Syntactically, an action is described as a string.

Definition 3.1
An action is a string with lower-case character. The string is sometimes with subscripts. For example, a, b, c, $a_0, a_1, ..., a_n$, open, close, post *and* get *are actions.*

3.2 Action Sequence

An action sequence is a set of ordered actions. Actions are ordered following the time of their occurrence in the system. If an action a occurs earlier than an action b, a appears before b in an action sequence.

Consider a system which performs the actions login, work and exit in this order. The behavior of the system is defined as login · work · exit.

3.3 Behavior

The whole behavior of a system is defined as a set of action sequences. We use a concurrent regular expression to describe the set of action sequences.

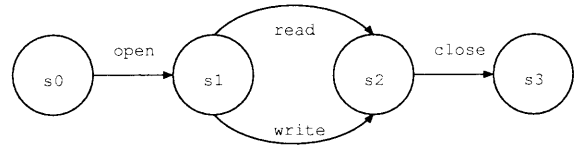

Figure 2: Automata

Let us consider a system whose behavior is defined by the automata shown in Figure 2. This automata starts at the initial state s_0. After the action open occurs, its state changes to s_1. Then after the action read or write, the system reaches the state s_2. Finally, the action close occurs and the system reaches the final state s_3. We can describe such behavior with the action sequences open·read·exit and open·write·exit. The behavior of the system is defined by the set {open·read·exit, open·write·exit}.

The behavior of a system could be too large and sometimes defined by an infinite set of action sequences. It is hard to write such a set directly. However, we can describe the behavior using a notation that makes loops and switches explicit. We use CREs as such a notation. CREs allow us to describe loops with closure operators and switches with choice operators.

By mapping actions to symbols on CREs, we can describe the behavior of a system as the language of a CRE. Intuitively, we can write P.Q if an action sequence Q occurs just after P. In the same way, P + Q means P or Q occurs exclusively. P^* means an arbitrary number of loops of sequence P. When P and Q execute concurrently, we write P||Q. If there is communication between concurrent P and Q, P[ ]Q is used.

For example, the behavior of the automata shown in Figure 2 can be described with the regular expression:
open.(read + write).close
The behavior corresponding to this expression is its language. i.e.

L(open.(read + write).close) =
{open · read · close, open · write · close}

3.4 Formalizing Analysis Model

An analysis model is defined as a system which consists of concurrent objects. The objects communicate with each other. Let $O_1, O_2, \ldots O_n$ be regular expressions which means behavior of objects in a system. The analysis model is defined as follows.

$O_1[\,]O_2[\,]\ldots[\,]O_n$.

We omitted the parentheses on the expression above because the associative law holds on [ ]. There is no || operators in the expression for an analysis model. We call such expression a *concurrent object expression.*

3.5 Formalizing Design Model

A design model is defined as a system which consists of concurrent threads. There is no communication between the threads. We formalize the design model with a *concurrent thread expression.* A concurrent thread expression is a concurrent regular expression without [ ] operators. For example, a.b.c||d.c.f and a.(b||c).d are concurrent thread expressions.

3.6 Formalizing Objects and Threads

Both objects and threads are defined as regular expressions. In our models, objects and threads have no internal concurrency. They are modeled in the same way except for the communication between them. Objects run concurrently and communicate with each other. On the other hand, threads run concurrently without communicating with each other.

3.7 Concurrency

Suppose that there is a system which consists of two threads. P and Q are CREs which define the behavior of the two threads in the system. These threads are executed concurrently. Then, the behavior of the system is defined as P||Q.

According to the language definition of ||, that is, 7 to 10 in the definition of L, concurrent threads are a set of interleaved action sequences of the threads. Suppose that P ≡ a.b and Q ≡ c.d. Then, the behavior of the system is a.b||c.d. L(a.b||c.d) represents all the interleaved sequences of a.b and c.d. Therefore, L(a.b||c.d) =
{a·b·c·d, a·c·b·d, a·c·d·b, c·a·b·d, c·a·d·b, c·d·a·b}

3.8 Communication

Suppose that there is a system which consists of two objects. P and Q are CREs which define the behavior of the two objects in the system. These objects run concurrently and communicate with each other. Then, the behavior of the system is defined as P[ ]Q.

P[ ]Q has the same meanings as P||Q if there is no communication between P and Q. In P[ ]Q, the same symbols appering in both P and Q are called *communication symbols.* These symbols mean actions for *synchronized communication* between P and Q. In our approach, all communication between objects are synchronized communication. Synchronized communication is a communication between objects which satisfies the following two rules:

- Objects block until the end of communication.
- No communication fails.

Assume that there are two objects defined as a.b and a.c in a system and these objects run concurrently. If there is no communication between the objects, the behavior of the system is the following.
{a · a · b · c, a · b · a · c, a · b · c · a, a · a · c · b}

This behavior contains some sequences which do not satisfy the synchronized communication rules. a · b · a · c and a · b · c · a are such sequences. These sequences represent the behavior where the objects did not block until the end of the communication. There must be no symbols between two communication symbols if the communication succeed. In a · b · a · c, there is the action b between the communication symbol a. This sequence means that the object a.b performs the action b before the object a.c finishes a. In other words, a.b performs b without blocking and waiting for object a.c to perform the action a. This behavior violates the synchronized communication rules.

The [ ] operator deletes such behavior and leaves:
{a · a · b · c, a · a · c · b}.

It is clear in the set above that once a occurs, the next symbol is also a. So the [ ] operator reduces a.a to a. Finally, we can get the set {a · b · c, a · c · b} as the meaning of L(a.b[ ]a.c).

According to the synchronized communication rules, communication must never fail. However, we can describe an expression that violates this rule. For example, in a.b.a[ ]a.b.c, the second occurrence of a in a.b.a cannot communicate with a.b.c because a.b.c has only one occurence of a. If communication fails in a system, the whole behavior of the system becomes an empty set. Thus, L(a.b.a||a.b.c) = ϕ

4 Transformation

We define the transformation from an analysis model to a design model as the transformation from a concurrent object expression to a concurrent thread expression. This transformation will never change the behavior of expressions. In our approach, we achieve this transformation with

an axiomatic system for equivalent transformation on concurrent regular expressions.

4.1 Axiomatic System for Transformation

The axiomatic system which we propose has two rules and 12 axioms for the [ ] operator. Table 1 shows the axioms. In addition to these axioms, we need some axioms on regular expressions which are known so far as algebraic properties of regular expressions. There are 2 rules in this

Reflection	$A[S]A = A$
Zero	$A[S]\perp = \perp$
Identity	$A[S]\epsilon = A$ if $S \cap \Sigma_A = \phi$
Commutative	$A[S]B = B[S]A$
Associative	$(A[\ ]B)[\ ]C = A[\ ](B[\ ]C)$
Distributive	$(A + B)[S]C =$ $(A[S \cup (\Sigma_B \cap \Sigma_C)]C) + (B[S \cup (\Sigma_A \cap \Sigma_C)]C)$
Synchronous	$(x.B)[S](x.C) = x.(B[S \cup \{x\}]C)$
Confliction	$x.A[S]y.B = \perp$ if $x, y \in S \cup (\Sigma_x.A \cap \Sigma_y.B)$ and $x \neq y$
Interleaving	$x.A[S]y.B = x.(A[S]y.B) + y.(x.A[S]B)$ if $x \neq y$ and $x \notin S \cup \Sigma_B$ and $y \notin S \cup \Sigma_A$
Spining	$(A.x.B)[S](C.y.D) = (A[\ ]C).(x.B)[S](y.D)$ if $x, y \in (\Sigma_{x.B} \cap \Sigma_{y.D}) \cup S$ and $(\Sigma_{A.x.B} \cup S) \cap \Sigma_C = (\Sigma_{C.y.D} \cup S) \cap \Sigma_A = \phi$
Optimizing	$A[S]B = A[S \cap (\Sigma_A \cup \Sigma_B) \cap (\overline{\Sigma_A \cap \Sigma_B})]B$
Threads	$A[\ ]B = A\|B$ if $\Sigma_A \cap \Sigma_B = \phi$

Table 1: Axioms for equivalent transformation

axiomatic system. One is the rewrite rule and the other is the reduction rule.

Let A and B be concurrent regular expressions. $A \Leftrightarrow B$ means that A and B are derivable from each other by applying a rule. $A \overset{*}{\Leftrightarrow} B$ means A and B are derivable from each other by applying some (more than zero) rules.

Definition 4.1 (Rewrite rule)
Let A, B *and* P *be concurrent regular expressions.*

- $A \Leftrightarrow B$ *if* $A = B$ *holds by axioms.*
- $P \Leftrightarrow P[A/B]$ *if* $A \overset{*}{\Leftrightarrow} B$.

Note that $P[A/B]$ *is an expression where all* A *in* P *are replaced with* B. *For instance, we can obtain* a.Y.X, X.Y.a *or* a.X.a *from* $X.Y.X[X/a]$

Definition 4.2 (Reduction rule)
Let S, A *and* B *be concurrent regular expressions.*
$S \Leftrightarrow A^*.B$ *if* $S \overset{*}{\Leftrightarrow} A.S + B$ *and* $\epsilon \notin L(A)$

4.2 Soundness

Our axiomatic system defined above is sound. In other words,

$$\forall P, P'.L(P) = L(P') \text{ if } P \overset{*}{\Leftrightarrow} P'.$$

holds. Therefore, two expressions have the same behavior if one is derivable from the other.

4.3 Completeness

Sometimes, there are some CREs which have the same behavior as another CRE. We say the axiomatic system is complete if all equivalent CREs can be derived from an expression. We define the completeness as follows.

$$\forall P.P'.P \overset{*}{\Leftrightarrow} P' \text{ if } L(P) = L(P').$$

5 PCM Device Driver Development

To evaluate our approach and to apply it in the development of a real application, we developed the PCM device driver using our approach. The driver is a synthesizer of PCM data streams. It synthesizes some PCM data on the fly so that some PCM channels can be played through only one digital to analog converter called a DAC or D/A converter. The PCM channel is an abstraction of a PCM data stream to control its volume and frequency.

5.1 Target Environment

We suppose that the target system for this driver has the following hardware.

- 1 CPU, 1 hardware clock and 1 DAC.

In addition, we assume that an operating system which has at least the following functions are running on the system.

- I/O function: It is used to write a value from the driver to the DAC.
- Interrupt handler: This function is used to notify an event from hardware clock to software.
- Semaphore: It is used to implement the mutual exclusion of threads in the driver.

We used C language for the implementation.

5.2 Driver Specification

Let us show the requirement specification for the PCM driver.

- Play and stop up to 3 PCM channels concurrently.
- Frequency is changed for each channel.
- Volume is changed for each channel.
- Both frequency and volume are changed on the fly.

Application Interface (API) of the driver is shown in Table 2

Entry	Function	Arguments
PLAY	Start playing	Number of channel, Top address of PCM data, End address of PCM data
STOP	Stop playing	Number of channel
FREQ	Change frequency	Number of channel, Frequency
VOL	Change volume	Number of channel, Volume

Table 2: Application Interface for the PCM driver.

5.3 Analysis Phase

In the analysis phase, we define classes and behavior of objects in the system. Then we define the behavior of objects with a concurrent object expression. Let us start by defining the classes.

5.3.1 Class Definition

First, we analyzed the system and defined some classes as follows.

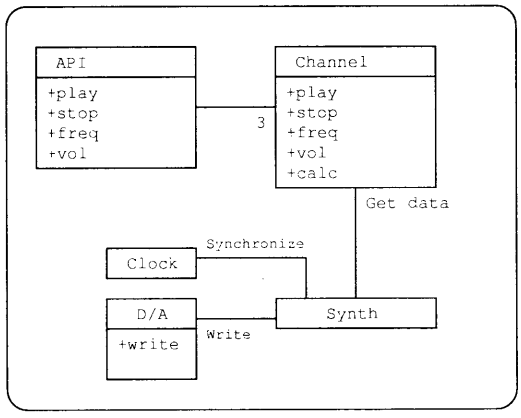

Figure 3: Class Diagram

API is a class which represents the API entries of the driver. Each method in API directly corresponds to a driver entry.

`Channel` is a class for each channel. The methods play, stop, freq and vol mean that start playing, stop playing, change frequency and change volume for the corresponding channel. `calc` is a method for calculating the output value of the channel. Once `calc` is invoked, it generates the output for 1 clock time.

`Synth` is a synthesizer for 3 channels and Clock is a class for hardware clock. `Synth` is synchronized with clock signals. Each time it synchronizes, it synthesized an output value from the output values of all the channels. Then, it writes the syhthsized value to the D/A converter.

`D/A` is an abstraction of D/A converter. `write` is a method for writing a value to D/A converter. The real output of D/A follows that value.

5.3.2 Behavior Definition

We defined the behavior of objects as shown in Table 3. The objects `API`, `SYN`, `CLK` and `DAC` in the Table 3 are instances of classes `API`, `Synth`, `Clock` and `D/A`. The driver plays 3 PCM channels concurrently, so three instances of the class `Channel` are required. The objects CH_0, CH_1 and CH_2 are these instances.

The actions `play`, `stop`, `freq`.`vol` in API correspond to API entries. p_i, s_i, f_i and v_i correspond to `play`, `stop`, `freq` and `vol` the methods of CH_i.

The action `calc` is an action which means communication among CH_0, CH_1, CH_2 and `SYN`. The `calc` occurrs in CH_0, CH_1, CH_2 and `SYN` simultaneously. In this action, the objects CH_0, CH_1 and CH_2 calculate their output values and `SYN` generates an output value for D/A from these 3 output values. `clk` corresponds to the event from the clock. `write` corresponds to the write method of the class `D/A`.

Object	Behavior
API	$(\texttt{play}.(p_0 + p_1 + p_2) +$ $\texttt{stop}.(s_0 + s_1 + s_2) +$ $\texttt{freq}.(f_0 + f_1 + f_2) +$ $\texttt{vol}.(v_0 + v_1 + v_2))^*$
CH_0	$(p_0 + s_0 + f_0 + v_0 + \texttt{calc})^*$
CH_1	$(p_1 + s_1 + f_1 + v_1 + \texttt{calc})^*$
CH_2	$(p_2 + s_2 + f_2 + v_2 + \texttt{calc})^*$
SYN	$(\texttt{clk}.\texttt{calc}.\texttt{write})^*$
CLK	$\texttt{clk}^*$
DAC	$\texttt{write}^*$

Table 3: Behavior of objects

- `API` invokes an appropriate channel after one of the API entries is called. For example, one of the `play` methods of the channel objects (p_0, p_1 or p_2) is called after the API entry PLAY(`play`) is called.
- CH_0, CH_1 and CH_2 start playing, stop playing, change frequency, change volume or calculate their output value repeatedly.
- `SYN` synchronizes with a clock event (`clk`) from the hardware clock, then calculates the output value (`calc`) and sends it to the D/A converter using the `write` method.
- `CLK` generates a clock event repeatedly.
- `DAC` accepts the `write` action from `SYN` repeatedly.

5.3.3 Analysis Model

The design model for the PCM device driver is defined as the following concurrent object expression.

$\mathtt{API[\,]CH_0[\,]CH_1[\,]CH_2[\,]SYN[\,]CLK[\,]DAC}$

Note that we use the name of object instead of concurrent regular expressions. For example, simply $\mathtt{CLK[\,]DAC}$ is the same as $\mathtt{clock^*[\,]write^*}$.

5.4 Design Phase

In this section, we design the system. We derive concurrent thread expressions from concurrent object expressions using our axiomatic system. If we apply axioms only, the steps for transformation are too large to complete by hand. Let us show some theorems to decrease the size of the transformation steps.

5.4.1 Preparation

Lemma 5.1
$\mathtt{P.X[\,]Q.Y} \overset{*}{\Leftrightarrow} \mathtt{(P.(X[\,]Q.Y)) + (Q.(P.X[\,]Y))}$
if $\mathtt{P} = \bigcup_{i=0}^{n} \mathtt{a}_i, \mathtt{Q} = \bigcup_{i=0}^{m} \mathtt{b}_i, \Sigma_P \cap \Sigma_Q = \phi$
where $\bigcup_{i=0}^{n} \mathtt{a}_i \overset{def}{=} \mathtt{a}_0 + \mathtt{a}_1 + ...\mathtt{a}_n$.

Proof
According to the assumptions for P, Q,
$\mathtt{P.X[\,]Q.Y} = (\bigcup_{i=0}^{n} \mathtt{a}_i).\mathtt{X[\,]}(\bigcup_{i=0}^{m} \mathtt{b}_i).\mathtt{Y}$
By distributive law of . and [ ],
$\overset{*}{\Leftrightarrow} (\bigcup_{i=0}^{n} \bigcup_{j=0}^{m} (\mathtt{a}_i.\mathtt{X[\,]b}_j.\mathtt{Y}))$
$\overset{*}{\Leftrightarrow} (\bigcup_{i=0}^{n} \mathtt{a}_i.(\bigcup_{j=0}^{m} (\mathtt{X[\,]b}_j.\mathtt{Y}))) +$
$\quad (\bigcup_{j=0}^{m} \mathtt{b}_i.(\bigcup_{i=0}^{n} (\mathtt{a}_i.\mathtt{X[\,]Y})))$
$\overset{*}{\Leftrightarrow} (\bigcup_{i=0}^{n} \mathtt{a}_i.(\mathtt{X[\,]}(\bigcup_{j=0}^{m} \mathtt{b}_j).\mathtt{Y})) +$
$(\bigcup_{j=0}^{m} \mathtt{b}_i.((\bigcup_{i=0}^{n} \mathtt{a}_i).\mathtt{X[\,]Y}))$
$\equiv \mathtt{(P.(X[\,]Q.Y)) + (Q.(P.X[\,]Y))}$

Theorem 5.2
$\mathtt{(P^*[\,]Q^*)} \overset{*}{\Leftrightarrow} \mathtt{(P+Q)^*}$
if $\mathtt{P} = \bigcup_{i=0}^{n} \mathtt{a}_i, \mathtt{Q} = \bigcup_{i=0}^{m} \mathtt{b}_i$ *and* $\Sigma_P \cap \Sigma_Q = \phi$

Proof
$\mathtt{P^*[\,]Q^*} \overset{*}{\Leftrightarrow} \mathtt{(\epsilon + P.P^*)[\,](\epsilon + Q.Q^*)}$
By the distributive axiom,
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + P.P^* + Q.Q^* + (P.P^*[\,]Q.Q^*)} \ldots (1)$
By Lemma 5.1,
$\mathtt{(P.P^*[\,]Q.Q^*)} \overset{*}{\Leftrightarrow} \mathtt{(P.(P^*[\,]Q.Q^*) + Q.(P.P^*[\,]Q^*))}$
$\overset{*}{\Leftrightarrow} \mathtt{(P.(\epsilon + P.P^*[\,]Q.Q^*) + Q.(P.P^*[\,]\epsilon + Q.Q^*))}$
$\overset{*}{\Leftrightarrow} \mathtt{(P.(Q.Q^* + (P.P^*[\,]Q.Q^*)) + Q.(P.P^* + (P.P^*[\,]Q.Q^*))}$
$\overset{*}{\Leftrightarrow} \mathtt{P.Q.Q^* + Q.P.P^* + (P+Q).(P.P^*[\,]Q.Q^*)} \ldots (2)$
By (1) and (2),
$\mathtt{\epsilon + P.P^* + Q.Q^* + (P.P^*[\,]Q.Q^*)}$
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + (P+Q).(\epsilon + P.P^* + Q.Q^* + (P.P^*[\,]Q.Q^*))}$
By the reduction rule,
$\mathtt{\epsilon + P.P^* + Q.Q^* + (P.P^*[\,]Q.Q^*)} \overset{*}{\Leftrightarrow} \mathtt{(P+Q)^*}$
Therefore, $\mathtt{(P^*[\,]Q^*)} \overset{*}{\Leftrightarrow} \mathtt{(P+Q)^*}$

Theorem 5.3
$\mathtt{(a.b + X)^*[\,]b^*} \overset{*}{\Leftrightarrow} \mathtt{(a.b+X)^*}$
if $\mathtt{a} \neq \mathtt{b}, \mathtt{b} \notin \Sigma_X$

Proof
$\mathtt{(a.b + X)^*[\,]b^*}$
By the distributive axiom,
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + (a.b+X).(a.b+X)^*[\,]\epsilon + b.b^*}$
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + ((a.b+X).(a.b+X)^*[b]\epsilon) + (\epsilon[b]b.b^*) + ((a.b+X).(a.b+X)^*[b]b.b^*)}$
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + (a.b.(a.b+X)^*[b]\epsilon) + (X.(a.b+X)^*[b]\epsilon) + (a.b.(a.b+X)^*[b]b.b^*) + (X.(a.b+X)^*[b]b.b^*)} \ldots (1)$
By the confliction axiom
$\mathtt{a.b.(a.b+X)^*[b]\epsilon} \overset{*}{\Leftrightarrow} \mathtt{a.(b.(a.b+X)^*[b]\epsilon)} \overset{*}{\Leftrightarrow} \perp$
Therefore,
$(1) \overset{*}{\Leftrightarrow} \mathtt{\epsilon + a.b.((a.b+X)^*[b]b^*) + X.(((a.b+X)^*[b]\epsilon) + ((a.b+X)^*[b]b.b^*))} \ldots (2)$
By the distributive axiom,
$\mathtt{((a.b+X)^*[b]\epsilon) + ((a.b+X)^*[b]b.b^*)}$
$\overset{*}{\Leftrightarrow} \mathtt{(a.b+X)^*[b](\epsilon + b.b^*)} \overset{*}{\Leftrightarrow} \mathtt{(a.b+X)^*[b]b^*}$
Hence,
$(2) \overset{*}{\Leftrightarrow} \mathtt{\epsilon + a.b.((a.b+X)^*[b]b^*) + X.((a.b+X)^*[b]b^*)}$
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + (a.b+X).((a.b+X)^*[b]b^*)}$
$\overset{*}{\Leftrightarrow} \mathtt{\epsilon + (a.b+X).((a.b+X)^*[\,]b^*)}$
By the reduction rule,
$\mathtt{((a.b+X)^*[\,]b^*)} \overset{*}{\Leftrightarrow} \mathtt{(a.b+X)^*}$

5.5 Transformation

We can transform the concurrent thread expression:
$\mathtt{API||SYN}$
from the concurrent object expression:
$\mathtt{API[\,]CH_0[\,]CH_1[\,]CH_2[\,]SYN[\,]CLK[\,]DAC}$

We show the outline of the transformation here.
First, we transform $\mathtt{CH_0[\,]CH_1[\,]CH_2}$. Assume that $\mathtt{CC_i} \equiv \mathtt{(p_i + s_i + f_i + v_i)}$ and by Theorem 5.2,
$\mathtt{CH_i} \equiv \mathtt{(CC_i + calc)*} \overset{*}{\Leftrightarrow} \mathtt{CC_i^*[\,]calc^*}$.
So, $\mathtt{CH_0[\,]CH_1[\,]CH_2} \equiv$
$\quad \mathtt{CC_0^*[\,]calc^*[\,]CC_1^*[\,]calc^*[\,]CC_2^*[\,]calc^*}$
By associative law of [ ],
$\overset{*}{\Leftrightarrow} \mathtt{CC_0^*[\,]CC_1^*[\,]CC_2^*[\,](calc^*[\,]calc^*[\,]calc^*)}$
$\overset{*}{\Leftrightarrow} \mathtt{CC_0^*[\,]CC_1^*[\,]CC_2^*[\,]calc^*}$

Next, we transform $\mathtt{API[\,]CC_i^*}$. By using theorem 5.2 repeatedly,
$\mathtt{CC_i} \overset{*}{\Leftrightarrow} \mathtt{p_i^*[\,]s_i^*[\,]f_i^*[\,]v_i^*}$.

Suppose that X is an expression which holds ($\mathtt{API} \overset{*}{\leftrightarrow} (\mathtt{play}.\mathtt{p}_i + \mathtt{X})$). By Theorem5.3,
$\mathtt{API}[\,]\mathtt{p}_i^* \overset{*}{\leftrightarrow} (\mathtt{play}.\mathtt{p}_i + \mathtt{X})^*[\,]\mathtt{p}_i^* \overset{*}{\leftrightarrow} (\mathtt{play}.\mathtt{p}_i + \mathtt{X})^*$
For $\mathtt{API}[\,]\mathtt{s}_i, \mathtt{f}_i$ or $\mathtt{v}_i$, we can transform with the same way as $\mathtt{API}[\,]\mathtt{p}_i^*$. $\mathtt{API}[\,]\mathtt{CC}_i^* \overset{*}{\leftrightarrow} \mathtt{API}[\,]\mathtt{p}_i^*[\,]\mathtt{s}_i^*[\,]\mathtt{f}_i^*[\,]\mathtt{v}_i^* \overset{*}{\leftrightarrow} \mathtt{API}$
Then,
$\mathtt{API}[\,]\mathtt{CC}_0^*[\,]\mathtt{CC}_1^*[\,]\mathtt{CC}_2^* \overset{*}{\leftrightarrow} \mathtt{API}$
Therefore,
$\mathtt{API}[\,]\mathtt{CH}_0[\,]\mathtt{CH}_1[\,]\mathtt{CH}_2[\,]\mathtt{SYN}[\,]\mathtt{CLK}[\,]\mathtt{DAC}$
$\overset{*}{\leftrightarrow} \mathtt{API}[\,]\mathtt{calc}^*[\,]\mathtt{SYN}[\,]\mathtt{CLK}[\,]\mathtt{DAC}$
$\overset{*}{\leftrightarrow} \mathtt{API}[\,]\mathtt{calc}^*[\,](\mathtt{clk.calc.write})^*$
$\overset{*}{\leftrightarrow} \mathtt{API}[\,](\mathtt{clk.calc.write})^*$
$\overset{*}{\leftrightarrow} \mathtt{API}||\mathtt{SYN}$

5.6 Extracting Threads

In this section, we obtain *subthreads* from the design model. An subthread can be directly implemented as a function of C program code. In the implementation phase, we implement our PCM driver based on subthreads.

As a result of the design, the behavior of the system is defined as the two concurrent threads API and SYN. These threads are still a little far from their implementation, so we extract subthreads of API and SYN.

5.6.1 Subthreads

We model a subthread as a thread which has the following two properties.

- A subthread is always ignited by an external event.
- Subthreads run exclusively if they belong to the same thread.

The external event is an action which corresponds to an event from outside of the driver.

Let us define the rule to obtain the subthreads.

Definition 5.1 (Subthreads)
Assume that T *is a CRE and* $\mathtt{P}_i$ *is a regular expression. Let* $\mathtt{e}_0, \mathtt{e}_1, \ldots \mathtt{e}_n$ $(\mathtt{n} \geq 0)$ *be actions which mean external events.*
$\mathtt{e}_0.\mathtt{P}_0, \mathtt{e}_1.\mathtt{P}_1, \ldots \mathtt{e}_n.\mathtt{P}_n$ *are the subthreads of* T
where $\mathtt{T} \equiv (\mathtt{e}_0.\mathtt{P}_0 + \mathtt{e}_1.\mathtt{P}_1 + \ldots + \mathtt{e}_n.\mathtt{P}_n)^*$

5.6.2 Extracting Subthreads

Let us obtain the subthreads of API. In the thread API, play, stop, freq and vol are the external events. All of them occurr when one of the API entries is called from a client of the driver. According to the definition of subthreads, we extract four subthreads PLAY, STOP, FREQ and VOL shown in Table 4 from API.

Name	Expression
PLAY	$\mathtt{play}.(\mathtt{p}_0 + \mathtt{p}_1 + \mathtt{p}_2)$
STOP	$\mathtt{stop}.(\mathtt{s}_0 + \mathtt{s}_1 + \mathtt{s}_2)$
FREQ	$\mathtt{freq}.(\mathtt{f}_0 + \mathtt{f}_1 + \mathtt{f}_2)$
VOL	$\mathtt{vol}.(\mathtt{v}_0 + \mathtt{v}_1 + \mathtt{v}_2)$
CLK	clk.calc.write

Table 4: The subthreads of API and SYN

In the thread SYN, the action clk is an external event because clk corresponds to an event from the hardware clock. We extract only one subthread CLK shown in Table 4

5.7 Implementation Phase

We directly implemented the five subthreads of Table 4 as C the functions. See appendix to find the code. The names of functions in the code are the same as the names of the subthreads shown in Table 4.

5.7.1 Interface for External Event

A subthread is always invoked when an external event occurs. There must be an interface that receives the external event at the beginning of the subthread.

In the subthreads PLAY, STOP, FREQ and VOL, their external events correspond to API invocation from clients of the driver. We can represent the interface for external events as C function entries themselves. The arguments of the functions follow the API specification. For example, API VOL has two arguments. One is the number of channel and the other is the amount of volume. The declaration of the function VOL is as follows.

```
void VOL(int ch, int value);
```

The subthread CLK is synchronized with the hardware clock. We suppose that the operating system observes the clock and it invokes the function CLK whenever the clock event occurs. We registered the address of CLK to the operating system as an interrupt handler for the hardware clock when the driver is initialized. [2]

5.7.2 Implementing the Body of Subthreads

We implemented the body of the functions taking into consideration the meaning of each action. Each function is filled with proper code. We also use some external variables shared across the functions.

For example, VOL is a thread whose behavior is $\mathtt{vol}.(\mathtt{v}_0 + \mathtt{v}_1 + \mathtt{v}_2)$. We implemented the body of the function as follows.

[2]This initialization code is omitted on the sample source.

```
void VOL(int ch, int value)
{
  volume[ch] = value;
}
```

The variable volume is an array which stores the volume of three channels. volume[ch] = value is an abbreviation of following code.

```
switch(ch){
case 0:
  volume[0] = value;
  break;
case 1:
  volume[1] = value;
  break;
case 2:
  volume[2] = value;
  break;
default:
  /* ERROR */
  break;
}
```

This code fragment intuitively corresponds with $(v_0 + v_1 + v_2)$.

5.8 Implementing the Mutual Exclusion

According to the result of the design, the threads PLAY, STOP, FREQ and VOL never run concurrently. However, we implement these four subthreads as functions. They run concurrently if the external events come again before the functions finish their processes. To prevent such situation, we use a binary semaphore to make the functions run exclusively.

We added some code to function PLAY, STOP, FREQ and VOL. We wrote ENTER(sem) at the beginning of the functions and LEAVE(sem) at the end of the functions. The variable sem means a binary semaphore. ENTER is a function which makes the semaphore up. The caller thread of this function enters the critical section. LEAVE is also a function which makes the semaphore down and the caller leaves the critical section. Only one thread can enter the critical section. The threads are blocked if the semaphore is up when they enter. Threads wait until the semaphore is down.

For instance, the function VOL is implemented as follows.

```
void VOL(int ch, int value){
  ENTER(sem);
  volume[ch] = value;
  LEAVE(sem);
}
```

Similarly, ENTER and LEAVE are added to the PLAY, STOP and FREQ functions. Thus, PLAY, STOP, FREQ and VOL share one semaphore and they are executed exclusively.

6 Conclusion

In this paper, we proposed the axiomatic system for the equivalent transformation on CREs. Using this transformation we can systematically obtain design models from an analysis model. We applied our transformation method to the PCM device driver development. Using our approach we successfully derived a suitable design model from the analysis model of the driver and we extracted a set of threads from the design model and implemented them using C language.

As a future work, we are planning to apply our approach to larger and more complex systems. For the development of such systems, the transformation steps become extremely large. We will implement a system to help the transformation.

References

[1] Vijay K. Garg, M.T. Ragunath: *Concurrent regular expressions and their relationship to Petri nets*, Theoretical Computer Science 96, pp.285-304, 1992.

[2] Toshiaki Aoki and Takuya Katayama: SES Model for Object-Oriented Time Critical System Development, Proceedings of the IEEE International Conference on Artificial Intelligence and Computational Intelligence for Decision, Control, and Automation in Engineering and Industrial Applications ACIDCA'2000, pp.19-24, 2000.

[3] Toshiaki Aoki and Takuya Katayama: *SES Model for Object-Oriented Embedded System Development*, Japan Society for Software Science and Technology, FOSE'2000 Foundations of Software Engineering VII, pp.157-164, 2000.

[4] Toshiaki Aoki, Akira Kawaguchi, Tomoji Kishi and Takuya Katayama: *Synchronized Execution Sequence Based Software Architecture for Object-Oriented Embedded Systems*, First Working IFIP Conference on Software Architecture WICSA1, 1999.

[5] Arto Salomma : *Two Complete Axiom Systems for the Algebra of Regular Events* , Journal of the Association for Computing Machinery, Vol.13, No. 1, pp158-169, 1966.

[6] C.A.R. Hoare: *Communicating Sequential Process*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.

[7] Robin Milner : *Communication and Concurrency*, Prentice Hall, 1989

[8] J.C.M. Baeten: *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[9] Maher Awad, Juha Kuusela and Jurgen Ziegler : *Object-Oriented Technology for Real-Time Systems* , Prentice Hall, 1996.

[10] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorenson: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Appendix: A part of implementation

```
/* Clock Frequency (Hz) */
#define INPUT_CLOCK 50000
#define CH_MAX 3
static int f[CH_MAX], v[CH_MAX];
static short *start_adr[CH_MAX];
static short *current_adr[CH_MAX];
static short *end_adr[CH_MAX];
static double counter[CH_MAX];
static double counter_step[CH_MAX];
static int playflag[CH_MAX];
static SEMAFO sem=0;

/* play.(p_0+p_1+p_2) */
void PLAY(int ch, short *start, short *end){
  ENTER(sem);
  playflag[ch] = 1;
  start_adr[ch] = start;
  end_adr[ch] = end;
  LEAVE(sem);
}

/* stop.(s_0+s_1+s_2) */
void STOP(int ch){
  ENTER(sem);
  playflag[ch] = 0;
  LEAVE(sem);
}

/* freq.(f_0+f_1+f_2) */
void FREQ(int ch, int value){
  ENTER(sem);
  f[ch] = value;
  counter[ch] = 0.0;
  counter_step[ch] = f[ch]/INPUT_CLOCK;
  LEAVE(sem);
}

/* vol.(v_0+v_1+v_2) */
void VOL(int ch, int value){
  ENTER(sem);
  v[ch] = value;
  LEAVE(sem);
}
```

```
/* clock.calc.write */
void CLK(){
  short mix = 0;
  int i;

  for(i=0;i<3;i++)
  {
    if(playflag[i])
    {
      /* calcurate the address counter */
      counter[i] += counter_step[i];
      if(counter[i]>=1.0)
      {
        counter[i]-=1.0;
        current_adr[i]++;
      }
      /* check the end of data */
      if(current_adr[i]==end_adr[i])
        playflag[i] = 0;

      mix += (*(current_adr[i]) * v[i]) >> 4;
    }
  }
  /* I/O Access */
  DAC_WRITE(mix);
}
```