

C-011

上位ハードウェア設計言語 Melasy+ による自己回復機能付き FIFO メモリの記述と検証

Code Generation and Verification of A FIFO-type Memory with Self-recovery Functions using A Meta Hardware Description Language "Melasy+"

白鳥 航亮† 和崎 克己†
Kousuke Shirotori Katsumi Wasaki

1 はじめに

IT 技術の発展により、様々な環境で電子機器が動作しており、年々規模・数とともに増加傾向にある。また、様々な社会システムがこれらの技術と信頼性に依存しており、とりわけデジタルシステムにおける技術と信頼性は社会システムの維持に必要不可欠である。仕様通りに確実に動作する信頼性の高いシステムを、より安価に効率よく設計できる環境が望まれている [1]。

ハードウェア設計の正当性を形式的にチェックするツールとしては、SMV[2]、NuSMV[3] 等のモデル検査ツール (Model Checker)[4] が存在する。これらのツールを用いることで設計の正当性の評価を自動で行うことができる。しかし、NuSMV によって実ハードウェアの設計を全て記述するには、極めて低級な言語を利用しなければならない。また、VHDL[5] や Verilog[6] 等の言語で設計したシステムを、検証目的のために、改めて NuSMV 等の言語で再記述する工程が発生する。同じ設計を異なる言語で複数回行うことは、実際の開発現場の状況に鑑みるとコスト・納期などの制約において困難である。

上記問題を解決する手段として、一つのコードから様々な処理系に対応するように設計された上位ハードウェア設計言語 Melasy+[10] がある。Melasy+ は C++ のライブラリとして実装されている。Melasy+ のコードは、Melasy+ 用の特殊な型や構文を用いて C++ のコードとして記述する。本稿では、自己回復機能付き FIFO メモリ [11] を実際に記述し、モデル検査器 NuSMV 向けの駆動可能なコードを得ることに成功した例について述べる。本研究でモデル化した FIFO 型メモリは、一過性の故障に対して有限回数の書き込み動作や読み出し動作の繰り返しにより、正常な状態に回復する能力を有する。CTL (Computation Tree Logic) 式にて仕様を記述し、NuSMV を用いて自己回復機能の検証を行った結果、記述した FIFO 型メモリのモデルが仕様を満足することが証明された。

2 研究背景

2.1 ハードウェア上位設計系

同じ設計を異なる言語で複数回記述することは一貫性を保つ観点から困難である。VHDL, Verilog, NuSMV などの異なる言語の上位に設計用の共通言語を設置し、このコードから各々のコードを生成するハードウェア上

位設計という手法が存在する。ハードウェア上位設計言語には、本報告で用いた Melasy+ の他に、HDcaml[7]、Confluence[8] などが存在するが、すでに開発は終息している。

2.2 組み込み自己テスト回路

外部にテスト回路を新たに設けるのではなく、被テスト回路と同じチップ上に埋め込んでしまう技術である。テストコストの削減や、テストの容易化などの利点がある反面、テスト時の電力増加による誤動作が懸念されるなどの注意点も存在する [9]。

3 上位ハードウェア設計言語 Melasy+

3.1 概要

Melasy+ は NuSMV や VHDL など、様々な処理系向けのコードを生成することを目的とした、上位ハードウェア記述言語である。Melasy+ と他の言語の関係について、図 1 に示す。Melasy+ コンパイラは、中間コード生成器と各対象言語向けコード生成器から構成される。中間コード生成器は C++ のクラスライブラリとして実装されている。Melasy+ コードを中間コード生成器に与えると、中間コードとして XML コードが出力される。得られた XML 中間表現コードを各言語向けコード生成器に与え、各言語のコードを得る。

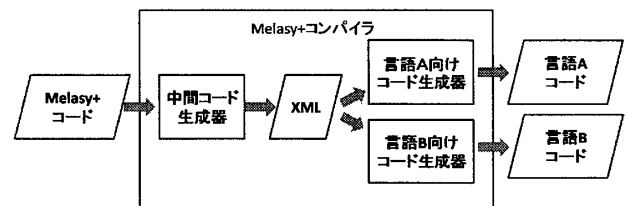


図1 Melasy+ コンパイラ処理系の位置づけ

3.2 言語仕様

Melasy+ コードは C++ のクラス拡張として記述し、文法は C++ の規則に従う。しかしながら、値の保持に int や char といった C++ の既存の型を利用することはできない。Melasy+ では Logic 型と Digit 型を値の保持に用いる。それぞれ、1 ビットと N ビットの値を表すのに用いる。Logic 型では char 型の '0', '1' のいずれかの値を代入することが可能である。Digit 型ではテンプレート引数にビット幅を指定することで、任意のビット幅の型として扱うことが可能であり、値の代入には文字列定数か int 型の値を代入することで行う。また、Melasy+ ではコンポーネント単位で回路を記述する。各コンポーネントは入出力を持ち、出力の動作、入力接続などを定義することで、ハードウェアの仕様を記述す

† 信州大学大学院工学系研究科, Graduate School of Science and Technology, Shinshu University.

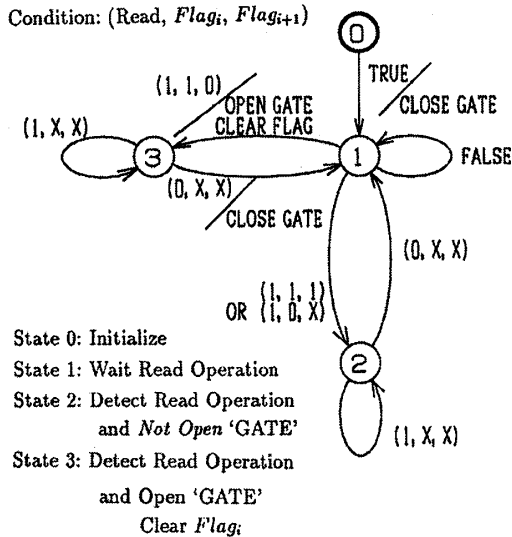


図2 セルの状態遷移

る。コンポーネントの記述に必要な機能は Component クラスが提供する。各コンポーネントの動作はコンストラクタ内で定義を行う。入出力は in, out, sync などの関数を呼び出すことで定義可能である。出力の定義には switch, case, default といった特殊な条件分岐構文を利用することが可能である。for や if のような C++ の言語機能をプリプロセッサの様に使うことが可能であり、さらに、再利用可能なコードを関数として定義し、呼び出すことも可能となっている。

3.3 XML 中間生成系

Melasy+ コードを C++ コンパイラでコンパイルして得られる実行可能ファイルを駆動することでコード生成が行われ、XML による中間表現を得ることが可能である。コード生成は getXML 関数によって行われ、この返り値を標準出力等に出力することで XML 中間コード生成器として動作する。

3.4 各言語向けコード生成器

現在 2つの言語向けのコード生成器が実装されている。対象となる言語は NuSMV と VHDL である。両コード生成器共に python を用いて記述されており、Melasy+ の中間コードとして得られた .xml ファイルを入力とし、NuSMV のコードである .smv ファイル又は、VHDL のコードである .vhd ファイルを出力する。

4 Melasy+ を用いた FIFO 型メモリの記述

4.1 自己回復機能を有する FIFO 型メモリのモデル

Melasy+ の高記述性を確認するためのケーススタディとして、実際に自己回復機能を有する FIFO 型メモリの記述を行う。基本的な動作を行うオートマトンを定義し、これをセルとする。図2にセルの状態遷移を示す。セルをメモリの段数分直列に接続する。

4.2 ハードウェア構成

図3に今回記述した FIFO 型メモリのハードウェア構成図を示す。メモリの一段分はセル、フラグ、データバッファの3つのコンポーネントで構成される。セルはデータの読み出し要求が行われた際に、自身のフラグと一つ

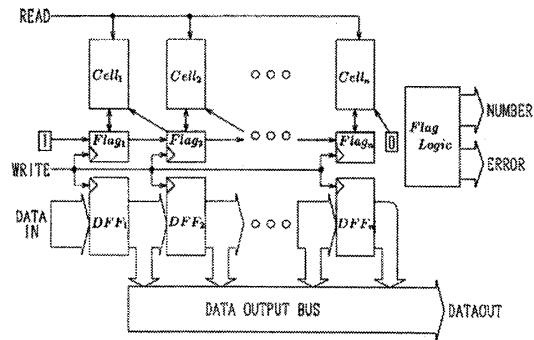
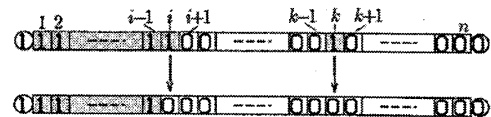
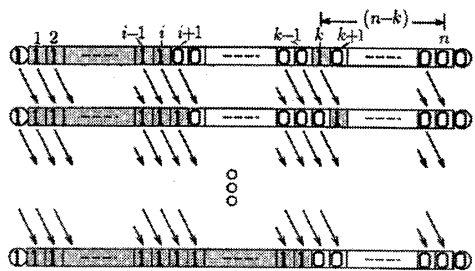


図3 FIFO 型メモリのハードウェア構成



(a) 一回の読み出し動作による回復



(b) 複数回の書き込み動作による回復

図4 一過性故障に対する自己回復例

先のフラグの状態から自身がデータ待ち行列の先頭であるかを判断し、読み出し動作を行う。フラグは有効データの存在を示すコンポーネントで、データ格納部であるデータバッファに有効データが存在するかを示す。

4.3 一過性故障に対する自己回復性

対象とする FIFO 型メモリは、有効データの存在を示すフラグに一時的な障害が発生した際に、有限回数のメモリ操作を行うことにより正常な状態に回復する機能を持つ。図4(a)、図4(b)に読み出し動作、書き込み動作それぞれにおける自己回復過程を示す。読み出し動作については、各セルは、自身以外にデータ待ち行列の先頭であると判断しているセルの有無に関わらず、読み出し操作を行う。よって、複数回読み出し動作を行えば先頭であると判断するセルは唯一つとなり、正常な状態に回復する。書き込み動作は、単純にデータ待ち行列が先頭方向にシフトする動作である。よってメモリが N 段で故障箇所を k 番目とすれば、 $n - k$ 回書き込み動作を行えば、故障フラグはデータ待ち行列から押し出され、正常な状態に回復する。

4.4 Melasy+ による記述

FIFO 型メモリの Melasy+ の記述 (一部抜粋) を図5に示す。class flag はフラグに該当するコンポーネント

```

class flag : public Component{
public:
    Logic prev, write, clear, initial;
    Logic own;

    flag(){
        in(prev); //一つ前のflagの値
        in(write); //書き込み信号
        in(clear); //クリア信号

        out(own); //自身のフラグ
    }
};

template<int N>
class Cyg_main : public Component{
public:
    CPU cpu;
    flag flag[N];
    Cell cell[N];

    Cyg_main(){
        //PortMap for flag
        for(int i=0; i < N; i++){
            PortMap pm[] = {
                flag[i].prev <= flag[i-1].own,
                flag[i].write <= cpu.write,
                flag[i].clear <= cell[i].clear,
            };
            instance(flag[i], pm);
        }
    }
};

```

図 5 FIFO 型メモリの Melasy+ による記述例 (抜粋)

```

MODULE Data_buff(data, gate, write)
VAR
    d : word[3];
    out_put : word[3];
ASSIGN
    init(d) := 0b3_000;
    next(d) := case
        write=0 : d;
        write=1 : data;
    esac;
    init(out_put) := 0b3_000;
    next(out_put) := case
        gate=0 : 0b3_000;
        gate=1 : d;
    esac;
MODULE main
VAR
    Data_buff_0 : Data_buff(CPU_0.data, ... );
    Data_buff_1 : Data_buff(Data_buff_0.d, ... );
    Data_buff_2 : Data_buff(Data_buff_1.d, ... );

```

図 6 生成された FIFO 型メモリの NuSMV コード (抜粋)

であり, class Cyg_main で N 個直列に接続を行う。

4.5 記述性に対する考察

図 5 に示した FIFO メモリの Melasy+ コードから生成された NuSMV コードの一部を図 6 に示す。以下では両コードを比較し, Melasy+ の NuSMV に対する記述性の優位性について考察を行う。

まず, 図 5 の for ブロックに注目すると, この for ブロック中において, フラグのコンポーネントを直列接続している。Melasy+ のコンポーネント単位の記述方法では N 段で抽象化できる。実際のインスタンス生成時に, 自由に N の値を変えることが可能である。一方, NuSMV など繰り返し構造を抽象化する記述方法を持たない言語を用いて直接記述する場合, 100 段ならば 100 段分の記述を直接行わなければならない, 多大な労力が必要である。

```

CTL AG((flag_0.own = 1 & flag_1.own = 1 &
        flag_2.own = 1 & flag_3.own = 1)
        -> AG(Collector_1.o = 0) )

```

図 7 CTL 式: 書き込み動作による自己回復

```

CTL AG( (CPU_0.read=1 & (Cell_0.ack=1 |
        Cell_1.ack=1 | Cell_2.ack=1 |
        Cell_3.ack=1)) -> AG(Collector_1.o = 0) )

```

図 8 CTL 式: 読み出し動作による自己回復

5 自己回復性仕様のモデル検査器 NuSMV による検証

Melasy+ によって記述した FIFO 型メモリの自己回復性について, モデル検査機 NuSMV を用いて検証する。NuSMV では CTL(Computation Tree Logic), LTL(Liner Temporal Logic) を仕様の記述に用いることが可能である。本研究では CTL を用いて仕様の記述を行う。

5.1 故障の実現と検出方法

本研究で扱う FIFO 型メモリは, フラグの一時的な故障について自己回復する能力を有する。そこで, Melasy+ コードの段階で故障フラグの埋め込みを行う。得られた故障付き NuSMV コードを用いて自己回復機能の検証を行う。本報告では故障は初期段階で唯一つのフラグの値が 1 に一時的に縮退することを前提とする。

フラグ列が正常な状態であるとき, 値 1 が連続した後, 値 0 が連続して続く。値 1 が連続する列の中に値 0 が, 値 0 が連続する列の中に値 1 が存在することはない。よって, あるフラグの値が 1 である場合には, 必ずそのフラグの一つ前のフラグの値も 1 である。

故障検出のためにメモリの各段に, 自身と同じ段のフラグと一つ前のフラグの状態を監視するチェッカーを設ける (次項図 9 を参照)。もし, 自身と同じ段のフラグの値が 1 であるとき, 一つ前のフラグの値が 0 であるならば故障状態とする。チェッカーは自身と同じ段のフラグが故障状態にあるならば error の値を 1 にセットする。各チェッカーの error の値の論理和をとる回路モジュール chCollector を用意し, この回路モジュールの出力を監視することでフラグ列全体の故障の有無を確かめる。

5.2 自己回復性の仕様記述

書き込み動作による回復は, 故障したフラグがデータ持ち行列から押し出されることに相当する。すなわち, いずれのフラグが故障した状態が初期状態だとしても, すべてのフラグが 1 である状態に到達すれば, 故障フラグは押し出される。そこで, すべてのフラグが 1 である状態に到達した後は, 新たに故障は発生しないという仕様を, CTL 式を用いて図 7 のように記述する。

次に読み出し動作における回復である。4.3 で述べたように各セルは自身が先頭であると判断すれば読み出し動作を行う。本報告における故障の想定は一カ所に限定したため, 一度読み出し動作を行えば, 故障を起こしているフラグはクリアされる。よって, 一度読み出し動作を行った後には故障を起こしているフラグが存在しないという仕様を, CTL 式を用いて図 8 のように記述する。

```

class Checker : public Component
{
public :
    Logic prev, own;
    Logic error;

    Checker(){
        in(prev);
        in(own);
        out(error);

        sync(error,
            switch_(own)[
                case_('0', '0'),
                case_('1',
                    switch_(prev)[
                        case_('0', '1'),
                        case_('1', '0')
                    ]
                )
            ]
        );
    }
};

```

図9 故障検出回路モジュール (抜粋)

5.3 段数 N である FIFO 型メモリに対する検査

段数 N である FIFO 型メモリは、一段目、 $2 \cdots N-1$ 段目、 N 段目の3つの構造に分けて考えることが可能である。一段目と N 段目が待ち行列の内側にあるメモリと接続が異なることを説明する:

- (1) 一段目のフラグには一つ前のフラグが存在しない。代わりに、常に値1のフラグを参照する。また、データバッファの入力部には、外部からのデータ入力線が接続される。
- (2) N 段目のセルは次段にメモリが存在しない。代わりに、常に0である特殊なフラグを参照する。

メモリの段数を変更する際、変更を加えるのは $2, \dots, N-1$ 段目についてである。段数に変更を加えたとしても、内側の様な部分の構造が変化するわけではない。すなわち、いくつかのケースを抜き出して検査を行い、仕様を満たすことが証明されるならば、 N がどのような値であっても同様の結果である。

5.4 計算機実験による検証結果と考察

今回の計算機実験によって、段数 $N = 3, 4, 5, 6$ について、初期状態でただ一ヶ所のみの故障を前提とし、すべてのフラグの故障パターンについてモデル検査を実行した。メモリの動作中に、新たな故障は発生しないものとする。5.2 で示した仕様の検査を行った結果、すべての故障パターンで仕様を満たされることが判明した。よって、今回の段数 N の範囲内で、自己回復性が満たされていることが証明できた。段数を増やした場合においても構造は変化しないため、任意の段数 N について、自己回復機能が正しく動作することが示された。

6 まとめと今後の課題

Melasy+ を用いて、自己回復機能付き FIFO メモリを記述し、得られた NuSMV コードを用いて自己回復機能についての検査を行った。記述したモデルは自己回復機能を有していることが証明された。Melasy+ を用いた記述では、for や if のような C++ の言語機能を用いる

ことができるため、NuSMV で直接記述を行うよりも記述が容易であった。

従来の各言語ごとに記述する方法では、NuSMV を用いて記述、検証した結果が仕様を満足するものであっても、実装用の言語で書き直す工程で、なんらかのエラーが発生する可能性がある。しかし、Melasy+ を用いることで、異なる言語のコードでも Melasy+ コードからの生成物であるため、検証を行い正しいと証明されたコードをそのまま実装コードとすることができる。さらに、設計の検証と実装を一つのコードで行うことができるため、Melasy+ を用いることで記述コストの低減が可能である。

Melasy+ では、CTL 等の時相論理式で記述される仕様の埋め込みをサポートしていない。仕様の記述に際しては for や if などの C++ の言語機能を利用せず、特に繰り返し構造に対する抽象化ができないため、記述性は低いままである。よって今後の課題の一つとして、仕様の記述をより簡単に行えるような工夫を行うことをあげる。本報告では各コードジェネレーターの細かい部分の整備も並行して行ったことから、NuSMV での検証に留まった。よって、VHDL などの実装用コードの出力を行い、実際に駆動させる必要がある。また、対応言語の増設や、さらなる適用事例の増加につとめたい。

参考文献

- [1] E.M.Clarke, O.Grumberg, D.Peled : "Model Checking" ; MIT Press, 2000.
- [2] The SMV System : Carnegie Mellon University, <http://www.cs.cmu.edu/modelcheck/smv.html>.
- [3] NuSMV: a new symbolic model checker, <http://nusmv.irst.itc.it/>.
- [4] B.Berard, M.Bidoit, A.Finkel, F.Laroussinie, A.Petit, L.Petrucci, Ph.Schnoebelen, P.McKenzie : "Systems and Software Verification Model-Checking Techniques and tools" ; Springer, 2001.
- [5] VHDL : VHSIC Hardware Description Language, <http://vhdl.org/>.
- [6] Verilog : IEEE Standard Verilog Hardware Description Language, IEEE 1364-2001 Revision C, IEEE Verilog Standardization Group, <http://www.verilog.com/IEEEVerilog.html>.
- [7] Tom Hawkins : "The Caml Hump: HDCaml" ; INRIA / VASY, <http://www.confluent.org/wiki/doku.php/>.
- [8] Tom Hawkins : "Confluence System Design Language" ; INRIA/VASY, <http://linux.softpedia.com/developer/Tom-Hawkins-24593.html>.
- [9] 米田友洋, 梶原誠司, 土屋達弘 : "ディペンダブルシステム—高信頼システム実現のための耐故障・検証・テスト技術" ; 共立出版, 2005.
- [10] N.Iwasaki, K.Wasaki : "A Meta Hardware Description Language Melasy for Model Checking Systems" ; Proc. of the 5th Int'l Conf. on Information Technology : New Generations (ITNG2008), pp.273-278, 2008.
- [11] 和崎克己, 不破 泰, 江口正義, 中村八東 : "セルオートマトンの概念を用いた自己回復能力をもつ通信用バッファ" ; 電子情報通信学会論文誌, Vol.J77-D-I, No.1, pp.41-52, 1994.