目次

特別寄稿

事例発表

ソリューション

社 内 寄 稿

ZIPC の″今″と″これから″

キャッツ(株) 取締役 副社長
渡辺 政彦
敬称略

広告

(株)オージス総研

ガイオ・テクノロジー(株)

日本電気(株)

富士通(株)

(株)東芝

状態遷移設計の基礎

神奈川大学 理学部 情報科学科
野口 健一郎

1．はじめに

初めはハードウェア設計用の技法だった状態遷移図や状態遷移表は、通信用ソフトウェアの設計にも広く用いられてきた。また、OS の設計への適用も試みられ[1]，その後さらに広い分野のソフトウェア設計にも適用されるようになった。[2] [3]

ここでは、状態遷移技法を用いた設計、すなわち状態遷移設計の基礎を、やや理論的に述べてみたい。それにより、状態遷移設計の特性が分かり、またその有用性がより明確になることを期待する。

2．外部仕様と状態遷移技法

設計の重要な過程として、外部仕様を明確に、漏れなく決め、またそれをきちんと表現することが必要である。なお、外部仕様には

- ***入力データと出力データの形式（構文規則＝シンタクス）***
- ***入力データと出力データの関係（意味規則＝セマンティクス）***

の二つの側面があることに注意しよう。

外部仕様を考え、記述するための手法をみてゆく。

（1）シーケンス図

外部仕様を考え、記述するとき、内部の構造、設計に立ち入らずにできることが望ましい。外部仕様を最も「外部的」に記述する手法として**シーケンス図（時系列図）**がある。[2]

これは設計対象物とその利用側とをそれぞれ縦棒で表現し、その間でのデータのやり取りを時系列的に表現したものである。（図１）

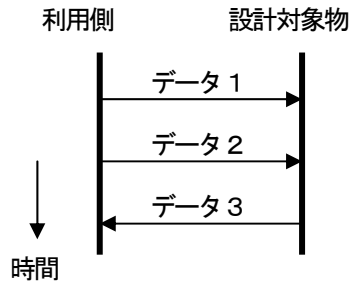


図１　シーケンス図

シーケンス図の利点は、

- ***入力と出力の関係、すなわちセマンティクスが、内部の動作・処理に関係することなく、わかりやすく表現される。***

しかし、根本的な欠点として、

- ***シーケンス図で記述できるのは有限個のシーケンスのみである。無限のシーケンスを持つ外部仕様（普通はそうである）のすべてを記述することは、理論的にできない。***

このように限界はあるものの、

- ***（シーケンスの種類が無限個であっても）シーケンス図を用いて外部仕様の主要なシーケンスを記述することは、理解を助け、特に初期の設計には有用である。***

（2）状態遷移図（表）

コンピュータの大きな特徴は記憶を持つことである。設計対象物が記憶を持つとき、入出力のシーケンスの種類は理論的には無限個になる。そこで、**内部状態**の概念を導入することにより、無限の入出力シーケンスを有限の表現として記述できるよう

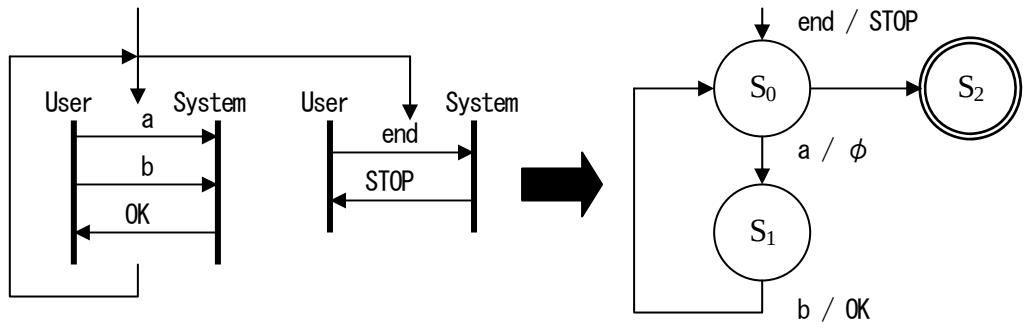


図２　繰返し型シーケンスを状態遷移図で記述

にしたものが**状態遷移図**（または**状態遷移表**）である(図２)。ただし、これで表現できるものは理論的には**有限状態機械**であり、内部状態数が有限個である、という制約がある。

状態遷移図の特徴を挙げる。

- *シーケンスの種類が無限個であっても、シーケンスに繰返し型の規則がある場合に適用でき、入出力の関係、すなわちセマンティクスがきちんと記述できる。*
  - *図であるため、個々のケースごとにそれにふさわしい構造的な表現をとることが可能である。外部仕様の構造について理解しやすく、外部仕様を考え、設計する道具として有用である。シーケンス図の次に使う道具としてよい。*
  - *内部状態数が無限の場合、または非常に多い場合には適用できない。ただし、主要な外部仕様は限られた数の内部状態で扱える場合も多く、その場合には適用できる。*

状態遷移表の特徴を挙げる。

- *状態遷移図の第１点と同じ。*
- *表であるため、内部状態と入力を組み合わせたすべてのケースについての動作が記述できる。したがって、外部仕様の細部まで明確になる。細部にわたる外部仕様を考え、設計する道具として有用である。*
- *状態遷移図の第３点と同様。さらに、次節で述べる拡張状態遷移表を用いることにより、これがやりやすい。*

３．内部設計と拡張状態遷移表

内部状態数が無限の場合、状態遷移図や表で完全な外部仕様を記述することはできない。また、内部状態数が非常に多い場合も、実用上だめである。記述できないのは外部仕様のうちのセマンティクスである。**これらの場合には、外部仕様を完全に記述するには、内部設計を進めて、セマンティクスを「内部的」に記述するしかない。**すなわち、外部仕様設計と内部設計が同時に進むことになる。

内部状態数を、実用上取り扱える範囲にまで減らす手法は二つある。一つは、設計対象物を複数の部分に分割することである。各部分ごとに、その部分の外部仕様を状態遷移技法で表す。10,000 個の内部状態を持つものをうまく２分割できた場合、各部分はたかだか100 状態ですむ。なお、複数の部分への分割に伴い、もともとの外部仕様には無かった各部分間の入出力(内部入力、内部出力)が出てくる。

もう一つの手法は、変数（内部変数）の導入である。１ビットの変数（フラグ）の導入により、うまい場合は状態数が半減する。４桁の１０進数の導入で、うまい場合は状態数が１万分の１に減る。

変数の導入により、状態遷移表は次のように拡張される。

- *出力を記述する部分が、変数の操作も含んだものとなる。*
- *状態遷移の判断に、変数の値の判定が必要に*

なる。（もともと多数の内部状態があったものを変数に縮退したのだから、これは当然である。）

このように変数を扱えるように拡張した状態遷移表を**拡張状態遷移表**と呼ぶ[2]（表1）。なお、変数は外部仕様には直接現れない、「内部的」なものである。

以上の二つの手法を組み合わせて用いることにより、内部状態数を実用上取り扱える範囲にまで減らすことができ（一つにまでなってもかまわない）、外部仕様を隅々まで明確にし、完全に記述することができる。それと同時に、内部設計のかなりの部分も進んだことになる。

この段階までの設計は、原理的には設計対象物がハードウェアかソフトウェアかには関係しない。ソフトウェアで実現するためには、次の設計段階として、状態遷移表ないしは拡張状態遷移表を、プログラムへと変換（理論的にはシーケンシャル処理へと変換）する作業が必要になる。[2]

4．オブジェクト指向＋状態遷移設計

最後に、近頃広まってきたオブジェクト指向に基づく設計と状態遷移設計の関係に触れたい。オブジェクト指向の考えはかなり以前からあるが、オブジェクト指向言語としてC++が広まり、またJava言語が登場するに至り、主流になりつつある。

オブジェクト指向の基本は、システムを複数のオブジェクトに分割し、かつオブジェクト間のインタフェースを明確にする、というシステム構成法の面にある。これは妥当な考え方である。しかし、オブジェクト指向の場合、オブジェクト間のインタフェース（それはオブジェクトの外部仕様である）としてシンタクスだけに注目し、セマンティクスは無視する傾向にある。（例えば、Java言語のinterfaceや、分散オブジェクト用のインタフェース定義言語（IDL）など。）

設計を進めるためには、むしろセマンティクスのほうが重要であり、またそれを完全に定義することの難しさは前節までの説明で理解されたと思う。そして、インタフェースのセマンティクスを記述する手法として、状態遷移技法が適用できることにも気付かれるだろう。すなわちオブジェクト指向設計と状態遷移設計は、組み合わせて使うことができるものである。

表1　拡張状態遷移表（一つの記述形式）

入力＼内部状態 / 条件		・・・	S_i	・・・
：				
I_a				
I_h	変数の値の判定１		変数の操作 出力 → S_j	
	値の判定２			
：				

オブジェクト指向設計のためのモデリング技法として UML (Unified Modeling Language) が注目されてきている。その中には、sequence diagram (シーケンス図) や statechart diagram (状態遷移図) も含まれている[4]。それらの必要性および位置付けを、本稿の読者は容易に理解されよう。

5. おわりに

本稿では状態遷移設計の基礎をやや理論的に説明することを試みた。その背景となる理論はオートマトン理論である。状態遷移設計は理論的な基礎がしっかりしており、かつ実用性もあるものとして、着実に使われていこう。オブジェクト指向設計とも矛盾することなく、組み合わせて使えるのは、理論的な基礎がしっかりしていることの証ともいえよう。

参考文献

[1] 野口健一郎、元岡達:オペレーティング・システムの記述に関する一考察、情報処理、Vol.14, No.2, pp.98-105 (1973)

[2] 野口健一郎:ソフトウェアの論理的設計法、共立出版 (1990)

[3] 渡辺政彦:拡張階層化状態遷移表設計手法 Ver.2.0、東銀座出版社(1998)

[4] J. Rumbaugh, I. Jacobson, and G. Booch: The Unified Modeling Language Reference Manual, Addison-Wesley (1999)

(のぐち　けんいちろう)

ソフトウェアエンジニアリングのすすめ　ー原理と潮流と予測と課題ー

(株)日立製作所 システム開発研究所
大槻 繁

はじめに

製品としてのソフトウェア開発に携わっておられる方々は、現世のご利益というか、適切なコードを効率よく開発するための支援環境に興味があると思いますが、ここでは、大局的な視点からソフトウェア生産技術について眺めてみようと思います。

組み込みシステムの分野においても、ソフトウェアの占める役割は、ますます重要になって来ています。図1はLSIの集積度と開発コード量の生産性に関する状況を単純化して示したものです。

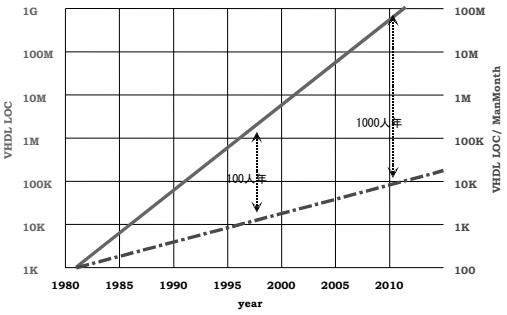


図1 LSIの集積度と開発コード量の推移

LSIの集積度は、2000年現在で10^7～10^8 Logic Tr／Chipにまで達しています。これはVHDL記述で数MLOC（百万ステップ）に相当する分量です。設計の生産性は, RTLレベルの記述からLSIレイアウトデータまで、種々のツールを駆使したとしてもせいぜい1KLOC（千ステップ）／人月であるから数百人年の工数がかかっていることになります。

集積度の向上は、2010年には10^9 ～10^{10} Logic Tr／Chip程度になると予想されており、これは年率で58%の伸びになります。一方、生産性は、記述の抽象度を上げることによって過去には年率で 21%の伸びを示して来ましたが、最近では配線遅延がゲートの遅延を越えてしまうという論理合成上の問題があり、このままでは、せいぜい 10KLOC／人月程度にとどまってしまうと言われています。

エンジニアリングというのは、常に原理的な限界を見極め、その中で、製品開発というミッションを満たすために解を見いだす活動ですから、こういった規模に関する推移は、時々、確認しておくことをおすすめします。

ソフトウェアエンジニアリングとは

ソフトウェアエンジニアリングとは、ソフトウェア開発に関わる広義の言語活動のことを示しています。

図２に示すように、ソフトウェア開発では、記述(プログラミング言語, 仕様記述言語のみならず, 図式や, ユーザインタフェース上の規約等) を媒体として用いるということは普遍的であり、これに関わる諸活動を体系化し、技術として蓄積することがソフトウェアエンジニアリングの目的と言えるで

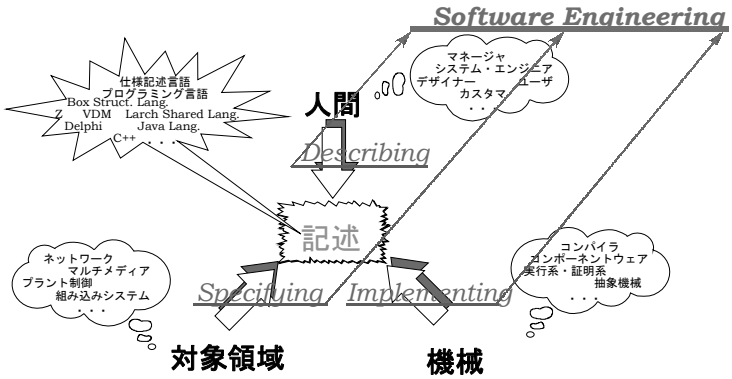


図2 ソフトウェアエンジニアリングに関する３つの世界

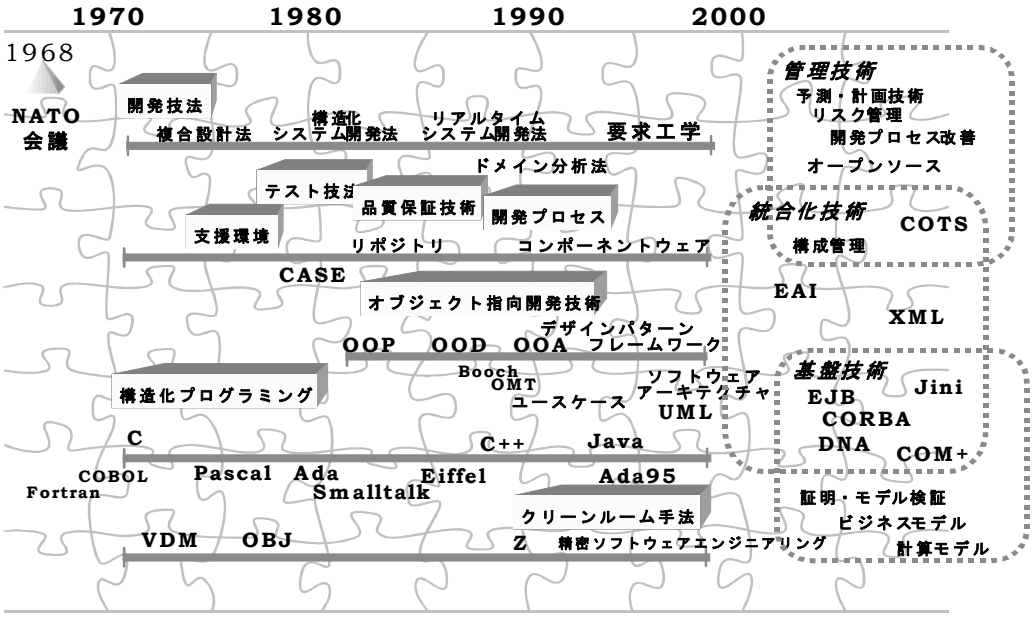


図3 ソフトウェアエンジニアリングの変遷

しょう。

(1)対象領域:Specifying/Modeling(言語設計論)

対象領域の問題をモデル化し、それを記述としてまとめることに相当しています。アルゴリズム、シミュレーション、科学技術計算、事務処理、制御分野といった問題領域を見据えてプログラミング言語は設計されてきました。計算機による実行特性がない言語では、伝達を目的とした図式言語、厳密な検証を目的とした仕様記述言語などもあります。言語を設計することは、対象領域の特性を見極め、それをモデル化してバランスよく仕上げる創造的活動です。

(2)機械:Implementation/Compiling(実装論)

言語を機械(計算機)上で処理・実行・検証等を行えるように実現することに相当しています。狭義の計算機実装論は、いわゆるコンパイラ技術である言語の構文解析技術や最適化技術を示しています。この領域で重要な事柄は、計算機の構造や処理方式、アーキテクチャを考慮し、それ等の資源をどのように有効に扱うかということです。対話型や複数のツールから構成される統合環境、さらには、ソフトウェアの構造や部品、ライブラリなども考慮する必要があります。

(3)人間:Description/Programming(言語用法論)

人間が言語を使って記述(伝達・定義)することに相当しています。初期のプログラム書法に始まり、最近の仕様記述法に至る技術で、狭義のソフトウェアエンジニアリングとして位置付けられる領域です。人間は考えをまとめるため、それを伝えるため、計算機に司令を出すため、出来上がったプログラムを保守するためなどに言語記述を使用します。ある言語の記述を得るために前提となる記述もあり、こういった複合的な記述手順として、種々の開発技法があみ出されてきました。

三つの世界(対象領域, 機械, 人間)の構造は全く異種のものであり,これ等を同時に扱わなくてはならないところにソフトウェアエンジニアリングの原理的な難しさがあります。

対象領域というのは、機械も人間もない世界として一般的には成立しています。物理現象、装置、

物体などから構成され、そこに存在・機能しています。機械の世界というのは、主としてフォンノイマン型の演算装置と記憶装置とからなる世界で、これも対象領域も人間もなくても成立しています。人間の世界もまた、その思考や認識の仕方、社会的な形態として有史以来存立しています。

ソフトウェアエンジニアリングの歴史

ソフトウェアエンジニアリングは、図3に示すように、その発祥より高々30 年たらずですが、概ね３つの世代に分けてとらえることができます。

第１世代：

1970 年代は、高レベルプログラミング言語が数多く提唱されるとともに、プログラムの規模の問題を扱った複合設計法や段階的詳細化法に基づく技術が整備されました。この時代に、ソフトウェアの構成単位である抽象データ型やモジュールの概念も整備されました。

第２世代：

1980 年代は、ここではオブジェクト指向開発技術に代表されるように生産パラダイム自身が見直され要求定義やプロトタイピング技術が提唱されました。ソフトウェアというものが単純なプログラムという対象から、より複合的なシステムという捉え方に変わり、この時代でようやく「仕様」という概念が実務レベルでも定着しました。顧客・ユーザの論理と、開発者との論理とを繋ぐ基本的な技術が確立したと言えるでしょう。

第３世代：

1990 年代以降は旧来の生産技術は熟成の域に達するとともに、分散システム向きの新しい計算モデルや、コンポーネントウェアに代表される広い意味での知識の再利用技術が台頭して来ています。実際の開発プロジェクトでも従来のウォータフォール型の生産パラダイムにとらわれつつも、チーム制やインクリメンタル開発等を導入し急変するソフトウェア市場に対応しようとしています。これ等の動向は、標準化やオープン化といった社会的な仕組みが急変し、ビジネスに直接影響を及ぼすようになったところに技術が集約されているとみることができるでしょう。

プロセス／プロダクトの技術開発の重点化という視点で言うと，低粒度から高粒度へと移行しつつ、開発プロセスの局面とプロダクト（中間成果物）の局面とが繰り返し波打つように発展してきているように思えます。

産業構造の変化

産業論上の経緯からみますと、図４に示すように、電子デバイス（ハードウェア）の領域とユーザ（顧客）領域とのギャップが年々広がってきているとみなせます。すなわち、モノリシックで未分化であったソフトウェア全体が、分化して来たと同時に、ユーザがやっていたことをソフトウェアが代替えするようになったということであり、社会の中でのソフトウェアの役割が拡大してきたことを意味していると言えるでしょう。

初期：

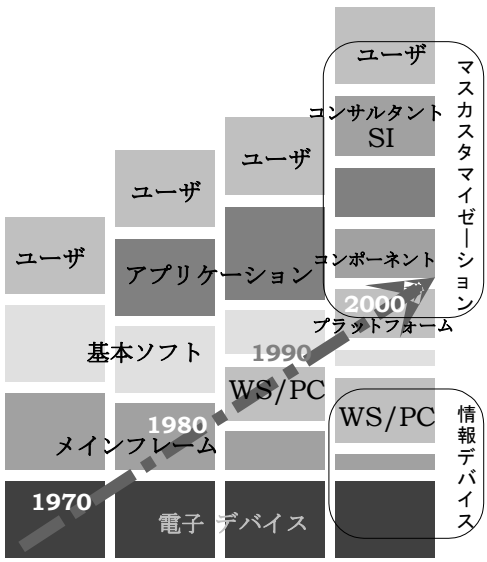


図4 産業構造の変遷

かつての大型機（メインフレーム）の時代では、電子デバイス（ハードウェア）とユーザ（顧客）との間には、メインフレームの OS（オペレーティングシステム）と基本ソフト（コンパイラやデータベースシステムなど）があればよく、これ等の開発に注力していました。この時代では、ハードウェアの知識を持つ者と、ソフトウェア（あるいはプログラミング）の知識を持つ者が協調する程度で済んでいました。

中期：

時代とともにユーザの要求も多様化し、アプリケーション（ユーザの要求する機能をそのドメイン／分野で実現するためのソフトウェア）が成長しました。その後、WS（ワークステーション）や PC（パソコン）が台頭し、旧来の汎用機／メインフレームの役割は小さくなりました。

近年：

最近では、ユーザの要求はますますエスカレートし、要求を満たすシステムを構築するためには専門のコンサルタントや SI（システム・インテグレータ）がいなくては対処できなくなって来ました。また、これと同時に、より大規模で複雑なシステムを効率よく構築するために、プラットフォーム（Windows, Unix, MacOS など）が整備され、その上でコンポーネント（部品）を組み合わせる技術が台頭して来ています。1990 年代に入ってからのインターネットの急速な普及とオープン化が技術開発を促進しており、1980 年代に提案されたオブジェクト指向開発技術やオブジェクト管理技術が実用段階に入り、部品のコンポーネント化も急速に進んできていると言えます。このユーザとプラットフォームの間の技術は、一言で言うならば、マスカスタマイゼーション（ユーザの多様な要求を個別に・迅速に満たす技術）と呼ぶべき領域であり、これはいわゆるミドルウェアと呼ばれている領域に相当しています。

ビジネスの視点からユーザ領域の変化も激しく、価値観も多様化してきています。ユーザの世界にある問題を計算機や周辺機器等のハードウェアを使って解いたり、支援したりするのがシステムの目的ですから、ユーザ領域にある要求や問題の構造と、ハードウェアの構造とを原理的につなぐためにソフトウェアの仕組みがますます重要になってきています。ここで言うマスカスタマイゼーションは、構造変換を迅速に行なうための系統的な方法を確立するための一つの方向として今後も重要と思われます。

一方、電子デバイスも高機能化し、WS や PC の機能、さらには、インターネットの機能などが埋めこまれた装置（デバイス）が台頭して来ています。これは、旧来の組み込み型のシステムや情報家電といった分野の発展形としてとらえることができます。この電子デバイスの発展分野は一言で言うならば、情報デバイスの分野と呼ぶことができます。特に、こういった装置産業に直結した領域は日本の得意とする領域であり、諸々のインパクトを与えることができる領域です。

技術の難易度

ソフトウェアそのものの技術も図５に示すように、当初の簡単な科学技術計算のレベルからより大規模で複雑なものへ、さらに、社会的な意味での重要性も大きくなってきています。

静的構造：

プログラムの制御構造（ブロック構造、条件分岐構造、くり返し構造）やデータ構造（構造体、配列構造、リスト構造）にだけ注意をはらえば、そこそこのソフトウェアが作れるという時代がかつてありました。

情報システム：

計算機というものが科学技術計算だけでなく、情報を扱う機械という位置付けが定着し、これに伴って情報システムの構築方法（組織のミッションや、扱っている情報から計算機システムを構築する方法）が確立されました。

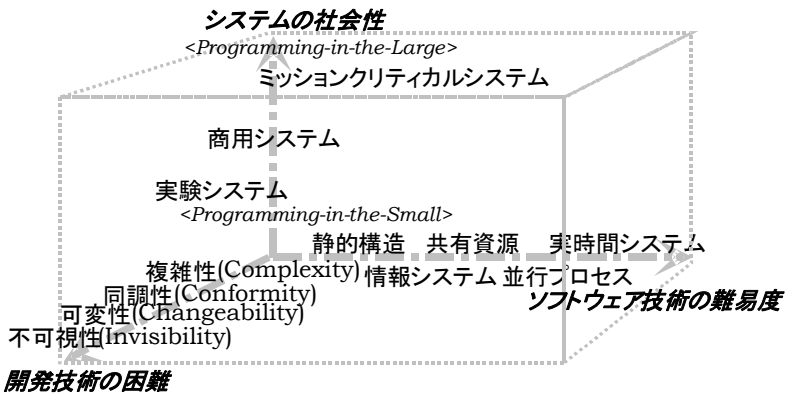


図5 システムへの要求と技術課題の変遷

共有資源：

計算のメカニズムもより複雑になり、同時にいくつかの資源（記憶領域や周辺装置など）を共有したり、これ等の資源を扱うプロセスを排他制御する機構が確立されました。

並行プロセス：

いくつかの計算や処理を行なうプロセスが同時に複数実行される並行プロセスの研究が進み、より複雑な問題を扱えるようになって来ました。

実時間システム：

対話型システムや、オンラインシステム、さらには，制御システム（FA: Factory Automation や航空管制，航空機制御）といった時間に依存した概念も直接扱えるようになり、より広い範囲の問題を扱えるようになって来ました。

こういったソフトウェア技術の難易度に関して、ここ数十年の間にわたって背景となる基礎理論の整備が進められてきていますが、並行プロセスや分散処理、実時間システムを基礎づける理論はまだこれからの課題であると言えるでしょう。

こういった技術展開を代表していると言えるのが天才M. A. Jacksonの仕事です。Jacksonは、1975年にJSP: Jackson Structured Programmingを発表し、構造化プログラミングの３つの基本構造（連接、選択、繰り返し）に基づく、実行単位としてのプログラムの仕様と実現との関係を明らかにしました。次に、1982年には、JSD: Jackson System Developmentという開発技法を提唱しています。JSDは、JSPの発展させたもので、プログラムの総体としてのシステムを開発するための技法です。この技法によって動的な側面（振る舞い）についての仕様化を基本に置いており、基礎的なモデルはCSP:Communicating Sequential Processという並行プロセスモデルを使っています。さらに、1995年に『問題フレーム』という数学における問題の解き方をソフトウェアに援用する方法を提唱しています。これは、近年のデザインパターンの考え方に通じるものがあります。

抽象度・粒度と再利用技術

システムは、広い意味で、抽象的な機械とみなすことができます。計算機というハードウェアは、機械語という命令体系を提供している汎用の機械です。より多くの高レベルの機能を提供する機械を構築するためには、抽象度の高い命令体系を持った抽象的な機械を積み重ねて行く必要があります。プログラミング言語も命令体系を提供している低レベルの抽象機械です。非常に単純化して述べるならば、プログラミング言語の言語要素をまとめて記述することによってオブジェクトが得られ、さらに、オブジェクトをまとめたものがコンポーネントとすることができます。システムのアーキテクチャは、コンポーネントの総体としてまとめることができます。いわゆるミドルウェアと呼ばれている領域は、フレームワークやコンポーネントといった中間層の抽象機械に相当しており、より大規模で複雑なシステムを開発するためには、洗練化されたものを

開発しておく必要があります。

オブジェクト指向技術(OOT:Object-Oriented Technology)は，ソフトウェアの再利用性や理解容易性を高める技術として近年急速に発展してきたものです。開発工程の中でもプログラミング工程に対応したオブジェクト指向プログラミング(OOP:Object-Oriented Programming)、設計工程に対応したオブジェクト指向設計、(OOD:Object-Oriented Design)、分析工程に対応したオブジェクト指向分析（OOA:Object Oriented Analysis)というように全体に渡って技術開発が進んできています。オブジェクト指向開発技法は，多くの流派がありますが，最近では徐々に淘汰され，Rational 社の提案した表記方法（UML: Unified Modeling Language）がデファクトになりつつあるようです。

システムの開発支援環境の視点は、システムの構成要素の抽象度や粒度とともに発展してきています。旧来のプログラミング言語レベルでは、コンパイラ（言語の処理系）を中心とした支援環境が中心でした。コンパイラは、エディタやライブラリやデバッガ等を一体化した統合環境に移行し、さらに、モジュールの内部ロジックと外部インタフェースとを分離して管理できるような設計支援環境へと発展して行きました。旧来このコンパイラが単体で提供されていた。最近では、この統合環境の考え方をさらに発展させ、ライブラリそのものを開発したり、ライブラリや部品を組み合わせてさらに大きな部品を開発・利用できる環境が整備されて来ています。再利用可能な部品としてのコンポーネントとして最初は GUI(Graphical User Interface) で使い易いものが普及しました。その後、データベースやクライアントサーバシステム、さらには、分散システム向きのコンポーネントを搭載した開発環境が提案されてきています。コンポーネントの標準化や信頼性を上げる努力も地道に行われて来ており、旧来の個人的で小さなソフトウェアを対象としたものから、企業システムレベルで利用可能なものも整備されて来ています。

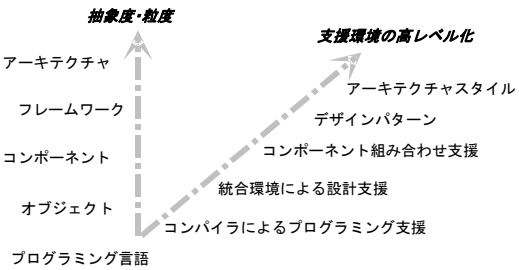


図6 再利用の単位と抽象度

また、設計や開発という本来の人間の知的創造活動を焦点にあてたアプローチとしてデザインパターンの考え方が有効です。これは、オブジェクトの組み合わせ方の雛形を、設計過程で現れる問題とその解という図式でとらえるものです。デザインパターンをより高粒度のレベルで行なうアプリケーションフレームワークやアーキテクチャスタイルといった開発ノウハウも、ドメインを特化した形で実用化されて来ています。

システムの構成単位の抽象度や粒度という見方と、支援環境のレベルとは密接に関係しており、最近では、OS やプラットフォームレベルでも従来のプログラミング言語レベルでの仕掛け（コンパイラの機能）が組み込まれてきています。

パターン技術の系譜

パターン技術に対する取組みは、広範囲にわたっており図７に示すように、理論面から実践面まで、汎用のものから特定の領域を対象としたものまで、さまざまなアプローチが試みられています。この図では、オブジェクト指向技術を中心に据えており、建築分野の C. Alexander の『パターンランゲージ』は対象領域が異なるものの特定領域の実務知識の集大成として位置づけることができます。「技術の難易度」の節の最後で述べた M.A. Jackson の『問題フレーム』は、パターン技術の伝道師の人々はあまり言及しませんが、ソフトウェアとそれを基礎づけている数学の世界とを結びつけ、かつ、豊富なコンサルティングの経験や哲学的な思索も加えて一般的ではあるものの実用性の高い方法を提案しています。並行プロセスやリアルタイムシステ

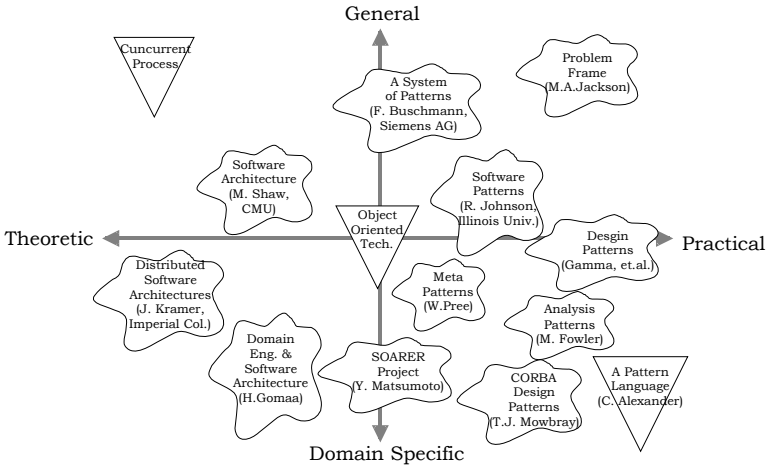


図7 パターン技術の系譜

ムに関する技術は、未だ学術領域での理論開発が盛んな分野であり、現状では一般的で理論レベルでの提唱に止まっています。

M. Shaw 等による『ソフトウェアアーキテクチャ』や R. Johnson の『ソフトウェアパターン』は、パターン技術の思想的な基礎付けを行ない、GoF の『デザインパターン』は、自らの実務上のコンサルティングの経験からオブジェクト指向に関する 23 個のパターンを具体的に提案しています。これ等は、パターン技術の先駆的でかつ啓蒙的な業績と言えるでしょう。オブジェクト指向より一般的なパターンについては、その粒度の概念とともに整理をした F. Buschmann 等の『パターン体系』があります。パターンそのものを導出する原理を示したものは未だ多くはありませんが、W. Pree の『メタパターン』の《ホットスポット》、《フローズンスポット》の考え方は説得力があります。

オブジェクト指向を中心としたパターン技術は、大きく２つの方向性が出てきています。一つは実装面、すなわち、Gof の『デザインパターン』に代表されるようなオブジェクト指向に基づく開発環境やコンポーネント、プラットフォームに特化した開発者の論理の中でパターンを開発して行くアプローチです。もう一つは、顧客との関係を主体にした『アナリシスパターン』の領域です。M. Fowler の『アナリシスパターン』では、ビジネス情報系についてオブジェクト指向の手法を使って業務分析を行ない構築すべきシステムの実務世界でのあり方を《ビジネスモデル》として構築して行く方法を提唱しています。

特定分野向けのパターン技術としては、ビジネス情報システムを主対象とした T. J. Mowbray 等の『CORBA デザインパターン』、リアルタイムシステムの分野を対象とした H. Gomaa の『開発技法』や、その技法や組み込みシステムの製品開発から設計知識を抽出して基盤整備を進めている『IPA-SOARER(SOftware Architecture Repository for Embedded and Realtime systems)プロジェクト』等があります。特に、リアルタイムシステムや分散システムに関しては、理論整備とともにアーキテクチャの提案をして行く必要があり、J. Kramer 等の「分散アーキテクチャ」の研究は並行プロセスを中心にすえた実務的なパターン整備を狙っています。

技術展望と課題

正しい技術が社会の中で普及するとは限らないですし、その発展が連続的であるという保証もありません。しかしながら、趨勢と大局的な方向性という視点からの課題を整理すると、ソフトウェアエンジニアリングに関する課題は次のように整理することができます。

(1) 高度再利用技術＝規模の問題：

LSI の集積度が年率 50～60%で上がる一方で、ソフトウェア開発の生産性は年率 10%程度の伸びに留まっています。規模の問題を解決するためには、原理的にソフトウェア開発時の抽象度を上げ、再利

用率を高める解決方法しかあり得ません。コンポーネント技術という視点からもより大きな粒度、あるいは、抽象度の高いものを部品として蓄積する技術（コンポーネントを作る技術）が必要ですし、部品を適確に組み合わせて顧客にソリューションを提供するマスカスタマイズの手法（コンポーネントを組み合わせてアプリケーションを構築する技術）も、人間のコミュニケーションや企業モデルを扱うことができるより要求工学的なアプローチを取り入れ、さらに、ハードウェア・ソフトウェア、ないし、プロダクト・プロセスを統合化したもの（コデザインプロセス）に進化して行かなくてはなりません。

⑵ 高度高品質化技術＝信頼性の問題：

ユビキタスコンピューティングを始めとして社会のあらゆる分野にソフトウェアが普及し、組込まれて行くと、従来のミッションあるいはセイフティクリティカルなシステムのみならず、あらゆる分野のソフトウェアに対する社会的な責任も重要になってきます。現在の技術では、そのソフトウェアの品質を保証する技術も未成熟であり、理論的な整備もフォンノイマン型の計算モデルの上でさえ完全とは言えません。ソフトウェアの規模の増大に伴って、ソフトウェアの構成部品や提供サービス一つ一つにかけることのできる費用も限度があり、開発プロセスや品質そのものの考え方に対してもブレークスルーが要求されています。形式的仕様記述や証明技術、計算の理論についてもソフトウェアエンジニアリングの視点から総合的に見直す必要があると言えるでしょう。私はこの領域のことを『精密ソフトウェアエンジニアリング（Precise Software Engineering）』と呼んでいます。

セキュリティやプライバシなどの基盤技術，標準化と標準に対するコンフォーマンスの問題などもその知識蓄積の上で取り扱い、ノウハウ（パターン）としてまとめて行かなくてはならない重要な課題です。

⑶ 多次元オープン化＝社会性の問題：

メインフレームレベルの技術から、組み込みシステムのレベルまで一貫したプラットフォーム整備を図る技術は、産業構造における水平・垂直型の開発役割モデルの視点から有望です。Windows やUNIX 等のオペレーティングシステムも、企業の基幹業務レベルのものから民生用の組込み製品レベルのものまで、必要に応じてオペレーティングシステムそのものがコンポーネント化しカスタマイズ可能なものになって行くでしょうし、こういったシリーズに適合した生産支援環境の開発も必要です。また、近年のオープン化の契機となっているインターネットの普及に見られるように、デジタルコンテンツそのものの拡大もソフトウェアと同期しており、ソフトウェアと情報との融合化も今後の大きな課題の一つと言えます。

以上のように、ソフトウェアエンジニアリングの技術というのは、ハードウェアやオペレーティングシステムレベルと、顧客へのソリューション提供という今後ますますギャップが広がってゆく２つの世界を繋ぐところにあり、原理的には抽象的な機械を構築し積み上げて行く技術（デザインパターン的な側面）と、顧客という人間世界の欲求・願望・要求や組織のミッションという社会的コード・規範（アナリシスパターン的な側面）を同時に視野に入

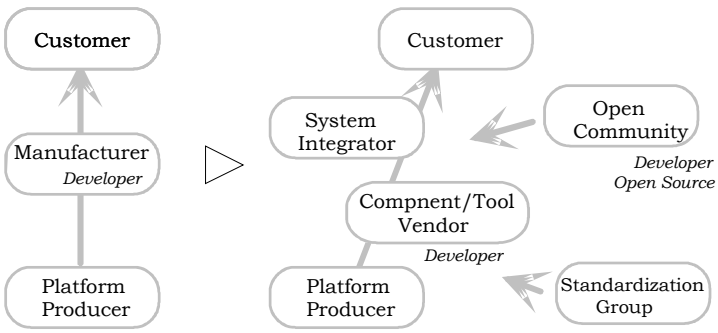


図8 ソフトウェア開発における産業構造の変化

れなくてはならないでしょう。そして、パターン技術の範囲は、旧来のメインフレームやワークステーションで培われてきた技術をダウンサイジングしたり、カスタマイズして分野をまたがって適用して行く広義の開発知識の再利用技術と、新たなる人間の要求に答えるべくより抽象度の高い機械を提供して行く方向性とがあります。システムの価値や、開発コストといった考え方自体が、旧来の物理的な工業生産物のパラダイムに基づいた所有権や交換経済の枠組みではとらえることができないということであり、新しい社会的なコード（規範）を組み入れたものになって行くことを示唆しています。私は、この領域を『記号論的ソフトウェアエンジニアリング(Semiotic Software Engineering)』と呼んでいます。

ソフトウェアエンジニアリングは、ソフトウェア開発にかかわる知識を、供給者（生産者）・利用者（消費者）といった人々が関わる情報流通・蓄積を行なって行くための基本的な技術として位置づけることができます。従来のソフトウェアエンジニアリングの成果というのは、技法や開発環境（ツール）といったどちらかと言うと開発ノウハウをガイドブックやアプリケーションプログラムという形で静的に閉じ込めたものでした。しかし、近年のソフトウェア開発プロジェクトでは、より市場性が求められ、ダイナミックに変動する社会の中で、適確な開発ノウハウとしてのソリューションをいかに得るかということが課題となっています。従って設計ノウハウを持っている人間と、それを使う人間とを旨く結びつけて行くことや、システムの提供者と利用者との共通了解を得る仕掛けを構築することがソフトウェアビジネス振興の要となります。いわゆるコンポーネントの開発に関しても、占有型の商用ソフト開発のみならず、オープンソース型の重要性も増してきています。こういったことは、近年のネットワーキングの発達によるところが大きいですが、図８に示すように、知的財産を共有化して行く新しい社会システムが台頭してきています。こういった産業構造の変化は、従来のメーカという位置づけでシステム開発を行なって来た組織が衰退し、ソフトウェアそのものの開発は優れたデザイナによる専門家集団（コンポーネントベンダやオープンソース供給）が行ない、顧客へのソリューション提供はシステムインテグレータやコンサルタントが行なうようになってきていることを意味しています。

〔この記事は、（社）日本電子工業振興協会「先端ソフトウェア技術に関する調査報告書」（2000 年 3 月）の執筆内容に基づき、加筆・修正したものです。〕

（おおつき　しげる）

状態遷移表設計によるソフトウエア開発プロセス改善

コニカ㈱ オフィスドキュメントカンパニー 機器開発統括部 第3開発センター　渡辺 智 ／ 滝 研司 ／ 黒畑 貴夫

1　はじめに

複写機をはじめとする情報機器は多機能、高性能化し、それを制御するソフトウエアも肥大、複雑化しています。また、商品サイクルも年々短くなり、開発期間の短期化が求められています。これらの機器を制御するソフトウエアの開発においても、このような要求に対応するための対策が望まれています。

従来のソフトウエア開発は、要求仕様書からソフトウエア設計仕様書を作成し、この設計仕様書を元にソフトウエアの最終段階であるソースコードを作成していました。

しかし、要求仕様書の大部分が図表でなく文章で書かれているために、ソフトウエア設計者が要求仕様書の文章を読解してソフトウエアを設計する段階で、誤解が生じたり、思考の漏れや抜けが生じる可能性があります。対象となるソフトウエアが複雑になればなるほどこの傾向は顕著となっています。また、ソフトウエア設計仕様書もほとんどが文章で書かれているため、デザインレビュー等においては、レビューの参加者が、設計内容をビジュアルに把握する事ができず、期待通りのレビューを行う事ができないといった問題も起こります。

さらに、時事刻々変化する市場ニーズへの対応や開発段階での課題に呼応して要求仕様が変更される事がありますが、従来は変更内容が記載された仕様書や変更部分が修正された仕様書をソフトウエア担当者が読解し、直接ソースコードを修正しています。このため変更部分の影響について漏れや抜けが生じ、仕様変更に伴う不具合が発生する事があります。

今回、上記の問題点を改善すべく、複写機のフィニッシングシステムおよびカラープリンタシステムの制御ソフトウエア設計に状態遷移表設計手法を導入しました。また、合わせて状態遷移表設計を支援するCASE（Computer Aided Software Engineering）ツール「ZIPC」の導入効果についいても評価を行ったので、その結果と有効性について紹介します。

2　情報機器制御システム制御の特徴

情報機器は、機器に組み込まれたコンピュータシステム（組込みシステム）によって制御されており、次のような特徴があります。[1]

第一に、組込みシステムはユーザーによる操作やセンサ信号等の外部からの事象（イベント）に対して、ランプを点けたりモータを回す等の処理（アクション）を行う反応型（リアクティブ）システムである事があげられます。このようなリアクティブなシステムでは、内部の状態と外部からの事象により制御（処理）を行う事が多いです。[2]

情報機器の一つである複写機について考えてみると、原稿をセットしてからコピーボタンを押してコピー動作を開始する手順を処理するためには、原稿がセットされた時点でその状態を内部に保持し、コピーボタンが押された時点で内部に保持した原稿がセットされているという状態と照らし合わせて、コピー動作（処理）を開始します。ただ、単純に思えるこの動作も、実際にはコピーボタンが押された時に機器がウオームアップ中なのか、すでにウオームアップが完了した状態なのかといったように、多くの状態が存在する事になります。

第二には、制御対象に予測・制御不可能な要素が含まれているために入力と出力の関係が一意に定まらない事があげられます。

例えば複写機では、給紙開始指令を出してから紙が通過検知センサに到達するタイミングは、紙の特性や環境要因等により複写動作の各紙毎で異なっており、同一のタイミングで制御する

事は不可能です。第一の特徴で示した多くの状態において、様々なタイミングで事象が発生する事となります。

即ち、複写機をはじめとする情報機器の制御設計においては、複雑にからみあった状態と、これらの状態において様々なタイミングで発生する事象を如何に整理するかが課題となります。

３　状態遷移表設計手法の導入

３．１　　目的

本手法導入の目的は、状態遷移表を使ってソフトウエア設計を行い、状態と事象を明示的に整理する事により、ソフトウエア設計段階で発生する不具合を低減する事です。

Fig.1は、複写機のフィニッシングシステムの開発段階で発生したソフトウエア不具合について、不具合の発生原因を、要求仕様そのものの不備、設計段階でのミス、コーディング段階でのミスの３つに大別して示したものです。

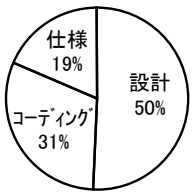


Fig.1 不具合発生の原因となった工程

Fig. １から、不具合の多くは設計段階で発生しており、ソフトウエア不具合を低減させるには、設計段階での不具合低減が有効である事がわかります。

設計段階で不具合が発生する原因として、次のような事があげられます。

第一には、設計が設計者の頭の中で行われており、その結果を明示的に示すものがないため十分なデザインレビューが行われないこと。第二には、納期圧力の関係から設計の一部を端折ってコーディング作業を開始してしまい、コーディング中に設計漏れや設計抜けに気付いても設計フェーズにもどらずその場のコーディングで対処してしまうこと。第三には、仕様変更があった場合、設計者が頭の中で考えていた初期設計時の情報が失われており、変更仕様の織込みに際して、影響範囲や関連個所の把握が十分に行えずに設計漏れや設計抜けが生じやすいこと。

状態遷移表設計手法の導入は、これらの問題を解決し、設計段階での不具合を低減する事にあります。

３．２　導入

状態遷移表設計手法は、制御対象となるシステムがとり得る全ての状態と、そのシステムで発生する全ての事象を洗い出し、この２つの要素より構成される表を作成し、ある状態である事象が発生した時の処理をその表の升目（セル）に記載すると共に、処理実行後にシステムがとる状態を記載する事により、システム全体の動作を設計するものです。Fig.2に状態遷移表の概要を示します。

事象＼状態		ｱｲﾄﾞﾘﾝｸﾞ	給紙中
	E＼S	1	2
ｺﾋﾟｰｷｰON	1	2 給紙ﾓｰﾀｽﾀｰﾄ	不可
給紙ｾﾝｻｰON	2	無視	3 給紙ﾓｰﾀ停止

Fig.2 状態遷移表（例）

今回、状態遷移表設計手法を２つの機種開発の一部に導入しました。

一つは、複写機のフィニッシングシステムの制御ソフトウエア設計への適用です。今回設計手法の導入対象としたのは、フィニッシングシステムのカバーシートフィーダー機能です。カバーシートフィーダー機能とは、複写機本体から排出されてくるコピー紙に対して、予めフィニッシングシステムに用意された紙を表紙や裏表紙として挿入する機能です。ユーザーが、予めフィニッシングシステムに用意された紙をどこに挿入するかを指定するオペレーション機能と、紙を搬送して複写機本体から排出されてくるコピー紙と合わせるために機構部品を制御す

る搬送制御機能とから構成されています。今回は、この両方の機能のソフトウエア設計に手法を導入しました。

もう一つは、カラープリンタの状態管理機能です。プリンタの制御ソフトウエアは、外部との通信を行う通信機能、機械の状態を監視しプリント作業全般について指令を出す状態管理機能、画像を形成する画像形成機能、紙の搬送を制御する紙搬送機能、トナーを紙に定着させる定着機能、特殊な動作指令に対応して動作する特殊機能より構成されていますが、今回は状態管理機能のソフトウエア設計に手法を導入しました。

4　結果

4．1　不具合発生率の低減

Fig.3は、フィニッシングシステムにおいて状態遷移表設計手法を採用してソフトウエア設計を行ったカバーシートフィーダー機能と、従来の設計手法でソフトウエア設計を行ったその他の機能でのソフトウエアの不具合発生率を示した表です。手法の導入により設計段階での不具合発生率が従来の約1/2に低減されています。

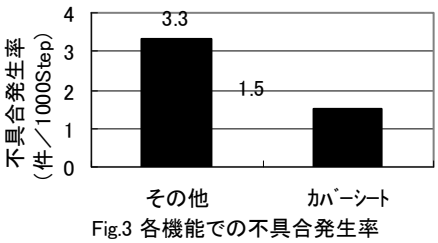


Fig.3 各機能での不具合発生率

Fig.4は、同様にカラープリンタにおいて状態遷移表設計手法を採用してソフトウエア設計を行った状態管理機能と、その他の機能でのソフトウエアの不具合発生率を比較した表ですが、このシステムにおいても、手法の導入により設計段階での不具合発生率が従来の約1/4に低減されています。

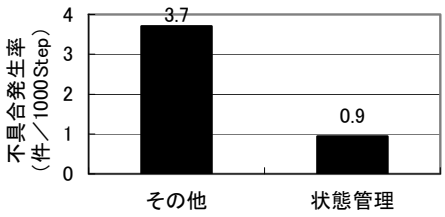


Fig.4 機能別の不具合発生率

4．2　仕様変更への対応

今回のカバーシートフィーダー機能の開発では、初期設計で機能の基本的な部分に対応した小規模な状態遷移表を作成し、開発の進捗に伴って発生する要求仕様を、状態・イベントの追加として折り込み、状態遷移表を拡張する方法を採用しました。

Fig.5は、初期設計での状態遷移表と最終的な状態遷移表のセル数を比較したものですが、この方法により開発の進捗に合わせて初期の約2.5倍の規模まで状態遷移表を拡張し、且つ、日程の遅延なく前記のように高い品質のソフトウエアを提供できました。

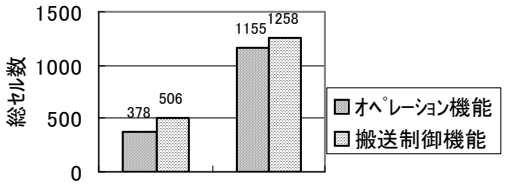


Fig.5 各設計段階でのセル数

5　考察

5．1　設計における手法の有効性

今回ソフトウエア設計に状態遷移表設計手法を導入した機能は、ユーザーインターフェイスの役割を担うオペレーション機能、紙搬送のための機構制御を担う搬送制御機能、プリンタシステムの状態監視と全体の制御を担う状態管理機能といったようにそれぞれ異なった機能ですが、これらは全て有限の状態を持ち、外部からの事象の変化を受けて、何らかの処理を行うリアクティブなシステムであると考える事ができ

ます。状態遷移表設計は、この様なシステムの設計段階において発生する不具合の防止に有効な方法であると言えます。これは、状態遷移表を活用して設計を行う事により、システムがとりうる全ての状態とイベントを洗い出すと共に、状態遷移表の全てのセルについてそこでの処理を考えるため、設計段階での漏れや抜けを防止できるからです。

5．2　デザインレビューでの有効性

Fig.6 は、フィニッシングシステムのカバーシートフィーダー機能の初期設計段階で作成した状態遷移表をもとにデザインレビューを行った際のデザインレビューでの不具合指摘件数とその後に指摘を受けた不具合の件数割合を示したものですが、不具合の多くがデザインレビューにより指摘されている事がわかります。これは、仕様を状態遷移表で表す事により、従来設計者の頭の中で行われていた設計がビジュアル化され、デザインレビューの参加者が設計内容を理解しやすくなり、適切な指摘が可能になったためと考えられます。

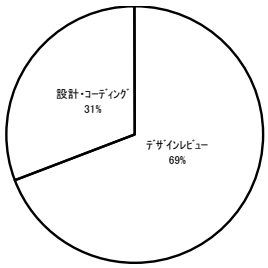


Fig.6 不具合指摘件数

5．3　仕様変更への適応性

Fig.7 は、状態遷移表設計手法を導入してフィニッシングシステムのカバーシートフィーダー機能を開発した際の工程別作業工数の割合を示したものです。

状態遷移表設計手法を導入したソフトウエア開発では、設計作業が全体の作業量の約６割を占めています。初期設計において設計の作業量

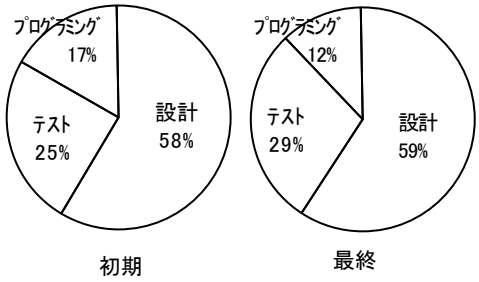


Fig.7 工程別作業工数

が大きなウェイトを占める事は容易に想像されます。しかし、仕様変更を行った最終段階においても、なお、設計作業が初期設計と同等のウェイトを占めるのです。

これは、状態遷移表設計では仕様変更時に、追加または削除される状態とイベントを整理した後、状態遷移表を修正するためです。即ち、仕様変更時も初期設計時と同様に状態遷移表による設計作業が行われるため、仕様変更作業後でも作業全体に占める設計作業のウェイトは変わらないのです。この再設計作業により、仕様変更に伴う設計漏れ、設計抜けを防止する事ができます。

そして、状態遷移表設計では、最新の仕様に対して、常に最適な設計がされている事になります。これにより、ソフトウエアの最新の設計状態を容易に把握できると共に、たとえ不具合が発生しても現状分析が容易であり、短時間で原因解析が可能であるという効果が期待できます。

6　状態遷移表設計支援ツールの評価

ソフトウエア設計に状態遷移設計手法を導入し、その有効性を確認する事ができましたが、今回の手法導入において幾つかの課題があった事も事実でした。

そこで、この課題を解決するため、状態遷移表設計手法を支援する CASE ツール「ZIPC」についても調査とその評価を行いました。

ZIPC については、ツールの各種機能や、ZIPC

を実際の開発業務に活用した事例が紹介されており、今回はこのZIPCを採用しました。[3] [4]

今回の手法導入の中で出てきた課題は、状態遷移表の肥大化と、状態遷移表とソースコードの不一致です。

フィニッシングシステムのカバーシートフィーダー機能の開発においては、市販の表計算ソフトを活用して状態遷移表の作成を行いました。しかし、Fig.5にも示したように、オペレーション機能、搬送制御機能とも状態遷移表のセル数が1000を超えており、全体を一度に把握する事が難しくなると同時に、状態遷移表のコンピュータの画面への表示、紙へのプリントアウトといった物理的な面での制約も発生しました。

また、フィニッシングシステムの開発では、ソースコードの作成は、状態遷移表を見ながらソフトウエア作成者がコードを作成する作業を行いましたが、この際に入力ミスが発生しました。今後、より多くのシステム開発に状態遷移表を採用した場合、この作業は大きな課題になると考えました。

今回の状態遷移表設計において、状態遷移表が肥大化した大きな要因は、セルの中に条件判断を上手く記述できない事にありました。今回のように市販の表計算ソフトを活用して状態遷移表を作成した場合は、操作上、セルの中を分割する事が難しく、新たな状態を作成して対処する事となります。この結果状態の数が多くなり、状態遷移表が肥大化してしまいました。これに対して、ZIPCはセル内での条件分割機能があります。フィニッシングシステムのオペレーション機能について、ZIPCの条件分割機能を使用して状態遷移表を再設計しました。その結果、状態遷移表のセル数を約1/6にでき、ZIPCが状態遷移表の肥大化防止に有効である事が確認できました。

ソースコードの不一致については、ZIPCが持つソースコード自動生成機能を活用する事によって改善が可能であると考え、今回状態遷移表設計手法を導入したカラープリンタの画像形成機能の一部について、ZIPCを活用して状態遷移表による再設計を行い、ZIPCの自動生成機能によるソースコードの自動生成を行いました。

その結果、状態遷移表で記述した分岐条件等が自動的にコード化され状態遷移表とソースコードの不一致が回避されたと同時に、ソフトウエア設計者によって入力するソースコードが1/3に軽減されました。また、最終的に生成されたオブジェクトコードは、ZIPCを使用せずに作成した時とほぼ同等でした。

ZIPCの活用は、以上のような効果がありますが、導入の初期段階ではツールに習熟するための時間が必要となります。

7 むすび

情報機器の制御ソフトウエア設計に状態遷移表設計手法を導入する事によりソフトウエア設計段階での不具合発生を低減できる事を確認しました。同手法は多数の状態と事象を管理するプログラムの設計に有効な手法であると考えます。

今後、同手法をより多くの開発に適用していくと共に、支援ツールも含めより効果的な導入方法についても検討していく必要があると考えます。

参考文献

[1] 高田広章,情報処理,Vol.38 No.10,870,(1997)

[2] 川口晃,岸知二,門田浩,情報処理,Vol.38 No.10,881,(1997)

[3] 渡辺政彦,リアルタイム制御CASE,初版,電子開発学園編,電子開発学園出版局,(1993)

[4] 渡辺政彦,石田哲史,戴志堅,リアルタイム環境へのCASE導入, 初版,電子開発学園編,電子開発学園出版局,(1994)

(わたなべ さとし / たき けんじ / くろはた たかお)

ビジュアルシミュレーション使用例

松下電工(株) システム開発センター
大景　聡

シミュレーションについて

組込みシステムのソフトウェア開発において、特に評価工程の工数削減が課題となっている。組込みソフトウェアの評価を行う場合は、実機上でテストを実施する場合が多く、デバッグ効率が、PC と比較すると良くないので、その際に発生する不具合は評価工数の増大の一因となっている。

我々の場合、状態遷移表を用いてソフトウェア設計を行う場合は、ZIPC を使用している。キャッツ(株)が提唱するように、状態遷移表による設計により、抜けや漏れが減少することがわかる。従って、状態遷移表設計手法は、状態遷移の多いソフトウェアにおける設計フェーズで、有効な方法であることは間違い無い。また、拡張階層化という方法で、状態遷移が複雑化した場合に、状態遷移表が複雑化してしまうという問題点も解決している。しかし、階層化は、複雑化した状態遷移表をシンプルにするというメリットがある反面、遷移表と遷移表の接続部分の関係を人間の頭で考えざるを得ないため、元来のソフトウェアの抜けや漏れを減少させるという目的と矛盾してしまう。そのために、人間の頭で考えるのではなく、コンピュータに全ての場合を検証させることが、シミュレーションだろうと考えている。実際、状態遷移表だけを使用している人の中にも、シミュレーションで、設計に抜けや漏れが無いことを確認したい人も多いようだ。

ビジュアルシミュレーションの例

ビジュアルシミュレーションは、製品を意識して視覚的にシミュレーションできるもので、ZIPC Ver.5.0 からサポートされている。弊社の火報システムの一部に応用した例を図1に示す。図1に示す状態遷移表 において入力イベントと実行中の処理がグリーンでハイライトされている。この処理によってMicrosoft･ Visual Basic (以下VB と記す) 外観図 ではLED が点灯した状態になっている。

ビジュアルシミュレーションの効果と要望

シミュレーションの効果は、先に述べたが、ビジュアルシミュレーションとしての効果はどうであろうか?現状のビジュアルシミュレーションは、遷移表への発行イベントをVB によるGUI からのイベントに置き換えるというものである。状態遷移表設計当事者であれば、遷移表の内容を把握できているために、VB で GUI を作成するという作業分だけ無駄なように思われる。従って、実際に使用するのは、遷移表の内容を把握できていない人が、視覚的にテストを行うための用途となりそうだ。このようなケースは、遷移表が複雑で複数人数で開発を行っている場合が考えられる。これらを考えると複数枚数の状態遷移表のシミュレーション速度等が要求される。

また、ビジュアルシミュレーションは、VB を用いるため、コスト的に安いというメリットがあるが、VB のプログラミング自体で不具合が発生する可能性もある。また、VB 自体に工数が発生するという問題もある。このあたりは、汎用的なGUI 部品が提供されれば、解決してくると思われるので、そのあたりの進化に期待したいところだ。

他のツールメーカーでは、ビジュアルプロトタイピングを指向しているところが多いようだ。私もビジュアルシミュレーションという用途より、製品の企画段階での検討に用いるためのビジュアルプロトタイピングの方が、ニーズが高いように思う。こちらは、簡単な設計をすれば、プロトタイプモデルを動作させることができることを特徴とする。こちらの方も指向してもらいたいものだと考えている。

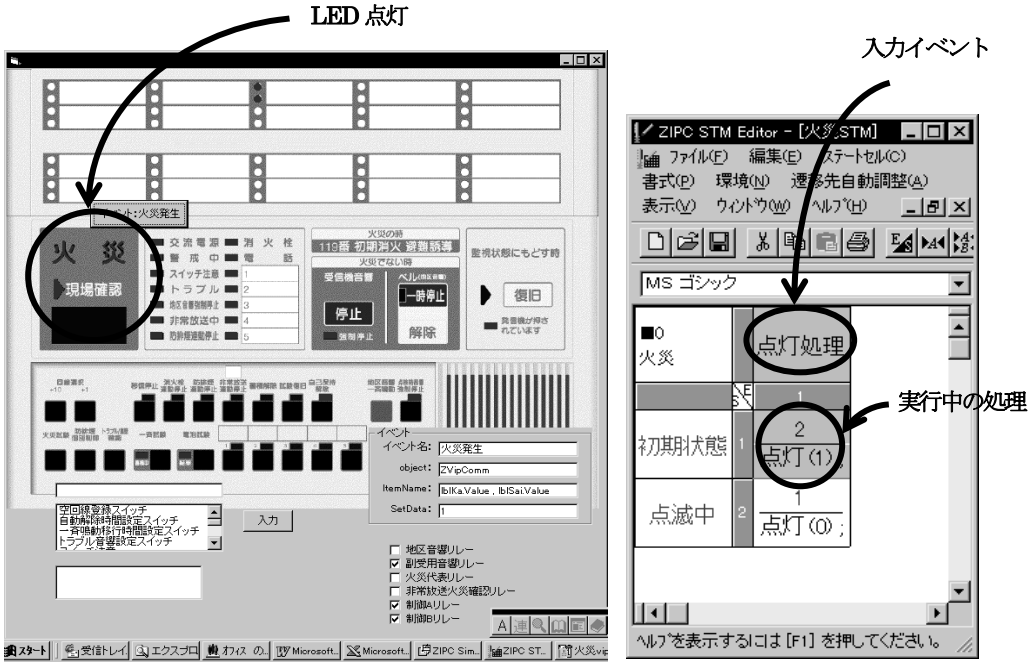


図1　火報システムの一部に応用した例

おわりに

ZIPC の特徴は、設計手法の敷居が低くて、誰にでも導入しやすいことだろうと思う。これは、現場のニーズから、発想されたためで、非常にユニークな点である。メーカー側の立場からすると、生産性向上だけでなく、なにかのソリューションが提供されるとさらに導入しやすくなるように思う。例えば、ZIPC で設計された設計部品を入手することが可能で、それらを組み合わせればソフトウェア開発が大幅に削減される等。

以上、いろいろと要望を中心に書いてきたが、日本発の CASE としてがんばってほしいものだと思う。

（おおかげ　さとし）

プリンター開発への適応をめざして

富士ゼロックス(株) DPC 商品開発統括部　IOT-PF 第一開発部
清水 哲

はじめに

弊社では、オフィス等におけるドキュメンテーションに関する機器・サービスを提供させていただいておりますが、私どものセクションにおいては、小型レーザープリンターの制御ソフトの開発を担当しております。レーザープリンターにおいては、ここ数年、ソフトウエア処理の比重がどんどん増加する傾向にあります。その背景のひとつには、カラープリンターの急速な立ち上がりがあげられます。これにともない、制御ソフトにおいても、従来の単色現像器のみの制御から、複数の現像器の制御へと、制御対象が拡大しただけではなく、画質を維持するためのさまざまな制御が求められるようになりました。もうひとつの背景としては、システムの複合機化があげられます。プリンターに、スキャナーやファクシミリなどのさまざまな機器を接続してひとつのシステムを構成するようにしたものですが、これにともない、サブシステム間での通信のサポートが、制御ソフトの中で、大きな割合を占めるようになってきました。このような「カラー化」あるいは「複合機化」という市場要求がある一方で、ここ数年で、プラットホームとなるCPUの事情も大きく変化しました。8 ビットから 16 ビットへ移行しても、コスト的には十分に見合うようになり、16 ビットの製品でも、処理速度がずいぶんと向上しました。このようにして、市場要求と、それを支える技術の向上とが、車の両輪のようにがっちりと結びつき、それが結果として、ソフトウエアの規模増大の要求へとつながってきたわけです。したがって私どもは、このような中にあって、高品質のソフトを、短期間で開発しなければならないという状況におかれています。

従来の開発における問題点

従来の設計プロセスは、まず、メカニカルエンジニアによって動作の概略が決定されると、それをもとにして、ソフトウエアエンジニアが一緒になって、ソフトでの実現性を勘案し再検討し、この過程を経てできあがった仕様をもとにして、ソフトウエアの概略・詳細設計に入っていくというステップをとることにしていました。まず初期的には、このようにしてスタートするのですが、ただ、当初の仕様がそのまま最後まで残ることはまれです。メカトラブルもありますが、新規技術を採用した場合は、そのカット・アンド・トライで、途中での仕様変更がかなり入ります。このような場合も、基本的には上記の設計プロセスで対応するのですが、ただ、常に概略設計からやり直すわけにはいかないので、その時々の仕様変更のメリットと、ソフト変更のリスクとを勘案して、ソフトとしてどこまで戻って設計変更するのか、という観点からの検討も実施し、最終的な落とし所をさぐります。これまでは、このようなプロセスで開発を進めてきたのですが、前任機の開発において、バグの多発が大きな問題になりました。その原因を分析してみると、仕様変更―再設計のプロセスの中で、設計段階における検討不足という問題が浮かび上がってきたのでした。

そこで、後継機の開発開始にあたり、バグ件数を低減するために、設計段階での検討不足に対する有効な手段はないかと、検討していた頃に出会ったのがZIPCでした。従来の設計プロセスにおいても、一部のサブシステムでは状態遷移表を設計資料として持っていましたが、全体としては、必須の設計資料とはされていませんでした。レーザープリンターを制御の面から考えると、基本的には、ステートマシンと考えることができるので、すべてのサブシステムで状態遷移表を作成することが可能なはず

です。そこで、後継機においては、一部のサブシステムでZIPCの導入を前提しながら、また今後の展開も考えて、すべてのサブシステムにおいて、状態遷移の考え方に基づいて、状態遷移表を作成して設計することにしました。

ZIPCの効果

状態遷移表を作成する設計プロセスにして良かったことは、まず状態定義をとおして、担当するサブシステムの状態モデルをじっくり検討することができたことです。将来の仕様変更も予想しながら、それにも対応できるような状態モデルを作成することは、ソフトウエアの基本設計のキーになるところです。ここは、レビューを重ねながら慎重に進めましたが、従来よりは見通しの良いモデルを作成できたと考えています。また状態遷移表にある入力条件をチェックすることによって、サブシステム間のインターフェイスの整合性を十分に検討することができました。システム間インターフェイスは、見落とすことが多い部分ですが、それを早期に発見することができました。また、イベントごとの処理の適否なども、じっくりと検討することができました。これらは、仕様変更への対応においても有効でした。このプロセスは、ZIPCのSTMエディタを中心に進められました。すなわち、STMエディタを使って、はじめは、処理をコメント文で記述しておいて、だんだんとC言語で書き加えていくという方法をとりました。これによって、最初から最後まで、同じフォーマットでレビューすることができました。そして、さらにジェネレータを使用すると、状態遷移表が完成すると同時に、コーディングまで終了することができました。

このように、設計プロセスに状態遷移の考え方を明示的に導入し、一部のサブシステムにおいてZIPCを導入したわけですが、この結果、後継機のソフト開発におけるバグ発生率は大きく改善されました。まだ、開発が終了していないので、正確な数字ではありませんが、おおよそ前任機に比べて、システム規模では約20%ほど大きくなったにもかかわらず、バグ発生件数では30%低減を達成するだろうと予測されています。この結果は、コストの面からも、十分に意味のあるものでした。プリンターのデバッグには、シミュレーションデバッグのほかに、実機で、実際に印字をしながらデバッグする方法とがあります。開発の最後のフェーズでは、完成度を上げるために、後者の方法による比重が大きくなります。このデバッグ方法は、コスト面から考えても、トラブルの再現工数だけでなく、その間のマシンのランニング費用も発生するために、あまりうまい方法とはいえません。それゆえ、バグの発生件数が抑えられたということは、品質だけでなく、コスト面でも大きな成果があったと考えています。

なお補足すると、ZIPCの導入にあたって懸念していたことは、ZIPCのメモリ消費量はどうなるのか、ということでした。メモリの消費量は製品のコストアップにつながるので、とくにRAMサイズについては制限が厳しく、ZIPCを採用したからといって、目標値をオーバーすることは許されませんでした。結果として、これは杞憂に終わりました。今回は、自分たちで同じ処理を作成した場合との比較はしていませんが、メモリ消費量が、状態遷移表の数や大きさに比例するのはやむを得ないとして、概ね納得のできるサイズに収まっていると判断しています。もしかしたら、今回は、ある程度、どんなコードに落とされるかを意識しながら、状態遷移表を作成したのが良かったのかもしれません。ただ、今後の展開の中では、そうでない場合はどうなのか、という検証もしておかなければならないと思っています。エンジニアのスキルへの依存度が小さいツールが望ましい、と考えるからです。

まとめ

今回は、ZIPCの豊富な機能のほんの一部分を利用したに過ぎませんが、現時点では、ZIPCは、制御用ソフトウエア開発にとって有効なツールであると認識しています。

今後、改善を希望するとすれば、以下の二点を申し上げたいと思います。

弊社では、現在、開発の最終段階を迎えています。納期がかなりタイトになってきたため、直接、プログラムに修正を加えています。ある程度目処が立ったところで、整理しようと考えていますが、状態遷移表まで戻るデバッグサイクルというのは、やはり時間がかかかるものです。また、弊社で採用したCPUは、ZIPCデバッガーのサポート対象外だったということもあり、デバッグ時に複数のツールを使用しなければならない煩わしさもありました。そういった意味で、今後は、ICEメーカーやツールメーカーなどと連携して、できるだけ多くのCPUをサポートした、統合的な開発環境を提供していただけると、さらに便利になるのではないかと期待しております。

また、商品開発に携わっていると、開発スケジュールに教育プログラムを入れることは、なかなか困難です。したがって、新しいツールであっても、すみやかに立ち上げなければなりません。その意味では、もう少し詳細なマニュアルがあると便利かとも思いました。また、新人教育を考えると、状態遷移表の作成手引きのようなものがあると良いとも思いました。また、ZIPCで作成されたソフトも、製品に組み込まれればメーカーの責任になりますので、ZIPCのコード生成仕様書を公開していただけると、なお安心して使用できるのではないかと思っています。

（しみず さとし）

W-CDMA向けSOC開発環境試行実験へのZIPC適用

日本電気(株) NECエレクトロンデバイス システムLSI事業本部 マイクロコンピュータ事業部 システム部
水瀬 晴美

W-CDMA向けSOC開発環境試行実験※1の一環として、設計/検証対象となるCPUソフトウェア※2の開発にZIPCを適用しました。ZIPCがMMI(Man-Machine Interface)の設計に適しているのは既に知られているので、ここでは専用ハードウェアに依存したドライバ部の設計に使用しています。これにより、SOC開発環境に適用する際の有効性や課題について述べたいと思います。

1．目的

ZIPCを適用した目的は次の３点です。

(A) 設計手法の有効性

(B) 早期検証の有効性

(C) 自動生成の有効性

2．システム概要

本試行実験におけるシステム構成を図 1 に示します。ベースバンド受信部は、ハードウェアブロック的な意味から BBIC(ベースバンド集積回路部)とも呼びます。本来W-CDMA基地局が生成したI/Qデータを入力として、止まり木チャネルデータの検出・受信を行います。「セルサーチ部」、「パスサーチ部」、「フィンガー部」及び「Rake 処理部」の４つのハードウェアブロックから構成されます。

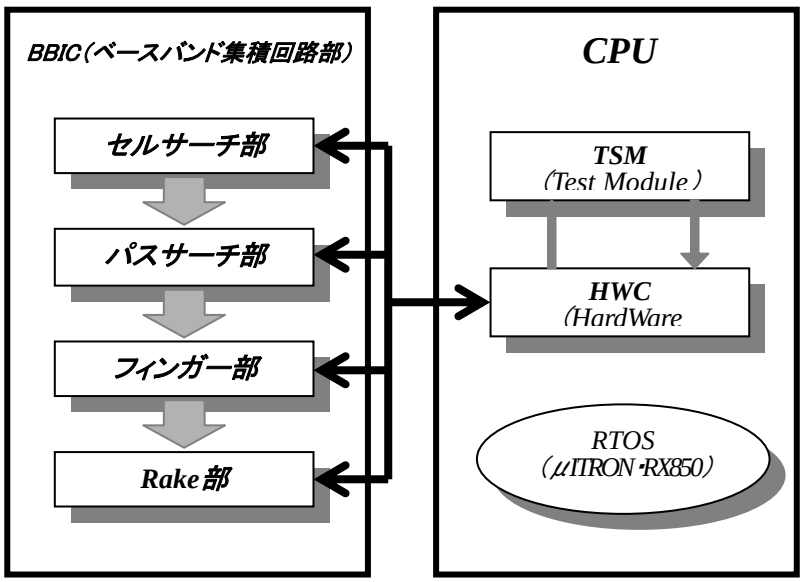


図１ システム概要

※1 本件で定義されるソフトウェア、ハードウェアは、試行実験を目的とするものであり、実システムへの流用等は考慮していません。よって、機能構成等は実システム(Program Interface Description for W-CDMA Mobile Station – Experimental System(Phase2)-:Ver.0.2 DoCoMo 発行)を参考にはしていますが、基本的に試行実験専用の独自仕様として位置付けます。
※2 ターゲットCPUはV850

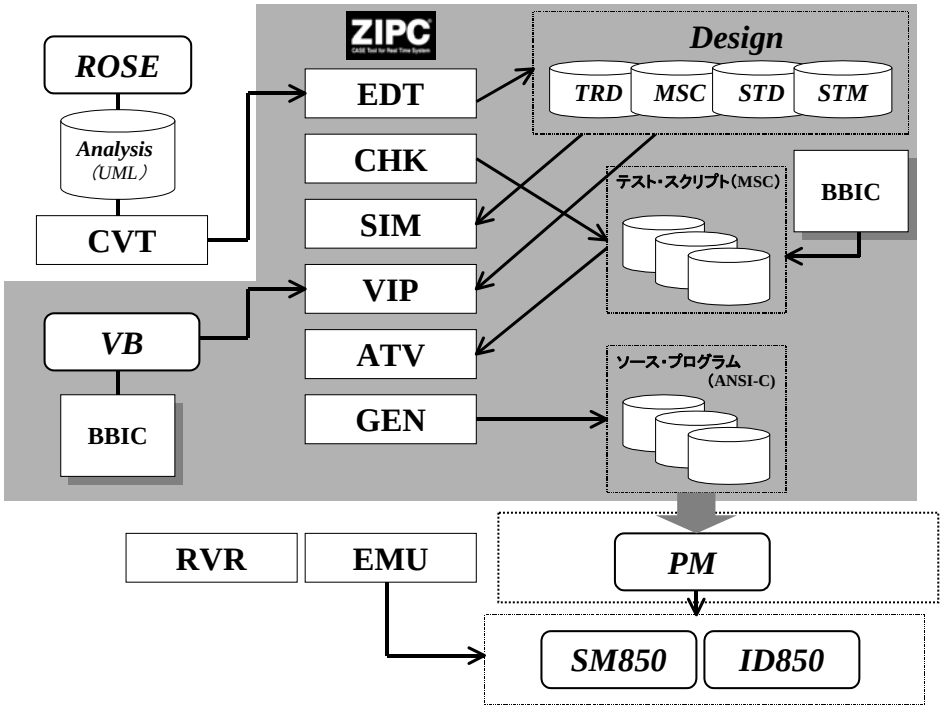


図 2　ZIPC 機能概要

CPUはV850 です。ソフトウェアは、RTOS(μITRON:RX850)上に構築される二つのタスク「TSM:Test Module」と「HWC:HardWare Control」からなります。「TSM」は「HWC」をテストするためのタスクですが、本稿では触れません。

3．環境概要

ZIPC には幾つかの機能が存在します。本稿で取り上げる機能は、図 2 で示す囲まれた内部のものです。枠外の上流ツール(UML※3 コンバート)や下流ツール(システム・シミュレータ／統合ディバッガなど)との連携機能については触れません。「EDT」は設計書エディタであり、タスク関連図(TRD: Task Relationship Diagram)、メッセージシーケンスチャート(MSC)、状態遷移表(STM: State Transition Matrix)や状態遷移図(STD: State Transition Diagram)等の表記法を用いて設計することができます。「CHK」はチェッカであり、前述のドキュメント群の静的な構文解析、整合性をチェックします。「SIM」はシミュレータであり、STM により記述されたステートマシンモデルをμITRON のタスクとしてシミュレートします。「ATV」(Auto Test and Verification)は自動試験／自動評価機能です。スクリプトを用意することで自動的にイベントをシミュレータ上の被タスクに対して発行し、その結果のログと検証スクリプトとを比較するものです。試験・検証スクリプトは MSC や TC(Timing Chart)で記述します。「VIP」(Visual Interface Prototyper)は Microsoft・ Visual Basic (以下 VB と記す) で作成した仮想ターゲットを「SIM」と連動させるものです。「GEN」とはジェネレータで、STM から ANSI-C のプログラム

※3 Unified Modeling Language

コードを自動生成します。

4．設計

ここでは、実際に設計した手順を示します。本手順は Trial 適用に限ったものでなく、実際の開発にも参考となるでしょう。ZIPC ではこのような手順に関して厳密なルールや手法がありませんので、ある程度の経験が必要と思われます。

［設計の手順］

(1) 要求仕様からシステムレベルの MSC を記述し、システムレベルのビヘイビアを掴む。（添付資料 A）
(2) システムレベルの MSC からシステムレベルの STD を記述し、より深くシステムレベルのビヘイビア、例えば並列状態等を洗い出す。（添付資料 B）
(3) システムレベルの STD を STM にコンバートし、システムレベルでの漏れ抜けを防止する。（添付資料 C）
(4) システムレベルの動作が明確になった時点で、HWC と BBIC に着目し、タスクレベルの MSC を設計する。（添付資料 D）
(5) タスクレベルの MSC からタスクレベルの STM を設計する。この段階で事象の型を定義する。（添付資料 E）

以上で HWC の設計が終了です。HWC がどのように動作するかが MSC と STD でまとめられ、矛盾や漏れ無く設計されているかが STM で表記されています。ZIPC ではこのようなドキュメントをツリー上で管理することができます。

5．検証

ZIPC で HWC を検証するには、様々な選択肢があります。選択肢は大きく３つのグループに分けられます。

1) 仮想ターゲット
2) STM レベル
3) 自動試験・検証

実際の BBIC の完成を待たずに CPU ソフトウ

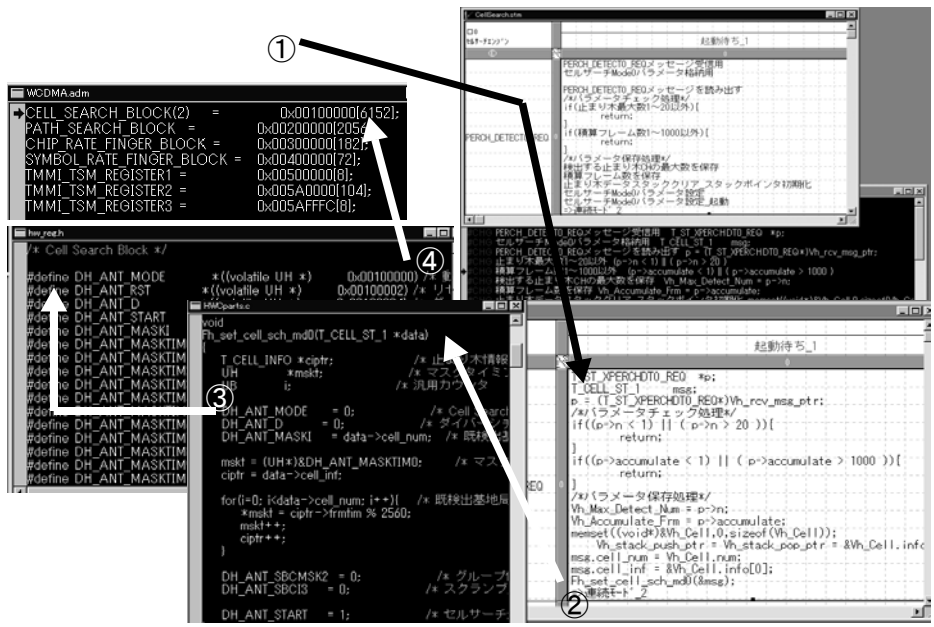


図３ SIM 定義

ェア側を設計書レベルで早期に検証できることは、次の点において有効です。

- BBIC と CPU ソフトウェア側の I/F を設計段階で検証可能
- CPU ソフトウェア側の論理設計が設計段階で完了
- CPU ソフトウェア側のタスク分割方針や BBIC の処理性能を考慮した設計が可能
- 既存関数を取り入れた形での検証が可能

以下、検証における手順を選択肢グループに分けて説明します。

5-1.仮想ターゲット

仮想ターゲットとは、CPU ソフトウェアと外部事象の関係を記述するものです。ここでは、BBIC を仮想ターゲットとします。ZIPC では、仮想ターゲットを表現する方法が二つあります。一つは BBIC を VB で記述する方法、もう一つは MSC スクリプトで BBIC の挙動を記述する方法です。それぞれ、VIP または ATV と SIM を接続して挙動をシミュレーションすることができます。本稿では VB による VIP 接続を選択します。

[VB の手順]

1. VB の Form 上に BBIC を描く。(添付資料 F 参考)
2. VB の Form 上に SIM とコミュニケーションをとるための OCX 部品(ZIPC により供給される)を配置する。
3. BBIC のビヘイビアを VB で記述する。

本項ではもう少し詳細に、SIM-VIP-VB の連携部分について説明します。図 3 に VB と接続するために SIM 側で必要なことは、以下の 4 点です。

[SIM 側の手順]

1. 日本語で記述された STM 内のアクションを置換するための定義を行う。
2. BBIC をアクセスする関数を C 言語で用意する。
3. BBIC レジスタマッピングを定義する。
4. SIM に BBIC レジスタ領域を知らせるためにアドレスマップを定義する。

[VIP 側の手順] (図 4)

VIP 側では VB で配置したそれぞれの BBIC のレジスタ・オブジェクトを選択します。それぞれのレジスタ・オブジェクトのポート名称にアドレスをマッピングします。これにより、SIM と VIP がリンクされることになります。つまり SIM 側で BBIC の IO レジスタ空間としてして定

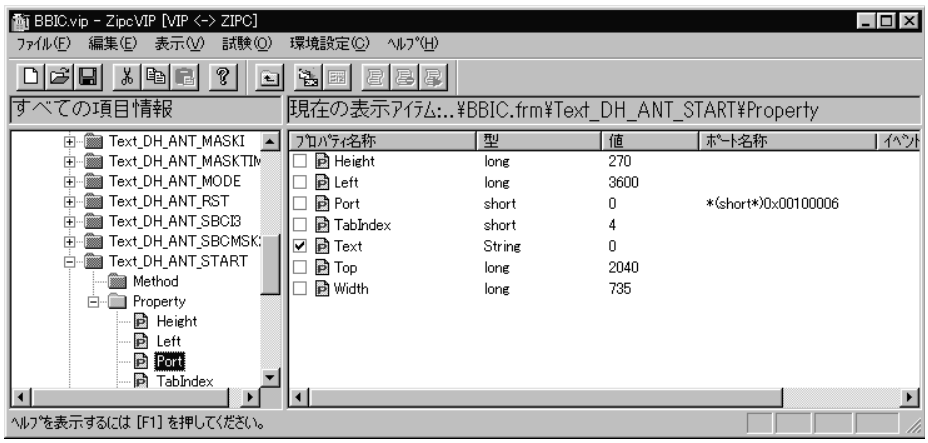


図 4　VIP 定義

義された領域にアクセスが行われると VIP を呼び出すことができます。(リスト 1)

[VB 側の手順]

ZIPC の OCX により ZvipComm1_GetVipEvent()を使用し、VIP からのイベントを受け取ることができます。VIP は VIP 上で定義されたオブジェクト名称を VB に受け渡します。

CPU ソフトウェアで BBIC のレジスタへ書き込まれた際の BBIC の挙動を VB でモデリングします。ライトの一定時間（ここでは 100μSec）後に CPU に割り込みを発生させる場合は、SIM にタイムアウトイベントを VB から要求し、一定時間経過後、SIM からタイムアウトの通知を VB が受けると、VIP に割り込み発生を要求し、SIM に VIP から割り込みを発生させます。BBIC から CPU 方向へのアクセスは ZVipComm1.SetVipEvent を使用します。(リスト 2)

5-2.STM レベル

ZIPC では設計書記述レベルをあるルールに基づいた自然言語（日本語）レベルと ANSI-C レベルの記述を選択することができます。本適用では、BBIC にアクセスする個所が既にライブラリ部品として存在したので ANSI-C レベルで行うことにしました。この際、必要とする情報ファイルは、5-1 で述べたように自然言語を C 言語に置換するための置換情報ファイルです。直接 STM に C 言語を記述しても良いですが、今回は置換して日本語の STM を見ながらシミ

```
Private Sub ZVipComm1_GetVipEvent(ByVal strEvtName As String, ByVal varEvtData As Variant)
    Dim MyString

    If strEvtName = "Text_DH_ANT_MODE.Port" Then
            MyString = CStr(varEvtData)         ' MyString には、DH_AND_MODE
            Text_DH_ANT_MODE.Text = MyString
            Line_DH_ANT_MODE.BorderColor = &HFF              ' &H80000008
    ElseIf strEvtName = "Text_DH_ANT_START.Port" Then
            MyString = CStr(varEvtData)
            Text_DH_ANT_START.Text = MyString
            Line_DH_ANT_START.BorderColor = &HFF
            '割込み発生待ち
            Call CellEndInterrupt
```

リスト 1

```
    Private Sub CellEndInterrupt()
        'ZIPC Simulator に 100 マイクロ後に VB にタイマを入れさせる
        ZVipComm1.SetVipTimer 0, 100000000

    End Sub

    Private Sub ZVipComm1_VipTimerTick(ByVal nTimerID As Long)
        If nTimerID = 0 Then
            'ここで割込みを発生させる
            ZVipComm1.SetVipEvent "TextINTERRUPT.Port", 1
            LineINT1.BorderColor = &HFF
            LineINT2.BorderColor = &HFF
            LineINT3.BorderColor = &HFF
        End If

    End Sub
```

リスト 2

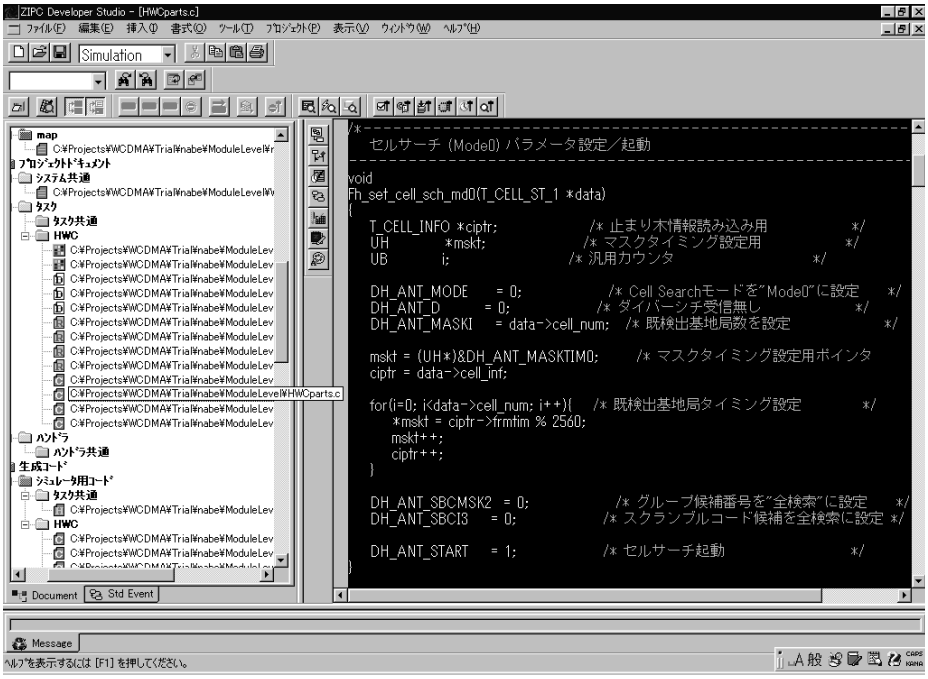


図 5　既存関数の取り込み

ュレーションを行える方式を選択しました。C 言語レベルといっても置換情報を用意すれば見かけは日本語レベルでデバックを進めることが可能です。BBIC アクセス部分の既存 C 言語関数はまったく変更せずに、そのまま ZIPC プロジェクトに登録してシミュレーションができます。(図 5)

5-3.自動試験・検証

本適用においては時間的な問題で、ATV の自動試験機能のみ適用しました。自動検証機能に関しては本項では触れません。

HWC は TSM からのコマンド要求を受けて BBIC に対してアクセスを開始します。これをシミュレーションする際に STM から直接イベントを発生させるやり方と、MSC によりイベントを自動的に発生させる方式を選択することができます。MSC による試験スクリプトは添付資料 G を参照して下さい。

6．自動生成

ZIPC は４つのコード生成方式を持っていて、オプション選択することができます。ZIPC による自動生成とプログラマによる開発のコードサイズ比較を表１に示します。

RAM について調査していないのは、ほとんど ZIPC とプログラマで差がないためです。

ROM 効率において、一般にアセンブラから C 言語に変換した場合のサイズアップが通常 1.3 倍といわれるのを考えると、プログラマ記述 C 言語からのサイズアップは約 1.2 倍というのは現実的な数値であり、プログラミング工数や後の保守を考えれば十分適用の範囲と云えます。

7．設計/検証に要する時間

本適用は、ZIPC 経験者と未経験者の二人で取り組みました。携帯端末のドライバ設計は二

表 1

	実行コード	const	Total
ZIPC	7508	336	7844
ZIPC(標準型生成)	7404	1072	8476
元の	6376	344	6720
比率(Offset 型生成)	*1.18	*0.97	*1.17
比率(標準型)	*1.16	*3.12	*1.26

単位 byte

人とも初めてです。検証までの環境を構築するのに要した時間は、以下の通りです。

- 要求仕様→システムレベルの MSC 化：3 日
- MSC→STD 化：3 日
- STD→STM 化：0.5 日
- タスクレベルの MSC 化：1 日
- タスクレベルの STM 化：1 日
- 仮想ターゲット環境構築：1 日

設計には時間を要します。この時間が、大規模化したシステム設計の後工程で不具合が発見された際にかかるフィードバックを低減することになります。

また、試験スクリプト一つを流すのにかかるシミュレーション時間は１分程度です。抽象化レベルの検証が有効といえる一つの数値だと思います。

8．今後の課題

ZIPC を SOC の CPU ソフトウェアに適用するには、今後以下のような課題があります。

まずは、ZIPC へお願いしたいことです。

1) 仮想ターゲットのモデリングは C 及び C++ が主流であり、それに対応できるようにしてほしい。
2) SOC として主流になっている CPU+DSP 等マルチプロセッサに対応してほしい。
3) シミュレーション速度向上のため、Visual C++を使用した Native Compile 環境を充実させてほしい。そのためには、Windows 上 RTOS の開発とその環境をサポートしてほしい。
4) 通信系の仕様表現の主流である SDL からのインポートに対応してほしい。

半導体ベンダとしての下流工程へのソリューションは、以下のように考えます。

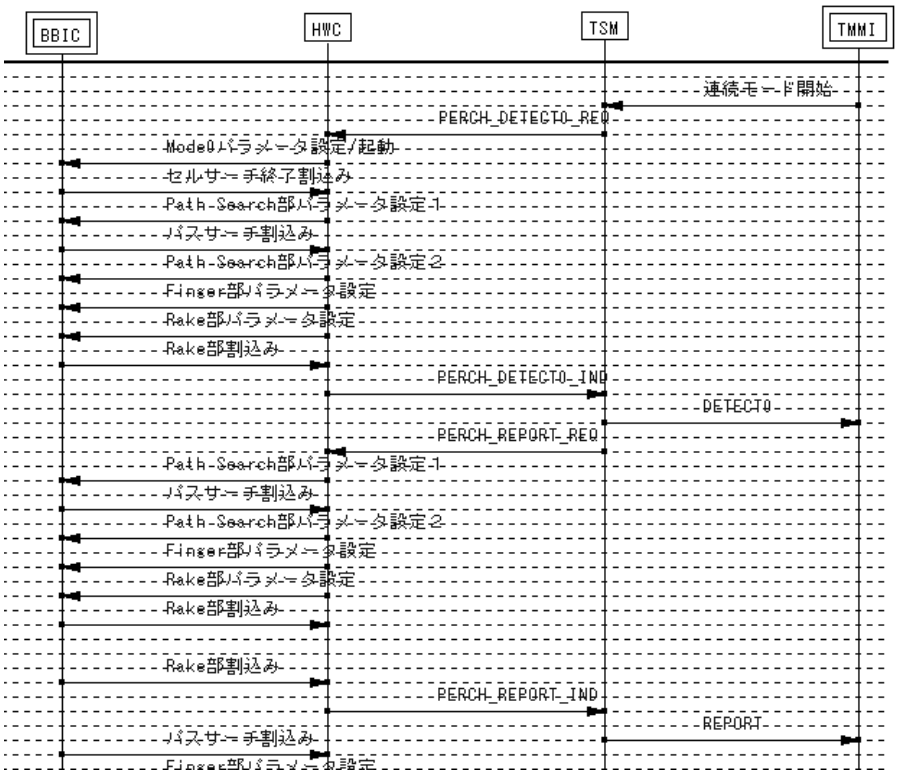
5) 上記 1)に対応して、仮想ターゲットを接続できるコードシミュレーション環境の実現。
6) 最後に実機環境の試験・検証と接続。

仮想ターゲットである BBIC を VB でモデル化するのは限度があり、デバイスの設計環境で構築したモデルと共有化すること、また上流環境で使用した仮想ターゲットを使用できるシミュレータの存在が重要になってきます。

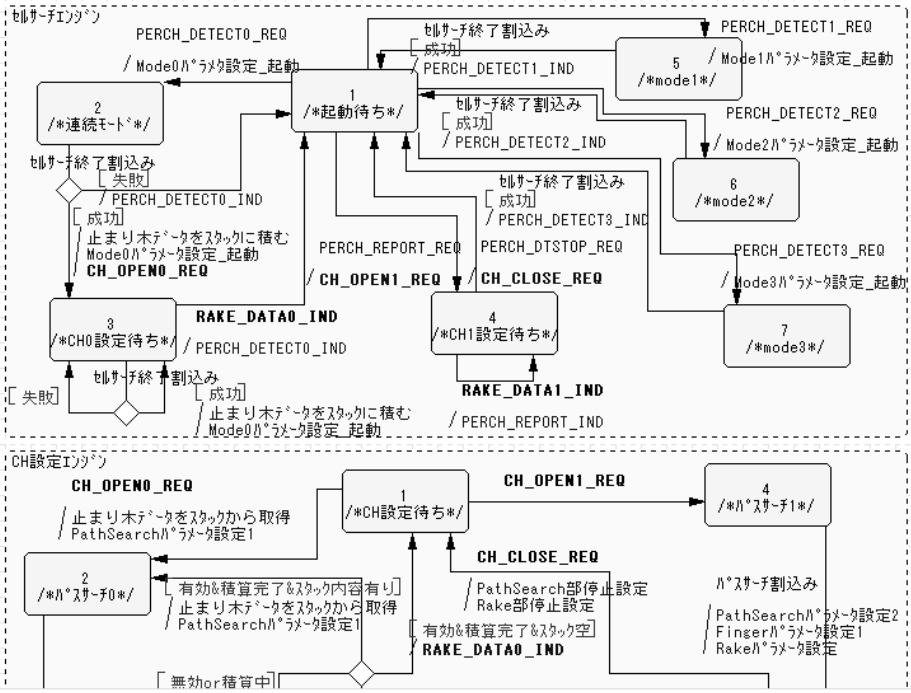
更には、今後いかに実機環境の試験・検証と接続できるかが需要です。上流環境で流した試験スクリプトを実機環境でも同じように流せ、かつシミュレーション時の結果と実機上の結果を自動的に付き合わせる機構が必要です。

（みずせ はるみ）

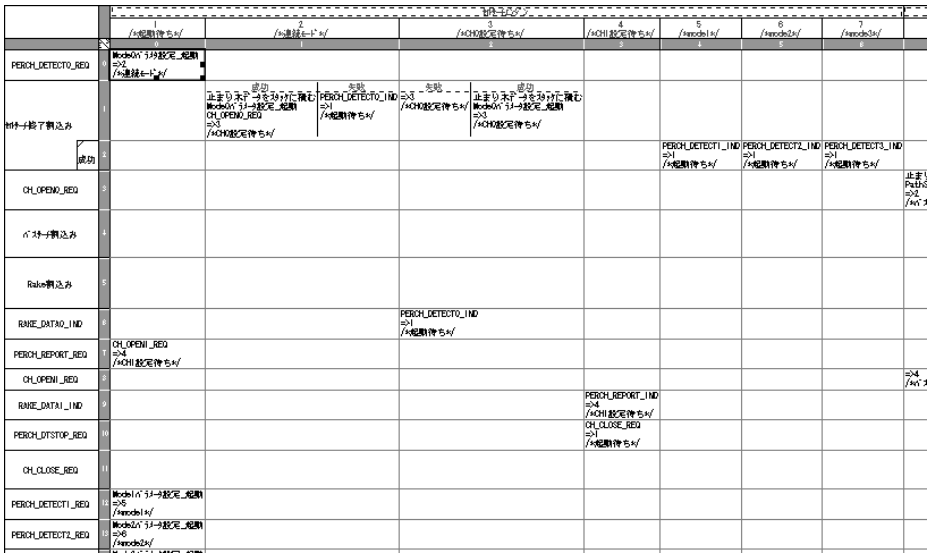
添付資料 A


BBIC
HWC
TSM
TMMI
連続モード開始
PERCH_DETECT0_REQ
Mode0パラメータ設定/起動
セルサーチ終了割込み
Path-Search部パラメータ設定1
パスサーチ割込み
Path-Search部パラメータ設定2
Finger部パラメータ設定
Rake部パラメータ設定
Rake部割込み
PERCH_DETECT0_IND
DETECT0
PERCH_REPORT_REQ
Path-Search部パラメータ設定1
パスサーチ割込み
Path-Search部パラメータ設定2
Finger部パラメータ設定
Rake部パラメータ設定
Rake部割込み
Rake部割込み
PERCH_REPORT_IND
REPORT
パスサーチ割込み
Finger部パラメータ設定

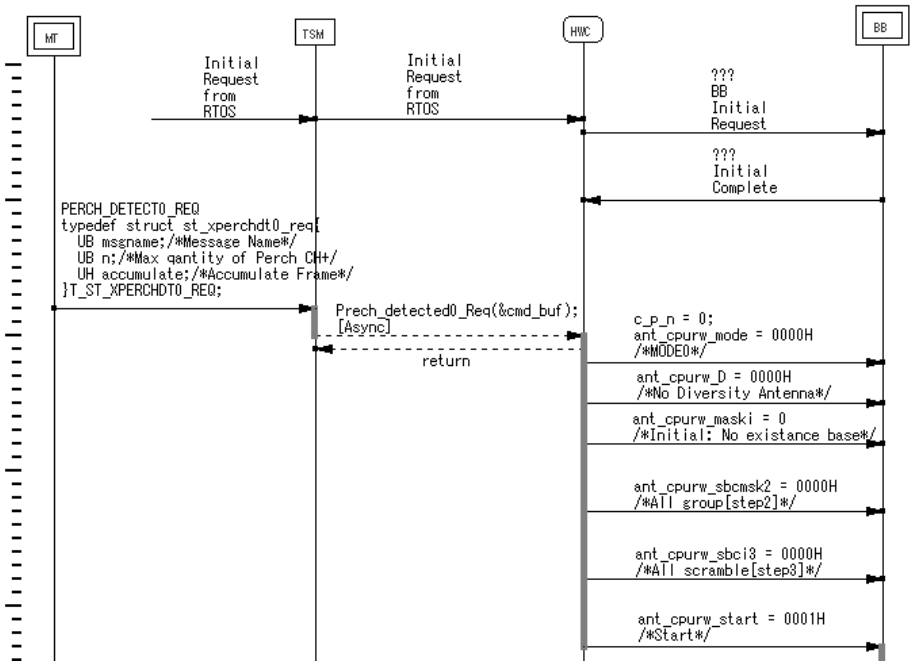
添付資料 B


ｾﾙｻｰﾁｴﾝｼﾞﾝ
PERCH_DETECT0_REQ
/ Mode0ﾊﾟﾗﾒﾀ設定_起動
ｾﾙｻｰﾁ終了割込み
[成功]
/ PERCH_DETECT1_IND
PERCH_DETECT1_REQ
/ Mode1ﾊﾟﾗﾒｰﾀ設定_起動
5
/*mode1*/
2
/*連続ﾓｰﾄﾞ*/
1
/*起動待ち*/
ｾﾙｻｰﾁ終了割込み
[成功]
/ PERCH_DETECT2_IND
PERCH_DETECT2_REQ
/ Mode2ﾊﾟﾗﾒｰﾀ設定_起動
6
/*mode2*/
ｾﾙｻｰﾁ終了割込み
[失敗]
/ PERCH_DETECT0_IND
[成功]
止まり木ﾃﾞｰﾀをｽﾀｯｸに積む
Mode0ﾊﾟﾗﾒｰﾀ設定_起動
CH_OPEN0_REQ
ｾﾙｻｰﾁ終了割込み
[成功]
/ PERCH_DETECT3_IND
PERCH_REPORT_REQ
/ CH_OPEN1_REQ
PERCH_DTSTOP_REQ
/ CH_CLOSE_REQ
PERCH_DETECT3_REQ
/ Mode3ﾊﾟﾗﾒｰﾀ設定_起動
3
/*CH0設定待ち*/
RAKE_DATA0_IND
/ PERCH_DETECT0_IND
4
/*CH1設定待ち*/
7
/*mode3*/
ｾﾙｻｰﾁ終了割込み
[失敗]
ｾﾙｻｰﾁ終了割込み
[成功]
止まり木ﾃﾞｰﾀをｽﾀｯｸに積む
Mode0ﾊﾟﾗﾒｰﾀ設定_起動
RAKE_DATA1_IND
/ PERCH_REPORT_IND
CH設定ｴﾝｼﾞﾝ
CH_OPEN0_REQ
/ 止まり木ﾃﾞｰﾀをｽﾀｯｸから取得
PathSearchﾊﾟﾗﾒｰﾀ設定1
1
/*CH設定待ち*/
CH_OPEN1_REQ
4
/*ﾊﾟｽｻｰﾁ1*/
2
/*ﾊﾟｽｻｰﾁ0*/
[有効&積算完了&ｽﾀｯｸ内容有り]
/ 止まり木ﾃﾞｰﾀをｽﾀｯｸから取得
PathSearchﾊﾟﾗﾒｰﾀ設定1
CH_CLOSE_REQ
/ PathSearch部停止設定
Rake部停止設定
ﾊﾟｽｻｰﾁ割込み
/ PathSearchﾊﾟﾗﾒｰﾀ設定2
Fingerﾊﾟﾗﾒｰﾀ設定1
Rakeﾊﾟﾗﾒｰﾀ設定
[有効&積算完了&ｽﾀｯｸ空]
RAKE_DATA0_IND
[無効or積算中]

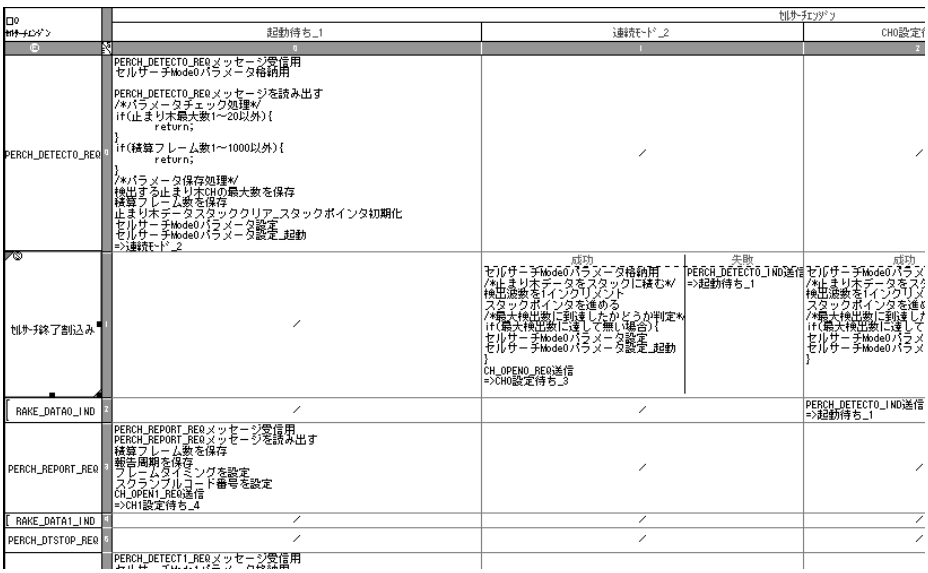
添付資料 C



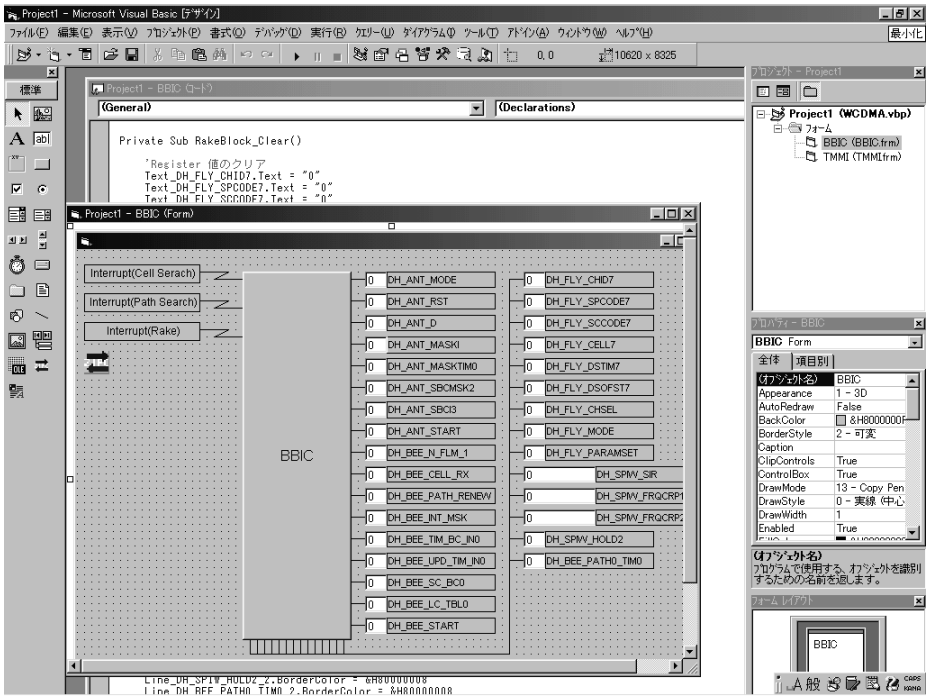
添付資料 D


MT
TSM
HWC
BB
Initial
Request
from
RTOS
Initial
Request
from
RTOS
???
BB
Initial
Request
???
Initial
Complete
PERCH_DETECT0_REQ
typedef struct st_xperchdt0_req{
UB msgname;/*Message Name*/
UB n;/*Max qantity of Perch CH*/
UH accumulate;/*Accumulate Frame*/
}T_ST_XPERCHDT0_REQ;
Prech_detected0_Req(&cmd_buf);
[Async]
return
c_p_n = 0;
ant_cpurw_mode = 0000H
/*MODE0*/
ant_cpurw_D = 0000H
/*No Diversity Antenna*/
ant_cpurw_maski = 0
/*Initial: No existance base*/
ant_cpurw_sbcmsk2 = 0000H
/*ATI group[step2]*/
ant_cpurw_sbci3 = 0000H
/*ATI scramble[step3]*/
ant_cpurw_start = 0001H
/*Start*/

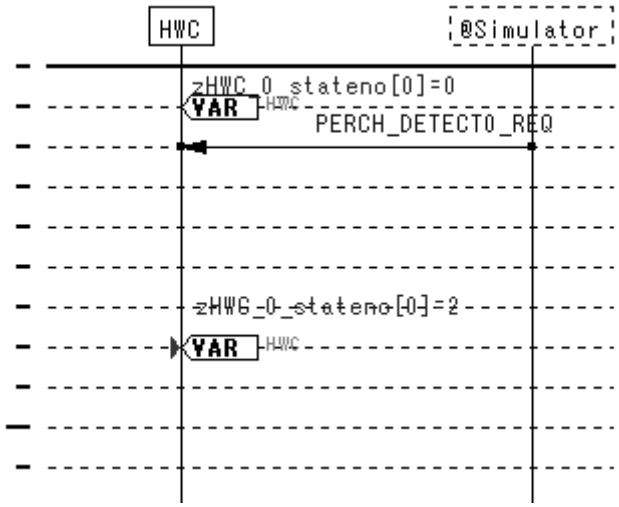
添付資料 E


起動待ち_1
連続ﾓｰﾄﾞ_2
PERCH_DETECT0_REQ
PERCH_REPORT_REQ
RAKE_DATA0_IND
RAKE_DATA1_IND
PERCH_DTSTOP_REQ
成功
失敗
PERCH_DETECT0_IND送信
=>起動待ち_1
CH_OPEN0_REQ送信
=>CH0設定待ち_3
CH_OPEN1_REQ送信
=>CH1設定待ち_4

添付資料 F


Project1 - Microsoft Visual Basic [デザイン]
Project1 - BBIC (Form)
Interrupt(Cell Serach)
Interrupt(Path Search)
Interrupt(Rake)
BBIC
DH_ANT_MODE
DH_ANT_RST
DH_ANT_D
DH_ANT_MASKI
DH_ANT_MASKTIM0
DH_ANT_SBCMSK2
DH_ANT_SBCI3
DH_ANT_START
DH_BEE_N_FLM_1
DH_BEE_CELL_RX
DH_BEE_PATH_RENEW
DH_BEE_INT_MSK
DH_BEE_TIM_BC_IN0
DH_BEE_UPD_TIM_IN0
DH_BEE_SC_BC0
DH_BEE_LC_TBL0
DH_BEE_START
DH_FLY_CHID7
DH_FLY_SPCODE7
DH_FLY_SCCODE7
DH_FLY_CELL7
DH_FLY_DSTIM7
DH_FLY_DSOFST7
DH_FLY_CHSEL
DH_FLY_MODE
DH_FLY_PARAMSET
DH_SPMV_SIR
DH_SPMV_HOLD2
DH_BEE_PATH0_TIM0
Project1 (WCDMA.vbp)
BBIC (BBIC.frm)
TMMI (TMMI.frm)
BBIC Form

添付資料 G


HWC
@Simulator
zHWC_0_stateno[0]=0
VAR
HWC
PERCH_DETECT0_REQ
zHWC_0_stateno[0]=2
VAR
HWC

ZIPC の UML によるモデル化について

日本電気（株）NECエレクトロンデバイス システムＬＳＩ事業本部 システムＬＳＩ設計技術本部 設計システム部
川口 晃

1. はじめに

近年、マイコン組込みシステムの高機能、短納期化が進んでおり、その上に組み込まれるソフトウェアにも同様なことが言える。これに応じて、組込みソフトウェアの開発ツールも高機能、複雑化している。そこで、本稿では組込みシステム開発に最適なツール構築のため、ツール自身をモデル化し要求仕様の定式化を試みた例を紹介する。対象ツールとして状態遷移設計ツール ZIPC を、モデル化の記法として UML(Unified Modeling Language)を使用した。

2. ZIPC のモデル化

ZIPC のモデル化は、次の 2 段階について行なった。まず、ツール自身の内部の振る舞いをモデル化すること。これは、ZIPC を実現する立場で、ZIPC というツールが内部でどのような処理をどのようなデータ構造を用いて行なっているのかを、明確に定式化する目的で行なった。次に、その内部のモデルを参照しながら、ZIPC ユーザがツールを使用する際、ユーザと ZIPC がどのように振る舞うのかをモデル化した。

2.1 ZIPC の内部モデル

■ 外部インタフェースと内部の振る舞いを規定

最初に、アクタとユースケースを用いて ZIPC の外部インタフェースと内部の振る舞いを規定した。アクタとは、ヤコブソンの OOSE(参考文献)でシステム外部にあってシステムに刺激を与えるものと定義されている。また、アクタの刺激を受けてシステムが示す振る舞いをユースケースという単位で扱う。

まず ZIPC から見て、アクタは何か、またそのアクタからの刺激を受け、ZIPC が行なう振る舞い、すなわちユースケースは何かを定式化

◆アクタと状態遷移表◆

アクタと状態遷移表の関係

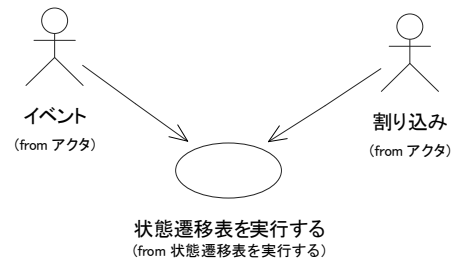


図2-1　アクタの定義

した。図 2-1 に、UML で表記したアクタとユースケースの定義を示す。

UML では、人の形をしたオブジェクトがアクタを示し、ユースケースは、楕円形で表わされる。ここでは、ZIPC の振る舞いを［状態遷移表を実行する］というユースケースで代表させ、そのユースケースに関係するアクタとして、［イベント］と［割込み］という 2 種類のアクタを定義している。これまで、このような情報は［暗黙の了解］、［規定の事実］、［個人の認識］など、あいまいに扱われる傾向があった。このような情報を、モデルとして可視化、定式化することで、モデルがどのような立場、視点で作成されているかを明確にすることができる。また、ツール開発に携わる人々（仕様決定者、開発者など）の開発対象の理解度の向上、開発者間で明示的な情報の共有を図ることができる。

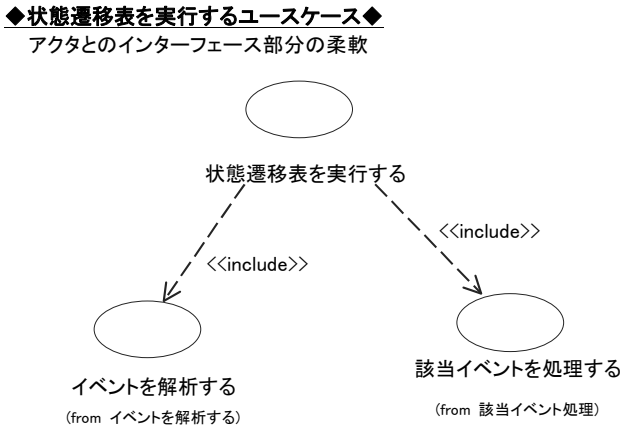


図2-2 ［ 状態遷移表を実行する ］ユースケースの分析

■ ZIPC の振る舞いの分析

次に、ZIPC の振る舞いを代表する［状態遷移表を実行する］というユースケースが、どのような振る舞いの集まりなのかを規定するため、ユースケース図を作成した(図 2-2)。

ユースケース図は、ユースケースの間にどのような関係があるかを示す図である。

この図では、ユースケース間の Include 関係が示されている。これは、ユースケース［状態遷移表を実行する］が、ユースケース［イベントを解析する］、［該当イベントを処理する］を含んでいることを示している。［イベントを解析する］、［該当イベントを処理する］の 2 種類のユースケースがモデル化されたのは、［イベント］アクタとのインタフェースを、［イベントを解析する］ユースケース内部に局所化し、［イベント］アクタと ZIPC とのインタフェースの柔軟性を、より少ないコストで確保するためである。

■ ZIPC の動的な振る舞いを規定

このように、ZIPC が内部でどのような振る

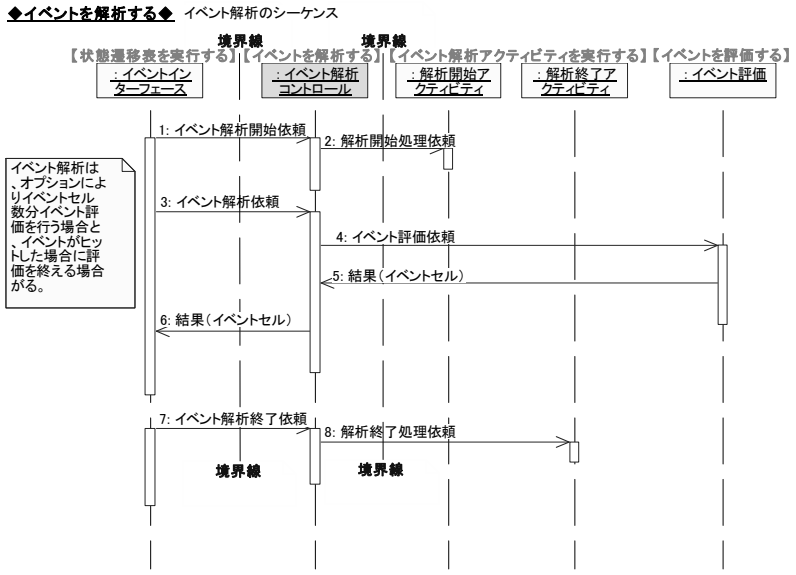


図2-3 イベントを解析する メッセージシーケンスチャート

舞いをするのかをユースケース単位でモデル化していく。この過程で、ユースケースの動的な振る舞いを規定することが必要になる。このため、メッセージシーケンスチャートを記述した。これは、ユースケースの振る舞いを、関係するクラス間のメッセージのやりとりで記述した図で、ユースケースの動的な振る舞いを詳細に表現できる。

■ ZIPC 内部のデータ構造のモデル化

次に、ZIPC 内部に存在するデータ構造をクラス図を用いてモデル化した。図 2-4 で示すのは、状態遷移表を管理するために必要なデータ構造を規定したクラス図である。クラスは四角形で表わされており、クラス間を結ぶ、一端がひし形になっている直線は集約を表わしている。集約は、part-of の関係を示す。このようにして、ZIPC 内部に存在するデータ構造を、クラス図として明示的に規定した。

2.2 内部モデルの効果

このように、アクタ、ユースケース、クラス、さらに、それらの間の関係を示す図（ユースケース図、メッセージシーケンスチャート、クラス図など）を用いて、ZIPC の内部的な振る舞いをモデル上で規定した。これらのモデルは、次の用途に用いることができる。

□設計の効率化

ユースケース間の関係を精密に分析し、再利用できるもの、拡張して使用できるものを抽出したり、クラス構造を見直し、より効率的なデータ構造を構築するなど、モデルを用いてツールの効率的な実現を行なうことができる。

□情報共有

理解しやすいモデルの形でツールの振る舞いを規定しているので、ツールの仕様決定者、開発者間で、齟齬の少ない情報共有を図ることができる。

□仕様書の作成

モデルは、ZIPC の内部構造を網羅的に規定している。モデルを構成するユースケースや、クラスなどの詳細な記述を追加することで、ZIPC の内部仕様書として使用することができる。

2.3 ユーザ側から見た ZIPC のモデル化

前節では、ZIPC を実現する側の立場で、内部モデルの作成を行った。ここでは、ZIPC を使用するユーザの立場で、ユーザと ZIPC の振る舞いのモデル化を行う。ユーザと ZIPC の振る舞いをモデル化するには、ユーザが状態遷移設計するために行なうべき振る舞いを、ZIPC の内部モデルを参照しながら、モデル化してい

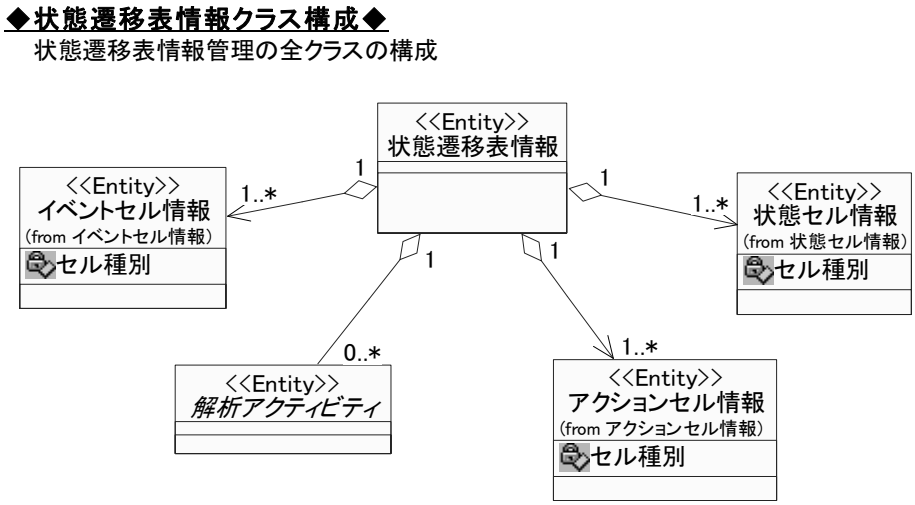


図2-4 データ構造をクラス図で表現

く。

■ ユーザと ZIPC の振る舞いを規定

ZIPC ユーザをアクタとした時の、ユーザと ZIPC の振る舞いをユースケースを用いてモデル化した(図 2-5)。2.2 で示した ZIPC を実現する側の視点と比較すると、モデルがまったく異なっている。どのような立場で、何を表現したいかを明確にすることが、有益なモデルを作成するために必要であることがわかる。

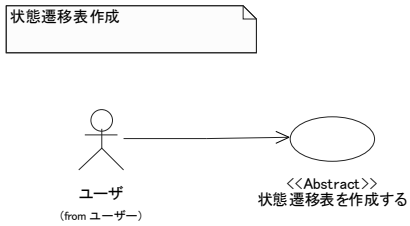


図2-5　ユーザとZIPCの振る舞いを規定

■ ユーザと ZIPC の振る舞いを分析

上述のユーザと ZIPC の振る舞いを、ユースケース図を用いて分析した。図 2-6 に一例を示す。

■ ユーザと ZIPC の動的な振る舞いを規定

次に、ユースケースのうち、動的な振る舞いを明示する必要があるものについてメッセージシーケンスチャートを作成した。図 2-7 に一例を示す。

■ ユーザが定めるべきデータ構造の規定

ZIPC を使用するにあたって、ユーザが定めるべきデータ構造をクラス図を用いて規定した。図 2-8 に一例を示す。

2.4 ユーザの視点に立ったモデル化の効果

□ツールとして不足している機能の洗い出し

ユーザ側の視点で、状態遷移表作成のプロセスをモデル化することで、ツールがサポートするべき機能の洗い出しが容易になる。

□ユーザインタフェースの改善点の洗い出し

ツールを使用するユーザのために、ツール側が、どのようなインタフェースを提供するべきかを議論するためのたたき台として使用できる。

□ユーザマニュアル

ユーザ側の視点で、状態遷移表作成のプロセスをモデル化しており、ユーザマニュアルとして使用できる。

まとめ

以上、簡単に UML を用いた ZIPC のモデル化について述べた。この試みを行なった感想として、次のようなことが言える。

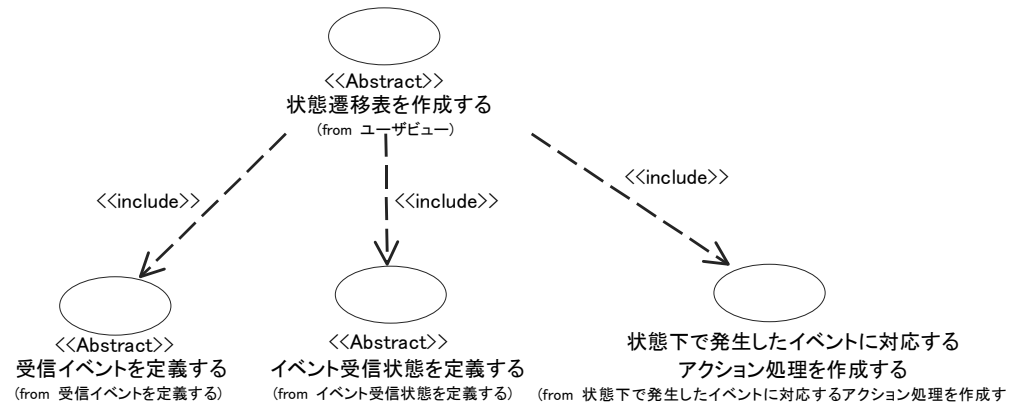


図 2-6 ユーザと ZIPC の振る舞いを分析

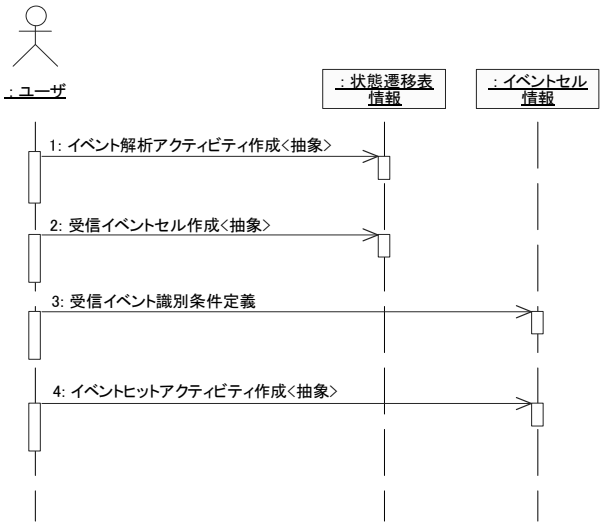


図 2-7 ユーザと ZIPC の動的な振る舞いを規定

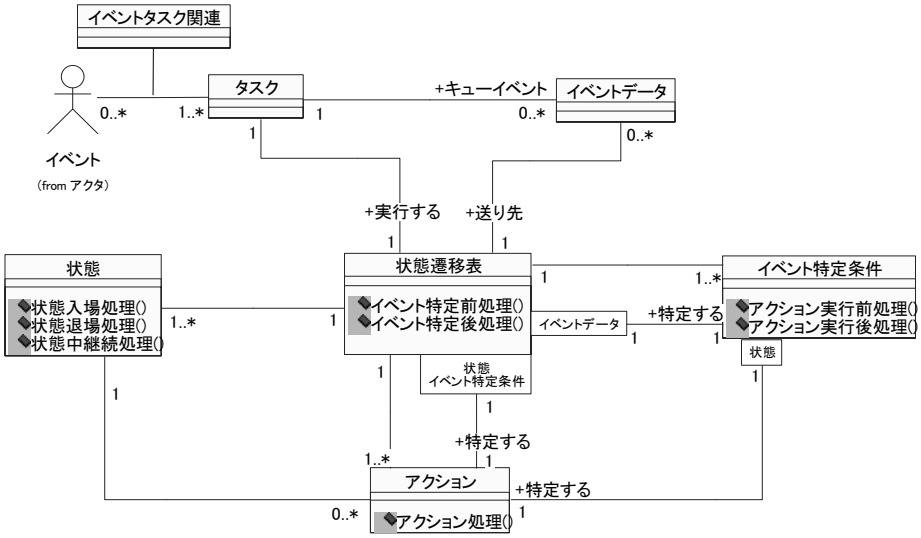


図 2-8　ユーザが定めるデータ構造の規定

□ツールの機能は非常に複雑、多岐にわたっており、全てを正確に理解し、把握することが難しい。

□UML により、ツールの機能やデータ構造、ユーザインタフェースを規定、可視化した内部モデル、ユーザ側のモデルを作成、活用すれば、過不足のない機能の作り込み、見通しの良いツール設計、使いやすいユーザインタフェースの設計が容易になる。

□このようなモデルは、機能仕様書、ユーザマニュアルなど、ドキュメントとしての価値も大きい。

最後に、今回の UML モデルの作成者である CATS の萩原氏と、このプロジェクトの機会を与えていただいた渡辺氏に感謝する。

参考文献

Jacobson,I.et al.:Object-Oriented Software Engneering, Addison-Wesley(1992)

（かわぐち あきら）

ZIPCを要として進化する富士通の SOFTUNE® 連携ソリューション

富士通(株) 電子デバイス事業本部 システムLSIソフトウェア部 基盤ソフトウェア開発部
五十嵐 純

富士通(株)とキャッツ(株)ZIPCとの連携の背景：ここまでのあらすじ

富士通(株)は、10年程前のSA/SD（構造化分析／構造化デザイン）CASEツールの一大ブームの際、富士通(株)独自で開発したCASEツールを提供していました。しかし，SA/SDはリアルタイム系に向いておらず，いつしかSA/SD主体のCASEツールブームは去っていきました。しかしそのなかで、状態遷移表CASEの「ZIPC」は生き残っていました。

一方、依然、我々のお客さまも、Softuneのアセンブラや C レベルより上流の設計ステージで有効なツールを探しておられました。

こうした背景のなか、我々富士通(株)が展示会やお客さまからのご要望などを通して着目したZIPCの利点は次の点でした。

（1）状態遷移表により処理モレのない設計ができる
（2）拡張階層化状態遷移表により構造化設計ができレビューしやすい
（3）ZIPC－VIPによるビジュアルI/Oシミュレータがある

こうして富士通(株)は、1998 年に開発環境Softuneとキャッツ(株)のZIPCを連携させました。(図1参照)

これがいわば第ニ期の連携と言えますが、さらにオブジェクト指向という第三の波をとらえ、富士通(株)のSoftune連携ソリューション

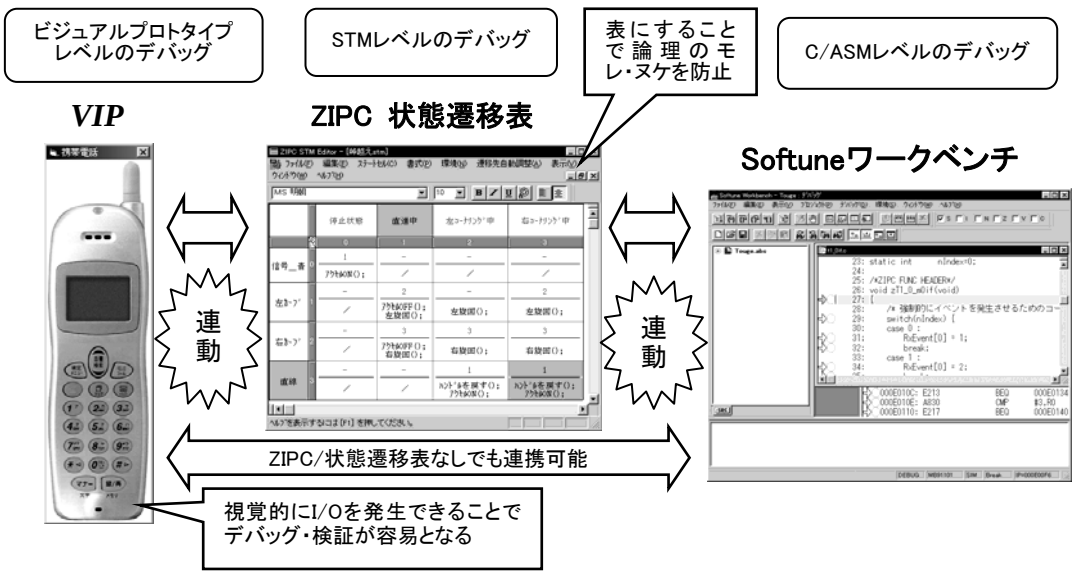


図1 ZIPC - SOFTUNE 連携

はさらなる進化を行っています。

高品質開発のための組織構造：ピラミッドサミットリンクモデル

前述のように、富士通(株)は Softune の C/ASM レベル以上の設計ステージを ZIPC の状態遷移表で設計できるソリューションを開発したことで、組み込みプログラムの実装レベルの開発プロジェクトに対しては、高品質・高効率の開発環境をご提供できるようになりました。

しかし、21 世紀の開発環境は、このソリューションだけでは不十分であると考えています。今後は、エンドユーザ様が望む製品は何かを分析し、その製品を開発するために、

全社組織的に

高効率／高品質に

開発できる必要があります。これを実現するための組織構造として富士通(株)は「ピラミッドサミットリンク」という開発体制モデルをイメージしています。(図 2 参照)

企業内（A 社）のなかには、いくつかの小さな開発プロジェクト（X, Y, Z,,,）があり，それぞれプロジェクトリーダを頂点とするピラミッド構造となっています。ピラミッドのなかの各ステージには、上層から下層に向け順に、

（1） システム全体を分析・設計する「アーキテクト」
（2） 実装レベルの設計を行う「デザイナ」
（3） コードを実装デバッグする「プログラマ」

が存在します。これは別の視点で言うと上層から順に、「ベテラン」「中堅」「フレッシュマン」と言う事ができます。

この企業の外には顧客様がおり、顧客は個人（エンドユーザ）の場合もありますし、発注側の企業組織（B 社）のこともあります。また、

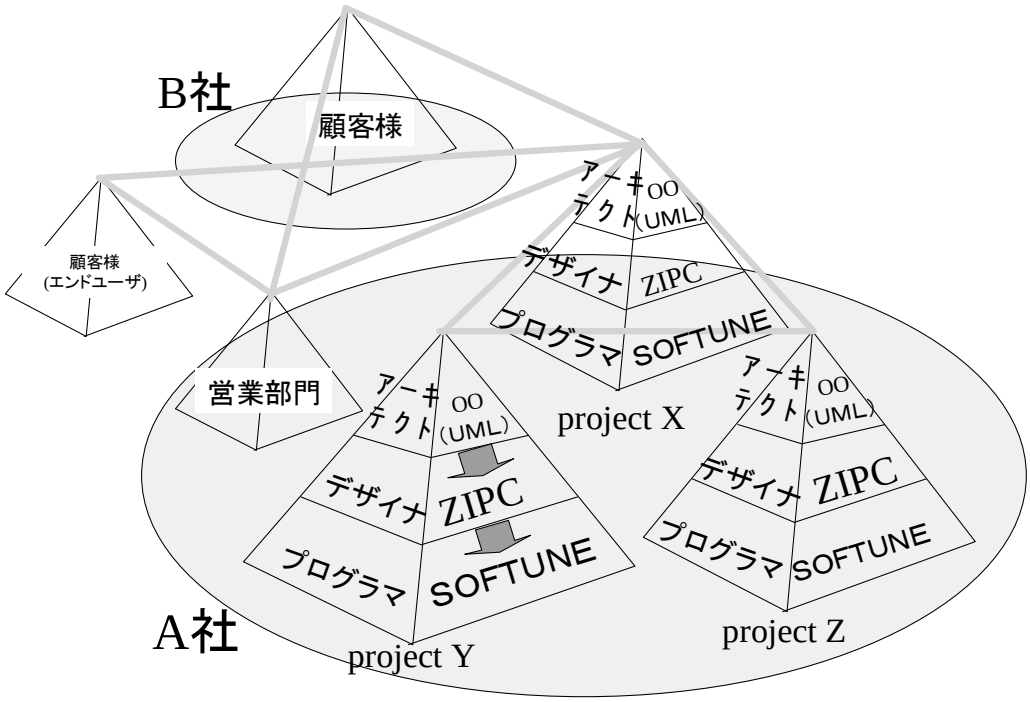


図 2　ピラミッドサミットリンク・モデル

企業内にも開発部門以外の部門、例えば営業部門などのピラミッドも存在します。

これからの組み込みソフト開発においては、顧客の要求を分析し、各プロジェクト間での仕様検討や同期開発を行い、また各プロジェクト内では上流から下流に向け仕様が首尾一貫して流通し、効率よくプログラム開発できなければいけません。

これらの要求に答えるためには、開発組織ピラミッドの頂点のステージには、頂点同士がリンクして意思疎通できる手段・手法が必要になります。また、この手段・手法は頂点ステージで決まった仕様をピラミッドの下層ステージに展開してプログラム開発できることも必要となります。またこうして出来上がったプログラム自体は他プロジェクト（他プログラム）からの影響を受けにくい部品となっていて、拡張性や再利用性に優れている必要があります。

ミッドの頂点にはオブジェクト指向（OO）を適用させるのが有効と考えます。

アーキテクトステージでは、UML（ユニファイドモデリングランゲージ）やステートチャート（状態図）などの手法を用います。UML は、C や C++などに比べ圧倒的に理解できる人の数が増えるため、UML を使うことで製品の仕様、設計は見通しのよいものにできます。また、UML は実装レベルより、主に顧客要求レベルを表現できるドキュメント手法であるため，顧客や営業部門や他の開発部門などのピラミッドの頂点との意思疎通が行いやすくなります。また、ピラミッド内での上流から下流への意思疎通や技術伝達もやりやすくなるため、ピラミッド全体の技術スキルも向上します。フレッシュマンが中堅、ベテランへと成長することにも合った、成長するピラミッド、成長する企業、のためのモデルと考えます。

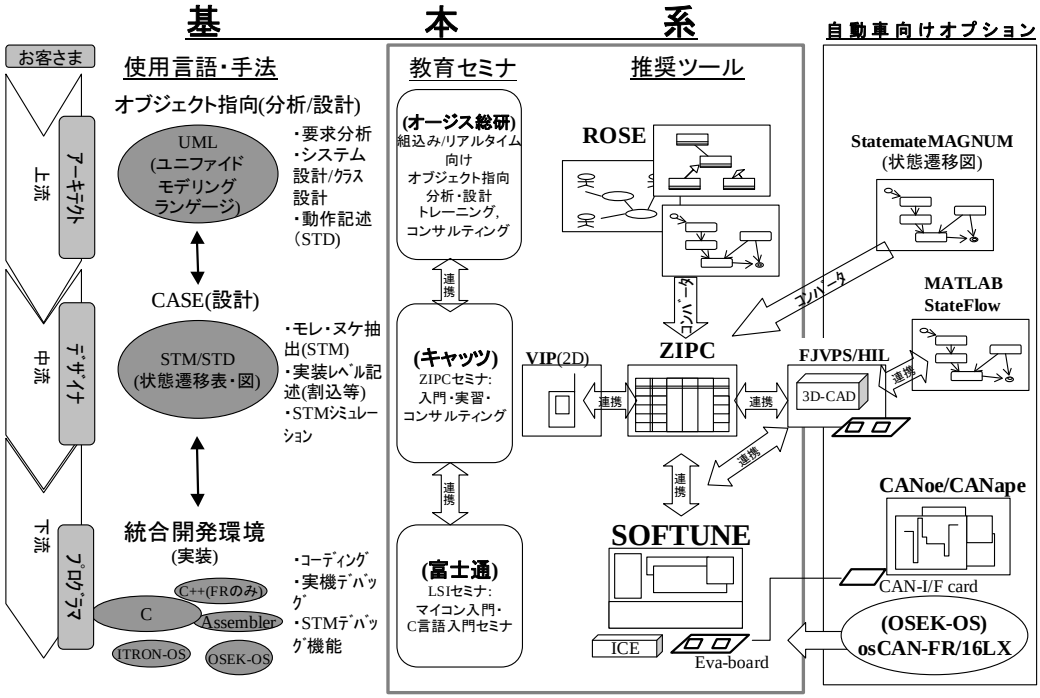


図3 富士通(株)が提案する「これからの組込ソフトウェア開発環境」

以上の要求を満たすためには、開発組織ピラ

富士通(株)が提案する「これからの組み込みソフトウェア開発環境」

前述した考えにそって富士通(株)がご提供する Softune 連携ソリューションは次の通りです。(図 3 参照)

Softune を下流に置き、その上に上流、中流ステージを配置し、それぞれ、オブジェクト指向の UML 手法と状態遷移表手法を用います。

オプション・ソリューションの位置づけとして、主に車業界向けの StatemateMAGNUM および OSEK (RTOS) ソリューションもご用意しています。また、富士通(株)独自の取り組みとして 3D-CAD との連携ソリューションである FJVPS／HIL もご用意しています。

各ステージで採用している具体的なツールと国内でのサポート会社名は下記の通りです。

（１）Softune（富士通(株)）

デバッガ

（２）状態遷移表および 2 次元パネル I/O：ZIPC および ZIPC VIP（キャッツ(株)）

→ セミナ・コンサルティング：キャッツ(株)

（３）UML(ユニファイドモデリングランゲージ）：Rational Rose（米 Rational Software Corporation、日本ラショナルソフトウェア(株)）

→ セミナ・コンサルティング：(株)オージス総研

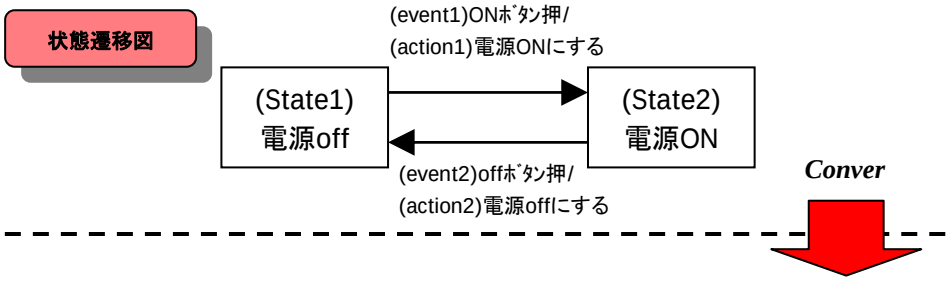
（４）3 次元仮想設計支援シミュレーション：FJVPS/HIL（富士通(株)）

→ サポート・コンサルティング：富士通(株)、富士通デバイス(株)、および(株)FFC

→ データ解析、ブロック線図、状態遷移図：MATLAB・Stateflow（米 The MathWorks Inc.）

（５）状態遷移図：StatemateMAGNUM（米 I-Logix Inc.）

→ セミナ・コンサルティング：伊藤忠テクノサイエンス(株)



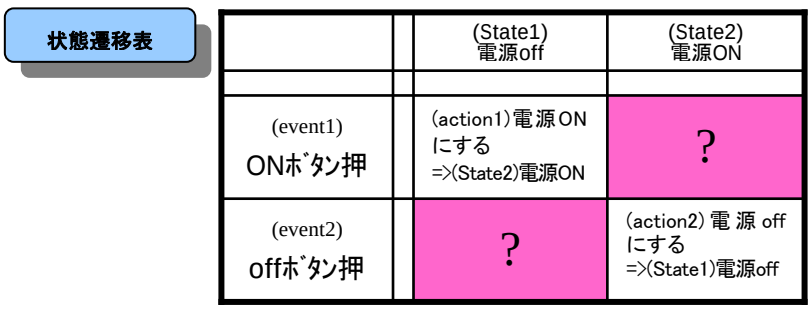


図 4　状態遷移表のメリット

→ MCU、ICE、評価ボード、コンパイラ、

（６）OSEK-OS、CAN 解析：osCAN/CANoe（独 Vector Informatik GmbH.）
→セミナ・サポート：ベクタージャパン(株)

ZIPC の必要性

図をみて分かる通り、富士通(株)のソリューションはほとんどすべて ZIPC を経由するようになっています。なぜ ZIPC なのかについては、一言で言って、設計のモレ・ヌケをとるため、です。(図 4 参照)

図 4 は状態遷移図で書いたものを状態遷移表で書き直した例です。このように簡単な例でも「?」のところが考えモレとなっています。

アーキテクトが用いる手法（UML）は、プログラムの構造を考えることから一旦離れ、要求分析レベルのシステム設計を行うことが主目的です。そのため、ここでできた設計書の内容をそのまま最下流のプログラマ実装レベルに持ってこようとした場合、モレ・ヌケが多数含まれてしまいます。このままでは結局はモレの大部分をプログラマが勝手に設計してしまい、アーキテクトの意思の伝わっていないものとなってしまいます。この理由から、顧客要求を分析するアーキテクトと実装するプログラマの間のギャップ、つまり、「モレ・ヌケ」をとるために中流の ZIPC を経由するソリューションとなっています。

以降では、それぞれのソリューション項目について簡単に内容を説明いたします。

Softune V3/V5 の主な特長

Softune は、16 ビット MCU の F^2MC-16 ファミリと 8 ビット MCU の F^2MC-8L ファミリ用に V3、32 ビット RISC チップの FR 用に V5 をご用意しています。Softune そのものも進化しており、1999 年に発売した Softune V5 には C++コンパイラと C++チェッカ、C++アナライザを搭載しています。

他社の開発システムと比べて特徴的と考える機能のみを箇条書きしておきます。

（１）デバッガ、エディタ、コンパイラを密結合させた統合開発環境
（２）富士通(株)の技術をフルにつぎ込んで開発した高性能な C/C++コンパイラ
（３）コンパイラ技術を応用した C/C++チェッカ、C/C++アナライザ
（４）RTOS アナライザ、RTOS コンフィグレータなどによる ITRON（および OSEK）の開発支援
（５）ソース世代管理ツール(Visual Source Safe、ClearCase、PVCS など)との連携
（６）外部エディタ(秀丸、WZ、MIFES、ViVi、その他)とのエラージャンプ連携
（７）日英同梱のセットアップ CD-ROM での日英版同時リリース(メニュー、ヘルプ、マニュアルについて日英対応)
（８）ZIPC などとの Softune 連携ソリューションの充実

UML との連携：
ROSE-ZIPC-Softune 連携

システムの分析／設計レベルまでは、UML 手法を使います。

顧客要求を ROSE のユースケース図やシーケンス図、クラス図、状態遷移図、などのドキュメントで表現し、それを使ってプロジェクト内またはサミット同士でレビューを繰り返し行うことで意思疎通ができ、整理され部品化された合意仕様ができあがります。(図 5 参照)

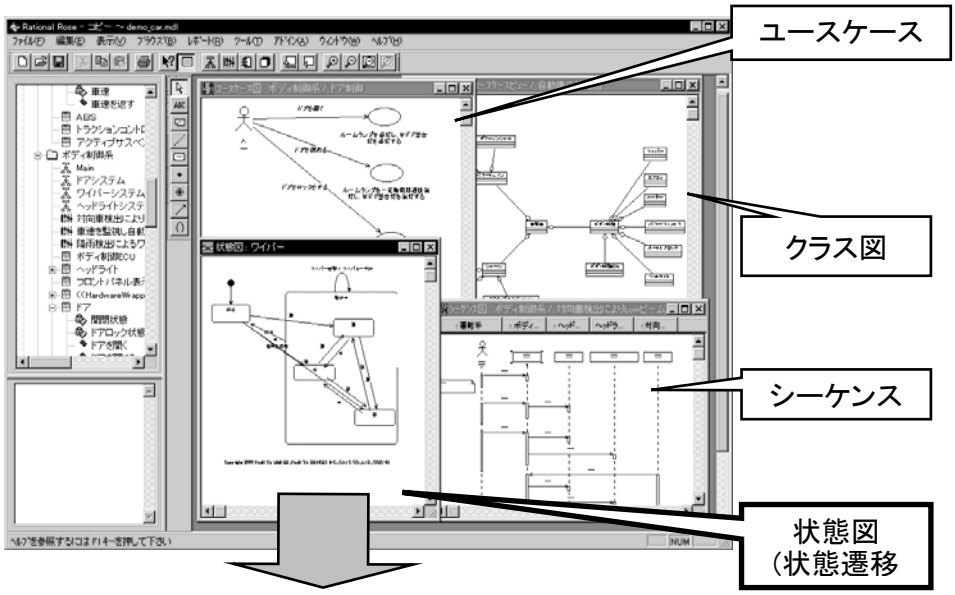


図 5　オブジェクト指向による上流設計: ROSE との連携

つぎに実装のための設計ステージでは ZIPC を使用します。ROSE で作成したドキュメントのなかの状態遷移「図」を ZIPC の状態遷移「表」へ ZIPC－ROSE コンバータを使用して変換します。状態遷移表により、状態図ではモレ・ヌケしていた状態／イベントの組み合わせについてもモレなく設計が行えます。このステージでは状態遷移表レベルでのシミュレーションデバッグも行います。

こうして出来上がった状態遷移表からいよいよ Softune でのコード実装デバッグのステージに入ります。富士通(株)の Softune では、他社の ZIPC 連携にはない機能として、ZIPC ジェネレータから生成されたＣソースを自動的に Softune のプロジェクトとして登録する機能が組み込まれています。これにより ZIPC から多数のファイルとして生成される C ソースを簡単に間違いなくプログラマステージに持って来られるため、スムーズな連携開発が行えます。Softune を使ってコードを追加開発し、シミュレータデバッガまたは、ICE と実機でデバッグを行います。デバッグの際には、ZIPC の状態遷移表と連動させたブレークポイントの設定やステップ実行が可能です。

合同セミナ、サンプル、手順書の提供

前節で、顧客要求を分析／設計し、コードに実装するまでの手順を説明しましたが、実際には ROSE, ZIPC, Softune というツールを揃えただけでは開発は進みません。各ステージで使用する手法のサポートと、各ステージ間の移行に関するサポートが必要となります。これに対し、富士通(株)では，キャッツ(株)、(株)オージス総研の各社と協力した次のソリューションをご提供します。

（１） ROSE-ZIPC-Softune のサンプルコードと連携手順書

ROSE-ZIPC-Softune のサンプルコードを(株)オージス総研、キャッツ、富士通デバイス(株)、富士通(株)で協力して作成したものをご用意しています。また、このサン

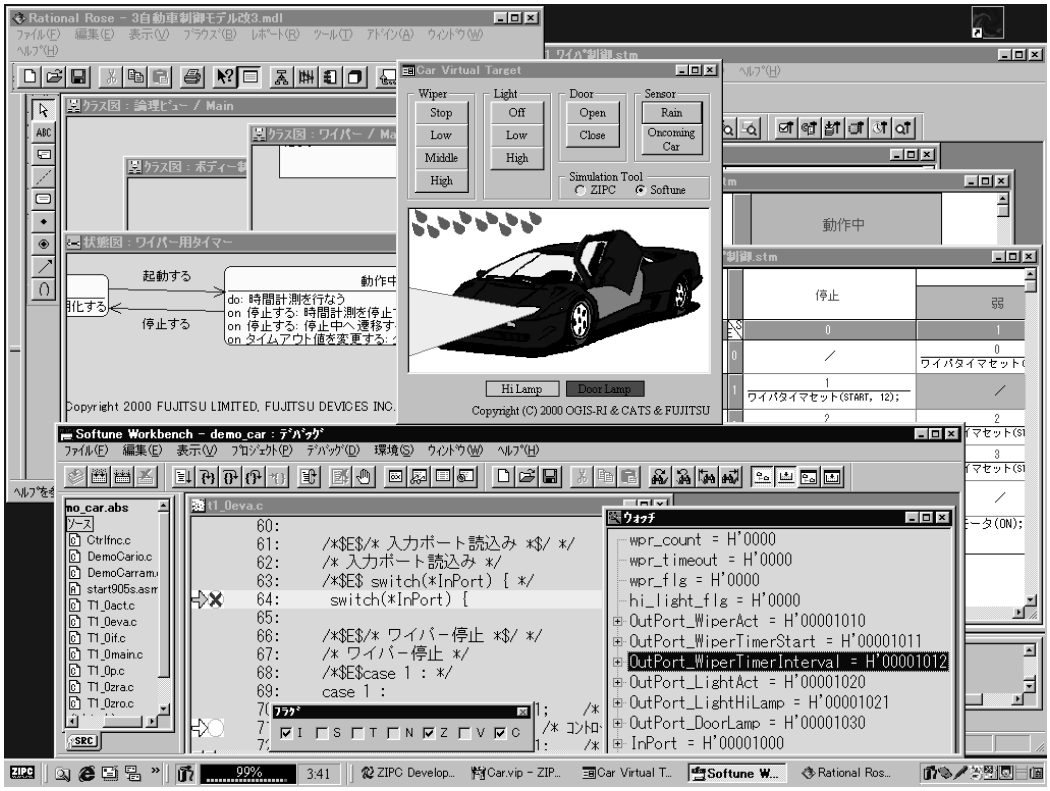


図 6　合同セミナでも使用する ROSE-ZIPC-SOFTUNE 連携サンプル

プルを使って Softune と上流のツールとの連携の手順書を富士通(株)でご用意しています。これらはいずれ Softune に添付して配付する予定です。このサンプルと手順書により、富士通(株)のお客さまの ROSE や ZIPC 導入がスムーズになります。

（2）合同または各社セミナやコンサルティング

- □ UML セミナ：(株)オージス総研
- □ ZIPC セミナ：キャッツ(株)
- □ Softune/C/ICE/MCU セミナ：富士通(株)（および富士通デバイス(株)）

Softune 連携のサンプルを使用しての三社合同セミナやコンサルティングにも対応予定。(図 6 参照)

3D-CAD との連携：FJVPS/HIL

富士通研究所で開発した FJVPS（Fujitsu Virtual Product Simulator）は、3 次元 CAD データと連携してメカ・シミュレーションするソフトウェアです。(図 7 参照)

市販の 3D-CAD ツールで作成したメカ機構モデル（例えば CD チェンジャ）の CAD データを取り込み、立体的に、

- ― 操作性検証
- ― 機構シミュレーション
- ― 動的干渉チェック
- ― 組立・分解のアニメーション

などを行うことができます。

さらに HIL（Hardware In the Loop）オプションを追加することで、実際のマイコンおよび制御ボードと FJVPS を I/O ボードで接続し、こ

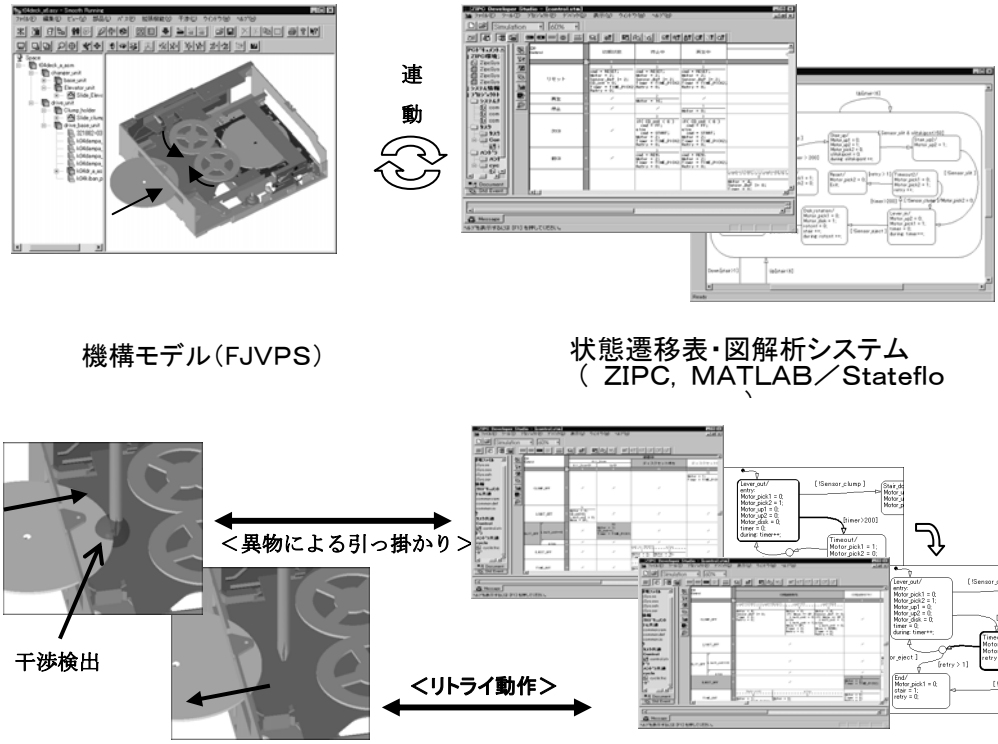


図 7　FJVPS と ZIPC の連携

の 3D-CAD で作成したメカ機構モデルの制御を行うことができるようになります。これにより、メカ機構を実際にハードウェアとして用意することなく仮想メカに代用させ、実マイコン上で動くタスク制御プログラムの開発／テストを効率的に行うことができます。

また、複雑なタスク制御プログラムを ZIPC の状態遷移表、または、MATLAB (Simulink/Stateflow) の状態遷移図と FJVPS の 3D モデルを連携させて開発することもできます。この組み合わせにさらに Softune デバッガ、および ICE との連携も現在追加開発中で近いうちにデモ可能となります。

自動車向けオプション・ソリューション

□ StatemateMAGNUM との連携

StatemateMAGNUM は主に自動車業界で使われているツールで、状態遷移図で設計するツールです。StatemateMAGNUM 自体にも C コード生成の機能があるのですが、モレ・ヌケ防止のためには ZIPC による開発ステージを通すのが有効です。現在、富士通(株)のお客さま向けに、Statemate の状態遷移図から ZIPC の状態遷移表に変換するソリューション (コンバータ) を富士通(株)が協力する形でキャッツ(株)にて開発予定です。

□ CAN/OSEK との連携

富士通(株)では CAN (Controller Area Network) のソリューションとして、CAN 内蔵のマイコン (F²MC-16LX、FR) をご提供しています。その開発環境として Softune に加え、ベクタージャパン(株)の CANoe, CANape などのツールも使っていただいています。また、CAN 関係の RTOS として OSEK がありますがこれもベクタージャパン(株)の osCAN をご提供しています。今後は、これらのソリューションと Softune および ZIPC の連携をより密にして、お客さまがより効率よく開発が進められるよ

う、キャッツ(株)、ベクタージャパン(株)とも協力してソリューションを開発することを検討していきたいと考えています。

まとめ

このように富士通(株)では、お客さまの開発効率アップと品質作り込みに役立つソリューションをZIPCとの連携を「要」として取りそろえております。本ソリューションは、単にツールだけのご提供ではなく、セミナ・コンサルティングを含めたものです。このソリューションの遂行にはキャッツ(株)をはじめとする各社との協力が重要です。富士通(株)は、関連会社である富士通デバイス(株)、および、(株)FFC を含めた体制で本ソリューションを強力にバックアップし、キャッツ(株)はじめ関連各社と協力して、顧客満足第一のソリューションをお客さまにご提供していきたいと考えております。今後ともSoftuneおよびZIPCをよろしくお願いします。

(いがらし じゅん)

システム仕様記述言語 SpecC と組み込みシステム設計ツール VisualSpec™

(株)東芝 研究開発センター システム技術ラボラトリー
荒木 大

1. はじめに

LSI の微細化が進んだ結果、ハードウェア（HW）とソフトウェア（SW）からなる大規模なシステムが一つの LSI に載るようになりました。また、通信分野、携帯端末、マルチメディア分野といった組み込みシステム製品は、仕様の複雑さが急激に増大するとともに製品のライフサイクルの短縮化も相まって、今まで以上に開発期間短縮とコスト削減が求められています。しかしながら、このような仕様の膨張と性能の向上に対して、組み込みシステム製品の設計技術が追いついてゆけないプロダクティビティ・ギャップの心配が大きくクローズアップされてきました。SW の開発者にとっても、LSI 設計者にとっても、HW の集積度の向上、あるいは、近年の組み込みシステム製品に求められる仕様の巨大化と複雑化に立ち向かえるほど設計技術の進歩が追いついてゆけないからです。

組み込み SW の設計技術の近年の動向は、ようやくオブジェクト指向設計技術が組み込みシステムの世界でも導入されつつあるが、リアルタイムな性能あるいは HW の制約がクリティカルな組み込みの世界においてオブジェクト指向設計がどこまで寄与できるかはいまだ未知数ではないでしょうか。一方で、システム LSI の設計技術の世界においては、ここ数年の一つの変化として、C 言語あるいは C++言語を使った仕様設計が、システム LSI 設計の最上流において行なわれ始めています。このことは大きな意味を持つと考えます。なぜならば、組み込みシステム設計の最上流においては、言語の上では、SW も HW も区別がなく設計を行えるようになるからです。そこで、21 世紀に向けた組み込みシステム製品の設計力向上の鍵は、① HW/SW コデザインの導入、② 概念設計（仕様設計）から詳細設計への連続性の確立、③設計事例の再利用、の三つにあるのではないでしょうか。

HW/SW コデザインとは、HW の設計と SW の設計を別々に行うのではなく、互いを考慮しながら同時に行うことを意味します。HW/SW コデザインの一般的な手順では、システム全体をまず統一的にモデル化してから、HW として実装する部分と SW として実装する部分に分割してゆきます。さらに、HW と SW で実現される機能との間のインタフェースと同期をとるメカニズムを設計します。最終的には、HW 部分は既存の HW 合成ツールを用いて ASIC、FPGA 等の専用回路で実現され、SW 部分はプロセッサ上で実行可能なコードにコンパイルされることになりますが、コデザインの途中段階では、形式的検証あるいはシミュレーション、あるいはプロトタイピングの技術を用いて性能およびコスト面での検証を行うことで、HW と SW のトレードオフを考慮した設計が、HW と SW の詳細設計前の段階で行えます。

次に重要な点が、「概念設計から詳細設計への連続性の確立」です。概念設計のレベルにおいては、HW や SW といった具体的な実装方法は未検討の状態で、製品のユーザ・インタフェースといった機能的な側面を中心に仕様を作り上げます。次に、ここが重要なのですが、概

念設計で作成した仕様記述を、そのまま加工してゆく形で、具体的なアーキテクチャの検討が行なえるようにすることが必要です。つまり、概念設計と詳細設計の両者が、設計プロセスとして見たときにも、仕様記述のデータとして見たときにも、別々にあるのではなくシームレスに結合しておれば、各設計フェーズで行なうべき作業と成果物が明確になり、上流での設計内容と下流での設計内容の整合性を確実に保証することができます。

また、再利用型あるいは部品組み立て型の設計手法が、組み込みシステム製品の設計においてますます重要度を持ってきています。すでにLSI設計のEDA業界においては、設計資産であるIPの流通が始まっており、システムオンチップの設計にはIPの利用が必要不可欠となっています。組み込みSWの世界においても、TCP/IPのような通信用APIといったレベルのIPから、ブラウザといったアプリケーションレベルに近い領域でのIPが広く使われつつあります。こういった外部からのIPの調達の形のほかに、自部門で蓄積された過去の設計資産を効率よく再利用できる仕組みを作り上げることも重要です。

2. SpecC言語による組み込みシステム設計手法

SpecC言語はUCI（カリフォルニア大学アーバイン校）のD.ガイスキー教授を中心にして開発されたシステム仕様記述言語で、HW/SW混在型システムの仕様記述を行なえるプログラミング言語です。また、このSpecC言語を中心にした組み込みシステムの設計方法論が提案されています。この設計方法論がカバーする範囲は、組み込みシステム製品の初期設計段階である機能設計から、システムアーキテクチャの割り当て、HW/SWへの機能分割、各モジュール間を接続するインタフェース仕様の作成を経て実装レベルの設計仕様に至るまでの上位設計プロセスを対象としています。この方法論は、HWだけで製品が構成される場合あるいは組み込みSWだけで構成される場合にも適用可能な枠組みとして考えることができます。

SpecC言語による組み込みシステムの設計手法を図1に示します。システム設計の第一段階は機能仕様を記述することです。機能仕様には実装の詳細までは含まれませんが、状態遷移図とアルゴリズム記述（C言語のスタイルの記述）で仕様を表現しますので、シミュレータを使ってそのまま実行することが可能です。このステップの目的は、製品設計の初期段階におけるラピッド・プロトタイプを開発するです。つまり、製品企画あるいは機能仕様を検討している段階で、機器の動作、製品の見た目、使い勝手などをユーザと同じ視点で検証するためのバーチャル・モックアップを開発します。このようなステップを設けることは、仕様の誤認や解釈の違いによる下流での設計変更・修正の削減が図れるなどの効果が期待できます。

SpecC言語を利用した設計手法の特徴は、このラピッド・プロトタイピングで作成した機能仕様記述から、アーキテクチャの選定、機能分割、スケジューリング、コミュニケーション合成に至る各設計作業を、それぞれの前段階で作成した設計モデルに対してなんらかの置き換えあるいは洗練を行う形で定義されており、それぞれを一貫的にかつ連続的に実行できる点です。したがって、アブストラクトな機能仕様の設計から、アーキテクチャレベルの詳細仕様設計までをシームレスにつなぐことができます。

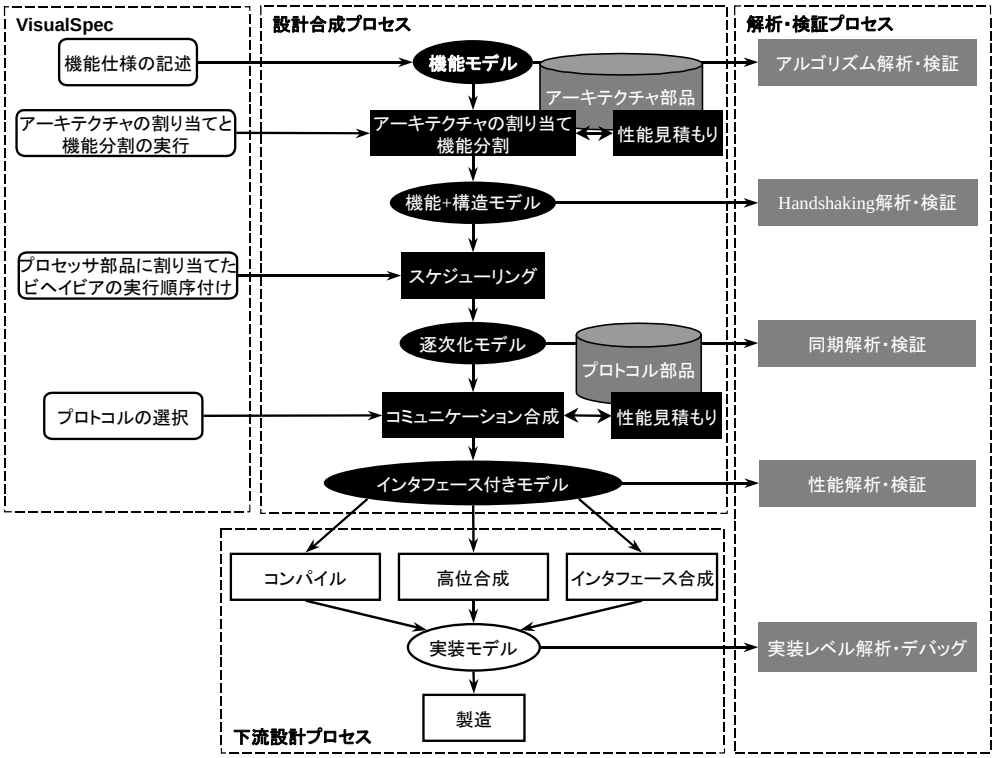


図 1　SpecC 言語を利用した組み込みシステム設計の流れ

具体的には、設計者は、図 1 の設計ステップにしたがって、仕様記述の各機能をターゲットシステムのアーキテクチャにマッピングしていくことで、設計をすすめることができます。設計合成プロセスには「アーキテクチャ割り当て(allocation)」「機能分割(partitioning)」「スケジューリング(scheduling)」「コミュニケーション合成」の 4 つのステップがあります。「アーキテクチャの割り当て」とは、システムを構成するコンポーネントのタイプとその数を決定することです。ここでいうコンポーネントとは、システムのビヘイビアを実装するために使用する、たとえばプロセッサ、ASIC、バスといった部品です、あるいは、全体を RTOS 上で動く一つの組み込み SW として構成するのであれば、RTOS のタスクがコンポーネントに対応します。「機能分割」とは、機能仕様記述で定義した各ビヘイビアを、システムコンポーネントに割り振ることを意味します。「スケジューリング」とは、個々のコンポーネントごとに割り振られたサブ・ビヘイビアの並べ替えを行って実行順序を決定することを意味します。「コミュニケーション合成」とは、ビヘイビア間の通信を実装するために、プロトコルとリソースの選択を行うことを意味します。

これらの設計合成ステップを経て最終的に出力される設計仕様(インタフェース付きモデル) は、図 1 の下部に示す下流設計ツールにそのまま渡すことができます。つまり、インタフェース付きモデルの設計仕様に含まれる SW 部分は C/C++言語のソースコードの形になり、HW 部分は VHDL のビヘイビアコードの形になります。下流設計ツールとは、SW の設計については既存のコンパイラを意味し、HW の設計については高位合成ツール、インタフェース合成ツール等が相当します。

各設計ステップでは、シミュレーションによって動作検証を行うことができます。たとえば、機能仕様の段階では、記述されている機能が設計意図に合致するかをシミュレーションで検証できます。機能分割とスケジューリングが終わった後では、異なるプロセッサ部品に振り分けられたビヘイビア間が仕様どおりに同期するかをシミュレーションで検証できます。コミュニケーション合成の後では、設計モデルを使ったシミュレーションで、演算部と通信部を含めたシステムの性能まで検証することができます。

3. SpecC 言語の特徴

SpecC 言語の文法的な特徴について簡単に説明します。SpecC 言語は C 言語をベースとしており、HW/SW 混在型システムの仕様記述を行なうために、並列性、割り込み処理、状態遷移、通信といった動作仕様を記述するための各種の構文が C 言語に追加されています。

図 2 に SpecC プログラムの基本構造を示します。SpecC 言語は、Program State Machine(PSM)という記述様式に基づいています。PSM とは、階層型並列有限状態機械と手続き型プログラミング言語を組み合わせたモデルです。システム全体は「ビヘイビア」と呼ぶブロック（図の X、Y、Z）の階層によって表現します。ビヘイビア階層のリーフに位置するビヘイビアでは、C 言語の形式でプログラムを埋め込むことができます。一方、ビヘイビア間の通信はポートとポート間の接続関係によって表現し、通信プロトコルは「チャネル」と呼ぶブロック（図の c1、c2）の中で手続きを定義することによって表現します。これらのビヘイビアやチャネルの記述には、オブジェクト指向プログラミングのスタイルを用います。

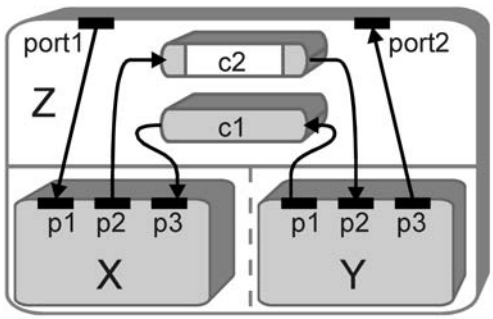


図 2　SpecC プログラムの基本構造

SpecC 言語の特長は、同一言語で、さまざまな抽象レベルのモデルが書ける点です。たとえば、システム仕様記述、システム構造記述、システム構築部品、各部品の設計仕様までを統一的に記述でき、設計の各段階でシミュレーション等により設計検証が行えます。SpecC 言語を利用することによって、仕様から設計へとシステム開発を進める際に、連続的に開発の基幹データを共有することができるとともに、IP あるいは設計データベースの構築による設計再利用も容易に実現できるようになります。また、SpecC 言語で記述された設計仕様は、HW と SW の部分に文法上の区別がありません。実装の詳細をニュートラルな状態のままで、各々の機能を HW あるいは SW として実装した場合のトレードオフを容易に分析・検討することができるのです。

SpecC 言語は、システムレベル仕様記述言語として広く言語仕様が公開されており、SpecC テクノロジ・オープン・コンソーシアム(http://www.specc.org)において標準化と普及活動が行なわれています。

4. 仕様オーサリングツール VisualSpec™

VisualSpec は、SpecC 言語を利用してシステムの仕様記述を作成し、シミュレーションを行う、あるいは、機能仕様をアーキテクチャレ

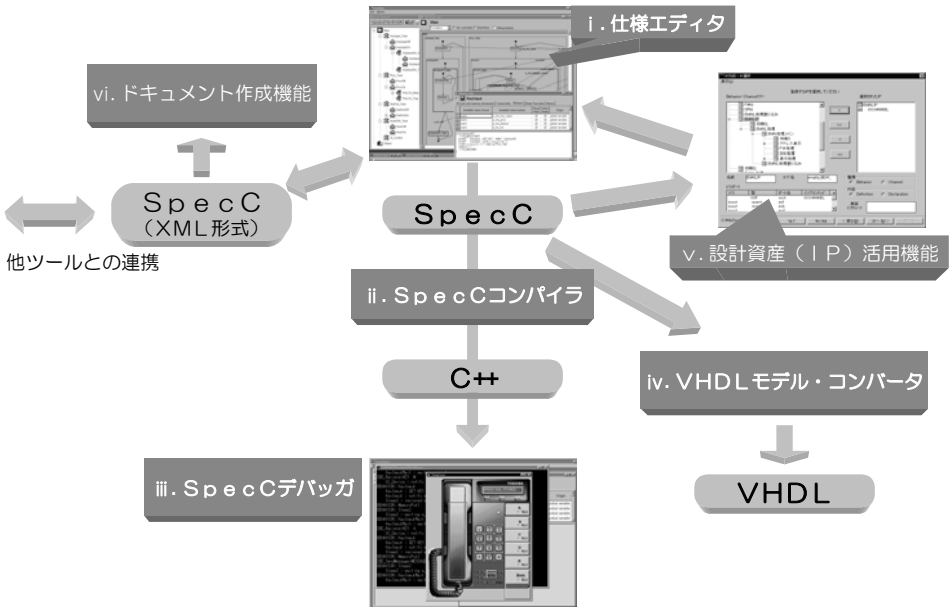


図 3　VisualSpec の機能構成

ベルの設計仕様に合成する、といった SpecC 言語を使用して組み込みシステム設計の作業を行なうためのツールです。

図 3 は VisualSpec の機能構成です。仕様エディタ(図 3-i)は SpecC 言語を使った仕様記述をエントリーするためのエディタであり SpecC 言語のソースコードを出力します。この仕様エディタは、SpecC 言語の持つ各構文に合わせて、プログラム状態間の階層関係や有限状態機械、通信、例外処理の構造と動作をビジュアルに編集することができ、仕様の記述・詳細化を支援します。仕様エディタから、コンパイラを始めとした各ツールが実行できます。仕様エディタの画面例を図 4 に示します。

SpecC コンパイラ(図 3-ii)は SpecC コードを C++コードに変換します。これを、C++コンパイラを利用すれば SW の実行モジュールができあがります。現在対応している C++コンパイラは、Microsoft® VisualC++6.0 と、Red Hat 社の GNUPro です。SpecC デバッガ(図 3-iii)は SpecC コードレベルで仕様記述のデバッガが行なえるデバッガで、仕様エディタと同様の GUI を持っています。

VHDL モデル・コンバータ(図 3-iv)は、SpecC コードの一部あるいは全部を VHDL コードに変換するツールです。IP 活用機能(図 3-vi)は、SpecC コードを切り出して部品化することによって、再利用が容易な形に変換するツールです。ドキュメント作成機能(図 3-v)は Microsoft® Word の形式で、仕様記述のドキュメントを自動作成します。

VisualSpec を使って作成した SpecC プログラムは、Microsoft® VisualBasic などの他のソフトを開発したプログラムと連動して実行させることが簡単に行なえます。たとえば、組み込み製品の GUI のモックアップを

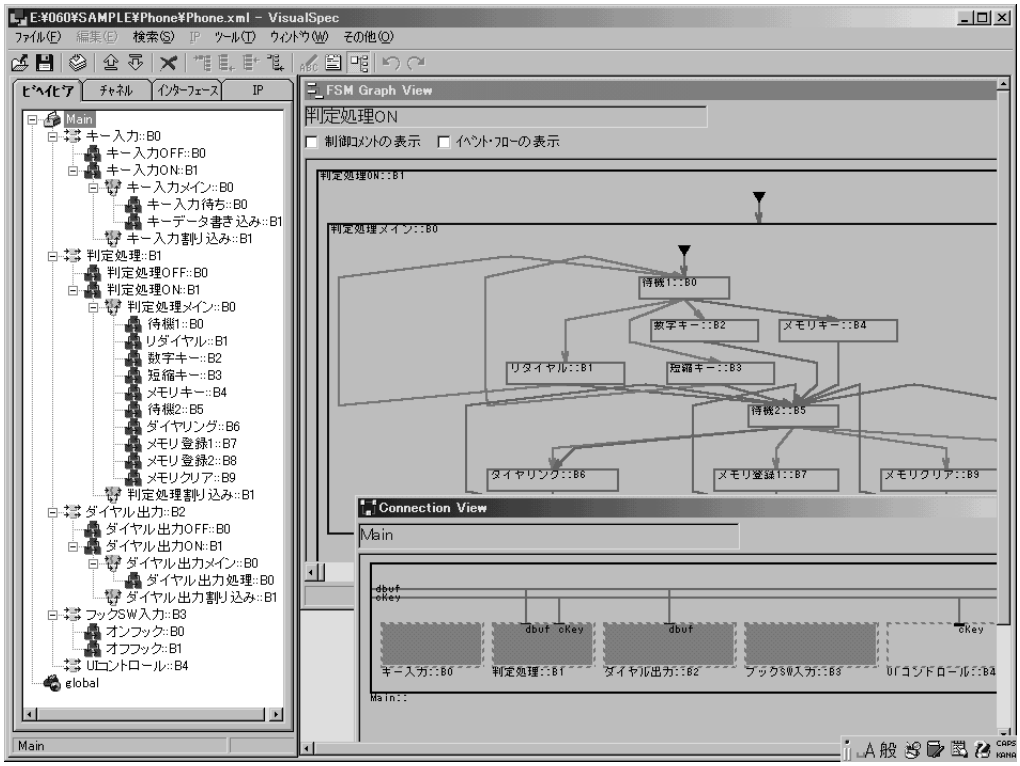


図 4　VisaulSpec の編集画面

Microsoft® VisualBasic で作成し、内部の機能仕様あるいは動作仕様は SpecC 言語を使って記述するといった使い分けを行ないます。このような設計環境を使うと、ターゲット製品のプロトタイプを非常に短期間に開発ができ、かつ内容の修正も非常に容易になります。このモックアップを使って機能あるいは製品の見た目や使い勝手のレビューが行なえるようになります。

5. VisualSpec と ZIPC の統合利用による組み込みソフトウェア開発

VisualSpec で作成した SpecC 言語のビヘイビアモデルをキャッツ(株)の ZIPC の状態遷移表(STM)の形式に変換する機能が開発されています。このデータ変換機能を使えば、VisualSpec と ZIPC を組みあわせて利用することが可能です。この組み合わせにより、組み込みシステム製品の仕様設計から SW としての実装までをシームレスに結合した設計環境が実現されます。

VisualSpec のレベルで設計する内容は、製品の仕様レベルでの設計であり、この段階では、ある程度の抽象度を持たせつつ、必要な機能・動作を明確に洗い出すことに焦点がおかれます。つぎに、VisualSpec で作成した SpecC 言語のビヘイビアモデルを ZIPC の状態遷移表(STM)の形式に変換します。ZIPC で行なうことは実装方法の明確化です。たとえば、SpecC 言語のレベルでは、タスク間の同期は SpecC 言語のイベントの概念を使って表現されています。それぞれのイベントを、RTOS のどの機能を使って実装するかといったレベルでの見直しは ZIPC 上で見直します。このように、組み込み SW あるいは RTOS 上での実装を意識した設計を ZIPC で行なうことができ、ターゲット OS 上で

の検証を行うことができます。

まとめ

SpecC 言語を利用した組み込みシステム製品の設計手法は、システムオンチップの時代に向けた大きな課題である、製品設計のプロダクティビティ・ギャップを解決する一つの道筋であると考えています。仕様の可視化とラピッドプロトタイプの活用は、顧客満足重視の商品開発が求められる中で、製品開発の最初の段階で正しく・早く・的確に製品仕様を把握・決定する手段として重要です。また、SpecC 言語を活用することで、製品の概念設計と詳細設計、あるいはSW設計とHW設計をシームレスに結合することができます。さらに、IP の活用を中心とした設計への転換も容易になると考えています。

SpecC 言語を利用するための設計ツールであるVisualSpec と ZIPC が結合されたように、今後、SpecC 言語を利用するための各種設計ツールの登場と、関連する設計ツールとの間での接続や連携が一層進むことが期待されています。SW 設計と HW 設計の垣根を越えた、組み込みシステム設計者にとって利用しやすい設計環境の整備が加速されることが望まれます。

（あらき だい）

ZIPC の“今”と“これから”

キャッツ(株)

渡辺 政彦

■はじめに

20 世紀最後の ZIPC Watcher です。20 世紀最後の ZIPC と 21 世紀の ZIPC がどうなるかをお話ししましょう。

■組込み開発環境の HUB ※1

われわれのような業界の人だと HUB と言えばネットワークを連想するでしょう。ダラス・フォートワース国際空港(DFW)を思い浮かべても結構です。沖縄の蛇を思い出す人は休暇を取りましょう。ダラス空港を走り回るようになって初めて国際的なビジネスマンだという言葉を昔聞いたことがあります。本当かどうかはともかくそのような経験をすることができるようになりました。ZIPC は Ver.6.0 となり、まもなく ZIPC Ver.7.0(製品名はきっと、ZIPC2000 ですな)がリリースされる予定です。ZIPC は組込み開発環境の HUB として成長しました。(図 1)

どうしてこのような HUB として ZIPC が存在しているのでしょう？それは ZIPC が設計にターゲットを絞っていることと拡張階層化状態遷移表(EHSTM)を支援しているからです。(図 2)

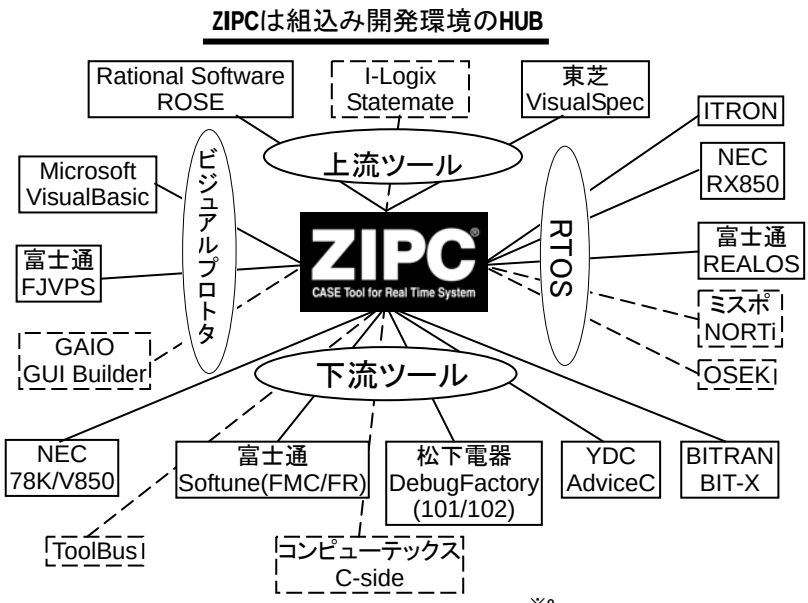


図 1　組込み開発環境 HUB ※2

上流工程ではシステムのドメイン分析と言うと格好が良いですが、ようするに「何をするか」をきちっとさせましょうというのが目的です。この段階で RTOS や IO のことなど考えるレベルではありません。じゃ、それをどこで考えるのかとなると設計フェーズであり、そこを支援する手法として EHSTM があり、ツー

※1 最近一般的に使われるケーブルは 10BASE-T と 100BASE-TX だ。これらのケーブルを使って LAN を構築する場合、接続するパソコンの台数分のケーブルを接続する必要がある。この接続するための機器を「ハブ」と言う。「HUB」と記述する。この「HUB」という言葉は中心と言う意味を持っている。
[http://www.itsui.com/network/contents/edu/hub.html]
※2 破線は 2000 年 4 月現在で開発中または案件のもの

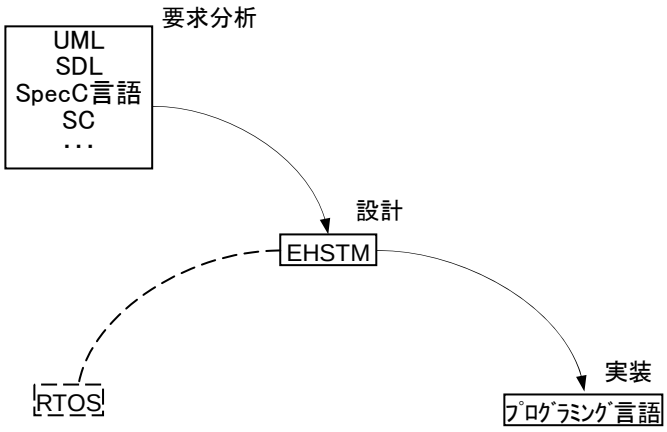


図 2 設計

ルとして ZIPC があるのです。ビジュアルプロトタイピングにしても GUI やメカの動きをシミュレーションするツールとしては確立していますが、マイコンを搭載してソフトウェアで制御するとなると ZIPC と連動させたくなってきます。RTOS の存在をリアルタイムフレームワークというものでラップしてアプリケーションモデルから RTOS を隠蔽してしまう方法を採用しているツールがあります。こうなると RTOS を使用した設計方法とはこのリアルタイムフレームワークをいじりまわすことになります。RTOS のタスクとモデル化したオブジェクトのマッピングに四苦八苦する羽目になりかねません。そこで直接 ITRON のシミュレーションエンジンを搭載する ZIPC 上でステートマシンタスクとして設計することが現実的な解となります。下流工程はエディタ、コンパイラ、デバッガが統合化された IDE 環境となっていますが、あくまでもプログラミング言語の世界に限られています。これに ZIPC を接続することで「いつ」「どこで」「何をする」を漏れなく記述された状態遷移表ベースでデバッグが可能となります。

まだ紙面上で接続する製品名を発表できませんが、21 世紀にはさらに多くの「上流ツール」、「ビジュアルプロトタイパ」、「RTOS」、「下流ツール」が ZIPC とリンクされる予定です。

■より専門化したツールへの進化

1999 年度の社団法人　日本電子工業振興協会(JEIDA)の調査でもお蔭様で２年連続国内１位を獲ることができました。(図 3)

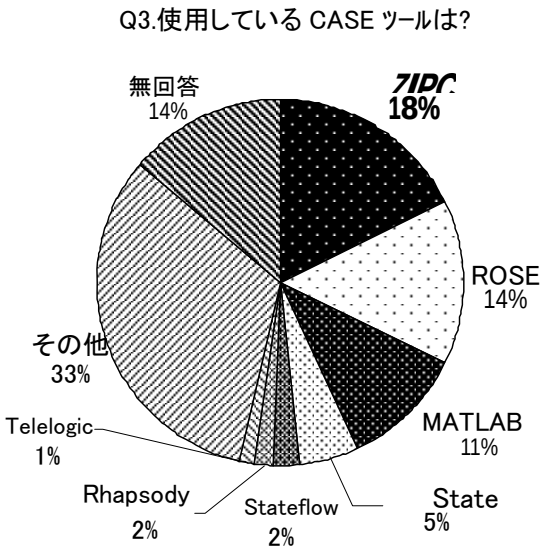


図 3　CASE ツールシェア

ツールのシェアを業界ごとに見てみると、それぞれのツールの使用率が変わってきます(図 4)。表記法についても同様です(図 5)。

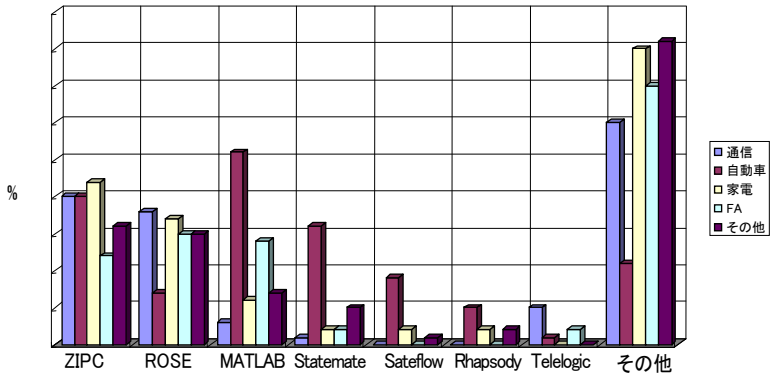


図4 業種別CASE

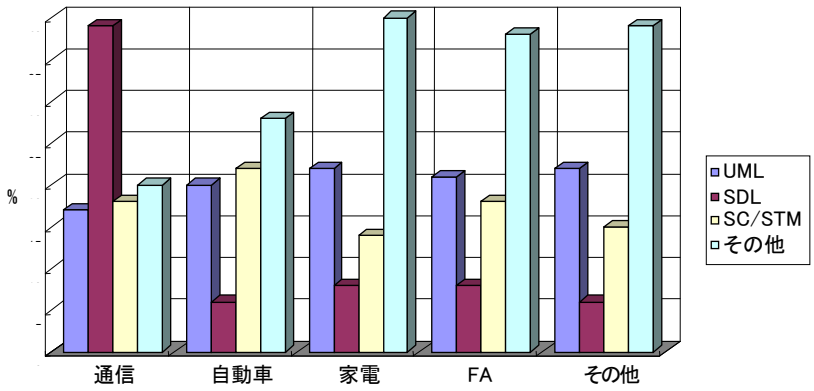


図5 業種別表記法

ZIPC 基盤技術への改良、要望対応することでより大きなHUBを目指します。そして大きなHUBで迷子にならないよう、あらゆる選択肢の中から各分野または制御対象ごとに特化したパッケージ、サービスがこれから必要になると考えております(図6)。

さて、組込み業界も国際的競争時代に突入します。皆さん、ZIPC の HUB を走り回って実りあるビジネスを！

（わたなべ まさひこ）

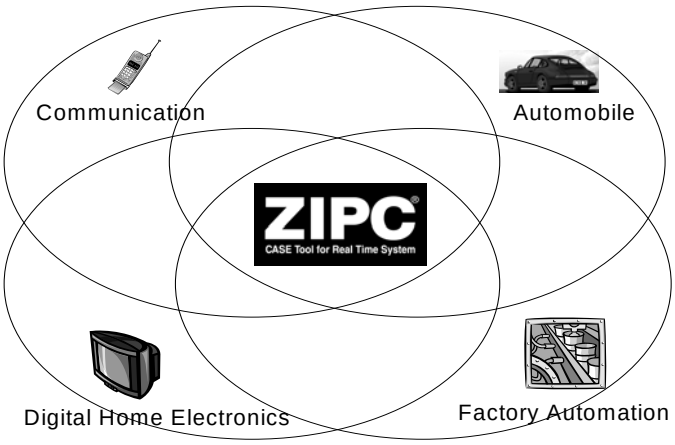


図 6 特化

ZIPC ウォッチャー バックナンバーサービスのお知らせ

キャッツ(株)では、本誌「ZIPC ウォッチャー」のバックナンバーのダウンロードサービスを行います。これは、前号・前々号の内容を Adobe 社 Acrobat による PDF 形式にて、弊社 Web サイトよりダウンロードする事ができるサービスとなっております。

ZIPC ウォッチャー Vol.3（1999 年 5 月 28 日 初版発行）

ZIPC ウォッチャー Vol.2（1998 年 6 月 19 日 初版発行）

ZIPC ウォッチャー 創刊号（1997 年 3 月 25 日 初版発行）

※ 敬称略
※ ご所属の部署は当時の物です。

ZIPC ウォッチャー 第 4 号

初版　　1999 年 6 月 16 日 発行

発行者 ／ 上島 康男
発行 ／ キャッツ(株)
〒222-0033 神奈川県横浜市港北区新横浜 2-7-19 天幸ビル 50 4F
電話 (045)473-2816
http://www.zipc.com/
