修士学位論文
関数型言語による Timed CSP 検証技法の提案

首都大学東京理工学研究科
数理情報科学専攻
山川 武志

平成 22 年 1 月 8 日

目次

1 序論

1.1 並行プログラミングの問題

近年，パーソナルコンピュータのマルチコア化からも分かるように，シングルコアの問題（発熱，クロックの限界，ノイマンのボトルネック）をマルチコアにより解決する方向にある．これにともなって従来の逐次的なプログラミングから脱却し並行プログラミングを行うことの重要性が増してきている．OS やミドルウェアによる支援に頼るだけでなくプログラムで明示的に並行性を記述することで，マルチコアの性能をより引き出すことが可能である．

並行プログラムとは 2 つ以上の処理を並行して実行するプログラムのことであり，複数の CPU 上で実行される場合と単一 CPU 上でタイムシェアリング（一定時間ごとに処理が切り替わる擬似並列処理）や待ち行列によって実行される場合がある．FA 機器，自動車，ロボットなどの組み込みシステムにおいては同時に幾つかの処理を行う必要があるため，以前から並行プログラミングが行われている．

しかしながら，並行プログラミングには逐次的なプログラミングにはなかった並行性特有の問題（競合の危険性，デッドロック，ライブロック，リソース不足など）が付きまとう．また，独立に動く複数のプロセスの処理のタイミングの違いや，不規則に起こる割り込みに対する処理の必要性などからその振る舞いは複雑になり，以前から並行性が必要なシステムにおいては，そのデバッグの困難さや検証方法が問題となっている．システムが大きくなるにつれ，もはや場当たり的なデバックでは対処できずに潜在的バグを持ったままシステムが運用されている可能性がある．

1.2 CSP 理論による問題解決

これらの問題を解消し，並行システムの設計を安全，容易にするために考えられた数学的理論がプロセス代数 CSP(Communicating Sequential Processes)[1] である．CSP は並行システムを検証するためのモデリング言語であり，CSP によって設計されたシステムはその振る舞いを数学的にとらえることができる．これにより様々な要求仕様[*1]の検証が可能となり，並行性の問題がシステム開発の早い段階で解消できることになる．さらに CSP 実装用言語としては Occam[2] が開発されており，また，java，c，verilog などの言語にも CSP 実装用のライブラリー [3][4] が開発されているので，設計後はそのままソフトウェアまたはハードウェアに実装可能である．

CSP では並行システムを，逐次的な処理をするプロセスの集まりが互いに相互作用しながらシステム全体として振る舞うものとしてとらえる．このプロセス間の相互作用をイベントと呼び，プロセスはイベントが起こるごとに次の状態へと移っていく．

システムで特に時間による割り込みのあるシステムや一定時間内に確実に処理を終了しなければならないようなシステム（産業用ロボットや自動車などの制御システム）に対して，CSP は時間的

[*1] システムが何をしなければならないかを記述したもの

振る舞いを直接モデリングすることはできない．このようなシステムに対しては CSP の理論に時間概念を加え拡張した Timed CSP 理論を用いることで検証が可能となる．

1.3 ツールの必要性

並行システムを CSP で記述した際，その正当性の証明は人の手によって証明問題を解くように行うことも可能であるが，ユーザビリティや人為的なミスを防ぐためにはツールの利用が望ましい．CSP 検証ツールを用いることで数学的理論や検証方法など高度な知識を必要とせずにその検証が可能となる．

CSP の代表的な検証ツールとしては FDR2(Failures/Divergence Refinement2)[5] がある．しかしながら，FDR2 は現在 Timed CSP をサポートしておらず，その開発が待たれている．Timed CSP の検証が行えるツールとして PAT(Process Analysis Toolkit)[6] が開発されているが，まだ新しく現状 FDR2 ほどの高い信頼性は期待できない．そこで本研究では新たに Timed CSP に対する自動検証の技法提案とプロトタイプの作成を行った．

2 CSP について

2.1 CSP 概要

CSP は，プロセス代数と呼ばれる並行システムを形式的にモデリングする手法のひとつであり，1978 年にオックスフォードのアントニー・ホーアによって考案され改良されていった．CSP では並行システムを，独立した逐次プロセス群が互いに相互作用や通信をしながら動作するものとして設計する．相互作用や通信のことをイベントと呼び，同期をとったりデータのやりとりが行われたりする．データはチャネルと呼ばれる通信路を介したメッセージパッシング方式で送受信される．つまり，一方のプロセスが送信可能かつ，もう一方のプロセスが受信可能になったときにデータの受け渡しが行われる．

イベントやチャネルについては構築するシステムに応じて名前を定義する．また，プリミティブなプロセスとして $SKIP$ と $STOP$ があり，それぞれ正常終了と停止を意味する．CSP の代数 operator はこれらプロセスとイベントを元に持つ集合上に定義されており，代数 operator によりプロセスとイベントから新たなプロセスを構成する．つまり，operator はイベントとプロセスまたはプロセス間の関係を定義し新たなプロセスをつくりだす構成子の役割を果たしている．CSP のプロセス定義を表 1 に示す．マシンで読み込むことが可能な CSP マシンリーダブル (CSP_M) 記述も示す．

CSP のプロセスとしてモデリングされた並行システムはその取りうる軌跡などを調べることができ，またその観点から様々な検証が可能であり，システムの振る舞いが仕様を満たしているかどうかを確かめることができる．このことによりシステムの設計段階で並行性に関する問題は解消されていることが保証されるので，実装後のデバック作業の軽減やバグの潜在をなくすことが可能となる．

2.2 CSP 意味論

CSP はその表現の形式的意味論として操作的意味論，公理的意味論（代数的意味論），表示的意味論の 3 つの意味論を持つ．まず操作的意味論をもとに各 operator の意味をプロセスの振る舞いから説明し，公理的意味論ではプロセスの代数変換規則の説明，表示的意味論ではその元にもなっている trace，failure，divergence などの重要な概念を説明する．

	CSP 記述	CSP_M 記述	
$Proc$ ::=	$STOP$	`STOP`	
|	$SKIP$	`SKIP`	
|	$c \rightarrow Proc$	`c -> Proc`	*prefix*
|	$c?x : A \rightarrow Proc$	`c?x:A -> Proc`	*input*
|	$c!v \rightarrow Proc$	`c!v -> Proc`	*output*
|	$Proc \fatsemi Proc$	`Proc ; Proc`	*sequential*
|	$Proc \triangle Proc$	`Proc /\ Proc`	*interrupt*
|	$Proc \setminus A$	`Proc \ A`	*hiding*
|	$Proc[c \leftarrow d]$	`Proc[[c <- d]]`	*renaming*
|	$Proc \Box Proc$	`Proc [] Proc`	*external choice*
|	$Proc \sqcap Proc$	`Proc |~| Proc`	*internal choice*
|	$Proc \triangleright Proc$	`Proc [> Proc`	*timeout*
|	$Proc \parallel Proc$	`Proc || Proc`	*parallel*
|	$Proc \interleave Proc$	`Proc ||| Proc`	*interleaving*
|	$Proc [\![A]\!] Proc$	`Proc [|A|] Proc`	*sharing*
|	if b then $Proc$	`if b then Proc`	*conditional choice*
	else $Proc$	`else Proc`	

表 1　CSP のプロセス定義

2.2.1　操作的意味論

CSP の操作的意味論は推論規則によって与えられている．推論規則とは，ある条件のもとでいくつかの前提条件が成り立つときに結論が導かれる規則のことで，ここでは次のような形で表す．

前提条件 1
：
前提条件 n
―――――――[条件]
結論

また，あるプロセス $P1$ がイベント μ によってプロセス $P2$ に状態遷移するとき，これを

$$P1 \xrightarrow{\mu} P2$$

で表す．$SKIP$ は正常終了を意味するプロセスであり，成功を意味する特別なイベント $\surd$ によって $STOP$ へと移る．

$$SKIP \xrightarrow{\surd} STOP$$

■event prefix

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

$a \to P$ はイベント a 実行後にプロセス P として振舞うプロセスである．event prefix に関する規則は１つだけであり，前提条件なしにプロセス $a \to P$ はイベント a によりプロセス P に遷移することを意味している．

これにより，例えばコインを入れるとジュースを一度だけ出す単純な自動販売機 VM は次のように記述される．

$VM = coin \to (juice \to SKIP)$　　括弧は省略可

VM は最初 $coin$ 以外のイベントを受け付けない．イベント $coin$ によって VM はプロセス $(juice \to SKIP)$ に遷移する．さらにこのプロセスはイベント $juice$ により $SKIP$ へと移る．この様子は次のように表される．

$$VM \xrightarrow{coin} (juice \to SKIP) \xrightarrow{juice} SKIP$$

■prefix choice

$$\frac{}{(x : A \to P(x)) \xrightarrow{a} P(a)}[a \in A]$$

$x : A \to P(x)$ は集合 A の中のイベント a が起きた後，プロセス $P(a)$ として振る舞うプロセスである．例えば，イベント $accept$ が起きた場合はプリントし，イベント $shutdown$ が起きた場合は停止するプロセス $PRINTER$ は次のように記述される．

$$PRINTER = x : \{accept, shutdown\} \to \begin{array}{l} \texttt{if}\ x = accept\ \texttt{then}\ print \to STOP \\ \texttt{else}\ STOP \end{array}$$

これはまた，次のように簡易的に記述してもよい．

$$\begin{array}{ll} PRINTER = & accept \to print \to STOP \\ & \mid shutdown \to STOP \end{array}$$

■recursive

$$\frac{P \xrightarrow{\mu} P'}{N \xrightarrow{\mu} P'}[N = P]$$

P はその表現に N を含んでいるプロセスである．前述の VM は一度の販売で終了してしまうプロセスであるが，何度でも繰り返し販売できる VMR はつぎのように記述する．

$VMR = coin \to juice \to VMR$

この VMR がどのように振る舞うかは直感的にも理解しやすいが，推論規則において $N = VMR$, $P = coin \to juice \to VMR$ として考えれば $VMR \xrightarrow{coin} (juice \to VMR)$ がわかる．

$$\frac{(coin \to juice \to VMR) \xrightarrow{coin} (juice \to VMR)}{VMR \xrightarrow{coin} (juice \to VMR)}[VMR = coin \to juice \to VMR]$$

よって，VMR は次のようにイベント $coin$ と $juice$ を交互に繰り返し続けるプロセスとして定義される．

$$VMR \xrightarrow{coin} (juice \rightarrow VMR) \xrightarrow{juice} VMR \xrightarrow{coin} (juice \rightarrow VMR) \xrightarrow{juice} \cdots$$

■output

$$\frac{}{(c!v \rightarrow P) \xrightarrow{c.v} P}$$

$c!v \rightarrow P$ はチャネル c に対してデータ v を送信した後，つまりイベント $c.v$ を起こした後，プロセス P として振舞うプロセスである．記号！は送信を意味しているがこのプロセスが起こすイベントは $c.v$ である．output の推論規則は prefix とほぼ同様である．

■input

$$\frac{}{(c?x \rightarrow P(x)) \xrightarrow{c.v} P(v)}$$

$c?x \rightarrow P(x)$ はチャネル c から送られてきたデータを変数 x で受け取った後，つまりイベント $c.v$ の後，プロセス $P(v)$ として振舞うプロセスである．記号？は受信を意味している．

例えば，チャネル in からデータを受け取り，２乗した値をチャネル out に対して送信するようなプロセス P は次のように記述される．

$$P = in?x \rightarrow out!x * x \rightarrow P$$

■external choice

$$\frac{P \xrightarrow{a} P'}{\begin{array}{l}(P \,\square\, Q) \xrightarrow{a} P' \\ (Q \,\square\, P) \xrightarrow{a} P'\end{array}} \qquad \frac{P \xrightarrow{\tau} P'}{\begin{array}{l}(P \,\square\, Q) \xrightarrow{\tau} (P' \,\square\, Q) \\ (Q \,\square\, P) \xrightarrow{\tau} (Q \,\square\, P')\end{array}}$$

$P \,\square\, Q$ はプロセスの選択を意味する．プロセス P，Q のどちらに関与するイベントが先に起こるかによって実行するプロセスが決定される．例えば，コインを入れたあとコーラかレモネードを選べる自動販売機 VMD は次のように記述できる．

$$\begin{array}{l}VMD = coin \rightarrow (\quad coke \rightarrow VMD \\ \qquad\qquad\qquad\quad \square\, lemonade \rightarrow VMD\ )\end{array}$$

推論規則にあるイベント τ は内部イベントを意味する．プロセス内部で起こるイベントに関してはプロセス選択の決定を行わないことを推論規則は意味している．τ に関しては internal choice および hiding の operator が関わっているので後ほど説明する．

■internal choice

$$\frac{P \xrightarrow{\tau} P'}{\begin{array}{l}(P \sqcap Q) \xrightarrow{\tau} P' \\ (Q \sqcap P) \xrightarrow{\tau} P'\end{array}}$$

$P \sqcap Q$ はプロセスの internal choice を意味する．この選択はプロセスの外部によらずプロセス内部で決定される．内部イベントを意味する τ はプロセス内部で起こり，この選択が決定される．つまり，internal choice は外部からは選択不可能（非決定的）な選択を意味する．例えば，次のように記述されたプロセス $PERSON$ はその外部からはコーラかレモネードのどちらのイベントを起こすかは選択できない．

$$\begin{aligned}PERSON = coin \rightarrow (\quad & coke \rightarrow SKIP \\ & \sqcap\, lemonade \rightarrow SKIP)\end{aligned}$$

プロセス $PERSON$ 内部でイベント τ が起こり，コーラかレモネードのどちらかが決定される．これに対して，extarnal choice を用いたプロセス VMD では，その外部（他のプロセス等）が決定に関与することを意味している．

■parallel　プロセスのアルファベットとインターフェースについてまず説明する．プロセス P のアルファベットとはそのプロセスの表記に現れるイベントすべての集合であり $\alpha(P)$ で表す．例えば，

$$\alpha(VMD) = \alpha(PERSON) = \{coin, coke, lemonade\}$$

である．これに対しプロセスのインターフェイスとはプロセスの外部と関わるイベントである．並行プロセス $P_A \parallel_B Q$ はプロセス P, Q の並行動作を表す．ただし，A，B はそれぞれのプロセスのインターフェイスを表し，その共通のイベントで P と Q は同期する．$P_{\alpha(P)} \parallel_{\alpha(Q)} Q$ のときこれを $P \parallel Q$ と省略表記する．parallel の推論規則は以下のとおりである．

$$\frac{P \xrightarrow{\mu} P'}{\begin{array}{l}P \;{}_A\!\parallel_B\; Q \xrightarrow{\mu} P' \;{}_A\!\parallel_B\; Q \\ Q \;{}_B\!\parallel_A\; P \xrightarrow{\mu} Q \;{}_B\!\parallel_A\; P'\end{array}} [\mu \in (A \cup \{\tau\} \setminus B)]$$

$$\frac{\begin{array}{l}P \xrightarrow{a} P' \\ Q \xrightarrow{a} Q'\end{array}}{P \;{}_A\!\parallel_B\; Q \xrightarrow{a} P' \;{}_A\!\parallel_B\; Q'} [a \in (A \cap B) \cup \{\surd\}]$$

前述した $PERSON$ と VMS が並行動作するプロセス $PERSON \parallel VMS$ について考えてみる．このプロセスは共通のイベントである $\{coin, coke, lemonade\}$ で同期する．両方のプロセスとも最初 $coin$ のイベントが起こせる状態にあるので，

$$(PERSON \parallel VMS) \xrightarrow{coin} (PERSON' \parallel VMS')$$

$$PERSON' = coke \to SKIP \sqcap lemonade \to SKIP$$
$$VMS' = coke \to VMS \,\square\, lemonade \to VMS$$

となる．次に VMS' は $coin$ と $lemonade$ のどちらも起こせる状態にあるが，$PERSON'$ は internal choice なのでイベント τ しか受け付けない．よって次に τ が起こり，$coke \to SKIP$ または $lemonade \to SKIP$ のどちらかが選択される．

$$(PERSON' \parallel VMS') \xrightarrow{\tau} (coke \to SKIP \parallel VMS')$$

次に $PERSON$ 側が $coke$ しか起こせないので，同期する必要がある $lemonade$ は起きない．よって $coke$ のイベントが起き，$PERSON$ 側は $SKIP$ に VMS' は最初の状態に戻る．

$$(coke \to SKIP \parallel VMS') \xrightarrow{coke} SKIP \parallel VMS$$

このプロセスの動作はここで終了となる．

また，並行プロセス P, Q の間でチャネルを用いて値をやりとりをするシステム TWO は以下ように記述する．

$$TWO = ch!10 \to P \parallel ch?x \to Q(x)$$

プロセス P から Q へチャネル ch を通じて値 10 が送られる．

$$TWO \xrightarrow{ch.10} P \parallel Q(10)$$

■interleaving

$$\frac{P \xrightarrow{\mu} P'}{\begin{array}{l} P \,|||\, Q \xrightarrow{\mu} P' \,|||\, Q \\ Q \,|||\, P \xrightarrow{\mu} Q \,|||\, P' \end{array}}[\mu \neq \surd] \qquad \frac{\begin{array}{c} P \xrightarrow{\surd} P' \\ Q \xrightarrow{\surd} Q' \end{array}}{P \,|||\, Q \xrightarrow{\surd} P' \,|||\, Q'}$$

$P \,|||\, Q$ はプロセス P，Q がイベントで同期せずに，完全に独立して動作することを表す．ただし，イベント $\surd$ に対してのみ同期して動作する．P，Q の共通イベントではどちらが実行されるかは非決定的である．

■sharing

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \,[\![\, A \,]\!]\, Q \xrightarrow{a} P' \,[\![\, A \,]\!]\, Q'}[a \in A \cup \{\surd\}] \qquad \frac{P \xrightarrow{\mu} P'}{P \,[\![\, A \,]\!]\, Q \xrightarrow{\mu} P' \,[\![\, A \,]\!]\, Q \quad Q \,[\![\, A \,]\!]\, P \xrightarrow{\mu} Q \,[\![\, A \,]\!]\, P'}[\mu \notin A \cup \{\surd\}]$$

$P \,[\![\, A \,]\!]\, Q$ は指定されたイベント集合 A のイベントおよび $\surd$ で同期し，それ以外のイベントについては interleaving と同じく別々に振舞う．$P \,[\![\, \{\} \,]\!]\, Q = P \,|||\, Q$ である．

■hiding

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}[a \in A] \qquad \frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A}[\mu \notin A]$$

$P \setminus A$ はプロセス P のインターフェイスからイベント集合 A を隠蔽する．隠蔽されたイベントは外部からは観測されず内部イベントとして起こる．例えば先ほどのプリンタのプロセスについて考える．

$$\begin{aligned} PRINTER = \quad & accept \rightarrow print \rightarrow STOP \\ & | \; shutdown \rightarrow STOP \end{aligned}$$

プリンタはイベント $accept$ が起こると，自動的にプリントアウトしそれを外部からは制御できないとする．このとき $print$ は $PRINTER$ のインターフェイスではない．$PRINTER$ の外部とのやり取りは $accept$ と $shutdown$ のみであるべきなので，そのプロセスは $PRINTER \setminus \{print\}$ と記述される．また，hiding はプロセスの抽象化に使われプロセス検証においてより重要な役割を果たす．

■sequential

$$\frac{P \xrightarrow{\mu} P'}{P \fatsemi Q \xrightarrow{\mu} P' \fatsemi Q}[\mu \neq \surd] \qquad \frac{P \xrightarrow{\surd} P'}{P \fatsemi Q \xrightarrow{\tau} Q}$$

$P \fatsemi Q$ はプロセス P, Q の逐次処理を表わす．ただし，プロセス P が イベント $\surd$ を実行できる状態になったときに内部イベント τ によりプロセス Q に実行が移る．

■interrupt

$$\frac{P \xrightarrow{\mu} P'}{P \triangle Q \xrightarrow{\mu} P' \triangle Q}[\mu \neq \surd] \qquad \frac{P \xrightarrow{\surd} P'}{P \triangle Q \xrightarrow{\surd} P'}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'} \qquad \frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}$$

$P \triangle Q$ はプロセス P の実行中に Q が割り込む可能性を表す．プロセス P 実行中にプロセス Q に関するイベントが起こると Q に実行が移る．その後 P に実行は戻らない．Q の割り込みが起こる前に P が正常終了した場合，Q は実行されない．

2.2.2 公理的意味論

これまで紹介してきた operator を用いて，ある振る舞いを持つプロセスを記述する際には多くの異なった記述の仕方が可能である．例えば，以下のプロセス $P1$ と $P2$ は明らかに同じ振る舞いを持つ．

$P1 = Q1 \,\square\, Q2$
$P2 = Q2 \,\square\, Q1$

逆に，プロセス $a \rightarrow P$ と $STOP$ は明らかに異なった振る舞いをするプロセスである．このことは操作的意味論からも分かるが，CSP の公理的意味論からも記述をされたプロセス同士が同じプロセスであるかどうかを証明することが可能である．

ここでは公理的意味論の一部のみを紹介する．prefix choice に関する法則を表 2 に示す．$x : A \rightarrow P(x)$ において，A が空集合の場合，このプロセスは $STOP$ に等しく，A がただひとつの元 b のみを持つ場合は $b \rightarrow P(b)$ に等しい．

$x : \{\} \rightarrow P(x) = STOP$	$(STOP - step)$
$x : \{b\} \rightarrow P(x) = b \rightarrow P(b)$	$(prefix)$

表 2　prefix choice の法則

external choice に関する公理を表 3 に示す．external choice に関しては，冪等であり，単位元と零元が存在し，結合律と交換律が成り立つ．また，最後の法則は A の中のイベントが起こった場合は $P1$ が選ばれ，B の中のイベントが起こった場合は $P2$ が選ばれるが，両方のプロセスが持つイベントが起こった場合，そのプロセスの選択は非決定的であることを意味する．例えば，

$(a \rightarrow P1 \parallel b \rightarrow P2) \,\square\, (b \rightarrow Q1)$

このプロセスにおいて，$P = a \rightarrow P1 \parallel b \rightarrow P2$ とし，$Q = b \rightarrow Q1$ として考えれば，イベント a が起きたとき a は P の受理できるイベントであり，Q は受理できないイベントであるので，

$$P \,\square\, Q \xrightarrow{a} (P1 \parallel b \rightarrow P2)$$

となる．イベント b が起きた場合はプロセス P も Q も受理できるので，その選択は非決定的になる．

$$P \,\square\, Q \xrightarrow{b} (a \rightarrow P1 \parallel P2) \sqcap Q1$$

$P \,\square\, P = P$		冪等 ($\square - idem$)
$P1 \,\square\, (P2 \,\square\, P3) = (P1 \,\square\, P2) \,\square\, P3$		結合律 ($\square - assoc$)
$P1 \,\square\, P2 = P2 \,\square\, P1$		交換律 ($\square - sym$)
$P \,\square\, STOP = P$		単位元 ($\square - unit$)
$P \,\square\, RUN = RUN$		零元 ($\square - zero$)
$x : A \rightarrow P1(x) \,\square\, y : B \rightarrow P2(y)$		($\square - step$)
$= z : A \cup B \rightarrow R(z)$ *where*		
$R(c) = P1(c)$	if $c \in A \setminus B$	
$= P2(c)$	if $c \in B \setminus A$	
$= P1(c) \sqcap P2(c)$	if $c \in A \cap B$	

表 3　external choice の法則

ここに載せたものはごく一部であり，CSP には多くの代数法則がある．プロセスの仕様検証においては，並行システムをモデリングしたプロセス IMP(implementation) と仕様検証のために用意したプロセス SPEC(specification) が代数変換により等しい関係にあるかどうかを検証する．

2.2.3　表示的意味論

CSP で記述したプロセスは trace（イベントの軌跡），failure（各時点で実行不可能なイベント），divergence（発散に至る軌跡）などの特性を持ち，これらをもとにプロセスの表示的モデルとしてトレースモデル，安定失敗モデル，失敗/発散モデルの 3 つがある．

■Trace　プロセス P の trace とは，P の実行において外部から観測し得るイベント（内部イベント τ 以外）の列である．P の持つすべての trace の集合を $traces(P)$ で表す．たとえば，$STOP$ は何も実行しないプロセスであるので，空列 <> がそのとり得るすべての trace となる．

$$traces(STOP) = \{<>\}$$

また，$SKIP$ はイベント $\surd$ によってのみ $STOP$ へと遷移するプロセスなので，

$$traces(SKIP) = \{<>, <\surd>\}$$

である．すべてのプロセスはその trace として <> を持ち，また，プロセスの trace 集合はプロセスを構成する operator，プロセス，イベントから決定される．例えば，prefix に関しては

$$\begin{aligned} traces(a \rightarrow P) = \quad & \{<>\} \\ & \cup \\ & \{< a > \frown tr \mid tr \in traces(P)\} \end{aligned}$$

である．ここで記号 $\frown$ は列の連結を表す．これより例えば $a \rightarrow b \rightarrow STOP$ の trace は，

$$traces(b \rightarrow STOP) = \{<>, < b >\}$$

であるから，

$$\begin{aligned} traces(a \rightarrow b \rightarrow STOP) = \quad & \{<>\} \\ & \cup \\ & \{< a > \frown tr \mid tr \in \{<>, < b >\}\} \\ = \quad & \{<>, < a >, < a, b >\} \end{aligned}$$

となる．また，external choice $P \,\square\, Q$ においては最初のイベントによって P， Q どちらかのプロセスが選択され実行されるので，その trace 集合は P の trace 集合と Q の trace 集合の和集合となる．

$$traces(P \,\square\, Q) = traces(P) \cup traces(Q)$$

また，trace は内部イベントを考慮しないので，internal choice に関してもその trace 集合は

$$traces(P \sqcap Q) = traces(P) \cup traces(Q)$$

である．これはプロセスのトレースモデルにおいて $P \,\square\, Q$ と $P \sqcap Q$ は区別されず同じプロセスであることを意味している．これらを区別できるモデルとしてトレースモデルを拡張した安定失敗モデルと発散/失敗モデルを後に導入する．

$P \parallel Q$ はプロセス P と Q の並行動作であり，P と Q の共通のイベントで同期する．よってその trace 集合は次のようになる．

$$\begin{aligned} traces(P \parallel Q) = \{tr \in TRACE \mid \quad & tr \uparrow \alpha(P) \cup \{\surd\} \in traces(P) \\ & \wedge tr \uparrow \alpha(Q) \cup \{\surd\} \in traces(Q) \\ & \wedge \sigma(tr) \subseteq \alpha(P) \cup \alpha(Q) \cup \{\surd\} \qquad \} \end{aligned}$$

ここで，TRACE はあらゆる trace の集合を表し，$tr \uparrow A$ は trace tr からイベント集合 A の要素だけを抜き出したイベント列を表す．例えば，

$$< a, b, a, c, d, e > \uparrow \{a, c, e\} =< a, a, c, e >$$

である．また $\sigma(tr)$ は tr に現れるイベントの集合である．

すべての operator に対して，このようにして構成要素のプロセスとイベントから trace が求まる．また，トレースモデルにおいてプロセス同士が等しいことを次で定義する．

$$P =_T Q \Longleftrightarrow traces(P) = traces(Q)$$

トレースモデルでは $P \,\square\, Q =_T P \sqcap Q$ である．これらのプロセスは次の安定失敗モデルにより区別される．

■Stable failures　安定状態とはあるプロセスが内部イベントを実行しない状態のことをいう．安定失敗モデルは，プロセスの安定状態における軌跡 s に対してプロセスが実行できないイベント（拒否イベント）が存在しその集合 X を追加したモデルである．プロセス P におけるその組 (s, X) が P の failure であり，P のすべての failure を $failures(P)$ で表す．例えば，

$$P = a \rightarrow (b \rightarrow STOP \sqcap c \rightarrow STOP)$$

について考えると，まだ何も実行していない状態のとき，プロセス P はイベント b，c を実行できないので，$(<>, \{b, c\}) \in failures(P)$[*2]である．

イベント a の後，P は τ によって $b \rightarrow STOP$ または $c \rightarrow STOP$ のどちらかのプロセスに遷移する．それぞれの安定状態に対応して，$(< a >, \{a, c\}), (< a >, \{a, b\})$ が failure となる．また，trace $< a, b >$ の後では $STOP$ となり，すべてのイベントを受け付けないので，$(< a, b >, \{a, b, c\}) \in failures(P)$ である．

安定失敗モデルにおけるプロセスの等価は次のように定義される．

$$P =_F Q \Longleftrightarrow traces(P) = traces(Q) \wedge failures(P) = failures(Q)$$

このモデルでは internal choice と external choice は区別され，$P \,\square\, Q \neq_F P \sqcap Q$ である．

■Failures/divergence　安定失敗モデルはプロセスの安定状態に対してその拒否集合を考えた．しかし，内部イベントを起こし続ける可能性がある状態（発散状態）においては拒否集合は使えない．例えば，

$$P = a \rightarrow D \setminus \{b\}$$
$$D = (b \rightarrow D) \,\square\, (c \rightarrow P)$$

このプロセス P において $traces(P) = \{<>, < a >, < a, c >, \cdots\}$ であるが，プロセス P はイベント a 実行後，隠蔽されたイベント b を内部イベントとして無限に起こし続ける可能性があり安定状態にならない．安定状態における拒否集合を考える安定失敗モデルは発散状態を考慮しない．

これに対して失敗/発散モデルでは，発散状態においても拒否集合 X を定義をする．プロセス P が発散状態に至る trace の集合を $divergence(P)$ で表し，$failure(s, X)$ を発散状態に至る $traces$ でも考えるよう拡張したものとして

$$failures_\perp(P) = failures(P) \cup \{(s, X) \mid s \in divergence(P)\}$$

を定義する．P はイベント a 実行後，発散状態となり何も受け付けなくなる可能性があるので，$(< a >, \{a, b, c\}) \in failures_\perp(P)$ である．プロセス P に対して $failures_\perp(P)$ と $divergence(P)$ を考えたものが失敗/発散モデルである．

$$P =_{FD} Q \Longleftrightarrow failures(P)_\perp = failures(Q)_\perp \wedge divergence(P) = divergence(Q)$$

[*2] $(s, X) \in failures(P) \wedge Y \subset X \Longrightarrow (s, Y) \in failures(P)$ であり，$(<>, \{a\}), (<>, \{b\})$ なども $failures(P)$ の要素である．

プロセスに対するこれらトレースモデル，安定失敗モデル，失敗/発散モデルの対応が CSP の表示的意味論である．

2.3 検証方法

CSP で記述したシステムは数学的な検証が可能である．検証には property oriented な方法と process oriented な方法があり，前者はプロセスの trace, failure などの性質に対する仕様をプロセスが満たしている (satisfaction) かどうかを検証する．後者は仕様もまた CSP のプロセスとして表現し，記述したプロセスと仕様プロセスの refinement 関係が成り立つかを確かめる．

2.3.1 satisfaction

プロセス P が仕様 S を満たすとき,

$$P\ sat\ S$$

と書く．仕様 S が特にプロセス P の trace に関することであるとき，P のすべての trace が S を満たすときに P は仕様 S を満たすという．

$$P\ sat\ S(tr) \Longleftrightarrow \forall\, tr \in traces(P) \cdot S(tr)$$

また，仕様として failure に関することであれば,

$$P\ sat\ S(tr, X) \Longleftrightarrow \forall (tr, X) \in failures(P) \cdot S(tr, X)$$

である．例えば，自動販売機 VM の所有者はコインを入れられた枚数以上のジュースを出したくない．そこでこの仕様は，$tr \in traces(VM)$ に対して tr に現れるイベント $juice$ の数はイベント $coin$ の数よりも少ないことを要求している．つまりこの仕様は次のように表される．

$$NOLOSS(tr) = \#(tr \uparrow \{juice\}) \leq \#(tr \uparrow \{coin\})$$

ここで $\#tr$ は tr の長さを表す．VM がこの仕様 $NOLOSS$ を満たすかどうかは次のことを証明すればよい．

$$\forall\, tr \in traces(VM) \cdot NOLOSS(tr)$$

また，購入者は自動販売機に入れたコインの枚数よりも出てきたジュースの本数が少ない場合は必ずイベント $juice$ が起こせることを望む．この仕様 $FAIR$ は次のように failure に関して書ける．

$$FAIR(tr, X) = \#(tr \uparrow \{juice\}) \leq \#(tr \uparrow \{coin\}) \Rightarrow juice \notin X$$

これに関しても，以下のこと証明すればよい．

$$\forall (tr, X) \in failures(VM) \cdot FAIR(tr, X)$$

2.3.2 refinement

CSP のプロセスの仕様は $trace, faliure$ などの性質に関して記述する以外にも，仕様としてプロセスを使うこともできる．並行システムをモデリングしたプロセス IMP に対して，その仕様もまたプロセス $SPEC$ として表現し，IMP と $SPEC$ の等価，詳細関係を見ることで様々な検証が可能である．プロセスの refinement 関係を以下で定義する．

$$P \sqsubseteq_T Q \Longleftrightarrow traces(P) \supseteq traces(Q)$$
$$P \sqsubseteq_F Q \Longleftrightarrow traces(P) \supseteq traces(Q) \wedge failures(P) \supseteq failures(Q)$$
$$P \sqsubseteq_{FD} Q \Longleftrightarrow failures_{\perp}(P) \supseteq failures_{\perp}(Q) \wedge divergence(P) \supseteq divergence(Q)$$

それぞれの refinement は，safety[*3], deadlock free, divergence free の検証や等価関係の検証に使われる．たとえば，任意のプロセス P に対して，そのアルファベットすべてについて internal choice をとった次のプロセスを考える．

$$DF = \sqcap_{a \in \alpha(P)} \ a \rightarrow DF$$

DF は内部イベント τ のあと，$\alpha(P)$ のイベントのどれか一つを起こすことを繰り返すプロセスであるが，次の refinement 関係を調べることで，並行システムの典型的なひとつの問題であるデッドロック[*4]の有無を確かめられる．

$$DF \sqsubseteq_F P$$

また，refinement と satisfaciotn の関係から例えば，

$$IMP \ sat \ NOLOSS(tr)$$

を検証したいときは，$SPEC \ sat \ NOLOSS(tr)$ となるようなプロセス $SPEC$ を考え，さらに $SPEC \sqsubseteq_T IMP$ が確認できれば，$IMP \ sat \ NOLOSS(tr)$ が言えることになる．

2.4 検証ツール

CSP でモデリングした並行システムのツールによる検証は仕様プロセスとの refinement 関係の検証によって行われる．ツールとして現在最も有名な FDR2 は実装プロセスと仕様プロセスから操作的意味論の推論規則にしたがって両方の状態遷移グラフを作り，各状態における $failures$, $divergence$ を考慮しながら走査することで refinement の検証を行っている．

これに対して，公理的意味論により検証するツールの開発も行われている [7]． refinement はまたプロセスの等価によっても表せる．これにより CSP プロセスの記述を代数変換していき $P = P \sqcap Q$ を証明することで仕様プロセスとの refinement 関係を検証することができる．

$$P \sqsubseteq_T Q \Longleftrightarrow P =_T P \sqcap Q$$
$$P \sqsubseteq_F Q \Longleftrightarrow P =_F P \sqcap Q$$
$$P \sqsubseteq_{FD} Q \Longleftrightarrow P =_{FD} P \sqcap Q$$

[*3] システムが予想外の振る舞いをしないという安全性

[*4] ここではプロセスがひとつのイベントも起こせない状態と定義する．

それぞれの特徴としては，FDR2 等のモデル検査器は完全自動検証が可能であるが，パラメータを固定する必要がありシステムによっては状態爆発が起こり事実上，検証不可能な場合がある．これに対して公理的意味論による定理証明器はパラメータを固定せずに検証することが可能であるが，証明の手続きをユーザーが指示する必要がある．

3 Timed CSP

Timed CSP は時間的概念を導入することで CSP を拡張した理論であり，時間に関する新たな operator の追加と，既存のプロセスや operator に対しては時間に関する意味論が新たに与えられている．CSP のプロセスはイベントによってのみ次の状態に遷移するのに対して，Timed CSP のプロセスは時間によってもプロセスの状態変化が起きる．このことにより時間割り込みを持つシステムや時間遅延を考慮したシステムに対しても設計，検証可能となる．プロセス $P1$ がイベント μ によってプロセス $P2$ に移るときこれを $P1 \xrightarrow{\mu} P2$ と表記したが，$P1$ が t 時間経過すると $P1'$ に移るときこれを次のように表記する．

$$P1 \overset{t}{\rightsquigarrow} P1'$$

3.1 Timed operator

Timed CSP に加えられた operator は基本的には timed event prefix と timeout および timed interrupt の 3 つであるが，timeout をもとにしてさらにいくつかの operator およびプロセスが定義されている．

■Timed event prefix

$$\frac{}{(a@u \to P) \xrightarrow{a} P[0/u]} \qquad \frac{}{(a@u \to P) \overset{d}{\rightsquigarrow} (a@u \to P[u+d/u])}$$

timed event prefix $a@u \to P$ はこのプロセスの実行開始からイベント a が実行されるまでの時間を変数 u で記録し，プロセス P のなかで u を用いることができる．$P[0/u]$，$P[u+d/u]$ はプロセス P の中で使われている変数 u をそれぞれ $0, u+d$ で置き換えることを意味している．例えばイベント $start$ が起きてから $finish$ が起きるまでの時間を計り送信するプロセスは次のように振る舞う．

$$\begin{aligned}
&(start \to finish@u \to out!u \to SKIP) \xrightarrow{start} \\
&(finish@u \to out!u \to SKIP) \overset{3}{\rightsquigarrow} \\
&(finish@u \to out!(u+3) \to SKIP) \overset{2}{\rightsquigarrow} \\
&(finish@u \to out!((u+2)+3) \to SKIP \xrightarrow{finish} \\
&(out!((0+2)+3) \to SKIP) \xrightarrow{out.5} SKIP
\end{aligned}$$

■Timeout

$$\frac{P \xrightarrow{a} P'}{P \overset{d}{\triangleright} Q \xrightarrow{a} P'} \qquad \frac{P \overset{d'}{\rightsquigarrow} P'}{(P \overset{d}{\triangleright} Q) \overset{d'}{\rightsquigarrow} (P' \overset{d-d'}{\triangleright} Q)}\,[0 < d' \leqq d]$$

$$\frac{P \xrightarrow{\tau} P'}{P \overset{d}{\triangleright} Q \xrightarrow{\tau} P' \overset{d}{\triangleright} Q} \qquad \frac{}{(P \overset{0}{\triangleright} Q) \xrightarrow{\tau} Q}$$

$P \overset{d}{\triangleright} Q$ はプロセス実行開始から時間経過と共にプロセスの状態が変化し，d 時間経過しても P のイベントが実行されない場合，内部イベント τ を起こした後プロセス Q として振舞う．timeout を使うことにより，例えば一定時間以上経つと電源が自動で切れるプリンタは以下のように記述することができる．

$$TPRINTER = (accept \rightarrow print \rightarrow STOP) \overset{10}{\triangleright} shutdown \rightarrow STOP$$

$TPRINTER$ は 10 分経っても $accept$ イベントが起きない場合はタイムアウトし停止する．

timeout により，さらに以下のようなプロセスおよび operator が定義される．

$$WAIT\ d = STOP \overset{d}{\triangleright} SKIP$$

遅延プロセス $WAIT\ d$ は d 時間経つまで何も実行しない遅延を表すプロセスである．これは $WAIT\ d \fatsemi P$ の形でよく使われ，d 時間経過後プロセス P が実行される．

$$a \xrightarrow{d} P = a \rightarrow (\ STOP \overset{d}{\triangleright} P)$$

event prefix $a \xrightarrow{d} P$ はイベント a 実行後，d 時間経過した後にプロセス P が実行される．

■Timed interrupt

$$\frac{P \xrightarrow{\mu} P'}{(P \triangle_d Q) \xrightarrow{\mu} (P' \triangle_d Q)}\,[\mu \neq \surd] \qquad \frac{P \xrightarrow{\surd} P'}{P \triangle_d Q \xrightarrow{\surd} P'}$$

$$\frac{}{P \triangle_0 Q \xrightarrow{\tau} Q} \qquad \frac{P \overset{d'}{\rightsquigarrow} P'}{(P \triangle_d Q) \overset{d'}{\rightsquigarrow} (P' \triangle_{d-d'} Q)}\,[d' \leqq d]$$

timed interrupt $P \triangle_d Q$ はプロセス P の実行中に d 時間経過しても P の実行が正常終了しない場合，プロセス Q に実行が移るという時間割り込みを表すプロセスである．Q 実行終了後に P には実行は戻らない．また，時間内に P が正常終了した場合は割り込みは起きない．

3.2 時間の扱い

既存のプロセスや operator に対しても新たに時間に対する振る舞いが定義される．$STOP$ や $SKIP$ に対する時間的な振る舞いは，時間によっては変化しないという単純なものである．

$$\frac{}{SKIP \xrightarrow{\surd} STOP} \qquad \frac{}{SKIP \overset{d}{\rightsquigarrow} SKIP}$$

また，2.2.1 節で説明した各 operator に対する時間的扱いに対しても基本的には時間経過によっては変化しないというものである．例えば，prefix に関する推論規則は以下である．

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P} \qquad \frac{}{(a \rightarrow P) \overset{d}{\rightsquigarrow} (a \rightarrow P)}$$

external choice や parallel,sharing,interleaving などに対しても追加される規則は単純なものであり，プロセス P，Q が d 時間経過後にそれぞれ P'，Q' に遷移するとき $P \,\Box\, Q$ は $P' \,\Box\, Q'$ へと移る．

$$\frac{P \overset{d}{\rightsquigarrow} P' \qquad Q \overset{d}{\rightsquigarrow} Q'}{P \,\Box\, Q \overset{d}{\rightsquigarrow} P' \,\Box\, Q'}$$

$$\frac{P \overset{d}{\rightsquigarrow} P' \qquad Q \overset{d}{\rightsquigarrow} Q'}{P \;{}_A\|_B\; Q \overset{d}{\rightsquigarrow} P' \;{}_A\|_B\; Q'}$$

イベントはプロセス外部（他のプロセス）との相互作用や通信であるので，イベントがいつ起こるかは外部環境による．イベントが起こるまではプロセスは時間遷移する．しかしながら，内部イベント τ に関しては外部に関係なく実行可能になった瞬間に起こるものとして考える．internal choice に関しては external choice に追加したような時間に関する推論規則はない．なぜなら，$P \sqcap Q$ は時間遷移することなく即座にイベント τ によって次の状態へと移るからである．同じく隠蔽に関しても追加される推論規則は以下のようなものである．

$$\frac{\begin{array}{l} P \overset{d}{\rightsquigarrow} P' \\ \forall\, a \in A \bullet \neg(P \xrightarrow{a}) \end{array}}{P \setminus A \overset{d}{\rightsquigarrow} P' \setminus A}$$

$P \xrightarrow{a}$ は P が a によって遷移可能であること，つまりイベント a に対する実行準備ができているときに真を返す．よってこの規則は，隠蔽されているすべてのイベントが実行できない状態のときに限って $P \setminus A$ は時間遷移することを表している．隠蔽されているイベントがひとつでも実行

可能なときは，直ちにそれが内部イベントとして実行される．例えば，再度 TWO プロセスを考える．

$$TWO = ch!10 \rightarrow P \parallel ch?x \rightarrow Q(x)$$

このプロセス P，Q はチャネル ch で１：１に繋がっているとする．チャネルでのやり取りについて外部のプロセスは関与しないので TWO の中に隠蔽されるイベントである．このときプロセス $TWO \setminus \{|\ ch\ |\}$[*5]は構成している２つのプロセスがそれぞれ送信，受信可能な状態になったら，外部の環境によらず即座にその通信は行われる．

$P \,^{9}_{9}\, Q$ に関しては，P の実行が終了後は内部イベントを起こし Q の実行に移っていく．これは瞬間的に起こるので時間に関する推論規則は次のようになる．

$$\frac{P \overset{d}{\rightsquigarrow} P' \qquad \neg(P \xrightarrow{\surd})}{P \,^{9}_{9}\, Q \overset{d}{\rightsquigarrow} P' \,^{9}_{9}\, Q}$$

P がイベント $\surd$ を起こせないときに限って時間による状態変化が起こる．

3.2.1 Timed traces

CSP の trace が実行し得るイベントの列であったのに対して，Timed CSP では実行し得るイベントとそれが起こった時間の組の列を timed trace として考える．例えば，次のような Timed CSP で記述した $HELEN$ と $CARL$ というプロセスを考える．

$$HELEN = (\ meet \xrightarrow{60} work \rightarrow SKIP) \overset{30}{\triangleright} work \rightarrow SKIP$$
$$CARL(d) = WAITd \,^{9}_{9}\, (\ meet \xrightarrow{60} home \rightarrow SKIP) \overset{45}{\triangleright} home \rightarrow SKIP$$

$HELEN$ と $CARL$ は会う約束をしているが，$CARL$ は d 分間遅れてしまう．$HELEN$ は会う($meet$) 準備ができており，30 分経つまでは $CARL$ を待っている．$CARL$ が現れずに 30 分経つと，あきらめて仕事 ($work$) に戻る．もし会えた場合は，それから 60 分間経つまでは仕事に戻らない．$CARL$ は最初の d 分は会う準備ができておらず $meet$ イベントを起こせない．d 分経った後にようやく $meet$ を起こすことができる．$HELEN \parallel CARL(d)$ は共通のイベント $meet$ で同期するが，このプロセスが $meet$ を起こせるかどうかは d による．

d = 15 の場合は，15 分経ったときに会うことができ，その 60 分後にそれぞれ仕事と家にもどることになる．その timed trace は例えば次のようになる．

$$< (15, meet), (75, work), (75, home) >$$

[*5] $\{|\ ch\ |\}$ でチャネル ch に関するすべてのイベント集合を表わす

d = 30 の場合は，30 分経ったとき HELEN は仕事へと戻るので 2 人は会えない．CARL はそこからさらに 45 分待つので，その timed trace は

$< (30, work), (75, home) >$

である．$CARL$ が 15 分遅れるか 30 分遅れるか分からないようなプロセスは次のようにその選択が internal choice（非決定的）になる．

$HELEN \parallel (CARL(15) \sqcap CARL(30))$

また，failure に対しても同様に，拒否集合を時間とイベントの組で表すことによって拡張した timed failures が定義されている．プロセス P の timed failures を $TF(P)$ で表す．

3.3 モデリングと検証

3.3.1 最小遅延，最大遅延

timed prefix を用いたプロセスは一般にイベントの最小遅延を表す．例えば，次のプロセスではイベント a が起きた後，少なくとも ϵ 時間はイベント b が起きることはない．

$C1 = a \xrightarrow{\epsilon} (b \rightarrow SKIP)$

これに対して，最大遅延は次のように表すことができる．

$C2 = a \rightarrow ((b \rightarrow SKIP) \overset{\delta}{\triangleright} STOP)$

このプロセスはイベント a が起きた後，timeout プロセスに移り，もしイベント b が起きるならそれは δ 時間以内であることになる．実際にイベント b が起きるかどうかは外部環境による．

3.3.2 Occam

CSP に時間的概念を入れた Timed CSP を用いることによりプロセスの時間的振る舞いを記述することができる．例えば，CSP の実装用の言語として Occam が開発されたが，Occam にはリアルタイムシステムに対応するべくタイマー機能が用意されている．タイマーは一定の時間（例えば 1/100 秒ごと）にその値が更新され，その値をチャネル入力と同じようにしてプログラムの中で取得することができる．例えば，

```
TIEMR timer:
```

によって timer をタイマーとして宣言したあと，次の記述で現在のタイマーの値を変数 now に取得できる．

```
timer? now
```

このタイマー機能によって次のように記述することでプログラムの実行遅延を発生させることができる．

```
timer? now
timer? AFTER (now + 遅延時間 d)
```

timer? AFTER (now + **遅延時間** d) の記述によりタイマーの値が now+ **遅延時間** d を超えるまでプログラムはアイドリングしつづける．また，extarnal choice を意味する Occam のコンストラクター ALT の中で用いることでタイムアウトが実現できる．

```
timer? now
ALT
  ch? x
    -- P default process
  timer? AFTER (now + 制限時間 t)
    -- Q timeout process
```

このプログラムはまず，現在のタイマーの値を変数 now に取得したあと，ALT 命令に移る．ALT ではチャネル ch からの入力を待ち入力後は P が実行されるが，入力がこないまま制限時間が過ぎるとタイムアウトし Q が実行される．

このような記述ができることはリアルタイムシステムにおいて有効であるが CSP では遅延を考慮できず，またタイムアウトに関しても次のように定義されており時間を考慮していない．

$$P1 \triangleright P2 = (P1 \,\Box\, P2) \sqcap P2$$

よって，これらタイマー機能等を使い時間を考慮して振る舞うシステムに対しては CSP では十分にとれえきれず，正確な検証が行えない場合がある．このようなシステムに対しては時間的振る舞いが記述できる Timed CSP が必要である．Timed CSP なら遅延プロセス $WAIT$ や，上記のタイムアウトプロセスは次のように正確にモデリングできる．

$$(ch?x \rightarrow P) \overset{t}{\triangleright} Q$$

3.3.3 timewise refinement

Timed CSP の検証においては untimed なプロセスと timed なプロセスの refinement 関係が重要である．なぜなら，時間的振る舞いを考慮し Timed CSP で記述されたプロセスであっても，その満たすべき仕様は必ずしも時間的振る舞いに関することであるとは限らないからである．例えば，並行システムにおける安全性やデッドロックの検証のための仕様プロセスはいずれも untimed なプロセスで表される．Timed CSP には以下の 3 つの timewise refinement と呼ばれる CSP プロセスと Timed CSP プロセスの refinement 関係が定義されている．

$$SPEC_{\ T} \sqsubseteq_{TF} IMP$$
$$SPEC_{\ F} \sqsubseteq_{TF} IMP$$
$$SPEC_{\ FD} \sqsubseteq_{TF} IMP$$

CSP の refinement と同様，それぞれ時間的振る舞いを持つ IMP に関する，safety, deadlock free, divergence free の検証に主に使われる．

4　Timed CSP Explorer

並行システムの設計においては CSP のような形式的な方法が必要不可欠である．また，リアルタイム性を考慮した検証を行うには時間の概念の入った Timed CSP のような理論を用いることになる．CSP の検証用ツールとしては FDR2 が有名であるが Timed CSP には現在対応しておらず，Timed CSP に対応したツールはまだ出始めたばかりである．そこで本研究では新たに Timed CSP 検証用ツールを作るための技法の提案およびプロトタイプ（Timed CSP Explorer）の製作を行った．

4.1　Timed CSP Explorer 概要

Timed CSP 検証用ツールの作成には関数型言語 ML(Standard-ML/New Jersey)[8] を用いた．その理由としては CSP が数学的な理論であり，c や java などの手続き型言語よりも関数自身が関数の引数や戻り値にできる言語の方が数学的な記述をサポートしやすいためであり，このことにより CSP プロセスを ML の関数としてより簡潔に定義できるためである．また，ML は関数型言語の中でも実行処理が速く，初期の FDR も ML でつくられたという実績がある．

CSP のプロセスはあるイベントを実行後に次のプロセスへと遷移する．関数型言語においてこのプロセスの振る舞いは関数として自然に実装できる．つまり，あるプロセスに対応する関数として，その関数はイベントを引数にとり，実行可能なイベントであれば遷移後のプロセス（関数）を返すのである．

Timed CSP Explorer は主に，Timed CSP の記述から対応する ML のコードを生成する部分と，その ML のコードを解釈し検証する部分から成る（図 1 参照）．生成部にはコンパイラの解析技術を利用する．Timed CSP 記述を読み込み，CSP_M の文法にしたがってプロセスの構造を解析し，そこから対応する ML コードを出力する．検証部では各 operator を操作的意味論に基づいて実装しておき，生成部が出力したコードを解釈しながらモデル検査を行う．

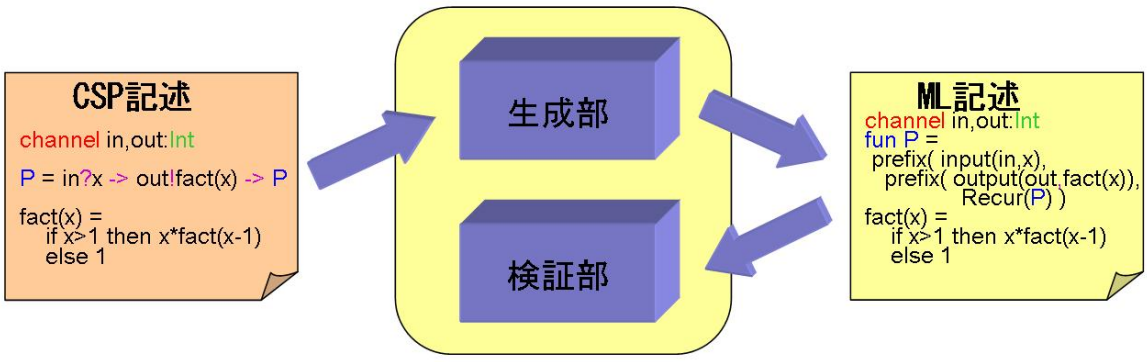

図 1　Timed CSP Explorer 概要図

4.2 関数型言語 ML

ここではこれからの説明に必要になる ML の記述について説明をしておく．ML では次のようにして新しい型を定義することができる．

```
datatype 型の名前 =
    データ構成子 1 of 型式 1
|   データ構成子 2 of 型式 2
          .....
|   データ構成子 n of 型式 n
```

例えば，

```
datatype fruit = Apple | Orange | Grape
```

これは Apple, Orange, Grape を要素として持つ型として fruit 型を定義している．また，

```
datatype number = Int of int | Real of real
```

は number 型として，整数 (int) と実数 (real) の和集合を定義しており， Int や Real はそれぞれ int 型，real 型から number 型を作り出すデータ構成子である．つまり int 型の 3 に対して Int を作用させた Int 3 や real 型の 3.1 に Real を作用させた Real 3.1 などが number 型の要素となる．

number 型を引数にとり number 型を返す関数で，引数が Int なら 2 乗を，Real なら 2 倍するような関数 calc は次のように定義する．

```
fun calc(Int x) = Int (x*x)
|   calc(Real x) = Real (x+x)
```

または，

```
fun calc(x:number) =
     case x
        of Int y => Int (y*y)
         | Real y => Real y+y
```

定数や関数の局所定義については以下のように記述する．

```
let
  局所定義 1
 :
  局所定義 n
in
  式
end
```

let と in の間で定義された定数，関数等は in と end の間にある式においてのみ適用される．例えば，半径を引数に取り円の面積を返す関数は次のように定義できる．fun は関数の宣言，val は定数

等の宣言である．また，関数型言語の多くは型推論の機能をもち，ML もまた型推論を持つ言語であるので，この関数の引数と戻り値は real 型の定数 pi との演算から real 型と推論され定義される．

```
fun circum(radius) =
   let
      val pi = 3.14
   in
      radius*raidus*pi
end
```

4.3 プロセス実装方法

CSP のプロセスを ML で実装する際の要点を以下に述べる．

4.3.1 プロセス型の定義

CSP のプロセスは受理したイベントによって次のプロセスに移る．つまりプロセスは，イベントを引数にとり遷移後のプロセスを返す関数として定義すればよい．ML では新しい型の定義ができ，さらにその要素として関数をとることができるので CSP プロセスは以下のように定義される．

```
datatype process
   = Stop
   | Skip
   | Proc of (event -> process)
   | Bleep
```

この記述は process という型は，Stop,Skip,Bleep または Proc(event を引数に取り process を返す関数) であるということを意味する．事実，CSP のプロセスはプリミティブなものとして $STOP$ と $SKIP$ があり，そこから構成されるプロセスはすべてイベントにより次のプロセスに遷移するものである．Bleep は関数が実行可能でないイベントを受け取ったときにエラーとして返すプロセスとして使用する．

次にイベントの型について考えると，イベントにはこれまで見てきたように $coin$，$juice$ などのような相互作用と，in.3，out.10 のようなチャネルによる通信がある．$coin$，$juice$ などは文字列 string 型として扱うのが自然であり，チャネル名に関しても string 型で扱う．チャネルによって送受信されるデータの型は整数，配列，組などシステムに応じて定義されるものである．そこで，$coin$，$juice$ などの相互作用はデータのやり取りのないチャネルとして考えることで相互作用と通信を別々に分けて考える必要がなくなる．

```
datatype event
   = Event of string*chanType
```

Event は string と chanType の組を event 型とする構成子であり，string は相互作用名またはチャネル名，chanType はチャネルで送受信するデータの型である．chanType もまた新しい型として必要に応じて定義する．

```
datatype chanType
   = Int of int
   | Seq of int list
   | String of string
   :
   | Tau
   | None
```

Tau は内部イベントに対して, None は相互作用に対して使用する．これにより相互作用 $coin$ は ML では Event("coin",None) として扱い，out.10 は Event("out",Int 10) として扱うことでどちらも event 型として同じように扱うことができる．また，イベント $\surd$ は Event("tick",None) とする．

4.3.2 代数 operator の実装

■event prefix　$a \rightarrow P$ はイベント a を実行後，プロセス P に移るプロセスであるので prefix は ML で次のような関数として実装できる

```
prefix(Event(ch,v),P:process) =
   let
      fun temp(Event(ch',v')) =
         if ch=ch' andalso v=v' then P
         else Bleep
   in
      Proc temp
   end
```

prefix は event 型の Event(ch,v) と process 型の P を引数にとり process 型の Proc temp を返す関数である．ここで temp は Event(ch',v') を引数に取り，もし ch が ch' と等しくかつ v と v' が等しければ P を返す．そうでなければ Bleep を返す関数である．Proc は，event -> process (event 型を引数にとり process 型を返す関数) から process 型をつくるデータ構成子であり，Proc temp は process 型となる．

関数 prefix をこのように定義することで，例えば $a \rightarrow SKIP$ はイベント a 実行後，$SKIP$ になるプロセスであるが，これは ML で次のように実現できる．

```
prefix( Event("a",None), Skip )
```

関数 prefix の戻り値である Proc temp は process 型であるから，$P = b \rightarrow (a \rightarrow SKIP)$ は次のようにできる．

```
val P = prefix( Event("b",None), prefix(Event("a",None),Skip) )
```

関数 prefix の 2 番目の引数に prefix(Event("a",None),Skip) を与えている．このプロセス P の実行には例えば，次のような関数 run を用意しておく．

```
fun run(Proc temp) = temp
```

関数 run は process 型の値からその実行部分 (temp 関数) を取り出しており，これにより，

```
val P1 = run( P )
```

とすると，P1 は prefix の実行部分であり，P1(Event("b",None)) は P がイベント b を実行した次の状態である prefix(Event("a",None),Skip) を返す．さらに

```
val P2 = run( P1(Event("b",None)) )
val P3 = run( P2(Event("a",None)) )
```

とすると，P3 には P がイベント a,b を実行した後の Skip が返ってきている．

■output　$ch!v \rightarrow P$ はイベント $c.v$ を起こした後，プロセス P として振る舞うプロセスであるので，その ML での実装は prefix と等しい．つまり，$ch!10 \rightarrow SKIP$ は，

```
prefix( Event("ch",Int 10), Skip )
```

である．

■input　$ch?x \rightarrow P(x)$ はイベント $ch.v$ の後にプロセス $P(v)$ として振る舞うプロセスである．output との違いはイベント $ch.v$ の v の値を変数 x で保持する必要があることである．変数の保持は記号表と呼ぶ連想リストによって行っており，プロセス P のなかで x が使われる場合はこの記号表から x の値を参照する．P before varTable := VarTable.acons(x, v, varTable) の記述で P を返す前に，記号表 varTable の更新（変数 x とその値 v の登録）を VarTable.acons 関数により行っている．

```
fun chanIn(Event(ch,x), P:process ) =
    let
        fun temp(Event(ch', v)) =
            if ch=ch' then
                P before varTable:=VarTable.acons(x, v, varTable)
            else
                Bleep
    in
        Proc temp
    end
```

■external choice　$P \square Q$ は最初に起こったイベントによりプロセス P，Q が選択されるプロセスである．ただし，P および Q に共通なイベントに関してはその選択は非決定的になる．

```
external(Proc P, Proc Q) =
   let
      fun temp(e:event) =
         if P(e)=Bleep then Q(e)
         else if Q(e)=Bleep then P(e)
         else internal(P(e),Q(e))
   in
      Proc temp
   end
```

イベント e がプロセス P に関するイベントでなければプロセス Q が e を実行した後のプロセスを返す．e が P に関するイベントであり，Q に関するイベントでなければ P が e を実行した後のプロセスを返す．イベント e がプロセス P も Q も実行可能な場合はこの選択は internal choice となる．

■internal choice　$P \sqcap Q$ はプロセス外部によらず，内部イベント τ によってその選択が行われる．これには L，R というイベントによってその選択を行うように実装した．L，R の chanType は Tau を用いる．temp は L，R 以外の event 型の引数に対しては Bleep を返す．

```
internal(Proc P, Proc Q) =
   let
      fun temp( Event("L",Tau) ) = Proc P
      |   temp( Event("R",Tau) ) = Proc Q
      |   temp( x:event ) = Bleep
   in
      Proc temp
   end
```

■parallel　$P \ {}_A\|_B \ Q$ は A と B の共通のイベントで同期し，それ以外のイベントでは関与しているプロセスのみが遷移する．その推論規則から parallel は次のように実装できる．A,B は event 型のリストであり，isMember(a,A) は a が A の要素であるときに真を返す．

```
fun parallel( Proc P,A,B,Proc Q ) =
   let
      fun temp(e:event) =
         if isMember(e, A) andalso isMember(e, B) then
            parallel( P(e),A,B,Q(e) )
         else if isMember(e, A) then
            parallel( P(e),A,B,Proc Q )
         else if isMember(e, B) then
            parallel( Proc P,A,B,Q(e) )
         else
            Bleep
   in
       Proc temp
   end
```

4.3.3 再帰プロセス

CSP の再帰プロセスは記述の中に自身のプロセスを記述することで定義される．この実装には再帰関数を使って定義する．しかし例えば,

$$COUNTER(n) = \quad up \rightarrow COUNTER(n+1)$$
$$\Box\ down \rightarrow COUNTER(n-1)$$

のような引数を持った再帰プロセスは以下のように記述すればよさそうであるが，SML/NJ (Standard-ML/New Jersey) は遅延評価[*6]を持つ関数型言語ではない．つまり，関数の引数に式がある場合，関数に与えられた時点で式の評価が行われる．

```
fun COUNTER(n) = external( prefix(up,COUNTER(n+1)),
                           prefix(down,COUNTER(n-1)) )
```

この記述ではまず prefix(up,COUNTER(n+1)) の COUNTER(n+1) 部分を評価しようと，自身を呼びだしてその中では再び同じことが起こり，延々と再帰し続けてしまうことになるので関数 COUNTER はうまく実行できないという問題が生ずる．遅延評価を持つ言語であれば prefix のなかで COUNTER(n+1) が使われるまでその評価は遅延される．

評価遅延を持たない ML でこれを回避するためにまず，process の定義を次のように拡張する必要がある．

```
datatype process
   = Stop
   | Skip
   | Proc of (event -> process)
   | Para of (int -> process) * int
   | Bleep
```

process の構成子として Para を追加する．Para は int を引数にとり process を返す関数と int の組を process として定義する構成子である．これにより,

```
fun COUNTER(n) = external( prefix(up,Para(COUNTER,n+1)),
                           prefix(down,Para(COUNTER,n-1)) )
```

と記述すると，Para(COUNTER,n+1) は単なる関数定義と int の組で COUNTER は評価されない．そして，prefix(up,Para(COUNTER,n+1) の戻り値として Para(COUNTER,n+1) を受け取ったときに COUNTER(n+1) として評価するようにすることでプロセス COUNTER の定義がうまくできる．また，Para of (int -> process) * int の int 部分を chanType に変更すれば，int 型以外の引数をもつプロセスも定義可能となる．

[*6] 遅延評価により繰り返し構造を自然に組み込むことができ無限を簡潔に取り扱える．

4.3.4 変数のスコープ

CSP 記述内で使われる変数のスコープは主に逐次的な部分がその範囲となる．例えば,

$$in?x \rightarrow (\ a \rightarrow out!x * x \rightarrow SKIP$$
$$\Box\ b \rightarrow out!x + x \rightarrow SKIP\ )$$

上記の記述において x はチャネル in からの入力の値となりその後に extarnal choice のプロセスから参照される．しかしながら，次のような並行プロセスの記述では x は各プロセスで異なった値を持つ.

$$(in1?x \rightarrow out1!x + x \rightarrow SKIP)$$
$$\| (in2?x \rightarrow out2!x * x \rightarrow SKIP)$$

x はそれぞれの並行プロセスで別々の値を持ち，かつこれらは同時に存在することになる．これらの変数の値の保持には並行プロセスごとに別々の記号表を用意する必要がある.

4.4 時間概念の導入

ここまでは untimed な CSP に関することを述べてきた．ここからは Timed CSP に対応するために必要な Timed operator の組み込み，さらに時間に関して今までの記述を拡張していく.

4.4.1 event の拡張

Timed CSP のプロセスはイベントによってだけでなく，時間経過によってもプロセスの状態を変えていく.

$$P \xrightarrow{\mu} Q \qquad P \overset{d}{\rightsquigarrow} P'$$

これはプロセスの時間的変化についてもイベントによる変化と同じく，プロセスに時間を与えた時にプロセスが次の状態に移ると考えてよい．そこで，先に定義した event に時間経過を加え拡張することで時間を取り扱えるようにした.

```
datatype event
   = Event of string*chanType
   | Time of int
```

4.4.2 operator の拡張

■event prefix　$a \rightarrow P$ は時間によって変化しないことが意味論の推論規則からわかる．よってその拡張は単に Time d を受けとった場合は，同じプロセス $a \rightarrow P$ を返す関数となる．

```
fun prefix(Event(ch,v), P:process) =
   let
      fun temp(Event(ch',v')) =
          if ch'=ch andalso v=v' then P
          else Bleep
      | temp(Time d) =
          prefix(Event(ch,v), P)
   in
      Proc temp
   end
```

■external choice　$P \ \square \ Q$ はその推論規則から時間に対しては，$P \xrightarrow{d} P'$，$Q \xrightarrow{d} Q'$ のとき $P \ \square \ Q \xrightarrow{d} P' \ \square \ Q'$ と変化する．よって Time d が与えられた時は P，Q それぞれに Time d を与えたものの external choice となる．

```
fun external(Proc P, Proc Q) =
   let
      fun temp(Time d) =
         external(P(Time d),Q(Time d))
      | temp(e:event) =
         if P(e)=Bleep then Q(e)
         else if Q(e)=Bleep then P(e)
         else internal(P(e),Q(e))
   in
      Proc temp
   end
```

4.4.3 timed operator の実装

■timed event prefix　$a@u \rightarrow P$ はプロセスの実行開始からイベント a が実行されるまでの時間を変数 u で記録し，プロセス P のなかで u を用いることができる．この実装には $input$ と同じく記号表を用いて u の値を保持する．このプロセスの実装は Time d を受け取った場合は記号表の値を更新した後に自身に戻る．VarTable.acons 関数は引数が Time d の場合は u の値にそれを加算するよう拡張しておく．

```
fun teprefix(Event(ch,v), u, P ) =
   let
      fun temp(Time d) =
         teprefix(Event(ch,v), u, P)
             before varTable:=VarTable.acons(u, Time d, varTable)
      | temp(Event(ch', v')) =
         if ch=ch' andalso v=v' then
             P
         else
             Bleep
   in
      Proc temp
   end
```

■timeout　$P \overset{d}{\triangleright} Q$ はプロセス実行開始から d 時間経つまでに P に関するイベントが発生しない場合，実行が Q に移る．d' 時間経過後のプロセスは $P \overset{d-d'}{\triangleright} Q$ に時間遷移する．また，内部イベントに関してはタイムアウトは解除されない．

```
fun timeout(Proc P, Time d, Proc Q) =
   let
      fun temp(Time d') =
            if d-d'>0 then
               timeout(P(Time d'),Time(d-d'),Proc Q)
            else
               Proc Q
      |   temp(Event(x,Tau)) =
            timeout(P(Event(x,Tau)),Time d,Proc Q)
      |   temp(e:event) =
            P(e)
   in
      Proc temp
   end
```

また，timed prefiix に関しては $a \xrightarrow{d} P = a \rightarrow (\ STOP \overset{d}{\triangleright} P)$ であったから，そのまま次のように実装できる．

```
fun tprefix(e:event, Time d, P:process) =
      prefix( e:event, timeout(Stop,Time d,P) )
```

■timed interrupt　$P \triangle_d Q$ はプロセス実行開始から d 時間経つまでに P が正常終了しない場合，実行が Q に移り，正常終了する場合は時間割り込みは起こらない．d' 時間経過後のプロセスは $P \triangle_{d-d'} Q$ に時間遷移する．

```
fun tinterrupt(Proc P,Time d,Proc Q) =
   let
      fun temp(Time d') =
            if d-d'>0 then
               tinterrupt(P(Time d'),Time(d-d'),Proc Q)
            else
               Proc Q
      |   temp( Event("tick",None) ) =
            P( Event("tick",None) )
      |   temp(e:event) =
            if P(e)=Bleep then Bleep
            else tinterrupt(P(e),Time d,Proc Q)
   in
      Proc temp
   end
```

4.5 Timed CSP から ML への変換

これまでは CSP の記述の意味をいかに ML により実装するかを説明してきた．CSP の自動検証を行うためにはその記述を読み込み，対応する ML コードを出力する生成部が必要である．これにはコンパイラ開発の字句解析と構文解析の技術が使える．

また，CSP に対しては，そのマシンリーダブルな記述である CSP_M が普及しているが，Timed CSP に対するマシンリーダブルな一般的な記述は特にない．そこで，Timed operater には以下のような記述を使うことにした．

Timed CSP 記述	マシンリーダブル記述
$a@u \rightarrow P$	`a@u -> P`
$P \overset{d}{\triangleright} Q$	`P [>#d Q`
$a \overset{d}{\longrightarrow} P$	`a ->#d P`
$P \triangle_d Q$	`P /\#d Q`

例えば，

$$a \overset{3}{\longrightarrow} (P \overset{5}{\triangleright} Q)$$

に対応するマシンリーダブルな Timed CSP 記述は次のようになる．

```
a ->#3 ( P [>#5 Q )
```

この Timed operator のマシンリーダブルな形は他にも色々と考えられたが，untimed な記述 ＋ ♯時間 という形が最も読みやすいだろうと判断した．この記述の変更は字句解析部を僅かに変更するだけで対応できる．

4.5.1 字句解析

字句解析ではまず，Timed CSP の記述を意味のある最小の単位（トークンと呼ぶ）に分割する．Timed CSP の記述を一文字ずつチェックしていき，その並びから表 4 のようなトークンの列として次の構文解析部へと渡している．Id(str) は英字から始まる英数字の並び，Dig(str) は数字の並び，Timed(str) は ♯ から始まる数字の並びをひとつのトークンとしている．例えば，

```
VM = coin -> (juice ->♯3 SKIP)
```

この記述に対しては次のようなトークン列を構文解析へと渡している．

```
Id("VM") Equal Id("coin") Arrow Lparen Id("juice") Arrow
Time("3") Id("SKIP") Rparen
```

また，字句解析部で空白，タブ，改行コードは取り除く．

関係演算子	Eq ”== ” Ne ”!= ” Lt ”< ” Le ”>= ” Gt ”> ” Ge ”>= ”
operator	Ndet ”| ˜	” Intl ”||| ” Box ”[] ” Intr ”/\ ” Timeout ”[> ” Semi ”; ” Arrow ”− > ” Par ”|| ” Lshare ”[| ” Rshare ”|] ” Lsquare ”[” Rsquare ”] ” Backslash ”\ ”
送受信	Query ”? ” Pling ”! ”
２項演算子	Minus ”- ” Plus ”+ ” Times ”* ” Mod ”% ” Slash ”/ ” Cat ”ˆ ”
括弧	Lparen ”(” Rparen ”) ” Lbrace ”{ ” Rbrace ”} ”
予約語	If ”if ” Then ”then ” Else ”else ” Let ”let” In ”in” Channel ”Channel ”
名前，数字	Id(str) ”名前” Dig(str) ”数字”
その他	Equal ”= ” Comma ”, ” Vertical ”| ” Dot ”. ” Dotdot ”.. ” Colon ”: ” At ”@ ”
時間	Timed(str) ”♯ 時間”

表 4　CSP_M のトークン

4.5.2 構文解析

構文解析では字句解析によって分けられたトークンの並びからその構造を解釈し，ML へと翻訳していく．Timed CSP の文法は CSP_M[9] を参考に Timed operator に対応すべくプロセス定義部分を一部拡張した．

```
proc = : proc Backslash set
       | proc Intl proc
       | proc Lshare set Rshare proc
       | proc Lsquare set PAR set Rsquare proc
       | proc Ndet proc
       | proc Box proc
       | proc Timeout proc
       | proc Intr proc
       | proc Semi proc
       | Id field Arrow proc
       | STOP
       | SKIP
       -- timed operator の追加 --
       | Id field Arrow Time proc
       | Id field At Id Arrow proc
       | proc Timeout Time proc
       | proc Intr Time proc
```

構文解析では CSP 記述からその構造をまず二分木でとらえる．さらにその木をたどりながら，目的の ML コードを出力する．例えば，次の記述の場合はその二分木は図 2 のような形になる．各ノードは operator と必要なデータ（イベント集合や時間）から成り，末端のリーフはイベントまたはプロセスである．

$$TEST = out!10 \rightarrow ((a \rightarrow P \ {}_A\|_B \ b \rightarrow Q) \triangle_{t1} \ c \xrightarrow{t2} R)$$

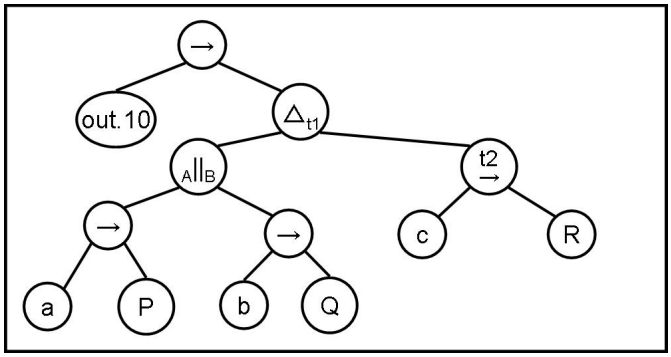

図 2　プロセス TEST の構造

二分木を行きがけ順 (根を調べてからそれにぶらさがる左右の部分木を調べる探索方法) でたどることにより CSP 記述は対応する ML 記述に変換される．つまり，図 2 では最初にルートの prefix を見て，その構成要素である左側の out.10 の後に右側の timed interrupt 側を見ていくことになる．timed interrupt のノードでも同じように左側の parallel を見終わった後に，右側の timed prefix へと移っていく．この順番で探索することで目的の ML コードを出力することができる．

```
fun test() = prefix(out.10,
                 tinterrupt(
                    parallel(
                         prefix(a,P), A, B, prefix(b,Q)
                    ),
                    Time t1,
                    tprefix(c, Time t2, R )
                 )
              )
```

4.6 プロセスの実行

構文解析により出力された ML の関数は CSP 記述の実装となっている．この関数は引数に実行可能なイベントを渡すことでプロセスの遷移に対応している次の状態関数を返してくる．つまり，前節の ML 関数 test 自体が CSP プロセス TEST に対応した図 3 のような状態遷移グラフとなっているのである．Timed operator を持つノードでは時間経過 Time によって保持している遅延の値が減っていき，0 になった時点で次の状態に遷移する．そうでないところでは時間経過 Time によっては変化しない.

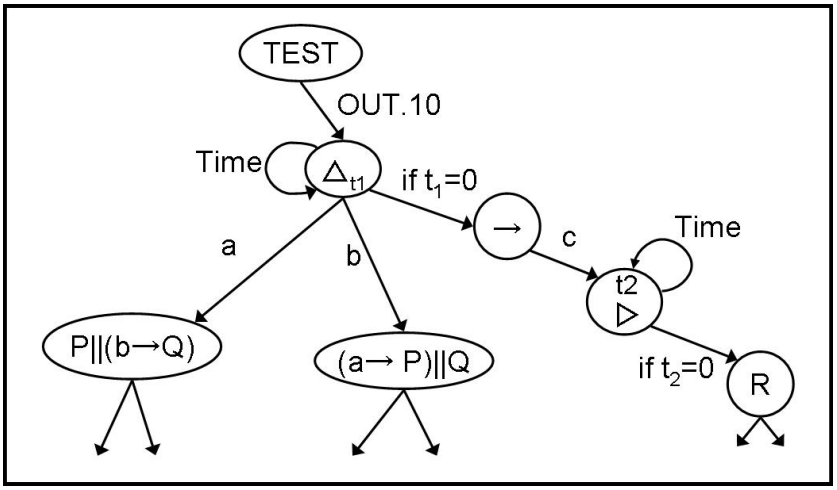

図 3　TEST の遷移グラフ

Timed CSP Explorer はプロセスの検証だけでなく，プロセスの振る舞いについてイベントを選択しながら見ていくことができる．Timed CSP Explorer は ML 上で動作する．その使い方はまず SML/NJ をコマンドプロンプトで実行し，cspexp.ml ファイルを読み込む.

```
C:\ > sml
Standard ML of New Jersey v110.70 [built: Wed Jun 17 21:06:31 2009]
- use"cspexp.ml";
```

これにより Timed CSP Explorer の動作に必要なファイル (字句解析，構文解析，operator の実装な

ど）すべてが読み込まれる．次に cspexp.ml の中で定義された main 関数に Timed CSP の記述をしたテキストファイルを与えれば字句解析，構文解析が行われ，その CSP 記述に対応した ML のコードを生成し ML 上で定義される．プロセス TEST に対応する関数 test を Timed CSP Explorer で実行する様子を図 4 に示す．ここでは $t1 = 10$，$t2 = 5$ とし，P，Q，R は $SKIP$ として実行した．関数 run は 4.3.2 節で使用したものであり，ここではさらに実行可能なイベントを表示するよう拡張している．赤い下線の箇所が実行できるイベントおよび状態変化が起こるまでの時間であり，`val Pn = run(・・・・)` 部分を入力することで状態を進めている．

P1 はプロセス TEST に対応しており，P1 が実行できるイベントは out.10 のみである．P1 に out.10 を与えたものが P2 であり，その実行できるイベントは a,b で，10time 経過するとプロセスの状態が変化する．P2 に Time 7 を与えた次の状態である P3 は，その実行できるイベントは a,b または 3time となる．P3 に Time 3 を与えると，このプロセスはタイムアウトし，次の状態に移る．P4 が実行できるイベントは c のみであり，このプロセスに時間経過を与えてもその状態は変わらない (P5)．P5 はイベント c 実行後，5 単位時間は何も実行できない．

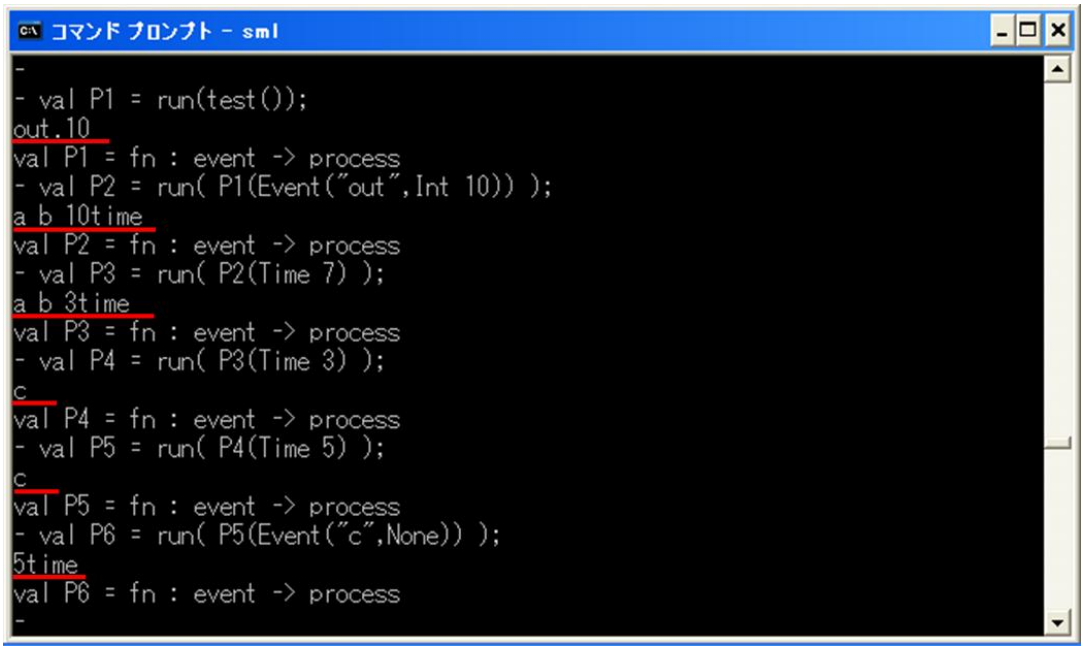

図 4　TEST の実行過程

4.7　refinement 検証方法

CSP の refinement の検証はモデリングを行った実装プロセスと仕様検証のために記述した仕様プロセスとの比較によって行われる．作成した Timed CSP Explorer は 3.3.3 節で説明した refinement のうち，trace refinement と trace timewise refinement の検証が可能である．trace refinement はプロ

セスの trace における包含関係を検証している

$$SPEC \sqsubseteq_T IMP \Longleftrightarrow traces(SPEC) \supseteq traces(IMP)$$

これには IMP 側の実行を全検査する中で，同じ実行を $SPEC$ 側が受理できるかどうかを調べればよい．また，timewise refinement は次のように定義されている．

$$SPEC \ _T \sqsubseteq_{TF} IMP \Longleftrightarrow \forall (s, X) \in TF[IMP] \cdot \sharp s < \infty \Rightarrow strip(s) \in traces(SPEC)$$

ここで $strip$ は timed trace s から trace への写像である．

$$strip(< (15, meet), (45, work), (45, home) >) =< meet, work, home >$$

つまり，IMP の timed trace から時間部分を取り去ることで得られる trace が $SPEC$ の trace になっているかどうかを検証している．これに関しては，IMP 側の実行の際に現れる内部イベントや時間経過に関しては $SPEC$ には与えずイベント (つまり $strip(s)$) のみを与え，それが実行できるかどうかを調べる．

この実行は refinement(SPEC, IMP) によって開始される．refinement 関数は受け取った IMP プロセスの状態と実行可能なイベントをスタックで保持しながらそのすべての実行を探索すると同時に，SPEC プロセスに対しても同じイベントを実行させ，それが実行可能であるかどうかを調べている．もしプロセスが実行できないイベントを与えられたときは Bleep プロセスを返すように実装している．SPEC プロセスが Bleep となったときは，その状態に至った反例 trace を表示する．SPEC が Bleep になることなく IMP のすべての振る舞いが実行できれば $SPEC \ _T \sqsubseteq_{TF} IMP$ が確かめられたことになる．

4.8 Ficher's アルゴリズムの検証

ここでは，システムの排他制御の方法のひとつであり参考文献 [10] で取り上げられている Ficher's アルゴリズムに対して Timed CSP Explorer を用いて検証することを考える．このアルゴリズムは時間的振る舞いを制御することによって正しく動作するものである．参考文献の中では Timed CSP の記述から正しく動作することを数学的に証明しているが，Timed CSP Explorer を用いることで CSP 記述から自動検証が可能である．ここではまず，untimed な CSP では検証に失敗することを検証したあと，プロセスに時間的振る舞いを加えることで正しく排他制御できることを示す．

排他制御とは複数のプロセスがある共有資源に対しアクセスするとき，一度にただひとつのプロセスのみがアクセスできるという制御である．この制御を図 5 のように共有メモリを使って実現する方法を検証する．説明を簡単にするためプロセスは 2 つとし，どちらかのプロセスのみが一度だけ排他領域に入れるものとする．それぞれのプロセスは要求があった場合に排他制御されるべき領域に入ろうとする．この領域にすでに他のプロセスが入っていないかどうかを確認するため，共有メモリには初めに 0 を書き込んでおき，0 はまだ誰も排他領域に入っていないことを意味する．

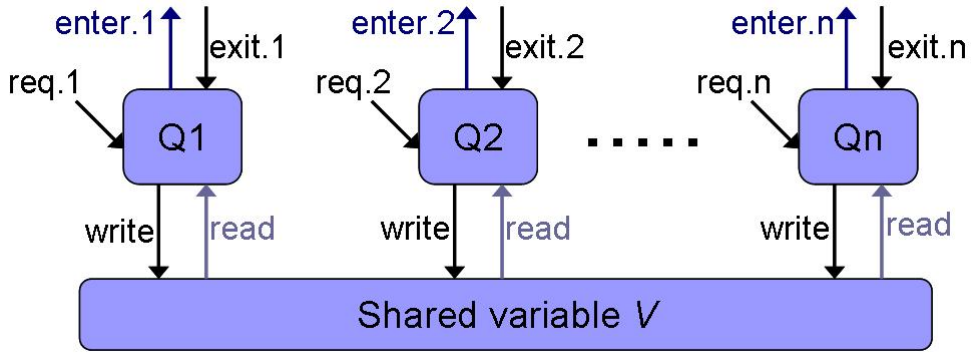

図 5　共有メモリによる排他制御

各プロセス Q_i はイベント $req.i$ が起こるとまず共有メモリの値を読み出す．もしこの値が 0 であればメモリに i を書き込んでから排他領域に入る．この動作を CSP により記述すると以下のようになる．

$$Q(i) = req.i \rightarrow read?x \rightarrow$$
$$\text{if } x \neq 0 \text{ then } SKIP$$
$$\text{else } write!i \rightarrow enter.i \rightarrow exit.i \rightarrow SKIP$$

各プロセスは互いに干渉することなく独立に動作するのでプロセス全体の動きは interleave によって記述される．

$$QS = Q(1) \,|||\, Q(2)$$

共有メモリは書き込みに対しては保持している値を変更し，読み出しに対しては保持している値を送信するプロセスと考えられるので次のように記述できる．

$$V(value) = \quad write?x \rightarrow V(x)$$
$$\Box\ read!value \rightarrow V(value)$$

V の初期値は 0 であり，プロセス QS と共有メモリ V はチャネル $read$，$write$ で同期するのでその関係は次のように記述される．

$$FIS = QS \,|[\,|\, read, write \,|\,]|\, V(0)$$

排他制御の仕様としてはただひとつのプロセスのみが $enter$ できるということなので，FIS の trace に $enter$ が 2 回以上現れないことを検証する．また，$etner$ 以外のイベントについては一切の制限を行わない．文献 [10] では次の satisfaction を調べている．

$$FIS \;\; sat \;\; \#(s \uparrow enter.\mathbb{N}) \leq 1$$

これに対して，ツールによる refinement 検証では次のような仕様プロセスを考える．

$$SPEC = (enter.1 \rightarrow exit.1 \rightarrow SKIP \,\Box\, enter.2 \rightarrow exit.2 \rightarrow SKIP) \,|||\, RUN$$

プロセス $SPEC$ は $enter.1$ または $enter.2$ についてはどちらかを一度だけ起こせる．また RUN は $entrer$ 以外のイベントをいつでも実行できるプロセス[*7]であり，それを interleave させることで $SPEC$ はそのイベントをいつでも実行可能となる．

もし，$SPEC \sqsubseteq_T FIS$ が成立すれば，プロセス FIS の振る舞いが $SPEC$ の中の振る舞いであることが分かり，FIS はあるプロセスが $enter$ を起こした後に別のプロセスが決して $enter$ を起こさないことが保証される．つまりプロセス FIS の排他制御が保証されることになる．

$SPEC$，FIS とも untimed な記述であるので，この検証は FDR2 でも可能である．FDR2 は CSP マシンリーダブルで記述されたテキストファイルを読み込み，プロセスの refinement 関係を自動検証することができる．refinement が成り立たない場合はその反例となる trace を表示する．FDR2 と Timed CSP Explorer によりこの仕様を検証すべく，プロセス FIS と $SPEC$ の refinement 関係を調べると，反例となる trace があることが図 6 または図 7 からわかる．

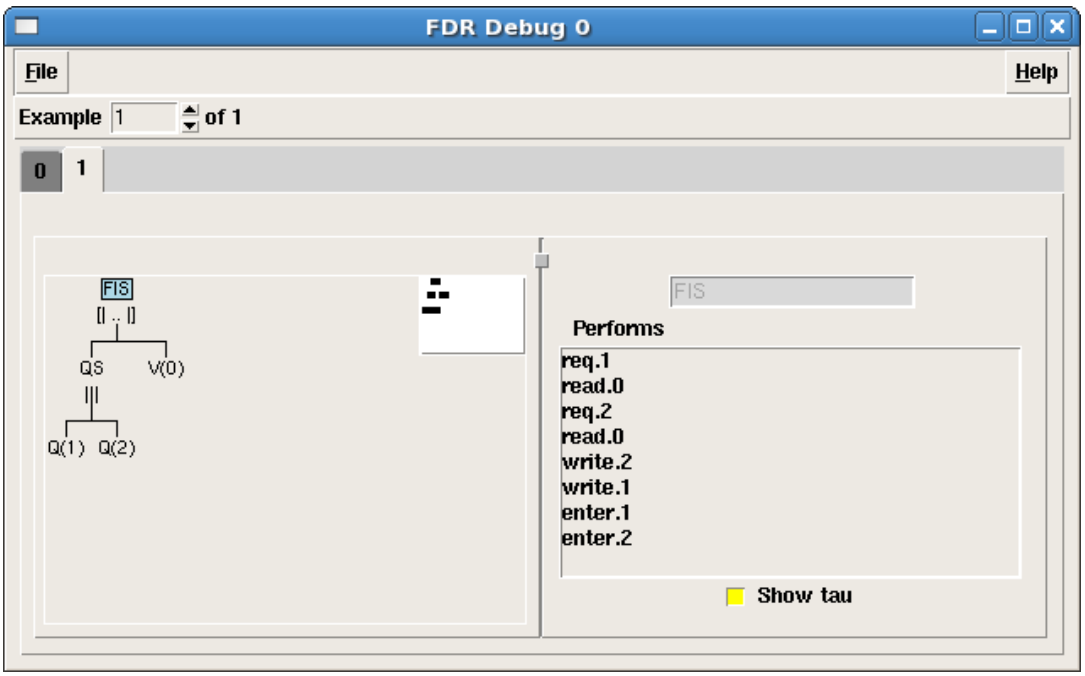

図 6　FDR2 のデバック画面

仕様を満たさない trace は一つではないので，Timed CSP Explorer と FDR2 で違う反例 trace を検出している．FDR2 側では画面の Performs に反例 trace が表示されている．図 7 は Timed CSP Explorer の検証結果であり赤線の部分が反例 trace である．L は<req.1,req.2>の後，$Q(1) \,|||\, Q(2)$ の両プロセスとも $read.0$ を起こせるので，internal choice で左の $Q(1)$ が選ばれたことを意味する．

[*7] 例えば，$RUN = (req?x \rightarrow RUN) \,\Box\, (read?x \rightarrow RUN) \,\Box\, (write?x \rightarrow RUN)$

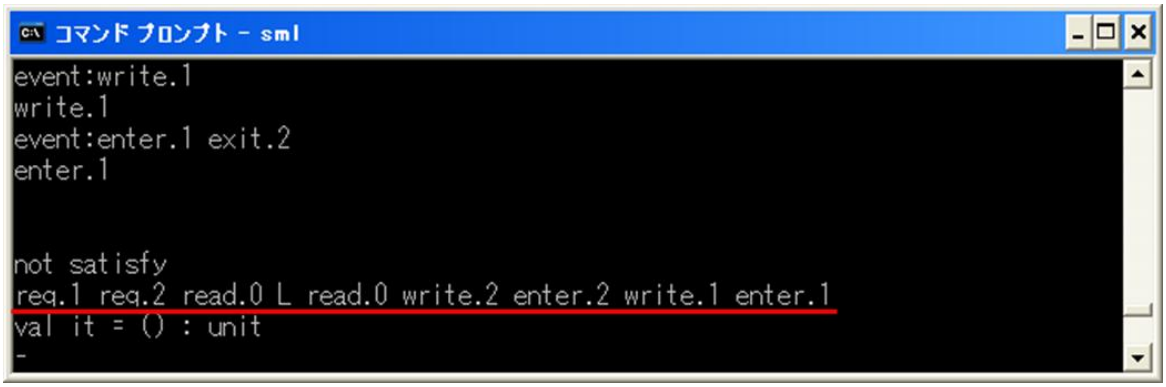

図 7　Timed CSP Explorer を使った場合の検証結果

2 つのプロセスに対して要求がほぼ同じタイミングで来たとき，一方のプロセスが共有メモリの値を読み込み 0 であることを判定している間にもう一方のプロセスもメモリの値を読み込むと，両プロセスとも 0 判定後に id をメモリに書き込み排他領域に入ってしまうことが起こり得る．つまり排他制御に失敗する．

そこで，これを回避するためプロセスに時間的振る舞いを加える．プロセスは共有メモリの値がもし 0 であれば自身の id を書き込む．そのあと ϵ 時間待ってから再びメモリの値を読み込み，もしその値が自身の id であれば排他領域に入っていく．もし他のプロセスが同じタイミングで値を読み込んでいれば，一定時間 ϵ 待つことで再び自身がメモリの値を読み込んだときに，別のプロセスがメモリの値を書き換えているので自身は $enter$ しない．ϵ はプロセスがメモリの値を読み込んでから自身の id を書き込むまでの最大遅延 δ[*8] より長く設定する必要がある．

$$
\begin{aligned}
Q(i) = {} & req.i \rightarrow read?x \rightarrow \\
& \text{if } x \neq 0 \text{ then } SKIP \\
& \text{else } (write!i \xrightarrow{\epsilon} read?y \rightarrow \\
& \qquad \text{if } y \neq i \text{ then } SKIP \\
& \qquad \text{else } enter.i \rightarrow exit.i \rightarrow STOP \;) \overset{\delta}{\triangleright} STOP
\end{aligned}
$$

プロセス Q の振る舞いをこのように変え先ほどと同じように FIS と $SPEC$ の refinement 関係を調べると，今度は反例がなく FIS は $SPEC$ の中で振る舞っていることが図 8 よりわかる．これにより FIS は決してイベント $enter$ が続くことはなく排他制御がうまくいっていることが保証される．

[*8] 3.3.1 節 最小遅延，最大遅延参照

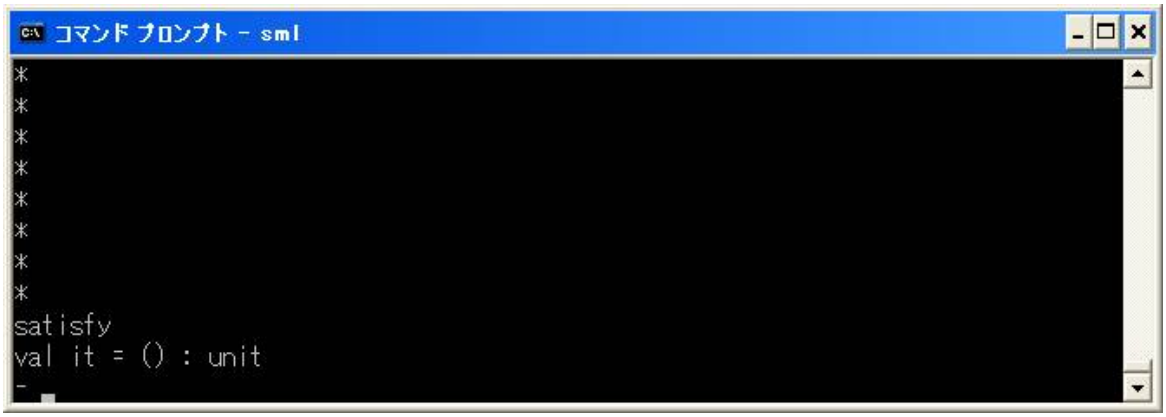

図 8　$\epsilon > \delta$ のときの検証結果

もし，$\epsilon < \delta$ 例えば，$\epsilon = 2, \delta = 3$ として再び refinement を調べると図 9 のような反例となる実行が検出される．プロセスがメモリの値を読み込み 0 であることを確かめ，メモリの値を自身の id で書き換えるまで最大 3time かかる．これと同時に別のプロセスが値を読み込み即座に自身の id を書き込む．その 2time 後に再びメモリの値を読み自身の id であることを確認し，$enter$ を起こす．その後，最初のプロセスがようやくメモリに自身 id を書き込み同じように 2time 後に $enter$ してしまうと排他制御に失敗する．

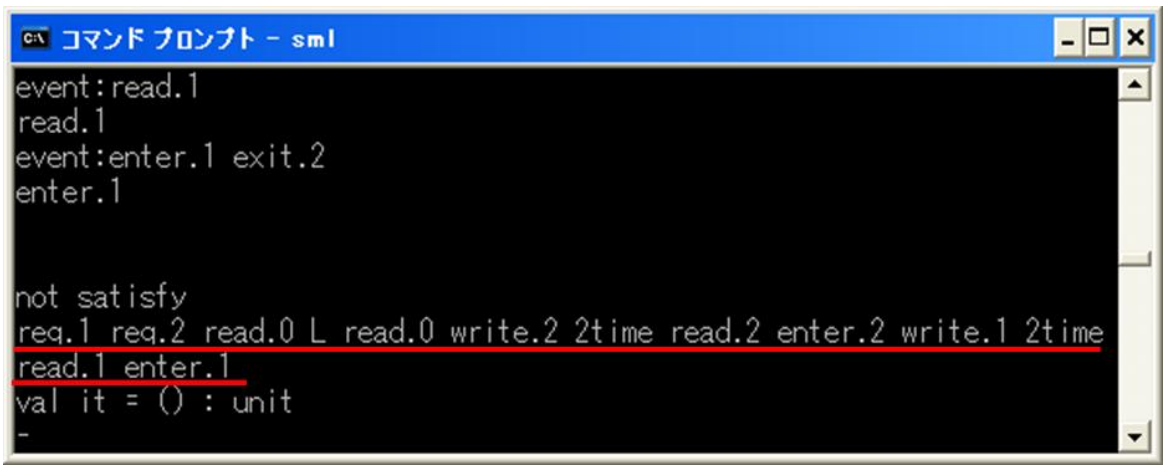

図 9　$\epsilon < \delta$ のときの検証結果

5 まとめ

5.1 結論

シングルコアの性能向上の限界や低電力化に向けてマルチコアの採用が進んでいる．マルチコアの性能を効果的に引き出すためには並行性を明示的に記述したプログラミングをする必要がある．しかしながら並行プログラミングには複数のプロセスによるデッドロック，競合などの問題があり，そのデバックも困難である．これに対する解決策として並行システムを形式的にモデリングし，検証することができるプロセス代数 CSP がある．CSP を用いて並行システムを設計することによりシステムの振る舞いを数学的にとらえるができ，開発の早い段階で正当性の検証が可能になる．検証後は CSP 実装用の各種言語を用いることで潜在的なバグのない安全なシステム開発が可能になる．また，CSP に時間の概念を入れ拡張した Timed CSP を用いることで，時間的な振る舞いを考慮したリアルタイムシステムに対してもモデリングおよび検証が行える．

Timed CSP でモデリングしたシステムは自動検証ツールを利用することで数学的理論を意識することなく CSP 検証可能であるが，Timed CSP に対応したツールはまだ出始めたばかりである．そこで，本研究では Timed CSP 検証のための技法の提案とプロトタイプの製作を行った．これには関数型言語を使いその利点をいかすことで，CSP のプロセスを簡潔に実装できることを示したうえで，Timed CSP への拡張を行った．そして，Timed CSP の検証方法のひとつである trace timewise refinement を行える能力の持ったプロトタイプを作成し，それを用いて排他制御アルゴリズムの検証を行った結果，誤った制御方法に対してはその反例となるケースを示し，改善した方法ではすべての状態を網羅的に検査しその正当性を示すことに成功した．

5.2 今後

現状, Timed CSP Explorer は CSP_M のコアな部分しか実装できていない．また, failures timewise refinement や timed failures refinement の検証はできない．今後は CSP_M の記述の完全サポートおよび timed failures refinement 等の検証ができるよう拡張していく必要がある．また，大規模なシステムに対してはモデル検査時に状態爆発を起こす可能性がある．大規模システムに耐えうるよう状態圧縮のアルゴリズムの考案とその実装が必要である．

6　謝辞

本研究を通じ多くのことを学びながら，非常に貴重な経験をつむ事ができました．本研究に取り組む機会を与えて頂き，適切なご指導と助言を下さった指導教官福永力教授に深く感謝いたします．また，ご指導と貴重な経験の場を与えていただいたプロミネントネットワークの松井氏，スマートスケープの吉田氏にも感謝いたします．今年度，昨年度と長い時間を共に過ごし仲間として互いに刺激しあえた，田中和人，岩波智史両氏にも感謝しています．そして，私を励まし温かく見守っていてくれた家族と友人に深く感謝いたします．

参考文献

[1] C.A.R.Hoare, 吉田信博 丸善株式会社
”Communicating Sequential Processes”「ホーア CSP モデルの理論」

[2] Inmos Ltd., ”Occam Programming Manual” (Prentice-Hall, 1984).

[3] P.H. Welch and P.D. Austin. The JCSP (CSP for Java) Home Page.
http://www.cs.kent.ac.uk/projects/ofa/jcsp/. (参照 2009-01-08)

[4] Neil Brown. C++CSP2 Easy Concurrency for C++ Home Page.
http://www.cs.kent.ac.uk/projects/ofa/c++csp/. (参照 2009-01-08)

[5] Formal Systems (Europe) Ltd.
Failures-Deivergences Refinement: FDR2 User manual, 1992-2008

[6] School of Computing, National University of Singapore. PAT: Process Analysis Toolkit.
http://www.comp.nus.edu.sg/ pat/. (参照 2009-01-08)

[7] 磯部祥尚（産業技術総合研究所）
「プロセス代数 CSP の定理証明器 CSP-Prover の紹介」第 2 回 CSP 研究会 2009

[8] the SML/NJ Fellowship.. Standard ML of New Jersey.
http://www.smlnj.org/. (参照 2009-01-08)

[9] Bryan Scattergood ”The Semantics and Implementation of Machine-Readable CSP”
Thesis submitted for the degree of Doctor of Philosophy at the University of Oxford, Trinity 1998

[10] Steve Schneider. ”Concurrent and Real-time Systems: the CSP Approach”.

[11] A.W.Roscoe. ”The Theory and Practice of Concurrency”. Prentice Hall, 1998

[12] 坂本達哉, 首都大学東京 修士論文
「CSP 記述によるモデル設計とツールによる検証」2007

[13] 田中和人, 岩波智史, 山川武志, 福永力 (首都大学東京),
松井和人 (プロミネントネットワーク), 吉田隆 (スマートスケープ)
「Proposal of CSP based Network Design and Construction」2007