ZIPC の適用によるシステム開発事例

～ 火報システムの開発を通じて ～

大景 聡

■ はじめに

組み込み型システム開発において、新規商品開発やある程度以上の規模のシステムになると、開発納期やコスト・品質面での問題が発生した場合その原因がソフトウェアである可能性が高いようです。

対象的に、ハードウェアの開発においてはかなりの新規要素が無い限り、大幅な開発納期の遅延や開発コストアップ、品質面での問題点が生じる可能性が低いように思われます。なぜでしょうか？

ハードウェアの開発では、全体の機能を達成するためのステップがソフトウェアと比較すると確立されています。基本ブロックの設計、タイミングの設計、回路図の作成、部品の選定・手配、P版設計、環境試験等の評価などがそれにあたると思います。それぞれのステップで、いろいろなスキルを持った人が前工程の流れを引継ぎ各工程の作業をこなすことが可能です。全技術者共通の設計書（回路図）があることもそうでしょうし、ノウハウがチップとして部品化されていることも見逃せません。

どのようにすれば、ソフトウェア開発が個人のノウハウによらずにオープンとなり、資産化及び再利用されていくのでしょうか？ZIPC による開発手法と開発支援の導入を通じて考えてみたいと思います。

■ ZIPC 導入の経緯

以前、組み込みの商品開発を行った時にソフトウェアのバグの原因を調査し集計したところ、組み込みシステムのソフトウェアの設計ミスにおける原因のトップは、状態遷移ミスであることが判明しました。それまで、私どもではソフトウェアの設計書に通信以外で状態遷移というものを記述していませんでした。それは、構造化言語で記述しているのだから順次と分岐、繰り返しの処理さえ記述しておけば良いという考えと、状態遷移図そのものに少し疑問を持っていたからです。要求仕様から正確な状態遷移図を作りだすためには相当な能力が必要であることや、DRにおいて、作成された状態遷移図の抜けを設計者以外の人が指摘するのが難しいといった疑問です。

しかし調査結果から、ソフトウェアの不具合を減少させるためには、通信以外のソフトウェアでも状態遷移を記述する必要があることがわかります。そのためには、状態遷移表を記述することが最適だという結論に達しました。状態遷移表の利点として、記述時にすべての場合を考慮しなければならないために、状態の抜けが減少するということがあげられます。当初はお絵描きツールにて状態遷移表を記述していましたが、ZIPC が状態遷移表を作成するのに便利であることを知り、導入しました。

■ 応用分野の説明

今回、ZIPC を弊社の火報システムの開発に導入しました。火報システムとは、ビル等の建物で

火災を検知し、瞬時に火災を報知し防火・防煙を行うシステムです。システムの特徴としては、組み込み型としてはソフトウェアの規模が大きいこと、国の検定で定められた基準を満たすために高い品質が要求されることがあげられます。

■ 上流における分析モデル作成時点における応用

システムを分析する段階で、内部構造や状態遷移とデータフローを記述する必要があります。特に、リアルタイム系の分析には状態遷移が非常に大きなウェイトを占めてきます。今回、状態遷移の記述に状態遷移表を用いました。状態遷移図を書くのが一般的なのですが、状態遷移図では仕様抜けや矛盾を抽出するための分析で、記述漏れが発生しやすいという本末転倒な結果を招く可能性があります。また、状態遷移表で階層化の概念を用いれば、複雑な遷移の記述も可能です。ここで、状態遷移表を記述するエディタとして ZIPC エディタを使用しました。音響部にて記述した状態遷移図のサンプルの一部を**図1**に示します。この段階で実装を無視したシステムの動作記述が可能です。この段階から ZIPC エディタを使用しておくと。後の設計以下の工程が楽になります。

■ 設計段階における応用

設計段階では、システム分析結果に対してハードウェアや RTOS の制限・制約によるシステムの分割及びタスク分割、それらに基づいた状態やイベントの付加等を行えば、理論上問題なく設計できるはずです。今回の開発の中でシステム全体の中の音響部や表示部を取り上げ、ZIPC によって設計以降の工程を行うことにしました。音響部は音響タスクとして、表示部は画面タスクとスイッチタスクの2タスク構成にして、RTOS 上で実現しました。

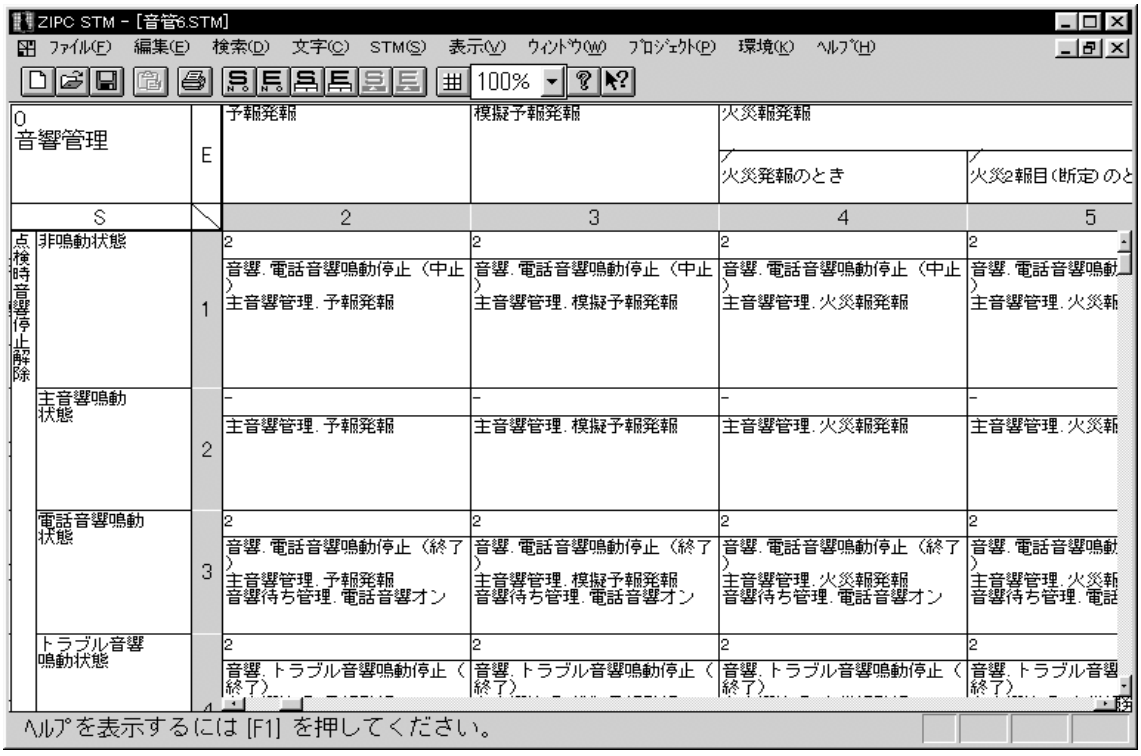

図1　状態遷移表の例

1）各タスクの概要

各タスクの動作の概要を**図2**に示します。

・音響タスクの概要

このシステムの音響は20種類程度の出力パターンがあり、それを発生イベントによってどのパターンの音響にするかを決めて、音声合成 IC を制御し出力します。

・画面タスクの概要

画面タスクは液晶のLCDにて表示を行い、発生イベントによって様々なシステムの内部情報をユーザーに伝える役割を担っています。また、ユーザーからシステム全体に対して制御を行う場合に、画面タスクを通じて他タスクにイベント送信を行います。画面表示するデータは、発生イベントによって受信したデータを元に約 30 種類のテーブルからデータを検索し、表示することになります。

・スイッチタスクの概要

スイッチタスクは画面内部に表れるスイッチの状態を保持し、他タスクにイベントを通知する役割を持っています。画面部から制御されることになります。

2）STM の作成

音響タスク、画面タスク、スイッチタスク共に、分析レベルで作成された状態遷移表から内容が大きく変更になることはありませんでした（そういった題材を選んだつもりです）。従って、分析レベルの STM から ZIPC の設計レベル

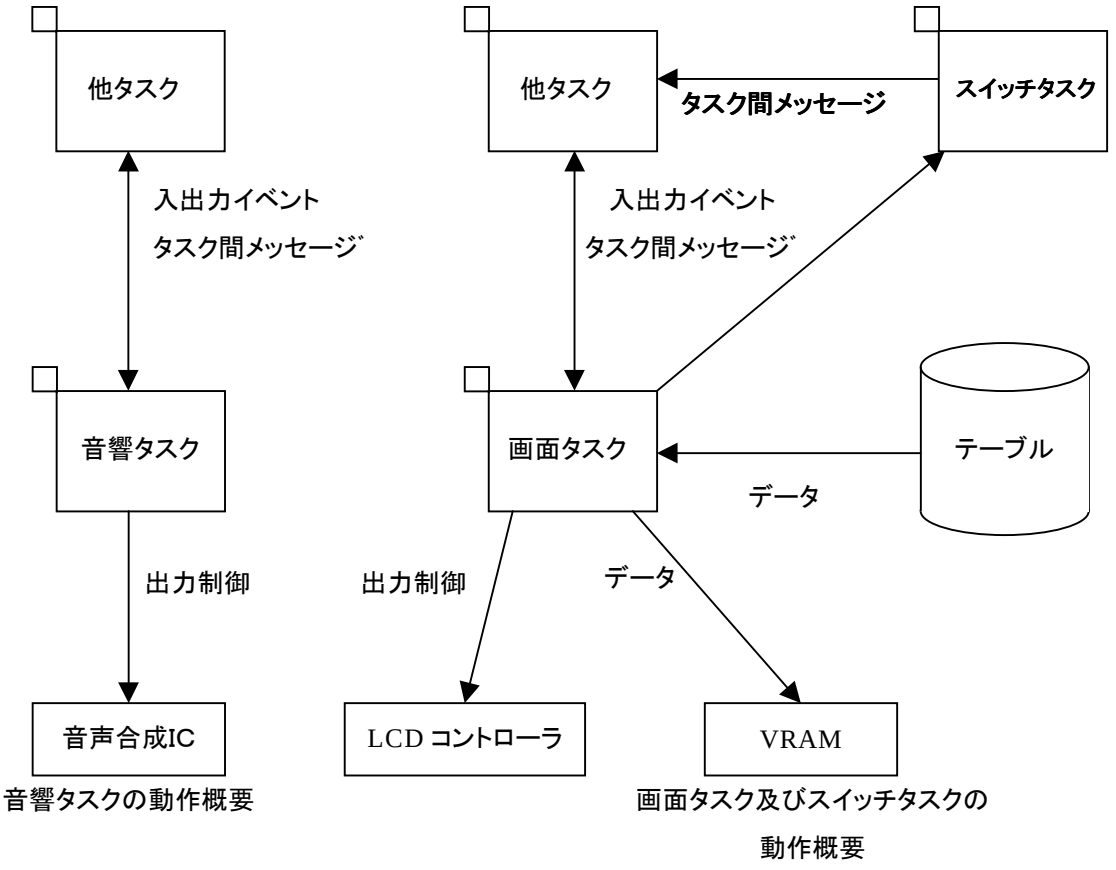

図2 各タスク動作概要図

に変換する工程を以下のように行いました。

・分析レベルで自由に記述された日本語を、ZIPC シミュレータや ZIPC ジェネレータで規定された文法へ置き換え

・日本語で記述されたイベントを設計レベルのタスク間メッセージ記述に置き換え

・ファンクションの抽出

・ファンクション詳細の記述(FNC 設計書)

・置換情報の作成

作成された STM のサイズを**表1**に示します。

3)論理シミュレーションとコードジェネレーション

論理シミュレーションは実機動作前にソフトウェアの論理デバッグができるため、ハードウェアと独立してソフトウェア開発スピードを上げるために、品質を向上させるという意味で非常に大きな効果があります。ハードウェア開発前に、かなりのレベルまでのソフトウェアのデバッグが可能です。コードジェネレーションは、コーディング費用を削減できると共にケアレスミスも防げるという意味で重要です。今回はこの2つの機能を試みました。各タスクの自動生成によるステップ数等を**表2**に示します。

① 音響タスクの場合

音響タスクの場合、STM 内の全セルに対して論理シミュレーションを行い、その結果を MSC ファイルの形式で保存し、その結果を全チェックするということを行いました。音響タスクにはデータフローが無く、状態遷移のみで記述できる性質のソフトウェアであるため、STM の論理チェックを行えば個々のファンクションレベルの試験と置換情報のチェック以外での実機テストを削減することが、理論上可能です。STM のセル数は他のタスクと比較してさほど多くはないのですが、それでも MSC ファイルで 174 枚あり、全チェックを行うのに2日程要しました。しかし、実機チェックを行うことを考えればはるかに速い時間で全パターンをチェックすることが可能です。

タスク名称	STM枚数	イベント数	状態数	ユーザー関数数
音響タスク	4	88	46	20
画面タスク	36	640	310	937
スイッチタスク	65	503	466	219

表 1 作成 STM のサイズ

	音響タスク	スイッチタスク	画面タスク
ROMテーブル形式	標準型	処理抜粋型	処理抜粋型
総ステップ数	4890	33531	72285
自動生成率(%)	97.5	74.9	55.7
自動生成時間	1分以内	約2時間	約2時間半

表2 各タスクのソース自動生成状況

② スイッチタスクの場合

スイッチタスクの場合も状態遷移でほぼ記述できるソフトウェアですが、STM の枚数からわかるようにサイズの大きなタスクとなっています。ただし個々の STM のサイズは小さいために、見通しのよい設計になっているといえます。

スイッチタスクの論理シミュレーションは、ZIPC Ver.4.0w の制限上タスク全体で行うことはできませんでした。従って、論理シミュレーションの効果はあまり無いと考えられます。ただし、STM はサイズが小さいため見通しが良く、規則的であるために状態遷移によるミスはかなり低いと予想されます。

コードジェネレーションに関しては、これもサイズが大きいせいか音響タスクとはかなりの違いがあり、2時間程度を要します。この数字は、デバッグ段階に入って設計書修正後すぐにソースが欲しいような局面にて不自由します。このあたりがツールとしての改善すべき点かと思います。

③ 画面タスクの場合

画面タスクの場合、サイズが大きく状態遷移も多いのですが、データフローが中心となってくるようなソフトウェアになります。データフローは STM 上関数として隠されてしまい、状態遷移表を記述しただけでは見えなくなってしまいます。今回は、開発期間の関係もあって FNC 設計書にて処理のみを記述していきました。

論理シミュレーションは、やはりサイズの制限で全体を行うことができませんでした。

4）評価結果

現状で評価工程における上記タスクの不具合件数は、画面タスクが 99%以上で他の2つのタスクは1%未満となります。画面タスクの不具合の原因も、8割以上はファンクションの内部に集中しています。それに対して、音響タスクやスイッチタスクは置換情報のケアレスミス程度しか発生していません。この結果は、画面タスクのサイズが大きいことを考慮したとしても、画面タスクにおける設計方法に問題点があるということと、音響タスクとスイッチタスクにおける設計には ZIPC が非常に適しているということを示しています。

それと同時に、リアルタイム性が強い、すなわち状態遷移にてほぼ記述できる性質のソフトウェアに対しては ZIPC にて提唱されている手法がかなり適しているが、データフローの多いソフトウェアに関しては、何らかの対策が必要であることも示していると思います。

5）開発工数と成果物について

ZIPC を導入することによって工数を削減できるかどうかですが、音響タスクは以前にほぼ同様の仕様にて作成した例があるので、それを元に比較することができます。以前の結果と ZIPC 導入後の結果を比較すると、開発完了までの期間はほぼ同程度という結果になります。**表3**に開発期間の比較を示します。

ただし、同じ期間で作成された成果物として、従来型の開発であれば処理のみを記述した設計書とソースという事になり、修正等により設計書を元にソースを見直すということを行う必要があります。一方、ZIPC の場合は状態遷移表と FNC 設計書、置換情報ということになりますから、修正等においても抽象度の高い書類を修正していくため誰でも修正できるし、修正点が明確になるというメリットがあります。

<table>
<tr><th></th><th>1W</th><th>2W</th><th>3W</th><th>4W</th><th>5W</th><th>6W</th><th>7W</th><th>8W</th><th>9W</th></tr>
<tr><td>従来型の開発</td><td colspan="2">分析</td><td colspan="3">設計</td><td>実装</td><td colspan="3">単体テスト</td></tr>
<tr><td>ZIPCを導入した場合</td><td colspan="3">分析</td><td>設計
STM作成</td><td colspan="2">シミュレーション</td><td colspan="2">置換情報生成
実装
単体テスト</td></tr>
</table>

表3 開発期間の比較

また、同様のアプリケーションの開発時にも、ずっと少ない修正で再利用できると思います。論理シミュレーションによる確認についても、全セルのチェックを行っているわけですから、試験項目も STM 作成時に出来上がっている事になり、従来型と比較するとずっと品質の良いソフトウェアになっているはずです。

現に、ZIPC導入による音響タスクの不具合は、状態遷移に関する物はありませんでした（置換情報の作成ミスの不具合が3件発生しました）。

■ 導入効果と問題点

以下に、今回の開発における ZIPC 導入についてまとめます。

1） 設計手法としての効果

状態遷移表を階層化して記述するということは、設計者のレベルを比較的問わなくなるという意味において設計段階で非常に大きな意味を持っていると考えられます。私どもでは、別の理由で一部同じものを状態遷移図にて記述していたのですが、遷移表にて仕上がったものと比較すると抜けがかなり多いことに気づかされます。レビュー段階においても、設計者の思考したレベルまで把握することができるので漏れがなくなります。ただし、仕様設定者に対するレビューにて概略を説明するという意味では、状態遷移図の方が良いかもしれません。このあたりは臨機応変に使い分ければ良いと思います。

状態遷移表はソフトウェアの抽象化にもかなりの効果があると考えられます。例えば、一度固有のアプリケーションを STM で記述してしまえば、動作は同じで OS や CPU を変更した場合のソフトウェアの生成については、置換情報やファンクションの変更にとどまると考えられます。さらにオブジェクト指向で設計されていれば、アプリケーションレベルより独立性が高まると思います。

2）設計手法としての問題点

弊社の場合、ソフトウェアの規模が大きくなるにしたがって、ソフト開発は外部の力に頼らざるを得ないのが現状です。その場合、階層化された状態遷移表を書くという作業がネックとなることがわかりました。現状、状態遷移表を書くという設計手法が浸透しているわけではなく、階層化にはある程度の慣れが必要なようです。そのため状態遷移表の作成をサポートする必要がありますが、そのためには状態遷移表を用いた設計手法の一般的な普及が望まれます。

次に弊社にて応用した例でわかるように、データフローが重視されるような性質のソフトウェアに関しては、状態遷移だけですべての内容を記述できないことがわかります。こういった性質のソフトウェアは、システムの規模が大きくなった場合に特に出現するように思われます。マンマシン系やシステムのメインとなって他タスクを制御するようなソフトにそのような

例を見ることができます。状態遷移と DFD の組み合わせで表現する必要がありそうです。

3）論理シミュレーションについて

PC 上でソフトウェアの論理シミュレーションができるということは、ハードウェアレスの環境でデバッグすることが可能であることを意味しており、後工程を大幅に短縮できる可能性を秘めています。実機上で全ケースの試験をするのは大変なことですし、論理シミュレーション上では実機上ほどの苦労を伴いません。また、今回の例でも論理シミュレーションを実施した音響タスクでは、実機レベルでの試験以前に状態遷移のミスはほぼ無くなりました。

今回の事例では、タスク全体のシミュレーションができなかったことが残念でした。ツール側の制限によるものでしたが、ツール側からタスクサイズを規定するようなことをしても良いのではないかと思いました。この点は次期バージョンで改善されるようです。

4）ソースコード自動生成について

設計書からコードが自動生成されて、その間に人為的なミスが介されないということは、プログラミング費用を削減するだけでなく修正やメンテナンス時に威力を発揮すると思われます。現に、今回の開発においても複数人数でデバッグによる変更や修正を行った場合に、かなりのスピードで変更を加えることができました。

残念だったのは、ジェネレーションに要する時間です。置換情報が増えるとジェネレーションが遅くなるようなのですが、このあたりも次期バージョンで改善されるものとして期待しています。

■ 終わりに

ソフトウェアの設計をサポートし、ソフトウェアの自動生成及び論理シミュレーションを行えるということは、ソフトウェア開発工数をより上流へシフトさせ、ハードウェアへの依存度を低くします。このことは、組み込み型のソフトウェアの開発工程を前倒しにできる可能性を示しています。アプリケーション特有の状態遷移を STM に記述することにより、設計書レベルの再利用が進むのではないでしょうか？

ZIPC は、状態遷移の多いリアルタイム系のシステムのソフトウェアを資産化・再利用・自動化、そして複数の開発者が共有・分業していくための切り口の一つといえると思います。

今まで同様、実戦に使える CASE ツールとして、発展していって欲しいものです。

[おおかげ さとし]