一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
ASN2018-104（2019-03）

センサIoTデバイスエミュレーションの抽象化に関する研究

広瀬　太志[†]　湯村　翼[††,†]　篠田　陽一[†,††]

† 北陸先端科学技術大学院大学　923–1292 石川県能美市旭台 1–1
†† 国立研究開発法人情報通信研究機構　923–1211 石川県能美市旭台 2–12
E-mail: †{s1710171,shinoda}@jaist.ac.jp, ††yumu@nict.go.jp

あらまし　センサを搭載したIoT (Internet of Things) デバイスを使用することで，市街地や圃場など広範囲な環境データの取得が容易になる．データ取得に用いられるIoTデバイスは大量に設置する必要があるため，事前検証が難しく，また現地でのデバイス再設置のコストが高い．その対策としてエミュレーションを用いた検証が行われるが，センサIoTデバイスは同時かつ大量に動作するため計算負荷が高い．一方で，テスト要件に対して忠実な実装が必要とは限らない．そこで本研究では，計算要求リソースを削減するため，テストで用いるMCU (Micro Control Unit) エミュレータの機能の一部を抽象化することを提案し，抽象化によりテストのカバレッジがどの様に変化するかを考察した．

キーワード　IoT，組込み，エミュレーション，シミュレーション，ソフトウェアテスト，ソフトウェア検証

Abstraction of Emulation for Sensor IoT Devices

Futoshi HIROSE[†], Tsubasa YUMURA[††,†], and Yoichi SHINODA[†,††]

† Japan Advanced Institute of Science and Technology　1–1 Asahidai, Nomi, Ishikawa, 923–1292 Japan
†† National Institute of Information and Communications Technology　2-12 Asahidai, Nomi, Ishikawa, 923–1211 Japan
E-mail: †{s1710171,shinoda}@jaist.ac.jp, ††yumu@nict.go.jp

Abstract　IoT (Internet of Things) makes it easy to measure and collect a wide range of environmental information. IoT devices equipped with various sensors are assumed to be used in urban areas, and farms. Since it is necessary to install a large amount of IoT devices used for data acquisition, preliminary verification is difficult, and the cost of reinstalling the device in the field is high. As a countermeasure against this, verification using emulation is performed, but since the sensor IoT devices operate simultaneously and in large quantities, the calculation load is high. Meanwhile, an implementation that is faithful to the test requirements is not necessarily required. In this research, therefore, we propose to abstract a part of the functions of the MCU (Micro Control Unit) emulator used in the test in order to reduce the computation request resource and how abstraction changes the coverage of the test was considered.

Key words　IoT, embedded, emulation, simulation, software testing, software verification

1. はじめに

1.1 背景

IoT (Internet of Things) の登場により，広範囲の環境情報計測と収集が容易になるとされている．計測センサを搭載するIoTデバイスは，市街地や圃場などで使用されることが想定される．しかし，デバイスが広範囲に分散配置されると，設置後の再配置などを伴う改修は難しい．そのため，設置前の検証が重要となるが，必要数のデバイスを用意して現地で検証作業を行うことは非効率的である．そこで，本研究ではコンピュータエミュレーションによるIoTデバイスシステムの検証について検討する．

エミュレーションによる検証において，検証対象のアプリケーションが動作するプロセッサと同じアーキテクチャのMCU (Micro Control Unit) エミュレータを用意することで，同じアプリケーションプログラムを用いることができる．また，ソフトウェアのエミュレータや環境を模倣するシミュレータなどのサブシステムを組み合わせることで，IoTシステムが置かれる環境を再現可能となる．

しかし，プロセッサの命令セットを完全に模倣するエミュ

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2019 by IEICE

レータはオーバーヘッドが大きく，計算要求リソースが高い．また，センサ IoT デバイスのエミュレーション実行は，一度に大量のデバイスエミュレータを実行しなくてはならず，多くの計算リソースを必要とする問題がある．

1.2 目　　的

本研究では，プロセッサとセンサ間の通信に使用する SPI (Serial Peripheral Interface) 通信の抽象化を行う．抽象化によって，一般にエミュレーションの計算要求リソースの削減が可能となる．しかしその一方で，実機と比較して忠実度の低下を招くため，抽象化したエミュレーションではテスト要件を満たせなくなる可能性がある．このように，エミュレータ抽象化には一種のトレードオフの関係がある．また，適切な抽象度のエミュレータを実装しなければ，必要以上に計算リソースを使うエミュレータを開発することになる．そこで本研究では，エミュレータを抽象化する際の忠実度とテスト要件のトレードオフ関係を明らかにし，エミュレーション抽象化の有効性の評価を行った．

2. 関 連 研 究

Lv らはソフトウェア開発プロセスにおいて，実機のハードウェアを用いないアプリケーションの方法として，ARM の命令セットシミュレータ (以下 ISS) である ARMISS の実装と評価をしている [1]．ISS の実行方式はインタプリタ方式とバイナリ変換方式があるが，ARMISS ではインタプリタ方式を採用している．これは動作させるアプリケーションを ARM アーキテクチャに逐次変換して実行する方式である．一方で，インタプリタ方式はバイナリ変換方式と比較して動作が遅い．この解決策としてキャッシュ機構を実装し，高速化を実現している．そこで本研究では，ARM の ISS を搭載したバイナリ変換方式のエミュレータを用いる．この方式は，変換オーバーヘッドが比較的小さく，使用メモリ量もインタプリタ方式と比較して少ない．

Platt は ARM 等のプロセッサをエミュレーション可能な QEMU [2], [3] 環境下で，仮想 GPIO (General-purpose input/output : 汎用入出力) エミュレータを実装しており，Raspberry Pi と周辺機器のエミュレーション環境を構築している [4]．QEMU では GPIO に相当する機能が無いため，UART や SPI などの通信インタフェースを用いた周辺機器を含むテストが難しい問題があるため，GPIO 相当の機能を模倣するレジスタを定義し，QEMU 内に内包した．またレジスタへのアクセスは，QEMU が動作するホストコンピュータ上の共有メモリにアクセスすることで実現している．そこで本研究では ARM エミュレータに SPI ハードウェアを模倣するレジスタを実装することで，SPI 通信を用いた IoT デバイスのアプリケーション検証を可能にする．

3. エミュレーション抽象度モデルの提案

本研究では，エミュレータの抽象化による忠実度とテスト要件のトレードオフ関係を明らかにするために，4 層ののレイヤを持つ抽象度モデルを定義した．また，エミュレータ抽象化の有効性評価のために，2 種類のエミュレータモデルを設計した．

抽象化レベル ＼ 実行レベル	アプリケーションレベルの実行	ライブラリレベルの実行	システムコールレベルの実行	ハードウェアレベルの実行
アプリケーションレベル				実行レイヤ
ライブラリレベル				
システムコールレベル				
ハードウェアレベル	抽象化レイヤ			

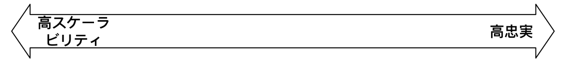

図 1: 抽象度レベル図

表 1: 抽象度レベル図における抽象化間の性質

	アプリケーションレベルの実行	ライブラリレベルの実行	システムコールレベルの実行	ハードウェアレベルの実行
抽象化する層	ライブラリ以下	システムコール以下	レジスタ以下	-
I/F	ライブラリの関数	ライブラリが使うシステムコール	レジスタを操作する関数	プロセッサの命令セット
検証可能項目	アルゴリズムや各種機能テスト	H/W を用いないプログラムテスト	H/W 無し，OS 機能ありのテスト	実機並みの詳細なテスト
利点	実装により軽量	中程度軽量	ドライバプログラム等をテスト可	実機に近いテスト環境
欠点	実装によりテスト範囲が狭い	H/W を機能のテスト不可	H/W を機能のテスト不可	忠実ゆえに動作が重い

3.1 抽象度モデル

本研究では，エミュレーションの抽象化のために，抽象度モデルを定義する．Seo らの組込みソフトウェアテストのためのツールである Justitia [5]，Jerraya らの組込みシステムのハードウェアとソフトウェア間のインタフェースデザインに関する研究 [6] や，次世代ユビキタスエミュレーション技術開発プロジェクトの報告書 [7] において提案された複数のエミュレーション形態の概念をもとに，これらの層を定義した．エミュレーションの抽象度は，アプリケーション，ライブラリ，システムコールとハードウェアで分かれる．これをふまえて，抽象度レベルを図 1 に示す．この図では各階層が抽象化の粒度を示している．例として，アプリケーションレベルの抽象化を実行すると，アプリケーションレイヤのみが実行され，ライブラリ以下が抽象化される．

表 1 は図 1 をもとに，抽象化の種別間において，抽象化する層，インタフェース（I/F），検証可能項目，それぞれの利点・欠点をまとめた．例として，ライブラリレベルの抽象化を実現するためには，システムコール以下を抽象化する．このときの抽象化されたインタフェースは，ライブラリが扱うシステムコールである．ライブラリレベルの抽象化を実現するためには，ライブラリが使用するシステムコールを変更する必要がある．これにより，検証対象のアプリケーションに変更を加えずに検証することができる．

3.2 エミュレータモデル

エミュレータの抽象化を行ううえで，IoT デバイスの構成として一般的な，プロセッサとセンサの組み合わせを想定する．これらのコンポーネント間は，SPI 通信インタフェースを介して制御し，データの送受信をする．エミュレータを用いてソフトウェアの検証を行う際，SPI インタフェースを模している必

表 2: SPI レジスタのモデル

レジスタ名	説明
status	r/w read status
rx_data	received data memory
tx_data	transmitted data memory

要があるが，これらのコンポーネント間はコマンドやデータが送受信されていれば良い．したがって，本研究では，SPI 通信を行うインタフェースを抽象化し，IoT デバイスのアプリケーションを実行できる環境を構築する．

図 1 と表 1 をもとに，2 種類の抽象度の異なるエミュレータを実装する．1 つ目はハードウェアレベルエミュレータである．これは SPI 通信をするハードウェアを模倣するエミュレータである．SPI 通信を忠実に再現するため，SPI レジスタを模したハードウェアをモデル化して実装する．

2 つ目はライブラリレベルエミュレータである．実機の SPI ハードウェアを操作するには，ハードウェアレベルエミュレータで定義した様な SPI レジスタを用いることが一般的である．しかし，ライブラリレベルの抽象化では，SPI 通信するデータはハードウェアインタフェースとシステムコールを経由しない．そのため，ハードウェアレベルエミュレータと比較してエミュレータの計算要求リソースが低くなるが，この方式では SPI 通信を正確に模倣できない．

4. 設計と実装

4.1 拡張対象のエミュレータ

本研究で使用する MCU エミュレータは，ARMv6-M 命令セットアーキテクチャ [8] を実装しており，一般に組込み用途などに用いられている Atmel SAMD21 ATSAMD21G18A (以下 Cortex-M0+) [9] を模倣するエミュレータである．Cortex-M0+ エミュレータは，SPI 通信を模倣する機構を備えておらず，メモリ I/O により通信が行われる．したがって，本論文中では，このエミュレータを図 2 の<1>擬似 SPI I/O (Pseudo SPI I/O : pspio) モデルと名付けて呼ぶ．

4.2 SPI レジスタモデル

ハードウェアレベルエミュレータの実装モデルを SPI をレジスタ (SPI Register : spireg) モデルと名付けて呼ぶ．SPI 通信を正確に模倣するため，SPI レジスタを表 2 のように定義する．表 2 より，SPI レジスタの状態を保持する status レジスタ，受信データを格納する rx_data レジスタ，送信データを格納する tx_data レジスタの 3 つのレジスタをエミュレータ内部に実装する．実装したエミュレータにより，プログラム中のデータがどの様に送受信されるかについて，SPI の通信のイメージを図 2 の<2>spireg_model に示す．

4.3 Exodus モデル

ライブラリレベルエミュレータの実装モデルを Exodus (Exodus : exodus) モデルと名付けて呼ぶ．システムコール以下を抽象化し，ライブラリレベルでエミュレーションするためには，アプリケーション中の送受信されるデータが，SPI 送受信関数を呼び出されたときに，エミュレータ内の処理から，エミュレータを動作させせているホスト OS 上の処理へと切り替わる必要がある．これにより，オーバーヘッドの大きいエミュレータの処理からホスト OS 上の処理に切り替わることで抽象化され，要求計算リソースの削減が可能になる．これを実現するために，ライブラリの書き換えとエミュレータの拡張を行い，独自の"exd"命令を実装した．exd 命令は，前述した SPI 通信によって送受信したいデータをエミュレータの処理からホスト OS の read/write 処理に切り替える疑似命令である．exd 命令の仕様を表 3 に示す．本実験では，アプリケーションが利用する SPI ライブラリが呼び出す通信関数を exd 命令を用いた関数に書き換える．また，エミュレータに exd 命令をデコード，実行するための機構を実装する．exd 命令の通信処理の流れを図 2 の<3>exodus_model に示す．

表 3: exd 命令の仕様

構文	exd {#imm,} rX
項目	説明
#imm	exd 命令が呼び出すシステムコールの種類を指定
rX	送信データが格納されるレジスタの指定 受信データはこのレジスタに格納される

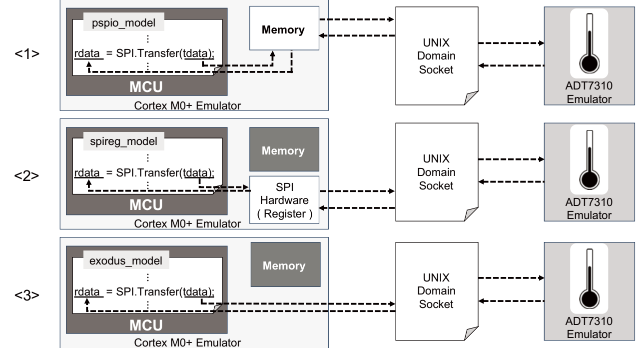

図 2: エミュレータモデルにおけるデータフローイメージ

4.4 センサエミュレータ

本研究では，処理の通信となる Cortex-M0+ エミュレータと，プロセッサの対向側にあるセンサを用意する．センサには，Analog Devices 社製温度センサの ADT7310 を用いる．SPI 通信を通じたコマンドによって，温度の測定，変換，送信をする [10]．ソースコードは Github を参照されたい [11].

5. 評 価 実 験

実装した 2 種類のエミュレータを評価するために，3 つの実験をした．1 つ目にエミュレータ実行時間の計測，2 つ目に SPI 通信処理時間の計測，最後に使用メモリ量の測定である．また，2 つ目の実験において実機と比較実験するため，Cortex-M0+ チップを使用している Arduino M0 Pro を利用する [12]．実験に用いた環境は，NICT の StarBED [13], [14] のノードを利用した．詳細を表 4 に示す．

5.1 凡　　例

グラフラベルの凡例を示す．特に断りがない限り，以下の凡例にしたがう．pspio は疑似 SPI I/O モデル，spireg は SPI レ

表 4: ノード Group P のハードウェア諸元とソフトウェア環境

ハードウェア構成項目	詳細
MODEL	DELL PowerEdge R430
CPU	Intel Xeon E5-2683 v4
MEMORY	DDR4-2400/Multi-bit ECC 384GB
ソフトウェア環境項目	詳細
OS	Ubuntu16.04 LTS 64bits
Compiler	gcc version 5.4.0 20160609

表 5: time コマンド出力の例

パラメータ	説明
real	プログラムの呼び出しから終了までの実時間
user	プログラム自体の処理 (カーネルの処理を除く)
sys	プログラム処理のためのカーネルの処理時間

ジスタモデル，exodus は Exodus モデルを示している．actual は実機の Arduino ボードによる実験結果を示している．

5.2 エミュレータ実行時間の計測

エミュレータの起動，エミュレーションの実行，終了までの一連の時間を測る．エミュレータ上で動作するプログラムの命令数を一定の量に固定することで，実行時間を測定する．これにより，計算要求リソース削減を実行時間を指標とする．

5.2.1 実験方法

エミュレータの起動から終了までの時間を Linux の time コマンドを使用して測定する．time コマンドから取得できる情報を表 5 に示す．ここではエミュレータが実行する命令数を 1000 万個の命令に統一することで，同じ処理にかかる時間を比較する．またこの実験は 100 回繰り返し，その平均値を算出している．

5.2.2 結果

実行結果を図 3 に示す．real 指標に注目すると，spireg と exsodus では実行時間に 1.3 秒の差があり，実行時間ベースで 9.95% の差が出ている．これは Exodus モデルのエミュレーション要求リソースが SPI レジスタモデルの要求リソースよりも少なかったことを示している．pspio と exodus の結果はほぼ同等 (0.27 秒，2.3%) の差であった．これらについて，他の指標から原因を考察する．

pspio と exodus の数値を比較すると，user 指標では exodus が 0.25 秒遅く，sys 指標では exodus が 0.02 秒遅い．user 指標の結果は，exodus のエミュレータの処理が大きいことを示している．sys 指標の結果は，SPI の通信処理をする Linux の read/write のシステムコールの処理がほとんど同等の速度であったことを示している．ここで，pspio と exodus のエミュレータで動作するアプリケーションプログラムは同じ処理をするプログラムであり，通常の SPI 通信をする関数を exd 命令を使用する関数に置き換えたものである．つまり，プログラム中で SPI 通信をする回数は同じであるため，エミュレータの最もプリミティブな通信処理である read/write の実行時間は同等である．そして，exodus ではエミュレータに独自設計の exd 命令処理を追加したため，エミュレータの処理が増加しことが考えられる．したがって，exodus は pipsio より read/write の

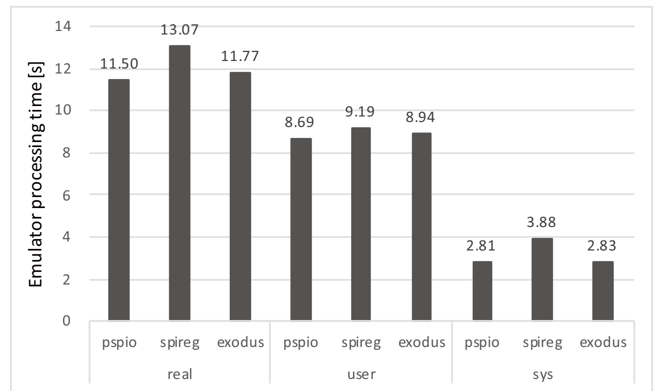

図 3: エミュレータの起動から終了までの実行時間

処理を除く exd 命令の処理のオーバーヘッドが大きいと考えられる．

以上から，この実験ではより抽象度を低くした spireg より，高い抽象度で実装した exodus の計算時間が短くなる結果となり，期待通りの結果を得られた．

5.3 SPI 通信処理時間の計測

実装した SPI 通信における I/O 処理にかかる時間を計測する．比較のために，実機 Arduino の処理時間を測定し用いる．

5.3.1 実験方法

エミュレータにおける SPI 通信処理時間の計測は，SPI 通信を行うために呼び出される Cortex-M0+ エミュレータ内の関数の処理時間の計測によって行う．Cortex-M0+ エミュレータは，アプリケーションプログラムから SPI 通信を行う際に特定のメモリアドレスへのアクセスを伴う ARM の ldr 命令と str 命令を使用する．これらの命令処理に対し，エミュレータを実行するコンピュータにおいて，C 言語の clock_gettime 関数を使用し時間計測を行う．

またこの実験では 1000 万回の命令を実行した際に含まれる SPI 通信の内，1 回あたりの往復時間を平均して算出する．

5.3.2 結果

計測結果を図 4 に示す．縦軸に SPI 通信の処理時間 (マイクロ秒) を示し，横軸に左から Arduino (actual)，エミュレータ各種 (pspio/spireg/exodus) を列挙した．

actual では SPI 通信を 300 回行って平均を算出した．エミュレータでの計測は 1000 万命令中で行われる SPI 通信から平均を算出した．1000 万回の内，SPI に関する read (ldr 命令) は 106260 回，SPI に関する write (str 命令) は 53130 回実行された．これは，SPI 通信の処理では ldr 命令が二回，str 命令が一回行われるためある．actual では 2 マイクロ秒程度の処理時間で read と write の処理時間の差は少なかった．pspio では，read 処理 0.72 マイクロ秒，write 処理が 1.48 マイクロ秒であった．spireg では命令レベル (ldr/str) の処理はグラフ中の Instruction level time で示す処理時間であった．read 処理が 1.95 マイクロ秒，write 処理が 1.58 マイクロ秒であった．SPI 通信は対向側の通信相手により，処理時間が通信相手に依存する．これは SPI レジスタモデルにおいて，エミュレータの内部に搭載される SPI レジスタを実装したモデルの実態に

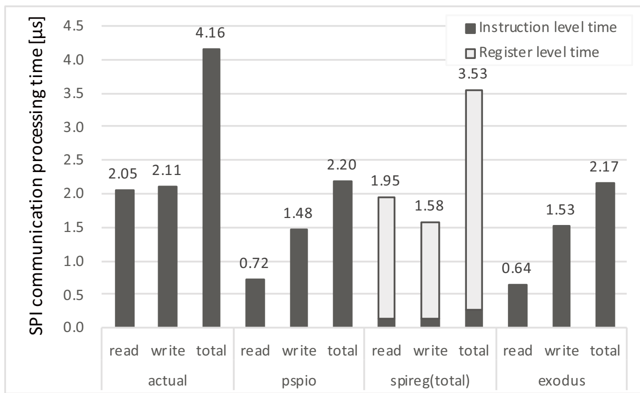

図 4: SPI 通信に関する処理時間

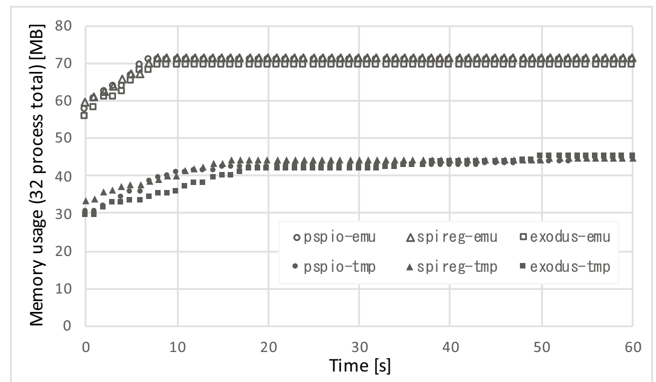

図 5: 各エミュレータのメモリ使用量 (32 並列/60 秒間)

対してデータを読み書きするため，非常に高速に処理されるためである．実際の SPI 通信処理にかかる時間を測定するためには，SPI レジスタのモデルの実装エンティティが通信処理を行う部分を合計することで求められる．その際の処理時間をグラフの Register level time に示した．Instruction level time と Register level time の合計時間は read で 1.95 マイクロ秒，write で 1.58 マイクロ秒であった．exodus では，read 処理 0.64 マイクロ秒，write 処理が 1.53 マイクロ秒であった．

図 4 で read/write で示した値を合計した値が実際の SPI 通信の処理時間である．これを図 4 の各種 total に示す．軸のスケールとラベルは図 4 と同様である．actual の SPI 通信時間は 4.16 マイクロ秒であった．これはエミュレータの SPI 通信実行時間と比較しても大きな値である．pspio のエミュレータの処理時間は 2.20 マイクロ秒，spireg モデルは 3.53 マイクロ秒，exodus は 2.17 マイクロ秒であった．この処理時間の傾向は，実験 5.2 の time コマンドを使用したエミュレータの処理実行時間を計測した際の傾向と同様であるが，pspio と exodus は実験 5.2 と比較して実行時間の傾向が逆転している．これについては計測誤差の範囲であると考えられる．

以上の結果より，SPI 通信の抽象化は通信時間の削減をしている．本実験では時間を指標として議論したが，エミュレーションの計算要求リソースに関しても同じ傾向であると考えられる．

5.4 使用メモリ量の測定

IoT デバイスエミュレーションのスケーラビリティを阻害する可能性の 1 つに，メモリ使用量がある．一般に大量のエミュレータを実行するには相応のメモリ使用が考えられるため，エミュレーションの数に比例した量のメモリを消費すると推測される．したがって，メモリの使用量を調査することで，IoT デバイスエミュレーションを行うためのプラットフォームの選定に影響があるか調査する．

5.4.1 実験方法

メモリの使用量を調査するために，Linux の ps コマンドを使用した．ps コマンドは実行中のプロセスの一覧を取得するコマンドであるが，u オプションを使用することでメモリ使用量を調べることができる．このとき表示されるメモリ指標は RSS (Resident Set Size) であり，プロセスが使用中の物理メモリの量である．本実装におけるエミュレータはプロセスとして実行される．本実験では 32 個のエミュレーションを並列に実行し，60 秒間のメモリ仕様の推移を取得した．

5.4.2 結果

図 5 に実験結果を示す．グラフ中の「-emu」はエミュレータ，「-tmp」は ADT7310 のメモリ使用量である．Cortex-M0+ エミュレータは最初にメモリの使用量が上昇し，その後一定の値に収束した．これは，OS によるメモリのページング処理の結果であると考えられる．

次に対向側の ADT7310 エミュレータのメモリ使用量の時間変化をグラフ化した．メモリ使用量は 60 秒の中で，徐々に上昇するが，ある一定のところで収束する．メモリの最大使用量が収束したときの値をもとに，それぞれの結果を表 6 に示す．

表 6: Cortex-M0+ のメモリ使用量 (32 並列/60 秒間)

エミュレータ	平均使用量 (-emu)	平均使用量 (-tmp)
pspio	2.22MB	1.40MB
spireg	2.23MB	1.40MB
exodus	2.17MB	1.41MB

表 6 より，Cortex-M0+ エミュレータで使用するメモリ使用量は 1 エミュレータあたり 2.2MB 程度，ADT7310 エミュレータで使用するメモリ使用量は 1 エミュレータあたり 1.4MB 程度である．これは 2019 年現在の計算機リソースとしては十分に小さい値であると考えられるため，IoT デバイスエミュレーション実行において，大量のデバイスの同時エミュレーションでも十分に展開可能である．

6. 議論

6.1 抽象度と忠実度のトレードオフ

抽象化により，忠実度の低下が検証項目に影響することを表 1 で示した．これについて，テスト対象のセンサやアクチュエータなどの性質によって，適切な抽象度を考慮するべきである．特にセンサ IoT デバイスの様な組込みシステムでは OS を搭載しないものも多い．システムコールはアプリケーション開発者がハードウェアを一意に操作し易くするための抽象化であるため，システム間の移植性をほとんど考慮しない組込みシステムでは，いわゆる OS のシステムコールは無い．

6.2 抽象度と実装方法

表 1 で示した各エミュレーションモデル間の実装方法は一意に定まらない．アプリケーションレベルのエミュレーション実行では，ライブラリ以下の抽象化を行う．このレベルの抽象化では，アルゴリズムやプロトコルの検証，あるいは IoT デバイスを組込んだシステムの起動や終了などのトリガを検証できる．しかし，これを実現する抽象化エミュレータの実装は，プロセッサエミュレータに大幅な拡張を必要とする．もしくは，エミュレータを用ずに，通常のコンピュータアーキテクチャで動作する入出力インタフェースを実装したプログラムによる検証も可能である．ただし，この様な実装方式では本来エミュレータを用いる利点であった，実機とテストで同じアプリケーションを用いることができる点を満たさなくなる．抽象化を過度に実装すると，その性質はエミュレーションからシミュレーションに変化する．

6.3 抽象度と忠実度のトレードオフ

完全に忠実なエミュレータの開発は，エミュレーション実行の計算要求リソースが大きくなるため，エミュレータの抽象化を適切にしなくてはならない．センサ IoT デバイスでは，市街地や圃場で大量に設置するため，このような実装のエミュレータでは，大量の IoT デバイスの同時エミュレーションが難しくなる．本研究で提案したエミュレータ機能の一部を抽象化する手法により，エミュレーション実行時間が削減できる．これにより，エミュレーションの計算要求リソースが削減され，大量の IoT デバイスの同時エミュレーションが可能となる．

6.4 エミュレータの実時間処理

IoT システムはセンサ IoT デバイスのようなエンドノードと，ゲートウェイなどの中間ノードを経由して，サーバシステムに蓄積・分析される．この様な形態のシステムを想定したとき，データの送受信などのタイミングも正確に再現する必要がある．つまり，IoT デバイスエミュレーションでは処理のリアルタイム性も考慮に入れ無くてはならない．本研究では，実際のプロセッサと同等の実行時間を担保しておらず，エミュレーションのオーバーヘッドが実時間処理を妨げる可能性がある．そのため本論文で示したエミュレータの抽象化により，エミュレーション実行の実時間処理に貢献すると考える．

7. まとめと今後の展望

本研究ではエミュレータ抽象化の提案と，抽象度と忠実度の関係について考察した．今後エミュレータによる IoT デバイスの設置前検証が有効な選択肢となるには，様々な機能拡充が必要となる．IoT デバイスを設置したい事業者にとって，エミュレータ開発負担は小さくない．したがって，プロセッサやセンサ類のエミュレータは事前に用意されるべきである．特にプロセッサに関して，代表的に利用が想定される MCU チップ，アーキテクチャのエミュレータを用意しておくことは有効である．また大量の IoT デバイスを同時にエミュレーションする際，IoT デバイス固有の ID やネットワークデバイスのアドレスなどの管理が必要となる．これを管理するためのフレームワークが岩橋らによって開発されている [15]．IoT システムの多くは無線通信を利用しているため，無線通信エミュレータが重要である．コンピュータ上の有線ネットワーク上で無線通信をエミュレーションするための研究が明石らによって行われている [16]．これらの研究の進展とともに，実機を用いない IoT デバイスエミュレーション環境が発展すると考えられる．

謝辞 本研究は, 国立研究開発法人 情報通信研究機構（NICT）が運用するテストベッド「StarBED」を用いて行われました．

文　献

[1] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, Ge Yu, "ARMISS: An Instruction Set Simulator for the ARM Architecture," 2008 International Conference on Embedded Software and Systems, 12 Aug. 2008.

[2] "QEMU," QEMU, `https://www.qemu.org/`, (accessed: 2018-02-02).

[3] Fabrice Bellard, "QEMU, a Fast and Portable Dynamic Translator," USENIX, 2005.

[4] Evan Robert Platt, "Virtual Peripheral Interfaces in Emulated Embedded Computer Systems," MASTER OF SCIENCE IN ENGINEERING Report, THE UNIVERSITY OF TEXAS AT AUSTIN, Dec. 2016.

[5] Jooyoung Seo, Ahyoung Sung, Byoungju Choi, Sungbong Kang, "Automating Embedded Software Testing on an Emulated Target Board," AST '07 Proceedings of the Second International Workshop on Automation of Software Test, pp.9, 2007.

[6] A.A. Jerraya, W. Wolf, "Hardware/software interface codesign for embedded systems," IEEE, Computer, Volume: 38 Issue: 2 pp.63–69, Feb. 2005.

[7] "平成 18 年度 次世代ユビキタスネットワークシミュレーション技術研究開発プロジェクト 研究開発報告書," NICT, Mar. 2007.

[8] "ARM コンパイラツールチェーンバージョン 4.1 ARM プロセッサをターゲットとしたソフトウェア開発," ARM, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0471bj/DUI0471BJ_developing_for_arm_processors.pdf`, (accessed: 2018-09-28).

[9] "Cortex™-M0 Revision: r0p0 Technical Reference Manual," ARM, `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf`, (accessed: 2019-02-01).

[10] "Data Sheet ADT7310," Analog Devices, `https://www.analog.com/media/en/technical-documentation/data-sheets/ADT7310.pdf`, (accessed: 2018-12-18).

[11] "adt7310_emulator, " Github, `https://github.com/rosev838/adt7310_emulator`, (accessed: 2018-01-31).

[12] "Arduino M0 Pro," Arduino.cc, `https://store.arduino.cc/usa/arduino-m0-pro`, (accessed: 2018-09-28).

[13] "StarBED4 プロジェクト ウェブサイト, " NICT, `http://starbed.nict.go.jp/`, (accessed: 2018-01-31).

[14] Toshiyuki Miyachi, Takeshi Nakagawa, Ken-ichi Chinen, Shinsuke Miwa, Yoichi Shinoda, "StarBED and SpringOS architectures and their performance," International Conference on Testbeds and Research Infrastructures, pp.43–58, 2011.

[15] 岩橋 紘司, 井上 朋哉, 篠田 陽一, "Internet of Things を対象とした大規模実証実験環境構築に関する研究," マルチメディア，分散，協調とモバイル (DICOMO2014) シンポジウム, pp.1258-1263, Jul. 2014.

[16] Kunio Akashi, Tomoya Inoue, Shingo Yasuda, Yuuki Takano, Yoichi Shinoda, "NETorium: high-fidelity scalable wireless network emulator," AINTEC '16 Proceedings of the 12th Asian Internet Engineering Conference, pp.25–32, Bangkok, Thailand, 2016.