コンテキストベース・プロダクトライン開発とVDM++の適用

鵜林尚靖[†1] 金川太俊[†2] 瀬戸敏喜[†2]
中島震[†3] 平山雅之[†4]

本論文では，コンテキストを考慮した組み込みシステム向けプロダクトライン開発手法を提案する．現状では主にシステム構成をどうするかという立場からプロダクトラインが定義されるため，システムとコンテキストの組合せによっては想定外の欠陥が生じる場合がある．本論文では，このような問題を解決するため，システムラインとコンテキストラインの2つからプロダクト仕様を構成する方法を提案する．また，プロダクトラインの仕様をVDM++により記述する方法，およびそれらの妥当性確認方法を示す．

A Context-based Product Line Approach Using VDM++

Naoyasu Ubayashi,[†1] Hirotoshi Kanagawa,[†2] Toshiki Seto,[†2]
Shin Nakajima[†3] and Masayuki Hirayama[†4]

We propose a new product line development method that takes into account the contexts of embedded systems. Most of the current approaches focus on the system configuration only. Unexpected defects might be found in a system due to conflicting combinations of the system and its contexts. In order to deal with this issue, we propose a method for constructing product specifications composed of both system and context lines. Additionally we show how to describe and validate the product line specifications using VDM++.

1. はじめに

本論文では，コンテキストを考慮した組み込みシステムの仕様記述とその妥当性確認方法をプロダクトライン開発の観点から提案する．組み込みシステムの最大の特徴は，センサやアクチュエータなどのハードウェアを通じて，システム自身が外部環境に影響を及ぼす点にある．また同時に，組み込みシステムはハードウェアを通じて外部環境からも影響を受ける．本論文では，外部環境や利用環境などシステムの振舞いに影響を与える現実世界をコンテキストと呼ぶことにする．安全，安心な高信頼性組み込みシステムを構築するには，コンテキストを考慮した開発手法が必要となる．一方，組み込みシステムのもう1つの特徴として，プロダクトライン（製品系列）型開発[4),10),28)]があげられる．プロダクトライン開発とは，あるプロダクト群で共通に利用可能なアーキテクチャやコンポーネントを資産（Asset）としてあらかじめ用意し，個々のプロダクトは資産を合成して開発する手法である．機能に複数のバリエーションがある携帯電話や家電機器などでは，シリーズ製品開発の中でプロダクトライン型の開発形態になっている．プロダクトライン開発では，プロダクト群に求められる機能や特徴を分析することが重要となる．この作業のことをフィーチャ分析（Feature analysis）[20)]という．

現状のプロダクトライン開発では，フィーチャ分析は「ハードウェアやそれを制御するソフトウェアのシステム構成をどうするか」という観点を中心に行われているが，コンテキストを明示的に必ずしも取り扱っていないため，システムとコンテキストの組合せによっては，想定外の欠陥が生じることがある．

本論文では，このような問題を解決するため，新たなプロダクトライン開発の枠組みを提案する．我々は，プロダクトラインをシステムラインとコンテキストラ

†1 九州工業大学情報工学部
Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology
†2 九州工業大学大学院情報工学研究科
Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology
†3 国立情報学研究所アーキテクチャ科学研究系
Information Systems Architecture Research Division, National Institute of Informatics
†4 情報処理推進機構ソフトウェア・エンジアリング・センター
Software Engineering Center, Information-Technology Promotion Agency

インの 2 種類の系列から定義すべきだと考える．システムラインとは，システムを構成するハードウェア，ソフトウェアに対してフィーチャ分析を行った結果として得られる系列であり，一方，コンテキストラインとは，プロダクトが利用される環境に対してフィーチャ分析を行った結果として得られる系列である．個々のプロダクトはシステムライン中の構成要素とコンテキストライン中の構成要素を合成することにより得られる．これにより，想定する環境を明示的に意識したプロダクトライン型開発が可能となる．本論文では，さらに，各構成要素の仕様を形式手法の 1 つである VDM++[7] により記述し，システムを構成するハードウェアとソフトウェア，想定するコンテキストの関係が妥当か否か，すなわち，思わぬ欠陥がないかどうかを，VDM++インタプリタが提供するテスト実行により確かめる方法を提案する．これにより，開発の早い段階でコンテキストを含めた仕様の妥当性確認が可能となる．

本論文では，まず 2 章で簡単な例を用いて，コンテキストの観点から現状の組み込みシステム開発の問題点を述べる．3 章では，この問題を解決する方法として，コンテキストを考慮したプロダクトライン開発を提案する．続く 4 章では，プロダクトラインを構成する資産の仕様を VDM++で記述する方法を提示するとともに，VDM++インタプリタを用いた仕様の妥当性確認について述べる．5 章では，提案手法の適用性と妥当性確認の有効性について評価する．6 章で関連研究を紹介し，7 章で今後の課題について述べる．最後に 8 章でまとめを行う．

2. 問題意識

本章では，電気ポットを例に，従来の組み込みシステム開発における仕様策定過程にどのような問題があるかを示す．そして，外部環境などのコンテキストを考慮する必要性について述べる．

2.1 例題：電気ポット

電気ポットはお湯を沸かす機能を提供する組み込みシステムである．機能が比較的簡単であることから教材として用いられることが多く，その代表的なものが組み込みソフトウェア管理者・技術者育成研究会（SESSAME）[30] の「話題沸騰ポット」である．ここでは，例題仕様として，話題沸騰ポットの仕様のうち，以下のみを考えることにする（**図 1**）．

- ポット内の水を沸騰させる．
- ヒータの On/Off によって水温を制御する．
- 水温が 100 度に達すると，保温状態に移行する．

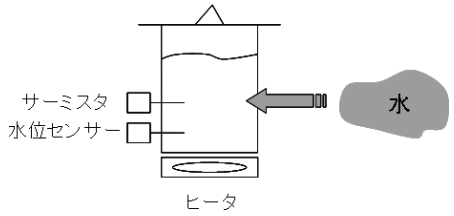


図 1 電気ポットの構成
Fig. 1 Configuration of an electric pot.

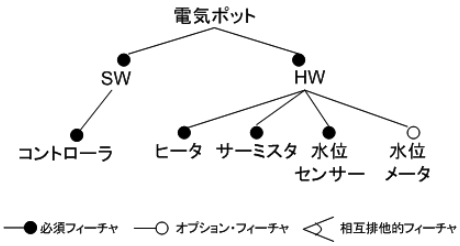


図 2 電気ポットのフィーチャ分析
Fig. 2 Feature analysis for an electric pot.

- 水位センサにより水量を観測する．最下層の水位センサはポットに水が入っているか否かを判定する．ここでは，簡略化して，水位センサは最下層のもの 1 つに限定する．

2.2 プロダクトラインに基づいた仕様策定

電気ポットをプロダクトラインにより開発する場合，まず最初に**図 2** に示すようなフィーチャ分析を行う．すべての機種で必須のフィーチャなのか，ある機種のみで必要となるオプショナルなフィーチャなのか，あるいはどれか 1 つだけ選択されるフィーチャなのかを分析する．個々の機種開発は，商品企画に基づいて，これらのフィーチャから必要なものを選び出すことから始まる．ここでは，加温を制御するソフトウェアである「コントローラ」と 3 つのハードウェア「ヒータ」「サーミスタ」「水位センサ」を選択することにする．

次に，コントローラの仕様を策定する．通常，ソフトウェアの仕様は特定のコンテキストを念頭に記述される．たとえば，上記の電気ポットの仕様では「通常気圧（1 気圧）の環境下で水を沸騰させる」と暗黙的に仮定されている．開発者は想定するコンテキストの像をソフトウェア内部に構築し，それに基づいて必要となるロジックを考える．この場合，「水温が 100 度になるまでヒータを On にし，水を加熱する」というロジックを組み立てることになる．以下は，この機能（Boil）を VDM++により記述したものである．

```
Boil: () ==> ()
Boil() ==
```

```
while thermistor.GetTemperature() < 100.0
  do heater.On();
```

1 行目は Boil のシグニチャ，2 行目以降は Boil の本体である．Boil には引数がなく，返却値もない．この仕様は，サーミスタ（thermistor）を通じて取得した水温（GetTemperature）が 100 度未満である限り，ヒータ（heater）を On にし続けることを示している．

2.3 仕様策定に関わる問題

上記のコントローラ仕様は「通常気圧（1 気圧）の環境下で水を沸騰させる」場合には正しい仕様であるが，想定しているコンテキストが変更になると，コントローラ仕様が仮定するコンテキスト像と実際のコンテキストとの間にズレが生じ，それが最終的なシステム上の欠陥につながってしまう．たとえば「低気圧（1 気圧未満）の環境下で水を沸騰させる」場合（気圧の低い山頂での利用など）を考慮したとき，前述の Boil では電気ポットに求められる機能を提供できなくなる．2.1 節で提示した仕様自体は満たしているが，結果的にコンテキストに合致した仕様にはなっていない．1 気圧未満では水の沸点は 100 度を下回るため，電気ポットは沸騰後も加熱し続け，最後には水がなくなる．最下層の水位センサが水がないことを観測し，電気ポットは加熱を停止する．

図 2 のように，システムを構成するハードウェアとソフトウェアのみを考慮したフィーチャ分析では，ここで述べたような問題に対処することが難しく，以下のような課題がある．

- 仕様の再利用性が低い：コンテキストを明示せずにロジックを決めてしまうことが多く，仕様を再利用することが難しい．
- システムの保守，発展がアドホック的：想定するコンテキストが変わるたびに仕様を見直すが，その対応が場当たり的になってしまうことが多い．システムの改良が重なると，仕様の一貫性を保つことが難しくなる．
- 仕様の妥当性確認が困難：上記 2 点の結果として，システムとコンテキストの間に生じる不整合や欠陥を見落とす可能性が高くなる．組み込みシステムの場合，仕様はハードウェア，ソフトウェア，想定するコンテキスト間のトレードオフを念頭に判断しなければならないため，仕様の妥当性確認はより困難になる．ハードウェアを追加すべきか，あるいはソフトウェアで頑張るべきかにより仕様が変わり，そのための妥当性確認が必要となる．

2.4 本研究の狙い

本論文では，上記の問題に対処するため，コンテキストを考慮したプロダクトライン型の仕様記述手法を提案する．この手法は以下の特長を持つ．

- プロダクトラインをシステムラインとコンテキストラインに分け，各々フィーチャ分析を行う．
- システムラインを構成するハードウェアおよびソフトウェア仕様，コンテキストラインを構成するコンテキスト仕様が資産になる．個々のプロダクトの仕様は，これらを組み合わせて定義する．
- 上記の各仕様記述に VDM++を適用する．妥当性確認は VDM++のテスト実行により行う．

3. コンテキストを考慮したプロダクトライン

本章では，コンテキストを考慮したプロダクトラインの考え方と開発手順について述べる．

3.1 プロダクトラインの構成

本論文が提案するプロダクトラインは，システムラインとコンテキストラインの 2 つから構成される．システムラインとは，システムを構成するハードウェア，ソフトウェアに対してフィーチャ分析を行った結果として得られる系列である．様々なハードウェアフィーチャとソフトウェアフィーチャの組合せにより製品のバリエーションを構成することができる．一方，コンテキストラインとは，プロダクトが利用される環境に対してフィーチャ分析を行った結果として得られる系列である．考慮すべきコンテキストの組合せも製品系列と同じように扱おうという考え方である．さらに，どのようなコンテキストを考慮すべきかをプロダクト開発時のノウハウとして資産化する狙いもある．従来のプロダクトラインでは，前者のシステムラインからの視点が主であり，対象コンテキストが明示されないという問題があった．すなわち，コンテキストを意識したとしても，その情報がシステムラインの中に埋没してしまうため，資産化することが難しかった．

図 3 は，電気ポットのシステムラインとコンテキストラインを示したものである．電気ポットを利用する想定コンテキストが明示的に表現されているのが分かる．たとえば，2 章の例に見落とされていた「1 気圧未満での利用」や「ポットの中にミルクを入れた場合の利用」がフィーチャ分析の中に含まれている．

個々のプロダクト仕様はシステムラインとコンテキストラインの構成要素を選択することにより得られる．想定環境が「通常気圧（1 気圧），水の加温」の場合は，システムラインから「コントローラ（ソフトウェア）」「ヒータ（ハードウェア）」「サーミスタ（ハード

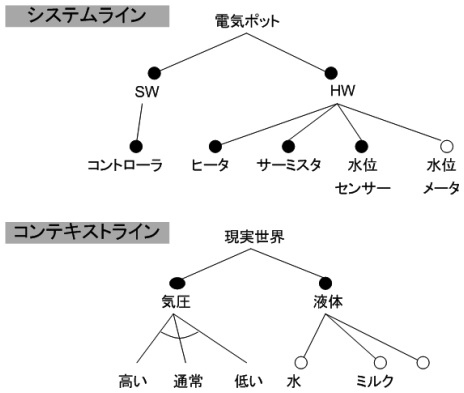


図 3 システムラインとコンテキストライン
Fig. 3 System line and context line.

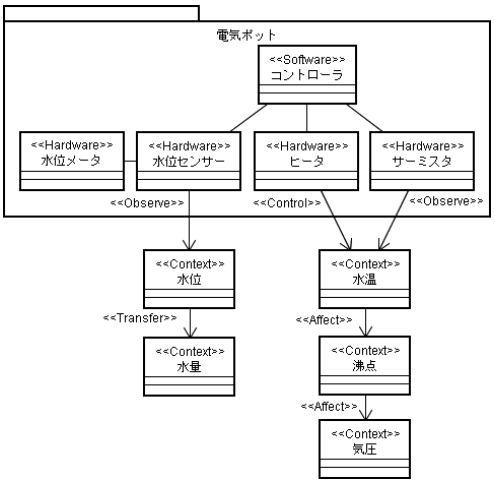


図 4 電気ポットの外部環境分析
Fig. 4 External environment analysis for an electric pot.

ウェア)」「水位センサ（ハードウェア)」が，コンテキストラインから「気圧–通常」「液体–水」が選択される．

3.2 プロダクトライン開発の手順

CMU（Carnegie Mellon University)/SEI（Software Engineering Institute）による SPLP（A Framework for Software Product Line Practice)[28] では，プロダクトラインの主要アクティビティは，コア資産（Core Asset）開発プロセス，プロダクト開発プロセス，管理プロセスの 3 つから構成される．コア資産開発とは，プロダクト系列のドメインを分析して再利用可能なコア資産を開発するプロセスである．ドメインエンジニアリングとも呼ばれる．プロダクト開発とは，コア資産を用いて実際のプロダクトを開発するプロセスである．アプリケーションエンジニアリングとも呼ばれる．管理プロセスは，コア資産開発とプロダクト開発のための要員配置，調整，監視を行うアクティビティである．

本論文の手法では，それぞれ以下のようにマッピングされる．

コア資産開発プロセス

(1) コンテキスト資産の開発：コンテキストのフィーチャ分析，仕様資産の開発を行う．
(2) システム資産の開発：ハードウェアとソフトウェアのフィーチャ分析，仕様資産の開発を行う．

プロダクト開発プロセス

(1) 想定コンテキストの定義：コンテキスト資産群からプロダクトが想定する仕様資産を選択する．
(2) プロダクト構成の定義：システム資産群からプロダクトを構成する仕様資産を選択する．
(3) 仕様の妥当性確認：ハードウェア，ソフトウェア，コンテキストの各仕様を組み合わせた結果に不整合がないか否かを VDM++のテスト実行により確かめる．不整合が発生した場合は，仕様資産の選択方法に問題ないか，既存の仕様資産を変更する必要がないか，確かめる．

SPLP とは，1) コンテキストそのものもコア資産に含める，2) プロダクト開発のアクティビティに形式手法による妥当性確認を追加する，ところが異なる．一方，管理プロセスは基本的に SPLP と同様であるが，「コンテキスト資産の開発者とシステム資産の開発者を別々にするのか」「別々にした場合はどのように両者の間で調整したらよいのか」などについて留意する必要がある．

3.3 コンテキストラインの構築方法

本論文が提案する手法を実践するには，システムラインとコンテキストラインをどのように分けるかが鍵となる．本論文では，この目的のために筆者らが提案する外部環境分析手法[18),32)~34)] を適用する．外部環境分析手法では，対象となる組み込みシステムがどのような要素から影響を受け，どのような構造を持っているかを明確にする．

システムとコンテキストの切り分け

図 3 に示したようなフィーチャ木を作成するには，まず，個々の製品のフィーチャをシステムとコンテキストに切り分ける必要がある．その後，抽出したフィーチャ群の共通部分と変動部分を分析し，プロダクトラインとしてのフィーチャ木を完成させる．後者は通常のプロダクトライン構築手法と基本的に同じなので，ここでは前者についてその方法を述べる．

図 4 は電気ポットに対して外部環境分析を適用した結果である．図の上側はシステム，下側はコ

表 1 外部環境記述用 UML プロファイル
Table 1 A UML profile for describing the external environments.

名前	適用する要素	定義
≪*Context*≫	Class	組み込みシステムの動作に影響する外部環境の要素を意味する．システムに対する外部からの入力やそれに影響を与える要素がこれに分類される．
≪*Hardware*≫	Class	組み込みシステムを構成するハードウェアを意味する．
≪*Software*≫	Class	組み込みシステムを構成するソフトウェアを意味する．
≪*Observe*≫	Association	ハードウェアが外部環境の要素を観測する関係を意味する．
≪*Control*≫	Association	ハードウェアが外部環境の要素を制御する関係を意味する．
≪*Noise*≫	Association	ハードウェアが外部環境の要素を観測する際に，観測の対象以外の要素が混ざった状態で入力されることがある．このときのハードウェアと観測の対象以外の要素との関係を意味する．
≪*Transfer*≫	Association	ハードウェアが外部環境の要素を観測する際に，直接的に観測することができず，別の要素に変換しなければならない場合がある．このときの要素どうしの変換の関係を意味する．
≪*Affect*≫	Association	外部環境の要素が変換される際に，ハードウェアが観測したい要素とは別の要素が影響を与える場合がある．このときの変換後の要素と影響を与える要素の関係を意味する．

ンテキストである．システムとコンテキストの境界には，コンテキストを観測，制御するハードウェアを配置する．図 4 では，**表 1** に示す外部環境記述用 UML プロファイル[31] を使用している．このプロファイルは，システムおよび外部環境を構成する要素とそれらの関連に関する拡張を定義したものである．具体的には，クラスに対する拡張として，≪*Context*≫，≪*Hardware*≫，≪*Software*≫ の 3 種類のステレオタイプを，関連に対する拡張として，≪*Observe*≫，≪*Control*≫，≪*Noise*≫，≪*Transfer*≫，≪*Affect*≫ の 5 種類のステレオタイプを定義している．なお，≪*Observe*≫，≪*Control*≫ の矢印は観測および制御先を，≪*Noise*≫，≪*Affect*≫ の矢印は影響元を，≪*Transfer*≫ の矢印は変換元をそれぞれ指している．

外部環境分析を適用する準備として，まずはシステムの構成要素を商品企画書あるいはシステム仕様書などからピックアップし，それらの関係を図 4 の上側のように記述する．その後，後述の方法に従って，システムが考慮しなければならないコンテキストを抽出していく．

システム資産の抽出

システムの構成要素として，コントローラ，ヒータ，サーミスタ，水位センサ，水位メータが抽出されており，これらはそのまま図 3 に示したシステムラインのフィーチャ木を構成するための入力となる．

コンテキスト資産の抽出

コンテキストラインを構成するには，その構成要素であるコンテキスト資産を抽出する必要がある．電気ポットの例では，液体や気圧をどのように切り出すかが求められる．

外部環境分析手法では，まず最初に，ハードウェア（≪*Hardware*≫）によって直接観測（≪*Observe*≫）あるいは制御（≪*Control*≫）されるコンテキスト（≪*Context*≫）を抽出する．電気ポットの場合は「水位」と「水温」がこれに該当する．「水位」は水位センサにより観測され，「水温」はサーミスタにより観測，ヒータにより制御される．

次にこれらのコンテキストすなわち「水温」と「水位」の状態に影響を与える影響要因をガイドワード[22] を適用して洗い出す．ガイドワードとは，関連する事柄を導き出す際に使用されるヒントとなる言葉であり，外部環境分析手法では次の 5 つを使用する．

- 上限を決定する要因
- 下限を決定する要因
- 特定の値に対応する要因
- システムによる観測を阻害する要因
- システムによる制御を阻害する要因

影響要因の分析には 2 つのステレオタイプ ≪*Noise*≫ と ≪*Affect*≫ を用いる．たとえば，「水温」に関する影響要因分析では，水温の「上限を決定する要因」というガイドワードを参考にすることで，水温は沸点より高くはならないという事実から，上限を決定する要因として「沸点」という影響要因を導き出すことができる．この一次的な影響要因に対して，さらにガイドワードを適用して，二次的な影響要因を導く．「沸点」に対して「特定の値に対応する要因」というガイドワードを適用すると，水の沸点は，1 気圧で 100 度となることから，「気圧」という二次的な影響要因を導き出すことができる．

初期段階で洗い出したコンテキストの中には，システムが本来観測したいものではなく代替的なものである場合がある．すなわち，ハードウェアが元々のコンテキストの値を直接観測できない場合である．電気ポットの場合，システムが本来観測したいのは「水量」であり，「水位」ではない．外部環境分析手法では，変

表 2 VDM++仕様の構成
Table 2 A set of specifications described in VDM++.

仕様の区分	VDM++仕様の命名規則 (仕様記述ファイル名)	内容
テスト仕様	UserTest	テスト実行用仕様
	RealWorld	テスト用コンテキスト設定
システムライン (SW)	SYSTEM-SW-controller	コントローラ
システムライン (HW)	SYSTEM-HW-heater	ヒータ
	SYSTEM-HW-thermistor	サーミスタ
	SYSTEM-HW-liquid-level-sensor	水位センサ
コンテキストライン	CONTEXT-atmospheric-air-pressureplace	気圧
	CONTEXT-atmospheric-air-pressureplace-high	1 気圧より大
	CONTEXT-atmospheric-air-pressureplace-normal	1 気圧
	CONTEXT-atmospheric-air-pressureplace-low	1 気圧未満
	CONTEXT-liquid	液体
	CONTEXT-liquid-water	水

換（≪*Transfer*≫）という関連でこれを表現する．

最終的に考慮しなければならないコンテキストは，ハードウェア（≪*Hardware*≫）によって直接観測（≪*Observe*≫）あるいは制御（≪*Control*≫）されるコンテキスト（≪*Context*≫）から変換（≪*Transfer*≫）をたどった最終コンテキストとそれに影響（≪*Noise*≫と ≪*Affect*≫）を与える最終コンテキストである．電気ポットの場合，前者については「水量」と「水温」が，後者については「気圧」が考慮しなければならないコンテキストになる．

外部環境分析手法では，値を持つものをコンテキストとして抽出している．そのため，「水量」と「水位」は別々に抽出されるが，これらは実際には「水」というコンテキストのプロパティである．コンテキストラインでは，「水」（実際にはそれをさらに一般化した「液体」）をコンテキスト資産として扱うことにする．

以上の手順により，図 3 に示したコンテキストラインの 2 つの資産「液体」と「気圧」を抽出することが可能となる．

4. VDM++による仕様記述と妥当性確認

VDM++は VDM-SL（The Vienna Development Method – Specification Language）[6),8),37)] のオブジェクト指向拡張で，ソフトウェアの厳密なモデル化を目的とした仕様記述言語である．VDM++のための開発環境として，VDM++ Toolbox [38)] と呼ばれるツールが提供されている．このツールは，仕様の構文/型チェック，証明課題生成，インタープリタなどの機能を持つ．

本章では，VDM++をソフトウェアのみに限定せず，ハードウェアやコンテキストにも適用する．

4.1 仕様記述とプロダクトラインへの割当て

VDM++仕様の構成方法

表 2 は VDM++による仕様記述を電気ポットのプロダクトラインに割り当てたものである．VDM++仕様はプロダクトラインの区分に合わせて記述される．具体的には，システムラインのソフトウェア/ハードウェア仕様は SYSTEM-SW-/SYSTEM-HW-，コンテキストラインの仕様は CONTEXT-という命名規則で作成する．これにより，仕様資産の管理がやりやすくなる．各 VDM++仕様の具体的な記述は付録を参照されたい．

図 5 は，電気ポットのためのテスト仕様，システム仕様，コンテキスト仕様の関係を示したものである．なお，テスト仕様はシステム仕様の内容が妥当か否かを確認するためのもので，後述のテスト実行の際に使用される．テスト仕様（UserTest）の中では，コントローラ（SYSTEM-SW-controller）のセットアップと電気ポット利用環境（RealWorld）の設定を行うとともに「水の沸騰（Boil）」をコントローラに指示している．コントローラには，水温が 100 度に達するまでヒータ（SYSTEM-HW-heater）を On にするロジックが組み込まれている．その際，「ポットの中身が空でない」ことが事前および事後条件として記載されている．ヒータはコントローラからの指示で，コンテキスト中に存在する水（Context-liquid(-water)）を加熱する．水の VDM++仕様には「熱が加えられるたびに 1 度だけ水温が高くなり，沸点に達しても熱せられ続けると蒸発する」と記載されている．

コア資産の記述方法

システムラインおよびコンテキストラインの個々の仕様は，資産としての再利用性を考慮し，独立性が保たれるように記述する．図 5 を見て分かるように，個々の VDM++仕様は「関心事の分離（Separation

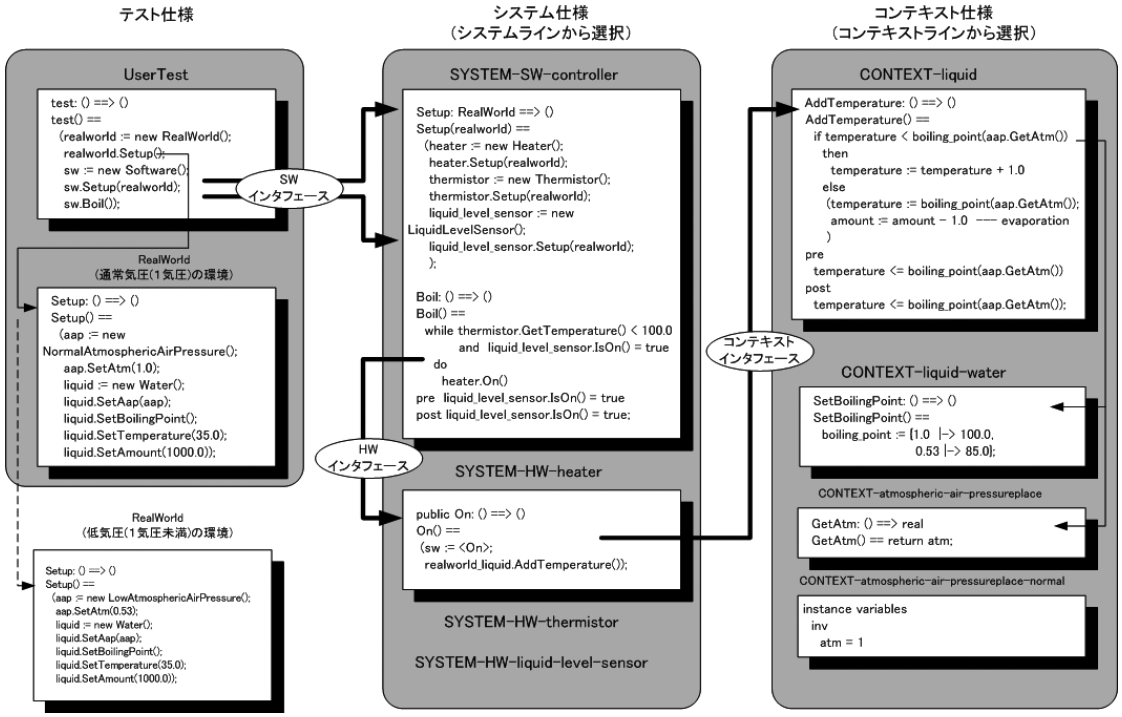


図 5 電気ポットの仕様記述
Fig. 5 Specification descriptions for an electric pot.

of concerns)」の原理に基づいて定義される．コンテキストの記述にはソフトウェアやハードウェアに関わるものはいっさい含まれず，コンテキストそのもののフィーチャのみが記述される．同様にハードウェアの記述にはソフトウェアに関するものは含まれない．一方，コンテキストを観測，制御するハードウェアは，コンテキストインタフェースを通じてのみコンテキストにアクセスできる．ソフトウェアインタフェース，ハードウェアインタフェースについても同様である．

システムラインの各仕様には，資産として提供すべき機能を VDM++のクラスとして記述する．具体的な機能はクラスを構成する操作（operation）として与える．各機能を利用するにあたっての前提条件を事前条件（pre 文）として，その機能を利用した結果として得られる保証を事後条件（post 文）として記述する．また，資産の利用前後で変化しない性質を不変条件として記述する．資産の本体は陽関数（explicit function）で記述する．陽関数とは結果を計算する式を陽に関数本体として与えるものである．

VDM++による仕様資産は，継承機構を利用してフレームワーク化することができる．たとえば，水の仕様（CONTEXT-liquid-water）は液体の仕様（CONTEXT-liquid）のサブ仕様である．後者に「液体」に共通なフィーチャを記述し，前者に「水」に特化したフィーチャ，たとえば水の気圧と沸点の関係などを記述する．

本論文の方式で特徴的なのは，システムラインの仕様だけでなくコンテキストラインの仕様も機能に基づいて記述する点にある．たとえば，コントローラ（ソフトウェア）を通じてヒータ（ハードウェア）で水（コンテキスト）を加温すると，「水は水温を上げる」と解釈する．水の仕様（CONTEXT-liquid-water）にはそのための機能が操作として記載されている．簡単にこの様子を見ていくことにする．以下はコントローラの Boil 仕様である．この中では，事前事後条件として，水位センサが On になっていること，すなわち，加温するにはポットの水が空でないことが設定されている．

```
// コントローラ（ソフトウェア）の仕様記述
public
Boil: () ==> ()
Boil() ==
  while thermistor.GetTemperature() < 100.0 and
        liquid_level_sensor.IsOn() = true
    do heater.On()
pre  liquid_level_sensor.IsOn() = true
post liquid_level_sensor.IsOn() = true;
```

Boil は水温が 100 度未満の間はヒータのスイッチ

を On にし続ける．その間，ヒータは水を温め続ける．以下はその記述である．

```
// ヒータ（ハードウェア）の仕様記述
public On: () ==> ()
On() ==
 (sw := <On>;
  realworld_liquid.AddTemperature());
```

上記の結果として，水温が実際に上昇する．以下は水の仕様である．ここでは簡単のため，沸点に到達後も加温すると 1 cc ずつ蒸発するように記述している．

```
// 水（コンテキスト）の仕様記述
public
AddTemperature: () ==> ()
AddTemperature() ==
  if temperature < boiling_point(aap.GetAtm())
     then
       temperature := temperature + 1.0
     else
      (temperature := boiling_point(aap.GetAtm());
        amount := amount - 1.0  --- evaporation
      )
pre  temperature <= boiling_point(aap.GetAtm())
post temperature <= boiling_point(aap.GetAtm());
```

4.2 プロダクト開発プロセス

3.2 節で説明したプロダクト開発プロセスを電気ポットの例で説明する．

想定コンテキストの定義

付録に示したコンテキスト資産群からプロダクトが想定する仕様資産を選択する．ここでは，通常気圧（1 気圧）の環境下で水を沸騰させる場合を想定する（以下，コンテキスト A と呼ぶことにする）．この場合，CONTEXT-atmospheric-air-pressureplace-normal と CONTEXT-liquid-water が選択される．また，これらのスーパクラスである CONTEXT-atmospheric-air-pressureplace と CONTEXT-liquid も同時に選択される．

プロダクト構成の定義

次にシステム資産群からプロダクトを構成する仕様資産を選択する．ここでは，付録のシステム資産すべてを選択することにする．すなわち，SYSTEM-SW-controller，SYSTEM-HW-heater，SYSTEM-HW-thermistor，SYSTEM-HW-liquid-level-sensor の 4 つを選択する．これらは電気ポットに最低限求められる必須資産である．

仕様の妥当性確認

電気ポットの仕様の妥当性を VDM++インタプリタのテスト実行により確認する．まず最初にコンテキスト A の具体的な設定を行う（加温前の初期水温など）．これを行うのが RealWorld という VDM++仕様である．また，テスト実行を行うための仕様 UserTest も用意する．この仕様の中では現実世界をインスタンス化するとともに，ソフトウェアモジュール（SYSTEM-SW-controller）に対し「沸騰させる（Boil）」という操作を実行させる．

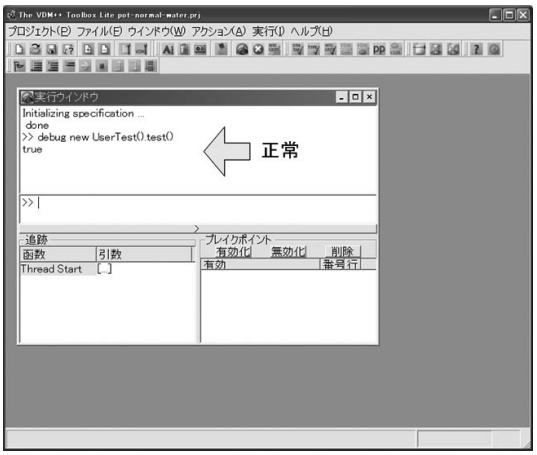


図 6 テスト実行結果（コンテキスト A）
Fig. 6 Result of test execution (context A).

図 7 テスト実行結果（コンテキスト B）
Fig. 7 Result of test execution (context B).

コンテキスト A でテスト実行すると**図 6** に示すように正常に終了する．すなわち，100 度に達したら，加温を停止する．しかしながら，上と同様な手順で，低気圧（1 気圧未満）の環境下で水を沸騰させる場合（コンテキスト B と呼ぶ）でテスト実行すると**図 7** に示すようにエラーとなる．

コンテキスト B の場合は，コンテキスト資産から CONTEXT-atmospheric-air-pressureplace-low と CONTEXT-liquid-water が選択されるが，システム側の仕様は両コンテキストで差異はない．しかしながら，コンテキスト B の場合は以下のように Boil の事後条件違反となり，仕様が妥当でないことが

分かる．

```
post liquid_level_sensor.IsOn() = true;
```

```
[VDM++インタプリタのメッセージ]
```

```
Run-Time Error 59:
  The post-condition evaluated to false
```

システム側の仕様記述は両者とも同一であるが，コンテキストが異なるとテスト実行の結果に差異が出ている．後者の場合，100 度未満で沸点に達するため，沸騰後も過熱し，最終的に水がすべて蒸発してしまい，事後条件を満たさなくなってしまう．

外部環境要素とシステム機能の組合せによる欠陥がテスト実行により発見された場合，システムとしてどのような仕様にすべきか（ハードウェアを追加すべきか，あるいはソフトウェアで頑張るべきか），その妥当性を調べることが可能となる．コンテキスト B に対応するには，沸点（気圧により変化）に応じた加温操作が必要となるため，気圧センサが必須となる．

ここで示した妥当性確認の方法に対して，2 つ議論すべき点がある．1 つは，上記の例は簡単な場合しか扱っておらず，考慮すべきコンテキストとして気圧を認識した段階で実際には問題に気づいている場合が多いという点である．したがって，わざわざテスト実行して確かめる必要性を感じないかもしれない．しかし，考慮すべきコンテキストは通常複数あり，それらの関係も単純ではない．そのような場合，開発者の頭の中だけで仕様上の不具合を発見することは容易ではなく，ここで述べたようなテスト実行は有効である．VDM++のテスト実行は仕様上のバグを取り除くためのデバッグツールと考えると分かりやすい．もう 1 つは，どうやればコンテキストとして気圧を認識できるかという問題である．そもそも考慮すべきコンテキストとして気圧が認識できなければ，テスト実行を行っても仕様上の不具合を発見することはできない．これについては，本論文では 3.3 節でその方法を提示した．従来は，開発経験を積むことによってしかプロダクトラインの資産を充実させることができなかった．

5. 評　　価

2 章で，コンテキストに起因する従来の組み込みシステム開発における問題点として，1) 仕様の再利用性が低い，2) システムの保守，発展がアドホック的，3) 仕様の妥当性確認が困難，の 3 点を指摘したが，本論文が提案する手法を適用することによりこれらの問題を軽減させることが可能になる．1) の問題は，コンテキストに関する仕様をシステム仕様から分離することにより解決される．2) の問題は，プロダクトラインをコンテキストラインとシステムラインに分離することにより解決される．3) の問題は，VDM++仕様記述とテスト実行による妥当性確認により解決される．

本章では，提案手法を適用性と妥当性確認の有効性の 2 つの側面から評価する．また，本論文の主題である「コンテキストの分離と明確化」について考察する．

5.1 適　用　性

筆者らは，提案手法の適用性を評価するため，電気ポット以外の組み込みシステムに対して 4 章で提示した手順を適用した．用いた例はライントレーサである．ライントレーサとは，白いコースに引いた黒いラインに沿って走行する自律制御ロボットで，その動作仕様は，車体直下部の光の反射率が低い（コースが黒，すなわちライン上にいる）とき右へ，反射率が高い（コースが白，すなわちライン外にいる）とき左に進行するというものである．

ライントレーサのコア資産は，電気ポットの場合と同様，表 2 のカテゴリ，すなわち，テスト仕様，システムライン（SW/HW），コンテキストラインに区分して容易に記述することができた．このカテゴリは，システムおよびコンテキストのフィーチャを分類するのに有効に利用できた．仕様の妥当性についても VDM++のテスト実行を用いて確認することができた．たとえば，外部から強い光を受けた場合，車体直下部の光の反射率に異常をきたしてしまい，正常にラインをトレースすることができなくなることを検出できた．

これらの経験から，提案手法は組み込みシステムに対して十分幅広く適用可能だと考えられる．ただし，VDM++による仕様記述はあくまでも機能中心であり，機能以外の側面に着目したい場合は別の形式手法を適用する必要がある．その場合でも，本論文が提案するコンテキストラインの考え方そのものは適用可能である．

5.2 妥当性確認の有効性

4 章で示したように本論文が提案する妥当性確認方法は有用であるが，その品質は VDM++のテスト実行に依存している．したがって，通常のプログラムテスト同様，テスト仕様の網羅率が重要となる．今回の VDM++仕様は機能ベースで記述されるため，テスト仕様も機能ベースの方が望ましい．すなわち，以下のような手順を踏むのが良いと思われる．

(1) システムに求められる機能をユースケース分析などで網羅的に洗い出す．
(2) 洗い出した機能をベースにテスト仕様を

VDM++で記述する．

(3) システム仕様とコンテキスト仕様を組み合わせたものに対し，テスト仕様を適用する．

形式手法の分野では，現在，Formal Methods Lite[17] や Lightweight Formal Methods[13] が注目されている．前者は本質的な側面を厳密に表現するための道具として形式手法を利用しようという考え方である．したがって厳格な証明などは必ずしも求められない．後者はこれに加えて自動化を強調したものである．本論文で使用した VDM++ Toolbox は前者の Formal Methods Lite の一例である．後者の例としては，Jackson らが開発した Alloy[1] などがある．

本論文で「検証」ではなく「妥当性確認」という用語を使用したのは，伝統的な形式手法の証明アプローチではなく，Formal Methods Lite の考え方をベースにしているからである．伝統的な形式手法と違って，健全性（soundness）も完全性（completeness）も保証されないが，比較的低コストで仕様の正しさを確認することができる．本論文の手法は，Formal Methods Lite のプロダクトライン開発への応用例と考えることができる．

5.3 コンテキストを分離することの利点

2 章で説明したように，組み込みシステム開発においてコンテキストを意識することは重要である．実際，既存手法の中にもプロダクトラインの開発時に何らかの形でコンテキストに言及しているものがある．たとえば，KobrA[2] では，開発の最初に対象システムのコンテキストを分析してコンテキスト実現モデルを作成する方法を提案している．KobrA ではコンテキスト実現モデルの段階から可変性/不変性を扱えるため，コンテキストラインの考え方と類似する部分がある．また，Kang らは，フィーチャをケーパビリティ（capability）層，オペレーティング環境（operating environment）層，ドメイン技術（domain technology）層，実装技法（implementation technique）層の 4 層にカテゴライズする方法を提案しているが[19]，最近では，利用コンテキスト（usage context）の視点を導入することの重要性を特に指摘している．このようにプロダクトラインに関する研究において，コンテキストは重要なテーマになりつつある．

しかしながら，既存研究の多くは，単にコンテキストの重要性を指摘するにとどまっている．また，コンテキストを取り扱ったとしても，システムのフィーチャ木の一部としての分析にとどまる場合が多く，本論文のように明示的にコンテキストそのものを扱う手法は少ない．KobrA の場合でも，コンテキストはシステム対象内の問題領域をシステム外と同時にとらえるものであり，システムとコンテキストを明確に分離する本論文のアプローチとは異なる．

これに対し，本論文の手法では，コンテキストに対するフィーチャ分析を主要事項として扱っている．また，システムとコンテキストの切り分け方法，コンテキストラインの構築方法を具体的に提示している．

筆者らは，システムの延長線上にコンテキストをとらえる方式よりも両者を分離した方がプロダクトライン構築には有利だと考えている．理由は，複数の製品群（あるいは本論文でいうシステムライン）で共通的に扱えるコンテキスト資産が存在すると思われるからである．もし，コンテキストがシステムのフィーチャ分析に従属するのであれば，このような場合に対処できない．ただし，両者のより深い評価については，定量的評価とともに今後の課題である．

6. 関 連 研 究

現実世界（Real world）をモデルの対象とする研究は古くから存在する．たとえば，Greenspan らは要求仕様の記述に現実世界の知識を織り込むことの重要性について言及している[11]．Jackson は，問題フレーム（Problem Frame）[15] の中で，機械（システム）と現実世界との関連をモデルとして表現する方法を提案している．本論文でいうコンテキストの記述は現実世界のモデリングに対応する．個別のプロダクト開発だけではなく，プロダクトラインとしても現実世界を考えていくことは重要である．

現実世界のモデリングを組み込みシステム開発に適用する試みもいくつか存在する．組み込みシステムは通常のビジネスシステムと比較して，高度な信頼性と安全性が求められるため，本論文の中でも述べたように，分析段階で可能な限り障害シナリオを網羅的に抽出することが重要となる．筆者らは，組み込みシステムとその外部の状態を組み合せることにより，例外的な動作条件を分析する手法を提案している[18),32)~34)]．また，Mise らは，マトリクスを用いて障害分析する手法を提案している[24),25)]．これらは人手による分析手法であり，形式手法をベースとする本論文の手法とは相補的な関係にある．両者を組み合わせて使用すると効果的である．本論文では形式手法を用いて，システムだけでなく，現実世界を表すコンテキストのモデルを厳密に記述し，その妥当性を確認する方式をとっている．しかしながら，確認できるのは記述したモデル内に不整合がないかどうかだけであり，モデルの記述範囲が妥当か否かまでは判断できない．たとえば，

プロダクトの安全性を高めるのにどこまでをそのプロダクトの外部環境として含めるべきかの判断は，形式手法による妥当性確認の範囲外である．この部分は要求獲得にかかわる部分であり，筆者らや Mise らの分析手法が有効である．開発プロセスの観点からいうと，事前に筆者らや Mise らの分析手法を用いて記述すべき範囲を明確にした後に，本論文が提案する手法を適用するのが望ましいと思われる．実際，3.3 節で述べたコンテキストラインの構築方法は筆者らの手法をベースとしている．

システムの安全性[22] の観点からプロダクトラインを構成するための研究はすでにいくつか存在する．たとえば，Dehlinger らはプロダクトラインに SFTA（Software Fault Tree Analysis）を適用する方法を提案している[5]．また，SFTA と SFMECA（Software Failure Modes, Effects and Criticality Analysis）手法を統合して適用する方法も提案している．Liu らは，この研究をさらに発展させてプロダクトラインの安全性分析に SFTA と状態ベースモデリングを利用する方法を提案している[23]．

システムの特徴のうち，本論文では，システム内の並行プロセスの存在とリソース競合の問題，動作のタイミング依存性，などへの対応方法については述べなかった．VDM++ではスレッドなどの並行性を扱えるが，モデル検査などとの連携が必要となる．たとえば，三好らは，VDM-SL で記述されたモデルから FSP（Finite State Processes）[26] で記述されたモデル検査可能な状態遷移モデルを抽出する方法を提案している[27]．

形式手法を用いてシステムの仕様を厳密に記述する試みは数多く行われているが[9]，これらは個別のプロダクトをモデル化するにとどまっており，本論文のようにコンテキストを明示的に考慮した形式化は少ない．Sun らは，フィーチャモデリングのための形式検証手法を提案している[35]．一階述語論理でフィーチャの意味を形式的に記述し，それを Z/EVES 定理証明器[29] で検証する．さらに，フィーチャモデルの一貫性とその構成を Alloy 解析器[1),14] で検証する．また，Höfner らはフィーチャを代数的に取り扱うためのフィーチャ代数（Feature algebra）[12] を提案している．ただし，フィーチャモデルそのものを形式的に扱う試みはまだ研究が緒についた段階にとどまっている．

7. 今後の課題

本章では，提案手法の将来の発展方向について，開発プロセスやアスペクト指向などの観点から考察する．

7.1 開発プロセス

本論文が提案する方法は，仕様化フェーズだけでなく，実装までを含めて適用するとより有効だと考えられる．現在，モデリング言語として最も普及しているのは UML である．したがって，UML と本手法を組み合わせて利用できるプロセスが望ましい．たとえば，実装言語として Java を考えた場合，1) UML でモデル化した資産を VDM++に変換しテスト実行によりその妥当性を確認する，2) VDM++からコードを生成する際，事前条件，事後条件，不変条件を JML（Java Modeling Language）[16] に変換し，それを ESC/Java2[3] などの JML ツールを用いて検証する，といったプロセスが考えられる．

現在のモデル駆動開発では，UML からコードを生成する部分に重きが置かれる傾向が強いが，上記プロセスでは，コード生成そのものよりは仕様段階で妥当性を確認した事前条件，事後条件，不変条件をいかに最終的なプログラムコードに反映させるかを重視している．すなわち，UML から VDM++，さらには Java に至る追跡性（Traceability）は，事前条件，事後条件，不変条件を軸に実現される．UML からコードを生成することは実際には難しく，多くの場合はスケルトンの生成にとどまっている．そのため，最終的なプログラムはスケルトンにコードを追加したものになるが，必ずしもそれが OCL（Object Constraint Language）[39] などで記述された UML モデルの制約条件を反映しているとは限らない．上記プロセスは，このような問題を考慮した結果である．

7.2 陽関数と陰関数

VDM の関数には陽関数（explicit function）と陰関数（implicit function）があるが，両者とも一長一短があり，どちらを使用すべきかは一概に判断できない．前者が結果を計算する式を陽に関数本体として与えるのに対し，後者は計算アルゴリズムを与えずに結果の性質のみを事前条件と事後条件で示す．仕様を陽関数として記述するとインタプリタによるテスト実行が可能になるが，その反面，仕様と実装の差異が曖昧になる．そのため，仕様記述という意味では陰関数を用いる方が望ましいが，逆にテスト実行できないため妥当性を確認するのが難しくなる．この場合，証明支援ツールを用いて仕様を検証する必要がある．

筆者らは仕様記述において最も重要なのは不変条件などの制約条件を特定化することだと考えている．一般に不変条件は抽出するのが難しく，テスト実行しながら見つけていくことは有効である．そのため，本論文では仕様を陽関数で記述する方式を採用した．しか

し，いったん制約条件を洗い出し，その妥当性を確認した後は関数本体は必ずしも必要でない．仮に関数本体がプログラムコードに変換できたとしても，仕様と実装では自ずと求められる記述レベルが異なる．むしろ，VDM++によるモデル化は制約条件を洗い出すための工程と見なすこともできる．

7.3 アスペクト指向の導入

電気ポットの場合，沸点を考慮したシステムを構築するには，ハードウェアとソフトウェアのバランスを考慮して新たな仕様を決める必要があるが，この場合，1) 気圧センサを追加する，2) 沸点に関するコンテキストの像をソフトウェアの中に構築する，などの修正が必要となる．これらを，元のソフトウェア仕様にそのまま反映すると，修正箇所が横断的に様々な箇所に散らばってしまい，ソフトウェアの保守，再利用という側面から見た場合，好ましくない．

このような問題を解決する方法として，我々はVDMにアスペクト指向を導入する研究を進めている[36]．アスペクト指向とは，横断的関心事をモジュール化するための技術である[21]．システムラインの資産には基本的な仕様しか記述せず，想定するコンテキストの像はアスペクトによりコンテキストラインの該当資産から取り込む．コンテキストラインの資産がプロダクト開発にも使用されるところが重要となる．

8. ま と め

本論文では，外部環境などのコンテキストを考慮した組み込みシステムの仕様記述とその妥当性確認方法をプロダクトライン開発の観点から提案した．本論文では組み込みシステムを例に述べたが，通常のビジネスシステムに対しても本手法は適用可能である．安全，安心なシステム構築にはコンテキストの視点が今後ますます重要になってくると考えられる．

謝辞 本論文の執筆にあたり，貴重なご助言を賜りました（株）東芝ソフトウェア技術センターの鷲見毅氏に感謝の意を表します．

参 考 文 献

1) Alloy. http://alloy.mit.edu/
2) Atkinson, C., et al.: *Component-Based Product Line Engineering with the UML*, Addison-Wesley (2001).
3) Cok, D. and Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML, *Proc. Program Analysis for Software Tools and Engineering* (*PASTE 2004*) (2004).
4) Clements, P. and Northrop, L.: *Software Product Lines: Practices and Patterns*, Addision-Wesley (2001).
5) Dehlinger, J. and Lutz, R.: Software Fault Tree Analysis for Product Lines, *Proc. 8th IEEE International Symposium on High Assurance Systems Engineering* (*HASE 2004*), pp.12–21 (2004).
6) CSK: VDMTools — The VDM-SL Language. http://www.vdmtools.jp/files/langmansl_a4E.pdf
7) CSK: VDMTools — The CSK VDM++ Language. http://www.vdmtools.jp/files/langmanpp_a4E.pdf（英語版），http://www.vdmtools.jp/files/langmanpp_a4J.pdf（日本語版）．
8) Fitzgerald, J. and Larsen, P.G.: *Modeling Systems, Practical Tools and Techniques in Software Development*, Cambridge University Press (1998).
9) Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N. and Verhoef, M.: *Validated Designs for Object-oriented Systems*, Springer Verlag (2005).
10) Greenfield, J.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, John Wiley & Sons Inc. (2003).
11) Greenspan, S., Mylopoulos, J. and Borgida, A.: Capturing More World Knowledge in the Requirements Specification, *Proc. International Conference on Software Engineering* (*ICSE'82*), pp.225–234 (1982).
12) Höfner, P., Khedri, R. and Möller, B.: Feature Algebra, *Proc. 14th International Symposium on Formal Methods* (*FM 2006*), pp.300–315 (2006).
13) Jackson, D. and Wing, J.: Lightweight Formal Methods, *IEEE Computer*, Vol.29, No.4, pp.21–22 (1996).
14) Jackson, D.: *Software Abstractions*, The MIT Press (2006).
15) Jackson, M: *Problem Frame*, Addison-Wesley (2001).
16) Java Modeling Language (JML). http://www.cs.iastate.edu/leavens/JML/
17) Jones, C.B.: A Rigorous Approach To Formal Methods, *IEEE Computer*, Vol.29, No.4, pp.20–21 (1996).
18) 金川太俊，瀬戸敏喜，鵜林尚靖，鷲見 毅，平山雅之：組込みシステムにおける外部環境分析の提案，第 8 回組込みシステム技術に関するサマーワークショップ SWEST8，pp.75–82 (2006).
19) Kang, K.C., Kim, S., Lee, J., Shin, E. and Huh, M.: FORM: A Feature-oriented Reuse

Method with Domain-specific Reference Architecture, *Annals of Software Engineering*, Vol.5, pp.143–168 (1998).
20) Kang, K.C., Lee, J. and Donohoe, P.: Feature-Oriented Product Line Engineering, *IEEE Software*, Vol.9, No.4, pp.58–65 (2002).
21) Kiczales, G., et al.: Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming* (*ECOOP'97*), pp.220–242 (1997).
22) Leveson, N.G.: *Safeware: System Safety and Computers*, Addison-Wesley Publishing Company (1995).
23) Liu, J., Dehlinger, J. and Lutz, R.: Safety Analysis of Software Product Lines Using State-Based Modeling, *Proc. 16th IEEE International Symposium on Software Reliability Engineering* (*ISSRE 2005*), pp.21–30 (2005).
24) Mise, T., Hashimoto, M., Katamine, K., Shinyashiki, Y., Ubayashi, N. and Nakatani, T.: An Analysis Method with Failure Scenario Matrix for Specifying Unexpected Obstacles in Embedded Systems, *Proc. 12th Asia-Pacific Software Engineering Conference* (*APSEC 2005*), pp.447–456 (2005).
25) Mise, T., Shiniashiki, Y., Nakatani, T., Ubayashi, N., Katamine, K. and Hashimoto, M.: A Method for Extracting Unexpected Scenarios of Embedded Systems, *Proc. Joint Conference on Knowledge-Based Software Engineering* (*JCKBSE 2006*) (2006).
26) Magee, J. and Kramer, J.: *Concurrency: State Models & Java Programs*, John Wiley & Sons (Worldwide Series in Computer Science) (1999).
27) 三好健吾，日下部茂，荒木啓二郎：データ型に着目した形式仕様記述からの状態遷移系の抽出，コンピュータソフトウェア，Vol.23, No.2, pp.211–224 (2006).
28) SEI: Product Line Approach to Software Development. http://www.sei.cmu.edu/plp/
29) Saaltink, M.: The Z/EVES system, *ZUM'97: Z Formal Specification Notation*, Bowen, J.P., Hinchey, M.G. and Till, D. (Eds.), Vol.1212 of Lecture Notes in Computer Science, pp.72–85, Springer-Verlag (1997).
30) SESSAME（組込みソフトウェア管理者・技術者育成研究会）. http://www.sessame.jp/
31) 瀬戸敏喜，金川太俊，鵜林尚靖，鷲見　毅，平山雅之：組込みシステムの外部環境分析のためのUML プロファイル，組込技術とネットワークに関するワークショップ ETNET2007，2007-EMB-4, pp.65–70 (2007).
32) 鷲見　毅，平山雅之，鵜林尚靖：組込みシステムにおける外部環境の分析，情報処理学会ソフトウエア工学研究会 SE-146，pp.33–40 (2004).
33) 鷲見　毅，平山雅之，鵜林尚靖：組込みシステムにおける動作条件分析手法の提案，電子情報通信学会ソフトウエアサイエンス研究会 SIG-SS-2005-36，pp.19–24 (2005).
34) 鷲見　毅，平山雅之，鵜林尚靖：組込みシステムの動作環境の特徴に着目した仕様分析手法の提案，情報処理学会組込みシステム研究会 EMB-1, pp.7–12 (2006).
35) Sun, J., Zhang, H., Fang, Y. and Wang, L.H.: Formal Semantics and Verification for Feature Modeling, *Proc. 10th IEEE International Conference on Engineering of Complex Computer Systems* (*ICECCS 2005*), pp.303–312 (2005).
36) Ubayashi, N. and Nakajima, S.: Context-aware Feature-Oriented Modeling with an Aspect Extension of VDM, *Proc. 22nd Annual ACM Symposium on Applied Computing* (*SAC 2007*), pp.1269–1274 (2007).
37) VDM-SL base language, ISO/IEC 13817-1 (1996).
38) VDMTools. http://www.vdmtools.jp/
39) Warmer, J. and Kleppe, A.: *The Object Constraint Language, 2nd Edition—Getting Your Models Ready for MDA*, Addison Wesley (2003).

付　　録

A.1 テスト仕様

A.1.1 UserTest

```
class UserTest

instance variables
  realworld : RealWorld;
  sw : Software;

operations
  public
  test: () ==> bool
  test() ==
    (realworld := new RealWorld();
     realworld.Setup();
     sw := new Software();
     sw.Setup(realworld);
     sw.Boil();
     return true);

end UserTest
```

A.1.2 RealWorld

コンテキスト A：通常気圧（1 気圧）の環境下で水を沸騰させる場合

```
class RealWorld

instance variables
  public aap: NormalAtmosphericAirPressure;
  public liquid : Water;
```

```
operations
  public
  Setup: () ==> ()
  Setup() ==
    (aap := new NormalAtmosphericAirPressure();
     aap.SetAtm(1.0);            //1 気圧
     liquid := new Water();
     liquid.SetAap(aap);
     liquid.SetBoilingPoint();
     liquid.SetTemperature(35.0);//初期水温は 35 度
     liquid.SetAmount(1000.0));  //初期水量は 1000cc

end RealWorld
```

コンテキスト B：低気圧（1 気圧未満）の環境下で水を沸騰させる場合

```
class RealWorld

instance variables
  public aap: LowAtmosphericAirPressure;
  public liquid : Water;

operations
  public
  Setup: () ==> ()
  Setup() ==
    (aap := new LowAtmosphericAirPressure();
     aap.SetAtm(0.53);           //0.53 気圧
     liquid := new Water();
     liquid.SetAap(aap);
     liquid.SetBoilingPoint();
     liquid.SetTemperature(35.0);//初期水温は 35 度
     liquid.SetAmount(1000.0));  //初期水量は 1000cc

end RealWorld
```

A.2 システムラインの仕様資産

A.2.1 SYSTEM-SW-controller

```
class Software
instance variables
  heater : Heater;
  thermistor : Thermistor;
  liquid_level_sensor : LiquidLevelSensor;

operations
  public
  Setup: RealWorld ==> ()
  Setup(realworld) ==
    (heater := new Heater();
     heater.Setup(realworld);
     thermistor := new Thermistor();
     thermistor.Setup(realworld);
     liquid_level_sensor := new LiquidLevelSensor();
     liquid_level_sensor.Setup(realworld);
     );

  public
  Boil: () ==> ()
  Boil() ==
    while thermistor.GetTemperature() < 100.0 and
          liquid_level_sensor.IsOn() = true
      do heater.On()
  pre  liquid_level_sensor.IsOn() = true
  post liquid_level_sensor.IsOn() = true;
end Software
```

A.2.2 SYSTEM-HW-heater

```
class Heater
types
  Switch = <On> | <Off>;

instance variables
  sw : Switch;
  realworld_liquid : Liquid;

operations
  public Setup: RealWorld ==> ()
  Setup(realworld) ==
    realworld_liquid := realworld.liquid;

  public On: () ==> ()
  On() ==
   (sw := <On>;
    realworld_liquid.AddTemperature());

  public Off: () ==> ()
  Off() ==
    sw := <Off>;
end Heater
```

A.2.3 SYSTEM-HW-thermistor

```
class Thermistor
instance variables
  realworld_liquid : Liquid;

operations
  public
  Setup: RealWorld ==> ()
  Setup(realworld) ==
    realworld_liquid := realworld.liquid;

  public
  GetTemperature: () ==> real
  GetTemperature() ==
    return realworld_liquid.GetTemperature();
end Thermistor
```

A.2.4 SYSTEM-HW-liquid-level-sensor

```
class LiquidLevelSensor
instance variables
  realworld_liquid : Liquid;

operations
  public
  Setup: RealWorld ==> ()
  Setup(realworld) ==
    realworld_liquid := realworld.liquid;

  public
  IsOn: () ==> bool
  IsOn() ==
    return realworld_liquid.GetAmount() > 0;
end LiquidLevelSensor
```

A.3 コンテキストラインの仕様資産

A.3.1 CONTEXT-atmospheric-air-pressureplace

```
class AtmosphericAirPressure

instance variables
  protected atm : real;

operations
  public
  GetAtm: () ==> real
  GetAtm() ==
    return atm;

  public
  SetAtm: real ==> ()
  SetAtm(a) ==
    atm := a;
end AtmosphericAirPressure
```

A.3.2 CONTEXT-atmospheric-air-pressureplace-normal

```
class NormalAtmosphericAirPressure
  is subclass of AtmosphericAirPressure

instance variables
  inv atm = 1
end NormalAtmosphericAirPressure
```

A.3.3 CONTEXT-atmospheric-air-pressureplace-high

```
class HighAtmosphericAirPressure
  is subclass of AtmosphericAirPressure

instance variables
  inv atm > 1
end HighAtmosphericAirPressure
```

A.3.4 CONTEXT-atmospheric-air-pressureplace-low

```
class LowAtmosphericAirPressure
  is subclass of AtmosphericAirPressure

instance variables
  inv atm < 1
end LowAtmosphericAirPressure
```

A.3.5 CONTEXT-liquid

```
class Liquid
instance variables
  protected aap : AtmosphericAirPressure;
  protected boiling_point : map real to real;
  protected temperature : real;
  protected amount : real;

operations
  public
  GetAap: () ==> AtmosphericAirPressure
  GetAap() ==
    return aap;

  public
  SetAap: AtmosphericAirPressure ==> ()
  SetAap(a) ==
    aap := a;

  public
  GetBoilingPoint: real ==> real
  GetBoilingPoint(atm) ==
    return boiling_point(atm);

  public
  GetTemperature: () ==> real
  GetTemperature() ==
    return temperature;

  public
  SetTemperature: real ==> ()
  SetTemperature(t) ==
    temperature := t;

  public
  AddTemperature: () ==> ()
  AddTemperature() ==
    if temperature < boiling_point(aap.GetAtm())
      then
        temperature := temperature + 1.0
      else
       (temperature := boiling_point(aap.GetAtm());
        amount := amount - 1.0  --- evaporation
       )
  pre  temperature <= boiling_point(aap.GetAtm())
  post temperature <= boiling_point(aap.GetAtm());

  public
  GetAmount: () ==> real
  GetAmount() ==
    return amount;

  public
  SetAmount: real ==> ()
  SetAmount(a) ==
    amount := a;
end Liquid
```

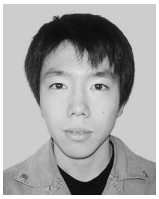
A.3.6 CONTEXT-liquid-water

```
class Water is subclass of Liquid
operations
  public
  SetBoilingPoint: () ==> ()
  SetBoilingPoint() ==
    boiling_point := {1.0  |-> 100.0, 0.53 |-> 85.0};
end Water
```

(平成 18 年 11 月 20 日受付)
(平成 19 年 5 月 9 日採録)

鵜林 尚靖（正会員）

1960 年生．1982 年広島大学理学部数学科卒業．1999 年東京大学大学院総合文化研究科広域科学専攻広域システム科学系博士課程修了．博士（学術）．1982〜2003 年（株）東芝に勤務．2002〜2003 年芝浦工業大学システム工学部非常勤講師．2003 年九州工業大学情報工学部助教授，現在に至る．2003 年度本学会山下記念研究賞受賞．ソフトウェア工学，プログラミング言語モデルに興味を持つ．ACM，IEEE-CS，日本ソフトウェア科学会，電子情報通信学会各会員．



金川 太俊

1983 年生．2006 年九州工業大学情報工学部知能情報工学科卒業．現在，同大学大学院情報工学研究科情報科学専攻博士前期課程に在籍．

瀬戸 敏喜

1983 年生．2006 年九州工業大学情報工学部知能情報工学科卒業．現在，同大学大学院情報工学研究科情報科学専攻博士前期課程に在籍．

中島　震（正会員）

1979 年東京大学理学部物理学科卒業．1981 年東京大学大学院理学系研究科修士課程修了．NEC 研究開発グループ，法政大学を経て，2004 年より国立情報学研究所教授．2005 年より総合研究大学院大学教授を併任．この間，オレゴン大学客員研究員，科学技術振興機構さきがけ・SORST 研究員兼務，北陸先端科学技術大学院大学 JJREX 客員教授，日本ソフトウェア科学会理事等．2000 年学術博士（東京大学）．2001 年度本学会山下記念研究賞，2003 年度日本ソフトウェア科学会論文賞を受賞．形式手法，形式検証，ソフトウェア・モデリングの研究に従事．日本ソフトウェア科学会，ACM 各会員．

平山　雅之（正会員）

1986 年東芝入社．2003 年より東海大学電子情報学部非常勤講師．2005 年より IPA/SEC 組込みプロジェクト領域責任者兼務．現在，（株）東芝ソフトウェアエンジニアリングセンター参事，東海大学専門職大学院客員教授（組み込み技術専攻），IPA/SEC 領域責任者として組み込みソフトウェアに関するエンジニアリング技術の研究と普及に従事．専門はソフトウェア品質・信頼性．