Vol.2018-SE-198 No.10
2018/3/9

安全性分析の結果を補完したドメインモデルに基づくコード変換の実現

岩田 紘成[1,a)]　大森 洋一[1,b)]　荒木 啓二郎[1,c)]

概要：ドメイン分析において安全性要求を明確にし, ドメインモデルを構築し利活用する事で, ソフトウェアの品質や再利用性の向上が期待できる. しかし, 安全性分析は実施者の技量に左右されるという問題や, 分析結果が自然言語で表されることにより解釈の齟齬が生じるといった問題がある. 本研究ではこのような問題に対し, 顧客の要求からハザード分析手法である STAMP/STPA により安全性要求を抽出し, 形式仕様言語である VDM-SL を用いてドメインモデルに反映させた後, Python によるトランスレーターを用いてコード生成を行うことで, 分析で得た安全性要求を下流の開発工程の成果物に反映させる手法を提案する. 本稿では, 提案手法の事例への適用を通じて, 分析により得られた安全制約を満たした VDM-SL モデルからトランスレーターによって生成されたプログラムが安全制約を満たしていることを確認する手順を示す.

キーワード：ドメイン分析, 形式手法, ハザード分析, コード生成

Code Generation based on Domain Model compensated for Safety Requirements

Hironari Iwata[1,a)]　Yoichi Omori[1,b)]　Keijiro Araki[1,c)]

Abstract: A domain model built by domain analysis with clean safety requirements is expected to help improving software quality and reusability. However, there were two problems. One was that safety analysis depend on the skill of the analysis. Another was that misinterpretation may occur from the ambiguities of a nature language. An effective use of domain models reflecting the result of safety analyses in order to solve these problems in this research. First, we extracted safety requirements from customer's requirements using a hazard analyses method STAMP/STPA and reflect the result to the domain model using a formal specification language VDM-SL. Next, the domain model was translated into the executable code by a code translation implemented in Python. We show procedure to confirm the generated code are satisfying the safe conditions with a case.

Keywords: Domain Analysis, Formal Methods, Hazard Analysis, Code Generation

1. はじめに

要求分析手法の 1 つであるドメイン分析 [8] は, ドメイン固有の要素や振る舞い, 要素間の関係等を明らかにし, 必ずしも十分に記述されるとは限らない顧客からの要求を補完にすることを目的としている. ドメイン分析においてドメイン固有の安全性要求を明確にし, モデルに反映させることでドメインモデルの品質や再利用性を向上できる. システムにおける安全性に関する問題の原因の多くは不十分な要件定義に起因するものだからである [4]. 本研究では顧客の要求からハザード分析手法である STAMP/STPA により安全性要求を抽出し, 形式仕様言語である VDM-SL を用いてドメインモデルに反映させた後, Python によるトランスレーターを用いてコード生成を行うことで, 分析で

[1] 九州大学, Kyushu University, 744 Motooka Nishi-ku Fukuoka 819-0395, Japan
[a)] h.iwata@nanotsu.ait.kyushu-u.ac.jp
[b)] yomori@ait.kyushu-u.ac.jp
[c)] araki@csce.kyushu-u.ac.jp

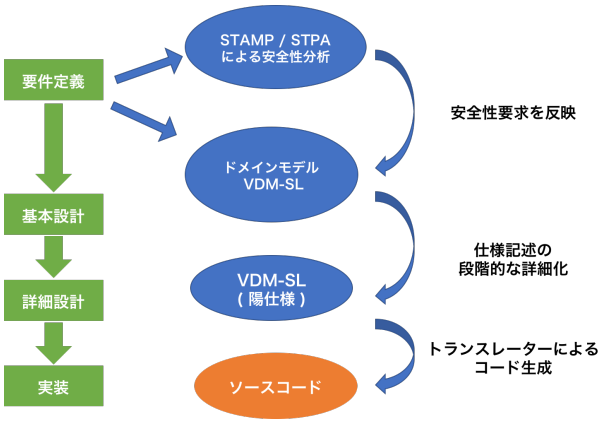

図 1 提案手法を適用した要件定義から実装までの流れ
Fig. 1 Development process applying the method

得た安全性要求を下流の開発工程の成果物に反映させる手法を提案する．手法を適用した開発の流れは**図 1** のようになる．

2. STAMP/STPA の概要

STAMP（Systems-Theoretic Accident Model and Process/システム理論に基づく事故モデル）は，システム論を利用した事故モデルであり，システムを構成するコンポーネント間の相互作用に着目してシステム全体の流れを表す事故モデルとなっている．このため，STAMP では事故は単純なコンポーネントの故障のみを原因とせず，コンポーネントの振る舞いやコンポーネント間の相互干渉が，システムの安全制約を違反した場合に起こると考える．STPA（System Theoretic Process Analysis）[3] は STAMP を利用したハザード分析手法であり，事故が起きる前に事故を引き起こす潜在的な要因を見つけ出すことを目的としている．文献 [9] に従い，ここでは大きく分けて以下の 4 つの順序に従って STPA の分析を行う．

2.1 準備 1：アクシデント，ハザード，安全制約の識別

準備 1 ではアクシデント（損失を伴う，システムの事故），ハザード（アクシデントにつながるシステムの状態），安全制約（システムが安全に保たれるために必要なルール）の 3 つを作成する必要がある．これらはシステムが回避すべき事象を事前に設定することで目的に沿ったハザード分析を行うためのものである．

2.2 準備 2：コントロールストラクチャの作成

準備 2 ではコントロールストラクチャを作成する．コントロールストラクチャは，システムを制御する各機能の相関関係を示した図である．コンポーネント間でやり取りされる制御の指示やフィードバックなどを矢印で結んで表す．各コンポーネントはコントローラー（Controller）と呼ばれ，その機能や役割に応じて階層構造になっている．上位のコントローラーからはコントロールアクション（Control Action）と呼ばれる指示が出され，センサーやアクチュエータなど下位のコントローラー（被コントロールプロセス：Controlled Process）からはフィードバック（Feedback）として情報が送られる．さらに各コントローラーには，そのコントローラーがどのような処理や指示を行うかというプロセスモデル（Process model）が含まれている．これらのコントローラー間のやり取りを表したものをコントロールループと呼ぶ．

2.3 STPA Step1：安全でないコントロールアクションの識別

STPA Step1 では安全でないコントロールアクション（Unsafe Control Action：UCA）の識別を行う．UCA の識別は，ハザードにつながる恐れのあるコントロールアクションの不具合を明確にすることを目的としており，大きく分けて，以下の 4 つの種類に分類される．

- 「Not Providing」与えられないとハザード
  - 安全のためのコントロールアクションが与えられない．
- 「Providing」与えられるとハザード
  - ハザードにつながる恐れのある，安全ではないコントロールアクションが与えられる．
- 「Wrong Timing/Order」早すぎ、遅すぎ、誤順序でハザード
  - コントロールアクションのタイミングが遅すぎる，早すぎる，または定められた順序に設置していない．
- 「Stopping Topp Soon / Applying Too Long」早すぎる停止、長すぎる適用でハザード
  - コントロールアクションがすぐに止まる，もしくは適用が長すぎる．

上記 4 つの種類以外に 5 番目のシナリオとして，「要求されたコントロールアクションが提供されているが，それに従っていない」という UCA が考えれる．この原因については，コントロールループ内での不具合や遅れなど不適切な動作が含まれる．5 番目のシナリオに関しては，STPA Step2 で分析していくことになる．

2.4 STPA Step2 Hazard Causal factor（誘発要因）の特定

STPA 最後の段階として，STPA Step1 で識別した UCA の原因となる Hazard Causal factor（HCF）と，予想される事故シナリオの特定を行う．HCF の特定には，UCA の引き金になる 11 個のガイドワードを使って，コントロールストラクチャ内の各コントロールループを分析していく．このプロセスでは，UCA に対してガイドワードの適用を行い，どのようにハザードが発生するかという部分にまで詳細に分析を進める必要がある．

Step1 での 4 つの標準的な分類（ガイドワード）をコントロールストラクチャに適用し, コントロールループの基本コンポーネントを調べて, 誤った操作を分類し, その標準の不適切な制御の原因となりうるかを決定する. 11 個のハザード要因（ガイドワード）は以下の通りである.

(1) コントロール入力や外部情報の誤りや喪失.
(2) 不適切なコントロールアルゴリズム.
（作成時の欠陥, プロセスの変更, 誤った修正や適用）
(3) 不整合, 不完全, または不正確なプロセスモデル. 不適切な操作.
(4) コンポーネントの不具合. 経年による変化.
(5) 不適切なフィードバック, あるいはフィードバックの喪失. フィードバックの遅れ.
(6) 不正確な情報の供給, または情報の欠如.
測定の不正確性. フィードバックの遅れ.
(7) 操作の遅れ.
(8) 不適切または無効なコントロールアクション, コントロールアクションの喪失.
(9) コントロールアクションの衝突. プロセス入力の喪失または誤り.
(10)未確認または範囲外の障害
(11)システムにハザードを引き起こすプロセス入力.

3. VDM-SL の概要

VDM-SL は, 形式手法 である VDM の形式仕様記述言語の 1 つである. VDM-SL では厳密な文法, 意味論をもつ言語を用いてシステムの仕様を抽象的にモデル化・記述することで, システムのデータ構造や状態, 振る舞いを明確にすることを目的としている.

3.1 制約条件の記述

VDM-SL は, 契約による設計（Design By Contract：DbC）[5] における事前条件（Pre Condition）と事後条件（Post Condition）, 不変条件（Invariant Condition）を仕様, 設計のレベルで記述することができる. VDM-SL におけるそれぞれの定義を以下に述べる [10].

事前条件（Pre Condition）
「関数や操作が評価される直前に保持しているべき条件」

事後条件（Post Condition）
「関数や操作が評価された直後に保持されているべき条件」

不変条件（Invariant Condition）
「型に対してシステムが存在する限り必ず成立すべき条件」

これらの制約条件を記述することで, 関数や操作で実現すべき振る舞いを明確にすることができ, また, 後述する仕様アニメーションを用いた検証において, 望まない入力や, 出力をエラーという形で確認することができる.

3.2 VDMPad を用いた仕様検証

本研究では, VDM-SL モデルの検証に VDMPad [6] を用いた. VDMPad は, 記述したモデルの構文検査や型検査, 仕様アニメーションと呼ばれるインタプリタによる対話的実行といった機能を備えている VDM-SL の Web IDE である. このツールを用いることにより, 記述したモデルの振る舞いを検証することができる.

4. トランスレーターを用いたコード生成

本研究では, 先述したような開発の実装工程において, VDM-SL モデルからプログラミング言語の 1 つである Python[1] で記述されたソースコードを生成するトランスレーターを作成し, 活用した. トランスレーターによってソースコードを生成することで, 人の手が介入せず, VDM-SL モデルから一貫性のあるプログラムが生成できる.

4.1 トランスレーターの構成

本研究で作成したトランスレーターは, 以下のような工程を経ることで, 入力として受け取った VDM-SL ファイルから Python のソースコードを生成する.

(1) VDM-SL の仕様を入力として受け取り, 構文解析器によって抽象構文木 (以下, AST) へと変換する.
(2) VDM-SL の AST から Python の AST へと変換する.
(3) Python の AST から Python のソースコードを生成する.

以下に, 各工程の実装方法を簡単に述べる.

4.1.1 VDM-SL の AST への変換工程

この工程では, 入力として受け取った VDM-SL ファイルを VDM-SL の AST へと変換する. 本研究では, 一般的な構文解析ツールである lex, yacc を Python で実装したライブラリである PLY[2] を活用することで構文解析器を実装した. PLY は David M. Beazley 氏によって開発された Python で記述された構文解析ツールである. PLY は, 字句解析を行う lex.py と 構文解析を行う yacc.py の 2 つのモジュールから構成されており, この 2 つのツールを組み合わせる事で, 構文解析器を実装できる.

4.1.2 Python の AST への変換

この工程では, 入力として受け取った VDM-SL の AST を, Python の AST へと変換する. Python には Python の抽象構文木を扱うためのモジュールである ast module が標準で用意されており, 本研究では, VDM-SL の AST の各ノードのクラスから Python の ast module で実装されている Python の AST のノードクラスへと対応付けを行い, 変換するプログラムを実装した.

4.1.3 Python のソースコードへの変換

この工程では, 入力として受け取った Python の AST を, Python のソースコードへと変換する. 本研究では, ast モジュールで表現 AST から Python のソースコードの生成機能をもつオープンソースのライブラリである astor[7] を用いて変換を行なった. astor は, Berker Peksag 氏によって開発されたライブラリであり, AST を用いた Python のソースコードの操作を容易にすることを目的として開発された. 本研究では, このライブラリの Python の AST から Python のコードを生成する機能を利用し, トランスレーターに組み込んだ.

4.2 Python における事前条件・事後条件式の検証

VDM-SL では操作や関数に対し, 事前条件, 事後条件といった入出力に関する制約条件を記述する事で, DbC における制約条件を仕様, 設計段階で記述することができる. 本研究で作成したトランスレーターは, VDM-SL における事前条件・事後条件式を生成される Python のソースコードに反映させることで, DbC に基づいた検証を可能とした. 具体的には Python における assert 文を用いて, 事前条件, 事後条件に関する検証を行うことで, 事前条件, 事後条件に違反した際, エラーとして検出できるように実装を行なった. assert 文は, プログラム内にデバッグ用のアサーションを仕掛けるための文法であり, 文中で指定した条件式が False となる時, AssertionError として検出される. 本研究ではこの機能を利用し, 事前条件, 事後条件を assert 文の条件式に指定し, 関数の呼び出し前と呼び出し後に検証を行う事で, 事前条件, 事後条件に違反していないかを確認し, 違反している場合はエラーとして検出するように実装した.

5. 提案手法の事例適用と考察

本節では, 提案手法を具体的に駐車場の入場手続きシステム（以下, Entry System ）に適用した結果について述べる.

5.1 Entry System の概要

Entry System は, 駐車場管理業務のうち, 車両の入場手続きを行うシステムである. ここでは, 本研究で想定する Entry System の概要について述べる.

5.1.1 Entry System の構成要素

システムの構成要素とその役割を以下に述べる.

Vehicle

Vehicle は, 実際に駐車場を利用する車両であり, 入場に必要な手続きを行うアクションを Entry Controller に対し発行し, 手続きを進める

Entry Controller

Entry Controller は, 入場手続きの管理を行うコントローラーであり, Vehicle のアクションに対し, 適切なアクションを Parking Deck , Entry Gate に発行することで入場を管理する

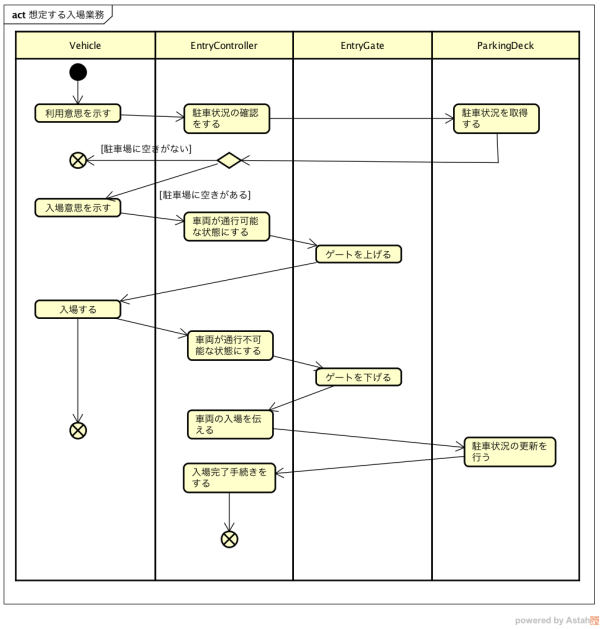

図 2 想定する入場業務
Fig. 2 Business flow of Entry System

Parking Deck

Parking Deck は, 駐車場の状況を管理するコンポーネントであり, Entry Controller からアクションに対し, 現在の駐車台数のフィードバックや更新を行う

Entry Gate

Entry Gate は, 駐車場内と駐車場外の領域を隔てるための物理的なゲートである. ゲートが通行不可能な状態では, Vehicle は駐車場内に入場できないものとする.

5.1.2 想定する入場手続き業務

本研究で想定する駐車場の営業状態としては以下のような状態を想定している.

通常運転中

「車両が入場手続きを行える状態」

点検中

「不正な入場手続きや, コンポーネントの故障といった場合に点検を行うため, 一時入場業務を停止する状態」

作業中

「駐車場の点検といった常にゲートを通行可能な状態としておく必要がある状態」

停止中

「営業時間外といった入場業務を行わず, 車両が利用することができない状態」

また, 通常運転中状態の駐車場における入場手続き業務は以下の**図 2** のような流れとなる.

表 1 Entry System のアクシデント， ハザード， 安全制約
Table 1 Accidents,Hazards,Safety constraints of Entry System

アクシデント	ハザード	安全制約
入場した車両が駐車するスペースがない	車両の入場がシステムに認識されない状態	車両の入場はシステムに認識されなければならない

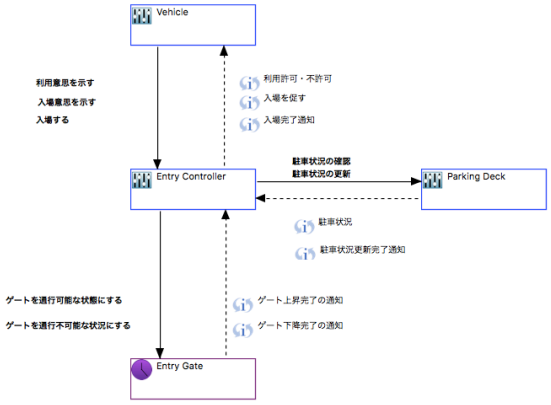

図 3 Entry System のコントロールストラクチャ
Fig. 3 Control Structure of Entry System

5.2 STAMP/STPA を用いた安全性要求の抽出

STAMP/STPA のハザード分析手順に沿って，Entry System が満たすべき安全性に関する要求を分析していく際に，実際に行なった手順と内容を以下に述べる．本分析では，Entry System の状態遷移にのみ着目し，コントロールアクションは発行されると同時に被コントロールプロセスに届くものと考える．

5.2.1 アクシデント，ハザード，安全制約の識別

この段階では Entry System のアクシデントとハザード，安全制約の識別を行なった．実際に作成したアクシデント，ハザード，安全制約を**表 1** に示す．

5.2.2 コントロールストラクチャの構築

この段階では，Entry System のコントロールストラクチャを作成する．実際に構築したコントロールストラクチャを**図 3** に示す．

5.2.3 STPA Step1．UCA の識別

STPA Step1 では UCA の識別を行なった．実際に適用した結果を**表 2** に示す．

5.2.4 STPA Step2 HCF の特定

STPA Step2 では，ガイドワードを参考に Step1 で識別した UCA の潜在要因（HCF）の分析を行い，UCA に繋がるハザードシナリオ（HS）とそれから導ける Entry System が満たすべき新しい安全制約（New Safety Constraint:NSC）の作成を行なった．また，NSC の VDM-SL モデルへの反映方法の検討も行なった．実際に得られた HS，NSC，VDM-SL モデルへの反映方法 を**表 3** に示す．

5.3 分析結果に基づいた VDM-SL モデリング

ハザード分析によって得た UCA，HCF，NSC に基づき，

表 2 STPA Step1 分析結果
Table 2 Result of STPA Step1

Controller	Controlled Process	Control Action	Not Providing causes hazard	Providing causes hazard	Too early/too late, wrong order causes hazard	Stopping too soon/applying too long causes hazard
Vehicle	Entry Controller	利用意思を示す			利用許可を得ている状態で，利用意思を示す -> 安全制約には違反しない	
Vehicle	Entry Controller	入場意思を示す	利用許可が出た後，入場意思を示さない -> 安全制約には違反しない			
Vehicle	Entry Controller	入場する	入場ゲートが上がった後，入場しない -> 安全制約には違反しない		[UCA1] 前の車両が入場ゲートを通過したあとに，ゲートが下降する前に入場することでシステムが認知していない車両が入場する	
Entry Controller	Entry Gate	通行可能な状態にする			[UCA2] 入場意思を示されていないにも関わらず，通行可能な状態にしてしまい，車両が入場する	
Entry Controller	Entry Gate	通行不可能な状態にする			[UCA3] 通行不可能な状態にするアクションを発行が遅れ，後続の車両が入場してしまう	[UCA4] 通行可能な状態になるまでの時間が長く，後続の車両が入場する
Entry Controller	Parking Deck	駐車状況を確認する				
Entry Controller	Parking Deck	車両の入場を伝える	[UCA5] 車両が入場したにも関わらず，車両の入場が伝えられない	[UCA6] 車両の入場を伝えたにも関わらず，駐車状況が更新されていない	[UCA7] 車両が入場していないにも関わらず，車両の入場が伝えられる	

表 3 Entry System の STAMP/STPA 分析結果
Table 3 Result of STPA Step2

UCA	ハザードシナリオ（HS）	NSC	VDM-SLモデルへの反映方針
[UCA1] 前の車両が入場ゲートを通過したあとに，ゲートが下降する前に入場することでシステムが認知していない車両が入場する	HS1.「Vehicle が Entry Controllerの手続き状態がゲートが下降中の状態で入場することで，利用許可なく入場することができ，駐車状況の更新が行われずシステムに認識されない」	NSC1.「Entry Controllerは，Vehicleが想定した状態以外のタイミングで入場してきた時に，事前に決めた対応をとれる状態になっている」	Entry Controllerが想定した状態以外のタイミングで，Vehicleが入場した場合，入場手続き業務を停止し，駐車場の状態を点検中にすることで安全性を高める．
[UCA2] 入場意思を示されていないにも関わらず，通行可能な状態にしてしまい，車両が入場する	HS2.「Entry Controllerのコントロールアルゴリズムに欠陥があり，入場意思を示されていない状態で，Entry Gateに対して通行可能な状態にするアクションを発行してしまう」	NSC2.「Entry Controllerは，入場意思を示されていない状態でEntry Gateに対して通行可能にするアクションを発行してはならない」	ゲートを通行可能にする操作に事前条件の制約を与え，想定しない状態で，操作が発行されたことを検知可能にする．VDM-SLモデルでテストを行い，検知できるかどうかを検証することで安全性を確認する．
[UCA3] 通行不可能な状態にするアクションを発行が遅れ，後続の車両が入場してしまう	HS3.「Entry Controllerのコントロールアルゴリズムに欠陥がありEntry Gate への指示の発行がおくれ，後続の車両に入場されてしまう」	NSC3.「Entry Controllerは車両の入場された際，後続の車両が入場する前に，入場ゲートを通行不可能な状態としなければならない」	実際にゲートの下降中に車両が入場できるかどうかはVDM-SLのモデルでは検証することはできない．従ってUCA1の対策と同様に，車両が不正な入場したことを検知した場合の対応を記述することで，安全性を高めるといった対策を行う．
[UCA4] 通行可能な状態になるまでの時間が長く，後続の車両が入場する	HS4.「Entry Gate の不具合により，通行不可能な状態にするまでに時間がかかり，後続の車両に入場されてしまう」	NSC4.「Entry Gateに不具合を検知できなければならない」	Entry Gateにゲートの異常を検知した時の操作を記述．Entry Controllerはその操作に対し，入場手続き業務を停止し，駐車場の状態を点検中にすることで対応する．
[UCA5] 車両が入場したにも関わらず，車両の入場が伝えられない	HS5.「Entry Controller のコントロールアルゴリズムに欠陥があり，入場された状態で Parking Deck に対して，車両の入場を伝えるアクションを発行しない」	NSC5.「Entry Controllerは，車両に入場された場合は，Parking Deckに入場を伝えなければならない」	VDM-SLモデルを用いて車両の入場操作をテストし，Parking Deck に入場が伝わることを検証することで安全性を確認する．
[UCA6] 車両の入場を伝えたにも関わらず，駐車状況が更新されていない	HS6.「Parking Deckのコントロールアルゴリズムに欠陥があり，駐車状況の更新が行われない」	NSC6.「Parking Deckは入場を伝えられたら駐車状況を更新しなければならない」	VDM-SLモデルを用いて車両の入場操作をテストし，駐車状況が更新されるかを検証することで，安全性を確認する．
[UCA7] 車両が入場していないにも関わらず，車両の入場が伝えられる	HS7.「Entry Controllerのコントロールアルゴリズムに欠陥があり，車両に入場されていない状態で，Parking Deckに対して，車両の入場を伝えるアクションを発行してしまう」	NSC7.「Entry Controllerは，車両の入場を伝えられた場合のみ，車両の入場を伝える」	車両の入場を伝える操作に事前条件を与えることで，想定していない状態で，操作が発行されたことを検知可能にする．VDM-SLモデルでテストを行い，検知できるかどうかを検証することで安全性を確認する．

```
入場される() == (
    手続き段階 := <入場ゲート下降中>;
    ゲートを通行不可能な状態にする();
    手続き段階 := <入場ゲート下降完了>;
    駐車状況の更新();
    入場完了手続き();
)
ext wr 手続き段階 rd ゲート
pre 手続き段階 = <車両入場待ち> and ゲート = <通行可>
post 手続き段階 = <待機中> and ゲート = <通行不可>;
```

図 4 NSC1 を反映する前の VDM-SL 仕様記述
Fig. 4 Specification description that NSC1 is not reflected

以下のような手順に従い VDM-SL モデルへ安全性対策を施す. まず, VDM-SL モデルにおいて, UCA に該当する操作を特定する. 次に, 特定した操作に対して安全性要求を満たすために必要な振る舞いを検討する. 検討した振る舞いを操作の制約条件, 処理として記述することで, VDM-SL モデルに安全性対策を施すことが可能となる.

5.3.1 HS に基づくテストケースの作成

安全性対策を施した VDM-SL モデルの安全性に関する検証を行うために, UCA が発行される HS に基づいたテストケースを 以下の手順に従い作成する. HS において, UCA が発行される可能性がある状態を再現するために, VDM-SL モデルの該当する状態変数を書き換えた後, UCA に該当する操作を発行する処理を行う操作を記述する事でテストケースを作成する. 安全性対策を施した VDM-SL モデルに対し, テストケースを仕様アニメーションで実行し検証することで, UCA が発行された際の VDM-SL モデルの振る舞いを確認することが可能となる.

5.3.2 安全性対策を施した VDM-SL モデルの検証例

前述した手順により, 安全性対策を施した VDM-SL モデルと, そのテストケースを作成し, 実際に仕様アニメーションによるテストを行なった. その結果, 作成したテストケースに対して VDM-SL モデルが想定通りの振る舞いをすることが確認でき, 作成した VDM-SL モデルが分析によって得た安全性要求を満たしていることが確認できた. ここでは一例として, 作成した VDM-SL モデルが 分析によって得られたハザードシナリオ HS1 に対する安全制約である NSC1 の要求を満たしているかどうかの検証結果を示す. まず, STPA の分析の結果得た NSC1 を満たすような記述を VDM-SL モデルに施した. NSC1 を反映させる以前の VDM-SL 仕様記述と, NSC1 を実現するための対策を施した VDM-SL 仕様記述の該当部分とをそれぞれ **図 4**, **図 5** に示す.

NSC1 を反映させる以前の仕様記述では, <車両入場待ち状態>以外の状態で車両が入場してきた場合, 事前条件違反と見なされるが, その後の振る舞いが記述されていないため, 対策としては不十分であると考えた. そこで, Entry Controller が車両の入場待ちをしている状態である<車両

```
-- <入場待ち>状態以外に入場されたら点検中にしてシステムの異常を知らせる
入場される() == (
    if 入場手続き状況 = <車両入場待ち>
        then (
            入場手続き状況 := <入場ゲート下降待ち>;
            ゲートを通行不可能な状態にする();
            入場手続き状況 := <入場ゲート下降完了>;
            車両の入場を伝える();
            入場完了手続き();
        )
    else (
        不正な入場動作を検知();
    )
)
ext wr 入場手続き状況 rd ゲート
pre ゲート = <通行可>
post ゲート = <通行不可>;

-- 不正な車両が入場した時の処理
不正な入場動作を検知() == (
    駐車場を点検中にする();
    入場手続き状況 := <手続き処理停止中>
)
ext wr 入場手続き状況 rd 駐車場の状態, ゲート
post 入場手続き状況 = <手続き処理停止中> and ゲート = <通行不可> and 駐車場の状態 = <点検中>;
```

図 5 NSC1 を反映した VDM-SL 仕様記述
Fig. 5 Specification description reflecting NSC1

```
-- HS1検証
-- 入場ゲートが下降中のタイミングでの入場動作の振る舞い
--  期待する結果：点検中に移行 & ゲート通行不可
HS1_3テスト() == (
    利用許可 := <有>;
    入場手続き状況 := <入場ゲート下降待ち>;
    ゲート := <通行可>;
    車両位置 := <ゲート前>;
    入口状態 := <車両有>;
    入場する();
)
ext wr 駐車場の状態, 入場手続き状況, ゲート, 利用許可, 車両位置, 入口状態;
```

図 6 HS1 を再現するための VDM-SL 仕様記述
Fig. 6 Specification description for test of HS1

入場待ち状態>以外の状態で車両に入場された場合, 不正な入場を検知したと判断し駐車場の状態を点検中にするような振る舞いを定義した. これは, 問題が解決するまで後続の車両が入場手続きを行えない状態にすることで, システムが安全な状態を保つことができると考えたからである. 次に, HS1 を再現するテストケースとして, **図 6** を記述し, 前章で紹介した VDMPad を活用し, 仕様アニメーションによる検証を行なった. 具体的には, Entry Controller が車両の入場を検知し, 入場ゲートを下降させている途中である状態を作り出し, その状態においてアクターである車両の入場命令を発行させた時, Entry Controller が前述したような振る舞いをするかを検証した. 分析結果から得られたハザードシナリオに対して VDM-SL モデルが想定した振る舞いを行なっていることが確認でき, 安全性要求を満たしていることが確認できた.

5.4 トランスレーターを用いたソースコードの生成と評価

前節で作成したハザード分析結果を反映した VDM-SL モデル にトランスレーターを適用し, Python のソースコードを生成した. また, VDM-SL モデルの検証で用いたテストケースを同様にトランスレーターで変換し, 生成されたテストケースを用いて検証を行なった結果, 生成されたソースコードは, VDM-SL モデルと同様の振る舞いが確

```
def 入場される():
    assert ゲート == '<通行可>', 'Pre Condition Error.'
    ret = 入場される_subroutine()
    assert ゲート == '<通行不可>', 'Post Condition Error.'
    return ret


def 入場される_subroutine():
    global 入場手続き状況
    if 入場手続き状況 == '<車両入場待ち>':
        入場手続き状況 = '<入場ゲート下降待ち>'
        ゲートを通行不可能な状態にする()
        入場手続き状況 = '<入場ゲート下降完了>'
        車両の入場を伝える()
        入場完了手続き()
    else:
        不正な入場動作を検知()


def 不正な入場動作を検知():
    ret = 不正な入場動作を検知_subroutine()
    assert (入場手続き状況 == '<手続き処理停止中>' and ゲート == '<通行不可>'
        ) and 駐車場の状態 == '<点検中>', 'Post Condition Error.'
    return ret


def 不正な入場動作を検知_subroutine():
    global 入場手続き状況
    駐車場を点検中にする()
    入場手続き状況 = '<手続き処理停止中>'
```

図 7 トランスレーターにより生成されるソースコード
Fig. 7 Python code generated from specification description with safety measures

```
def HS1_3テスト():
    ret = HS1_3テスト_subroutine()
    return ret


def HS1_3テスト_subroutine():
    global 駐車場の状態, 入場手続き状況, ゲート, 利用許可, 車両位置, 入口状態
    利用許可 = '<有>'
    入場手続き状況 = '<入場ゲート下降待ち>'
    ゲート = '<通行可>'
    車両位置 = '<ゲート前>'
    入口状態 = '<車両有>'
    入場する()
```

図 8 トランスレーターにより生成されたテストケース
Fig. 8 Python code generated from specification description for test of HS1

認でき, 安全性要求を満たしていることが確認できた. 以下にその一例を示す.

5.4.1 生成されたソースコードの検証例

ここでは, 生成されたソースコードのユニットテストにおける検証の一例として, 前小節で扱った VDM-SL モデルからトランスレーターによって生成されるソースコードの検証結果を示す.

VDM-SL モデルをトランスレートして得られたソースコードの内, NSC1 を実現するための対策を施した VDM-SL 仕様記述に当たる部分を**図 7** に示す. また, 図 6 で表される仕様記述から生成した**図 8** のようなプログラムに対し, Python のユニットテストフレームワークである unittest モジュールを活用し, **図 9** のようなテストケースを記述しユニットテストを行なった. その結果, ユニットテスト後の Entry System の状態変数は前小節での VDM-SL モデルに対する仕様アニメーション結果である**図 ??**と同様の

```
import unittest
import entry_system as es

class TestEntrySystem(unittest.TestCase):
    def print_es_state(self, es):
        print("実行後の状態変数")
        print('ゲート：{0}'.format(es.ゲート))
        print('入口状態：{0}'.format(es.入口状態))
        print('入場手続き状況：{0}'.format(es.入場手続き状況))
        print('利用許可：{0}'.format(es.利用許可))
        print('現在の駐車台数：{0}'.format(es.現在の駐車台数))
        print('車両位置：{0}'.format(es.車両位置))
        print('駐車場の状態：{0}'.format(es.駐車場の状態))

    def test_hs1_3(self):
        self.assertIsNone(es.HS1_3テスト())
        self.print_es_state(es)


if __name__ == '__main__':
    unittest.main()
```

図 9 unittest モジュールを用いたテストケース
Fig. 9 Test case with unittest module

```
Hironari-no-MacBook-Air:output hironari$ python unit_test.py
実行後の状態変数
ゲート：<通行不可>
入口状態：<車両無>
入場手続き状況：<手続き処理停止中>
利用許可：<有>
現在の駐車台数：0
車両位置：<駐車場内>
駐車場の状態：<点検中>
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
Hironari-no-MacBook-Air:output hironari$
```

図 10 図 9 のテストケース実行結果
Fig. 10 Result of unittest

状態となっており, VDM-SL モデルと同様の振る舞いが確認できた. これにより, トランスレーターによって生成されたソースコードが VDM-SL モデルの性質を満たし, また, 安全性要求を満たしていることが確認できた.

この例以外のテストケースにおいても, 同様の手順で検証を行うことで, VDM-SL モデルにおける仕様アニメーションによる検証と同様の結果が得られ, トランスレーターにより生成されるプログラムが, トランスレートする前の VDM-SL モデル の性質を満たし, 安全性要求を満たしていることが確認できた.

6. おわりに

本研究では要求定義工程においてドメイン分析を行う開発において, ハザード分析手法である STAMP/STPA の分析結果に基づいた VDM-SL を用いたドメインモデルの構築し, トランスレーターを用いることで VDM-SL で記述したドメインモデルからコード生成を行うことで安全性要求を反映したドメインモデルとソースコードを得る手法を提案した.

6.1 提案手法の考察

体系的な分析による要求漏れの防止

本研究では, 安全性要求の獲得にハザード分析手法である STAMP/STPA を用いた. 先述した事例では, UCA の識別時にハザードには直接繋がらないが, 想定していなかったコントロールアクションの発行が発見できた. このように, STPA のようなガイドワードに沿って分析していくと, 安全性に関する要求の漏れを防ぐことができる.

テストケースとしてのハザードシナリオ

本研究では, 記述した VDM-SL モデル, トランスレートされたプログラムの安全性に関する検証にハザード分析の成果物であるハザードシナリオを活用した. また, ハザードシナリオは VDM-SL を用いて実行可能であり, トランスレーターで変換可能であることが確かめられた. 従って, 分析によって得た 7 つのハザードシナリオを用いることで, 続く工程の成果物である VDM-SL モデルや最終的なプログラムに対して, 効果的な検証が行える.

ドメイン単位での仕様の再利用

提案手法では, ドメイン固有の要素や属性, 振る舞い, 安全性に関する制約を VDM-SL を用いてモデル化しそれ以降の開発工程で活用する. 構築した VDM-SL モデルは, ドメインを対象とする同系統のシステムの共通の仕様として見ることができ, VDM-SL モデルを資産として管理することで, ドメイン単位での仕様の再利用が可能であると考えられる. 例えば, 駐車場入場管理業務というドメインを対象とするシステムの可変性としては, 入場ゲートの数などが考えられるが, 1 つのゲートに関する制御の流れは入場ゲートの数には影響されない. 従って, 今回作成した仕様の大部分は再利用可能であると考える.

回帰テストにおけるテストケースの活用

本研究の事例において, Entry Controller が Vehicle の入場を想定している入場手続き状態は<車両入場待ち>のみであり, それ以外の場合は不正な状態として処理されるような仕様になっている. これからの開発で<車両入場待ち>以外の状態での車両の入場を許可するような仕様変更がなされた場合, トランスレーターによって生成されるプログラムに対して, 車両の入場を許可していない状態で車両の入場を検知した場合は不正な入場として処理されるかどうかを検証する必要がある. このような場合, 今回作成したテストケースを回帰テストのテストケースとして活用できると考える.

トランスレートされたテストケースの活用

トランスレーターで生成されたプログラムの検証に, VDM-SL モデルの検証で用いたテストケースをトランスレーターで変換し, 生成したテストケースを活用することで, モデルとプログラム間で一貫性のある検証が行えることが確認できた. トランスレーターを活用することで, 開発の各工程で同じテストケースを利用することができるため, 開発の効率向上が期待できると考えられる.

6.2 今後の課題

本研究では DbC の概念における事前条件, 事後条件の検証に対応したプログラムを生成するトランスレーターを作成した. DbC において満たすべき条件として, 事前条件, 事後条件に加えて不変条件がある. 本研究で作成したトランスレーターは, 不変条件に関する記述のトランスレートに対応していない. 不変条件に関する記述のトランスレート機能の実装は今後の課題である.

参考文献

[1] Python software foundation, python language reference, version 3.6. available at, `http://www.python.org`.
[2] David Beazley, PLY (Python Lex-Yacc) Version 3.11, `https://github.com/dabeaz/ply`, 2017.
[3] Nancy G. Leveson, An STPA Primer Version 1, August 2013 (updated June 2015), `https://psas.scripts.mit.edu/home/home/stpa-primer/`.
[4] Nancy G. Leveson, *SAFEWARE System Safety and Computers*, Addison-Wesley, 1995.
[5] Bertrand Meyer, *Object-Oriented Software Construction SECOND EDITION*, chapter 11, Prentice Hall, 2000.
[6] Tomohiro Oda, Keijiro Araki, and Peter Gorm Larsen, VDMPad: A Lightweight IDE for Exploratory VDM-SL Specification, In *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*, Formalise '15, pp. 33–39, Piscataway, NJ, USA, 2015, IEEE Press.
[7] Berker Peksag, astor – AST observe/rewrite, `https://github.com/berkerpeksag/astor`, 2017.
[8] Rubén Prieto-Díaz, Domain analysis: An introduction, *SIGSOFT Softw. Eng. Notes*, Vol. 15, No. 2, pp. 47–54, April 1990.
[9] システム安全性解析手法 WG, はじめての STAMP/STPA ～システム思考に基づく新しい安全性解析手法～, 独立行政法人 情報処理推進機構 (IPA), 2016.
[10] 荒木啓二朗, 張漢明, プログラム仕様記述論, オーム社, 2003.
[11] 栗田太郎, フォーマルメソッドの新潮流 : Part II: 産業界への応用 : 3. 携帯電話組込み用モバイル FeliCa IC チップ開発における形式仕様記述手法の適用, 情報処理, Vol. 49, No. 5, pp. 506–513, may 2008.