サービス実利用におけるスマートフォンの省電力化に関する研究

神山, 剛

https://doi.org/10.15017/1931931

出版情報：九州大学, 2017, 博士（工学）, 課程博士
バージョン：
権利関係：

サービス実利用におけるスマートフォンの省電力化に関する研究

平成 30 年 2 月

神山　剛

目次

図目次

表目次

概要

近年，スマートフォンはモバイル無線通信技術とともにハード・ソフト両面で進化・普及し，モバイルコンピューティングの中核的な役割を担うようになった．また，個人でもアプリケーション（以下，アプリ）を開発する環境が整備され，従来よりも多種多様な用途のアプリが利用可能になったことで，ユーザは生活の様々なシーンにおいて，各自の目的や趣味嗜好にあったサービスを幅広く選択できるようになった．

このようなスマートフォンおよびサービスの進化の一方で，端末の電池持ち時間の改善が強く要望されるようになり，ユーザにとって電池持ち時間は端末機種やサービスを選択する上で重要な判断材料の一つになっている．

しかし，実際の電池持ち時間は，端末のスペックだけでなく，ユーザが使用するアプリ，さらにはその使い方にも依存する．このため，真に効果のある改善を行うためには，実際のユーザの利用実態をも考慮し，問題の分析と改善を検討する必要があるが，端末やアプリなども含めた従来のサービス開発には以下の問題がある．

1) 実測する以外に端末消費電力を定量的に分析する手段がない

2) 開発時に想定できるユーザ利用シナリオは限定的である

そこで本研究では，実際のサービス利用における端末省電力化を実現するため，ハードウェアからユーザレベルまで統合的に分析可能な手法を実現することを目的とし，以下の課題解決を目指す．

1）端末消費電力の推定技術とアプリ開発者にも利用可能な端末消費電力の評価ツールの提供

2) ユーザ実利用シナリオを考慮した評価に利用可能な根拠データの提供

本研究の成果は以下の通りである．

1. スマートフォンを構成する各ハードウェアコンポーネントの特性と消費電力の関係から生成した端末の消費電力モデルを用いることで，アプリ消費

電力の推定する手法を提案した．また，近年の端末の消費電力を妥当な精度で推定できること，推定に必要なログ収集の負荷が低いことを要件とした評価手法を実現するため，マルチコア CPU やモバイル無線インタフェースとその特徴を考慮したモデル拡張を行った．さらに，本手法に基づいてアプリ消費電力分析ツールを実装し，一般的なアプリ利用のシナリオを対象に 10% 前後の誤差で電力推定できること，3.8% 程度の低いオーバヘッドで電力推定に必要なログ収集が実現できることを確認し，実際のソフトウェア開発において実用的なツールであることを示した．

2. 実際のスマートフォンユーザ約 700 名に対するアンケート調査と約 400 名の端末ログ収集調査によるデータを用い，アプリ使用など実際のスマートフォン利用パターンを導出し，パターン毎にその特徴を示すことで，様々なサービス企画・研究開発に有益なスマートフォン利用モデルを提案した．本モデルは，1 日単位のスマートフォン利用を，ユーザ属性やアプリ使用傾向などの特徴を定量的に示すものであり，例えば何らかの実機を用いた効果検証において端末にインストールすべきアプリなど，実験環境のセットアップに活用出来るデータである．クラスタリングによる利用パターン分析の結果，全体的に 6 つの利用パターンの存在が確認された．また，同一ユーザでも日によって異なる利用パターンが存在するという仮説を検証したところ，例外的なパターンを除くと，90% のユーザの利用パターンは高々 2 つ程度であることが確認された．

第 1 章　序論

1.1　背景

近年，モバイルコンピューティングの中核的な役割を担っているスマートフォンは，従来の日本の携帯電話と同様，モバイル無線通信技術とともに，ハードウェア，ソフトウェア両面で進化し，PC よりも身近な情報通信端末として成熟してきた [1)]．また，個人でもアプリケーション（以降，アプリ）を作成・配布できる環境が整備され，従来よりも多種多様な用途のアプリが提供されるようになったことで，ユーザは生活の様々なシーンにおいて，各自の行動や趣味嗜好にあったサービスを幅広く選択できるようになった．

このようなスマートフォンおよびサービスの進化の一方で，端末の電池持ち時間の改善が強く要望されるようになり，電池持ち時間はユーザにとって端末機種，サービスを選択する上での重要な判断材料の一つになっている．

しかし，現在のスマートフォンおよびサービスの利用実態は極めて多種多様であるが故に，「電池がもたない」ことの原因は，ハードウェアレベルからユーザの使い方まで様々な要因を考慮する必要があり，解決すべき課題を明確にすることに大きな労力・時間を要してしまうという問題がある．

図 1.1 は，同一機種のスマートフォンを使用するユーザ 194 名における，単位時間あたりの平均電池消費量毎のユーザ数の分布である．まず，単位時間あたりの電池消費量が 2〜4.5% の範囲のユーザだけで 163 名，大部分の約 84% を占めている．これは一見，主要な同一傾向として見えるが，この電池消費量の範囲を，満充電状態からの電池持ち時間におきかえてみると，電池持ち時間の範囲は 22.5〜50 時間となり，ユーザ体感的には大きな差であるといえる．さらに，電池消費量 1.5% 以下および 5% 以上のユーザも約 16% と一定数存在することも考慮すると，全体傾向としてユーザによって電池持ち時間が大きくばらついていることがわかる．

フィーチャーフォン時代は，音声通話，ブラウザ，メールなど限られた利用シナリオのもとで，電池持ち時間に対する検討がなされてきた．しかし，スマートフォン利用における電池持ち時間は，図 1.1 が示すように，ユーザがどのアプリを使用しているかなど，ユーザの使い方に依存していることがわかる．

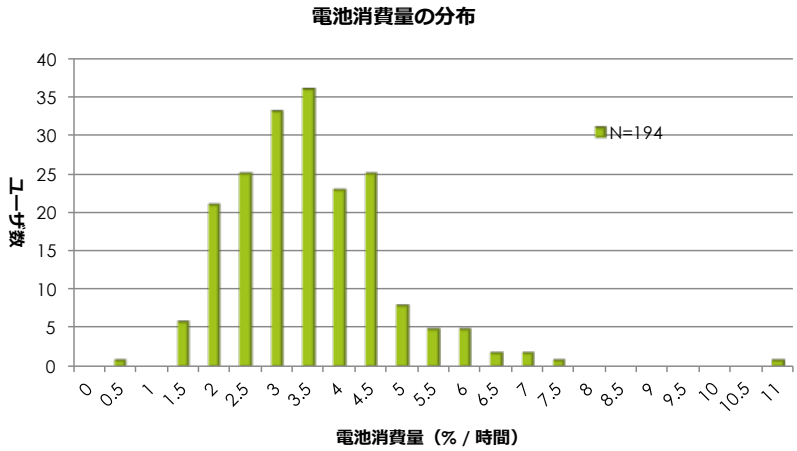

図 1.1　同一機種を使用するユーザの電池消費量毎の分布

ここで，スマートフォン利用におけるエネルギー消費要因を整理する．図 1.2 は，端末におけるエネルギー消費を決定付ける要因とその関係性を示したものである．いうまでもなく，実際に電池のエネルギーを消費するのは端末ハードウェアであるが，そのハードウェアを制御するのは OS やアプリなどのソフトウェアであり，ハードウェアの挙動はソフトウェアに依存する．さらに，ソフトウェアの挙動は，ユーザがどのアプリをどのくらいの時間使用するか，どのような環境で端末を使用するかなど，ユーザの使い方に依存する．

よって，実際のスマートフォンにおけるサービス利用を前提にした省電力化の検討において，局所的ではない実効性のある改善を行うためには，ハードウェアからユーザの使い方まで全てを考慮にいれる必要がある．

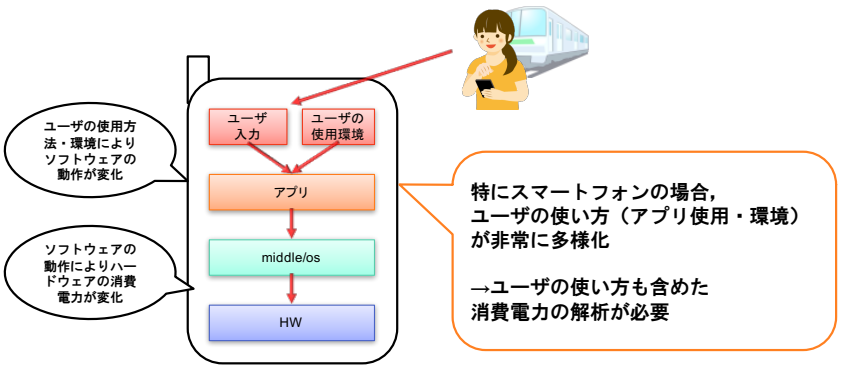

図 1.2　端末のエネルギー消費の因果関係

1.2　サービス開発における課題分析

次に，前述のように実際のサービス利用における省電力化を実現するためには，サービス企画からリリース後の対応まで，実際のサービス開発のプロセスにおいて，この考え方をどのように適用するか検討する必要がある．

1.2.1　サービス開発とは

本論文におけるサービス開発とは，ビジネス企画からシステム開発まで，一連の役割やプロセスを経て顧客にサービスを提供することを指す．例えば，図 1.3 に示すように，ビジネス企画部門はマーケティング分析によりサービスの対象ユーザやニーズなどの分析を行い，サービスイメージとともにサービス利用シナリオを作成する，また，システム開発部門はサービス利用シナリオに基づき，アプリ等の設計・実装・試験を行う，といった役割がある．

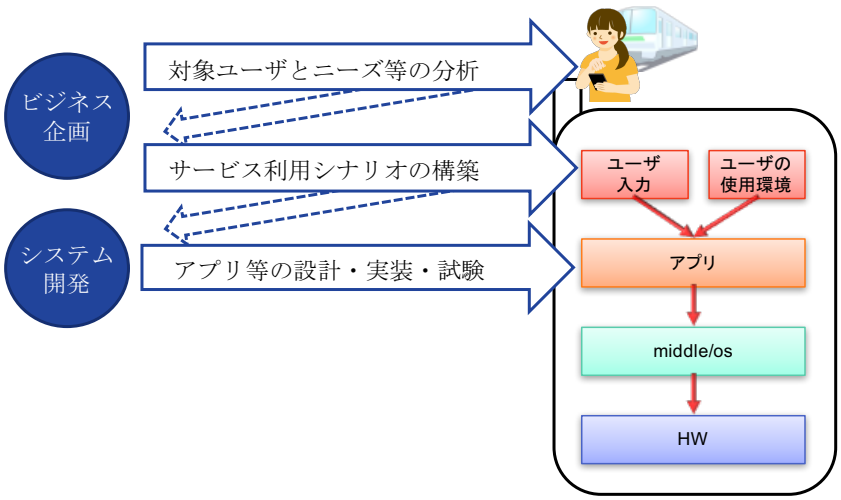

図 1.3　サービス開発における役割

ここで，電池持ち時間の問題に限らず，サービス開発における問題解決のポイントを整理する．図 1.4 は，主にアプリ開発を例に，PCDA サイクルでの開発プロセスと，前述した端末からユーザ要因との対応関係を示したものである．

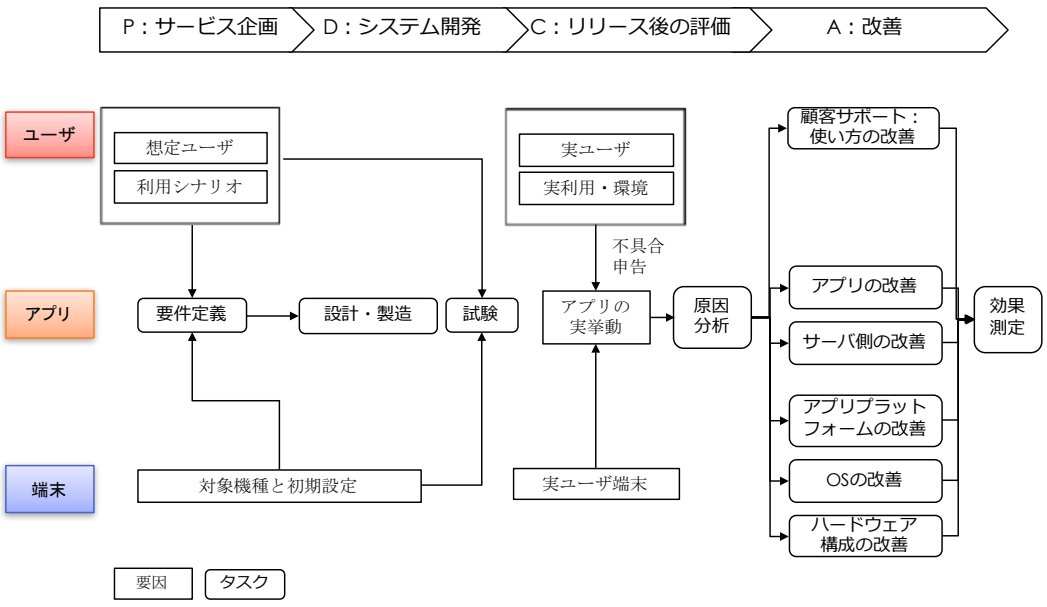

図 1.4　サービス開発プロセスにおける端末〜ユーザ要因との対応関係

まず，一般的なスマートフォンアプリ開発の流れについて，サービスのリリース前と後に分けて説明する．前半のフェーズでは，サービスのリリースがゴールなのに対し，後半のフェーズでは，継続的にサービス運用上の問題を解決しつつ顧客満足度や利益の最大化に寄与することが主要なミッションである．

サービス企画のフェーズにおいて，マーケティングなどの過程でサービスの想定ユーザや利用シナリオなどサービス要件が定義され，続いてアプリやシステムなどの機能要件が定義される．開発フェーズでは，設計・製造を行うとともに，予め作成した試験シナリオのもと試験を経て，開発が完了する．なお，スマートフォンアプリを前提にしたサービス開発の場合，企画もしくはシステム開発のフェーズにおいて，対象ユーザ/利用シナリオとともに，サービスが対象とする端末の機種も予め定めるケースが多い．その主な理由としては，同じアプリケーションプラットフォームが搭載された端末であっても，機種が異なるとアプリ挙動が変わることがあり，対象機種が増えることでその分の試験工数を確保しなければならないなど，開発コストやスケジュールの制約をうけるためである．

アプリおよびシステムが完成し，アプリがマーケットなどを通じて提供されることでサービスがリリースされ，アプリは実際のユーザに使用される．このリリース後の評価フェーズは，システムの運用保守フェーズであり，マーケットやサポート窓口を通じて実ユーザからの評価や不具合申告を受けることになる．なお，この評価フェーズでは，実際には売上げなど営業上の達成度合いからシステムの不具合対応まで様々な確認観点があるが，ここでは後者に限定して議論する．ここで発覚する問題は，サービス企画やシステム開発フェーズでは想定されていない，実際のユーザおよび端末でサービスが利用されて初めて確認される未知のケースが多いと考えられる．このような問題の原因を特定するためには，申告情報を基に問題事象を再現させ，アプリ挙動などシステムの動作を詳細に解析する．次に改善フェーズでは，図 1.4 に示すように，ユーザの使い方をサポートするなどのフロント対

応もあれば，端末ハードウェアからアプリ，さらにサーバ側などのシステム面の改善など，改善対象は事例ごとに多岐にわたる．

1.2.2 サービス開発の問題点

サービス開発における一般的な問題

しかし，エンドユーザからの申告は断片的で不完全なものであることが多く，ユーザの利用実態を把握することが難しいため，問題事象の再現と原因の分析に非常に大きな労力を要してしまう恐れがある．また，問題が未知かつクリティカルなものであるほど，サービス企画やシステム開発など前のフェーズに逆戻りしてしまう恐れがある．このような事態を回避するためには，図 1.5 のように，前半のサービス企画やシステム開発のフェーズの段階で，事前に，ユーザと端末利用など実態を考慮して，できる限り現実的な試験シナリオを作成することが望ましいが，想定と実利用の差を埋めることは容易ではない．

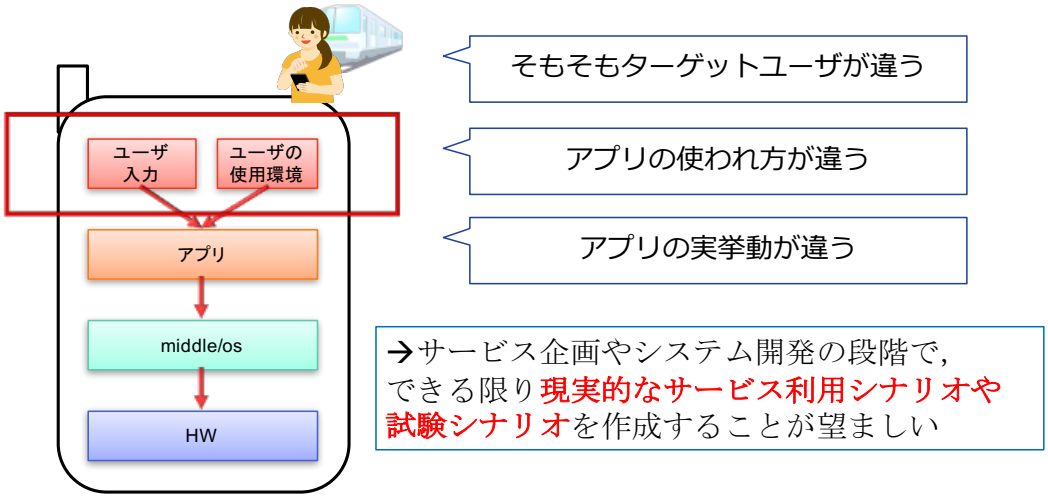

図 1.5 サービス利用シナリオと現実との乖離

電池持ち時間特有の問題 (1)

さらに，電池持ち時間の問題においては，アプリまたはシステムの仕様上の不具合ではなくても，ユーザ体感を損なうレベルの大きな電池消費が起こ

る．このため，一般的なバグ改修と異なり，ソフトウェアの挙動解析だけでは原因の特定が困難であるという問題がある．このような問題を解決するためには，図 1.6 に示すように，前述のハードウェアからユーザレベルの各要因の実態を考慮した形で，端末で生じている電力消費を定量的に把握し，原因の所在を特定すること，さらに改善効果を評価することが必要である．従来より計測機器を用いた測定手段があるが，例えばアプリ開発者に対して，測定環境，測定に要するスキル，試験工数を確保することは現実的ではない．また，端末全体の消費電力しか測定できないため，CPU やディスプレイなど，省電力化のために注目すべきモジュールを特定することが難しい．

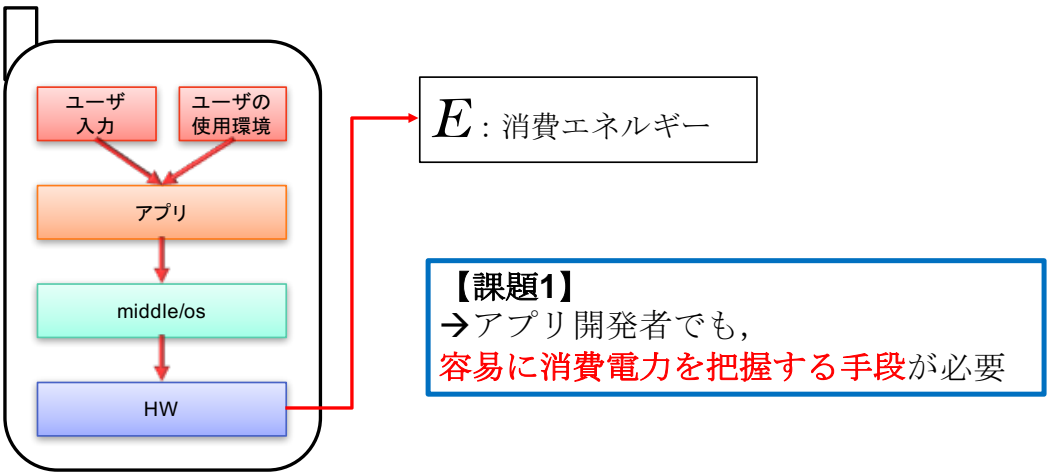

図 1.6　電池持ち時間特有の課題 1

電池持ち時間特有の問題 (2)

また，現実の端末利用においては，開発対象のサービス，つまり特定のアプリだけが電池のエネルギーを消費するのではなく，様々なアプリが利用された結果，電池持ち時間が決定する．サービス提供者の立場では，前述したように，まず自身のアプリが必要最小限のリソース消費で動作するよう設計する必要がある．しかし，ユーザの立場では，自身の 1 日の生活シーンで様々なアプリを利用する必要があり，開発対象のサービスはこのうちの一つに過ぎない．言い換えると，ユーザが望む「1 日の端末利用」にたえられる電池持ち時間を確保できるかが重要である．

よって，サービス開発において，開発対象のアプリによるエネルギー消費量を単に把握するだけでなく，ユーザの 1 日の端末利用パターンに与える影響を評価することが必要である．例えば，図 1.7 に示すように，他のアプリとの比較で自身のアプリの消費量を把握する，または，自身のアプリに閉じずにエネルギー消費の原因を理解することが，サービス提供者と利用者双方にとっての最適解を導くことになると考える．

そのためには，実際のユーザが，端末にどのアプリをインストールし，どの程度使用しているか，どのような設定設定で使用しているかなど，1 日の端末の利用実態の全体像を知る必要がある．

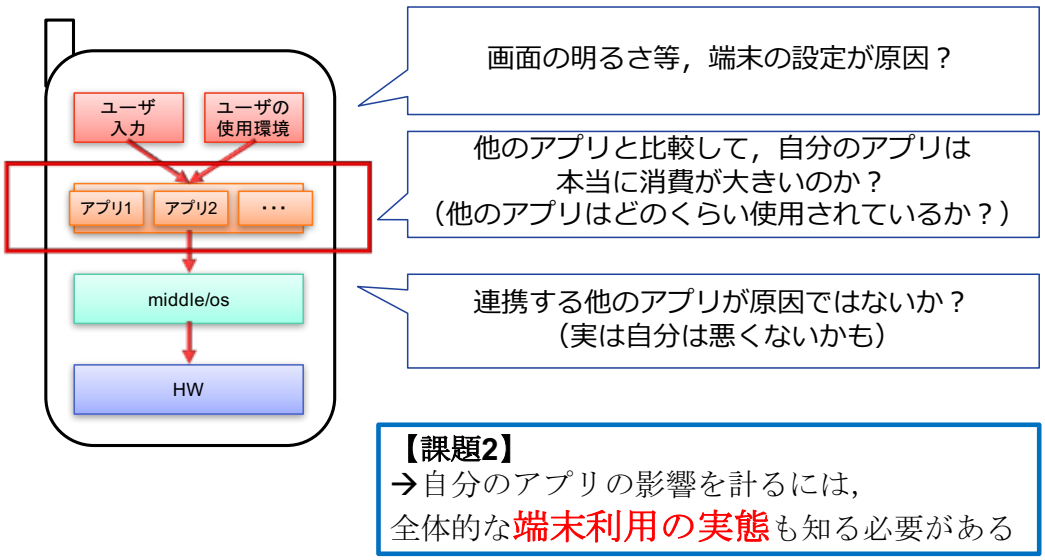

図 1.7　電池持ち時間特有の課題 2

実際のユーザによる端末の利用実態を知ることは，サービス開発だけでなく端末開発における省電力化にも有効であると考えられる．以下に事例を示す．

OS・ミドルウェアの改善

川崎ら [2] は，複数の Android 端末のアプリによる通信が広域的に同時発生し，ネットワーク側で輻輳を起こすという問題に対し，OS でのアプリ動作制御の改善を提案している．この問題は，アプリが，ディスプレイの点灯など端末状態の変化に応じたタスク実行を可能にする仕組みに起因してお

り，前述の論文は原因となるアプリ挙動を模擬する評価用アプリ用いた実験により，改善効果を評価している．しかし，この挙動の発生はアプリの設計やユーザの端末操作に依存しているため，より実効的な効果を示すためには，評価端末にインストールされるアプリの組み合わせや，ディスプレイの点灯などの端末状態の変化頻度など，実際のユーザの使用状態を考慮した評価条件を設定することが望ましい．

端末カタログの電池持ち時間表記

フィーチャーフォン時代より，端末メーカーおよびキャリア各社は自社が取り扱う端末のカタログにおいて各端末の電池持ち時間を表記している．表記の仕方は「連続通話時間」や「連続待受時間」など様々であるが，端末の機能や性能などのスペックと同様，ユーザが端末を購入する際の参考として重要な情報である．しかし，ユーザの実際の利用時に，ユーザが期待するカタログ表記値通りに使用できないことで，端末品質に対するユーザの不満を生じさせてしまうことがある．特にスマートフォンの場合は，前述したように，非常に多種多様なアプリ利用が可能であり，実際の電池持ち時間はユーザの使い方に強く依存するため，カタログ表記と実態との乖離が生じやすい．よって，このような不満を生じさせないようにするには，どのような使い方を想定した電池持ち時間であるか，その想定はユーザが望む端末利用の実態に近いかなどを考慮したカタログ表記が必要である．

このように，ユーザの利用実態を考慮することは，サービス開発でも端末開発でも，満たすべき品質要求を適切に把握する上で重要である．しかし，端末開発者の立場ではユーザは遠い存在であることが多く，利用実態を把握することは困難である．また同時に，サービス開発者の立場でも，自身のサービス以外の利用実態を把握することは困難である．

1.3 研究の目的

本研究は，実際のサービス利用における端末省電力化を実現するため，ハードウェアからユーザレベルまで統合的に分析可能な手法を実現することを目的とする．

また，前節のサービス開発における課題分析の議論より，本研究において解決すべき課題は以下の通りである．

課題 1）ソフトウェア開発者でも容易に利用可能な消費電力の把握

課題 2）端末利用の実態の把握

図 1.8 は，本研究のコンセプトの全体像と，前述の課題と具体的な取り組みの対応関係を示したものである．端末全体の消費エネルギー（E）は，端末の各ハードウェアコンポーネントが動作する際の消費電力（H），アプリなどのソフトウェアによるハードウェアコンポーネントの使用（A），さらにユーザによるアプリ使用（U）の 3 つの要因が掛け合わされることで決定するものとし，各要因に対し，1) 端末の消費電力の可視化，2) アプリなどのソフトウェア消費電力の分析ツール，3) 実際のユーザによる端末利用実態データの提供を行うことで，サービス実利用を考慮した消費電力評価手法の実現を目指す．各事項の概要は後述の通りである．

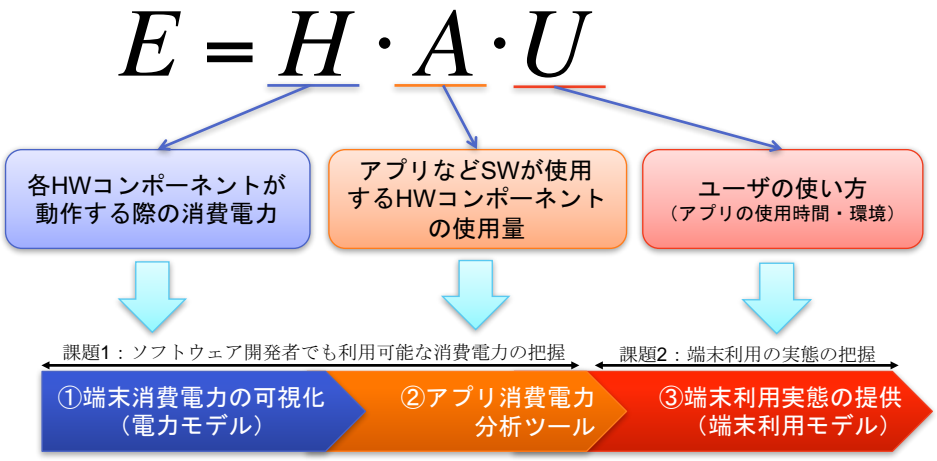

図 1.8 本研究のコンセプト

端末消費電力の可視化技術

いかなる端末の省電力化検討においても，端末が実際にどの程度の電力またはエネルギーを消費しているか定量的に把握し，改善効果を評価することが不可欠である．従来より計測機器を用いた測定手段があるが，例えばアプリ開発者に対して，測定環境，測定に要するスキル，試験工数を確保することは現実的ではない．また，端末全体の消費電力しか測定できないため，CPU やディスプレイなど，省電力化のために注目すべきモジュールを特定することが難しい．本テーマでは，ソフトウェアやサービスの開発者にも利用可能な端末の消費電力評価を実現するため，実際の端末の消費電力をモジュール単位で精度良く推定する手法を提案する．

アプリ消費電力分析ツール

前述の端末消費電力の可視化技術の応用として，アプリなどソフトウェア開発者が，自身が開発したソフトウェアによる端末消費電力を評価するための消費電力分析ツールを構築する．本ツールは，端末エミュレータではなく実際の端末上で評価対象ソフトウェアを動作させた際に得られるログに基づき，ソフトウェア動作時の端末消費電力またはエネルギーを定量的に評価可能である．本ツールを用いて，端末消費電力の推定精度を評価するとともに，既存のツールで問題となる端末側ログ取得に要するオーバヘッドも評価し，実際のソフトウェア開発において実用的なツールであることを示す．

端末利用実態データの提供

実際のユーザ使用において効果のある省電力化を検討するためには，実際のスマートフォン利用実態を考慮した，適切なシナリオや条件のもと課題分析や評価を行うことが重要である．例えば，前述のアプリ電力分析ツールのように特定のアプリに着目した評価手段が提供されたとしても，評価用端末の各種設定やその他のアプリのインストール状況など評価環境のセットアップが実態と乖離していると，正確な分析結果を得ることができない．し

かし，現在のスマートフォン利用は非常に多種多様であり，アプリ開発者がユーザの端末利用の実態を的確に把握し，実態にあわせた評価シナリオを作成することが困難である．そこで，本テーマでは，スマートフォン利用を前提としたサービスの企画から開発における様々な課題設定や効果検証において，ユーザ利用実態を考慮した評価シナリオの構築を支援できるデータ，すなわち端末利用モデルを提供することを目的とする．具体的には，実際のスマートフォンユーザ約 700 名を対象にした端末利用実態調調査で得たデータを元に，アプリ利用などの端末利用パターンを抽出し，パターン毎の特徴を分析した端末利用モデルを構築することで，前述の様々な検討に活用可能なデータの作成を目指す．

1.4 本論文の構成

本論文の構成は以下の通りである．第 2 章では本研究に関連する先行研究を紹介し，本研究との比較を行う．第 3 章では，前述した 1) 端末の消費電力の可視化とともに，2) アプリ消費電力の分析ツールについて提案し，第 4 章では，3) ユーザ実利用を考慮した評価シナリオの策定のための端末利用モデルについて述べる．最後に第 5 章でまとめを述べる．

第 2 章　関連研究

本章では，前章にて設定した研究課題に関連する既存研究について紹介する．

2.1　ソフトウェアの消費電力評価

本節では既存の電力モデルに基づく電力推定手法やアプリ電力評価手法についてまとめ，3.3 に後述するアプリ消費電力評価手法を実現するための課題への適合性について述べる．

2.1.1　電力モデルを用いた電力推定手法

電力モデルを用いた電力推定手法として，線形式で特定のハードウェアコンポーネントの消費電力をモデル化しておき，電力推定の際にはモデルパラメータを抽出するためのログを取得し，モデルへの入力とすることで推定する手法が提案されている．

Lee ら [3] は CPU の命令をパラメータとした線形モデルを提案し，高い精度でプロセッサの電力を推定している．一方，モデル化の対象が CPU に限定されているため，端末全体の消費電力を推定するためには無線通信など他のハードウェアコンポーネントも対象にしたモデルが必要である．また，このようにハードウェアの構成や挙動に近い粒度のデータをパラメータに採用したモデルでは，推定のためのパラメータ取得には高負荷なハードウェアエミュレーションやデータ観測が必要であることから，実機におけるアプリの実利用時に用いることは現実的ではない．

石原ら [4]，Kaneda ら [5] の手法は，システムを構成する主要なコンポーネントの消費電力を，CPU 稼働率など OS レベルで容易に取得可能なデータをパラメータとしてモデル化することで，システム全体の消費電力を精度良

く推定する手法を提案している．この手法は，モデルの設計概念としてシステム全体をカバーしやすく，様々なコンポーネントに対応するログ収集のオーバヘッドが小さいという利点がある．一方で，モデル自体がマルチコア CPU とその周波数制御や，3G/LTE といったモバイル無線インタフェースとその挙動など，近年のスマートフォンのハードウェア構成と動作が考慮されていない．近年の CPU は省電力化のためにマルチコア化されコア毎に多段階の周波数制御が可能な設計になっている．また，モバイル無線通信はモバイル環境での端末使用には不可欠であり，フィーチャーフォンにおいても大きな電力消費を伴うコンポーネントである．したがって，これらのコンポーネントの特徴を考慮しないモデルでは，実際のアプリ使用時の消費電力推定に大きな推定誤差が生じると考える．

2.1.2 スマートフォンソフトウェアを対象にした電力評価

近年のスマートフォンとそのアプリ評価を対象にした研究としては Michigan 大による ARO (Application Resource Optimizer) と呼ばれるツール [6] があげられる．本ツールはモバイル無線インタフェースの電力モデルに特徴がある．3G/LTE の無線通信では，端末と無線基地局間における複数種類の無線チャネル割当を制御している RRC（Radio Resource Control）と呼ばれる機構がある [7]．この割当状態（以下，RRC State）によって，モバイル無線インタフェースの消費電力は大きく異なるとされており，ARO は RRC State をパラメータとした電力モデルを用いることで，モバイル無線通信における電力を精度よく推定している．ただし，RRC State を特定するために必要なログとしてパケットキャプチャを用いており，ログ取得に特権が必要であることと，ログ取得にかかるオーバヘッドが大きいことから，実機におけるアプリの実利用時への適用は困難である．本研究では対象開発者を広く取るためこのような制約は認められない．

Yoon らは，実際の端末を構成する主要なコンポーネントを幅広くカバーした精度の高いモデル [8] と，そのモデルに基づくアプリ消費電力を評価する手法を提案している [9]．しかし，消費電力の推定に必要なパラメータを得るために，システムコールを記録するなどカーネルレベルでのログ取得を伴う

手法であるため，カーネルの改変が前提になる．カーネルの改変やシステムコールの記録はシステム上特権を必要としており，セキュリティの観点から市販の端末では通常は許可されていない．このため，前述と同様に，モデルの利用者を広く取るため本研究ではこの前提を受け入れることはできない．

また，特定のアプリを対象に電力評価を行った事例として，スマートフォンにおける主要な端末利用のひとつであるブラウザを対象にした研究がある．Sampson らは，WebChar（for Web Characterization）と呼ばれるシステムを構築し，消費電力量の増加を引き起こす HTML と CSS の実装の特徴を分析している [10]．本システムでは，Web ページをロードする際，HTML タグや CSS のプロパティ，使用セレクタの頻度などのコンテンツの特徴と，消費電力量を学習データとし，SVR（Support Vector Regression）モデルを生成する．このモデルを利用した分析により，opacity プロパティの使用や float を用いたページレイアウトなど，消費電力の増加を引き起こす HTML や CSS の実装を特定している．また，Thigarajan らも，E コマース，ソーシャルネットワーキングサービス，Web メール，動画サービスなど様々なカテゴリの Web コンテンツを対象に，実機で各コンテンツをダウンロードし描画されるまでの端末の消費電力を実測し，CSS や JavaScript などのコンテンツ構成要素ごとに分解してプロファイリングする手法を提案している [11]．両者らの評価手法は，Web コンテンツの実装改善を目的にしたものであり，また，端末の消費電力を Web コンテンツの構成や挙動で説明させる，回帰分析的なアプローチであるといえる．このようなアプローチは，Web コンテンツの実装変更など改善策を直接的に導出するという点において有益であると考えられるが，分析結果が分析対象の母集団の片寄りに依存し一般性を担保することが難しいこと，説明変数がアプリやコンテンツ特有のものであり異なる種類のアプリに対してはそのまま適用できない，といった問題がある．また，端末の消費電力を測定すること自体がソフトウェア開発者にとっては大きな負担となるため，別途，正確かつ容易に消費電力を把握する手段が必要である．

2.1.3 研究課題の位置づけ

ソフトウェアの消費電力評価に関する既存研究について，前項までの議論を整理し，本研究における課題設定の位置づけを述べる．表 2.1 は，既存研究と本研究の位置づけを示したものである．

前述した既存研究における消費電力の評価手法は，評価に用いるモデルの設計概念の違いから，ハードウェア依存型，システムワイド型，アプリ特化型に区分できる．各手法の分類結果は表 2.1 の通りである．

次に，本研究の目的のひとつである，実際の端末やサービス開発/利用状況下においても利用可能な消費電力の評価手法の実現に対する，既存手法の適合性議論は，主に 1) モデル生成あるいはモデル利用時に必要なデータ収集の適用性と，2) 電力評価の有用性の 2 点で各手法の特徴を整理できる．さらにこの 2 点は，1) はデータ取得に要するオーバヘッドと一般に市販される実機への適用性，2) は評価対象アプリの多様性と電力の推定精度に細分化される．

表 2.1 電力評価における既存研究と本研究の位置づけ

モデル分類	既存研究	1 データ収集の適用性		電力評価の有用性	
		オーバヘッド	実機適用性	アプリ多様性	推定精度
ハードウェア依存型	Lee	×	×	△	◎
システムワイド型	Yoon	△	×	○	○
	ARO	△	×	○	△ (3G/LTE は◎)
	石原	○	○	○	○ (汎用端末のみ)
	提案手法	○	○	○	○
アプリ特化型	Sampson	○	○	×	△
	Thigarajan	○	○	×	△

ハードウェア依存型モデルに基づく手法は，ハードウェアレベルの正確な挙動に基づくものであるから，このアプローチを端末全体に拡張することで非常に高精度な電力推定を実現できると考えられるが，オーバヘッドの問題などデータ収集の観点で現実的に採用することは難しい．一方，アプリ特化型の特徴はハードウェア依存型の逆傾向にあり，特定のアプリに特化した評価手法であるがゆえ，異種のアプリに対しての有効性はない．システムワイド型は，データ収集の適用性や電力評価の有用性の両観点において，各手法

が一長一短あるものの，モデル自体がアプリに依存せず，端末全体をカバーしようとする設計概念ゆえ，評価対象アプリの多様性については適合性が高い．

よって，サービス実利用における消費電力評価を実現するためには，既存研究には以下の問題があり，これらを同時に解決することが必要であると考える．

1) 実際のサービス利用環境では許容できないデータ収集が前提である
2) 近年のスマートフォンの特徴を考慮した電力推定ができない

そこで，本研究では，システム全体をカバーしやすくデータ収集オーバヘッドの小さい石原らの手法のモデルを踏襲しつつ，かつ ARO のように近年のハードウェアコンポーネントの特徴を考慮したモデル拡張を行うことにより，上記問題を解決可能なアプリ消費電力の評価手法を提案する．

2.2 スマートフォン利用におけるユーザ行動理解

本節では，省電力化の問題に限定せず，スマートフォンを前提としたサービスや技術の新規提案や問題解決に幅広く活用できる根拠データとしてモデルを提案するため，モデルの活用対象となりうる事例やユーザの利用実態に関する分析事例について議論する．

2.2.1 モデルの活用領域

Tomita らは，新商品・サービスの商品化プロセスについて，中高齢者向け携帯電話の商品化を事例に報告している [12]．Tomita らは，旧来の商品化プロセスについて，マーケティングにはじまる企画プロセスと要素技術の研究開発プロセスが独立し，商品コンセプトや機能・品質要件のすりあわせが効率的に進まないことにより，商品化まで多大な時間を要し，結果的に顧客ニーズとはずれた商品化がなされてしまうという問題を指摘している．前述の携帯電話の事例では，携帯電話市場は若年層が中心であった 2001 年当時，研究開発部門もマーケティングに参画し，中高齢者向けを潜在市場として

見いだし，徹底したニーズ分析と技術課題の深掘りを行うことで，商品化における生産性の向上と，新たな顧客開拓に成功した．近年のスマートフォン利用は，従来よりも自由度が高く，顧客ニーズも多様化していることから，対象ユーザに対する分析と明確化がさらに重要であると考える．

また，近年のサービス動向のひとつとして，スマートフォンで取得した位置情報を活用したサービスが多く提案されており，地図だけでなく現在地に応じたクーポン配信など様々なサービスが提供されている [13,14]．一方，位置情報は，氏名・住所などと同様に，重要な個人情報としてプライバシー保護の対象であるため [15]，端末の位置情報取得は初期状態では無効化されている．この設定を有効化するには，ユーザによる設定変更が必要であり，アプリなどが自動的にこの設定を有効化することはできない．また，スマートフォンにおける位置情報取得機能は，取得時間の高速化や取得精度の向上のため，携帯電話基地局の情報や GPS 測位情報などを相補的に用いて位置取得を行う仕組みが採用されており，それぞれどの情報を用いるか設定が細分化されている．このため，なんらかのサービスを提供したとしても，各ユーザの設定状態に依存して，当初の想定したユーザ規模を下回ったり，サービスの品質低下をまねく恐れがあるため，このような端末設定の利用実態は事前に考慮しておくことが望ましい．

次に，近年のスマートフォンにおける要素技術に関連する先行研究について述べる．川崎ら [2] は，複数の Android 端末のアプリによる通信が広域的に同時発生し，ネットワーク側で輻輳を起こすという問題に対し，OS でのアプリ動作制御の改善を提案している．この問題は，アプリが，ディスプレイの点灯など端末状態の変化に応じたタスク実行を可能にする仕組みに起因しており，前述の論文は原因となるアプリ挙動を模擬する評価用アプリ用いた実験により，改善効果を評価している．しかし，この挙動の発生はアプリの設計やユーザの端末操作に依存しているため，より実効的な効果を示すためには，評価端末にインストールされるアプリの組み合わせや，ディスプレイの点灯などの端末状態の変化頻度など，実際のユーザの使用状態を考慮した評価条件を設定することが望ましい．他の先行研究においても，有機 EL ディスプレイの省電力化のために表示コン

テンツの表示色を変化させる手法[16]や，GPU の省電力化のためにアプリの描画処理を推定し GPU を省電力状態に遷移させる手法[17]などが提案されているが，同様に主要なアプリのみに限定した課題検討や評価にとどまっており，実効的な効果を示すには至っていない．特に，モバイル向けソフトウェア工学においては，実際のユーザの使用状況を考慮した前提条件の設定が重要であると指摘されている[18–20]．

2.2.2 ユーザ行動の理解

Ferreira ら[21]は，アプリのユーザビリティやユーザ体験の向上を開発者が検討するための知見として，ユーザによる 15 秒以下の短時間のアプリ使用に着目し，その発生要因となるユーザコンテキストとの関係を分析している．被験者端末でのログ収集とインタビューに基づき，40% のアプリ起動は短時間の使用であること，ユーザが自宅など一人でいるときに最もこのようなアプリ使用がなされていることを示している．また，Parate ら[22]は，アプリ利用時のユーザ体験を向上させるため，アプリ利用に伴うコンテンツ取得の遅延を抑えるためのプリフェッチを実現させるための要素技術として，ユーザが次に使用するであろうアプリとその起動タイミングを予測する手法を提案している．その予測手法を実現するため，系列データとしてアプリの起動履歴や起動間隔に着目した特徴量選定を行っている．これらの先行研究は，スマートフォン利用におけるユーザ行動傾向を理解するという観点で，アプリ利用に着目することの有効性を示す事例であり，本論文における分析でも，この考え方と同様にアプリ利用を主要な特徴量として採用している．しかし，これらの先行研究はそれぞれ特定の目的に特化したデータの分析となっているため，前述したユーザの利用実態を考慮した端末省電力化の効果検証などに活用できるデータとはなりにくい．

Falaki ら[23]は，2008〜2009 年における実際のスマートフォンユーザ 255 名の端末で収集したログを用いて，ユーザの利用実態を分析している．この調査では，アプリ利用以外にも，ディスプレイ状態，ネットワークトラフィック，バッテリ状態などに関するログを解析し，例えば一日のデータ送受信量など，様々な観点でユーザの利用傾向はユーザによって大きく異な

ることが示されている．しかし，この調査は現在の主要なスマートフォンが登場した初期に実施されたものであることから被験者選定に片寄りがあり，この分析結果はスマートフォンが広く一般に普及した現在の利用実態とは整合しないと考える [24]．また，1 日のアプリ毎の使用時間も示されているが，全ユーザで平均化されたものであり，ユーザによるスマートフォン利用の多様性や特徴を詳細に示したものではないため，本論文が目指すサービスや技術の新規提案や問題解決に幅広く具体的に活用できるデータとしては分析粒度が粗いと考える．スマートフォンが広く普及し，非常に多種多様なアプリが提供されている今日，ユーザの利用実態を的確に把握するためには，アプリをはじめとする利用パターンをとらえることが重要である．

2.2.3 利用パターンの把握

スマートフォン利用の多様性をパターンに分類し，その特徴を分析した先行事例について述べる．Patil ら [25] は，実際のユーザ 33 名によるアプリ使用時における CPU/GPU 負荷の傾向をクラスタリングにより分析している．しかし，調査対象の被験者数が少なく，主要なアプリケーションの使用時に限定したデータ収集および分析しかなされておらず，分析結果の有用性を十分に示せていない．本論文における我々の分析結果は，前述のような調査設計に対しても有益な知見となることが期待される．また，Li ら [26] は，数百万人ものユーザのアプリ管理操作などのログからスマートフォン利用傾向を分析しているが，アプリマーケット全体の視点で，アプリ人気度やユーザのアプリ選択の特徴といった全体傾向を示すものであり，次項にて本論文が示すアプリの利用パターンのように，ユーザによる 1 日のアプリ使用を詳細に示すものではない．Zhao ら [27] は，本論文と同様のモチベーションである，実データに基づくユーザ理解と研究開発への応用のもとに，クラスタリングにより約 10 万ユーザのアプリ使用傾向からユーザを 382 パターンに分類している．しかし，Zhao らの分析は，1 時間毎に起動されたアプリ名のみが列挙された粗いサンプリングデータに基づいたものであるため，例えば，1 日にどのアプリを何分使用したかなど，詳細な考察を行うことはできない．また，382 もの非常に多くのパターン（クラスタ）を導出している

が，そのうちの特徴的な 6 パターンのみの傾向を説明するにとどまっているように，パターンが多すぎるゆえにパターン毎の特徴付けが困難になるだけでなく，分析結果を活用する際にも考慮すべきパターン数が多く大きな労力を要すると考える．サービス企画でも研究開発においても，特に初期検討時のように，検討対象となるユーザが絞り込めていない段階では，まずは全体傾向を俯瞰してとらえることができる粒度でパターン分けを行うこと重要であり，必要に応じて細分化していくことが一般的である．

2.2.4 研究課題の位置づけ

我々のスマートフォン利用モデルは，特に，前述の初期検討時に幅広く活用されることを目指している．また，マーケティングにおけるセグメンテーション分析 [28] のように，サービス/製品の対象セグメントの存在とボリュームを定量的に把握することと，技術課題・仮説/効果検証を行うことを一つのモデルで同時に実現することで，効率的に実際のユーザを考慮したサービス・製品企画や研究開発ができるようになると考える．

第 3 章　電力モデルに基づくアプリ消費電力評価手法

本章では，Android アプリケーションの実利用環境において利用可能なアプリ消費電力の評価手法を提案する．

3.1　概要

本手法は，スマートフォンを構成する各ハードウェアコンポーネントの特性と消費電力の関係から生成した端末の消費電力モデルを用いることで，アプリ消費電力の推定を可能にする．本章では，近年の端末の消費電力を妥当な精度で推定できること，推定に必要なログ収集の負荷が低いことを要件とした評価手法を実現するため，マルチコア CPU やモバイル無線インタフェースとその特徴を考慮したモデル拡張を行う．本手法において，一般的なアプリ利用のシナリオを対象に 10% 前後の誤差で電力推定できること，3.8% 程度の低いオーバヘッドでログ収集が実現できることを確認した．

3.2　はじめに

近年，Android 端末をはじめとするスマートフォンの普及に伴い，利用者から電池持ち時間の改善が要望されるようになった．スマートフォンが消費する電力は，無線インタフェースや CPU などのハードウェアリソースの稼動により決定され，その利用率はアプリケーションプログラム（以降，アプリ）の挙動による．従って，例えば必要以上に通信を行うなどの非効率な挙動を示すアプリを改善することで電池持ち時間の改善が可能である．古庄ら[29] は特定アプリにおいて利用者の平均的な操作パターンに対しアプリの挙動を最適化することでそのアプリが消費する電力（アプリ消費電力）を低減させられることを示した．この知見を進めるなら，アプリが市場に出てから

ではなく開発時点でそのコードの消費電力や電池持ち時間への影響を開発者にフィードバックし，実現される機能そのものや実装の品質に対するコストとして消費電力を意識してもらうことが有効であると考えられる．

本研究は，開発者自身によるアプリ消費電力の最適化の実現のために，AndroidOS でのアプリ消費電力の評価手段を開発者が容易に利用可能な形態で提供することを目的とする．具体的には，アプリ実行時に現実のハードウェアで消費される電力を精度よくソフトウェアのみで推定することによるアプリ消費電力の評価手法およびツールを提案する．本手法は，スマートフォンを構成する各ハードウェアコンポーネントの特性と消費電力の関係から生成した端末の消費電力モデルを用いることで，アプリ消費電力の推定を可能にする．

アプリ消費電力の評価手段をソフトウェアで実現することは，測定器などの機材を用いた従来の測定手段に起因する課題を回避するとともに低コスト化，流通の容易性といったメリットがある．機材を用いた従来の電力測定をアプリ開発の現場に持ち込むことは，必要な機材の手配による時間や費用が開発コストを押し上げる，さらに屋外や移動中など実際のアプリ使用状況において専用の測定器を使用しながら測定することは自然なアプリ利用を妨げる，といった諸点を考慮しなければならない．昨今のアプリ開発環境は SDK(Software Development Kit) の形態で配布され，実機エミュレータを内包するなどソフトウェアのみで完結しており，アプリ消費電力の評価手段を開発者に提供する際にも同じアプローチを取ることで安価に広く配布することが可能になり開発者の利便性は高まるであろう．

本章の構成を以下に示す．3.3 では本研究の課題とアプリ消費電力の評価手法に対する要件設定を行い，3.4 では関連研究について議論し本研究の位置付けを明かにする．3.5.1 では，提案手法が取り入れる電力モデルの基本設計と近年のスマートフォンのハードウェア特性に対応するためのモデル拡張について述べ，3.6 では，拡張した電力モデルを用いたアプリ消費電力可視化ツールを紹介する．最後に，3.7 では本ツールの電力推定精度と推定に必要なログ収集のオーバヘッドを評価し，3.8 でまとめと今後の課題を述べる．

3.3 課題と要件

自然な利用状況におけるそのアプリ消費電力の評価手段をソフトウェアにて実現するには以下の課題がある．

課題 1) ハードウェアの特性をモデル化した消費電力推定手段を備えること．

課題 2) 実際のアプリ利用状況をデータ化しこれに基づいた推定が可能であること．

課題 1) の解決によってソフトウェアのみによる消費電力の測定手段の構築が可能になり，全ての開発者に測定機材を配備する必要がなくなる．さらに課題 2) の解決は，アプリの利用シーンに過度に介入することなく簡易なデータ取得のみによって評価を成立させるために必要である．これにより，実際の様々な利用シナリオ・環境における評価が可能になると考える．

我々はスマートフォンの電力モデルとアプリ実行時のログを用いた，アプリ消費電力可視化ツールを提案する．

課題 1) の解決のため，電力推定手法を以下のとおりとする．石原ら [4] の提案する電力モデルを採用し，スマートフォンのハードウェアコンポーネント毎の稼動量と実測した消費電力をもとに，対象とする端末の特性に適合させるべく重回帰分析によりパラメータ係数を求めることで電力モデルを作成する．近年のスマートフォンにおいても高い精度で推定するために，マルチコア CPU におけるコア数や周波数の切替えなど近年のハードウェアコンポーネントとその特徴を考慮すべくモデル拡張を行う．

課題 2) の解決のため，端末上でアプリを実行している間に所定のログを収集しておき，そのログからハードウェアコンポーネント毎の稼動量をパラメータとして求め，電力モデルに適用しアプリ消費電力を推定する．本ツールの構成では測定器による実測は機種毎の電力モデル作成時のみであり，個別のアプリ・利用シナリオ毎の実測は不要である．

上記を踏まえ，以下 2 点の要求条件を設定するものとし，後述の評価にお

ける指標とする．

1）システム全体を対象に電力推定精度が妥当であること．

2）実利用時のログ収集にかかる負荷が低いこと．モデルが妥当であっても実アプリの評価時の測定動作がハードウェアコンポーネントの稼動を増やすことは好ましくない．

3.4 関連研究

本節では既存の電力モデルに基づく電力推定手法やアプリ電力評価手法についてまとめ，3.3 に述べた課題への適合性について述べる．

電力モデルを用いた電力推定手法は，線形式で特定のハードウェアコンポーネントの消費電力をモデル化しておき，電力推定の際にはモデルパラメータを抽出するためのログを取得し，モデルへの入力とすることで推定する手法が提案されている．

Lee ら [3] は CPU の命令をパラメータとした線形モデルを提案し，高い精度でプロセッサの電力を推定している．一方，対象が CPU など特定のコンポーネントに限定されていること，推定のためのパラメータ同定には高負荷なハードウェアエミュレーションが必要であることから，アプリの実利用時に用いることは現実的ではない．

石原ら [4]，Kaneda ら [5] の手法は，システムを構成する主要なコンポーネントの消費電力を，CPU 稼働率など OS レベルで容易に取得可能なデータをパラメータとしてモデル化することで，システム全体の消費電力を精度良く推定する手法を提案している．この手法は，モデルの設計概念としてシステム全体をカバーしやすく，様々なコンポーネントに対応するログ収集のオーバヘッドが小さいという利点がある．一方で，モデル自体がマルチコア CPU とその周波数制御や，3G/LTE といったモバイル無線インタフェースとその挙動など，近年のスマートフォンのハードウェア構成と動作が考慮されていない．

近年のスマートフォンとそのアプリ評価を対象にした研究としては Michigan 大による ARO (Application Resource Optimizer) と呼ばれるツール [6] が提案されている．本ツールはモバイル無線インタフェースの電力モデルに特徴がある．3G/LTE の無線通信では，端末と無線基地局間における複数種類の無線チャネル割当を制御している RRC（Radio Resource Control）と呼ばれる機構がある [7]．この割当状態（以下，RRC State）をパラメータとした電力モデルを用いることで，モバイル無線通信における電力を精度よく推定している．ただし，RRC State を特定するために必要なログとしてパケットキャプチャを用いており，ログ取得に特権が必要であることと，ログ取得にかかるオーバヘッドが大きいことから，実機におけるアプリの実利用時への適用は困難である．本研究では対象開発者を広く取るためこのような制約は認められない．

Yoon らは，実際の端末を構成する主要なコンポーネントを幅広くカバーした精度の高いモデル [8] と，そのモデルに基づくアプリ消費電力の推定手法を評価する手法を提案している [9]．しかし，消費電力の推定に必要なパラメータを得るために，システムコールを記録するなどカーネルレベルでのログ取得を伴う手法であるため，カーネルの改変が前提になる．カーネルの改変やシステムコールの記録はセキュリティの観点から市販の端末では通常は想定されておらず，本研究ではこの前提を受け入れることはできない．

以上より，既存研究ではアプリの実利用におけるアプリ消費電力評価における前述の課題・要件全てを満たすことはできない．本研究は，システム全体をカバーしやすくログ収集オーバヘッドの小さい既存研究 [4] のモデルを踏襲しつつ，かつ ARO のように近年のハードウェアコンポーネントの特徴を考慮したモデル拡張を行うことにより，上記要件を満たすアプリ消費電力の評価手法を提案する．

3.5 モデルによる電力推定

3.5.1 モデル概要

電力推定に用いる端末の電力モデルについて述べる．本電力モデルは，CPU など端末を構成するハードウェアコンポーネント毎にモデル化された電力の総和をとることで，端末全体の電力を表現する．

$$P_{est} = c_{offset} + \sum_{i}^{N} c_i p_i$$

前記コンポーネント毎の電力は，コンポーネント i 毎にその稼働状態を示す値をパラメータ p_i とし，所定のパラメータ係数 c_i およびオフセット定数 c_{offset} からなる数式で求められる．c_{offset} は，モデル式より，コンポーネントの稼動状態に依存しない端末消費電力の基準値であり，端末が Deep Sleep 状態 [30] ではないが，ディスプレイが消灯し，CPU などのその他のコンポーネントが idle 状態であるときに必ず消費する電力とみなすことができる．

3.5.2 モデル生成方法

モデルの生成方法を図 3.1 に示す．モデルの生成は従来手法と同様，各コンポーネントを様々な負荷レベルの下で稼動させるように構成されたトレーニングベンチを端末上で動作させた際の端末全体の消費電力と各モデルパラメータ p_i の実測値をサンプルデータとして上記モデル式に代入し，重回帰分析によりパラメータ係数 c_i を求める手法となる．

サンプルデータ取得のためのトレーニングベンチは，以下の通り動作するよう作成した．

＜ CPU ベンチ＞

1 スレッドで一定の浮動小数点演算を周期的に繰り返す処理．周期はループ処理の内に，シナリオで指定した期間の Sleep 処理を含めることで，一定の CPU 負荷状態を維持する．また，マルチコア CPU の負荷状態を幅広くサン

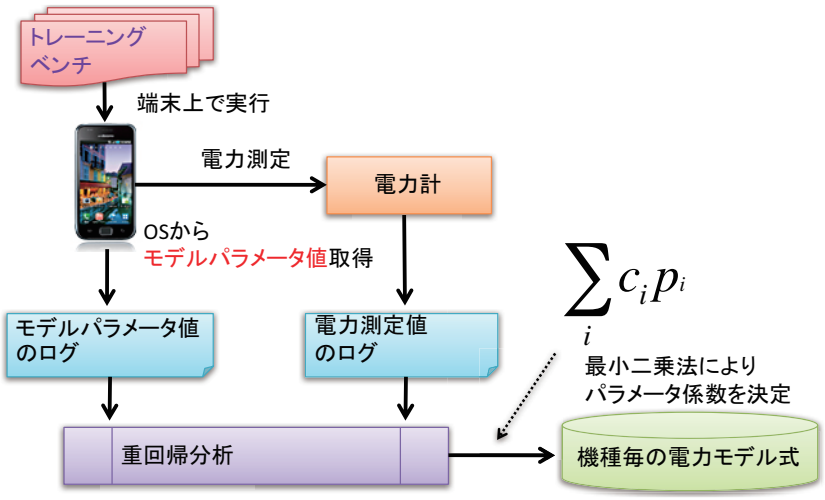

図 3.1　電力モデルの生成方法

プル取得するため，任意の数だけ前記スレッド処理を複数同時に実行できるようシナリオ指定できるようにした．

＜無線通信ベンチ＞

任意サイズのデータ送信または受信を繰り返す処理．前述の CPU ベンチと同様，シナリオでデータサイズや Sleep 期間を指定することで，一定のスループットを維持しつつ様々なパターンのサンプルを取得する．本ベンチは，3G/LTE や Wi-Fi 無線通信インタフェースに全てに対して用いる．

＜ディスプレイベンチ＞

Android システム上定義されているディスプレイ輝度を任意の値に維持した状態で画面表示を行う処理．

＜ GPS ベンチ＞

シナリオで指定した期間，GPS を測位状態にする処理．

＜ディスクベンチ＞

SD カードなど所定のストレージ領域に対し，任意サイズのデータを読み込

みまたは書き込みを繰り返す処理.

＜ GPU ベンチ＞
シナリオで指定された個数のキューブを指定期間継続して描画する処理．描画処理には OpenGL ES 2.0[31] を使用.

本研究においては，実際に様々な機種の電力モデルを用いて評価することを想定しているため，モデルの基本構造として，CPU 稼働率など一般性があり抽象度の高いパラメータを用いることで，モデル生成過程や使用時における機種依存性を極力排除できるようにする.

本提案手法におけるモデル生成では，従来研究 [4] のモデル設計を基に，マルチコア CPU およびモバイル無線インタフェースのモデル設計を検討しモデル拡張を行う．本論文では，拡張箇所のみ 3.5.3 および 3.5.4 に後述し，それ以外のコンポーネントも含めた全体のモデル式および生成結果については 3.5.5 に後述する.

3.5.3 モデル拡張:マルチコア CPU のモデル化

近年のスマートフォンでは，マルチコア CPU が搭載されており，省電力化などの目的でシステム稼働状況に応じて使用するコア数や周波数を切り替える制御がなされていることが一般的である．従来研究 [4] は，シングルコア CPU を対象に，CPU 稼働率のみをパラメータにしたモデルであるため，前述の制御による影響を考慮した電力推定を行うことができない．よって，本提案手法による CPU 電力を説明すべきパラメータの拡張を行うことにより，コア数や周波数の切替えも考慮したモデル生成を行う．マルチコア CPU のモデル化の考え方としては，従来研究 [32] において，コアやキャッシュなど CPU を構成するコンポーネントごとに独立にモデル化し，それらを加算する形で定式化する方法が示されている．本提案手法においても，同様の考え方を踏襲してモデル化を行う.

本提案手法における CPU の電力 P_{cpu} を以下に示す．

$$P_{cpu} = P_{core1} + P_{core2} + ... + P_{coreN} = \sum_{i}^{N} c_i p_i = \sum_{i}^{N} c_i f_i u_i$$

ここで，P_{core1}〜P_{coreN} はそれぞれ CPU を構成するコア毎の電力であり，P_{cpu} はコア毎の電力を加算したものである．近年の周波数制御はコア毎になされることから各々独立にモデル化する．次に，コア i の電力 P_{corei} を構成する c_i，p_i，f_i，u_i は，それぞれ，コア i に対するパラメータ係数，パラメータ，動作周波数，CPU 稼働率である．本提案手法においては，周波数毎の性能差を加味しつつ CPU による仕事量を示す値をパラメータとするため，動作周波数に CPU 稼働率を乗算した値をパラメータとした．

本提案手法における CPU 電力モデルのパラメータの妥当性を確認するため，3.5.5 に後述する端末と 3.5.2 に前述したトレーニングベンチを用いて予備検証を行った．トレーニングベンチは，様々な負荷レベルで CPU を稼動させるよう動作し，その動作パターン毎に端末の消費電力と，パラメータ導出の元になる動作周波数および CPU 稼働率を測定した．上記試験により測定したパターン毎の消費電力とパラメータとの関係を図 3.2 に示す．本提案手法において設定したパラメータと消費電力の間に線形性が見られることから，CPU 電力モデルのパラメータとして妥当であると考えられる．

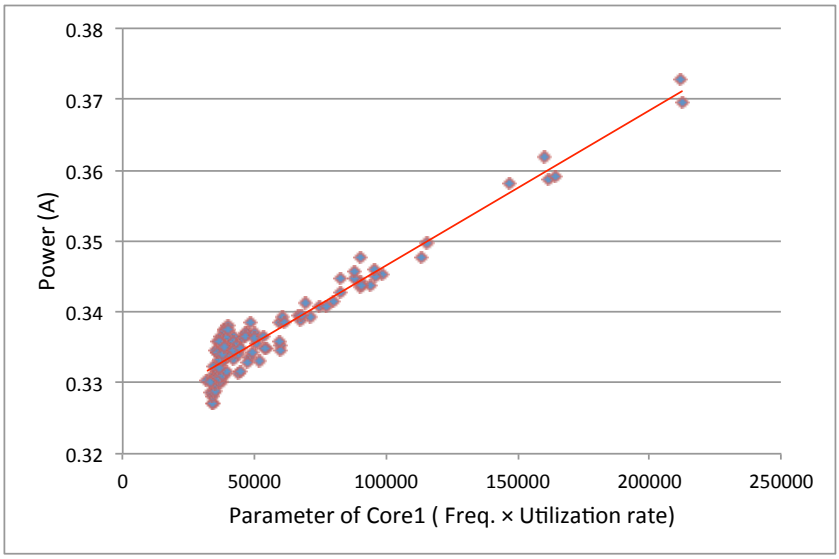

図 3.2　消費電力と CPU パラメータの関係

3.5.4　モデル拡張:モバイル無線インタフェースのモデル化

RRC State の概要

3G/LTE 無線通信では，RRC と呼ばれる端末と無線基地局間における複数種類の無線チャネル割当制御を行う仕組みがある．RRC は，多数の端末が在圏する基地局配下における無線リソースの効率利用のため，各端末からの通信要求に応じて，端末と無線基地局間で複数種類のチャネル割当状態（以下，RRC State）を切り替えるなどの制御を行っている．

3G/LTE における RRC State 遷移の概略図を図 3.3 に示す．例えば 3G においては，アプリ利用時など高スループットでのデータ送受信を行うために用いる個別チャネル（DCH），低スループットでデータ送受信が可能な共通チャネル（FACH），無通信状態で待機的な状態として一定期間割り当てられるチャネル（PCH），データ送受信のための無線リソースの解放状態を指す IDLE など，複数の RRC State が定義されており，データ送受信の発生により上位の DCH に，一定期間の無通信状態を検知することにより下位の State に遷移するなど，State 間の遷移のトリガとなる条件が規定されている．端末側の消費電力という点では，State 毎に消費電力が大きく異なり，

上位の State であるほど大きな消費電力を消費すること，消費電力は IP レベル以上のデータ送受信の有無に関わらず，どの State にあるかでほぼ決まることが知られている．図 3.4 は，3G の各 State における端末の平均消費電力の例である．

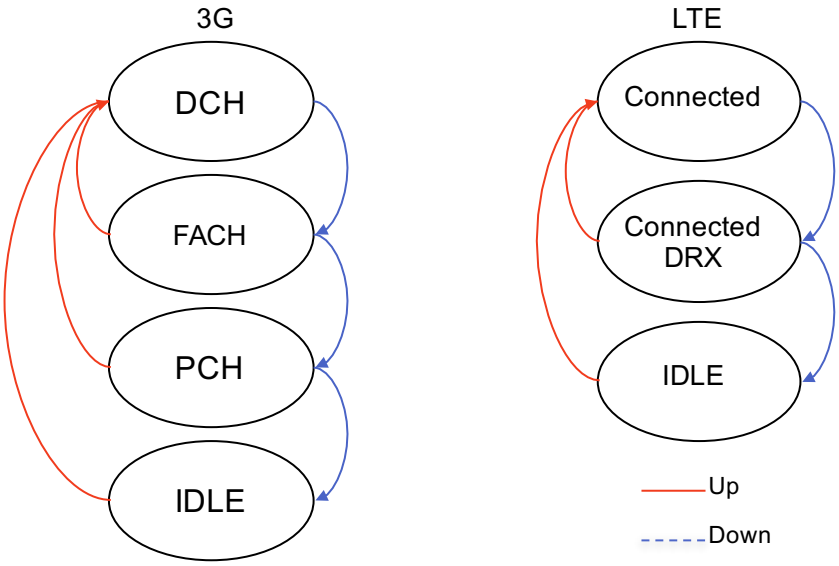

図 3.3　3G/LTE の RRC State 遷移の例

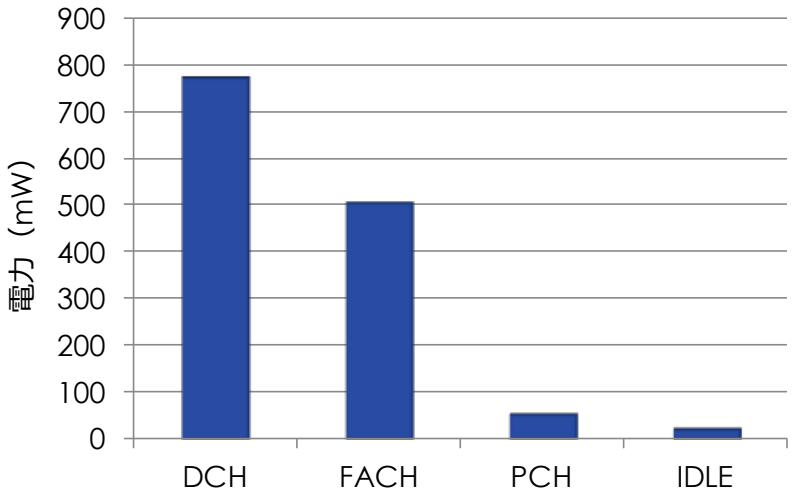

図 3.4　3G の各 State における端末の平均消費電力の例

State 遷移条件について説明する．図 3.5 は，LTE を例に，State 遷移と遷移条件となるトリガを示したものである．A はアプリ等がデータ送受信のための通信要求が発生した際に発動するトリガであり，これを受けてデータ送受信が可能な上位の State，すなわち Connected に即座に遷移する．一方，B および C は下位の State に遷移するために定義されており，それぞれ，B は予め定められた期間の無通信状態が継続されることを契機に発動するトリガ（無通信タイマー），C は画面 OFF などユーザ操作による端末状態の変化を契機に発動するトリガである．B の無通信タイマーの設定期間は通信キャリアや State により異なるが，おおよそ 10〜120 秒の範囲内の値が設定される．また，B だけでは最も低消費電力な IDLE に遷移するまでに少なくともタイマーの設定期間の時間を要してしまい，この間の電力消費が無駄になってしまう．このため，C の画面 OFF のように，ユーザの端末利用の終了と見なせる端末状態の変化を契機に即座に IDLE へ遷移させる Fast Dormancy という仕組みが備わっている．

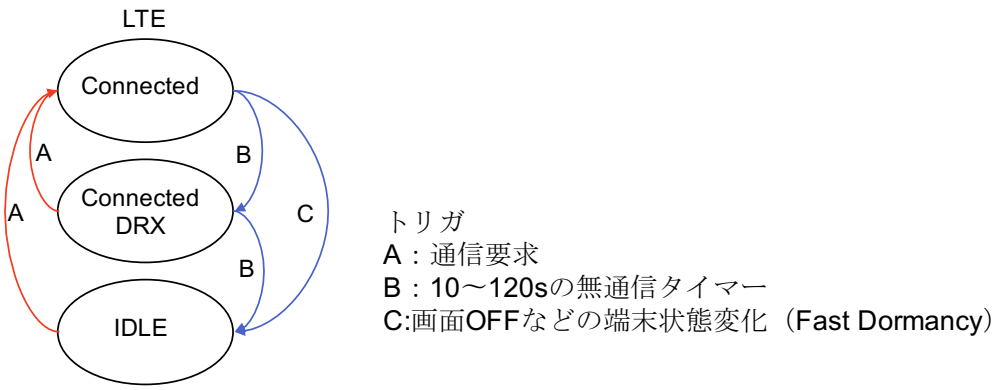

図 3.5　LTE における RRC State 遷移条件のトリガ

RRC State の推定によるパラメータ取得

ARO ではこの RRC State をパラメータとしたモデルを用いることで，モバイル無線通信における電力を精度よく推定することが可能である．しかし，一般に市販されている端末で RRC State を直接取得することはできないため，ARO ではパケットキャプチャを端末上で収集し，データ送受信の

タイミングや量からトリガを検知し，RRC State を推定する手法を提案している．

しかし，2.1 に述べたように，この手法では端末上でログ収集オーバヘッドが大きいことに加え，取得に特権が必要な手段でありセキュリティ上の理由から広く一般の開発者が利用するアプリ評価手段を実現する上で適切さに欠ける．

また，パケットキャプチャだけでは画面 OFF などの端末の状態変化を検知できないため，前述したトリガ C による State 推定を行うことができず，結果的に消費電力の推定精度の悪化を招く恐れがある．

本提案手法では，ARO と同様の RRC State 推定ロジックを踏襲するが，前述の問題を回避するため，パケットキャプチャ以外で AndroidOS で容易に取得可能なデータをもとに A〜C のトリガを検知するステートマシンを設計し，バックグラウンド常駐型の Android アプリとして実装した．具体的には，A および B のトリガ検知のため，一定周期でデータ送受信量の統計値を取得し，周期ごとに送受信データの有無や量を確認する．また，C のトリガ検知のために，BroadcastReceiver と呼ばれる端末状態イベントを検知する API を使用し，画面 OFF 状態を得る．パケットキャプチャと比較し，収集するログのデータ量を抑えられることから，ログ収集オーバヘッドを削減できると考えられる．なお，本ロジックの詳細は付録 A のソースコードの通りである．

提案手法による RRC State 推定精度と推定遅延の評価

パケットキャプチャはデータ送受信をトレースしたデータであり，ARO はトラフィック発生タイミングや状況を正確に把握し，RRC State 遷移推定に反映できるが，本提案手法では一定周期のサンプリングデータから遷移推定を行うという仕組み上，推定結果に 1 周期分の遅延が生じるという欠点がある．本提案手法の実装にあたっては，RRC State や消費電力推定に大きな誤差を起こさない範囲で，オーバヘッドを削減することを優先するため，ログ収集の周期を 1 秒とする．

本提案手法における推定誤差の影響を確認するため，前述の 1 秒周期でのログ取得による RRC State 推定の精度と推定遅延の予備検証を行った．図 3.6 に実験環境の構成を示す．検証実験では，端末上でランダムにデータ送受信を発生させる試験用プログラムを動作させた状態で，本提案手法による RRC State 推定を行うと同時に，QxDM[33] と呼ばれる試験用ツールを用いて実際の RRC State 遷移ログを取得した．

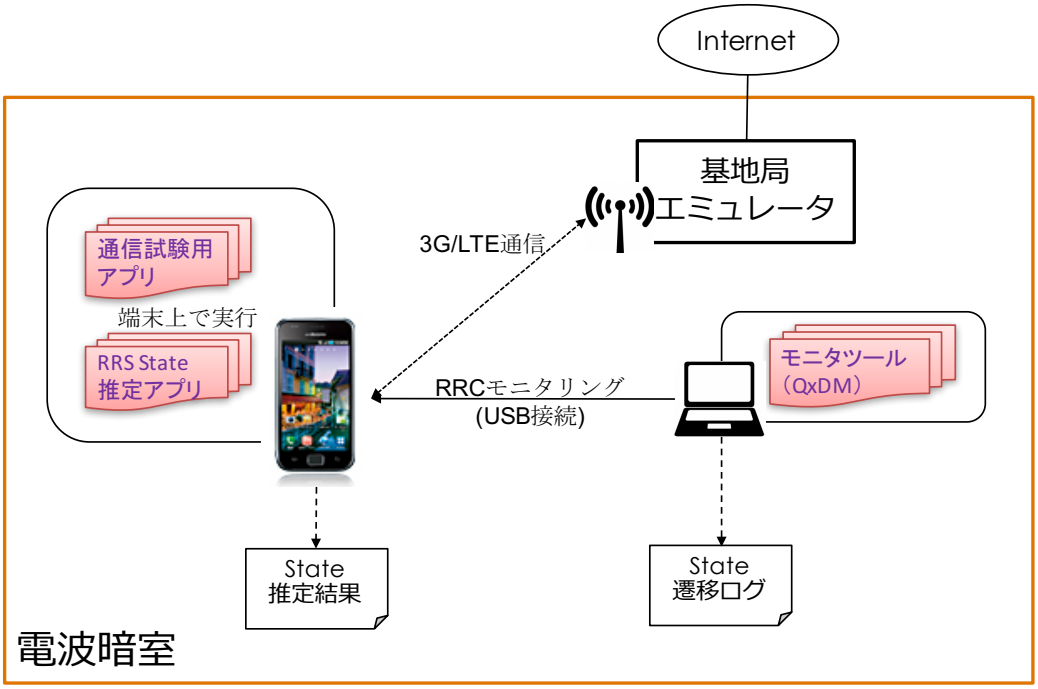

図 3.6　実験環境の構成

推定結果と QxDM による測定結果との比較により，推定精度と推定遅延の評価した結果を表 3.1 に示す．推定精度は，試験プログラムの動作期間のうち，正しく RRC State が推定されていた期間が占める割合とする．LTE の場合，96.4% と精度良く推定できていることが示された．一方，3G の場合は，PCH から IDLE への遷移において大きな推定遅延が生じたことにより，推定精度は 71.3% となった．この推定遅延に対する改善は今度の課題とするが，1）PCH から IDLE 以外の遷移では大きな遅延なく推定できていること，2）3.4 に示した通り，PCH および IDLE 状態における実際の消費電力は非常に小さく [34]，両者に大きな差が見られないことから，消費電力の

推定精度には影響がないと考えられる．

表 3.1　RRC 推定精度と推定遅延の検証結果

Time overlap	**Estimation delay of state transition (sec.)**			
3G	**to DCH**	**to FACH**	**to PCH**	**to IDLE**
71.3%	0.425	1.679	0.926	71.681
LTE	**toConnected**	**to CDRX**	**to IDLE**	
96.4%	0.532	1.654	1.826	

3.5.5　モデル生成結果

本論文において用いるモデルの生成結果を示す．Android OS が稼働するスマートフォンのうち，Qualcomm 社の MSM8960 チップセット [35] を搭載した評価端末 A と，Texas Instruments 社の OMAP4460 チップセット [36] を搭載した評価端末 B の 2 機種を対象にモデルを生成した．

モデル化の対象とするコンポーネントと，パラメータを含むモデル式を表 3.2 に示す．端末全体の消費電力モデルは，コンポーネント毎の消費電力をモデル化し，線形結合したものになる．モデル生成の結果得られたパラメータ係数と，各パラメータに多重共線性の有無を確認するための指標である VIF（Variance Inflation Factor）を表 3.3 に示す．一般的に，VIF が 10 以上の値をとると多重共線性の可能性が高いとされているが，表 3.3 の通り，全ての係数に係る VIF が 1〜3 程度と十分に低い結果となった．重回帰分析によるモデルの当てはまりの良さを示す指標として調整済み R 二乗があるが，評価端末 A，B ともにそれぞれ 0.9664，0.9875 と高く，モデルの有意性が確認できた．評価端末 B のモバイル無線インタフェースは LTE 非対応のため該当する係数（i=10〜14）は生成していない．

表 3.2　コンポーネント毎のモデル式

コンポーネント	モデル式
CPU	$c_1 f_{core1} u_{core1} + c_2 f_{core2} u_{core2}$
Display	$c_3 p_{brightness}$
3G	$(c_4 + c_5 p_{sendBps} + c_6 p_{rcvBps}) p_{Dch} + c_7 p_{fach} + c_8 p_{pch} + c_9 p_{idle}$
LTE	$(c_{10} + c_{11} p_{sendBps} + c_{12} p_{rcvBps}) p_{LTEconnected} + c_{13} p_{LTEcdrx} + c_{14} p_{LTEidle}$
Wi-Fi	$(c_{15} + c_{16} p_{sendBps} + c_{17} p_{rcvBps}) p_{flagWiFi}$
GPS	$c_{18} p_{gps}$
Disk	$c_{19} p_{readByte} + c_{20} p_{writeByte}$
GPU	$c_{21} p_{gpuload}$

表 3.3　パラメータ係数

	評価端末 A		評価端末 B	
i	c_i	VIF	c_i	VIF
0	1.462e-01		1.517e-01	
1	1.375e-07	3.028175	2.482e-07	2.204509
2	1.162e-07	2.938385	1.642e-07	1.741155
3	2.340e-04	1.632474	6.380e-04	1.107095
4	1.563e-01	2.425848	1.531e-01	2.945161
5	3.265e-07	1.716143	5.328e-07	2.216406
6	1.107e-07	1.384222	1.020e-07	1.421324
7	1.179e-01	1.074541	7.726e-02	1.170108
8	1.384e-02	1.074810	2.630e-02	1.005708
9	1.402e-02	1.074463	2.343e-02	1.005638
10	2.054e-01	1.947452		
11	8.000e-08	1.411644		
12	2.048e-08	1.296661		
13	3.892e-02	1.074483		
14	1.723e-02	1.074530		
15	3.210e-02	2.043827	4.648e-02	2.066004
16	1.633e-07	1.410098	9.862e-08	1.449676
17	1.218e-07	1.410647	6.724e-08	1.438688
18	3.712e-02	1.018166	6.119e-02	1.011056
19	3.307e-10	1.061168	6.669e-10	1.043131
20	1.336e-09	1.039160	3.315e-09	1.032754
21	7.797e-10	1.057354	1.515e-01	1.034463

本研究においてモデル化したコンポーネントは，近年の様々なアプリにおいて利用度が高いことと，アプリ動作時に端末全体の消費電力に占める割合が大きいコンポーネントを優先的にモデル化の対象とした．各コンポーネントに対して，検討したパラメータが連続値をとり，電力値との関係に線形性がある場合，あるいはパラメータ自体がコンポーネントの状態を示す値で，各状態に対応した電力値が定まる場合には，同じモデル生成の考え方，方法

のもと，新たなコンポーネントを追加したモデル拡張も可能であると考える．例えばカメラなど，既知の未対応コンポーネントに対しては，モデル生成方法との適合性確認も含め，今後の課題としてモデル拡張を検討する予定である．

一方，本研究においては，アプリから OS にアクセスすることで容易に取得可能なコンポーネントの稼働量を利用することを要件としているため，3.7.1 で議論するような，OS がアプリに挙動を観測する手段を提供しないコンポーネントについてはモデルとして扱えない．さらに OS が提供する稼働量が実際のコンポーネント稼働を反映しない場合もモデルとして扱えない．例えば CPU 仮想化により同一端末内に複数の OS が実行される環境には対応できない [37]．

3.6 アプリ消費電力可視化ツール

前記電力モデルを搭載したアプリ消費電力可視化ツールについて述べる．

本ツールの機能構成を図 3.7 に示す．本ツールは，端末側で電力評価に必要なログを収集するログ収集アプリと，PC 側で前記ログを解析し，アプリ実行時の電力推定及び表示などを行うデータ分析用ツールから構成される．

ログ収集アプリは，端末上で評価対象のアプリが動作する間，電力モデルのパラメータ値生成に必要なログを 1 秒毎に収集する．データ分析用ツールは，特定の端末機種に対してあらかじめ作成した電力モデルを備えており，ログ収集アプリから得たログから生成したモデルパラメータから，その端末の 1 秒毎の平均電力を算出する．本ツールにより，アプリ利用による消費電力を，図 3.8 に後述するような時系列データとして可視化することができる．ハードウェアコンポーネント毎の内訳も可視化できることにより，何が原因で電力消費されているか，アプリ仕様や動作と対応づけて開発者が分析しやすくなると考える．

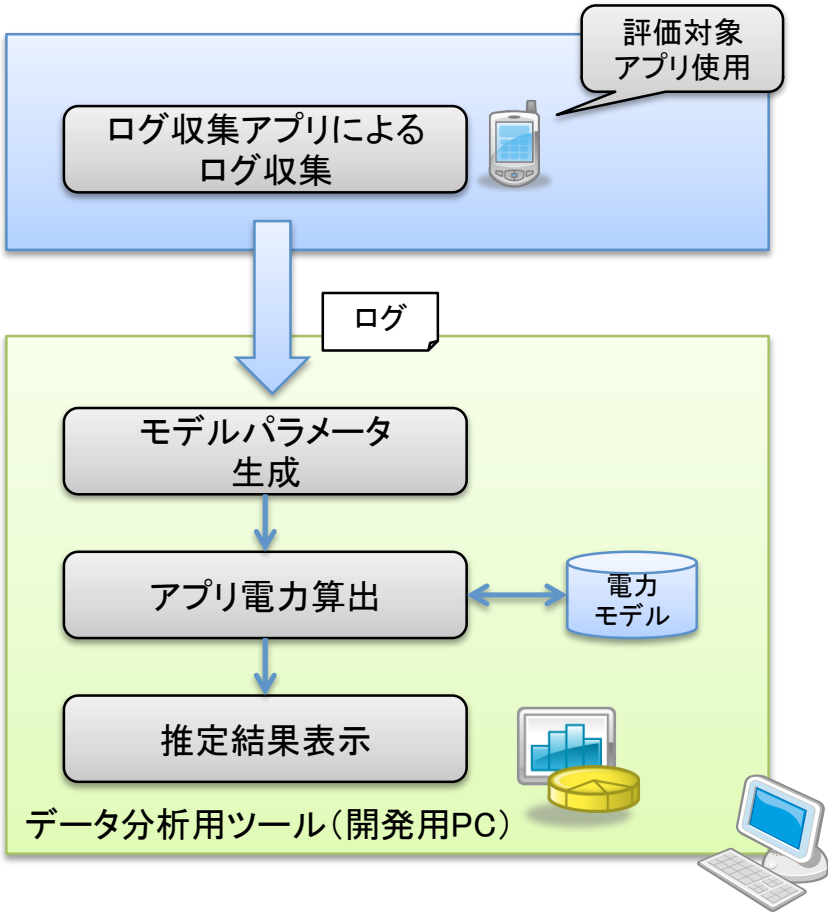

図 3.7　アプリ消費電力可視化ツールの機能構成

3.7　評価

3.3 に掲げた要件に対してそれぞれ評価を行った．要件 1）に対して電力推定精度を示す．要件 2）に対してログ収集にかかるオーバヘッドの影響を確認する．所定の評価シナリオに従い，端末上でアプリが動作する際の電力を測定器により実測し，推定結果と比較することで電力推定精度を評価する．ログ収集にかかるオーバヘッドは，ログ収集アプリによるログ収集を行った際の CPU 稼働率を測定し，ログ収集なしの場合と比較した増分を評価する．

3.7.1 電力推定精度の評価

評価方法

電力推定精度の評価方法について述べる．評価用アプリには，実ユーザが一般的に利用するアプリ（メール，地図，カレンダー，動画プレーヤ，電話帳）と，CPU/GPU バウンドなベンチマークアプリとして AnTuTu[38] を用い，各々以下のような操作によるアプリ利用を評価シナリオとする．

＜メールアプリ＞
アプリ起動→新規メール作成→電話帳より宛先選択→メール件名・本文の入力→送信→アプリ終了
＜地図アプリ＞
アプリ起動→現在地取得→地図表示・閲覧→アプリ終了
＜カレンダーアプリ＞
アプリ起動→スケジュール新規作成→件名など入力→登録→アプリ終了
＜動画プレーヤアプリ＞
アプリ起動→コンテンツ一覧から動画選択→動画再生→再生終了後，アプリ終了
＜電話帳アプリ＞
アプリ起動→新規登録→氏名，電話番号など入力→登録→アプリ終了
＜ AnTuTu　CPU 負荷ベンチ＞
アプリ起動→ CPU 負荷ベンチ選択・実行→アプリ終了
＜ AnTuTu　GPU 負荷ベンチ＞
アプリ起動→ GPU 負荷ベンチ選択・実行→アプリ終了

上記のシナリオに従ったアプリ利用中に，本ツールのログ収集アプリを動作させログ収集を行い，同時に端末全体の消費電力を実測する．測定器は Agilent 社製 N6705B を用いて，測定器からの DC 電源出力にて端末を動作させている．

推定精度は，ツールによる推定値と測定器による実測値との比較により評価する．各シナリオの実施にあたっては，端末のディスプレイを消灯し，端

末を極めて消費電力の低い Deep Sleep 状態にした後，シナリオを開始する．時系列に精度評価を行うためには，両データ間でシナリオの開始タイミングを同期させる必要があるが，端末・測定器同士の時刻同期ができなかったため，代わりに，前述の操作の結果，データから観測される消費電力の下降を開始タイミングとみなすようにした．（図 3.8 の 14 秒目付近）

評価結果

本評価における測定結果として，メールアプリ利用における時系列に推定/実測結果を示したものを図 3.8 に，各アプリの測定期間での平均誤差を図 3.9 に，各アプリの測定期間での消費エネルギーを図 3.10 に示す．

図 3.8 に示す通り，アプリ使用区間全体を通して，実測値の変動に対応付けて電力を推定できており，平均誤差は約 10% である．他のアプリについても，図 3.9 に示す通り，全体的に 10% 前後程度の誤差で推定できていることから，本ツールは一定の精度でアプリ電力の推定が可能であると確認できた．

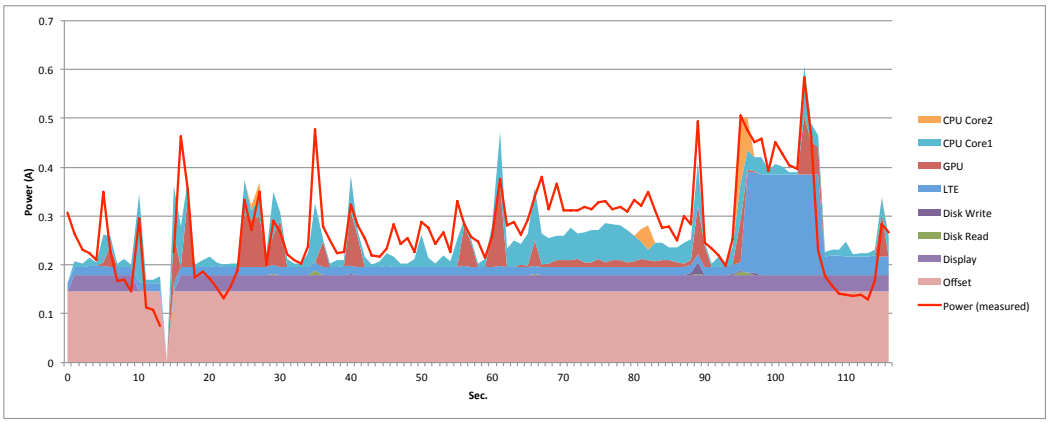

図 3.8　メールアプリ使用時の消費電力推定値および実測値

一方，図 3.8 の 20 秒付近，40〜90 秒，110 秒付近では，最大で約 0.1A 程度の誤差が出ている．これは，ディスプレイ電力の推定誤差であると考えられる．評価端末は有機 EL ディスプレイを搭載しているが，今回のディスプレイ電力モデルは，元々 LCD ディスプレイを対象に，Android OS 上で規定

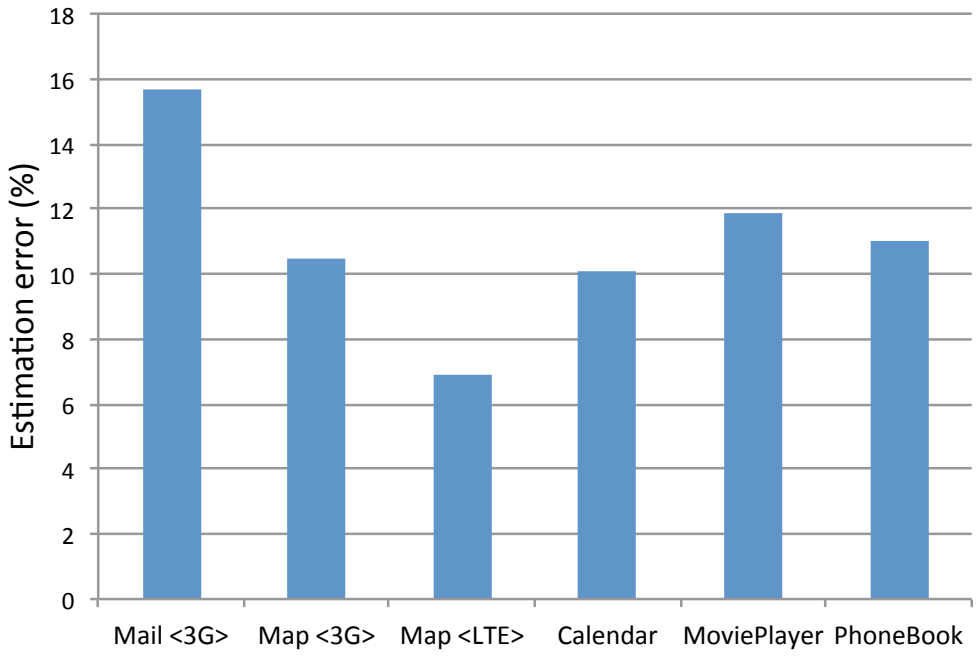

図 3.9　アプリ毎の電力推定誤差

されているディスプレイ輝度値のみをパラメータにしており，有機 EL の特徴をパラメータ化したモデル生成を行っていない．従来研究 [16] によれば，有機 EL ディスプレイの消費電力は画素毎の表示色に依存し，RGB 値により説明できるとされている．前述の区間では，ユーザ操作を伴うメールの宛先検索や文字入力など多くの画面遷移を伴い，様々な色調の変化が起こっていることが誤差要因であると考えられる．有機 EL ディスプレイ対応については今後の課題とする．

図 3.8 に示す評価では，全体の消費エネルギーに対して Offset 分が大きく占める結果となった．3.5.1 に前述した通り，Offset 分は端末が Deep Sleep していない Idle 状態において必ず消費してしまう電力であるが，それゆえに，アプリの改善により実行時間を短縮させることも，効果的に省電力化を図る一事例であることがわかる [39]．特に，ユーザに見えない周期的なバックグラウンドタスクの実行を伴うアプリに対する評価観点の一つとして，本ツールの使用が有効であると考える．

一方で，Offset 分は重回帰分析の結果得られた端末消費電力の静的成分に

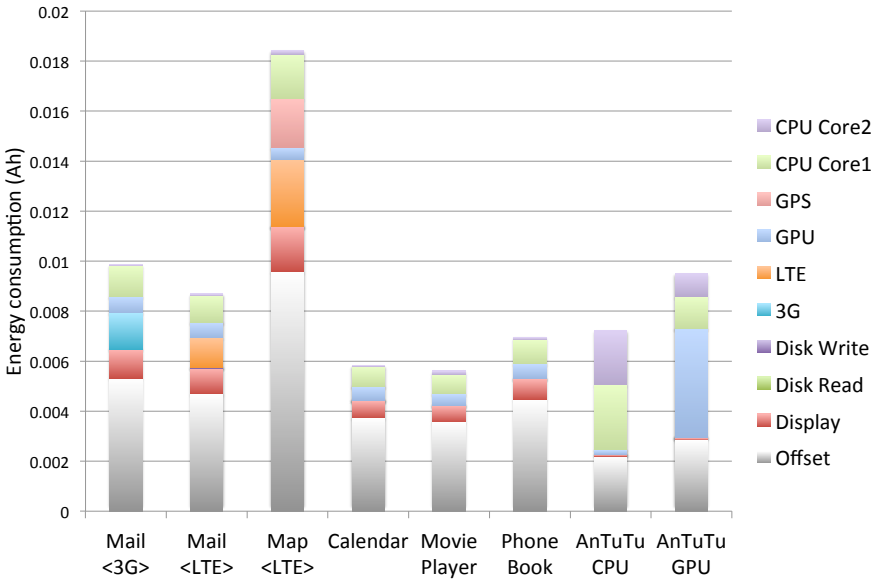

図 3.10　アプリ毎の消費エネルギー推定値

過ぎず，ツールを用いて Offset 分に対する詳細な分析を行うことはできない．Offset 分は，1）上述の通り CPU を含むチップセット全体での待機電力であること，2）システムが制御しアプリから挙動が観測できないようなコンポーネントの動作，例えば加速度や照度などの何らかのセンサーが OS から観測不能な形で動作することがあった場合には，これらのコンポーネントの動作分の消費電力も含むことになる．従って Offset は機種ごとのモデルの予測誤差となり得る．上記 2）に該当するコンポーネントの ON/OFF などの制御手段なり動作条件が与えられていればモデル化における精度低下を回避できると考える．

ログ収集におけるオーバヘッドの評価

本ツールのログ収集アプリにおけるログ収集動作によるオーバヘッドを評価する．本提案手法においては前述の通り，アプリの実利用を妨げとならな

いようログ収集の負荷が低いことを要件 [2] としている．3G および LTE の無線インタフェースの消費電力推定精度を高めるため，本提案手法は ARO で提案されている無線インタフェースの電力モデルの考え方を踏襲しているが，前述の要件を満たすため ARO よりもログ収集の負荷が小さい方式である点が特徴の一つである．本項では，ARO との比較により，本提案手法による前述のオーバヘッド低減の効果を示す．

ログ収集アプリは，評価対象アプリの動作中， CPU 稼働率などモデルパラメータの生成に必要なログを 1 秒毎に収集する．したがって CPU 稼働率が本来のアプリ由来のものより大きくなることが課題である．

まず，このログ収集に伴うオーバヘッド評価を行うため，以下の試験パターンにて端末全体の CPU 稼働率の測定を行った．

(1) ログ収集動作無しに DisplayON，操作なしにて放置
(2) ログ収集動作ありに DisplayON，操作なしにて放置

測定結果は，図 3.11 の通り，(1) ログ収集無において 1.3%，(2) ログ収集有において 5.1% となり，ログ収集に要するオーバヘッドは 3.8% 程度であることが示された．

次に，ARO におけるログ収集と比較し，本手法のログ収集に伴うオーバヘッドが低いことを示す．以下 (3) 〜 (5) の試験パターンで CPU 稼働率の測定を行った．

ARO で用いる収集ログはアプリ動作時のパケットキャプチャデータ (pcap ファイル) であり，測定時に用いる試験用アプリはデータ送受信を伴う必要があるため，複数の Web ページを順次自動でダウンロード・表示する簡易ブラウザを試験用アプリとして用いた．

(3) 簡易ブラウザのみ (ログ収集無)
(4) 簡易ブラウザ (提案手法のログ収集有)
(5) 簡易ブラウザ (パケットキャプチャ有)

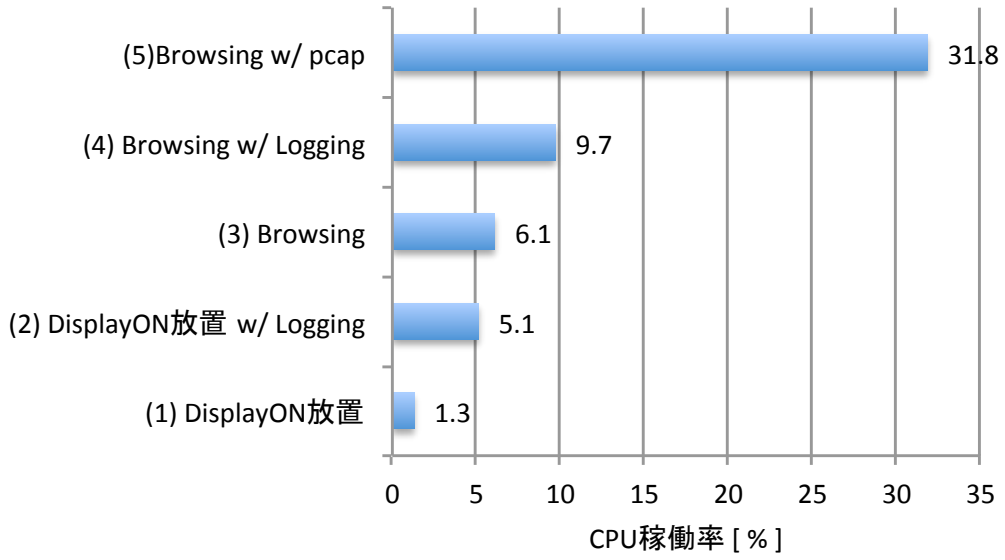

図 3.11　ログ収集によるオーバヘッド評価

測定結果を図 3.11（3）〜（5）に示す．前述と同様に，（3）と（4）との比較により，提案手法のログ収集オーバヘッドは 3.6% 程度の増加となる．一方，ARO を想定したパケットキャプチャ収集では，（3）と（5）の比較により，25.7% の CPU 稼働率増加が観測された．

3.5.1 に示したように，CPU 稼働率は CPU 電力のモデルパラメータであり，ログ収集処理による稼働率の増加分に比例して端末で電力が消費されることになるため，パケットキャプチャにより収集したログを用いた消費電力の推定結果は，実際のアプリ利用による端末の消費電力と乖離する． 本評価の測定によって乖離の一例が示された．パケットキャプチャ取得に要する処理負荷は，アプリや使い方によるトラフィックパターンや量に依存して増減するため，その補正は単純ではない．さらに，スマートフォン向けアプリはデータ送受信を伴うものが多く，パケットキャプチャ動作による消費電力の増加は一般的なアプリ利用の妨げになる可能性もある．

一方，本提案手法は，一定の周期で所定のログを収集する仕組みを用いているため，アプリの種別や利用シナリオを問わず，低いオーバヘッドでログ

収集が可能である．さらに本提案手法では，パケットキャプチャを用いずに 3G/LTE の RRC State 遷移を推定する仕組みを実現したことで，ログ収集のオーバヘッドを抑えつつも，ARO と同様に無線電力の推定精度を向上させることが可能になった．

3.8 まとめ

本章では，実際のアプリ使用において容易に利用可能なアプリ消費電力の評価を実現するため，電力モデルに基づく消費電力の推定によるアプリ消費電力の評価手法およびツールを提案した．モデルに基づく電力推定手法において，昨今の端末のハードウェア構成に対応すべく，マルチコア CPU のコア数および周波数の変動や，3G/LTE の RRC State を考慮するよう既存研究に見られる電力モデルを拡張した．電力推定精度を評価した結果，一般的なアプリで 10% 前後程度の誤差で推定できることが示された．さらに，従来手法（ARO）と比較し，3.8% 程度と小さなオーバヘッドかつ特権を必要としない RRC State 遷移の推定手段を用いたことで，一定の精度を保ちつつ，容易に利用可能なアプリ消費電力の評価手段を実現することができた．

今後は新たなデバイスである有機 EL ディスプレイへの対応など，誤差の要因に対して推定するアプリ消費電力のさらなる高精度化を行う予定である．

第 4 章　ユーザ利用実態調査に基づくスマートフォン利用モデル

4.1　概要

本章では，実際のスマートフォンユーザ約 700 名に対するアンケート調査と約 400 名の端末ログ収集調査によるデータを用い，アプリケーション使用など実際のスマートフォン利用パターンを導出し，パターン毎にその特徴を示すことで，様々なサービス企画・研究開発に有益なスマートフォン利用モデルを提案する．本モデルは，1 日単位のスマートフォン利用を，ユーザ属性やアプリケーション使用傾向などの特徴を定量的に示すものである．クラスタリングによる利用パターン分析の結果，全体的に 6 つの利用パターンの存在が確認された．また，同一ユーザでも日によって異なる利用パターンが存在するという仮説を検証したところ，例外的なパターンを除くと，90% のユーザの利用パターンは高々 2 つ程度であることが確認された．

4.2　はじめに

近年，スマートフォンの契約者数は従来のフィーチャーフォンを超え，スマートフォンは広く一般に利用されるモバイル端末となった [1)]．また，端末機種の拡充とともに，個人でもアプリケーション（以降，アプリ）を作成・配布できる環境が整備され，従来よりも多種多様な用途のアプリが提供されるようになったことで，ユーザは生活の様々なシーンにおいて，各自の行動や趣味嗜好にあったサービスを幅広く選択できるようになった．

このようなスマートフォンおよびサービスの実現のため，サービス企画におけるマーケティングから，アプリを含めたシステムを構成する要素技術の研究開発まで，各々の役割で様々な課題解決がなされてきた．

しかし，スマートフォン利用の進化の方向性や可能性は無数に考えられるがゆえに，各々の取り組みの位置づけや価値を明確に示すことが難しくなることがある．いずれの役割・取り組みにおいても，妥当性を示しつつ目標・課題を設定し，課題解決の効果検証を行うことが重要である．例えば，新規サービスの提案では，対象ユーザは誰か，本当に対象ユーザが存在するのか，何人くらい存在するのか，サービス提供によりどのくらいの効果が見込まれるのかなどを定量的に見積もることが求められる．また，なんらかの問題解決のための技術課題にアプローチする際にも同様である．

本章は，アプリ使用や端末設定など，実際のユーザによるスマートフォンの利用パターンがどのくらい存在し，それぞれのパターンの特徴を明らかにすることで，スマートフォンを前提にしたサービス，アプリ，ミドルウェア，オペレーティングシステム（OS）における新規提案や問題解決を目指した研究に対し，各々の課題設定や効果検証において現実的に考慮すべき前提条件や，評価条件となるデータ（以降，スマートフォン利用モデル）を提供することを目的とする．

我々は，モデル検討に必要なデータを獲得するため，調査モニタとして採用した実際のスマートフォンユーザに対し，各自のスマートフォン利用に関するアンケートと，ログ収集アプリを用いた約 1 ヶ月間の端末利用ログ収集による利用実態調査を実施した．アンケートにより，年齢・性別・居住地・職業などユーザの基本属性や主観的な情報を得ることができ，また，端末ログ収集により，アプリをはじめとする端末の利用履歴・設定，ディスプレイやバッテリなどの端末状態を把握できるため，サービス企画における対象ユーザの分析や，ソフトウェア品質改善での効果検証において前提とする端末の設定値を定めるなど，幅広いデータの活用が可能であると考える．

本章の構成は以下の通りである．4.3 では，関連研究を踏まえ，サービス企画や研究開発に幅広く活用可能なスマートフォン利用モデルとして満たすべき要件と全体像を示す．4.4 では，モデル作成のための利用実態調査の内容について説明し，調査で得られたデータの基本集計結果を示す．4.5 では，4.3 の検討を踏まえ，日毎のアプリカテゴリ毎の使用時間データを主なデータセットとしたクラスタリングにより，1 日のスマートフォン利用パターン

を導出する．また，クラスタ毎のユーザ属性などの特徴をさらに分析する．最後に，クラスタ毎に，クラスタ中心にもっとも近いサンプルをクラスタの代表的な 1 日の利用データとして選定し，スマートフォンの 1 日のユースケースを示す．以上の分析結果を本論文におけるスマートフォン利用モデルとし，本モデルが関連研究をはじめとする様々なサービス企画，研究開発における課題設定，効果検証において有益なデータであることを示す．

4.3 スマートフォン利用モデルの要件

本節では，スマートフォンを前提したサービスや技術の新規提案や問題解決に幅広く活用できるモデルを提案するため，モデルの活用対象となりうる事例やユーザの利用実態に関する分析事例について関連研究とともに議論し，モデルの要件を整理する．

4.3.1 関連研究

モデルの活用領域

新商品・サービスの商品化プロセスについて，中高齢者向け携帯電話の商品化を事例にした報告がある [12)]．筆者らは，旧来の商品化プロセスについて，マーケティングにはじまる企画プロセスと要素技術の研究開発プロセスが独立し，商品コンセプトや機能・品質要件のすりあわせが効率的に進まないことにより，商品化まで多大な時間を要し，結果的に顧客ニーズとはずれた商品化がなされてしまうという問題を指摘している．前述の携帯電話の事例では，携帯電話市場は若年層が中心であった 2001 年当時，研究開発部門もマーケティングに参画し，中高齢者向けを潜在市場として見いだし，徹底したニーズ分析と技術課題の深掘りを行うことで，商品化における生産性の向上と，新たな顧客開拓に成功した．近年のスマートフォン利用は，従来よりも自由度が高く，顧客ニーズも多様化していることから，対象ユーザに対する分析と明確化がさらに重要であると考える．

また，近年のサービス動向のひとつとして，スマートフォンで取得した位置情報を活用したサービスが多く提案されており，地図だけでなく現在地に

応じたクーポン配信など様々なサービスが提供されている [13,14]．一方，位置情報は，氏名・住所などと同様に，重要な個人情報としてプライバシー保護の対象であるため [15]，端末の位置情報取得は初期状態では無効化されている．この設定を有効化するには，ユーザによる設定変更が必要であり，アプリなどが自動的にこの設定を有効化することはできない．また，スマートフォンにおける位置情報取得機能は，取得時間の高速化や取得精度の向上のため，携帯電話基地局の情報や GPS 測位情報などを相補的に用いて位置取得を行う仕組みが採用されており，それぞれどの情報を用いるか設定が細分化されている．このため，なんらかのサービスを提供したとしても，各ユーザの設定状態に依存して，当初の想定したユーザ規模を下回ったり，サービスの品質低下をまねく恐れがあるため，このような端末設定の利用実態は事前に考慮しておくことが望ましい．

次に，近年のスマートフォンにおける要素技術に関連する先行研究について述べる．川崎ら [2] は，複数の Android 端末のアプリによる通信が広域的に同時発生し，ネットワーク側で輻輳を起こすという問題に対し，OS でのアプリ動作制御の改善を提案している．この問題は，アプリが端末状態の変化（ディスプレイの点灯など）に応じたタスク実行を可能にする仕組みに起因しており，前述の論文は原因となるアプリ挙動を模擬する評価用アプリ用いた実験により，改善効果を評価している．しかし，この挙動の発生はアプリの設計やユーザの端末操作に依存しているため，より実効的な効果を示すためには，評価端末にインストールされるアプリの組み合わせや，ディスプレイの点灯などの端末状態の変化頻度など，実際のユーザの使用状態を考慮した評価条件を設定することが望ましい．他の先行研究においても，有機 EL ディスプレイの省電力化のために表示コンテンツの表示色を変化させる手法 [16] や，GPU の省電力化のためにアプリの描画処理を推定し GPU を省電力状態に遷移させる手法 [17] などが提案されているが，同様に主要なアプリのみに限定した課題検討や評価にとどまっており，実効的な効果を示すには至っていない．特に，モバイル向けソフトウェア工学においては，実際のユーザの使用状況を考慮した前提条件の設定が重要であると指摘されている [18–20]．

ユーザ行動の理解

Ferreira ら [21] は，アプリのユーザビリティやユーザ体験の向上を開発者が検討するための知見として，ユーザによる 15 秒以下の短時間のアプリ使用に着目し，その発生要因となるユーザコンテキストとの関係を分析している．被験者端末でのログ収集とインタビューに基づき，40% のアプリ起動は短時間の使用であること，ユーザが自宅など一人でいるときに最もこのようなアプリ使用がなされていることを示している．また，Parate ら [22] は，アプリ利用時のユーザ体験を向上させるため，アプリ利用に伴うコンテンツ取得の遅延を抑えるためのプリフェッチを実現させるための要素技術として，ユーザが次に使用するであろうアプリとその起動タイミングを予測する手法を提案している．その予測手法を実現するため，系列データとしてアプリの起動履歴や起動間隔に着目した特徴量選定を行っている．これらの先行研究は，スマートフォン利用におけるユーザ行動傾向を理解するという観点で，アプリ利用に着目することの有効性を示す事例であり，本論文における分析でも，この考え方と同様にアプリ利用を主要な特徴量として採用している．しかし，これらの先行研究はそれぞれ特定の目的に特化したデータの分析となっているため，前述したユーザの利用実態を考慮した端末省電力化の効果検証などに活用できるデータとはなりにくい．

Falaki ら [23] は，2008〜2009 年における実際のスマートフォンユーザ 255 名の端末で収集したログを用いて，ユーザの利用実態を分析している．この調査では，アプリ利用以外にも，ディスプレイ状態，ネットワークトラフィック，バッテリ状態などに関するログを解析し，例えば一日のデータ送受信量など，様々な観点でユーザの利用傾向はユーザによって大きく異なることが示されている．しかし，この調査は現在の主要なスマートフォンが登場した初期に実施されたものであることから被験者選定に片寄りがあり，この分析結果はスマートフォンが広く一般に普及した現在の利用実態とは整合しないと考える [24]．また，1 日のアプリ毎の使用時間も示されているが，全ユーザで平均化されたものであり，ユーザによるスマートフォン利用の多様性や特徴を詳細に示したものではないため，本論文が目指すサービス

や技術の新規提案や問題解決に幅広く具体的に活用できるデータとしては分析粒度が粗いと考える．スマートフォンが広く普及し，非常に多種多様なアプリが提供されている今日，ユーザの利用実態を的確に把握するためには，アプリをはじめとする利用パターンをとらえることが重要である．

利用パターンの把握

スマートフォン利用の多様性をパターンに分類し，その特徴を分析した先行事例について述べる．Patil ら [25] は，実際のユーザ 33 名によるアプリ使用時における CPU/GPU 負荷の傾向をクラスタリングにより分析している．しかし，調査対象の被験者数が少なく，主要なアプリケーションの使用時に限定したデータ収集および分析しかなされておらず，分析結果の有用性を十分に示せていない．本論文における我々の分析結果は，前述のような調査設計に対しても有益な知見となることが期待される．また，Li ら [26] は，数百万人ものユーザのアプリ管理操作などのログからスマートフォン利用傾向を分析しているが，アプリマーケット全体の視点で，アプリ人気度やユーザのアプリ選択の特徴といった全体傾向を示すものであり，次項にて本論文が示すアプリの利用パターンのように，ユーザによる 1 日のアプリ使用を詳細に示すものではない．Zhao ら [27] は，本論文と同様のモチベーションである，実データに基づくユーザ理解と研究開発への応用のもとに，クラスタリングにより約 10 万ユーザのアプリ使用傾向からユーザを 382 パターンに分類している．しかし，Zhao らの分析は，1 時間毎に起動されたアプリ名のみが列挙された粗いサンプリングデータに基づいたものであるため，例えば，1 日にどのアプリを何分使用したかなど，詳細な考察を行うことはできない．また，382 もの非常に多くのパターン（クラスタ）を導出しているが，そのうちの特徴的な 6 パターンのみの傾向を説明するにとどまっているように，パターンが多すぎるゆえにパターン毎の特徴付けが困難になるだけでなく，分析結果を活用する際にも考慮すべきパターン数が多く大きな労力を要すると考える．サービス企画でも研究開発においても，特に初期検討時のように，検討対象となるユーザが絞り込めていない段階では，まずは全体傾向を俯瞰してとらえることができる粒度でパターン分けを行うこと重

要であり，必要に応じて細分化していくことが一般的である．

我々のスマートフォン利用モデルは，特に，前述の初期検討時に幅広く活用されることを目指している．また，マーケティングにおけるセグメンテーション分析 [28] のように，サービス/製品の対象セグメントの存在とボリュームを定量的に把握することと，技術課題・仮説/効果検証を行うことを一つのモデルで同時に実現することで，効率的に実際のユーザを考慮したサービス・製品企画や研究開発ができるようになると考える．

4.3.2 モデルの貢献と要件

前項の議論を踏まえ，本論文におけるスマートフォン利用モデルが目指す貢献とモデルの要件を整理する．

はじめに本モデルの貢献について述べる．本モデルは，スマートフォンに関連するサービス/製品企画や研究開発各々の領域における以下の課題に対し，ユーザ利用実態を考慮するための活用できるデータを提供することを目指す．

サービス/製品企画における課題

- ユーザセグメントの把握
- 既存サービスの利用実態の把握

研究開発における課題

- 技術課題設定における前提条件の妥当性と実効性の事前検証
- 利用可能な端末リソースの把握
- 実利用を考慮した評価端末のセットアップ

サービス/製品企画においては，前項で述べた通り，ユーザセグメントを定量的に把握できることは，サービス/製品の対象顧客の存在を示すことや見込まれる市場規模を計る上で重要である．また，スマートフォンアプリを通じたサービス利用の実態が把握できることは，競合する類似サービスとの差別化や自身のサービスの利用機会の有無を検討することにつながると考える．

研究開発においては，上述の企画と同様，課題解決により恩恵を受ける対

象ユーザの存在とそのスマートフォンの利用実態を知ることは，課題設定における背景や前提条件に妥当性を示すことと，実際に見込まれる効果の大きさを事前に計ることにつながる．また，アプリなどの使用を定量的に把握することは，言い換えると，バッテリなど現実的に利用可能な端末リソースがどの程度存在するか把握することであり，モバイルコンピューティングにおける研究開発ではこの制約を考慮することが特に重要である．さらに，本モデルは，スマートフォン実機を用いた実験を行う際に端末にセットアップすべき情報として，ユーザのアプリ使用や主要な端末設定を示すデータを提供することで，実利用を想定した評価を行うことに貢献する．

以上の議論より，本モデルの要件として，スマートフォン利用実態として着目するデータと，その分析に基づき導出されるモデルの構造を以下に示す．

R.1 対象データ

- ユーザ属性
  年齢，性別などのデモグラフィック情報
- 端末設定・状態
  ディスプレイ輝度などの端末設定・状態値
- アプリ使用
  アプリの使用頻度や時間

R.2 モデル構造

- 1 日の利用パターン分類
  パターン毎に *R.1* の情報を備える
- パターン数
  10 パターン以下

R.1 に示す通り，サービス・製品の企画・研究開発など幅広く活用するためには，モデルはユーザに依存するユーザ属性，端末設定・状態，アプリ使用の傾向を縦断的に表現すべきであると考える．

また，ユーザによって異なるスマートフォンの利用を明確に表現するため

には，*R.1* の各項について単純に平均化するのではなく，各項の組み合わせによる利用パターンがどのように存在するのかを明示する必要がある．さらに，同一のユーザであっても，平日や休日など利用シーンの違いから利用パターンが異なる可能性が考えられる．よって，*R.2* に示すように，本論文ではこの利用パターンを抽出するため，1 日のアプリ使用時間を主な説明変数としたクラスタリングにより，「1 日のスマートフォン利用パターン」を分類する．スマートフォンの場合，音声通話も含め全てアプリ単位で機能が提供されていることから，基本的にアプリ使用がスマートフォン利用を漏れなく表現する特徴量になると考えられる．

また，前項で述べたように，本論文では，サービス企画・研究開発における対象ユーザが絞り込めていない段階でのモデル活用に注目するため，数百もの多くのパターンを導出するのではなく，全体傾向を俯瞰しやすいよう 10 パターン以下に抑えたパターン抽出を目指す．ここで得られた利用パターン毎に，ユーザ属性やアプリ使用などの特徴を定量的に表現したものを，本論文におけるスマートフォン利用モデルとする．

本モデルを参照することで，前述したような製品/サービスの新規提案・改善検討においては，対象ユーザの範囲・規模と同時に，解決すべき技術課題も把握しやすくなる．また，要素技術の研究開発においても，本モデルが課題設定や評価の前提となる諸条件を定量的に示すことで，基本的な根拠データを揃える手間が軽減でき，さらなる課題の深掘りと妥当性のある評価を行うことが可能になると考える．

4.4 利用実態調査

本節では，前節で定義したモデルを作成するため，実際のユーザを対象に実施した利用実態調査の方法と基本集計結果を示す．

4.4.1 調査方法

本調査では，調査会社を通じ，主要な Android 端末 2 機種を使用する実際のユーザを調査モニタとして採用し，2013 年 10 月から約 2 ヶ月間にアンケート調査と端末ログ収集調査を実施した．調査の実施概要を表 4.1 に示す．調査モニタの選定にあたっては，未成年のユーザは調査会社のポリシーにより調査対象外としたが，それ以外に特別な条件を設けずに選定した．2013 年度の日本国内における AndroidOS 搭載スマートフォン出荷台数報告 [40] によると，対象機種のメーカ製端末は約 750 万台と全体の半数以上を占めている．この 750 万台を母数として必要な調査サンプル数を信頼度 95%，許容誤差 5% 以内で算出すると 385 であり，本調査ではそれを大きく上回るサンプル数を獲得できた．また，3.2.1 に後述するが，結果的に獲得した調査モニタの年齢別・性別ユーザ分布は他の調査報告結果 [41] とほぼ同様であり，前述の通り大規模なシェアを占める端末ユーザにおいて，一般性のある分析結果を得るためのデータとして充足していると考える．なお，調査にあたっては，調査内容およびデータ利用に関するユーザの許諾を得ており，氏名，住所，位置情報などの個人情報はデータ収集の対象外とした．

アンケート調査は，全ユーザ 694 名に対し，専用の Web サイトを通じて実施し，主に年齢や性別などのデモグラフィックデータと，後述の端末ログ収集では取得できない項目を収集した．

端末ログ収集調査では，アンケート回答者のなかからログ収集に同意したユーザ 391 名の端末に，専用のログ収集アプリをインストールし，主にアプリ使用（アプリ名と使用開始・終了時刻）などの端末使用ログを収集した．なお，ログ収集アプリは，ユーザの普段通りの端末使用に影響を与えないよう，バックグラウンド状態で動作し，バッテリなどの端末リソースを極力消費しないように設計されている．

表 4.1　調査の実施概要

対象機種	端末 **A:Galaxy S4**	端末 **B:Xperia A**
アンケート期間	2weeks from the end of Oct.2013	
回答者数	221	473
ログ収集期間	1month from the mid. of Nov.2013	
ログ収集対象者数	112	279
ログ収集量（人日）	3136	7812

4.4.2　基本集計結果

ユーザ属性

アンケートで得たデモグラフィックデータの集計結果として，年齢，性別，居住地，職業の分布をそれぞれ図 4.1〜4.4 に示す．ほぼ全ての年代，性別，居住地，職業をカバーしており，片寄りなく調査モニタを採用できている．年齢分布は端末 A のほうが年代層がやや低めであるが，両機種による大きな違いは見られていない．

また，今回の対象ユーザは，その約 95% が調査端末のみを単体使用しているユーザであった（図 4.5）．

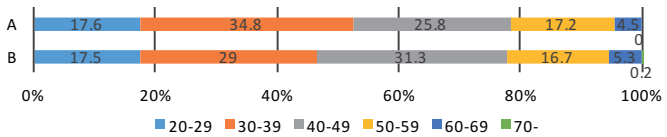

図 4.1　年齢別ユーザ分布

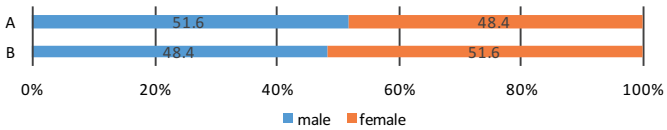

図 4.2　性別ユーザ分布

その他

ユーザ属性以外のアンケート調査項目の集計結果を以下に示す．

ホームアプリの使用状況（図 4.6）は，両機種とも大半以上のユーザがプリインストールされたものを使用しているが，端末 A の場合はそれ以外のホームアプリも 3 割強と比較的多くのユーザが存在することがわかった．なお，ホームアプリとは，各アプリ起動のショートカットアイコンなどが配置されるホーム画面を表示するためのアプリである．

次に，画面ロック解除方法の使用状況を図 4.7 に示す．使用率が高い順にスワイプ/タッチ（Swipe/Touch）とパターン入力（Pattern）で 7 割，ロックなし（None）が 2 割を占めている．

アプリの定期アップデート設定（図 4.8）は，端末メーカ独自のプリインストールアプリを対象にした定期的な自動アップデートの有効化設定である．

現在地情報へのアクセス設定（図 4.9）は，アプリに対する位置情報取得

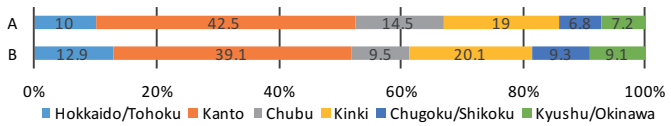

図 4.3　居住地別ユーザ分布

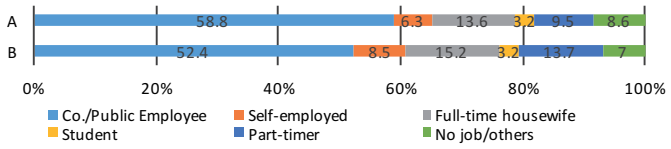

図 4.4　職業別ユーザ分布

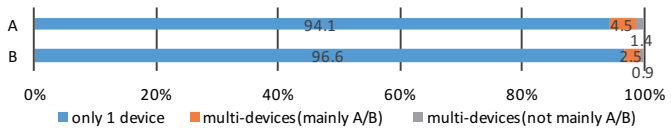

図 4.5　一人あたりの端末所有台数

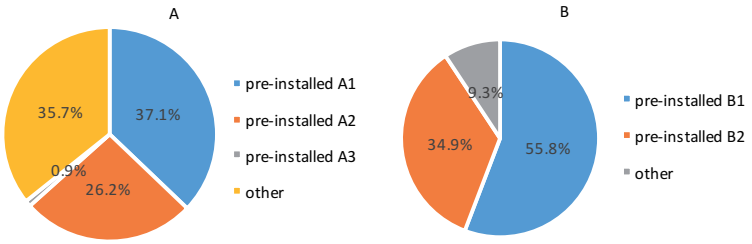

図 4.6　ホームアプリの使用状況

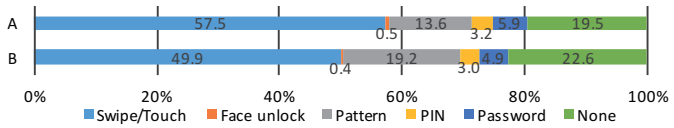

図 4.7　画面ロック解除方法の使用状況

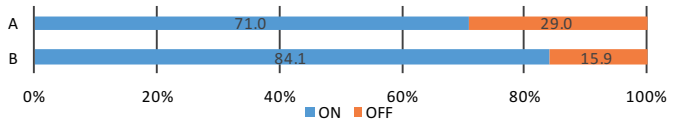

図 4.8　定期アップデート設定

機能の許可設定であり，両機種とも 6 割前後のユーザが許可している．また，無線ネットワークによる位置情報取得設定（図 4.10）は，これを有効にすることで位置情報取得時に Wi-Fi やモバイルネットワークの情報を使用した位置測位が可能になる [42)][16]．両機種とも約 6 割のユーザがこの設定を有効化している結果となった．一方，GPS 設定の利用状況（図 4.11）については，常に ON 状態で使用しているユーザは全体の 25〜30% 程度であり，50% 程度のユーザは必要に応じて ON/OFF を切り替えていることがわかった．前述したように，位置情報関連の設定はプライバシー保護の観点からアプリなどによる自動的な設定変更ができないため，利用状況を考慮することが重要である．近年，ユーザの位置情報に基づくサービスが多く提案されているが，これらの設定の利用状況を考慮せずにサービスを提供すると，例えば，全体の 3 割程度のユーザしか獲得できない可能性がある．また，GPS

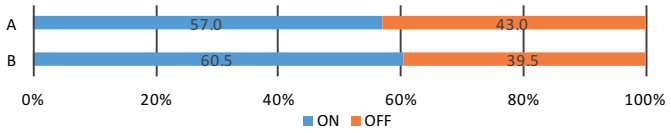

図 4.9　現在地情報へのアクセス設定

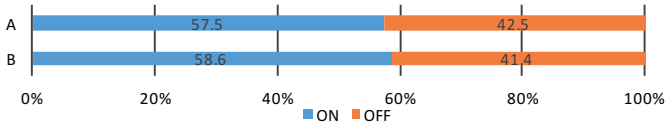

図 4.10　無線ネットワークによる位置情報取得設定

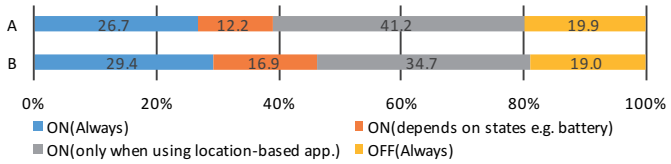

図 4.11　GPS 設定の利用状況

の利用状況によっては，想定していた精度で位置情報が取得できないことでサービスの品質低下をまねく恐れもある．

アプリ使用状況

ログ収集アプリにより得られたアプリ使用ログから，以下の集計結果を示す．なお，アプリ使用ログからは 3339 件のアプリが確認された．

- 各アプリの総使用時間
- 1 日あたりの各アプリ総使用時間

ここでのアプリ使用とは，アプリが画面表示を伴う，フォアグラウンド状態での使用を指し，画面消灯状態は除外している．また，端末の充放電状態を示すログから，充電状態によるアプリ使用傾向の違いも考察する．

なお，基本集計や後述の分析の結果から端末 A，B で大きな違いは見られなかったことから，以降，サンプル数の多い端末 B のみ対象に記載する．

全端末 B ユーザによる，放電時・充電時それぞれの各アプリの総使用時間（上位 20 件）を表 4.2，4.3 に示す．表中の UU はユニークユーザ数である．

放電時・充電時ともブラウザの使用時間が最も長く，次いでホーム画面，メッセンジャー，メールが他のアプリよりも突出して使用時間が長い．また，コミュニケーション手段として，メール，電話よりもメッセンジャーアプリほうが使用時間が長いことが示された．さらに，ゲームが上位 20 アプリのうち 5 アプリを占めており，各々 UU 数は 50 以下と少ないものの，総使用時間は第 6 位以下の他のアプリと大きな差は見られないことから，1 ユーザあたりの使用時間が長いことがわかる．

充電時の使用傾向としては，表 4.3 の※の通り，放電時と比較すると，音声・動画再生を行うアプリや RPG ゲームなど特に使用時間が長いアプリがよく使用されている．また，UU 数が一桁代のアプリも多く含まれることから，特定のアプリにおいて少数のヘビーユーザが存在していることがうかがえる．

表 4.2　各アプリの総使用時間（放電時）

No.	アプリ	**UU**	総使用時間 **hh:mm:ss**
1	Preinstalled browser	267	4191:12:43
2	Preinstalled home 1	164	1216:35:23
3	Messenger	202	1159:40:30
4	Mail	278	1133:54:54
5	Preinstall home 2	123	789:46:01
6	Puzzle game 1	43	479:08:16
7	Browser 1	150	470:59:14
8	Web portal	108	453:19:06
9	SNS1	151	402:41:21
10	Phone call	276	354:17:55
11	Puzzle game 2	26	343:02:26
12	Twitter client	61	283:11:13
13	SNS2	15	226:59:36
14	Puzzle game 3	42	218:36:15
15	Puzzle game 4	29	195:43:32
16	Puzzle game 5	38	192:00:07
17	Online video viewer	175	190:50:48
18	Photo album	247	136:39:56
19	Puzzle game 6	3	112:16:05
20	Textboard viewer	14	108:40:41

表 4.3　各アプリの総使用時間（充電時）

No.	アプリ	**UU**	総使用時間 **hh:mm:ss**
1	Preinstalled Browser	248	957:39:52
2	Preinstalled Home 1	162	805:58:07
3	Preinstalled Home 2	116	663:53:53
4	Messenger	193	268:01:01
5	Mail	271	211:53:56
6	Slideshow *	9	208:39:57
7	Browser 1	93	134:34:15
8	Video viewer 1 *	7	115:39:28
9	SNS2	11	104:15:44
10	Web portal	91	96:23:57
11	RPG game	1	95:04:22
12	Media player *	38	93:10:21
13	Exchange trading *	1	91:57:42
14	Screensaver *	1	90:58:01
15	SNS1	105	85:03:34
16	Twitter client 2 *	1	81:49:43
17	Puzzle game 1	41	81:25:59
18	Alarm clock *	172	68:10:10
19	Online video viewer	87	66:03:47
20	Twitter client 1	53	65:03:29

次に，UU 数上位 25 件の代表的なアプリについて，1 人 1 日あたりの総使用時間を表 4.4 に示す．この 25 件のアプリは，1 人 1 日平均 1 分 30 秒以上の利用時間があり，UU 数が 100 人以上のものを代表的なアプリとして選定した．

この観点で列挙したアプリ 25 件をみると，第 10，17，25 位の 3 アプリ以外の全てのアプリが端末の初期状態からプリインストールされているものであった．

表 4.4　1 人 1 日あたりの各アプリ平均使用時間

No.	アプリ	**UU**	**1 人 1 日の** 平均使用時間	中央値
1	Mail	278	00:11:05	00:04:25
2	Phone call	277	00:04:54	00:01:29
3	Google Play	276	00:02:26	00:00:49
4	Camera	273	00:01:57	00:00:37
5	Configurations	273	00:02:07	00:00:26
6	Preinstalled browser	269	00:48:34	00:21:38
7	Photo album	248	00:03:08	00:01:17
8	Web portal 2	245	00:02:05	00:00:39
9	Alarm clock	206	00:02:16	00:00:31
10	Messenger	203	00:18:29	00:09:18
11	Schedule	193	00:02:21	00:00:44
12	Map	191	00:06:46	00:02:15
13	Online video viewer	183	00:16:40	00:05:14
14	Preinstalled home 1	167	00:27:42	00:13:13
15	Calculator	156	00:03:16	00:00:56
16	Browser 1	153	00:22:04	00:07:48
17	SNS1	151	00:12:08	00:06:09
18	Auto. App. Update	148	00:01:54	00:00:22
19	Mail 2	143	00:04:05	00:01:48
20	Video viewer 2	127	00:04:07	00:00:54
21	Web search	124	00:02:21	00:01:05
22	Preinstalled home 2	123	00:28:06	00:12:17
23	Music player 1	119	00:03:33	00:01:02
24	Media player	112	00:23:24	00:03:12
25	Web portal 1	109	00:17:35	00:05:43

アプリカテゴリ別使用状況

前項に示した通り，アプリの使用状況には UU 数が多い代表的なアプリの使用だけでなく，一部のユーザの使用ではあるが使用時間が長い特定アプリの使用も確認されたことから，これらの組み合わせによるユーザの利用パターンにも多様性が存在しうると考えられる．しかし，今回のアプリ使用ログからは 3339 件ものアプリが確認されており，個別のアプリ単位で利用パターンを分析することは困難である．そこで，本項では，全体を俯瞰した

傾向をとらえるため，アプリをカテゴリ分類し，カテゴリ毎の使用時間を導出する．アプリのカテゴリ分けの基準として Google 社のアプリストアである Google Play ストア[*1]でのアプリ分類を用いる．分類手順は以下の通りである．

1) Google Play のカテゴリに基づき分類
2) 利用ユーザが多く，使用時間が長いアプリを単独カテゴリとする
3) 特定のユーザが長時間利用している未分類アプリを既存カテゴリに分類する

なお，1）では，Google Play に登録されているカテゴリ全 30 種類を採用した．アプリ使用ログに記録されないウィジェット・ライブ壁紙カテゴリは除外し，ゲームカテゴリについては，6 種類に細分化されたカテゴリを採用した．なお，このカテゴリは 2014 年 1 月 10 日時点のものである．

2）では，UU 数が多く使用時間の長いアプリは，カテゴリ別に集計する際に同一カテゴリに与える影響が大きく，集計結果を歪める可能性があるため，単独カテゴリと見なすことにした．結果，このようなアプリが 85 件存在し，似た用途のアプリは統合し，26 の単独カテゴリを追加した．

3）の未分類アプリは，2）と同じ理由により，24 件のアプリを既存カテゴリに手動で分類した．

以上により，未分類含め定義した 57 種類のアプリカテゴリを表 4.5 に示す．11 行目以下のカテゴリは 2）で独自に作成した単独カテゴリである．

上記のアプリカテゴリに基づき集計したカテゴリ毎の総使用時間を図 4.12 に示す．表 4.2 では，総使用時間が極めて長い，あるいは UU 数が極めて多いアプリが上位を占め，アプリ種別としては片寄りが見えていたが，カテゴリ集約を行うことで，ニュース・雑誌やショッピングなど様々なカテゴリの使用傾向を示すことができた．

[*1] https://play.google.com/

表 4.5　アプリカテゴリの定義

Application Categories		
Lifestyle	Personalization	Productivity
Weather	Photography	Game_Music
Entertainment	Tools	Communication
Social	Comics	Game_Casual
Game_Sports	Media_Video	Books_Refference
Finance	Game_Casino	Travel_Local
Game_Arcade_Action	Shopping	Transportation
Game_Puzzle	Sports	Game_Racing
Libraries_Demo	Business	Education
Health_Fitness	News_Magazines	Medical
Camera	Map	Phone call
Google Play	Mail	Calculator
Online video viewer	Web portal 1	Schedule
System	Browser	Agent 1
Preinstalled home	Configuration	Calendar
Data Backup	Music Player 1	Virus scan
Web portal 2	Clock	Agent 2
SNS1	MMS	Restaurant review
Messenger	Home	The others

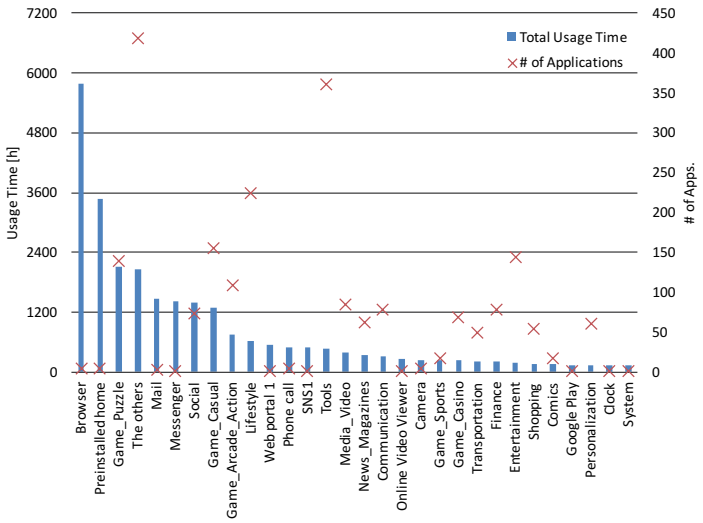

図 4.12　アプリカテゴリ毎の総使用時間（上位 30 件）

4.5　モデル分析

前節ではアプリ使用の全体傾向を把握するためにアプリをカテゴリ分類し，カテゴリ毎の使用時間をを示した．本節では，様々なアプリおよびカテゴリの組み合わせとなるユーザの利用パターンをクラスタリングにより導出する．導出された利用パターンの特徴を説明することで，本論文におけるスマートフォン利用モデルとして示す．

モデル化の対象は「ユーザを区別しない 1 日のスマートフォン利用」とする．一般的に，ユーザの生活サイクルは 1 日毎であり，同一ユーザであっても平日・休日で生活パターンが異なり，利用パターンも変わる可能性があるためである．後者の仮説は利用パターンの分類をした後に別途考察する．

クラスタリングに基づくモデル分析結果の妥当性については，1) モデル要件を満たしていること，2) クラスタ所属メンバのまとまりの良いクラスタ数を決定できること，3) 各クラスタの特徴を明確に説明できることの 3 点を，後述する分析結果をうけた考察により評価する．

4.5.1 アプリカテゴリに基づく 1 日の利用パターン分類

変数選択

1 日の利用パターン（ユーザ×日）を分類するため，日毎に以下の変数データを用意した．この特徴量選択はモデル要件 *R.1* に示した通り，利用パターンの傾向として示すべき情報であることから選定している．なお，レコード数は分析対象日数 7784 人日分である．

- アプリカテゴリ毎の使用時間（56 変数）
- 画面 OFF 状態での端末稼働時間
- ディスプレイ輝度

アプリカテゴリ毎の使用時間は，前章で定義したカテゴリから未分類カテゴリを除外した 56 カテゴリを変数とする．なお，そのままの値を用いるとユーザや利用日によって飛び値が存在し，全体的に値域が大きくなっているため，極端な値が強調されないように対数変換で飛び値を緩和した上で正規化する．

また，アプリカテゴリ毎の使用時間に加え，ログ収集アプリにより計測した画面 OFF 状態での端末稼働時間，ディスプレイ輝度を変数に加える．この 2 変数を加えた理由は以下の通りである．ユーザが直感的に利用していると認識するのは画面点灯 (ON) 状態におけるアプリ使用であるが，実際のアプリは画面 OFF 状態でもバックグラウンドで動作することがある．このため端末リソース利用，特にバッテリ消費への影響を検討するためにはこの動作時間を考慮することが必要である．また，ディスプレイ輝度もアプリ毎に任意の値を設定可能であり，画面構成や表示コンテンツにあわせて随時異なる値をとることから，端末リソース利用の観点で同様に考慮されるべきである．今回，アプリの使用時間をアプリカテゴリ毎に集約した値を主な変数としたことで，上記の情報が欠けてしまうことから，補完するためにこの 2 変数を追加する．なお，この 2 変数データは，ほぼ正規分布で値域も大きくないことから，対数変換せずに正規化する．

クラスタ分類

前述の変数データを用いたクラスタリング結果について述べる．今回，クラスタリング手法は k-means 法 [43] を用い，クラスタ数を 4〜10 個の 7 パターンのクラスタリングを実施する．各パターンのクラスタリングによるクラスタ毎の所属人日数の構成比は表 4.6 の通りである．

どのクラスタ数での分類が最も適切か一概に決定することは難しいが，ここではクラスタ毎の構成比が極端に大きいまたは小さいクラスタが存在しない程度にできるだけ分割するという考え方で，クラスタ数 5 と 6 に着目する．

次に，クラスタ数 5 か 6 どちらが適切か決定するため，クラスタ数を 5 から 6 に変更した場合のサンプルの所属クラスタの移行割合を確認する．表 4.7 に赤字で示す通り，変更前のクラスタ 1 から変更後のクラスタ 1，6 に分割される以外，その他のクラスタに大きな移行は見られない．

さらに，表 4.8 に示す通り，クラスタ数を 6 から 7 に変更した場合の移行割合を見ると，変更前の全てのクラスタから少しずつ移行するかたちで小さなクラスタ 7 が作成されており，既にクラスタ数 6 でクラスタが固定化されていることがわかる．

以上により，クラスタのまとまりの良さの観点ではクラスタ数は 6 が適切であり，かつ，モデル要件で定めた 10 クラスタ以下での利用パターン分類を行うことができた．以降，6 つのスマートフォン利用パターンがあるものとして，それぞれの特徴を分析する．

表 4.6　クラスタ数別構成比

Composition Ratio(%)		The number of clusters(k)						
		4	5	6	7	8	9	10
Cluster No.	1	32.2	30.1	22.9	21.9	19.3	16.9	16.3
	2	24.2	21.1	18.7	17.9	16.2	15.4	15.0
	3	24.2	17.6	17.3	16.7	15.7	12.8	11.3
	4	19.5	15.9	15.1	14.7	13.6	11.9	11.1
	5		15.2	14.6	14.4	12.3	11.3	10.3
	6			11.4	10.8	10.2	10.3	9.4
	7				3.7	9.0	9.6	8.0
	8					3.7	8.0	7.5
	9						3.7	7.4
	10							3.7

表 4.7　クラスタ数 5 → 6 変更時の移行割合

Transfer Rate(%)		Source Cluster No. (k=5)				
		1	2	3	4	5
Dest. Cluster No. (k=6)	1	69	5	1	4	1
	2	0	88	0	0	1
	3	0	0	98	0	0
	4	0	0	0	94	0
	5	0	0	0	0	95
	6	31	7	1	1	2

表 4.8　クラスタ数 6 → 7 変更時の移行割合

Transfer Rate(%)		Source Cluster No. (k=6)					
		1	2	3	4	5	6
Dest. Cluster No. (k=7)	1	95	1	0	0	0	0
	2	0	93	0	0	0	4
	3	0	0	96	0	0	0
	4	0	0	1	96	0	0
	5	0	0	0	0	96	1
	6	0	0	0	0	0	93
	7	4	6	2	2	3	2

4.5.2　クラスタ毎の特徴分析

端末使用傾向からみたクラスタの特徴

各クラスタに所属するの利用日のアプリカテゴリ毎の使用時間，ディスプレイ輝度，電池消費量の平均値を表 4.9 に示す．なお，アプリカテゴリ毎に赤字または青字で示している箇所は，同一カテゴリにおける使用時間が相対的に他のクラスタよりも長い，または短いものである．

最もアプリ使用時間が短いライトユースなクラスタは，891 日分類されているクラスタ 6 であり，1 日の平均アプリ使用時間は 1 時間 20 分程度である．一方，使用時間が長いヘビーユースなクラスタは，クラスタ 5，次いでクラスタ 4 であり，1 日平均 4〜5 時間程度アプリを使用している．ディスプレイ輝度が最も高いのはクラスタ 3 であった．このクラスタは 1 日の使用時間はクラスタ 4 に次ぐ長さであるが，パズルゲームの使用が特徴的であることから，このアプリカテゴリがディスプレイ輝度を押し上げる要因になった可能性がある．電池消費量が最も多いのはクラスタ 5，最も少ないのはクラスタ 6 であり，前述のヘビーユース，ライトユースの違いがあらわれた結果だと考える．

アプリカテゴリはじめ各クラスタの特徴を要約したものを表 4.10 に示す．

クラスタ 1 は，1 日のアプリ使用時間はヘビーとライトの中間，ミドル

ユースであるが，強いていえばツールカテゴリの使用が長い．このクラスタで特徴的なのは，全体集計において UU 数，使用時間ともに上位に位置するメッセンジャーや SNS1 の個別アプリの使用がほとんどないことがあげられる．

クラスタ 2 は，1 日の使用時間からはクラスタ 1 同様ミドルユースであるが，アプリカテゴリからみると対照的にブラウザやメッセンジャー，SNS1 など使用が特徴的である．

クラスタ 3 は，前述した通り，パズルやスポーツなどゲーム使用が際立っており，ディスプレイ輝度が高い．

クラスタ 4，クラスタ 5 はともにヘビーユースな特徴をもつクラスタであるが，アプリカテゴリ毎にみると，クラスタ 4 はホーム，メール，SNS カテゴリの使用，クラスタ 5 はカジュアル，ゲーム，通信カテゴリの使用が特徴的である．また，クラスタ 5 の電池使用量は際だって多いことから，アプリカテゴリの違いが端末リソース消費に影響を与えているものと思われる．

クラスタ 6 は前述した通り，最もライトユースなクラスタであり，強いて言えばコミックカテゴリの使用が相対的に長い傾向にあった．

以上より，アプリ使用をはじめとする端末使用傾向からクラスタ毎の特徴を明確に示すことができた．

表 4.9: クラスタ毎の特徴 - 正規化前の各変数平均値

Cluster No.	1	2	3	4	5	6
# of days	1779	1457	1346	1174	1137	891
Total app use time	135.98	188.20	218.58	241.16	291.35	78.17
Display brightness	116.72	126.60	135.22	124.74	118.14	110.70
Battery Usage(%/h)	2.52	3.07	3.56	3.76	4.30	1.91
Browser	52.28	58.16	42.11	51.50	43.74	0.31
Preinstalled home	19.34	32.65	24.59	40.46	28.26	15.33
Messenger	0.16	23.32	15.55	10.55	13.39	3.20
Mail	9.87	11.55	12.70	16.81	11.31	5.50
Web portal 1	1.68	2.54	6.45	7.64	4.33	4.19
SNS1	1.96	6.48	2.12	5.60	5.27	1.05

次ページに続く

前ページからの続き

Cluster No.	1	2	3	4	5	6
Phone call	2.48	5.81	3.65	3.92	4.03	3.88
Clock	0.69	0.91	1.30	0.93	1.95	0.93
Online Video Viewer	1.43	2.11	1.25	3.42	3.19	0.53
Camera	1.12	2.88	1.90	3.06	1.52	0.79
Home	1.16	1.51	0.65	0.34	0.87	0.32
Google Play	1.09	0.76	0.86	1.27	1.63	1.51
System	0.73	0.87	1.12	1.86	1.67	0.32
Configuration	0.43	0.45	0.37	0.66	2.84	0.25
Map	0.66	0.62	0.41	1.38	0.27	0.09
Schedule	0.36	0.29	0.53	0.21	0.21	0.65
Web portal 2	0.32	0.29	0.16	0.31	0.21	0.56
Calculator	0.53	0.25	0.17	0.20	0.14	0.21
Virus scan	0.16	0.34	0.08	0.35	0.13	0.05
Music Player 1	0.22	0.24	0.21	0.53	0.15	0.31
Restaurant portal	0.13	0.30	0.18	0.16	0.25	0.07
Agent 1	0.14	0.05	0.13	0.27	0.47	0.07
Agent 2	0.07	0.03	0.07	0.27	0.31	0.08
Calendar	0.02	0.13	0.04	0.04	0.01	0.01
MMS	0.00	0.00	0.00	0.00	0.00	0.00
Data Backup	0.05	0.04	0.02	0.02	0.02	0.01
Game Puzzle	0.08	0.04	64.81	1.39	31.93	0.39
Game Casual	0.16	0.21	0.13	0.87	64.08	2.74
Social	0.37	0.15	5.02	48.95	18.34	0.69
Game Arcade Action	5.82	3.90	7.13	2.76	14.07	0.66
Lifestyle	5.98	4.60	3.08	6.49	3.89	3.49
Communications	1.73	1.05	2.15	1.10	7.14	2.77
Game Casino	0.97	0.53	1.11	3.34	2.97	2.56
Tools	6.10	4.76	0.90	3.10	2.41	2.48
News Magazines	2.82	4.52	0.58	3.82	2.90	0.74
Productivity	0.56	0.90	0.40	0.96	1.04	0.62
Media Video	1.93	4.66	1.17	4.97	3.66	2.04
Entertainment	2.23	2.13	1.01	2.24	1.79	0.40
Finance	3.51	0.42	0.78	0.28	1.09	2.94

次ページに続く

前ページからの続き

Cluster No.	1	2	3	4	5	6
Game Sports	0.83	0.05	5.24	0.70	2.76	1.23
Transportation	1.73	2.19	1.99	0.83	2.01	1.06
Comics	0.52	1.80	0.31	0.04	0.32	6.14
Shopping	1.13	0.95	1.73	2.75	0.85	0.39
Personalization	0.25	0.80	2.58	0.93	0.02	2.88
Health Fitness	0.42	0.32	0.39	1.09	1.91	1.02
Books Reference	0.38	0.32	0.21	0.66	0.49	1.68
Photography	0.10	0.20	0.35	1.16	0.76	0.04
Education	0.10	0.14	0.18	0.14	0.16	0.47
Game Music	0.27	0.24	0.21	0.17	0.21	0.09
Business	0.09	0.43	0.26	0.17	0.15	0.11
Travel Local	0.19	0.20	0.10	0.21	0.05	0.18
Weather	0.09	0.08	0.11	0.19	0.18	0.06
Sports	0.38	0.03	0.02	0.05	0.00	0.07
Game Racing	0.14	0.00	0.00	0.04	0.00	0.02
Medical	0.00	0.00	0.04	0.00	0.02	0.00
Libraries Demo	0.00	0.00	0.00	0.00	0.00	0.00

以上

表 4.10　各クラスタの特徴 - 要約

Cluster	Main features of app usage	Pattern name
C1	Tools (Little messenger)	Mid. use but no-Messenger
C2	**Browser**, **Messenger** SNS1, Phone_call	Mid. use & Messenger
C3	**Game_Puzzle** Game_Sports	Puzzle game
C4	**Home**, **Mail**, **Social** Web_portal_1, Lifestyle	Non-verbal communication
C5	**Game_Casual**, **Game_Arcade/Action**, **Communication**	Heavy use
C6	**Comics**	Light use

クラスタ毎のユーザ属性

ユーザデモグラフィックデータとのクロス集計により，各クラスタのユーザ属性の傾向を示す．前述したように，このクラスタリング結果は利用日の分類であるため，クロス集計では利用日毎に該当するユーザの属性値を集計する．

クラスタ毎の年齢属性図 4.13 に示す．クラスタ 1，6 では 40 歳以上のユーザによる利用日が約 7 割を占め，クラスタ 2，3，4，5 では 39 歳以下のユーザによる利用日が半数以上を占めており，クラスタによって年齢層に片寄りがあることがわかる．

クラスタ毎の性別属性を図 4.14 に示す．クラスタ 1，6 では男性が，クラスタ 2，3，4，5 では女性が，それぞれ半数以上を占めており，性別割合もクラスタによって異なる傾向がみられた．なお，全体集計で図 2 に示した通り，端末 B ユーザの男女比はほぼ同じである．

クラスタ毎の地域属性を図 4.15 に示す．クラスタ 2 において，近畿地方の割合が他のクラスタよりもやや高いが，全体的にどのクラスタも同様の分布であり，際立った特徴の違いはみられない．

クラスタ毎の職業属性を図 4.16 に示す．地域属性と同様，クラスタ毎の特徴に大きな違いは見られないが，年齢属性で示したように，クラスタ 2，3，4，5 は若年層の比率が高いことから，学生の比率がクラスタ 1，6 よりも高い．

同一ユーザにおける利用パターンの多様性

4 章冒頭に前述した通り，スマートフォン利用は，同一ユーザであっても平日・休日など生活パターンに依存して異なる可能性があると仮定し，クラスタリングはユーザを区別せず 1 日単位のスマートフォン利用パターンを分類した．この仮説を検証するため，前述のクラスタリング結果における同一ユーザのもつ利用パターンの数，つまり同一ユーザの所属クラスタ数を確認する．図 4.17 は，所属クラスタ数ごとのユーザ数の分布である．約 1 ヶ月間のログ収集期間における所属クラスタ数のカウント条件として，同一ユー

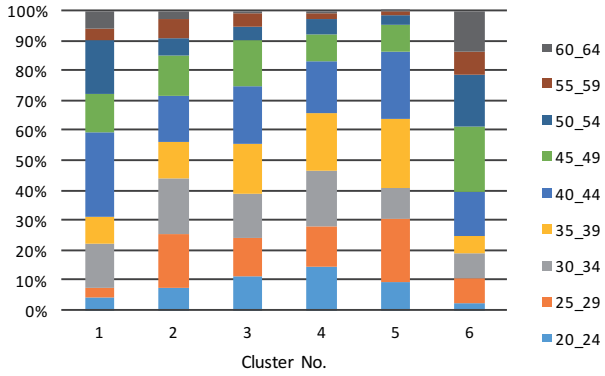

図 4.13　クラスタ毎の年齢属性

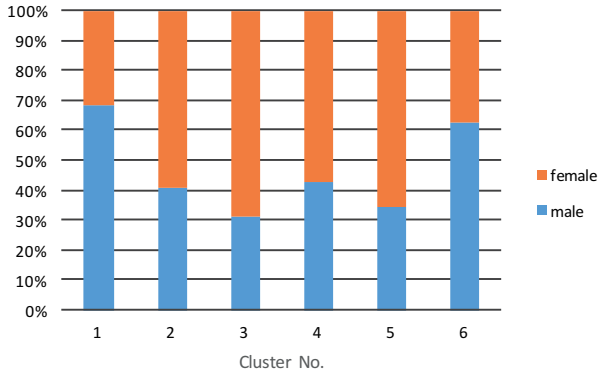

図 4.14　クラスタ毎の性別属性

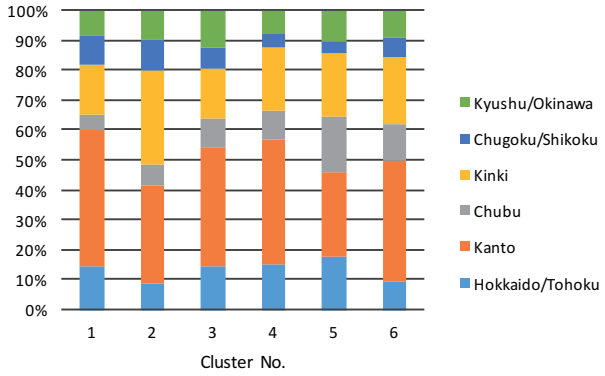

図 4.15　クラスタ毎の地域属性

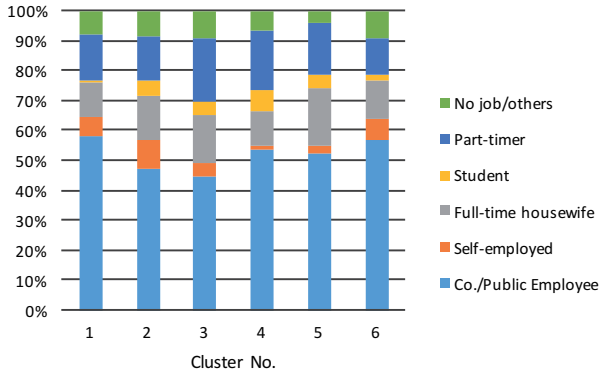

図 4.16　クラスタ毎の職業属性

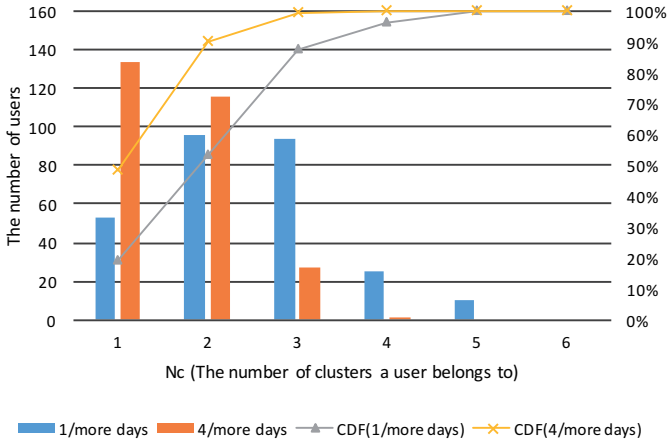

図 4.17　所属クラスタ数毎のユーザ数分布

ザの利用日がクラスタに 1 日以上所属している場合と，4 日以上所属している場合の 2 パターンの条件で集計した．

1 日以上の分布から，全体の約 9 割のユーザの利用パターン数は 3 以下程度であることがわかる．

しかし，1 日以上の分布には，約 1 ヶ月間のログ収集期間において数日しか存在しない，例外的な利用パターンも含まれると考える．そこで，主要な利用パターンだけの傾向を確認するため，4 日以上の利用日があった場合を主要なものとしてみなすと，全体の 9 割のユーザにおいてパターン数は 2 以下であることがわかる．

表 4.11　同一ユーザにおける主要な利用パターンの組み合わせ

No	Combo[1]	UU[2]	Common feature	Difference
1	C1&C2	24	Mid.-use	<Messenger> C1:rarely, C2:often
2	C1&C6	22	Light-use	<Browser> C1:1hour, C6:1min
3	C1&C4	14	Browser: 50min	C4: long usage of Mail C4: 100min longer total usage time
4	C3&C5	10	Heavy-use Long usage of PuzzleGame	C5: long usage of the other kinds of Games
5	C2&C3	7	Mid.-use	C2: long usage of Browser&Messenger C3: long usage of Puzzle game

[1]Combination of usage patterns(cluster #s)
[2]The number of unique users

次に，主要な利用パターンを複数もつユーザには，どのような利用パターンの組み合わせが存在するか確認する．表 4.11 は，組み合わせの出現回数が多い上位 5 つについて，組み合わせを構成する利用パターン同士の特徴を示したものである．例えば，第 1 位のクラスタ 1 とクラスタ 2 の組み合わせにおいては，ともに 1 日のアプリ使用時間が同等のミドルユース日であるが，SNS アプリの使用の有無で異なるなど，部分的に類似性のある利用パターン同士の組み合わせが多いことがわかる．

以上により，ユーザの日常的な生活において，日によって異なる複数の利用パターンが一定数存在することが示された．

4.5.3　クラスタ毎のユースケース

表 4.9 に示したデータは，アプリカテゴリなど各項目を平均化した値，つまりクラスタ中心の特徴を示したものであり，同一クラスタであっても利用日によってはアプリカテゴリの組み合わせや使用時間が異なるため，そのままで実際の 1 日のスマートフォン利用を示すものではないと考える．このため，クラスタ中心に最も近い代表日を選定し，クラスタ毎のユースケースとして付録 A.1〜6 に示す．このデータを参照することで，2.2 に前述したように，実機を用いてなんらかの提案手法の効果を検証する際，端末のセットアップでインストールすべきアプリを選定するための情報として活用でき

る．なお，表中，同一ユースケース内に同じアプリカテゴリが複数存在する場合は，該カテゴリ内の別のアプリが使用されたことを示している．

4.5.4 議論

本モデル検討は，スマートフォン利用を前提にした様々なサービス企画や，研究開発における課題設定，効果検証に対して，現実的に考慮すべき前提条件や評価条件となる有益なデータを提供することを目的としている．本節では，本モデルの有用性，特徴量選択によるさらなる分析の可能性，本モデルの制約について，それぞれ議論する．

モデルの有用性

本項では，各々の役割においてどのように本モデルを活用できるか例示し，モデルの有用性を議論する．

はじめに，サービス企画における活用例を議論する．本モデルは年齢，性別などのユーザ属性とその規模を定量的に示したものであるため，例えば新規サービスの企画においては，サービスの対象ユーザ層を把握するためのセグメンテーション分析が可能になり，見込まれる市場規模を推定する根拠データとして利用可能である．さらに，セグメントと紐付いた利用パターンの傾向を参照することで，対象ユーザ像やサービスの利用シーンを想像し定義されるペルソナ [44] の構築を容易にすることができる．また，アプリを通じたサービスの提供を検討する際には，自身のサービスに対するユーザの接触機会が十分に得られるか熟慮することが重要である．本モデルが示すアプリなどの利用パターンは，類似または関連サービスの利用頻度・時間だけでなく，ユーザの趣味嗜好も読み取れるものであるから，対象ユーザの時間的な利用機会の有無と，サービスの受容性も検討することができる．

次に，研究開発における活用例を議論する．本モデルでは，付録のユースケースのように，1 日単位でのスマートフォン利用を総合的かつ詳細に示しており，使用するアプリとその利用頻度・時間や端末の諸設定などから，ユーザ利用に起因する端末挙動を再現しやすいデータである．このため，例えば関連研究で述べたようなスマートフォンの OS やミドルウェアレベルに

おける技術課題への取り組み [2,16,17] において，本データを前提に，アプリのインストールなどの検証用端末のセットアップを行うことで，より実効的な仮説検証や効果検証を行うことができると考える．我々の活用例の一つとしては，本モデル検討の結果に基づき，スマートフォン各機種のカタログに掲載する電池持ち時間を評価するためのシナリオとして採用[*2]された [45]．フィーチャーフォン時代の評価では，通話・ブラウザといった代表的なアプリ単体での使用可能時間を示すのみであったが，1 日を通した利用を考慮した評価ができたことで，端末購入を検討するユーザにわかりやすい情報を提示するとともに，端末開発においても妥当な根拠に基づく目標値を設定することができた．

本モデルは，実際のユーザによるスマートフォン利用の特徴を，デモグラフィック属性と端末利用パターンを同時に定量的に示すものであるから，サービス企画，研究開発いずれの取り組みにおいても，着目した特徴に対するアプローチによる効果範囲を検証できるデータであると考える．

特徴量選択によるさらなる分析の可能性

本論文では，4.1.1 に前述したように 1 日のアプリカテゴリ毎の使用時間を主な特徴量として採用し，1 日のスマートフォン利用を分析したが，下記の通り，アプリ使用を細分化するなど，別の考え方で特徴量を定義することでさらなる洞察を得ることができると考える．今後の課題として，本論文で用いたデータセットに対しても適用し，さらに詳細なスマートフォン利用実態の把握に繋がる分析を行う予定である．

1 回のアプリ使用時間に基づく特徴量の細分化

同一のアプリでも，1 回の使用時間が数秒と短い場合も数時間と長時間にわたる場合もあり，ユーザのなんらかの状況によって使用時間が異なることがある [21]．アプリ毎に分布は異なると考えられるが，アプリ使用時間の値域毎に別々の特徴量としてクラスタリングを行うことで，使用時間の大小を考慮したパターン抽出ができる．このように

[*2] 現在のシナリオは，本論文に記載の調査結果に基づくものではなく，以後の調査により更新されている

アプリの使用時間の傾向とその原因となる要因を明らかにすることは，サービス提供者やアプリ開発者にとってサービスの提供機会を検討する上で有益な情報になると考える．

日・時間帯に基づく特徴量の細分化

平日や休日，あるいは朝・昼・晩のように，日や時間帯によってユーザの生活シーンが異なることに着目すると，各シーンにおいて異なるアプリ使用傾向が得られる可能性がある．実際の生活シーンは，各ユーザの職業や年齢など別の要因にも依存するため，単純に日や時間帯だけで定義することは難しいと考えられるが，分析の詳細化を行うための分類軸のひとつとして意味があると考える [27]．

ユーザコンテキストに基づく特徴量の細分化

ユーザの周囲に誰がいるか，どこにいるか，何の目的で行動しているかなど，日時以外のユーザコンテキストにも依存してスマートフォン利用傾向は変化すると考えられる．端末で取得可能なログから直接的にユーザコンテキストを得ることは難しいが，例えば，端末が接続する携帯電話基地局や無線 LAN アクセスポイントとの接続状況の変化を単位時間ごとに観察することで，ユーザの静止・移動状態を推定することはできる．移動中におけるスマートフォン利用実態を考察できることは，位置情報利用を伴うアプリの新規検討からアプリによるサービス品質の向上を検討するなど，コンテキストを考慮することで各アプリに特化したデータの活用ができる．

ハードウェアリソース消費に基づくアプリカテゴリ分類　サービス利用やユーザ行動の理解という観点から，本論文では Google Play のカテゴリと手動での判別に基づくアプリカテゴリ分けを行ったが，これらの観点を除外すると，CPU，無線，ストレージ，GPS などのセンサデバイスなどハードウェアリソースの消費量によってアプリをカテゴリ分けすることも可能である．例えば，このカテゴリ毎の使用時間を特徴量として 1 日のスマートフォン利用をクラスタリングすることで，分析結果からユーザの行動を直接的に把握できる情報が欠けてしまうが，結果的にユーザの実利用を考慮しつつ，ハードウェアリソース観

点でより正確なスマートフォン利用パターンの抽出ができると考えられるため，CPU のスケジューリングなど OS レベルの最適化問題に有用な知見が得られることが期待される．

制約

本項における議論の最後に，本モデルの制約事項について述べる．本調査では，日本国内において大きなシェアを占める Android 端末ユーザを調査対象に実施したものであるが，調査期間が限られていること，全ての機種，OS，通信事業者を網羅したものではないため，本モデルの利用にあたっては以下の制約がある．はじめに，調査期間が限られていることから，アプリをはじめスマホ利用には季節に依存した傾向が含まれている可能性がある．また，将来的に新たな人気アプリが登場するなど流行に依存してスマホ利用が変化していく可能性があるため，今後の検討においてモデルをアップデートしていく必要がある．さらに，一部のアプリには通信事業者固有のものが存在する．今回，3.2.3 に示した主要なアプリの使用状況からは事業者固有のアプリは現れていなかったが，収集データの網羅性という観点では限定的であるため，前述の流行依存性の問題とあわせて，今後の検討においてさらに網羅性のある調査を実施したい．

次に，本モデルが示すデータは，個別のサービス・製品や要素技術の仕様やユースケースに特化した課題解決を行うために必要な情報を全て網羅しているわけではない．例えば，位置情報の扱うアプリの動作試験において試験シナリオを作成する際には，どのような頻度で位置情報測位を想定するかなど，本モデルでは網羅していないため，別途，詳細なアプリ挙動を収集する分析ツール [29] などを用いて分析する必要がある．

4.6 まとめ

本章では，実際のユーザの利用実態を考慮したスマートフォン利用モデルを作成するため，アプリカテゴリ毎の使用時間など計 58 変数からなる特徴量を用いたクラスタリングにより，6 種類の 1 日の利用パターンを分類し，

各パターンのユーザ属性や端末設定状況などの傾向を分析した．さらに，各ユーザの主な利用パターンの多様性を分析したところ，約半数のユーザは 1 パターンのみ，残りの半数はそのほとんどが高々 2 パターン程度であり，その組み合わせは共通項を持つパターン同士で構成されることが示された．

今回のモデル化は，ユーザに依存して決定されるスマートフォン利用の諸条件を対象にしたものであるが，今後は，特定のアプリに特化した分析やハードウェアリソースの消費傾向との関係など，さらに分析を進めていく予定である．

第 5 章　おわりに

本論文では，実際のサービス利用における端末省電力化を実現するため，ハードウェアからユーザレベルまで統合的に分析可能な手法を実現することを目的とし，1) 端末の消費電力の可視化によるソフトウェア消費電力の分析ツールと，2) ユーザ実利用シナリオを考慮した評価の提案を行った．

1) では，実際のアプリ使用において容易に利用可能なアプリ消費電力の評価を実現するため，電力モデルに基づく消費電力の推定によるアプリ消費電力の評価手法およびツールを提案した．モデルに基づく電力推定手法において，昨今の端末のハードウェア構成に対応すべく，マルチコア CPU のコア数および周波数の変動や，3G/LTE の RRC State を考慮するよう既存研究に見られる電力モデルを拡張した．電力推定精度を評価した結果，一般的なアプリで 10% 前後程度の誤差で推定できることが示された．さらに，従来手法（ARO）と比較し，3.8% 程度と小さなオーバヘッドかつ特権を必要としない RRC State 遷移の推定手段を用いたことで，一定の精度を保ちつつ，容易に利用可能なアプリ消費電力の評価手段を実現することができた．

2) では，実際のユーザの利用実態を考慮したスマートフォン利用モデルを作成するため，アプリカテゴリ毎の使用時間など計 58 変数からなる特徴量を用いたクラスタリングにより，6 種類の 1 日の利用パターンを分類し，各パターンのユーザ属性や端末設定状況などの傾向を分析した．さらに，各ユーザの主な利用パターンの多様性を分析したところ，約半数のユーザは 1 パターンのみ，残りの半数はそのほとんどが高々 2 パターン程度であり，その組み合わせは共通項を持つパターン同士で構成されることが示された．本モデルは，スマートフォン利用を前提にした様々なサービス企画や，研究開発における課題設定や効果検証に対して，現実的に考慮すべき前提条件や評価条件となる有益なデータであり，サービス実利用における端末の消費電力の分析においては，前述のユーザレベルの要因，すなわちユーザ実利用シナ

リオを構築するために利用可能なデータであると考える．

最後に，今後の展望を述べる．これまで，スマートフォンは高い演算能力，高速なモバイル無線通信，GPS や各種センサの搭載による高多機能化が進み，プライベートからビジネス用途まで様々なアプリが利用可能になった．一方で，電池持ち時間以外にも，OS などが異なるマルチデバイス環境での利用に対応するためのアプリ開発コストの増大などの問題が指摘されている．このため近年では，端末側に強く依存していたアプリの構成要素である機能やデータをクラウド側へシフトさせることで，端末側のリソース消費を軽減し，Web アプリ化など端末仕様に極力依存しない設計が求められている．しかし，逆にクラウド側との連携に伴う通信による電力消費が生じるなど，まだシステム全体で最適なリソース配分を行う設計手法が確立されていない．さらに，5G モバイル通信環境や IoT サービスの普及が進むことで，この課題がさらに具体化されていくと考えられる．今後は，将来の IoT サービスを実現するための新たなアプリケーションプラットフォームの設計を探求し，特に前述のリソース最適化の観点でのアプリ/システム設計手法について取り組む予定である．

謝辞

本研究を進めるにあたり，度重なるご指導とご助言を賜り，また多くのご支援を頂戴致しました九州大学大学院 システム情報科学研究院 久住憲嗣准教授に深く感謝の意を表します．本論文を取りまとめるにあたり，ご助言とご指導を賜りました九州大学大学院システム情報科学研究院 福田晃教授ならびに鵜林尚靖教授，公立はこだて未来大学 稲村浩教授に深く感謝いたします．最後に，日頃の研究活動において様々な協力を頂きました九州大学大学院 福田・久住・アシル研究室諸氏に深く感謝し，お礼を申し上げます．

参考文献

[1] Ministry of Internal Affairs and Communications: *2015 White Paper on Information and Communications in Japan*, The Government of Japan (2015).

[2] 川崎仁嗣, 神山剛, 小西哲平, 大久保信三, 太田賢, 稲村浩：Android OS における状態変化通知による通信集中の削減手法，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 7, No. 1, pp. 23–34 (2014).

[3] Lee, D., Ishihara, T., Muroyama, M., Yasuura, H. and Fallah, F.: An Energy Characterization Framework for Software-Based Embedded Systems, *Proc. of 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pp. 59–64 (2006).

[4] 石原亨, 奥平拓見, 久住憲嗣, 神山剛, 関根和寿, 片桐雅二：OS から解析可能な無線通信端末の消費電力モデルとその生成手法，電子情報通信学会技術研究報告ディペンダブルコンピューティング，Vol. 108, No. 464, pp. 25–30 (2009).

[5] Kaneda, Y., Okuhira, T., Ishihara, T., Hisazumi, K., Kamiyama, T. and Katagiri, M.: A run-time power analysis method using OS-observable parameters for mobile terminals, *Proc. of International Conference on Embedded Systems and Intelligent Technology*, ICESIT2010, pp. 1–6 (2010).

[6] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S. and Spatscheck, O.: Profiling Resource Usage for Mobile Applications: A Cross-layer Approach, *Proc. of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys2011, pp. 321–334 (2011).

[7] 3GPP Website: 3G Specifications (online), available from ⟨http://www.3gpp.org/3GPP-specifications⟩ (accessed 2014-04-10).

[8] Jung, W., Kang, C., Yoon, C., Kim, D. and Cha, H.: DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components, *Proc. of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS2012, pp. 353–362 (2012).

[9] Yoon, C., Kim, D., Jung, W., Kang, C. and Cha, H.: AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring, *Proc. of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC 2012, pp. 387–400 (2012).

[10] Sampson, A., Cacaval, C., Ceze, L., Montesinos, P. and Gracia, D. S.: Automatic discovery of performance and energy pitfalls in HTML and CSS, *Proc. of the 2012 IEEE International Symposium on Workload Characterization*, IISWC 2012, pp. 82–83 (2012).

[11] Thiagarajan, N., Aggarwal, G., Nicoara, A., Boneh, D. and Singh, J. P.: Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption, *Proc. of the 21st International Conference on World Wide Web*, WWW 2012, pp. 41–50 (2012).

[12] Tomita, T., Igarashi, Y., Yamasawa, M., Chujo, K., Kato, M., Iida, I. and Mineno, H.: A study of the Product Development Process through Strengthened Fundamental R&D - Based on a Case Study of Mobile Phone Businesses for Senior Users -, *Proc. of International Workshop on Informatics*, IWIN2014, Informatics Society, pp. 69–78 (2014).

[13] Ministry of Internal Affairs and Communications, Japan: White Paper 2014 | Information and Communications in Japan (online), available from ⟨http://www.soumu.go.jp/johotsusintokei/whitepaper/eng/WP2014/2014-index.html⟩ (accessed 2017-12-01).

[14] Zickuhr, K.: Location-based services, *Pew Research*, pp. 1–25 (2013).

[15] Shokri, R., Theodorakopoulos, G., Le Boudec, J.-Y. and Hubaux, J.-P.: Quantifying location privacy, *Proc. of 2011 IEEE Symposium on Security and Privacy*, IEEE, pp. 247–262 (2011).

[16] Dong, M. and Zhong, L.: Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays, *Proc. of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys2011, pp. 85–98 (2011).

[17] 野呂正明，村上岳生，上和田徹，石原輝雄：1 フレーム分の描画処理終了推定による GPU 状態制御，情報処理学会論文誌，Vol. 57, No. 2, pp. 394–405 (2016).

[18] Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L. and Clause, J.: An Empirical Study of Practitioners' Perspectives on Green Software Engineering, *Proc. of the 38th International Conference on Software Engineering*, ICSE2016, pp. 237–248 (2016).

[19] Nagappan, M. and Shihab, E.: Future Trends in Software Engineering Research for Mobile Apps, *Proc. of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER2016, Vol. 5, pp. 21–32 (2016).

[20] Lu, X., Liu, X., Li, H., Xie, T., Mei, Q., Hao, D., Huang, G. and Feng, F.: PRADA: Prioritizing Android Devices for Apps by Mining Large-scale Usage Data, *Proc. of the 38th International Conference on Software Engineering*, ICSE2016, pp. 3–13 (2016).

[21] Ferreira, D., Goncalves, J., Kostakos, V., Barkhuus, L. and Dey, A. K.: Contextual Experience Sampling of Mobile Application Micro-usage, *Proc. of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services*, MobileHCI2014, pp. 91–100 (2014).

[22] Parate, A., Böhmer, M., Chu, D., Ganesan, D. and Marlin, B. M.: Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones, *Proc. of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp2013, pp. 275–284 (2013).

[23] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R. and Estrin, D.: Diversity in Smartphone Usage, *Proc. of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys2010, pp. 179–194 (2010).

[24] Rogers, E. M.: *Diffusion of Innovations*, Free Press, 5th edition (2003).

[25] Patil, S., Kim, Y., Korgaonkar, K., Awwal, I. and Rosing, T. S.: Characterization of User's Behavior Variations for Design of Replayable Mobile Workloads, *Proc. of the 7th International Conference on Mobile Computing, Applications, and Services*, MobiCASE2015, pp. 51–70 (2015).

[26] Li, H., Lu, X., Liu, X., Xie, T., Bian, K., Lin, F. X., Mei, Q. and Feng, F.: Characterizing Smartphone Usage Patterns from Millions of Android Users, *Proc. of the 2015 ACM Conference on Internet Measurement Conference*, IMC2015, pp. 459–472 (2015).

[27] Zhao, S., Ramos, J., Tao, J., Jiang, Z., Li, S., Wu, Z., Pan, G. and Dey, A. K.: Discovering Different Kinds of Smartphone Users Through Their Application Usage Behaviors, *Proc. of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp2016, pp. 498–509 (2016).

[28] フィリップ・コトラー：コトラーのマーケティング入門，丸善出版，4th edition (2014).

[29] 古庄裕貴, 久住憲嗣, 神山剛, 稲村浩, 中西恒夫, 福田晃：Android アプリケーションの利用情報に基づく消費電力分析手法，情報処理学会論文誌，Vol. 55, No. 8, pp. 1807–1816 (2014).

[30] ARM, Inc.: ARM Information Center: Cortex-M3 テクニカルリファレンスマニュアル（オンライン），入手先 ⟨http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337gj/Cihjbbge.html⟩（参照 2014-04-14）.

[31] The Khronos Group, Inc.: OpenGL ES Overview (online), available from ⟨http://www.khronos.org/opengles/⟩ (accessed 2013-04-10).

[32] Basmadjian, R. and de Meer, H.: Evaluating and modeling power consumption of multi-core processors, *Proc. of 2012 Third International Conference on Future Systems: Where Energy, Computing and Communication Meet*, e-Energy 2012, pp. 1–10 (2012).

[33] QUALCOMM: QxDM Professional: QUALCOMM eXtensible Diagnostic Monitor (online), available from ⟨http://www.qualcomm.com/solutions/testing/diagnostics-software⟩ (accessed 2014-04-10).

[34] Puttonen, J., Virtej, E., Keskitalo, I. and Malkamki, E.: On LTE performance trade-off between connected and idle states with always-on type applications, *Proc. of 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications*, PIMRC, pp. 981–985 (2012).

[35] QUALCOMM: Snapdragon Mobile Platforms, Processors, Modems and Chipsets (online), available from ⟨http://www.qualcomm.com/snapdragon⟩ (accessed 2014-04-10).

[36] Texas Instruments Inc.: OMAP4460 (online), available from ⟨http://www.ti.com/product/omap4460⟩ (accessed 2014-04-10).

[37] VMware and Trango: Mobile Virtualization Platform (MVP) (online), available from ⟨http://www.vmware.com/jp/company/acquisitions/trango⟩ (accessed 2014-04-14).

[38] AnTuTu Hong Kong: AnTuTu Benchmark (online), available from ⟨http://www.antutu.com/en/index.htm⟩ (accessed 2017-12-01).

[39] 小西哲平, 稲村浩, 川崎仁嗣, 神山剛, 大久保信三, 太田賢：画面オフ状態におけるバックグラウンドタスク同時実行による Android 端末の省電力化，情報処理学会論文誌，Vol. 55, No. 2, pp. 587–597 (2014).

[40] 株式会社 MM 総研：2013 年度通期国内携帯電話端末出荷概況 (2014).

[41] 株式会社博報堂 DY ホールディングス：全国スマートフォンユーザ 1000 人定期調査 (2014).

[42] Mohapatra, D. and Suma, S. B.: Survey of location based wireless services, *Proc. of 2005 IEEE International Conference on Personal Wire-*

less Communications, ICPWC2005, pp. 358–362 (2005).

[43] MacQueen, J.: Some methods for classification and analysis of multivariate observations, *Proc. of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, University of California Press, pp. 281–297 (1967).

[44] ジョン・S・プルーイット：ペルソナ戦略-マーケティング，製品開発，デザインを顧客志向にする，ダイヤモンド社 (2007).

[45] 株式会社 NTT ドコモ：実使用時間について（オンライン），入手先 ⟨https://www.nttdocomo.co.jp/product/battery_life/⟩（参照 2016-04-31）.

本研究に関する著者の発表論文

(1) 神山剛，稲村浩，太田賢，電力モデルに基づくアプリ消費電力可視化ツールの評価，マルチメディア, 分散協調とモバイルシンポジウム 2013 論文集, DICOMO 2013, pp.286-292, 2013.

(2)Takeshi Kamiyama, Hiroshi Inamura and Ken Ohta: A model-based energy profiler using online logging for Android applications, Proc. of the 7th International Conference of Mobile Computing and Ubiquitous Networking, ICMU 2014, pp.7-13, 2014.

(3) 神山剛，稲村浩，太田賢，電力モデルに基づくアプリ消費電力可視化ツールの評価，情報処理学会論文誌，Vol.55, No.8, pp.1866-1875, 2014.

(4) 神山剛，久住憲嗣，稲村浩，小西哲平，太田賢，福田晃, ユーザ利用実態調査に基づくスマートフォン利用モデル, マルチメディア，分散協調とモバイルシンポジウム 2016 論文集, DICOMO 2016, pp.443-455, 2016.

(5)Takeshi Kamiyama, Kenji Hisaumi, Hiroshi Inamura, Teppei Konishi, Ken Ohta, Akira Fukuda: Smartphone Usage Analysis Based on Actual-Use Survey, Proc. of the 8th EAI International Conference on Mobile Computing, Applications and Services, MobiCASE 2016, pp.108-116, 2016.

(6) 神山剛，久住憲嗣，稲村浩，小西哲平，太田賢，福田晃, ユーザ利用実態調査に基づくスマートフォン利用モデル, 情報処理学会論文誌コンシューマ・デバイス＆システム, 2018-01-22 採録決定.

付録

A　RRC State 推定ロジック

```
1
2  package jp.co.nttdocomo.powerbench.android.rrcstate;
3
4  import java.io.BufferedReader;
5  import java.io.File;
6  import java.io.FileNotFoundException;
7  import java.io.FileReader;
8  import java.io.IOException;
9  import java.util.Timer;
10 import java.util.TimerTask;
11
12 import jp.co.nttdocomo.powerbench.android.service.osresource.
       DeviceParameters;
13 import jp.co.nttdocomo.powerbench.android.service.osresource.
       NetworkPhoneUsageCollector;
14 import jp.co.nttdocomo.powerbench.android.service.osresource.
       NetworkPhoneUsageCollector.NetworkPhoneUsageModel.RRC_STATE;
15 import android.content.Context;
16 import android.content.SharedPreferences;
17 import android.content.SharedPreferences.Editor;
18 import android.os.PowerManager;
19 import android.telephony.TelephonyManager;
20
21 public class RrcStateMachine implements
       NetworkPhoneUsageCollector.RrcStateMachineListener<State> {
22
23     public static final boolean DEBUG_RRC_STATE = false;
24     public static final int NETWORK_TYPE_LTE =
           NetworkPhoneUsageCollector.NETWORK_TYPE_LTE;
25
26     private State state = null;
27     private Context masterContext;
28     private String m3GLteDirectory;
29     private boolean oldNetWorkLTE=false;
```

```
30      private boolean netConnectFlg=false;
31
32      private boolean init=true;
33      private boolean PreservationStart = false;
34      public Timer PreservationTimer = null;
35      private long PreOldTxSendSize;
36      private long PreOldRxSendSize;
37
38      private boolean FastDormancyStart = false;
39      private Timer FastDormancyTimer = null;
40      private long FdOldTxSendSize;
41      private long FdOldRxSendSize;
42
43      public boolean FastDormancyRel8Start = false;
44      public Timer FastDormancyRel8Timer = null;
45
46      //FD状態遷移の保留フラグ：画面OFF時にFD保留であれば状態遷移.
47      public boolean FDreserveflg = false;
48      public boolean FDRel8reserveflg = false;
49
50      public long Fd8OldTxSendSize;
51      public long Fd8OldRxSendSize;
52
53      private boolean pch_Trans_TimerStart = false;
54      public Timer pch_Trans_Timer_Timer = null;
55      private long pch_Trans_TimerOldTxSendSize;
56      private long pch_Trans_TimerOldRxSendSize;
57
58      private boolean idle_Trans_TimerStart = false;
59      public Timer idle_Trans_Timer_Timer = null;
60      private long idle_Trans_TimerOldTxSendSize;
61      private long idle_Trans_TimerOldRxSendSize;
62
63      public long BolckFDTime;
64      public long BolckFD8Time;
65
66      private long  nowtxsize;
67      private long  nowrxsize;
68
69      //IDLE_Trans_Timerグループ(ELTのタイマー)
70      private RRC_STATE[] IDLE_Trans_TimerGr
71        = { RRC_STATE.RRC_LTE_CONNECTED,
72          RRC_STATE.RRC_LTE_CONNECTED_TAIL,
73          RRC_STATE.RRC_LTE_CONNECTED_DRX,
```

```
74          null,
75          };
76
77      //Preservationタイマグループ
78      private RRC_STATE[] PreservationStateGr
79        = { RRC_STATE.RRC_3G_DCH,
80          RRC_STATE.RRC_3G_DCH_TAIL,
81          RRC_STATE.RRC_3G_FACH,
82          RRC_STATE.RRC_3G_FACH_TAIL,
83          RRC_STATE.RRC_3G_PCH,
84          null,
85          };
86
87      //PCH_Trans_Timer遷移グループ
88      private RRC_STATE[] pch_Trans_TimerGr
89        = { RRC_STATE.RRC_3G_DCH,
90          RRC_STATE.RRC_3G_DCH_TAIL,
91          RRC_STATE.RRC_3G_FACH,
92          RRC_STATE.RRC_3G_FACH_TAIL,
93          null,
94          };
95
96      //FastDormancyステートグループ
97      private RRC_STATE[] FastDormancyStateGr
98        = { RRC_STATE.RRC_3G_DCH,
99          RRC_STATE.RRC_3G_DCH_TAIL,
100         RRC_STATE.RRC_3G_FACH,
101         RRC_STATE.RRC_3G_FACH_TAIL,
102         null,
103         };
104
105     //FastDormancyRel8ステートグループ
106     private RRC_STATE[] FastDormancyStateRel8Gr
107       = { RRC_STATE.RRC_3G_DCH,
108         RRC_STATE.RRC_3G_DCH_TAIL,
109         RRC_STATE.RRC_3G_FACH,
110         RRC_STATE.RRC_3G_FACH_TAIL,
111         null,
112         };
113
114     private String RRCSTATE_PREFERENCES = "RrcStatePreferences";
115
116     //各種タイマー初期値
117     private int fach_Trans_Timer = 8000;
```

```
118     private int dch_Trans_Timer_Rx_Byte = 1638;
119     private int dch_Trans_Timer_Tx_Byte = 4096;
120     private int dch_Trans_Timer = 1000;
121     private int rel8_FD_PCH_Trans_Timer = 0;
122     private int pch_to_IDLE_Timer = 330000;
123     private int preservation_Timer = 90000;
124     private int fd_UE_Time = 5000;
125     private int rel8_FD_UE_Timer = 5000;
126     private int fd_UE_Time_BlockTimer = 0;
127     private int rel8_FD_UE_Timer_BlockTimer = 90000;
128     private int pch_Trans_Timer = 70000;
129     private boolean fast_Dormancy = false;
130     private boolean fast_Dormancy_Rer8 =false;
131
132     //LTE関連タイマーだけmsecにする
133     private int traffic_Inactivity_Timer=6400;
134     private int ta_Timer=0;
135     private int drx_Inactivity_Timer=1920;
136     private int idle_Trans_Timer =120000;
137
138     //実際に各ステートに設定される値
139     public int fach_Trans_TimerValue;
140     public int dch_Trans_Timer_Tx_ByteValue;
141     public int dch_Trans_Timer_Rx_ByteValue;
142     public int dch_Trans_TimerValue;
143     public int rel8_FD_PCH_Trans_TimerValue;
144     public int pch_to_IDLE_TimerValue;
145     public int preservation_TimerValue;
146     public int fd_UE_TimeValue;
147     public int rel8_FD_UE_TimerValue;
148     public int pch_Trans_TimerValue;
149     public boolean fastDormancyFlg;
150     public boolean fastDormancyRel8Flg;
151
152     //LTE関連
153     public int traffic_Inactivity_TimerValue;
154     public int ta_TimerValue;
155     public int drx_Inactivity_TimerValue;
156     public int idle_Trans_TimerValue;
157
158
159     public long fd_UE_Time_BlockTimerValue;
160     public long rel8_FD_UE_Timer_BlockTimerValue;
161
```

```
162     public boolean lcdOn=true;
163     /** ディスプレイ制御 */
164     protected PowerManager mPowerMng;
165
166     public RrcStateMachine(Context context ,TelephonyManager
            telephonyManager) {
167
168         masterContext = context;
169         mPowerMng = (PowerManager)context.getSystemService(
                Context.POWER_SERVICE);
170
171         //設定値取得.
172         getPreferencesValue();
173         NetworkPhoneUsageCollector.setRrcStateMachineListener(
                this);
174
175         //接続状態の初期値を決定する。
176         if (!isDataDisconnectedState(telephonyManager.
                getDataState())) {
177             if (telephonyManager.getNetworkType() ==
                    NETWORK_TYPE_LTE) {
178                 state = State_LTE_IDLE.getInstance();
179             } else {
180                 state = State_IDLE.getInstance();
181             }
182         } else {
183             state = State_DISCONNECTED.getInstance();
184         }
185     }
186
187     public void setState(State state) {
188         this.state.timerCancel();
189         this.state = state;
190     }
191
192     public void set_bps(long Txbps, long Rxbps) {
193         //        String nowState;
194         int counter = 0;
195
196         if(init){
197             Txbps=0;Rxbps=0;init=false;
198         }
199
200
```

```
201         if ((Txbps != 0) || (Rxbps != 0)) {
202
203             PreservationStart = false;
204
205             //タイマーキャンセル
206             timerCancel();
207         } else {
208             //無通信状態になったときの遷移関連
209
210             if((fastDormancyFlg)&&(!FastDormancyStart)){
211                 setFastDormancy();
212             }
213
214             if ((fastDormancyRel8Flg) && (!FastDormancyRel8Start
                    )) {
215                 setFastDormancyRel8();
216             }
217
218             //Preservationタイマ
219             while (PreservationStateGr[counter] != null) {
220
221                 //Preservationグループか？
222                 if (PreservationStateGr[counter] == this.
                        get_rrcstate()) {
223                     if (PreservationStart == false) {
224
225                         //現在の送信量取得
226                         getNowSendSize();
227                         PreOldTxSendSize = nowtxsize;
228                         PreOldRxSendSize = nowrxsize;
229
230                         //タイマースタート
231                         PreservationTimer= new Timer();
232                         PreservationTimerSet(this,
                                PreservationTimer,
                                preservation_TimerValue);
233
234                         PreservationStart = true;
235                     }
236                 }
237
238                 counter++;
239             }
240
```

```
241             //PCH_Trans_Timerタイマ
242             counter=0;
243             while (pch_Trans_TimerGr[counter] != null) {
244
245                 //Preservationグループか？
246                 if (pch_Trans_TimerGr[counter] == this.
                        get_rrcstate()) {
247                     if (pch_Trans_TimerStart == false) {
248
249                         //現在の送信量取得
250                         getNowSendSize();
251
252                         pch_Trans_TimerOldTxSendSize = nowtxsize
                                ;
253                         pch_Trans_TimerOldRxSendSize = nowrxsize
                                ;
254
255                         //タイマースタート
256                         pch_Trans_Timer_Timer= new Timer();
257                         CellPchTimerSet(this,
                                pch_Trans_Timer_Timer,
                                pch_Trans_TimerValue);
258
259                         pch_Trans_TimerStart = true;
260                     }
261                 }
262
263                 counter++;
264             }
265             //IDLE_Trans_Timerタイマ
266             counter=0;
267             while (IDLE_Trans_TimerGr[counter] != null) {
268
269                 //IDLE_Trans_TimerGrグループか？
270                 if (IDLE_Trans_TimerGr[counter] == this.
                        get_rrcstate()) {
271                     if (idle_Trans_TimerStart == false) {
272
273                         //現在の送信量取得
274                         getNowSendSize();
275
276                         idle_Trans_TimerOldTxSendSize =
                                nowtxsize;
```

```
277                         idle_Trans_TimerOldRxSendSize =
                                nowrxsize;
278
279                         //タイマースタート
280                         idle_Trans_Timer_Timer= new Timer();
281                         IDLE_Trans_TimerSet(this,
                                idle_Trans_Timer_Timer,
                                idle_Trans_TimerValue);
282                         if (RrcStateMachine.DEBUG_RRC_STATE)
                                System.out.println("RRCState Change
                                : IDLE_Trans_Timer_Timer Start
                                ------------");
283
284                         idle_Trans_TimerStart = true;
285                     }
286                 }
287
288                 counter++;
289             }
290
291         }
292
293         state.setBps(this, Txbps, Rxbps);
294
295     }
296
297     public RRC_STATE get_rrcstate() {
298         return state.get_rrcstate(this);
299     }
300
301     //FastDormancy処理
302     public void setFastDormancy() {
303         int fdCounter = 0;
304
305         if(!fastDormancyFlg){
306             //セットの１回目にプレファレンスに書き込む
307             SharedPreferences pref =
308                     masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.
                            MODE_PRIVATE);
309             fastDormancyFlg = true;
310             Editor edit = pref.edit();
311             edit.putBoolean("FAST_DORMANCY", true);
312             edit.commit();
```

```
313
314         }
315
316         //FDが発動してFdBlockTimerValue（ｍ秒）
317         //までは受け付けない
318         if ((System.currentTimeMillis()) >= BolckFDTime +
                fd_UE_Time_BlockTimerValue) {
319
320             //FastDormancyタイマ
321             while (FastDormancyStateGr[fdCounter] != null) {
322
323                 //FastDormancyグループか？
324                 if (FastDormancyStateGr[fdCounter] == this.
                        get_rrcstate()) {
325                  if ((FastDormancyStart == false)) {
326
327                         //現在の送信量取得
328                         getNowSendSize();
329                         FdOldTxSendSize = nowtxsize;
330                         FdOldRxSendSize = nowrxsize;
331
332                         //タイマースタート
333                         FastDormancyTimer= new Timer();
334                         FastDormancyTimerSet(this,
                                FastDormancyTimer, fd_UE_TimeValue);
335
336                         FastDormancyStart = true;
337                     }
338                 }
339                 fdCounter++;
340             }
341         }
342     }
343     //FastDormancyキャンセル処理
344     public void setFastDormancyCancel() {
345             FastDormancyStart = false;
346
347             SharedPreferences pref =
348                     masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.
                            MODE_PRIVATE);
349             fastDormancyFlg = false;
350             Editor edit = pref.edit();
351             edit.putBoolean("FAST_DORMANCY", false);
```

```
352             edit.commit();
353
354     }
355
356     //FastDormancyRel8処理
357     public void setFastDormancyRel8() {
358
359         if (!fastDormancyRel8Flg) {
360             SharedPreferences pref =
361                     masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.
                            MODE_PRIVATE);
362             fastDormancyRel8Flg = true;
363             Editor edit = pref.edit();
364             edit.putBoolean("FAST_DORMANCY_REL8", true);
365             edit.commit();
366         }
367         int fdCounter = 0;
368         //fastDormancyRel8状態時の処理
369         //FD8が発動してFd8BlockTimerValue（ｍ秒）
370         //までは受け付けない
371         if ((System.currentTimeMillis()) >= BolckFD8Time +
                rel8_FD_UE_Timer_BlockTimerValue) {
372
373             //FastDormancyタイマ
374             while (FastDormancyStateRel8Gr[fdCounter] != null) {
375
376                 //FastDormancyグループか？
377                 if (FastDormancyStateRel8Gr[fdCounter] == this.
                        get_rrcstate()) {
378
379                     //現在の送信量取得
380                     getNowSendSize();
381                     Fd8OldRxSendSize = nowrxsize;
382                     Fd8OldTxSendSize = nowtxsize;
383
384                     //タイマースタート
385                     FastDormancyRel8Timer = new Timer();
386                     FastDormancyRel8TimerSet(this,
                            FastDormancyRel8Timer,
                            rel8_FD_UE_TimerValue);
387
388                     FastDormancyRel8Start = true;
389                 }
```

```
390                 fdCounter++;
391             }
392         }
393
394     }
395
396     //FastDormancyRel8キャンセル処理
397     public void setFastDormancyRel8Cancel() {
398             FastDormancyRel8Start = false;
399             SharedPreferences pref =
400                     masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.
                            MODE_PRIVATE);
401             fastDormancyRel8Flg = false;
402             Editor edit = pref.edit();
403             edit.putBoolean("FAST_DORMANCY_REL8", false);
404             edit.commit();
405     }
406
407     //IDLE_Trans_Timerタイマー
408     private void IDLE_Trans_TimerSet(final RrcStateMachine
            context, Timer timer, int TimerCount) {
409         TimerTask task = new TimerTask() {
410             public void run() {
411                 //実行処理
412
413                 //現在の送信量取得
414                 getNowSendSize();
415
416                 //送受信量に変化がなかったら。
417                 if ((nowtxsize == idle_Trans_TimerOldTxSendSize)
                            && (nowrxsize ==
                            idle_Trans_TimerOldRxSendSize)) {
418
419                     //タイマーをキャンセル
420                     if(idle_Trans_Timer_Timer != null){
421                         idle_Trans_Timer_Timer.cancel();
422                         idle_Trans_Timer_Timer = null;
423                     }
424
425                     //IDLEにステートチェンジ
426                     context.setState(State_LTE_IDLE.getInstance
                            ());
```

```
427                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                            out.println("RRCState Change :
                            IDLE_Trans_TimerGr -> LTE_IDLE
                            ------------");
428
429                     idle_Trans_TimerStart = false;
430
431                     //タイマーをキャンセル
432                     if(idle_Trans_Timer_Timer != null){
433                         idle_Trans_Timer_Timer.cancel();
434                         idle_Trans_Timer_Timer = null;
435                     }
436
437                 } else {
438
439                     //変化があったら何もしない
440                     idle_Trans_TimerStart = false;
441
442                     //タイマーをキャンセル
443                     if(idle_Trans_Timer_Timer != null){
444                         idle_Trans_Timer_Timer.cancel();
445                         idle_Trans_Timer_Timer = null;
446                     }
447
448                 }
449
450             }
451         };
452         timer.schedule(task, TimerCount);
453     }
454     //Preservationタイマー
455     private void PreservationTimerSet(final RrcStateMachine
            context, Timer timer, int TimerCount) {
456         TimerTask task = new TimerTask() {
457             public void run() {
458                 //実行処理
459
460                 //現在の送信量取得
461                 getNowSendSize();
462
463                 //送受信量に変化がなかったら。
464                 if ((nowtxsize == PreOldTxSendSize) && (
                        nowrxsize == PreOldRxSendSize)) {
465
```

```
466                     //IDLEにステートチェンジ
467                     context.setState(State_IDLE.getInstance());
468                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                            out.println("RRCState Change :
                            PreservationGr -> IDLE ------------");
469
470                     PreservationStart = false;
471
472                     //タイマーをキャンセル
473                     timerCancel();
474
475                 } else {
476
477                     //変化があったら何もしない
478                     PreservationStart = false;
479
480                     //タイマーをキャンセル
481                     if(PreservationTimer != null){
482                         PreservationTimer.cancel();
483                         PreservationTimer = null;
484                     }
485                 }
486             }
487         };
488         timer.schedule(task, TimerCount);
489     }
490
491     //PCH_Trans_Timer
492     private void CellPchTimerSet(final RrcStateMachine context,
            Timer timer, int TimerCount) {
493         TimerTask task = new TimerTask() {
494             public void run() {
495                 //実行処理
496
497                 //現在の送信量取得
498                 getNowSendSize();
499
500                 //送受信量に変化がなかったら。
501                 if ((nowtxsize == pch_Trans_TimerOldTxSendSize)
                        && (nowrxsize ==
                        pch_Trans_TimerOldRxSendSize)) {
502
503                     //PCHにステートチェンジ
504                     context.setState(State_PCH.getInstance());
```

```
505                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                            out.println("RRCState Change :
                            PCH_Trans_Timer -> State_PCH
                            ------------");
506
507                     pch_Trans_TimerStart = false;
508                     //タイマーをキャンセル
509                     if(pch_Trans_Timer_Timer != null){
510                         pch_Trans_Timer_Timer.cancel();
511                         pch_Trans_Timer_Timer=null;
512                     }
513
514                 } else {
515
516                     //変化があったら何もしない
517                     pch_Trans_TimerStart = false;
518
519                     //タイマーをキャンセル
520                     if(pch_Trans_Timer_Timer != null){
521                         pch_Trans_Timer_Timer.cancel();
522                         pch_Trans_Timer_Timer=null;
523                     }
524
525                 }
526
527             }
528         };
529         timer.schedule(task, TimerCount);
530     }
531
532
533     //FastDormancyタイマー
534     private void FastDormancyTimerSet(final RrcStateMachine
            context, Timer timer, int TimerCount) {
535         TimerTask task = new TimerTask() {
536             public void run() {
537                 //実行処理
538
539                 //現在の送信量取得
540                 getNowSendSize();
541
542                 //送受信量に変化がなかったら。
543                 if ((nowtxsize == FdOldTxSendSize) && (nowrxsize
                        == FdOldRxSendSize)) {
```

```
544
545                     if( context.mPowerMng.isScreenOn() == false
                            ){
546                   //画面OFF状態：FD遷移可能.
547
548                   //タイマーをキャンセル
549                      timerCancel();
550
551       //システム時刻を取得
552       BolckFDTime = System.currentTimeMillis();
553
554                      //IDLEにステートチェンジ
555                      context.setState(State_IDLE.getInstance());
556                      if (RrcStateMachine.DEBUG_RRC_STATE) System
                            .out.println("RRCState Change :
                            FastDormancyGr -> IDLE ------------");
557                   } else {
558                    //画面ON状態：FD遷移保留(画面
                          OFFを契機に状態遷移).
559                    context.FDreserveflg = true;
560                   }
561                  } else {
562
563                      //タイマーをキャンセル
564                      timerCancel();
565
566                      //変化があったら何もしない
567                      FastDormancyStart = false;
568
569                  }
570
571              }
572          };
573          timer.schedule(task, TimerCount);
574      }
575      //FastDormancyRel8タイマー
576      private void FastDormancyRel8TimerSet(final RrcStateMachine
             context, Timer timer, int TimerCount) {
577          TimerTask task = new TimerTask() {
578              public void run() {
579                  //実行処理
580
581                  //現在の送信量取得
582                  getNowSendSize();
```

```
583
584                 //送受信量に変化がなかったら。
585                 if ((nowtxsize == Fd8OldTxSendSize) && (
                        nowrxsize == Fd8OldRxSendSize)) {
586
587                     if (mPowerMng.isScreenOn() == false) {
588                         //画面OFF状態：FD-Rel8遷移可能.
589                         //タイマーをキャンセル
590                         fdRel8timerCancel();
591
592                         //システム時刻を取得
593                         BolckFD8Time = System.currentTimeMillis
                                ();
594
595                         //PCHにステートチェンジ
596                         setState(State_PCH.getInstance());
597                         if (RrcStateMachine.DEBUG_RRC_STATE)
                                System.out.println("RRCState Change
                                : FastDormancyRel8 -> PCH
                                ------------");
598
599                         //PCH_TRANSタイマーが
600                         //作動していたらキャンセル
601                         if (pch_Trans_Timer_Timer != null) {
602                             pch_Trans_Timer_Timer.cancel();
603                             pch_Trans_Timer_Timer = null;
604                             pch_Trans_TimerStart = false;
605                         }
606
607                         //同期的に動作させる為、set_bpsをコールする。
608                         set_bps(0, 0);
609
610                     } else {
611                         //画面ON状態：FD-Rel8遷移保留(画面
                                OFFを契機に状態遷移).
612                         FDRel8reserveflg = true;
613
614                     }
615                 } else {
616                     //タイマーをキャンセル
617                     fdRel8timerCancel();
618                 }
619             }
620         };
```

```
621         timer.schedule(task, TimerCount);
622     }
623
624     private void getNowSendSize() {
625
626         //現在の送信量をゲット
627
628         //****************
629         //  3G/LTE 関連ファイル初期化
630         //****************
631         String line_phone = null;
632         File networkReceiveStats_phone = null;
633         FileReader networkReceiveStatsFileReader_phone = null;
634         BufferedReader networkReceiveStatsBufferdReader_phone =
                null;
635         File networkSendStats_phone = null;
636         FileReader networkSendStatsFileReader_phone = null;
637         BufferedReader networkSendStatsBufferdReader_phone =
                null;
638         String networkSendSize_phone_line = null;
639         String networkReceiveSize_phone_line = null;
640
641         try {
642
643             //無線送受信(3G/LTE)情報取得設定 (NETWORK_TYPE.
                    NETWORK_3G || NETWORK_TYPE.NETWORK_LTE)
644             {
645                 //無線情報可能先ディレクトリの取得
646                 m3GLteDirectory = DeviceParameters.
                        getNETWORK_3G_LTE_DIRECTORY();
647
648                 //無線送信情報取得
649                 networkSendStats_phone = new File(
                        m3GLteDirectory + "tx_bytes");
650                 networkSendStatsFileReader_phone = new
                        FileReader(networkSendStats_phone);
651                 networkSendStatsBufferdReader_phone = new
                        BufferedReader(
                        networkSendStatsFileReader_phone, 1024);
652
653                 while ((line_phone =
                        networkSendStatsBufferdReader_phone.readLine
                        ()) != null) {
654                     networkSendSize_phone_line = line_phone;
```

```
655                 }
656
657                 //無線受信情報取得
658                 networkReceiveStats_phone = new File(
                        m3GLteDirectory + "rx_bytes");
659                 networkReceiveStatsFileReader_phone = new
                        FileReader(networkReceiveStats_phone);
660                 networkReceiveStatsBufferdReader_phone = new
                        BufferedReader(
                        networkReceiveStatsFileReader_phone, 1024);
661
662                 while ((line_phone =
                        networkReceiveStatsBufferdReader_phone.
                        readLine()) != null) {
663                     networkReceiveSize_phone_line = line_phone;
664                 }
665
666                 nowtxsize =  Long.parseLong(
                        networkSendSize_phone_line);
667                 nowrxsize =  Long.parseLong(
                        networkReceiveSize_phone_line);
668
669             }
670         } catch (FileNotFoundException e) {
671             //デバイス・ファイルが見つからなかった場合
672         } catch (IOException e) {
673             //デバイス・ファイル・アクセスに失敗した場合
674             e.printStackTrace();
675         } finally {
676             //****************
677             //  3G/LTE 関連ファイル後始末
678             //****************
679             try {
680                 if (networkReceiveStatsBufferdReader_phone !=
                        null)
681                     networkReceiveStatsBufferdReader_phone.close
                            ();
682             } catch (IOException e) {
683             }
684             try {
685                 if (networkReceiveStatsFileReader_phone != null)
686                     networkReceiveStatsFileReader_phone.close();
687             } catch (IOException e) {
688             }
```

```
689             networkReceiveStatsBufferdReader_phone = null;
690             networkReceiveStatsFileReader_phone = null;
691 
692             try {
693                 if (networkSendStatsBufferdReader_phone != null)
694                     networkSendStatsBufferdReader_phone.close();
695             } catch (IOException e) {
696             }
697             try {
698                 if (networkSendStatsFileReader_phone != null)
699                     networkSendStatsFileReader_phone.close();
700             } catch (IOException e) {
701             }
702             networkSendStatsBufferdReader_phone = null;
703             networkSendStatsFileReader_phone = null;
704             networkSendStats_phone = null;
705         }
706 
707     }
708 
709     public void timerCancel() {
710         //タイマーキャンセル
711         if (PreservationTimer != null) {
712 
713             PreservationTimer.cancel();
714             PreservationTimer= null;
715          }
716         if (FastDormancyTimer != null) {
717 
718             FastDormancyTimer.cancel();
719             FastDormancyTimer= null;
720 
721         }
722         if (FastDormancyRel8Timer != null) {
723 
724             FastDormancyRel8Timer.cancel();
725             FastDormancyRel8Timer= null;
726 
727         }
728         if (pch_Trans_Timer_Timer != null) {
729 
730             pch_Trans_Timer_Timer.cancel();
731             pch_Trans_Timer_Timer= null;
732 
```

```
733         }
734         if(idle_Trans_Timer_Timer != null){
735             idle_Trans_Timer_Timer.cancel();
736             idle_Trans_Timer_Timer = null;
737         }
738 
739         PreservationStart = false;
740         FastDormancyStart = false;
741         FastDormancyRel8Start = false;
742         pch_Trans_TimerStart = false;
743         idle_Trans_TimerStart = false;
744         FDreserveflg = false;
745   FDRel8reserveflg = false;
746     }
747 
748     public void fdRel8timerCancel() {
749         //FD-Rel8タイマキャンセル
750         if (FastDormancyRel8Timer != null) {
751             FastDormancyRel8Timer.cancel();
752             FastDormancyRel8Timer=null;
753             if (RrcStateMachine.DEBUG_RRC_STATE) System.out.
                    println("RRCState Cancel : FastDormancyRel8
                    Timer Cancel ------------");
754         }
755         if (FastDormancyTimer != null) {
756             FastDormancyTimer.cancel();
757             FastDormancyTimer=null;
758             if (RrcStateMachine.DEBUG_RRC_STATE) System.out.
                    println("RRCState Cancel : FastDormancy Timer
                    Cancel ------------");
759         }
760         FastDormancyRel8Start = false;
761         FastDormancyStart = false;
762         FDreserveflg = false;
763   FDRel8reserveflg = false;
764     }
765 
766 public void LcdStateChangeFD() {
767  //FD遷移が保留状態であった場合、画面OFFを契機に状態遷移.
768 
769  if(FDreserveflg){
770   //FD保留中.
771   timerCancel();
772   BolckFDTime = System.currentTimeMillis();
```

```
773             setState(State_IDLE.getInstance());
774             if (RrcStateMachine.DEBUG_RRC_STATE) System.out.println(
                    "RRCState Change : FD-LCD OFF -> IDLE ------------"
                    );
775
776      } else if(FDRel8reserveflg){
777       //FD-Rel8保留中.
778             int fdCounter = 0;
779       while (FastDormancyStateRel8Gr[fdCounter] != null) {
780        if (FastDormancyStateRel8Gr[fdCounter] == this.get_rrcstate
              ()) {
781         //FDRel8Grであれば状態遷移可能.
782               BolckFD8Time = System.currentTimeMillis();
783               setState(State_PCH.getInstance());
784               if (RrcStateMachine.DEBUG_RRC_STATE) System.out.
                      println("RRCState Change : FDRel8-LCD OFF -> PCH
                      ------------");
785                     //PCH_TRANSタイマーが
786                     //作動していたらキャンセル
787                     if (pch_Trans_Timer_Timer != null) {
788                         pch_Trans_Timer_Timer.cancel();
789                         pch_Trans_Timer_Timer = null;
790                         pch_Trans_TimerStart = false;
791                     }
792
793        }
794        fdCounter++;
795       }
796       //FD-Rel8状態クリア.
797       fdRel8timerCancel();
798      } else {
799       //保留無し：nop
800      }
801    }
802
803        //LTE--3G切替処理
804        public void changeNetworkType(int state ,int networkType) {
805
806             boolean netWorkLTE = false;
807
808            if(networkType == NETWORK_TYPE_LTE){
809                netWorkLTE = true;
810            }else{
811                netWorkLTE = false;
```

```
812         }
813
814         if (isDataDisconnectedState(state)) {
815             timerCancel();
816             netConnectFlg = false;
817             setState(State_DISCONNECTED.getInstance());
818             if (RrcStateMachine.DEBUG_RRC_STATE) System.out.
                println("RRCState  :State_DISCONNECTED Now!!
                ------------");
819         } else {
820             if (!netConnectFlg) {
821                 if (netWorkLTE) {
822                     //NETWORK_TYPE_LTE
823                     setState(State_LTE_IDLE.getInstance());
824                     oldNetWorkLTE = true;
825                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                        out.println("RRCState Change :
                        State_DISCONNECTED -> State_LTE_IDLE
                        ------------");
826                 } else {
827                     //LTE以外
828                     setState(State_IDLE.getInstance());
829                     oldNetWorkLTE = false;
830                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                        out.println("RRCState Change :
                        State_DISCONNECTED -> State_IDLE
                        ------------");
831                 }
832                 netConnectFlg = true;
833             } else if ((netWorkLTE == false) && (oldNetWorkLTE
                == true)) {
834                 //LTEから3Gに切り替わった
835                 if (this.get_rrcstate() == RRC_STATE.
                    RRC_LTE_CONNECTED) {
836                     setState(State_DCH.getInstance());
837                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                        out.println("RRCState Change :
                        State_LTE_Connect -> State_DCH
                        ------------");
838                 } else {
839                     setState(State_IDLE.getInstance());
840                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                        out.println("RRCState Change :
                        State_LTE_IDLE -> State_IDLE
```

```
                        ------------");
841                 }
842                 //IDLE_Trans_Timerタイマーをキャンセル
843                 if (idle_Trans_Timer_Timer != null) {
844                     idle_Trans_Timer_Timer.cancel();
845                     idle_Trans_Timer_Timer = null;
846                 }
847                 timerCancel();
848                 oldNetWorkLTE = false;
849             } else if ((netWorkLTE == true) && (oldNetWorkLTE ==
                     false))
850             {
851                 //3GからLTEに切り替わった
852                 if (this.get_rrcstate() == RRC_STATE.RRC_3G_DCH)
                         {
853                     setState(State_LTE_Connect.getInstance());
854                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                         out.println("RRCState Change : State_DCH
                          -> State_LTE_Connect ------------");
855                 } else {
856                     setState(State_LTE_IDLE.getInstance());
857                     if (RrcStateMachine.DEBUG_RRC_STATE) System.
                         out.println("RRCState Change :
                         State_IDLE -> State_LTE_IDLE
                         ------------");
858                 }
859                 timerCancel();
860                 oldNetWorkLTE = true;
861             }
862         }
863     }
864
865     public boolean isDataDisconnectedState(int state) {
866         boolean ret = true;
867         switch(state) {
868         case TelephonyManager.DATA_CONNECTED:
869         case TelephonyManager.DATA_CONNECTING:
870         case TelephonyManager.DATA_SUSPENDED:
871             ret = false;
872             break;
873         }
874         return ret;
875     }
876
```

```
877     public void setLcd(boolean b) {
878         //バックライト状態取得
879         lcdOn =b;
880 
881         if(false == lcdOn){
882         //画面OFF時にFD保留があれば状態遷移.
883          LcdStateChangeFD();
884         }
885     }
886 
887     public void setConnection(int state) {
888         //通信状態取得フラグ
889 
890         //LCDがONになった時、通信状態がDISCONNECTED
891         //だったらステートチェンジ
892         //ディープスリープ復帰時の為の保険処理
893         if (isDataDisconnectedState(state)) {
894             netConnectFlg = false;
895             setState(State_DISCONNECTED.getInstance());
896             if (RrcStateMachine.DEBUG_RRC_STATE) System.out.
                    println("RRCState  LCD_ON:State_DISCONNECTED
                    ------------");
897             timerCancel();
898         }
899     }
900 
901     //==========================================
902     //===========  設定値GET&SET処理  =============
903     //==========================================
904 
905     //Fach_Trans_Timer設定値セット
906     public void setFach_Trans_Timer(int value) {
907 
908         SharedPreferences pref =
909                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
910         fach_Trans_TimerValue = value;
911         Editor edit = pref.edit();
912         edit.putInt("Fach_Trans_Timer", value);
913         edit.commit();
914 
915     }
916     //Fach時にDchにシフトアップするときの送信容量閾値セット
917     public void setDCH_Trans_Timer_Tx_Byte(int value) {
```

```
918
919         SharedPreferences pref =
920                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
921         dch_Trans_Timer_Tx_ByteValue = value;
922
923         Editor edit = pref.edit();
924         edit.putInt("DCH_Trans_Timer_Tx_Byte", value);
925         edit.commit();
926
927     }
928
929     //Fach時にDchにシフトアップするときの受信容量閾値セット
930     public void setDCH_Trans_Timer_Rx_Byte(int value) {
931
932         SharedPreferences pref =
933                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
934         dch_Trans_Timer_Rx_ByteValue = value;
935         Editor edit = pref.edit();
936         edit.putInt("DCH_Trans_Timer_Rx_Byte", value);
937         edit.commit();
938
939     }
940
941     //DCH_Trans_Timer値セット
942     public void setDCH_Trans_Timer(int value) {
943
944         SharedPreferences pref =
945                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
946         dch_Trans_TimerValue = value;
947         Editor edit = pref.edit();
948         edit.putInt("DCH_Trans_Timer", value);
949         edit.commit();
950
951     }
952     //Rel8_FD_PCH_Trans_Timer値セット(FD-Rel8用)
953     public void setRel8_FD_PCH_Trans_Timer(int value) {
954
955         SharedPreferences pref =
956                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
957         rel8_FD_PCH_Trans_TimerValue = value;
```

```
958         Editor edit = pref.edit();
959         edit.putInt("Rel8_FD_PCH_Trans_Timer", value);
960         edit.commit();
961
962     }
963
964     //PCH_to_IDLE_Time設定値セット
965     public void setPCH_to_IDLE_Timer(int value) {
966
967         SharedPreferences pref =
968                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
969         pch_to_IDLE_TimerValue = value;
970         Editor edit = pref.edit();
971         edit.putInt("PCH_to_IDLE_Time", value);
972         edit.commit();
973
974     }
975
976     //Preservation_Timer設定値セット
977     public void setPreservation_Timer(int value) {
978
979         SharedPreferences pref =
980                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
981         preservation_TimerValue = value;
982         Editor edit = pref.edit();
983         edit.putInt("Preservation_Timer", value);
984         edit.commit();
985
986     }
987
988     //FD_UE_Time設定値セット
989     public void setFD_UE_Time(int value) {
990
991         SharedPreferences pref =
992                 masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
993         fd_UE_TimeValue = value;
994         Editor edit = pref.edit();
995         edit.putInt("FD_UE_Time", value);
996         edit.commit();
997
998     }
```

```
999
1000    //Rel8_FD_UE_Timer設定値セット
1001    public void setRel8_FD_UE_Timer(int value) {
1002
1003        SharedPreferences pref =
1004                masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1005        rel8_FD_UE_TimerValue = value;
1006        Editor edit = pref.edit();
1007        edit.putInt("Rel8_FD_UE_Timer", value);
1008        edit.commit();
1009
1010    }
1011
1012    //FD_UE_Time_BlockTimer設定値セット
1013    public void setFD_UE_Time_BlockTimer(int value) {
1014
1015        SharedPreferences pref =
1016                masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1017        fd_UE_Time_BlockTimerValue = value;
1018        Editor edit = pref.edit();
1019        edit.putInt("FD_UE_Time_BlockTimer", value);
1020        edit.commit();
1021
1022    }
1023
1024    //Rel8_FD_UE_Timer_BlockTimer設定値セット
1025    public void setRel8_FD_UE_Timer_BlockTimer(int value) {
1026
1027        SharedPreferences pref =
1028                masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1029        rel8_FD_UE_Timer_BlockTimerValue = value;
1030        Editor edit = pref.edit();
1031        edit.putInt("Rel8_FD_UE_Timer_BlockTimer", value);
1032        edit.commit();
1033
1034    }
1035    //PCH_Trans_Timer設定値セット
1036    public void setPCH_Trans_Timer(int value) {
1037
1038        SharedPreferences pref =
```

```
1039                masterContext.getSharedPreferences(
                        RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1040            pch_Trans_TimerValue = value;
1041            Editor edit = pref.edit();
1042            edit.putInt("PCH_Trans_Timer", value);
1043            edit.commit();
1044
1045        }
1046
1047        //LTE関連●●●●●●●●●●
1048        //traffic_Inactivity_Timer設定値セット
1049        public void setTraffic_Inactivity_Timer(int value) {
1050
1051            SharedPreferences pref =
1052                    masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1053            traffic_Inactivity_TimerValue = value;
1054            Editor edit = pref.edit();
1055            edit.putInt("Traffic_Inactivity_Timer", value);
1056            edit.commit();
1057
1058        }
1059        //ta_Timer設定値セット
1060        public void setTa_Timer(int value) {
1061
1062            SharedPreferences pref =
1063                    masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1064            ta_TimerValue = value;
1065            Editor edit = pref.edit();
1066            edit.putInt("Ta_Timer", value);
1067            edit.commit();
1068
1069        }
1070        //drx_Inactivity_Timer設定値セット
1071        public void setDRX_Inactivity_Timer(int value) {
1072
1073            SharedPreferences pref =
1074                    masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1075            drx_Inactivity_TimerValue = value;
1076            Editor edit = pref.edit();
1077            edit.putInt("DRX_Inactivity_Time", value);
1078            edit.commit();
```

```
1079
1080        }
1081        //IDLE_Trans_Timer設定値セット
1082        public void setIDLE_Trans_Timer(int value) {
1083
1084            SharedPreferences pref =
1085                    masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1086            idle_Trans_TimerValue = value;
1087            Editor edit = pref.edit();
1088            edit.putInt("IDLE_Trans_Timer", value);
1089            edit.commit();
1090
1091        }
1092
1093
1094
1095        private void getPreferencesValue() {
1096            //SharedPreferencesを見て、設定されてなかったら
1097            //デフォルト値を各ステートに渡す
1098            SharedPreferences pref =
1099                    masterContext.getSharedPreferences(
                            RRCSTATE_PREFERENCES, Context.MODE_PRIVATE);
1100            Editor edit = pref.edit();
1101
1102            if ((fach_Trans_TimerValue = pref.getInt("
                    Fach_Trans_Timer", -1)) == -1) {
1103                //設定値が無いのでデフォルト値
1104                fach_Trans_TimerValue = fach_Trans_Timer;
1105                //ついでにデフォルト値を書き込む
1106                edit.putInt("Fach_Trans_Timer", fach_Trans_Timer);
1107                edit.commit();
1108            }
1109            if ((dch_Trans_Timer_Tx_ByteValue = pref.getInt("
                    DCH_Trans_Timer_Tx_Byte", -1)) == -1) {
1110                dch_Trans_Timer_Tx_ByteValue =
                        dch_Trans_Timer_Tx_Byte;
1111                edit.putInt("DCH_Trans_Timer_Tx_Byte",
                        dch_Trans_Timer_Tx_Byte);
1112                edit.commit();
1113            }
1114            if ((dch_Trans_Timer_Rx_ByteValue = pref.getInt("
                    DCH_Trans_Timer_Rx_Byte", -1)) == -1) {
```

```
1115            dch_Trans_Timer_Rx_ByteValue =
                    dch_Trans_Timer_Rx_Byte;
1116            edit.putInt("DCH_Trans_Timer_Rx_Byte",
                    dch_Trans_Timer_Rx_Byte);
1117            edit.commit();
1118        }
1119        if ((dch_Trans_TimerValue = pref.getInt("DCH_Trans_Timer
                ", -1)) == -1) {
1120            dch_Trans_TimerValue = dch_Trans_Timer;
1121            edit.putInt("DCH_Trans_Timer", dch_Trans_Timer);
1122            edit.commit();
1123        }
1124        if ((rel8_FD_PCH_Trans_TimerValue = pref.getInt("
                Rel8_FD_PCH_Trans_Timer", -1)) == -1) {
1125            rel8_FD_PCH_Trans_TimerValue =
                    rel8_FD_PCH_Trans_Timer;
1126            edit.putInt("Rel8_FD_PCH_Trans_Timer",
                    rel8_FD_PCH_Trans_Timer);
1127            edit.commit();
1128        }
1129        if ((pch_to_IDLE_TimerValue = pref.getInt("
                PCH_to_IDLE_Timer", -1)) == -1) {
1130            pch_to_IDLE_TimerValue = pch_to_IDLE_Timer;
1131            edit.putInt("PCH_to_IDLE_Timer", pch_to_IDLE_Timer);
1132            edit.commit();
1133        }
1134        if ((preservation_TimerValue = pref.getInt("
                Preservation_Timer", -1)) == -1) {
1135            preservation_TimerValue = preservation_Timer;
1136            edit.putInt("Preservation_Timer", preservation_Timer
                    );
1137            edit.commit();
1138        }
1139        if ((fd_UE_TimeValue = pref.getInt("FD_UE_Time", -1)) ==
                 -1) {
1140            fd_UE_TimeValue = fd_UE_Time;
1141            edit.putInt("FD_UE_Time", fd_UE_Time);
1142            edit.commit();
1143        }
1144        if ((rel8_FD_UE_TimerValue = pref.getInt("
                Rel8_FD_UE_Timer", -1)) == -1) {
1145            rel8_FD_UE_TimerValue = rel8_FD_UE_Timer;
1146            edit.putInt("Rel8_FD_UE_Timer", rel8_FD_UE_Timer);
1147            edit.commit();
```

```
1148            }
1149            if ((fd_UE_Time_BlockTimerValue = pref.getInt("
                    FD_UE_Time_BlockTimer", -1)) == -1) {
1150                fd_UE_Time_BlockTimerValue = fd_UE_Time_BlockTimer;
1151                edit.putInt("FD_UE_Time_BlockTimer",
                        fd_UE_Time_BlockTimer);
1152                edit.commit();
1153            }
1154            if ((rel8_FD_UE_Timer_BlockTimerValue = pref.getInt("
                    Rel8_FD_UE_Timer_BlockTimer", -1)) == -1) {
1155                rel8_FD_UE_Timer_BlockTimerValue =
                        rel8_FD_UE_Timer_BlockTimer;
1156                edit.putInt("Rel8_FD_UE_Timer_BlockTimer",
                        rel8_FD_UE_Timer_BlockTimer);
1157                edit.commit();
1158            }
1159            if ((pch_Trans_TimerValue = pref.getInt("PCH_Trans_Timer
                    ", -1)) == -1) {
1160                pch_Trans_TimerValue = pch_Trans_Timer;
1161                edit.putInt("PCH_Trans_Timer", pch_Trans_Timer);
1162                edit.commit();
1163            }
1164            if ((fastDormancyFlg = pref.getBoolean("FAST_DORMANCY",
                    false)) == false) {
1165                fastDormancyFlg = fast_Dormancy;
1166                edit.putBoolean("FAST_DORMANCY", fast_Dormancy);
1167                edit.commit();
1168            }
1169            if ((fastDormancyRel8Flg = pref.getBoolean("
                    FAST_DORMANCY_REL8", false)) == false) {
1170                fastDormancyRel8Flg = fast_Dormancy_Rer8;
1171                edit.putBoolean("FAST_DORMANCY_REL8",
                        fast_Dormancy_Rer8);
1172                edit.commit();
1173            }
1174            if ((traffic_Inactivity_TimerValue = pref.getInt("
                    Traffic_Inactivity_Timer", -1)) == -1) {
1175                traffic_Inactivity_TimerValue =
                        traffic_Inactivity_Timer;
1176                edit.putInt("Traffic_Inactivity_Timer",
                        traffic_Inactivity_Timer);
1177                edit.commit();
1178            }
```

```
1179            if ((ta_TimerValue = pref.getInt("Ta_Timer", -1)) == -1)
                     {
1180                ta_TimerValue = ta_Timer;
1181                edit.putInt("Ta_Timer", ta_Timer);
1182                edit.commit();
1183            }
1184            if ((drx_Inactivity_TimerValue = pref.getInt("
                    DRX_Inactivity_Timer", -1)) == -1) {
1185                drx_Inactivity_TimerValue = drx_Inactivity_Timer;
1186                edit.putInt("DRX_Inactivity_Timer",
                        drx_Inactivity_Timer);
1187                edit.commit();
1188            }
1189            if ((idle_Trans_TimerValue = pref.getInt("
                    IDLE_Trans_Timer", -1)) == -1) {
1190                idle_Trans_TimerValue = idle_Trans_Timer;
1191                edit.putInt("IDLE_Trans_Timer", idle_Trans_Timer);
1192                edit.commit();
1193            }
1194        }
1195    }
```

B　クラスタ毎のユースケース

B.1　クラスタ 1

表 1　クラスタ 1 のユースケース

性別	年齢	職業	総使用時間	放電時使用時間	ディスプレイ平均輝度
35〜39 歳	男性	自営業	02:14:38	02:02:03	80
カテゴリ	使用時間	起動回数	放電時使用時間	放電時起動回数	放電時使用率
ブラウザ	01:58:13	45	01:50:00	39	93.0%
プリインホーム	00:08:01	36	00:03:58	29	49.5%
メール	00:03:38	46	00:03:38	46	100.0%
ライフスタイル	00:01:22	4	00:01:22	4	100.0%
電卓	00:01:21	2	00:01:21	2	100.0%
通話	00:00:59	1	00:00:59	-	100.0%
写真	00:00:20	5	00:00:00	-	0.0%
未分類	00:00:16	1	00:00:16	1	100.0%
GooglePlay	00:00:13	2	00:00:13	2	100.0%
システム	00:00:12	1	00:00:12	1	100.0%
通話	00:00:04	1	00:00:04	1	100.0%

B.2　クラスタ 2

表 2　クラスタ 2 のユースケース

性別	年齢	職業	総使用時間	放電時使用時間	ディスプレイ平均輝度
25〜29 歳	女性	会社員	02:16:23	02:08:25	80
カテゴリ	使用時間	起動回数	放電時使用時間	放電時起動回数	放電時使用率
ブラウザ	02:04:02	36	01:56:32	34	94.0%
プリインホーム	00:04:25	55	00:04:10	51	94.0%
メール	00:01:56	11	00:01:46	10	91.8%
SNS1	00:01:51	5	00:01:51	5	100.0%
メッセンジャー	00:01:27	13	00:01:27	13	100.0%
ツール	00:01:10	20	00:01:07	19	96.5%
通話	00:01:01	2	00:01:01	2	100.0%
メール	00:00:24	5	00:00:24	5	100.0%
カメラ・写真	00:00:04	1	00:00:04	1	100.0%
ブラウザ	00:00:03	1	00:00:03	1	100.0%
ウィルススキャン	00:00:01	1	00:00:01	1	100.0%

B.3 クラスタ 3

表 3 クラスタ 3 のユースケース

性別	年齢	職業	総使用時間	放電時使用時間	ディスプレイ平均輝度
25〜29 歳	女性	無職	02:43:15	02:39:34	80
カテゴリ	使用時間	起動回数	放電時使用時間	放電時起動回数	放電時使用率
パズル	01:10:29	10	01:10:29	10	100.0%
ブラウザ	00:46:25	16	00:46:25	16	100.0%
メール	00:25:10	43	00:25:10	43	100.0%
メッセンジャー	00:12:14	17	00:10:03	13	82.1%
プリインホーム	00:05:59	44	00:04:29	40	74.9%
写真	00:01:02	3	00:01:02	3	100.0%
カメラ・写真	00:00:48	3	00:00:48	3	100.0%
通話	00:00:24	2	00:00:24	2	100.0%
システム	00:00:15	3	00:00:15	3	100.0%
未分類	00:00:12	2	00:00:12	2	100.0%
健康・フィットネス	00:00:11	3	00:00:11	3	100.0%
ウィルススキャン	00:00:03	3	00:00:03	3	100.0%
システム	00:00:02	1	00:00:02	1	100.0%

B.4 クラスタ 4

表 4 クラスタ 4 のユースケース

性別	年齢	職業	総使用時間	放電時使用時間	ディスプレイ平均輝度
35〜39 歳	女性	会社員	03:12:11	02:53:18	182.5
カテゴリ	使用時間	起動回数	放電時使用時間	放電時起動回数	放電時使用率
ブラウザ	02:17:52	135	02:09:31	116	93.9%
プリインホーム	00:23:30	117	00:17:05	86	72.7%
ソーシャルネットワーク	00:17:12	70	00:15:49	56	91.9%
システム	00:04:20	55	00:04:06	47	94.6%
カメラ・写真	00:02:14	7	00:02:09	6	96.7%
メッセンジャー	00:01:37	11	00:01:14	8	77.0%
カメラ・写真	00:01:06	2	00:01:06	2	100.0%
GooglePlay	00:01:03	2	00:00:03	1	5.5%
メール	00:01:00	13	00:00:22	8	37.4%
メール	00:00:50	3	00:00:50	3	100.0%
通話	00:00:31	2	00:00:31	2	100.0%
時計	00:00:27	1	00:00:27	1	100.0%
ツール	00:00:26	3	00:00:00	-	0.0%
システム	00:00:02	1	00:00:02	1	100.0%
健康・フィットネス	00:00:02	2	00:00:02	2	100.0%

B.5 クラスタ 5

表 5 クラスタ 5 のユースケース

性別	年齢	職業	総使用時間	放電時使用時間	ディスプレイ平均輝度
20〜24 歳	女性	パート・アルバイト	05:10:24	04:20:06	80
カテゴリ	使用時間	起動回数	放電時使用時間	放電時起動回数	放電時使用率
ブラウザ	03:41:57	57	03:14:43	50	87.7%
カジュアル	00:40:05	14	00:25:23	7	63.3%
メッセンジャー	00:18:11	47	00:13:15	35	72.9%
プリインホーム	00:12:45	94	00:10:09	79	79.6%
メール	00:08:09	26	00:07:38	23	93.6%
カジュアル	00:04:19	2	00:04:19	2	100.0%
パズル	00:01:38	1	00:01:38	1	100.0%
メール	00:01:25	19	00:01:25	19	100.0%
通信	00:01:11	4	00:01:11	4	100.0%
カメラ・写真	00:00:14	2	00:00:00	-	0.0%
ライフスタイル	00:00:11	2	00:00:11	2	100.0%
GooglePlay	00:00:06	2	00:00:06	2	100.0%
未分類	00:00:05	2	00:00:05	2	100.0%
ツール	00:00:03	1	00:00:00	-	0.0%
カメラ・写真	00:00:01	-	00:00:00	-	0.0%
ツール	00:00:01	1	00:00:01	1	100.0%
メディア・動画	00:00:01	1	00:00:01	1	100.0%

B.6 クラスタ 6

表 6 クラスタ 6 のユースケース

性別	年齢	職業	総使用時間	放電時使用時間	ディスプレイ平均輝度
55〜59 歳	男性	会社員	00:20:21	00:17:23	80
カテゴリ	使用時間	起動回数	放電時使用時間	放電時起動回数	放電時使用率
プリインホーム	00:12:18	21	00:11:08	17	90.6%
メール	00:02:02	30	00:01:20	17	65.4%
ライフスタイル	00:01:36	3	00:01:36	3	100.0%
通話	00:00:57	2	00:00:57	2	100.0%
ライフスタイル	00:00:55	8	00:00:50	7	90.3%
仕事効率化	00:00:33	4	00:00:02	1	5.9%
システム	00:00:33	12	00:00:31	11	95.1%
通話	00:00:31	7	00:00:06	2	18.3%
通信	00:00:24	2	00:00:24	2	100.0%
ツール	00:00:13	4	00:00:13	4	100.0%
仕事効率化	00:00:10	5	00:00:08	4	78.5%
ブラウザ	00:00:08	1	00:00:08	1	100.0%
ウィルススキャン	00:00:03	2	00:00:01	1	56.4%