CSP 記述によるモデル設計と
ツールによる検証

Sakamoto Tatsuya
06878309

2008 年 1 月 23 日

目次

1 序論

1.1 本研究の背景と意義

近年あらゆるシステムは複雑になるばかりである。その一方で複雑になったシステムはそれだけデバッグが困難になってしまう。このデバッグが困難であるということと、システムの検討よりも先にまず「もの」を作るべきであるという姿勢から、これまではまずは実際にシステムを制作してしまい、実際に動作チェックを行う中でバグが発見されたらそれを取り除くという場当たり的なデバッグが行われてきていた。

並列処理システムは、それぞれ独立に動く複数のサブプロセスが互いに同期処理やデータ通信処理などの連携をしながら、全体として一つのシステムを構築していると考えられる。この一つ一つのサブシステムについてデバッグをすることは比較的容易だが、並列処理によって結びついたシステムには独特の制約が発生するため、システム全体としてのデバッグは非常に困難であるといえる。

これらを踏まえると、もはや場当たり的なデバッグを繰り返すだけではシステム全体としてのバグを完全に洗い出すことは不可能であり、明確にバグが取り除かれたことを示すことができないこの方法では、いつまでもシステムは不安を抱えたままになってしまう。このことが、時間的に開始、終了、実行時間に関する制約の厳しい並列処理システムの構築を実際に行うにあたり、システムの正当性が数学的に証明されていることが必要だと言われる所以である。

こうした状況を踏まえ、私は実際にシステム制作を行う前のモデル設計の段階で正当性が数学的に証明されたシステムを構築することが必要不可欠と考え、形式手法の一つとして知られる CSP 記述（Communicating Sequential Process、通信逐次プロセス）によるモデル設計を行った。CSP 記述はプロセスをイベントで書き表し、そこから様々な数学的演繹、推論を展開することを可能にした記述方法である。CSP 記述については次章で詳しく触れる。また、これに関連するツールは CSP 記述の正当性を裏付けし、信頼性の非常に高いシステムを構築する手助けとなる。これら関連ツールに関しては 4 章、6 章で詳しく触れる。これによって作られたシステムは、実際の動作テストをするまでもなく数学的に正当性が示されていることになる。

以上のような背景、意義を持って私はモデル設計の段階でシステムの正当性が検証されていることがシステムの制作全体に与える影響を示すため、実際に CSP 記述によってモデル設計を行うべく本研究に着手した。

1.2 本研究の目的

CSP 記述と、それから導き出される様々な推論について把握し、論理的にシステムホールのないモデル設計を行う。そして、そのモデル設計が論理的正当性をもつことを、FDR2、UPPAAL といったツールで検証する。

2 CSP 記述とは

2.1 CSP 記述の概要

CSP（Communicating Sequential Process）は Tony Hoare 氏とその仲間が 1980 年代に理論展開したもので、外部の環境と互いに反応し合いながら連続運転されるシステムを、ある共通環境下で互いに通信し合いながら並行動作するサブシステムへと分解して考察していくということが基本的なアイディアになっている。最初の CSP は操作系の開発において発生した並列性問題を解決するためのプログラミング言語であった。この言語は通信路を通して交信するプロセスによって並行プログラムを作成することの有効性を検証することが目的であった。こうして登場した言語のうち、変数や代入文などを捨象し、交信や並行性の検討に的を絞ったものがここで取り上げる CSP（第 2 次 CSP）である。

CSP 記述は、システムを理解するだけでなく、実際にモデル構築するための方法であり、そのための手段として、いくつかの独立したシステムについて根本的な定理によって議論し、それらがいかにして組み合わされているかという点に着目する。その上で、全体のシステムは、システムが適当に組み合わされてできた、より大きなシステムと考える。

CSP 記述では、システムをプロセスの状態遷移によってとらえる。状態遷移は周囲の環境によって大きく左右される。プロセス自身が環境に左右されず状態遷移をする場合もあるが、多くは環境からの何らかの作用によって状態遷移が発生する。この作用をイベントと呼び、システムにおけるプロセスはイベントの集合と考える。

例えば電話機に関して言えば、プロセスの状態遷移に関わるイベントは 12 個のボタンを押すことと、受話器を取る、戻すという作業、そして、着信によって電話が鳴るという動作に集約されるので、この電話機のプロセスは、

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \#, *, \mathit{handset.lift}, \mathit{handset.replace}, \mathit{ring}\}$$

というイベントの集合として考えられる。ここで、この電話機のプロセスを $PHONE$ と表すならば、$PHONE$ に含まれるイベント全ての集合は $\alpha(PHONE)$ と表す。

また、これらのイベントはいつでも同様の状態遷移を引き起こすわけではない。受話器を置いたままでボタンを押しても状態遷移は起こらない。また、受話器を取った状態で電話が鳴ることもなく、既に受話器を取った状態では、さらに受話器を取ることはできない。このように、ある状態でそれによって状態遷移するイベントを列挙し、起こりうる状態遷移の経過やプロセスに影響しないイベントなどを、論理的に検討していく。

CSP 記述に関しては [1] に詳しく書かれている。以下に、その中でも触れられている CSP 記述で重要ないくつかの基本的な規則を述べる。

2.1.1 逐次処理

P_1、P_2 をプロセス、μ をイベントとする。このとき、プロセス P_1 がイベント μ を受理し、それによってプロセス P_2 に状態遷移するとき、これを

$$P_1 \stackrel{\mu}{\longrightarrow} P_2$$

で表す。またこれを P_1 について考えれば、P_1 というプロセスは

$$P_1 = \mu \longrightarrow P_2$$

と記述される。つまりこれらを合わせると

$$(\mu \longrightarrow P_2) \xrightarrow{\mu} P_2$$

という関係がわかる。

さて、前述したとおり CSP 記述の目指しているのは、連続運転されるシステムである。この連続運転を実現するためには、以下のような再帰定義が有効である。

$$LIGHT = on \longrightarrow off \longrightarrow LIGHT$$

例えばこれは、電源の入り切りを繰り返すシステムの CSP 記述である。$LIGHT$ というプロセスは、on、off というイベントを実行した結果、最初の $LIGHT$ という状態に戻る。これによってこの $LIGHT$ というプロセスは無限に on、off を繰り返すことが示される。もちろんこの記述によって、on の後に off が実行されることだけでなく、off の後に on が実行されることも示されている。

またこの再帰表現は、いくつかのプロセスが相互に作用して一つの再帰表現を示す場合もある。

$$LIGHT = on \longrightarrow ON$$
$$ON = off \longrightarrow LIGHT$$

この記述では $LIGHT$ だけではただの状態遷移にしかなっていないが、ON と合わせることで一組の再帰表現になっている。このことは、一組の再帰表現がいくつのプロセスから成り立っているかは関係ない。

ここで入出力を考える。システムにおける入出力は、全てチャンネルを介して行われると考える。ここで、チャンネル in からから入力した情報をそのままチャンネル out に出力するプロセス $COPY$ を考えると以下のようになる。

$$COPY = in?x \longrightarrow out!x \longrightarrow COPY$$

ここで、?は入力を、!は出力を表している。

システムの可能性を拡張する表現として選択が挙げられる。選択は大きく分けて、外部選択と内部選択の二種類が存在する。

外部選択は環境に依存するイベントによって選択が決定されるものである。例えば a、b をイベント、P、Q をプロセスとすると、

$$a \longrightarrow P \ \square \ b \longrightarrow Q$$

という外部選択は、イベント a が発生すれば状態 P に、イベント b が発生すれば状態 Q に遷移する。外部選択には結合法則が成り立ち、一般に有限個に拡張しても成り立つ。

内部選択は、外部の環境によらずプロセス内で選択が独自に決定されるものである。例えば a、b をイベント、P、Q をプロセスとすると、

$$a \longrightarrow P \ \sqcap \ b \longrightarrow Q$$

という内部選択は、プロセス自身が選択を決定してしまう。そのため、例えばイベント a が発生しても状態 P に遷移せず、イベント b が発生したときのみ状態 Q に遷移する。この選択については、毎回この選択の状態になったときに抽選で決定する可能性もあれば、最初から b が選択されることがないよう設計されている可能性もあるが、それらは全体のモデル設計には関係のないこととして扱う。内部選択も有限個の選択に拡張することができる。

2.1.2 並列処理

CSP 記述でシステムを記述する場合、複数のプロセスが同時に動作するということは、基本的にそれぞれが独立に動作することを意味する。しかし中には、リレー競技におけるバトンパスのような、握手をするように二つのプロセスで同時に一つのイベントを実行するものもある。例えば P_1、P_2 をプロセスとし、A を P_1 に含まれるイベントの集合、B を P_2 に含まれるイベントの集合とすると、

$$P_1 \ {}_A\|_B \ P_2$$

は P_1 と P_2 の並列処理を表す。このとき $A \cap B$ に含まれるイベントは P_1 と P_2 の両方で同時に実行されなければならない。ここで特に集合を明記せず

$$P_1 \ \| \ P_2$$

と書いた場合には、

$$P_1 \ {}_{\alpha(P_1)}\|_{\alpha(P_2)} \ P_2$$

を表すものとする。さらに、$X = A \cap B$ とおくと、このプロセスは

$$P_1 \ \underset{X}{\|} \ P_2$$

とも書く。またこれにおいて特に $X = \{\}$ のとき、この並列処理を P_1 と P_2 のインターリーブといい、

$$P_1 \,|||\, P_2$$

で表す。これらの並列処理も一般的に有限個に拡張することができる。

並列処理について議論する上で最も重要な概念の一つに deadlock がある。deadlock は、並列処理を行う二つのプロセスの間で受理可能なイベントが食い違うことなどがが原因で起こるシステムの停止状態である。例えば以下の例が挙げられる。

$$\begin{aligned} PETE &= lift\_piano \longrightarrow PETE \\ &\quad \sqcap\ lift\_table \longrightarrow PETE \\ DAVE &= lift\_piano \longrightarrow DAVE \\ &\quad \sqcap\ lift\_table \longrightarrow DAVE \\ TEAM &= DAVE \ \| \ PETE \end{aligned}$$

これは、$DAVE$ と $PETE$ がともに、いくつかのピアノとテーブルを運ぼうとしているプロセスである。ピアノもテーブルも一人で運ぶことはできず、二人が両方とも同じイベントを選択しなければそのイベントは実行されない。以上を踏まえてこのプロセスを見る。

前節で内部選択は、外部の環境やイベントに関係なく独自に選択が決定されることを説明した。この場合も例外ではなく $DAVE$ も $PETE$ も独自に選択を決定してしまう。よって、二人ともピアノを運ぶことを選択すればイベントは実行されるが、二人の選択が異なってしまった場合、二人ともう一方が同じ選択をすることを待ち続けてしまう。これが deadlock である。

モデル設計にあたって deadlock の可能性を残すことは絶対にあってはならない。例えばこの場合であれば、

$$\begin{aligned} PETE' &= lift\_piano \longrightarrow PETE' \\ &\quad \Box\ lift\_table \longrightarrow PETE' \end{aligned}$$

のように $PETE$ が自ら選択を決定することを止め、$DAVE$ の選択次第で自身の選択を決定することにすれば、並列処理

$$TEAM' = DAVE \parallel PETE'$$

は deadlock の可能性を排除することができる。

並列処理は小さなシステムを組み合わせて大きなシステムを構築するという概念に基づくが、各システムの独自性を最大限維持するための概念として遮蔽というものがある。

単にイベントという場合は普通外部イベントのことを差し、プロセスの状態遷移を決定するイベントも外部イベントの発生によって決定される場合が多い。しかし、外部的な環境に依存する部分を増やすことは、システムの拡張性としては足枷を増やすことに繋がってしまう。そこでシステムを拡張するにあたって、外部に曝す必要のないものはそのシステムの中だけで閉じてしまおうというのがこの遮蔽である。例えば次のようなものがある。

$$\begin{aligned} S &= in?x \longrightarrow mid!x \longrightarrow ack \longrightarrow S \\ R &= mid?y \longrightarrow out!y \longrightarrow ack \longrightarrow R \\ SAWP &= (S \parallel R) \setminus \{mid, ack\} \end{aligned}$$

このシステムは、二つのプロセス S と R が協力して、チャンネル in への入力をチャンネル out に出力するというものである。この二つのプロセスは、データ通信チャンネルの mid で繋がっている他、同期をとるためのイベント ack で結ばれている。この部分を遮蔽することによって、実際には更に細かく別れているが、全体のシステムとしては $SAWP$ を最小の単位とみなすことになる。なぜなら、mid や ack は外部からイベントの発生を起こせないので、システムの内部的に発生した内部イベントだとみなすことになるからである。

2.1.3 trace

プロセスの挙動を示す一つの表現として trace というものがある。これは、あるプロセスが逐次実行されていく中で取り得る外部イベント列を列挙したものである。イベント P における trace の集合を $\mathrm{traces}(P)$ と書き表す。

この trace もいくつかの簡単な演算法則を適用することができる。例えば前述の $TEAM$ や $TEAM'$ に適用

すると、

$$\begin{aligned}
\mathrm{traces}(DAVE) &= \mathrm{traces}(\mathit{lift_piano} \longrightarrow DAVE \\
&\qquad \sqcap \mathit{lift_table} \longrightarrow DAVE) \\
&= \mathrm{traces}(\mathit{lift_piano} \longrightarrow DAVE) \\
&\quad \cup \mathrm{traces}(\mathit{lift_table} \longrightarrow DAVE) \\
&= \{\langle\rangle\} \cup \{\langle \mathit{lift_piano}\rangle \frown tr | tr \in \mathrm{traces}(DAVE)\} \\
&\quad \cup \{\langle\rangle\} \cup \{\langle \mathit{lift_table}\rangle \frown tr | tr \in \mathrm{traces}(DAVE)\} \\
&= \{tr | tr \in \{\mathit{lift_piano}, \mathit{lift_table}\}^*\} \\
\mathrm{traces}(PETE) &= \mathrm{traces}(DAVE) \\
\mathrm{traces}(TEAM) &= \mathrm{traces}(DAVE \parallel PETE) \\
&= \{tr \in TRACE | tr \upharpoonright \alpha(DAVE) \in \mathrm{traces}(DAVE) \\
&\qquad \wedge tr \upharpoonright \alpha(PETE) \in \mathrm{traces}(PETE) \\
&\qquad \wedge \sigma(tr) \subseteq \alpha(DAVE) \cup \alpha(DAVE) \cup \{\surd\} \\
&= \{tr | tr \in \{\mathit{lift_piano}, \mathit{lift_table}\}^*\} \\
\mathrm{traces}(PETE') &= \mathrm{traces}(PETE) \\
\mathrm{traces}(TEAM') &= \mathrm{traces}(TEAM)
\end{aligned}$$

とできる。ここで $\{\mathit{lift\_piano}, \mathit{lift\_table}\}^*$ は、空列または *lift_piano*、*lift_table* からなる任意のイベント列を表す。つまり、$TEAM$ も $TEAM'$ も無限にイベントを繰り返す可能性があることを示唆している。

ここで重要なのは、traces($TEAM$) に含まれるイベント列が、必ずしもプロセス $TEAM$ で実行されるとは限らないことである。実際このプロセス $TEAM$ は既に見てきたとおり deadlock に陥ることがあり、$\langle \mathit{lift\_table}, \mathit{lift\_piano}, \mathit{lift\_piano}\rangle$ は確かに traces($TEAM$) に含まれるが、これが実行されるより先に deadlock に陥り、このイベント列が実行されない可能性は残されている。

2.1.4 failure

trace とは逆に、ある状態において実行されないイベントを列挙した集合を failure という。あるプロセス P に対して、あるイベントの集合 X に含まれる全てのイベントが実行されない可能性があるとき、集合 X はプロセス P の refusal であるという。またあるプロセス P に対して、ある $tr \in \mathrm{traces}(P)$ があるとする。ここで P が tr のイベント列を実行して状態遷移した状態 P' において、あるイベントの集合 X' に含まれる全てのイベントが実行されない可能性があるとき、trace tr と集合 X' の組み合わせ (tr, X') はプロセス P の failure であるという。プロセス P の failure となる組み合わせ全体の集合を $\mathcal{F}[[P]]$ で表す。

このとき $\mathcal{F}[[P]]$ の要素として $\{(tr, X) | X = \alpha(P)\}$ を含むということは、プロセス P が trace tr によって状態遷移した状態では全てのイベントが実行されない可能性があるということである。この全てのイベントが実行されない状態が deadlock に対応している。

この failure もいくつかの簡単な演算法則を適用することができる。例えば前述の $TEAM$ や $TEAM'$ に適

用すると、

$$
\begin{aligned}
\mathcal{F}[[DAVE]] &= \mathcal{F}[[lift\_piano \longrightarrow DAVE \\
&\qquad \sqcap\ lift\_table \longrightarrow DAVE]] \\
&= \mathcal{F}[[lift\_piano \longrightarrow DAVE]] \\
&\quad \cup \mathcal{F}[[lift\_table \longrightarrow DAVE]] \\
&= \{(\langle\rangle, X) | lift\_piano \notin X\} \\
&\quad \cup \{(\langle lift\_piano \frown tr\rangle, X) | (tr, X) \in \mathcal{F}[[DAVE]]\} \\
&\quad \cup \{(\langle\rangle, X) | lift\_table \notin X\} \\
&\quad \cup \{(\langle lift\_table \frown tr\rangle, X) | (tr, X) \in \mathcal{F}[[DAVE]]\} \\
&= \{(tr, X_1) | tr \in \{lift\_piano, lift\_table\}^* \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq X_1\} \\
\mathcal{F}[[PETE]] &= \mathcal{F}[[DAVE]] \\
&= \{(tr, X_2) | tr \in \{lift\_piano, lift\_table\}^* \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq X_2\} \\
\mathcal{F}[[TEAM]] &= \mathcal{F}[[DAVE \parallel PETE]] \\
&= \{(tr, X_1 \cup X_2) | tr \in \{lift\_piano, lift\_table\}^* \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq X_1 \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq X_2\} \\
&= \{(tr, X) | tr \in \{lift\_piano, lift\_table\}^*\} \ldots\ldots\ldots\ldots\ldots\ldots (*) \\
\mathcal{F}[[PETE']] &= \mathcal{F}[[lift\_piano \longrightarrow PETE' \\
&\qquad \Box\ lift\_table \longrightarrow PETE']] \\
&= \{(\langle\rangle, X) | ((\langle\rangle, X) \in \mathcal{F}[[lift\_piano \longrightarrow PETE']] \\
&\qquad\qquad \cap \mathcal{F}[[lift\_table \longrightarrow PETE']])\} \\
&\quad \cup \{(tr, X) | tr \neq \langle\rangle \wedge ((tr, X) \in \mathcal{F}[[lift\_piano \longrightarrow PETE']] \\
&\qquad\qquad\qquad\qquad \cup \mathcal{F}[[lift\_table \longrightarrow PETE']])\} \\
&= \{(tr, \{\}) | tr \in \{lift\_piano, lift\_table\}^*\} \\
\mathcal{F}[[TEAM']] &= \mathcal{F}[[DAVE \parallel PETE']] \\
&= \{(tr, X_1 \cup \{\}) | tr \in \{lift\_piano, lift\_table\}^* \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq X_1 \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq \{\}\} \\
&= \{(tr, X_1) | tr \in \{lift\_piano, lift\_table\}^* \\
&\qquad \wedge \{lift\_piano, lift\_table\} \not\subseteq X_1\} \ldots\ldots\ldots\ldots\ldots\ldots (**)
\end{aligned}
$$

とできる。ここで、$(*)$ から、$\mathcal{F}[[TEAM]]$ は任意の tr に対して $\{(tr,X)|X = \alpha(P)\}$ を含むことが分かる。一方、$(**)$ から、$\mathcal{F}[[TEAM']]$ はどんな tr に対しても $\{(tr,X)|X = \alpha(P)\}$ を含まないことがわかる。つまり、$TEAM$ は任意の状態で deadlock に陥る可能性があること、$TEAM'$ は deadlock に陥る可能性がないことが、この演算から導き出される。

trace と同様に failure も、集合に含まれるイベントの中でも、その時点で実行される可能性のあるものが入っていることがある。そもそも一つの trace に対して、その結果辿り着く状態が一つとは限らないことに注意したい。

3　CSP 記述によるモデル構築

3.1　CSP 記述の有用性

CSP 記述の最大の長所は、trace、failure などの概念記述に数学的な意味が与えられ、命題論理の発展としてシステムの演繹、推論が行われ、プロセス記述の正当性が数学的に証明されることである。

並列処理システムでは、複数のプロセスが独立に動作している。単独のプロセスについても、システムが拡大するにつれて満たされるべき仕様の幅が広がり、徐々にシステムのデバッグが困難になっていくが、これはまだ比較的容易である。一方これに加えて同期通信処理やチャンネル通信によるデータの入出力など、プロセス間の連携が懸念要素に含まれると、システム全体のデバッグは急激に困難になる。

システム設計においてシステム不良による再設計は、これまで想像以上のコストを費やしてきた。これは、理論的設計が個人の裁量に任されるだけでなく、デバッグ作業についても、場当たり的な対処しかできていなかったことによる。システム設計の段階が進んだ後に基礎の再設計をすることがプロジェクトに多大な影響を与えることは、言うまでもない。

単に並列処理システムというだけでこれだけの労力を必要とすることを鑑み、時間的制約条件の厳しい並列処理システムを構築するにあたって、一つの数学的に確立された検討方法によってそのシステムの正当性を証明することにより、システム不良による再設計を排除することが必要である。

CSP 記述における数学的検討可能性はこの要素を満たし、CSP 記述による演繹、推論が、システムの正当性を証明する理論的根拠となる。

4　FDR2 によるモデルの検証

4.1　FDR2 の概要

CSP 記述に触れる中で、CSP の概念を取り入れたいくつかのツールに触れた。一つは CSP 記述における deadlock の可能性についての検証ツールとしての FDR2 である。

FDR2 は Formal Systems Ltd によって開発された CSP 記述の検証ソフトウェアである。FDR2 は、システムにおける逐次実行の可能性としての trace、システムのある状態におけるイベントの拒否可能性としての failure、内部イベントによる無限遷移状態としての divergence という三つの観点からシステムをチェックする。

CSP 記述を用いるのみでは問題を記述しただけであり、これから数学的推論により正当性の証明を行うことは原理的に可能だが、FDR2 はその証明を PC 上で処理することが可能なソフトウェアである。したがって、この FDR2 によって、CSP 記述されたシステムの正当性の判断ができる。

4.2　FDR2 による検証例

ここでは先ほど紹介した

$$
\begin{aligned}
PETE &= lift\_piano \longrightarrow PETE \\
&\quad \sqcap\ lift\_table \longrightarrow PETE \\
DAVE &= lift\_piano \longrightarrow DAVE \\
&\quad \sqcap\ lift\_table \longrightarrow DAVE \\
TEAM &= DAVE \parallel PETE \\
PETE' &= lift\_piano \longrightarrow PETE' \\
&\quad \Box\ lift\_table \longrightarrow PETE' \\
TEAM' &= DAVE \parallel PETE'
\end{aligned}
$$

のプロセスを用いて説明する。

4.2.1　検証方法

FDR2 で検証するためには、検証するプロセスをテキストエディタで編集する必要がある。このための文法は元々の CSP 記述と一対一に対応していて、CSP 記述で示されたプロセスをほぼそのままの形で FDR2 が読み取れる形に変換することができる。

以下にプロセスの CSP 記述を FDR2 で検証できる形に変換したプログラムを記述する。

```
channel lift_piano,lift_table

 PETE = lift_piano -> PETE
        |~| lift_table -> PETE
 DAVE = lift_piano -> DAVE
        |~| lift_table -> DAVE
 TEAM = DAVE [|{|lift_piano,lift_table|}|] PETE
```

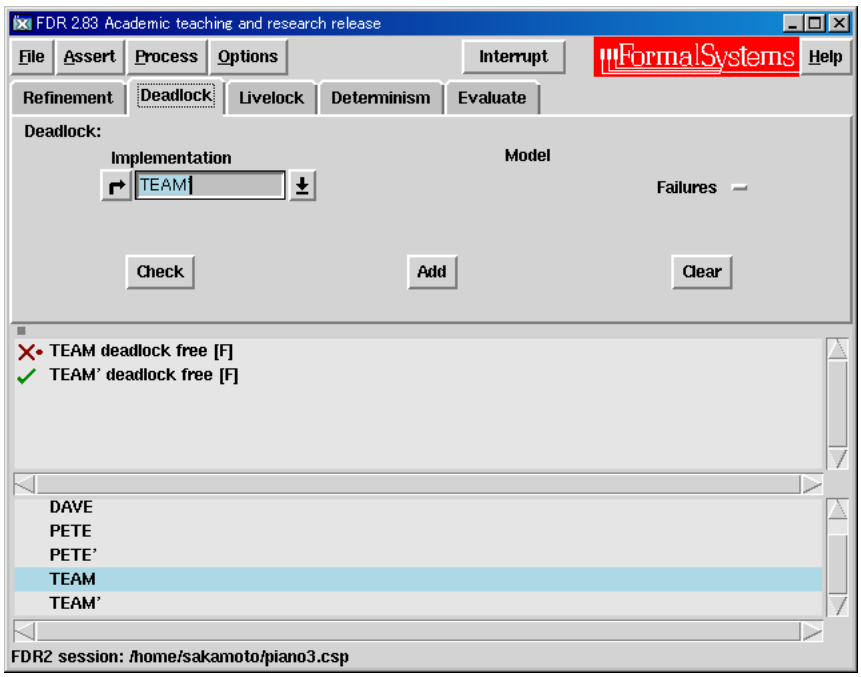

図 1　FDR2 による検証実行画面 1

```
PETE' = lift_piano -> PETE'
        []  lift_table -> PETE'
TEAM' = DAVE [|{|lift_piano,lift_table|}|] PETE'
```

4.2.2　FDR2 における machine readable な表記

CSP 記述を FDR2 で検証するための代表的なイベントの表記は以下の通りである。

Operator	Syntax	ASCII form
Prefixing	$a \rightarrow P$	a -> P
Output	$c!v \longrightarrow P$	c ! v -> P
input	$c?m : T \longrightarrow P(m)$	c?m:T -> P(m) channel c : S where T $\subseteq$ S
Recursion	$N = P$	N = P
External choice	$P_1 \Box P_2$	P_1 [ ] P_2
Internal choice	$P_1 \sqcap P_2$	P_1 | ~ | P_2
Interleaving	$P_1 \mid\mid\mid P_2$	P_1 | | | P_2
Interface parallel	$P_1 \underset{A}{\parallel} P_2$	P_1 [| A |] P_2
Hiding	$P \backslash A$	P \ A

4.2.3 検証結果

これらのプロセスを deadlock について検証した結果が図 1 の通りである。緑色の $\surd$ は検証に成功したことを、赤色の $\times$ は検証に失敗したことを示す。この場合では、*TEAM* は deadlock に陥る可能性があること、*TEAM'* は deadlock free に陥る可能性がないことが示されている。

既に示したとおり、プロセス *TEAM* は deadlock に陥る可能性があること、一方プロセス *TEAM'* は deadlock に陥る可能性がないことが、FDR2 によっても検証されたことがわかる。以上のことから、プロセス *TEAM'* が並列処理システムとして deadlock を含まない健全なシステムであることが検証された。

また FDR2 は同様の検証によって、内部イベントの無限循環による livelock の有無、プロセス外部からの選択の決定性 deterministic の有無を検証することが可能である。それだけでなく、trace や failure に包含関係のある refinement の関係も検証することが可能である。

例えば、プロセス *PETE* はイベントの選択を内部選択によりプロセス自身で決定してしまうため、プロセス外部からの選択の決定性はない。それに対してプロセス *PETE'* はイベントの選択を外部選択によりプロセス外部からの選択に委ねるため、プロセス外部からの選択の決定性があるといえる。つまり *PETE'* は deterministic があり、*PETE* は deterministic がない。また、前章から trace(*PETE*)=trace(*PETE'*) が成り立つことがわかる。このように trace の集合が等しいことを $PETE =_T PETE'$ と表すが、このことも FDR2 によって検証することができる。

5　CSP 記述や FDR2 の脆弱性

CSP 記述が数学的検討を可能にしていること、また、FDR2 がこの検証のためのツールとして優れた効果を発揮していることをこれまで見てきた。

実際のシステムで一番の問題となるのが、予期せぬシステムダウンである。例外処理を細かく設定することももちろん重要ではあるが、これらの例外に陥る可能性を列挙し、それら一つ一つの処理を的確に明示することによって、例外の可能性を減らすことも、システムダウンを回避する重要な要素である。

特に再帰処理についてこのことは顕著に現れる。再帰処理に関して正しく再帰が完了しない場合、プロセスが元の待機状態に帰着しない場合が存在すると、システムが deadlock に陥ることがしばしばある。例えば

$$\begin{aligned} P(i) = \mathit{down} &\longrightarrow P(i-1) \ (\text{if } i > 0) \\ \sqcap\ \mathit{up} &\longrightarrow P(i+1) \ (\text{if } i < 10) \end{aligned}$$

というプロセスを考える。ここで i は $0 \leq i \leq 10$ の値を取り得る変数である。

このプロセスは一見再帰が完了しているように見えるが、保持する変数が変化するため再帰が正しく定義されていない。例えば $P(0)$ の状態において、内部選択において $\mathit{down} \longrightarrow P(i-1)$ (if $i > 0$) が選択されてしまうと、$P(0)$ は down によって状態遷移ができないため deadlock に陥ってしまう。

しかし FDR2 はこれを見逃さずに指摘する。これにより、常にシステムの正当性を保ったまま構築を続けられることになる。

ここまでの検証で、deadlock によるシステムの停止、livelock によるシステム状態の発散の有無を検証してきたが、これによってシステムの正当性が確証されるのは、全体の時間軸に対して、プロセスの実行時間が無視できるほど小さい場合である。

リアルタイムシステムでは、イベント選択や再帰処理だけでなく、プロセスの実行時間がシステムの正当性そのものに大きく関わる。しかし、従来の CSP 記述だけではこの実行時間を有効に記述、推論する方法がなく、FDR2 でもこれを検証することはできない。リアルタイムシステムについてシステムの正当性を検証するためには、これまでの議論に加えて時間的概念を導入しての検証をする必要がある。

以降の章では、時間的概念を加えて議論をするために、新たな検証ツールを交えて議論を進める。

6 UPPAAL によるモデルの検証

6.1 UPPAAL の概要

時間的制約の厳しいシステム（リアルタイムシステム）では、さらにシステムの行動の時間的検討も行わなければならない。本論文で議論するもう一つのツールは、時間的設計を実験的、総当たり的に検証する UPPAAL [4] である。

UPPAAL は Uppsala University によって開発された時間検証ツールである。UPPAAL は、CSP 記述から導出される有限時間オートマトンをグラフィカルに設計することによって、リアルタイムシステムの妥当性や、ある状態への到達可能性といった動的な振る舞いを、網羅的に調査する。NP 問題に属する問題など、調査対象が指数関数的に増大する場合でも、予め設定されている Hash table（最大 512MB）を埋め尽くすまで調査は続けられる。UPPAAL を用いることによって、一つ一つの動作が想定された処理時間内に収まっていることを確認、もしくは想定時間内だけでは処理が終了しないことを検証することができる。

UPPAAL はそれ自体が CSP 記述に準拠しているわけではなく本質的には CSP 記述とは全く別のものであるが、そこに使われているチャンネル通信による同期など、CSP 記述の概念が根底に含まれていると言って差し支えなく、また、CSP 記述に使われている多くの表現は UPPAAL でも表現することができる。

6.2 UPPAAL による検証例

ここでは TSP(traveling Salesman Problem、巡回セールスマン問題) という問題を解くことを考える。これは、あるセールスマンがいくつかの都市を一度ずつ訪問して出発点に戻ってくるときに、移動距離が最短になる経路を求める問題で、都市の数が大きくなると、一般には多項式計算時間内では解くことのできない、いわゆる NP 問題に属する。

今回のケースは、X 社を出発したセールスマンが、A 社、B 社、C 社、D 社を一回ずつ訪問して再び X 社に戻ることを考える。各社では毎回 20 分ずつ滞在し、各社間の移動時間は以下の表の通りである。

	X 社	A 社	B 社	C 社
A 社	5			
B 社	7	12		
C 社	16	19	19	
D 社	18	22	13	20

TSP は NP 問題であるが、今回のケースでは訪問する都市が 5 つと少ないため短い計算時間内で調査を終了できる。

このプロセスが健全なシステムであることは既に示されている。

6.2.1 検証方法

UPPAAL で検証するためには、検証するプロセスをグラフィカルに描画する必要がある。UPPAAL の画面上で有限オートマトンを描画し、CSP 記述で示されたプロセスを対応するオートマトンに変換することで、UPPAAL での検証を可能にする。

以下にこの問題を UPPAAL で検証するために描画したオートマトンを掲載する。

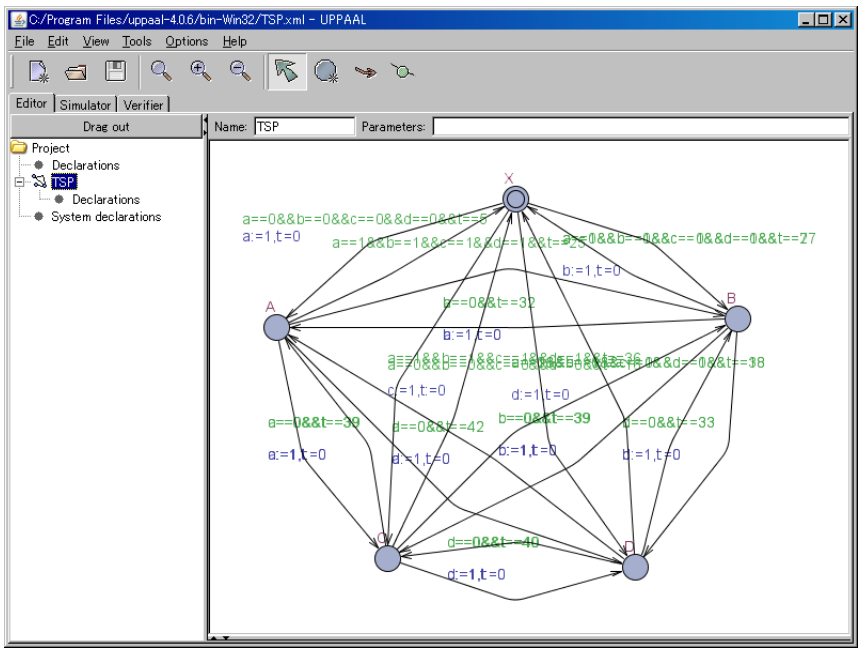

図 2　UPPAAL による TSP の描画

6.2.2　UPPAAL におけるシステム表現

UPPAAL では、一つのプロセスの状態を一つの節で表し、状態遷移を矢印で示す。一つの節から複数の矢印が出ている場合、その状態には状態遷移の選択があることが分かる。このシステム全体はグローバル変数を持つことができるほか、各プロセスはローカル変数を持つことができる。変数の型としては、論理型、整数型、時間型が挙げられるが、実数型を扱うことはできない。

以下、TSP を UPPAAL で描画するにあたり、a,b,c,d を整数型の変数、t,$clock$ を時間型の変数として設定されているものとする。

図 3 において、X、A の○はそれぞれプロセスの状態で、それぞれセールスマンが X 社、A 社にいることを表す。X が◎になっているのは、状態 X が初期状態であることを表す。X から A への矢印は状態 X から状態 A への遷移があることを表し、このときに変数を使って状態遷移の条件を設定することができる。緑色の $t == 5$ は Guard と呼ばれ、時間変数 t が $t = 5$ のときのみ状態 X から状態 A への遷移の可能性があることを表し、これは X 社から A 社までの移動に 5 分かかることを表現している。青色の $a := 1, t := 0$ は Update と呼ばれ、状態 X から状態 A への遷移の直後に整数変数 a を $a = 1$ に、時間変数 t を $t = 0$ にするという設定であり、これは $a = 0$ が A 社にまだ訪問していないことを、$a = 1$ が A 社を既に訪問したことを表すことによって一度ずつの訪問を可能にし、時間変数を $t = 0$ リセットすることによって、状態 A でもう一度 0 から時間を計り直すことを可能にする。状態遷移に伴う設定はこの他にも、複数のプロセス間で同期通信処理を行うための Sync などを挙げることができる。

図 3 から更に状態 B を追加したのが図 4 である。

各状態から 2 本の矢印が出ていることは、イベントの選択が起こりうることを意味する。例えば図 4 の状態は、初期状態の状態 X から状態 A へ遷移した後、状態 B へ更に遷移するか状態 X へ戻るかの選択があることを意味する。しかし今回の場合に限って言えば、Guard から状態 X の遷移はできないので、結果的に状態

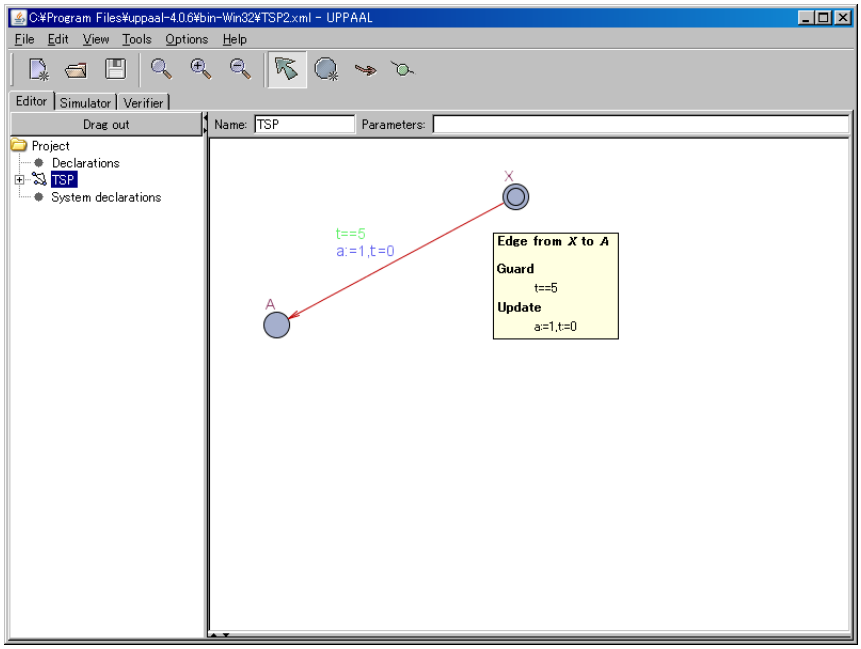

図 3　UPPAAL の描画設計画面 1

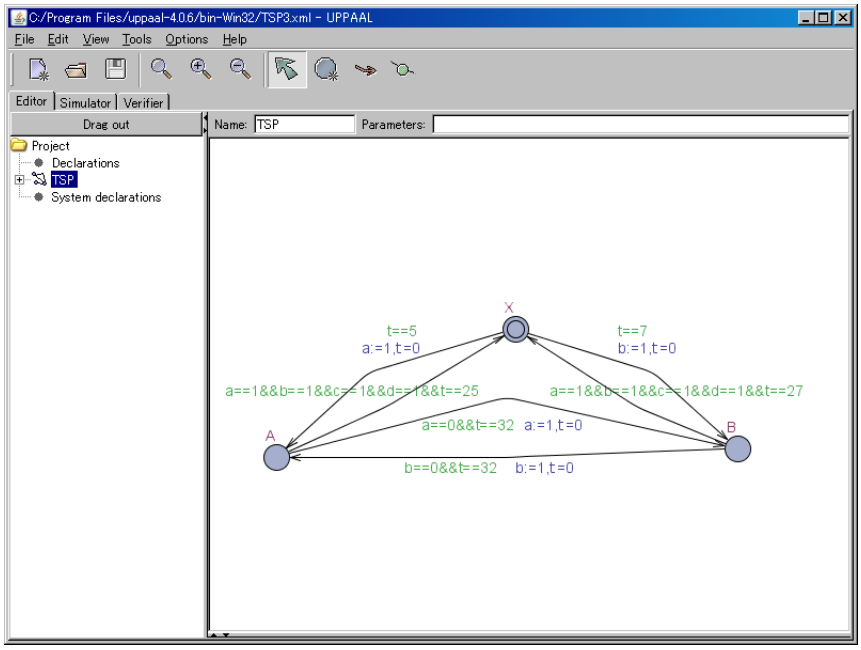

図 4　UPPAAL の描画設計画面 2

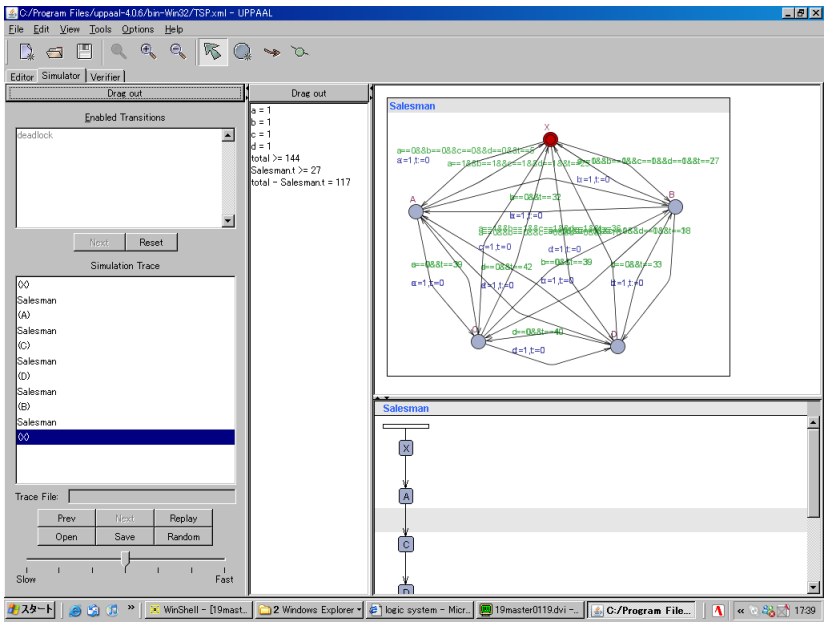

図 5　UPPAAL による検証実行画面全体図

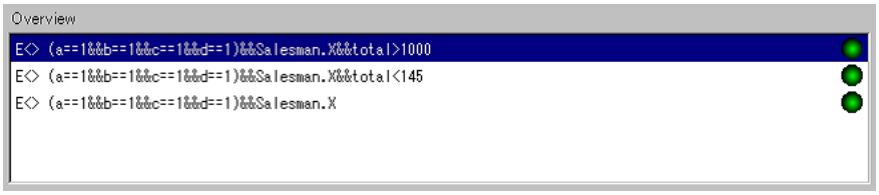

図 6　UPPAAL による検証実行画面 1

A へ遷移した瞬間を $t = 0$ とし、$t = 32$ の瞬間に状態 B へ遷移する可能性しかないことが分かる。

このようにシステムを表現し、UPPAAL で検証することになる。

6.2.3　検証結果

このプロセスを時間的検証した結果が図 5 の通りである。図 6 は検証画面の一部で、各検証項目について緑色のランプが点灯していることは、その項目について検証が成功したことを示す。例えば

$$E\langle\rangle(a == 1\&\&b == 1\&\&c == 1\&\&d == 1)\&\&Salesman.X$$

は各社を一回ずつ訪問した後に X 社に再び戻ってくるという状態が存在するかどうかを表し、これに緑色のランプが点灯しているということは、少なくとも 1 パターン以上その可能性が存在するということである。

ここで UPPAAL はその最短時間を調査することがオプションから選択可能で、実際に最短時間パターンを調査した結果が図??および図??である。図??において $total >= 144$ と書かれた部分が、各社を一回ずつ訪問して戻ってくるのに最低でも 144 分かかることを示し、図??からそのためには A 社、C 社、D 社、B 社の順に訪問すればよいことがわかる。また、このパターンのように 145 分未満で訪問を終えられるパターンが存在することは、図 6 の

$$E\langle\rangle(a == 1\&\&b == 1\&\&c == 1\&\&d == 1)\&\&Salesman.X\&\&total < 145$$

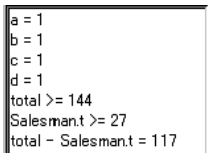

図 7　UPPAAL による検証実行画面 2

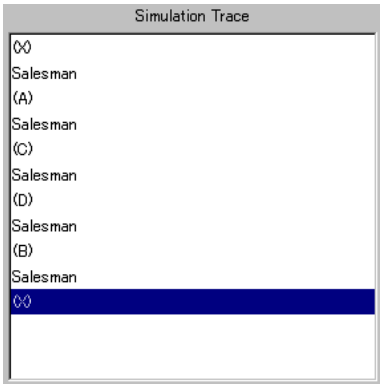

図 8　UPPAAL による検証実行画面 3

の項目に緑色のランプが点灯していることからも分かる。

同様にして適切に検証項目を記述することによって、どんなに時間のかかる訪問順番をとろうとも 167 分かければ訪問を終えられることも求められる。

既に示されているとおり、各社を一回ずつ訪問して戻ってくることが可能なことが UPPAAL によっても示されただけでなく、最低で 144 分で戻ってくることが可能なこと、最高でも 167 分かければ戻ってくることが可能なことも検証された。

UPPAAL ではこの他にも、あるタイミングで 2 つ以上のイベントが選択可能なときにどちらかの選択を優先的に選択することや、複数のプロセスを同時に実行させることも可能である。

7　割り込みルータシステムへの応用

CSP 記述や、FDR2、UPPAAL といういくつかのツールを総合的に応用する方法として、これらのツールを利用して割り込みルータシステムを実際に開発、検証した結果について述べる。

自動車を自動制御するリアルタイムシステムを考えたとき、このシステムは、車体前方に設置された衝突防止用のセンサーや、現在の速度やエンジンの回転数を感知する装置、さらにはハンドルやペダルといったいろいろなシステムから、絶えず適当な短い時間周期ごとにシグナルを受け取らなければならない。この割り込みルータシステムは、数百〜数千 μs 周期で発生するトリガーシグナル 60 本を、統合管理して上位モジュールへと伝える役割を持つものである。自動制御システムは、この結果生成されたシグナルをもとに制御を行うことになる。

これらのトリガーシグナルは、最終的に運動制御系（自動ブレーキなど）やマルチメディア系（カーナビゲーションなど）の制御に使うため、シグナルは最上位モジュールからハンドラシステムに引き渡され、ハンドラシステムから各制御系へ処理命令が下る。この処理にあたって各トリガーシグナルには、システムへの直結の度合いやシグナル受信周期の厳密さによって優先度が設定されている。例えば、受信周期の誤差をほぼ許さないものについては優先度を高く、多少の誤差が許されるものについては優先度を低くするといった具合である。

また、これらのトリガーシグナルは、シグナルの発生源やその用途によって大まかにグルーピングされる。例えば同じく安全制御に関係する衝突防止センサーやブレーキ支援システムからのシグナルは同じグループに、それらと関連のない空調に関連する車内温度センサーやカーナビゲーションのための GPS システムは違うグループにするといった具合である。この結果全体が大きく 8 つにグルーピングされ、このうち第 3 グループについては、さらに 6 つのサブグループに分けることで、合計 13 のグループが形成されている。

集約される過程でシグナルは干渉が引き起こされる可能性がある。基本的にシグナルは発生した順に逐次処理されるが、優先度の低いシグナルが発生した後十分短い時間間隔内で優先度の高いシグナルが発生した場合、これらのシグナルが処理される順番が入れ替わるというものである。

具体的な各シグナルの発生周期とそれらの優先度、そしてそれらのグルーピングについては付録を参照されたい。

従来こういったモジュールの開発は、大まかなシステムの設計をした後は実際のコーディングが中心になり、モジュールが完成した後の実行テストで予期せぬエラーが発生した場合、もう一度コーディングをやり直し、エラーが発生しなくなるまでこれを繰り返すということが多く行われてきた。この作業は半ば現場の経験と勘によるところが大きく、エラーへの対処方法、デバッグ終了の見切りなどは客観的なものが存在しなかった。これを設計の段階で陥る可能性のあるエラーを全て数学的に検証することによって、コーディングをするより先に、システムの正当性を示そうとしたのが新しい試みであり、今回の提案である。

7.1　CSP 記述を用いて

まず与えられた仕様をもとに CSP 記述によるモデル構築を行う。なお、紙面の都合上 8 本のみを処理するモデルを掲載したが、60 本の場合に比べても一般性を失わないことを予め断っておく。

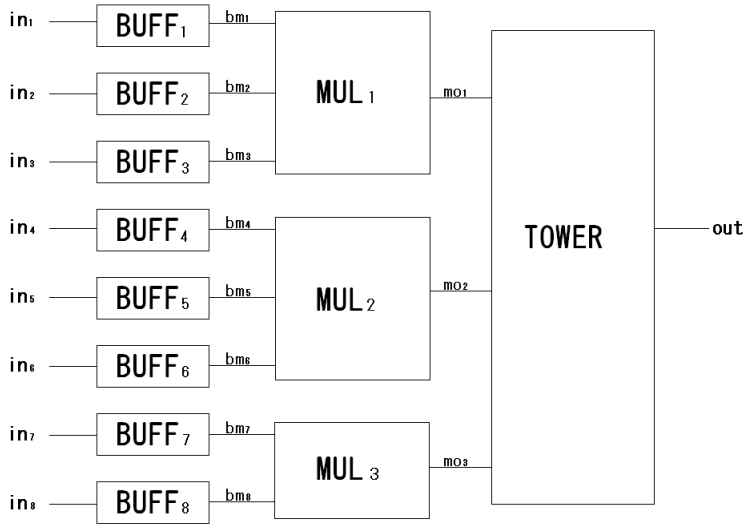

図 9　ROUTER1

$$
\begin{aligned}
BUFF_1 &= in_1?x \longrightarrow bm_1!x \longrightarrow BUFF_1 \\
&\vdots \\
BUFF_8 &= in_8?x \longrightarrow bm_8!x \longrightarrow BUFF_8 \\
MUL_1 &= \ \Box\ {}^{3}_{i=1} bm_i?y \longrightarrow mo_1!y \longrightarrow MUL_1 \\
MUL_2 &= \ \Box\ {}^{6}_{i=4} bm_i?y \longrightarrow mo_2!y \longrightarrow MUL_2 \\
MUL_3 &= \ \Box\ {}^{8}_{i=7} bm_i?y \longrightarrow mo_3!y \longrightarrow MUL_3 \\
TOWER &= \ \Box\ {}^{3}_{i=1} mo_i?z \longrightarrow out!z \longrightarrow TOWER \\
ROUTER_1 &= BUFF \parallel MUL \parallel TOWER
\end{aligned}
$$

なお、ここで

$$\Box\ {}^{3}_{i=1} bm_i?y \longrightarrow mo_1!y \longrightarrow MUL_1$$

は

$$
\begin{aligned}
MUL_1 = \ & bm_1?y \longrightarrow mo_1!y \longrightarrow MUL_1 \\
& \Box\ bm_2?y \longrightarrow mo_1!y \longrightarrow MUL_1 \\
& \Box\ bm_3?y \longrightarrow mo_1!y \longrightarrow MUL_1
\end{aligned}
$$

を表すものとする。その他も同様である。

7.2　FDR2 を用いて

さらにこのモデルの正当性を検証するため、FDR2 を利用する。

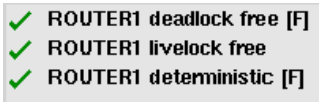

図 10　FDR2 による検証実行画面 2

先ほど構築したモデルの CSP 記述を、FDR2 で検証できるプログラムに変換すると以下の通りである。

```
nametype Ind1 = {1..8}
nametype Ind2 = {1..3}
nametype Ind3 = {4..6}
nametype Ind4 = {7..8}

channel in,bm : Ind1
channel mo : Ind2
channel out

BUFF(i) = in.i?x -> bm.i!x -> BUFF(i)
BUFFS = ||| i : Ind1 @ BUFF(i)

MUL(1) = [] i : Ind2 @ bm.i?y -> mo.1!y -> MUL(1)
MUL(2) = [] i : Ind3 @ bm.i?y -> mo.2!y -> MUL(2)
MUL(3) = [] i : Ind4 @ bm.i?y -> mo.3!y -> MUL(3)
MULS = ||| j : Ind2 @ MUL(j)

TOWER = [] i : Ind2 @ mo.i?z -> out ! z -> TOWER

ROUTER1 = ( BUFFS [|{|bm|}|] MULS ) [|{|mo|}|] TOWER
```

これを実際に FDR2 で検証した結果が図 10 の通りである。この結果からこのシステムは deadlock、livelock の可能性がないこと、そして deterministic があることにより、発生したトリガーシグナルに応じてイベントが正しく選択される、この場合ではシグナルがルータシステムに正しく受理されることが FDR2 を用いても検証されたことがわかる。

ここまでの検証で、システムが deadlock に陥ることがないことは検証できたが、これによってシステムの正当性が確証されるのは、プロセスの実行時間が全体の時間に比べて無視できるほど小さい場合のみである。リアルタイムシステムではこの実行時間がシステムの正当性に大きく関わるため、さらに時間的検証を行う必要がある。

7.3　UPPAAL を用いて

時間的制約条件を加えることによってシステムを拡張し、検証ツールの UPPAAL によって実験調査することで、既に作られたシステムがリアルタイムシステムとして制約を満たしているかを検証する。

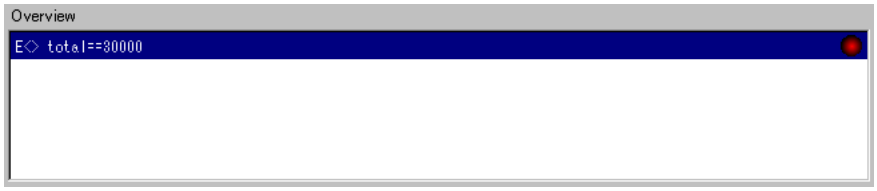

図 11　UPPAAL による検証実行画面 3

この割り込みルータシステムは、当研究室で開発された TPcore と呼ばれる CPU プロセッサ [5] での実装を前提としているため、このプロセッサで実際に稼働されなくてはならない。そのため、シグナルの発生周期を付録の通りに、各モジュールの演算速度は実際の TPcore と同じくシグナル 1 本あたり 22μs かかるものと設定し、実際に UPPAAL の総当たり演算で検証した。

この確認方法としては、十分長い時間が経過するまでにその間発生したシグナルに 1 本も処理漏れがなければ検証成功、1 本でも処理しきれずに処理漏れが起こるならば検証失敗という形をとった。

この検証の結果が図 11 からわかる。赤色のランプが点灯していることは、この項目について検証が失敗したことを示す。つまり現在の TPcore の性能では、今回設計した割り込みルータシステムを正しく運用できないことが検証からわかった。

続いて、今後の研究による TPcore の性能上昇の可能性を考え、各モジュールがシグナル 1 本の処理にかける時間をどの程度まで減らすことができればこのシステムは正しく運用できるかを確認した。この結果、シグナル 1 本の処理にかかる時間を約 5.1μs まで減らして検証を実行したところで緑色のランプが点灯した。つまり、TPcore においてシグナル 1 本の処理にかかる時間を約 5.1μs まで減らせばシステムは正しく運用できることがわかった。しかし、この値を実現するためには技術的な大躍進を必要とするため、すぐに実現するのは非常に難しい。つまり、従来の方法のままでは今すぐにはシステムの要件を満たせないことが導き出される。

この原因として、最終的に 60 本全てのシグナルが一つのモジュールに伝達される従来の方法では、結局最上位モジュールは処理しきれないほどのタスクを抱えてしまうため、並列処理の特性を活かせていないことが考えられる。

7.4　改良案

システムの要件を満たすための方法として、前述の通り、最終的にシグナルが集約されてしまうと並列処理の特性を活かせないので、シグナルを一つのモジュールに集約しない方法を考える。

そこで、先ほどのグルーピングに従い、互いに干渉する同一グループ内のシグナルのみを一つのモジュールに集約し、グループの違うシグナル同士が干渉し合わないようにする。発生源も用途も似通った同一グループ内のシグナル同士において干渉は必要不可欠だが、グループの枠を越えた干渉は不要である。また、ハンドラシステムに引き渡すにあたっても、安全制御系やマルチメディア系など各制御系に伴って複数のハンドラシステムとの並列処理を構成することによって、干渉の不要な命令を同時に実行することもできる。

新たに改良されたシステムの CSP 記述は以下の通りである。最大の変更点は、互いに干渉する必要のないシグナルは全て、最後まで全く干渉がないように分離されていることである。

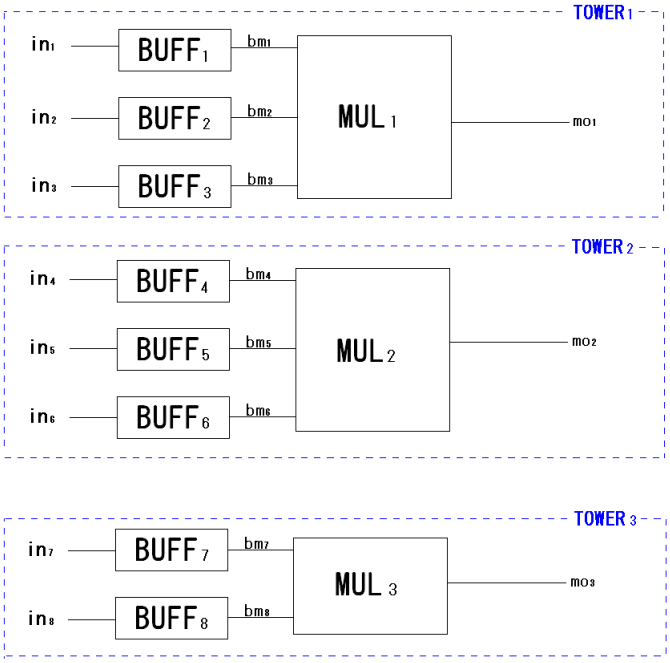

図 12　ROUTER2

$$
\begin{aligned}
BUFF_1 &= in_1?x \longrightarrow bm_1!x \longrightarrow BUFF_1 \\
&\vdots \\
BUFF_8 &= in_8?x \longrightarrow bm_8!x \longrightarrow BUFF_8 \\
MUL_1 &= \Box\, {}^{3}_{i=1} bm_i?y \longrightarrow mo_1!y \longrightarrow MUL_1 \\
MUL_2 &= \Box\, {}^{6}_{i=4} bm_i?y \longrightarrow mo_2!y \longrightarrow MUL_2 \\
MUL_3 &= \Box\, {}^{8}_{i=7} bm_i?y \longrightarrow mo_3!y \longrightarrow MUL_3 \\
TOWER_i &= MUL_i \underset{\{bm\}}{\parallel} \Big(\underset{j\in\{1..8\}}{|||} BUFF_j \Big) \\
ROUTER_2 &= \underset{k\in\{1..3\}}{|||} TOWER_k
\end{aligned}
$$

各 $TOWER_k$ がそれぞれ各制御系グループに対応していて、グループを越えたシグナルの干渉が起きないことが図 12 からわかる。

同様にモデルの正当性を FDR2 で検証するために、FDR2 で読み込める書式に変換する。

```
nametype Ind1 = {1..8}
nametype Ind2 = {1..3}
nametype Ind3 = {4..6}
nametype Ind4 = {7..8}

channel in,bm : Ind1
```

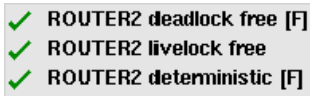

図 13　FDR2 による検証実行画面 3

```
channel mo : Ind2
channel out

BUFF(i) = in.i?x -> bm.i!x -> BUFF(i)
BUFFS(1) = ||| i : Ind2 @ BUFF(i)
BUFFS(2) = ||| i : Ind3 @ BUFF(i)
BUFFS(3) = ||| i : Ind4 @ BUFF(i)

MUL(1) = [] i : Ind2 @ bm.i?y -> mo.1!y -> MUL(1)
MUL(2) = [] i : Ind3 @ bm.i?y -> mo.2!y -> MUL(2)
MUL(3) = [] i : Ind4 @ bm.i?y -> mo.3!y -> MUL(3)

TOWER(j) = BUFFS(j) [|{|bm|}|] MUL(j)

ROUTER2 = ||| k : Ind2 @ TOWER(k)
```

図 13 からこのシステムの正当性を FDR2 で検証できたことを確認しておく。

更に UPPAAL を使って、先ほど同様各モジュールがシグナル 1 本の処理にかける時間をどの程度まで減らすことができればこのシステムは正しく運用できるかを確認した。この結果、シグナル 1 本の処理に対して約 95μs 以内の時間内で処理すればよいことがわかった。これはシステムの要件を十分満たすことができる現実的な数値である。

これにより、実際のコーディングより先にモデル設計の段階で、当初予定されていたシステムは要件を満たせないことを発見することができ、さらに改良したシステムは要件を満たすだけでなく、deadlock の起こらない健全なシステムであることを数学的推論とツールによる検証の両方から証明できたことになる。この CSP 記述に従って正しく制作された割り込みルータシステムは、実行テストの結果を待たずして、正しく動作する保証がされているといえる。

8 結論と今後の課題

8.1 研究結果

この論文で紹介したいくつかのツールを利用することにより、ターゲットとしている今回構築されたシステムは、並列処理システムとして deadlock を原理的に含まない健全なシステムであり、かつ 60 本のシグナルは単一モジュールによって管理することはできないが、適切にグルーピングし複数のモジュールによって管理することによって正しく動作することが期待されることの検証に成功した。

つまり、CSP 記述、FDR2、UPPAAL を総合的に用い、開発において実際にコーディングする以前の設計の段階で数学的にモデルの正当性を検証することに成功したことになる。

8.2 今後の課題

CSP 記述に時間的概念を加えた Timed CSP [2] という記述方法が開発されている。これは従来の CSP 記述における演算だけでなく、一定時間の状態保持やタイムアウトといった時間経過による状態遷移を記述することができ、それによるいくつかの演算も定義されている。

時間的概念を加えた記述方法としては他に DC（Duration Calculus） [3] などいくつかの方法がある。これらの記述方法に触れる中で、新たな側面からの推論が期待される。

9 まとめ

最後に本論文で述べてきたことをまとめる。

私は並列処理によるリアルタイムシステムを構築するにあたり、モデル設計の段階でシステムの正当性が数学的に証明されていることの必要性から、今回の CSP 記述をはじめとする研究に着手した。

システムは複雑になるほどそのデバッグは困難になる。こと並列処理システムにおいては、複数のプロセスが互いに連携しながらそれぞれ独立に動いているが、このプロセス一つ一つのデバッグだけでなく、システム全体が並列処理であることによる別の新たなデバッグが必要になるため、システム全体としてのデバッグは困難を極める。そのためどうしてもシステム制作終了後の実際の動作テストでのバグを元に場当たり的なデバッグが中心になってしまう。この方法ではある段階でデバッグが完了したことを明確に示す方法がなく、システムは不安を抱えたままになってしまう。

モデルを設計する手段として CSP 記述が挙げられる。この CSP 記述は、意味論を用いてシステムを形式的に数学的論証を可能にした記述方法である。CSP 記述を用いることによって、設計段階でシステムの正当性を演繹的に演算することが可能であり、このことがシステムの正当性を証明する理論的根拠となる。

CSP 記述による論証を裏付けるツールとして FDR2 が挙げられる。FDR2 は deadlock の可能性の有無を検証するソフトウェアツールである。ここまでの検証によって、deadlock の可能性は完全に取り除かれなければならない。

リアルタイムシステムにおいてはさらに時間的検討が必要となる。ここでは検証ツールとして UPPAAL を用いる。UPPAAL はシステムの条件設定に時間軸を加えることが可能であり、ある状態への到達可能性を網羅的に調査できるソフトウェアである。

本論文では CSP 記述とこれらの検証ツールを利用して、実際にモデル設計の段階でシステムの正当性を証明するということを行った。対象となったモデルは、数百〜数千 μs 周期で発生するシグナル 60 本を優先度に従って統合管理して上位モジュールへと伝える役割を持つ割り込みルータシステムである。このシステムは当研究室で開発した TPcore と呼ばれる CPU プロセッサに搭載することを狙っているため、この TPcore が処理可能な限界である、シグナル 1 本の処理あたり 22μs という処理速度を下回ってもシステムが正常に動作しなくてはならない。FDR2 による検証の結果、対象となった従来のモジュールは deadlock の可能性がないことは確認できたが、UPPAAL による検証の結果、シグナル 1 本あたりの処理時間を 5.1μs 以内にしなければならないという実現不可能な値が算出されてしまった。従来の一つのモジュールにシグナルが集約されるという方法ではリアルタイムシステムとしては要件を満たせないことがわかったため、シグナルをグルーピングすることでシステムの再構築を行い、改良した新たなシステムについて同様の検証を行った。その結果、シグナル 1 本あたりの処理時間を 95μs 以内にしなければならないという値が算出されたが、これは十分実現可能な値であるといえる。つまり改良されたシステムの正当性が証明されたことになる。

このことによって、モデル設計の段階でシステムの正当性を数学的に検証し、健全なシステムであるということを証明されたシステムを制作するという一例を示せたことになる。

参考文献

[1] Steve Schneider. *Concurrent and Real-time Systems: the CSP Approach.*

[2] Jim Davies, Steve Schneider. *A brief history of Timed CSP.*

[3] Fonathan Bowen, Martin Fränzle, Ernst-Rüdiger Olderog, Anders P. Ravn. *Developing Correct Systems.*

[4] 情報処理学会 ソフトウェア工学研究会『モデル検査ツール UPPAAL を使った時間制約の検証』

[5] 田中　誠：東京都立大学　修士論文『並列処理プロセッサの設計開発とその FPGA への実装について』

謝辞

本研究に取り組む機会を与えていただき、時に厳しい叱咤激励を、時に優しい助言を与えてくださった指導教官の福永力教授に深く感謝いたします。本研究のきっかけを与えていただき、研究方法などの議論にお付き合いいただいたエミネントの松井和人氏、SmartScape の吉田隆氏にも深く感謝いたします。また、研究初期から長くともに研究を続け、互いに大きく刺激をすることになった山本聡、田中和人両氏にも感謝しております。

また公私にわたって時に私を励まし、時に支えとなった多くの友人と家族にこの場を借りて感謝したいと思います。

10 付録

10.1 割り込みルータシステムにおいて発生するシグナル

groop	signal-name	priority	Trigger	period
1	INT-000	1(Highest)	time	1ms
	INT-001	2	↑	4ms
2	INT-002	3	↑	2.5ms
	INT-003	4	clank-angle	30CA
	INT-004	5	↑	30CA
	INT-005	6	↑	720CA
	INT-006	7	↑	10CA
	INT-007	8	↑	720CA
	INT-008	9	↑	240CA
	INT-009	10	↑	240CA
	INT-010	11	↑	240CA
	INT-011	12	↑	240CA
3-1	INT-012	13	↑	90CA
	INT-013	14	time	2ms
3-2	INT-014	15	crank-angle	45CA
	INT-015	16	↑	45CA
	INT-016	17	↑	45CA
	INT-017	18	↑	45CA
	INT-018	19	↑	45CA
	INT-019	20	↑	45CA
	INT-020	21	↑	45CA
3-3	INT-021	22	↑	45CA
	INT-022	23	↑	45CA
	INT-023	24	↑	45CA
	INT-024	25	↑	45CA
	INT-025	26	↑	45CA
	INT-026	27	↑	45CA
	INT-027	28	↑	45CA
	INT-028	29	↑	45CA
	INT-029	30	↑	45CA

groop	signal-name	priority	Trigger	period
3-4	INT-030	31	crank-angle	45CA
	INT-031	32	↑	45CA
	INT-032	33	↑	45CA
	INT-033	34	↑	45CA
	INT-034	35	↑	45CA
	INT-035	36	↑	45CA
	INT-036	37	↑	45CA
	INT-037	38	↑	45CA
3-5	INT-038	39	time	40ms
	INT-039	40	↑	40ms
3-6	INT-040	41	crank-angle	360CA
	INT-041	42	↑	360CA
	INT-042	43	↑	90CA
	INT-043	44	↑	90CA
	INT-044	45	↑	90CA
4	INT-045	46	↑	10CA
	INT-046	47	↑	30CA
	INT-047	48	↑	360CA
5	INT-048	49	time	1ms
	INT-049	50	↑	1ms
	INT-050	51	↑	1ms
	INT-051	52	↑	1ms
	INT-052	53	↑	1ms
	INT-053	54	↑	1ms
6	INT-054	55	↑	1ms
	INT-055	56	↑	4ms
7	INT-056	57	crank-angle	90CA
8	INT-057	58	time	40ms
	INT-058	59	↑	4ms
	INT-059	60(lowest)	↑	4ms