**МАГІСТЕРСЬКА РОБОТА**

МР. ІПм - 40.00.00.000 ПЗ

**Група ІПм-23-1**

**Довбонос Олександр**

**2024**

**Івано-Франківський національний технічний університет нафти і газу**

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

**Довбонос Олександр Сергійович**

(прізвище, ім’я, по батькові)

УДК 004.942

(індекс)

# МАГІСТЕРСЬКА РОБОТА

**Моделі та методи підвищення можливостей верифікації в комп'ютеризованих системах**

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Довбонос О.С.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Піх Володимир Ярославович, к.т.н., доцент

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

**Допущено до захисту**

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

**Івано-Франківський національний технічний університет нафти і газу**

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

**ЗАТВЕРДЖУЮ:**
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

# ЗАВДАННЯ

## НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

**Довбоносу Олександру Сергійовичу**

(прізвище, ім’я, по-батькові)

**1. Тема магістерської роботи “ Моделі та методи підвищення можливостей верифікації в комп'ютеризованих системах ”**

керівник проекту (роботи) Піх Володимир Ярославович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

**2. Строк подання студентом проекту (роботи)** 15 грудня 2024 р.

**3. Вихідні дані до проекту (роботи)** Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій верифікації програмного забезпечення

**4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)**

1. Дослідження предметної області процесів верифікації в контексті спеціальних моделей
2. Методи моделювання, аналізу та верифікації компонентів і моделей
3. Підходи до інтеграції засобів перевірки моделей
4. Методи підвищення верифікації в комп'ютеризованих системах через перетворення моделі

**5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)**

1. Модель шаблону трансформації (рис. 1.1)
2. Створення моделі засобами Eclipse Modeling Framework (рис. 1.2)
3. Архітектура ITS-інструментів (рис. 1.3)
4. Робочий процес перевірки моделі mCRL2 (рис. 1.4)
5. LTSmin архітектура (рис. 1.5)

**6. Консультанти розділів проекту (роботи)**

| Розділ | Консультант | Підпис, дата |
|---|---|---|
| Перевірка на плагіат | доц., к.т.н. Вовк Р.Б. | |
| | | |
| | | |

**7. Дата видачі завдання** 04 вересня 2024 р.

**Керівник** ______________________________
(підпис)

**Завдання прийняв до виконання**______________________________
(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № п/п | Назви етапів магістерської роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Підбір і вивчення літератури по темі магістерської роботи | 15.09.2024 | виконано |
| 2 | Аналіз концепцій та алгоритмів предметної області | 29.09.2024 | виконано |
| 3 | Дослідження предметної області процесів верифікації в контексті спеціальних моделей | 15.10.2024 | виконано |
| 4 | Методи моделювання, аналізу та верифікації компонентів і моделей | 08.11.2024 | виконано |
| 5 | Підходи до інтеграції засобів перевірки моделей | 20.11.2024 | виконано |
| 6 | Методи підвищення верифікації в комп'ютеризованих системах через перетворення моделі | 01.12.2024 | виконано |
| 7 | Затвердження пояснювальної записки роботи завідувачем кафедри | 15.12.2024 | виконано |

**Студент – магістр** ____________________
(підпис)

**Керівник роботи** ____________________
(підпис)

## АНОТАЦІЯ

**Магістерська робота:** 83 с., 20 рис., 5 табл., 53 джерел.

**Тема:** Моделі та методи підвищення можливостей верифікації в комп'ютеризованих системах

**Об'єкт дослідження:** процеси верифікації у комп'ютеризованих системах з використанням модельно-орієнтованої інженерії.

**Мета роботи:** дослідження методів і моделей для підвищення можливостей верифікації у комп'ютеризованих системах, що дозволить покращити якість програмного забезпечення шляхом забезпечення коректності та відповідності систем заданим вимогам.

**Предмет дослідження:** моделі, методи та інструменти для верифікації та перевірки комп'ютеризованих систем на основі предметно-спеціальних моделей.

**Результати дослідження**

В роботі представлено підхід до створення компонентних моделей з функціональними обмеженнями, що забезпечує коректне відображення станів і синхронізацій у верифікації.

**Висновок**

Запропоновано метод перетворення моделей для підвищення ефективності перевірки складних систем, що дозволяє поліпшити виявлення аномалій та помилок у системах з різними компонентами

**МОДЕЛЬНО-ОРІЄНТОВАНА ІНЖЕНЕРІЯ, ВЕРИФІКАЦІЯ, КОМП'ЮТЕРИЗОВАНІ СИСТЕМИ, ПРЕДМЕТНО-СПЕЦІАЛЬНІ МОДЕЛІ, ТРАНСФОРМАЦІЯ МОДЕЛЕЙ, ФУНКЦІОНАЛЬНІ ОБМЕЖЕННЯ, МЕРЕЖІ ПЕТРІ**

## ABSTRACT

**Master Thesis:** 83 pp., 20 fig., 5 tab., 53 sources.

**Thesis Subject:** Models and methods of increasing verification capabilities in computerized systems

**Research object:** verification processes in computerized systems using model-oriented engineering.

**The purpose of the work:** the study of methods and models for increasing the verification capabilities in computerized systems, which will allow to improve the quality of the software by ensuring the correctness and compliance of the systems with the specified requirements.

**Research subject:** models, methods and tools for verification and testing of computerized systems based on subject-specific models.

**Research results**

The paper presents an approach to creating component models with functional limitations, which ensures the correct display of states and synchronizations in verification.

**Conclusion**

A model transformation method is proposed to increase the effectiveness of checking complex systems, which makes it possible to improve the detection of anomalies and errors in systems with various components



**MODEL-ORIENTED ENGINEERING, VERIFICATION, COMPUTERIZED SYSTEMS, SUBJECT-SPECIFIC MODELS, MODEL TRANSFORMATION, FUNCTIONAL CONSTRAINTS, PETRI NETS**

# ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ........ 9
ВСТУП ........ 10

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПРОЦЕСІВ ВЕРИФІКАЦІЇ В КОНТЕКСТІ ПРЕДМЕТНО-СПЕЦІАЛЬНИХ МОДЕЛЕЙ ........ 13
1.1. Опис предметної області дослідження ........ 13
1.2. Особливості модельно-орієнтованої інженерії ........ 16
1.2.1. Предметно-орієнтовані мови ........ 18
1.2.2. Метамодель, метаметамодель і трансформації ........ 20
1.2.3. Платформа моделювання Eclipse ........ 21
1.3. Дослідження інструментів перевірки моделей ........ 22
Висновки до розділу ........ 28

РОЗДІЛ 2. МЕТОДИ МОДЕЛЮВАННЯ, АНАЛІЗУ ТА ВЕРИФІКАЦІЇ КОМПОНЕНТІВ І МОДЕЛЕЙ ........ 29
2.1. Синтаксис і семантика інтерфейсу платформи для моделювання та аналізу компонентів ........ 29
2.2. Створення артефактів верифікації засобами платформи моделювання 35
2.2.1. Можливості перевірки та верифікації моделей ........ 37
2.2.2. Обмеження поточної методології верифікації ........ 39
2.3. Підходи до інтеграції засобів перевірки моделей ........ 41
2.4. Відображення моделей з використанням мережі Петрі ........ 43
2.5. Особливості, компоненти та архітектура ITS-інструментів верифікації символьної моделі ........ 48
Висновки до розділу ........ 51

РОЗДІЛ 3. МОДЕЛІ ТА МЕТОДИ ПІДВИЩЕННЯ МОЖЛИВОСТЕЙ ВЕРИФІКАЦІЇ В КОМП'ЮТЕРИЗОВАНИХ СИСТЕМАХ ЧЕРЕЗ ПЕРЕТВОРЕННЯ МОДЕЛІ ..... 53
3.1. Відображення для моделей інтерфейсу ..... 53
3.1.1. Попередньо визначені типи ..... 55
3.1.2. Перерахування, вектори, записи та карти ..... 56
3.2. Відображення для компонентних моделей ..... 60
3.2.1. Функціональні обмеження для верифікації ..... 61
3.2.2. Відображення для функціональних обмежень на основі стану ..... 62
3.2.3. Вкладені компоненти ..... 65
3.2.4. Синхронізаційні моделі інтерфейсу та функціональні обмеження на основі стану ..... 67
3.3. Перевірка компонентів із функціональними обмеженнями ..... 69
Висновки до розділу ..... 76

ВИСНОВКИ ..... 78
ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ ..... 79

## ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ACP - Algebra of Communicating Processes.

ADT - Abstract Data Type.

AST - Abstract Syntax Tree.

BDD - Binary Decision Diagram.

CBS - Component-Based System.

CST - Concrete Syntax Tree.

CTL - Computation Tree Logic.

DDD - Data Decision Diagrams.

DSL - Domain-Specific Language.

EMF - Eclipse Modelling Framework.

GAL - Guarded Action Language.

GLT - Generic Language Technology.

GPL - General Purpose Language.

HLPN - High-Level Petri Net.

ITS - Instantiable Transition System.

LKS - Labelled Kripke Structure.

LTL - Linear Temporal Logic.

LTS - Labeled Transition System.

MDE - Model-Driven Engineering.

MOF - Meta-Object Facility.

OMG - Object Management Group.

PN - Petri Net.

PNML - Petri Net Markup Language.

ROBDD - Reduced Ordered Binary Decision Diagram

SDD - Hierarchical Set Decision Diagrams.

SeDD - Sentential Decision Diagram.

SUT - System Under Test.

# ВСТУП

**Актуальність теми.**

В сучасному світі інформаційні технології є основою багатьох критичних галузей, таких як медицина, енергетика, транспорт, аерокосмічна та військова промисловість. Ці сфери потребують особливо високого рівня надійності та коректності програмного забезпечення, яке відповідає за керування складними комп'ютеризованими системами. Будь-які помилки чи невідповідності в роботі програмного забезпечення можуть призвести до серйозних наслідків, включаючи людські втрати, фінансові збитки та руйнування інфраструктури. Тому верифікація — процес підтвердження коректності та відповідності системи вимогам — є критично важливою на всіх етапах розробки.

Актуальність цієї роботи обумовлена необхідністю створення ефективних моделей та методів верифікації, які дозволяють здійснювати формальний і надійний контроль за правильністю функціонування складних програмних систем. Існуючі підходи часто мають обмеження в контексті перевірки складних компонентних структур і взаємодій між ними. Предметно-орієнтовані моделі, зокрема метамоделі та спеціалізовані мови, відкривають нові можливості для вирішення цих завдань, дозволяючи не тільки підвищити точність верифікації, а й оптимізувати процеси розробки.

Робота є також актуальною через необхідність інтеграції автоматизованих інструментів верифікації в сучасні процеси розробки. Такі інструменти, як платформа Eclipse, надають широкий спектр можливостей для моделювання, проте потребують подальших досліджень та адаптації під специфічні потреби верифікації. Удосконалення методів моделювання та верифікації, запропоноване в цій роботі, дозволить значно покращити управління якістю програмного забезпечення, особливо для критичних галузей.

Крім того, зростаючі вимоги до складності та функціональності сучасних систем вимагають ефективних методів для роботи з великим обсягом даних та складними архітектурами. Дослідження, спрямовані на оптимізацію процесів верифікації через моделі та трансформації, дозволяють забезпечити масштабованість та ефективність розробки. Таким чином, результати даної роботи є актуальними для розвитку технологій надійного програмного забезпечення, необхідного для безпечного функціонування комп'ютеризованих систем в сучасних умовах стрімкого технологічного прогресу.

**Метою дослідження** є дослідження методів і моделей для підвищення можливостей верифікації у комп'ютеризованих системах, що дозволить покращити якість програмного забезпечення шляхом забезпечення коректності та відповідності систем заданим вимогам.

**Об'єкт дослідження –** процеси верифікації у комп'ютеризованих системах з використанням модельно-орієнтованої інженерії.

**Предмет дослідження** – моделі, методи та інструменти для верифікації та перевірки комп'ютеризованих систем на основі предметно-спеціальних моделей.

**Задачі дослідження:**

- Провести дослідження та аналіз предметної області процесів верифікації у контексті предметно-спеціальних моделей.

- Вивчити особливості модельно-орієнтованої інженерії та методів, що використовуються для створення та трансформації предметно-спеціальних моделей.

- Розглянути наявні інструменти для перевірки моделей та дослідити їх можливості і обмеження.

- Розробити та обґрунтувати методи підвищення ефективності верифікації компонентів у комп'ютеризованих системах через використання перетворення моделей.

- Провести експериментальну перевірку запропонованих методів для підтвердження їх ефективності.

**Методи дослідження**

У дослідженні використовуються методи формального моделювання, аналізу та верифікації моделей, синтаксичний та семантичний аналіз, а також методи симуляції та експериментальної оцінки ефективності моделей. Для побудови та перевірки моделей застосовано інструменти модельно-орієнтованої інженерії та платформи Eclipse.

**Наукова новизна отриманих результатів**

Запропоновано методи відображення для моделей інтерфейсів та компонентів з використанням попередньо визначених типів, що підвищують зручність і точність верифікації.

**Практичне значення результатів**

Представлені методи і моделі можуть бути використані у процесах розробки та верифікації комп'ютеризованих систем для покращення їхньої якості та надійності. Результати дослідження також можуть бути застосовані в навчальних і дослідницьких установах для підготовки фахівців з програмного забезпечення, а також інтегровані в процеси розробки в компаніях, які займаються складними проєктами в галузі критичних систем.

**Структура магістерської роботи.** Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 83 сторінки, і містить 20 рисунків, 5 таблиць, список використаних джерел із 53 найменувань.

# РОЗДІЛ 1. ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПРОЦЕСІВ ВЕРИФІКАЦІЇ В КОНТЕКСТІ ПРЕДМЕТНО-СПЕЦІАЛЬНИХ МОДЕЛЕЙ

## 1.1. Опис предметної області дослідження

Складні вбудовані програмні системи розробляються на основі компонентів, щоб зробити проектування, валідацію та обслуговування більш керованим. Система на основі компонентів (CBS) [3] розділена на модульні частини з високою згуртованістю та низьким зв’язком, які називаються компонентами, кожна з яких має чітко визначене призначення. Вбудоване програмне забезпечення кожного компонента розробляється незалежно, а дані та функції, що містяться, відокремлені від зовнішніх компонентів. Зрештою, ці компоненти повинні синхронізуватися та працювати для досягнення спільної мети, вимагаючи спілкування. Компоненти обмінюються інформацією через повідомлення, використовуючи парадигму зв’язку клієнт-сервер. Важливо, щоб ці компоненти були розроблені таким чином, щоб їх було легко обслуговувати та використовувати повторно. Один із способів досягти цього — абстрагувати функціональні можливості компонента від його реалізації, забезпечивши стандартизований метод доступу до базових служб.

Ця мета досягається за допомогою компонентних інтерфейсів, які описують прийнятні типи повідомлень, формати даних і задану поведінку шляхом деталізації обмежень на доступ до конкретних ресурсів. Абстрагуючи реалізацію компонента від його ролі вищого рівня, розробка стає більш керованою за рахунок спрощення зв’язку та синхронізації з іншими компонентами. Поки протокол зв’язку між компонентами залишається незмінним, внутрішні оновлення компонентів можна виконувати незалежно. Крім того, стандартизація інтерфейсів компонентів за допомогою точної та чіткої семантики дозволяє покращити зв’язок між

зацікавленими сторонами, залученими до процесу проектування та розробки, що може призвести до ще ефективнішої інтеграції компонентів.

Іншою тенденцією є збільшення попиту та використання методологій, які використовують методи керованого моделюванням (MDE) [4]. Разом із розробкою програмного забезпечення на мові загального призначення створюються та підтримуються предметно-орієнтовані мови (DSL) [5], за допомогою яких можна розробляти високорівневі моделі системних модулів. DSL спеціально розроблено для конкретного контексту, а абстрактне моделювання забезпечує підвищений рівень виразності, що дозволяє розробникам визначати більше функціональних можливостей із меншим кодом. Ці моделі використовуються як вихідні точки для різних методів, таких як моделювання, аналіз і перевірка. Інструменти розробки програмного забезпечення на основі моделей, такі як Eclipse, дозволяють створювати метамоделі мови, які визначають граматику та семантику DSL. Ці інструменти дозволяють генерувати артефакти, такі як код мовою загального призначення, генератори тестів, документація та візуалізації.

Розробка програмного забезпечення для вбудованих систем може бути складною через відсутність стандартизованої методології створення та підтримки компонентних інтерфейсів. Постійна проблема полягає в тому, що розробники часто не можуть надати чіткого та точного визначення того, що передбачає інтерфейс. Це може призвести до неточної та неоднозначної семантики, що призведе до різних інтерпретацій серед членів команди. Такі різні інтерпретації можуть спричинити конфлікти та неузгодженості під час інтеграції та обслуговування компонентів, що зрештою може призвести до збоїв системи. Одним із основних джерел таких збоїв є ненавмисні зміни протоколу глобальної взаємодії та поведінки синхронізації. Відповідне керування інтерфейсом на стадії розробки, таким чином, має важливе значення для еволюції системи, оскільки перевірка того, що вся система підтримує очікувану функціональність після оновлення компонентів, не є тривіальним завданням.

Зацікавленість кількох компаній у пошуку вирішення цієї загальної проблеми разом із зростаючою тенденцією двох вищезгаданих принципів (тобто CBS, MDE) призвела до створення пакету компонентного моделювання та аналізу, або ComMA. ComMA — це DSL на основі Eclipse, який підтримує розробку на основі компонентів шляхом формалізації методології розробки надійного інтерфейсу та специфікацій компонентів. Різні компанії успішно використовували ComMA. ComMA дозволяє користувачам визначати моделі інтерфейсу та компонентів, які можна використовувати для створення всебічної абстрактної моделі протоколу зв'язку. ComMA використовує семантику кінцевого автомата для опису послідовності подій, які може приймати інтерфейс. Це також дозволяє специфікацію типів даних, типів повідомлень, а також даних і часових обмежень. Кілька інтерфейсів можна комбінувати для створення моделі компонентів більшої підсистеми.

Ці моделі використовуються для генерації різноманітних артефактів. Такі візуалізації, як діаграми кінцевих автоматів і діаграми послідовностей, дають змогу зрозуміти поведінку моделей, а шаблони документації допомагають пришвидшити процес створення документації після оновлення. Підтримується обмежена генерація проксі-коду інтерфейсу для певних типів проміжного програмного забезпечення. Тестові клієнти та генератори тестів можуть ефективно створювати повний набір тестів, що зменшує витрати на обслуговування оновлень програмного забезпечення. Нарешті, моніторинг і моделювання під час виконання можна використовувати, щоб перевірити, чи перевірена система (SUT) справді відповідає очікуваному протоколу зв'язку.

Використовуючи модель для завдання генерації, важливо спочатку перевірити, чи є її основна поведінка узгодженою та правильною щодо властивостей, які перевіряються. ComMA був розроблений як дуже виразний, щоб користувачі могли вказувати широкий спектр інтерфейсів із складною поведінкою. Наприклад, моделі можуть включати змінні як примітивних, так і визначених користувачем типів даних, над якими можна виконувати різні

операції. Крім того, кілька конструкцій можна використовувати для визначення недетермінованих шаблонів для порядку, в якому можуть відбуватися різні види подій. Нарешті, кілька інтерфейсів можна використовувати разом у межах одного компонента, дозволяючи накладати ще більше обмежень на дозволений порядок подій. Крім того, оскільки ці моделі не генеруються з коду, розробники програмних інтерфейсів, які мають глибоке розуміння їх очікуваної поведінки, намагаються визначити їх семантику за допомогою ComMA. Однак через основну складність інтерфейсів програмного забезпечення людський фактор також може призвести до ненавмисної, неочікуваної та небажаної поведінки в поведінці вказаних моделей. Таким чином, розробка правильних і надійних моделей у ComMA без допомоги формальної перевірки є нетривіальною та може бути схильною до помилок.

Нездатність підтвердити правильність моделей щодо їх бажаної поведінки може мати негативні наслідки для загальної всеосяжної мети використання ComMA. Моделі використовуються як вхідні дані для створення симуляторів, тестових клієнтів і моніторів виконання, які призначені для допомоги розробникам у життєвому циклі програмних інтерфейсів. Використання дефектних моделей як вхідних даних безпосередньо впливає на надійність цих створених артефактів і, таким чином, на загальні переваги використання ComMA. Таким чином, наявність можливості стверджувати правильність моделей формально призводить до підвищення впевненості, яку користувач отримує після використання їх для генерації артефактів.

## 1.2. Особливості модельно-орієнтованої інженерії

Модельно-орієнтована інженерія (MDE) — це методологія розробки та розробки програмного забезпечення, зосереджена на використанні абстрактних предметно-спеціальних моделей як основних артефактів у

процесі розробки [4]. Головною метою цієї методології є спрощення процесу проектування, підвищення рівня автоматизації та покращення сумісності між компонентами та системами. Використання абстрактних моделей і включення предметно-спеціальної інформації при розробці цих моделей дозволяє підвищити виразність у визначенні функціональних можливостей і очікуваної поведінки системи, а також покращити зв'язок між залученими зацікавленими сторонами за допомогою використання абстрактного вищого рівня. діалект, що покращує чіткість і ефективність.

Історично першим інструментом підтримки цієї методології є уніфікована мова моделювання (UML), яка була стандартизована групою управління об'єктами (OMG). UML є першою стандартизованою мовою візуального моделювання, яка забезпечує нотацію для численних видів діаграм, які зосереджуються на різних аспектах системи, таких як поведінка, взаємодія та структура. Спочатку ці діаграми використовувалися лише для опису системи в широких архітектурних термінах, а потім системи розроблялися на конкретній мові загального призначення (GPL).

Підтримка інструментів для MDE значно покращилася за останні роки, і тепер кілька інструментів і екосистем використовуються в дослідницьких і промислових програмах. Деякі відповідні мовні верстаки включають Eclipse EMF [6] і JetBrains MPS [12]. Ці інструменти пропонують необхідну технологію для визначення повноцінних доменних мов (DSL), які використовуються для визначення моделей. Ці верстаки також пропонують можливість використання моделей для створення артефактів, таких як код проміжного програмного забезпечення в GPL, візуалізації, генератори тестів і багато іншого. Це, у свою чергу, передбачає різку зміну важливості моделі як основного артефакту в розвитку. Якщо раніше моделі UML використовувалися лише для визначення архітектури системи, сьогодні моделі можуть бути критичним фактором у розробці, а також у підтримці та перевірці систем.

Для сучасних вбудованих систем правильність коду в кінцевому підсумку відіграє життєво важливу роль, особливо для критично важливих для безпеки систем, де помилки можуть призвести до величезних витрат. З цієї точки зору також можна побачити, як підвищена експресивність DSL, а також генерація коду та інструментів перевірки можуть відігравати вирішальну роль у вдосконаленні стандартизованої методології розробки.

### *1.2.1. Предметно-орієнтовані мови*

Доменно-орієнтовані мови (DSL) — це мови програмування з вищим рівнем абстракції, які використовують предметно-специфічну інформацію для спеціалізованого класу проблем або конкретних аспектів системи. DSL використовують численні концепції та правила з відповідної області, щоб підвищити свою виразність, і це очевидно в їхній виділеній семантиці та синтаксисі. Зазвичай вони менш складні, ніж ліцензії GPL, і розробляються в тісній співпраці з експертами тієї області, для якої вони розроблені. Часто їх розробляють експерти з розробки програмного забезпечення, а кінцевими користувачами є інженери домену, які є експертами та вільно володіють своєю сферою практики.

Порівняно з GPL, DSL, як правило, мають коротший термін служби, і, будучи доменно-спеціальними, вони, як правило, мають невеликі спільноти користувачів, які складають експертів у домені, як правило, від однієї компанії до кількох компаній, які створюють DSL у співпраці. Однак існують також численні DSL, в основному створені дослідницькими установами, які з часом стають проектами з відкритим кодом. Вони також мають дуже швидкий розвиток і еволюцію, що представляє технічну складність, але також є перевагою в гнучкості. Сьогодні багато компаній, які займаються розробкою вбудованого програмного забезпечення, інвестують у розробку DSL, щоб покращити свої методології розробки.

Generic Language Technology (GLT) зосереджується на інтуїтивно зрозумілому та простому створенні та підтримці DSL за допомогою

спеціальних мовних робочих місць та інструментів. Цей тип технології спрямований на створення експресивних і гнучких методів для визначення специфічних для мови аспектів, а також багаторазових рішень для загальних аспектів, спільних для всіх DSL. Вони також пропонують багатофункціональну екосистему, яка дозволяє користувачам легко впроваджувати генерацію інструментів і артефактів. Існує два типи методів розробки DSL:

■ Методи граматичного програмного забезпечення зосереджені на розробці DSL з використанням функціональних мов метапрограмування. Метапрограмування — це техніка розробки метапрограм, які мають можливість розглядати інші програми як свої дані. Таким чином, вони можуть читати, генерувати, перевіряти обмеження та трансформувати програми. Використовуючи цю методологію, розробники DSL можуть використовувати мову, яка пропонує виразні методи реалізації граматики генератора DSL. Workbench містить вбудований лексер і синтаксичний аналізатор, які дозволяють перетворювати програми в конкретні та абстрактні синтаксичні дерева (тобто, які пояснюються нижче). Потім розробник DSL може визначити створення артефактів, обмеження перевірки, а також перетворення, використовуючи абстрактне синтаксичне дерево як вхідні дані. Прикладом робочого середовища, яке використовує підхід граматичного програмного забезпечення, є Rascal [13].

■ На методи моделювання сильно впливають методи об'єктно-орієнтованого моделювання. У рамках цього підходу фокус і мета відрізняються, оскільки, по суті, мова більше не розглядається як набір речень, які є похідними від контекстно-вільної граматики, а радше як набір моделей, які відповідають метамоделі. У цьому контексті conforms можна визначити як функцію, яка відображає всі елементи з модельного графа на метамодельний граф. Метамодель представляє основу визначення DSL. Методи модельного програмного забезпечення часто також пропонують готові інструменти, які дозволяють розробникам мови створювати

конкретний текстовий синтаксис, а також графічні редактори для розробленого DSL.

*1.2.2. Метамодель, метаметамодель і трансформації*

Метамодель DSL представляє абстрактну структуру мови, і це ключовий артефакт у підході модельного програмного забезпечення. Замість метапрограмування мовні верстаки пропонують інтерактивний графічний інтерфейс, який пропонує можливість розробки метамоделей, які мають форму графової структури. Цю структуру можна розглядати як модель високого рівня мови, кожен окремий екземпляр моделі або програми, написані цією конкретною мовою, які розглядаються як моделі, що відповідають метамоделі. Самі метамоделі відповідають метаметамоделі, яку можна розглядати як спеціальну мову моделювання, за допомогою якої можна моделювати мови моделювання.

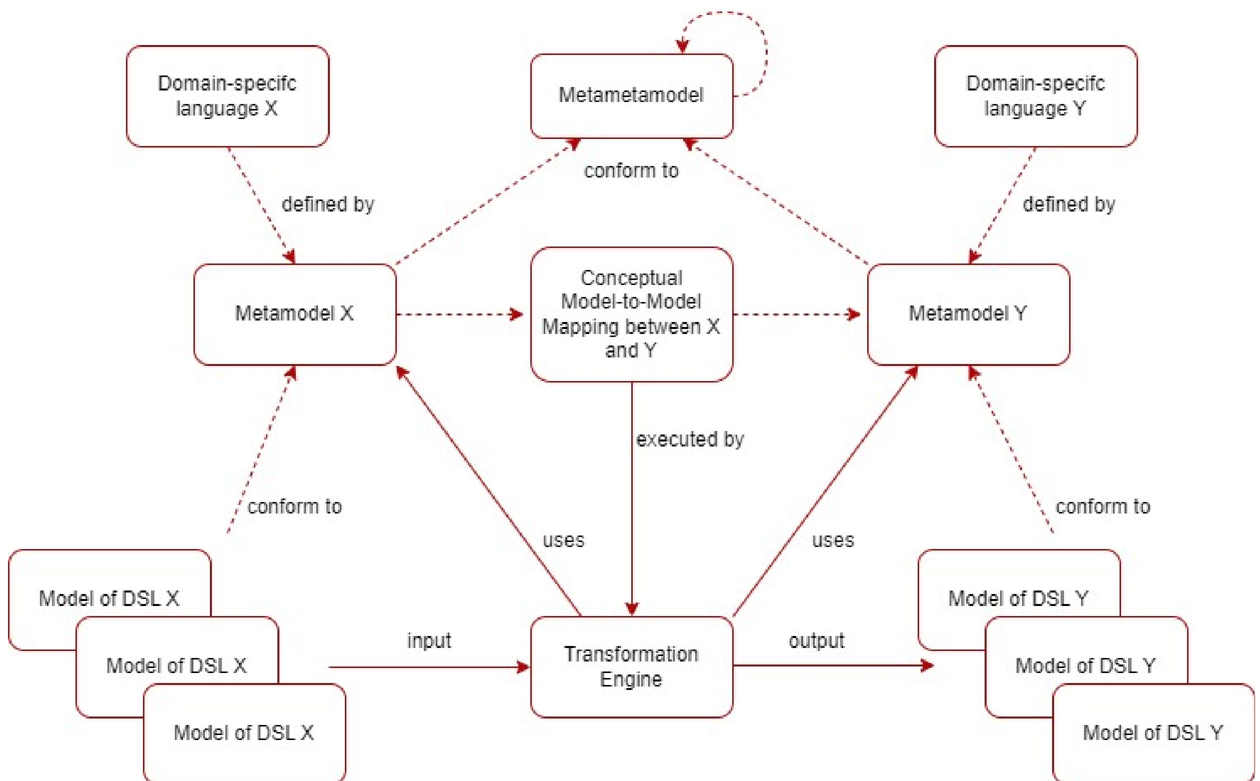


Рис. 1.1. Модель шаблону трансформації

Метаметамодель — це стандартизована мова, яка сама собі відповідає. Стандартом, який описує ці концепції, є стандарт Meta-Object Facility (MOF) [14], опублікований групою OMG. Численні завдання керування моделлю можуть бути застосовані до моделей, такі як генеративні завдання (наприклад, генерування коду/тексту, а також інструментів з моделей), операції підтримки моделі (наприклад, рефакторинг, злиття, міграція моделей у міру розвитку їхньої метамоделі), а також здійснення від моделі до моделі трансформації з одного DSL в інший. На рисунку 1.1 представлено шаблон перетворення моделі.

Ключовий висновок із цього шаблону полягає в наступному: між метамоделями двох мов робиться концептуальне відображення. Це концептуальне відображення потім реалізується, і отриманий механізм перетворення виконує відображення на вхідних моделях, які відповідають вихідній метамоделі. Він виводить моделі, що відповідають цільовій метамоделі. Обидві метамоделі відповідають метаметамоделі, яка відповідає сама собі.

Також можливо, що вихідна мова не має метамоделі. У цьому випадку виконується перетворення моделі в текст (наприклад, при створенні шаблонів документації). Також може бути так, що цільовою мовою є не DSL, а GPL, у цьому випадку може бути згенерований прямий виконуваний текстовий код.

### *1.2.3. Платформа моделювання Eclipse*

ComMA було розроблено з використанням Eclipse Modeling Framework (EMF), яка є структурою моделювання, яка підпадає під підхід модельного програмного забезпечення. Він підтримує визначення моделей домену або метамоделей за допомогою мови Ecore , яка є основою EMF. Ecore є де-факто еталонною реалізацією стандарту MOF. Усі метамоделі Ecore зберігаються в XMI (XML Metadata Interchange), але розбір виконується прозоро для користувачів. За допомогою метамоделі Ecore можна створити код Java, який

містить класи для елементів у метамоделі, а також фабричні методи. За допомогою фабричних методів ці класи Java створюються в структуровану колекцію об’єктів Java, які є моделями DSL.

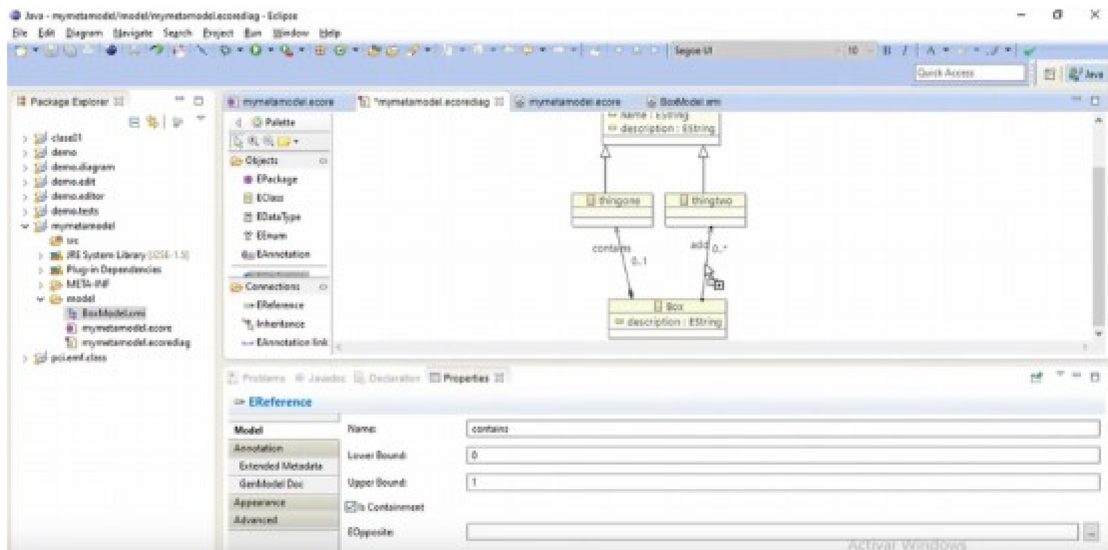

Рис. 1.2. Створення моделі засобами Eclipse Modeling Framework

Також можна створити та анотувати конкретну контекстно-вільну граматику, яка визначається за допомогою Xtext. Xtext пропонує інтегровану IDE Eclipse з численними функціями, такими як автозавершення та підсвічування. Іншою корисною мовою, яка походить від Xtext, є Xtend, яку можна використовувати для розробки перетворень модель-модель і модель-текст. Xtend базується на Java, забезпечуючи функціональні можливості вищого рівня, такі як визначення типу та перевантаження операторів. Він спрямований на зменшення багатослівності Java, покращення її виразності та пропонування нових функцій, які особливо корисні для перетворень моделі. Xtend є основною мовою, яка використовується для реалізації.

### 1.3. Дослідження інструментів перевірки моделей

ITS-tools — це засіб перевірки символьної моделі, який підтримує перевірку властивостей досяжності, а також CTL і LTL. Він використовує

бібліотеку libDDD і, як такий, пропонує можливість представлення простору станів як DDD, а складніші моделі використовують SDD. Перетворення від моделі до моделі від різних формалізмів введення до спеціальної мови введення Guarded Action Language (GAL) є центральною точкою в його розвитку. ITS-tools пропонується як окремий інструмент Eclipse, тоді як серверні програми програмуються на C та C++. Він призначений для того, щоб запропонувати DSL можливості перевірки моделі за допомогою технології EMF модельного забезпечення. ITS-інструменти також можуть нативно приймати HLPN у представленнях PNML. З точки зору алгоритмів, ITS-інструменти використовують ефекти насичення в обчисленнях із фіксованою точкою та гомоморфізми, які дозволяють ефективно кодувати операції DD.

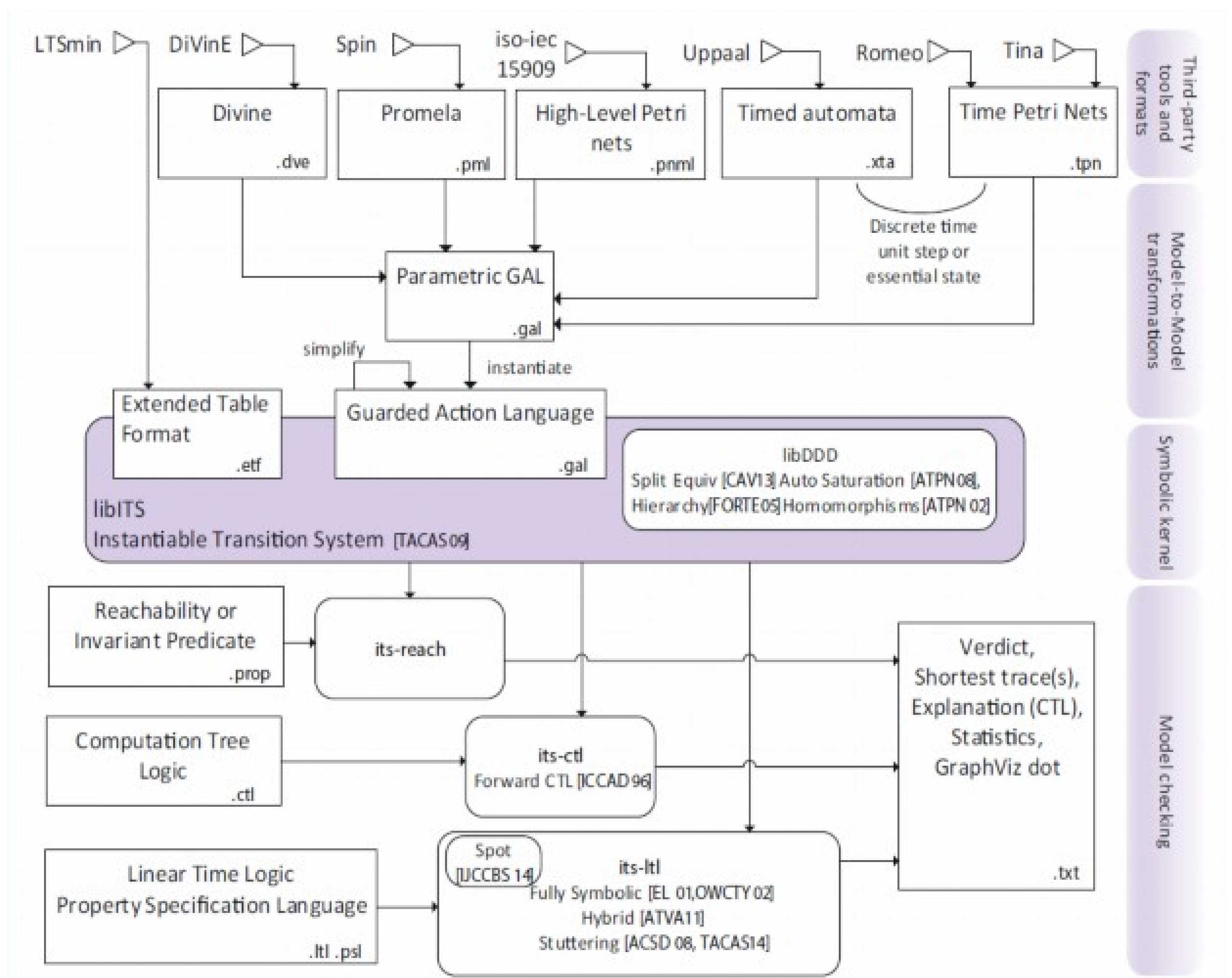


Рис. 1.3. Архітектура ITS-інструментів

На рисунку 1.3 зображено архітектуру ITS-tools, набору інструментів для модельної перевірки. Її можна описати наступним чином:

1. Вхідні формати:

- Підтримується широкий спектр форматів моделей, включаючи мови, що використовуються в популярних засобах модельної перевірки, таких як LTSmin, DiVinE, Spin, Uppaal тощо.

- Кожен формат має своє розширення файлу (.dive, .pml, .xml тощо).

- Всі формати зводяться до єдиної проміжної мови - Parametric GAL (Guarded Action Language).

2. Проміжний шар:

- Parametric GAL - мова високого рівня, яка дозволяє описувати системи з використанням охоронних дій. На цьому рівні виконуються операції спрощення та конкретизації моделей.

- З Parametric GAL модель перетворюється на Guarded Action Language (GAL), який є вхідним формалізмом для ядра ITS-tools.

- Також існує альтернативний шлях перетворення - через Extended Table Format (ETF).

3. Ядро ITS-tools (libITS):

- Засноване на формалізмі Instantiable Transition System (ITS), який дозволяє ефективно представляти та маніпулювати простором станів системи.

- libITS надає функціональність для перевірки різних властивостей:

- Досяжність/інваріантність: перевірка, чи може система досягти певного стану або чи завжди виконується певна умова.

- Властивості CTL (Computation Tree Logic): перевірка властивостей, що стосуються можливих шляхів виконання системи (наприклад, "завжди можливо досягти стану X").

- Властивості LTL (Linear Time Logic): перевірка властивостей, що стосуються окремих шляхів виконання системи (наприклад, "зрештою буде досягнуто стан Y").

4. Зовнішні інструменти та бібліотеки:

- libDDD - бібліотека для роботи з діаграмами рішень.

- Spot - засіб модельної перевірки для LTL.

- Інші інструменти та бібліотеки, що використовуються для символьних обчислень, редукції простору станів тощо.

5. Вихідні дані:

- Результати перевірки властивостей, включаючи:

- Вердикт (виконано/не виконано).

- Найкоротший контрприклад (якщо властивість не виконано).

- Пояснення (для CTL).

- Статистика.

- Візуалізація графу станів (у форматі GraphViz dot).

Загалом, архітектура ITS-tools є модульною та розширюваною, що дозволяє інтегрувати нові формати моделей, методи перевірки та інструменти.

Дослідницька група, яка розробляє та підтримує ITS-інструменти, також допомагає в організації щорічного наукового конкурсу з перевірки моделі, який є подією, присвяченою оцінці інструментів перевірки моделі для паралельних систем. Більшість моделей шашок, представлених у наступних підрозділах, також брали участь. Моделі, які використовуються в конкурсі, найчастіше мають форму HLPN. ITS-інструменти вигідно відрізняються від інших інструментів у конкурсі, будучи на подіумі в більшості категорій конкурсу в останні роки.

mCRL2, що розшифровується як micro Common Representation Language 2, є повноцінним набором інструментів, який надає такі функції, як аналіз, візуалізація, маніпуляції та перевірка розподілених систем. Інструмент приймає специфікації, написані з використанням виразної форми алгебри процесів, заснованої на Алгебрі комунікаційних процесів (ACP), яка розширена для включення даних і концепцій часу. Ця мова введення має багату виразність, оскільки дозволяє користувачам визначати системи за

допомогою різних підтримуваних типів (наприклад, Bool, Pos, Nat, Int, Real) і надає численні вбудовані структури даних, такі як набори, списки та пакети, а також функції і функціональні типи даних. Розширене використання mCRL2 вимагає глибокого розуміння алгебри процесів, тому були написані різні публікації та книги про поширення функцій мови.

На рисунку 1.4 показано типовий робочий процес перевірки моделі. LPS — це процес у звичайній формі, де вся поведінка стану транслюється в параметри даних.

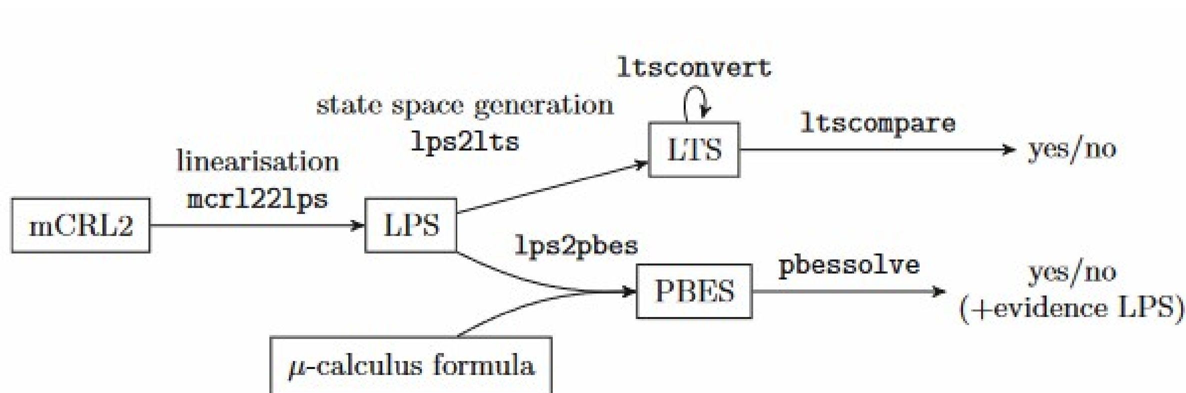


Рис. 1.4. Робочий процес перевірки моделі mCRL2

Вимоги записані в модальному обчисленні, яке є більш виразним, ніж LTL, CTL і CTL*, але воно також має значно більш складну семантику. Специфікація mCRL2 містить кілька процесів, які виконуються паралельно, які перетворюються на лінійну специфікацію процесу (LPS), яка є символічним представленням простору станів, з якого видалено весь паралелізм. Враховуючи лінійний процес і вимогу, проблема перевірки моделі перекладається на систему параметризованих булевих рівнянь (PBES). Рішення цих PBES вказує, чи справедлива формула для вхідного процесу. Система PBES також може бути розв’язана для нескінченних просторів станів за допомогою алгоритмів часткування. Також можна створити LTS з LPS, який можна візуалізувати та взаємодіяти з ним. Крім того, можна виконати скорочення LTS за модулем різних еквівалентностей,

щоб отримати менший LTS, який зберігає властивості вихідної моделі. Цей LTS можна перетворити на LPS для продовження символічної перевірки за допомогою PBES. mCRL2 також надає альтернативний метод перевірки моделі, який називається уточненою перевіркою. Правильність моделі перевіряється шляхом перевірки відношення уточнення між LTS, що представляє реалізацію, і LTS, що представляє специфікацію.

LTSmin спочатку був призначений як загальний набір інструментів для маніпулювання міченими системами переходів, який потім був розширений до символічної перевірки моделі LTL/CTL-числення. Він забезпечує модульну архітектуру, яка дозволяє підключати численні інтерфейси мови моделювання (наприклад, алгебру процесів mCRL2) до різних сімейств алгоритмів аналізу.

LTSmin пропонує розширений аналіз неявних просторів станів за допомогою шпильок (Pins): аналіз досяжності, включаючи виявлення тупикових ситуацій, виявлення дій та перевірку інваріантів/тверджень, а з недавнього часу також перевірку властивостей лінійного часу (LTL) та модального μ-числення. Актуальний огляд LTSmin наведено на рисунку 1.5.

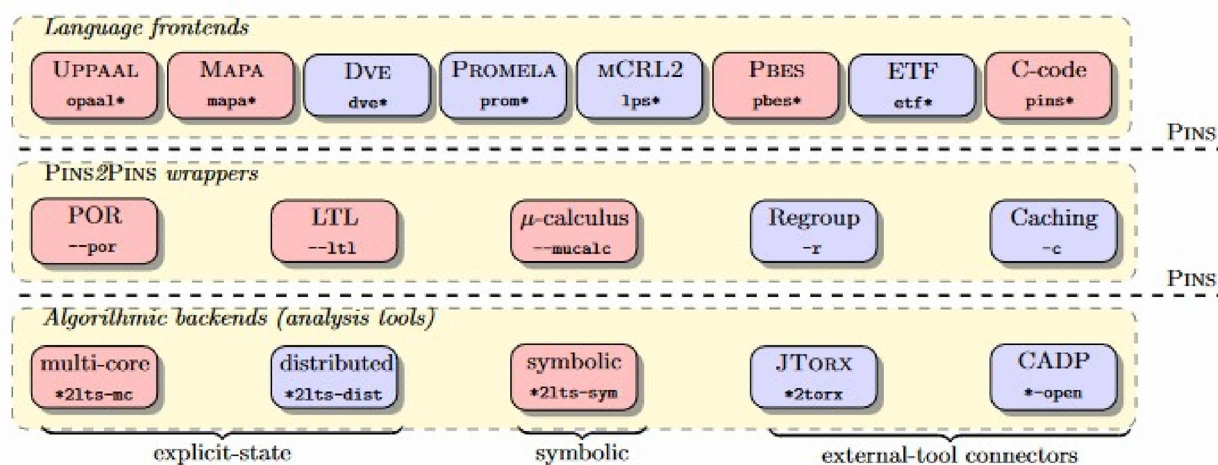


Рис. 1.5. LTSmin архітектура

Цей об'єднуючий інтерфейс називається розділеним інтерфейсом до наступного стану (PINS). Реалізація підтримки нового мовного формату

автоматично забезпечує підтримку різних інструментів, найактуальнішим з яких є перевірка символьної моделі, яка використовує багатоядерний паралельний пакет діаграм рішень Sylvan. Були проведені дослідження багатоядерних реалізацій методів редукції, таких як мінімізація символічної бісимуляції, насичення, а також редукція часткового порядку на льоту. LTSmin не може нативно вводити HLPN, а лише прості мережі Петрі в стандарті PNML.

### Висновки до розділу

У першому розділі було здійснено детальний аналіз предметної області процесів верифікації в контексті предметно-спеціальних моделей. Зокрема, розглянуто основні концепції та методологічні аспекти модельно-орієнтованої інженерії, що є базовими для розвитку технологій верифікації. Охарактеризовано специфіку процесів верифікації в контексті розробки програмного забезпечення. Було визначено значення коректної верифікації моделей, що забезпечує надійність та відповідність системи вимогам.

Розглянуто предметно-орієнтовані мови, які дозволяють створювати спеціалізовані моделі, а також обговорено метамоделі та метаметамоделі, що є основою для налаштування процесів моделювання. Важливу роль відведено трансформаціям моделей, які забезпечують взаємодію між різними рівнями абстракції та платформами.

Також було досліджено платформу моделювання Eclipse, яка є одним із широко використовуваних інструментів для реалізації підходів модельно-орієнтованої інженерії. Було підкреслено її важливість у створенні гнучкого середовища для моделювання та верифікації програмних систем.

У завершальному підрозділі розглянуто інструменти перевірки моделей, що дозволяють забезпечувати коректність розроблених моделей та відповідність заданим вимогам.

# РОЗДІЛ 2. МЕТОДИ МОДЕЛЮВАННЯ, АНАЛІЗУ ТА ВЕРИФІКАЦІЇ КОМПОНЕНТІВ І МОДЕЛЕЙ

## 2.1. Синтаксис і семантика інтерфейсу платформи для моделювання та аналізу компонентів

Платформу ComMA було розроблено для вдосконалення процесу розробки та проектування програмних інтерфейсів для компонентів вбудованої системи. Його головна мета — усунення та запобігання проблемам, які виникають під час інтеграції компонентів (наприклад, після оновлень). Ці проблеми часто виникають через відсутність точного визначення поведінки інтерфейсів, що призводить до неявних припущень щодо базового протоколу зв'язку. Це може призвести до ненавмисних змін у протоколі поведінки, що призведе до проблем із сумісністю, коли компонент інтегрується у більшу систему. ComMA дозволяє визначати специфікації інтерфейсу, а також специфікації компонентів, які містять функціональні обмеження. Специфікації інтерфейсу складаються з трьох частин: сигнатури, автоматів стану поведінки та обмежень даних і часу.

Підпис інтерфейсу визначає набір прийнятих типів повідомлень, які має інтерфейс. Існує три типи повідомлень ComMA:

1) Команди — це синхронні виклики функцій від клієнта до сервера. Клієнт блокується до отримання відповіді від сервера;

2) Сигнали — це асинхронні виклики від клієнта до сервера, які не потребують відповіді сервера;

3) Сповіщення — це асинхронні повідомлення від сервера до клієнта.

Ці типи повідомлень також називаються подіями. Команди, відповіді, сигнали та сповіщення можуть містити один або декілька параметрів різних типів.

У підписі інтерфейсу також можна визначити тип повідомлень. ComMA містить такі стандартні типи даних: bool, int, real, string, bulkdata

(тобто неінтерпретовані масиви байтів) і void (тобто може використовуватися як тип повернення для команд, які не дають результату). Крім того, підтримуються такі складні типи:

■ Визначені користувачем типи, які базуються на стандартних типах даних. Їх можна використовувати для покращення читабельності або для відображення типів, що стосуються домену. Наприклад: тип Pixel на основі int.

■ Типи перерахування фактично є списком літералів перерахування. Вони особливо корисні для визначення керуючих повідомлень, які сервер надсилає клієнту через сповіщення та відповіді.

■ Векторні типи зі змінною кількістю напрямків і необов’язковою довжиною. Їх можна використовувати для моделювання динамічних масивів і матриць довжини, а також більш складних структур.

■ Типи записів, які є колекціями будь-яких типів змінних. Їх можна розглядати як структури, і вони можуть бути корисними у випадках, коли численні події повертають однаковий набір параметрів і, таким чином, можна використовувати один тип запису.

■ Типи карт не можна використовувати як типи повернення для подій, але їх можна використовувати для моделювання статичної інформації, що зберігається на сервері. Для ключів можна використовувати лише основні типи даних і перерахування.

Лістинг 2.1 представлений приклад підпису для камери. CommandStatus — це перелік, який визначає можливі відповіді на команду takePicture, є сигнали для ввімкнення/вимкнення камери (тобто turnOn, turnOff) і її скидання (тобто reset), а також сповіщення про залишок фотографій ( тобто залишок ємності), для низького стану батареї (тобто lowBattery) і для внутрішнього перезавантаження (тобто перезавантаження).

Можна вказати кінцевий автомат(и), який містить стани, які містять переходи, які описують поведінку взаємодії, вказуючи прийнятні послідовності команд, сигналів і сповіщень. Глобальні змінні, їх типи та

початкові значення можна оголошувати та використовувати у виразах. У моделі можна визначити кілька незалежних кінцевих автоматів. У цьому випадку незалежність означає, що кінцеві автомати не можуть взаємодіяти, однакові типи подій не можуть використовуватися в кількох кінцевих автоматах, і ім'я кожного стану з кожного автомата також має відрізнятися.

Лістинг 2.1. Приклад підпису інтерфейсу

```
1  //A signature definition for a Camera component
2  signature Camera
3  //Custom defined types
4  types
5  //An enumeration type used for the reply of the takePicture command
6  enum CommandStatus {
7    Successful = 1
8    Interrupted = 2
9  }
10
11 commands
12 //takePicture takes parameters of type int
13 CommandStatus takePicture (int nrOfPictures)
14 signals
15 turnOn
16 turnOff
17 reset
18
19 notifications
20 remainingCapacity (int number)
21 lowBattery
22 reboot
```

Стан інтерфейсу з кількома кінцевими автоматами задається кортежем, який містить посилання на поточний стан кожного з кінцевих автоматів.

Перш ніж перейти до деталей синтаксису кінцевих автоматів, лістинг 2.2 наведено приклад, який використовує підпис, визначений у лістингу 2.1 і моделює поведінку камери. Кінцевий автомат Camera містить три стани: TurnedOff (який також є початковим станом), TurnedOn і Error. Після отримання сигналу turnOn у стані TurnedOff змінна потужності встановлюється на 20, і поточний стан стає TurnedOn. Після отримання команди takePicture для знімків nrOfPictures виконується захист, який перевіряє, чи запитане число не є нульовим. Якщо караул не тримає, перехід не стріляють. Якщо охорона тримається, недетермінований вибір робиться між двома реченнями, розділеними АБО. Якщо вибрано перше положення, відповідь (CommandStatus::Successful) надсилається клієнту, змінна ємності оновлюється та повертається через сповіщення про залишок ємності , а стан

не змінюється. Якщо вибрано другий пункт, відповідь (CommandStatus::Interrupted) надсилається клієнту, а наступним станом стає Error.

Лістинг 2.2. Приклад кінцевої машини інтерфейсу

```
1  //Global variables with initialization
2  variables
3  int capacity
4  init
5  capacity := 20
6
7  //State Machine with three states
8  machine Camera {
9
10   // The initial state of the Camera state machine
11   initial state TurnedOff {
12
13     // Transition triggered by a signal
14     transition trigger: turnOn
15       do:
16       capacity := 20
17       next state: TurnedOn
18   }
19
20   state TurnedOn {
21
22     // Transition triggered by a command with a parameter
23     transition trigger: takePicture(int nrOfPictures)
24       guard: nrOfPictures != 0
25         do:
26         reply(CommandStatus::Successful)
27         if (capacity > nrOfPictures) then
28           capacity := capacity - nrOfPictures
29           remainingCapacity(nrOfPictures)
30         fi
31         next state: TurnedOn
32
33       // Used to determine non-determinism
34       OR
35         do:
36         reply(CommandStatus::Interrupted)
37         next state: Error
38
39     transition trigger: turnOff
40       next state: TurnedOff
41
42     // Non-triggered transition
43     // The body of the transition must contain a notification
44     transition
45       do:
46       lowBattery
47       next state: TurnedOff
48   }
49
50   state Error {
51
52     transition trigger: reset
53       do:
54       reboot
55       next state: TurnedOn
56
57     transition trigger: turnOff
58       next state: TurnedOff
59   }
60 }
61
62 // The following code is present in a separate parameter model
63 // It specifies the possible starting values of the nrOfPictures parameter
64 interface: Camera
65 // Each combination of a parametrized trigger and a state needs to be treated
66 trigger: takePicture
67 state: TurnedOn
68 params: (1)
69 params: (2)
70 params: (3)
```

У стані TurnedOn також можливе отримання сигналу про вимкнення або надсилання сповіщення про низький заряд батареї, в обох випадках

наступним станом стане TurnedOff. У стані Error можна отримати сигнал скидання, надіслати сповіщення про перезавантаження, а наступним станом буде TurnedOn. Крім того, можна отримати сигнал вимкнення, а наступним станом буде вимкнено.

Для кожного кінцевого автомата існує рівно один початковий стан. Також можна визначити блок у всіх станах , де містяться переходи можуть відбуватися в будь-якому стані (або можна вказати набір станів). Стани містять ініційовані та неініціовані переходи. Ініційовані переходи пов’язані з командою чи сигналом, тоді як неініціовані переходи можуть бути ініційовані сервером недетерміновано, але вони повинні мати принаймні одне сповіщення в своєму тілі. Ось такі конструкції, які можна використовувати для визначення тіла переходів:

- Тригер, який є або сигналом, або командою. Для переходів, ініційованих командою, має бути відповідь у тілі переходу. Якщо тіло містить оператор if-else і немає відповіді, визначеної поза самим оператором if-else, тоді всі гілки оператора if-else повинні мати відповідь. Переходи представляють моделі зв'язку між клієнтом і сервером певного інтерфейсу. З цієї точки зору тригери представляють повідомлення, надіслані клієнтом, тоді як тіла містять повідомлення, надіслані сервером, та інші потенційні обчислення, які виконуються на стороні сервера. Область параметра, пов’язаного з сигналом або командою (наприклад, параметр nrOfPictures для команди takePicture у лістингу 2.1) є локальним для переходів, тому будь-які призначення, виконані для параметра, відкидаються після обробки переходу. Значення цих параметрів надходять з окремого файлу параметрів, де користувачі повинні вказати серію можливих значень параметрів для кожного активованого переходу для кожного стану.

- Додатковий guard, який є логічним виразом для глобальних змінних, а також змінних параметрів із сигналів і команд. Якщо предикат guard не виконується, перехід не запускається.

■ Для вираження недетермінізму можна використовувати блок АБО , який дозволяє визначити кілька блоків дій (наведено далі). Для одного переходу можна використовувати декілька конструкцій АБО.

■ Блок do містить серію дій, що виконуються послідовно. До них належать:

□ Присвоєння виразу змінній.

□ Можливо вкладений оператор if-else .

□ Шаблон сповіщення – це подія сповіщення, доповнена обмеженнями на кількість випадків. Вони визначаються такими символами: ? - необов'язково, + - принаймні один раз, * - нуль або більше, або фактична кількість разів, яку потрібно переглянути. Якщо шаблон входження не використовується, сповіщення має відбутися рівно один раз.

□ Відповідь , яка містить параметр, який має той самий тип, що й тип повернення пов’язаної команди. Якщо конкретне значення не повертається, замість параметра використовується символ підстановки *, а якщо тип команди недійсний, жодні параметри не включені у відповідь.

□ Речення where із виразом, що містить значення, що містяться у відповіді чи сповіщенні.

□ Шаблон будь-якого порядку , який є набором сповіщень, відповідей і фрагментів . Фрагмент — це набір повідомлень, які можуть бути оголошені перед визначенням кінцевого автомата. Фрагменти можуть визначати серію серверних подій, які зазвичай надсилаються групою. Події з шаблону будь-якого порядку можуть чергуватися в будь-якому порядку, але необхідно враховувати всі можливі шаблони сповіщень, пов’язані з окремими сповіщеннями.

Обмеження щодо даних і часу, і вони базуються на командах, сповіщеннях, сигналах і відповідях. При визначенні цікавих подій також можна вказати стан, у якому вони спостерігаються, а також умову для змінних, присутніх в інтерфейсі (наприклад, у стані TurnedOn notification remainingCapacity, коли ємність > 10). Після цього можна створити правила

синхронізації, які накладають нижні та верхні межі на допустимі інтервали між подіями. Так само обмеження даних використовуються для визначення очікуваної послідовності подій, а також обмежень на спостережувані параметри в послідовності. Синтаксис і семантика цих обмежень докладно не представлені, оскільки вони не є предметом цього проекту.

## 2.2. Створення артефактів верифікації засобами платформи моделювання

Можна створити наступні артефакти, як показано на рисунку 2.1:

- Починаючи з моделі ComMA, можна створити генератор тестів, який дозволяє автоматично створювати тестові випадки Java (JUnit), які потім можна використовувати для перевірки існуючих реалізацій інтерфейсів. Для його генерації користувачеві необхідно надати вхідні параметри для команд і сигналів, а також адаптер для взаємодії з SUT. Таким чином, замість того, щоб вручну писати нові тести після зміни інтерфейсу, ComMA може спростити це завдання. Цю концепцію ілюструє одна з крилатих фраз, які використовуються для просування ComMA - «Нехай тестові приклади пишуться самі».
- Подібним чином існує функція експериментального тестового клієнта. Основна відмінність від тестового генератора полягає в тому, що послідовність дій тестового клієнта є недетермінованою та визначається під час виконання, можливо, залежно від відповідей від SUT. Це відрізняється від згенерованих тестів, де послідовність дій і відповіді від SUT заздалегідь визначені. Тестовий клієнт має працювати у фоновому режимі протягом кількох годин, і він може виявляти потенційні умови перегонів.
- Новим підходом є створення моніторинг. Моніторинг — це процес перевірки відповідності спостережуваної поведінки SUT його специфікаціям. Таким чином, монітор записує траси виконання з SUT (наприклад, отримані шляхом реєстрації перехоплення), і він перевіряє, чи

траси відповідають багатьом залученим протоколам кінцевого автомата, їхнім даним, синхронізації та функціональним обмеженням на основі стану. Загальноприйнятим є те, що сліди записуються протягом більш тривалого періоду часу, тоді як монітор повідомляє про будь-які можливі невідповідності.

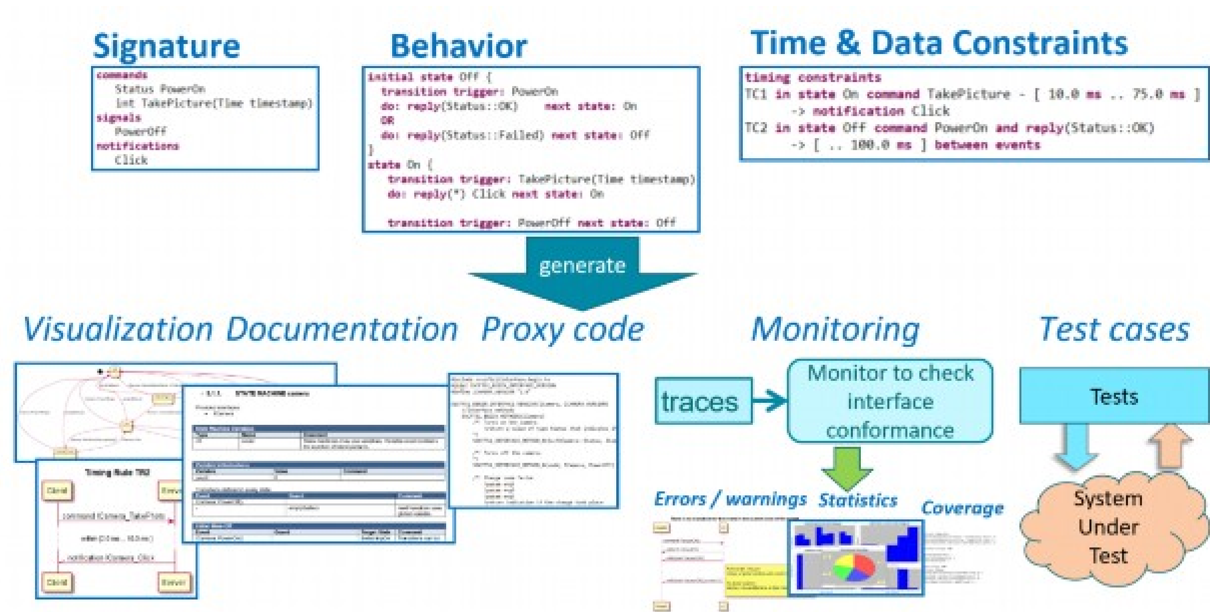


Рис. 2.1. Згенеровані артефакти ComMA

Спрощений неофіційний опис алгоритму моніторингу компонента такий:

□ Для кожного порту монітор інтерфейсу зберігає набір поточних шляхів проходження, які отримані шляхом споживання спроектованих подій на цьому порту та виконання прогресу для кінцевого автомата інтерфейсу. Оскільки моделі інтерфейсу мають недетерміновану поведінку, одна й та сама траса події може перевести модель інтерфейсу в різні стани, вибираючи різні шляхи обходу. Таким чином, монітор інтерфейсу досліджує всі можливі шляхи обходу. Якщо для заданого шляху наступна спостережувана подія не дозволена в поточному стані інтерфейсу, тоді шлях відкидається. Якщо всі

шляхи відхилено, виявляється помилка моніторингу інтерфейсу, і монітор компонентів більше не перевіряється.

□ У компоненті може бути кілька портів, кожен з яких має монітор інтерфейсу, який підтримує кілька шляхів проходження. Таким чином, монітор компонентів перевіряє функціональне обмеження на основі стану, використовуючи лише події з блоку подій використання та всі комбінації шляхів проходження від усіх моніторів інтерфейсу з портів. Помилка моніторингу компонента виявляється, якщо немає комбінації шляхів обходу, за якими можна обробити наступну спостережувану подію.

■ Інтерактивні симулятори можна генерувати як для окремих інтерфейсів, так і для компонентів, і їх можна використовувати, щоб перевірити, чи модель має очікувану поведінку, а також для випадків, коли присутня помилкова поведінка, і необхідно визначити першопричину цієї помилкової траси. ідентифіковані.

■ Нарешті, можливе створення документації для інтерфейсів і візуалізацій кінцевих автоматів і обмежень, що може допомогти користувачам у процесі підтримки та оновлення документації.

### *2.2.1. Можливості перевірки та верифікації моделей*

Перш ніж використовувати моделі ComMA для створення тестових наборів, тестових клієнтів і моніторів, які допомагають у життєвому циклі інтерфейсів компонентів, необхідно переконатися в узгодженості та правильності цих моделей. Припустимо, модель з небажаною поведінкою використовується для різних завдань генерації. У цьому випадку неможливо гарантувати застосовність результатів цих методів при застосуванні до SUT. Таким чином, не можна дати жодних гарантій якості реалізації. Таким чином, для того, щоб ComMA надавав усі потенційні переваги кінцевому користувачеві, створення узгоджених моделей із очікуваною поведінкою є надзвичайно важливим.

Ручне моделювання можна використовувати для підтвердження того, що моделі мають очікувану поведінку, але це не вичерпний метод. Таким чином, ComMA також пропонує легкі можливості перевірки, перевіряючи, що моделі інтерфейсів відповідають певним властивостям безпеки. Перш ніж перейти до технічних деталей, наведемо типи перевірених властивостей. Перші п'ять властивостей передбачають послідовне виконання для дій, що містяться в переході інтерфейсу (тобто семантика від запуску до завершення для переходів). Це означає, що дії клієнта (тобто сигнали та команди) і дії сервера (тобто дії в тілі переходу, відповіді та сповіщення) виконуються в послідовності, представленій у моделі.

- Недосяжні стани - це стани, до яких неможливо дійти з початкового стану.
- Тупикові стани – це доступні стани, з яких переходи неможливо зробити.
- Стани прийому – це доступні стани, які не можна залишити після досягнення. Відмінність від взаємоблокувань полягає в тому, що стани приймача приймають самоцикли.
- Стани Livelock — це доступні стани, з яких неможливо досягти жодного з домашніх станів. Домашній стан — це кортеж, що містить стан для кожної машини в інтерфейсі (якщо інтерфейс містить декілька паралельних кінцевих автоматів). Користувач може вказати початкові стани або, альтернативно, будуть використані початкові стани кінцевих автоматів.
- Стани вибору – це доступні стани, з яких можна виконати як запущений, так і не запущений перехід.

У системі реального світу клієнт і сервер інтерфейсу також спілкуються через канал, який може мати різні рівні надійності. Таким чином, останні три властивості припускають одночасне виконання між клієнтом і сервером. Клієнт і сервер розвиваються незалежно, кожен з них підтримує окремий стан. Це передбачає більший і складніший простір станів. Однак це корисно для того, щоб гарантувати, що клієнти та сервер

залишаються синхронізованими та мають однаковий стан, незважаючи на те, що вони розвиваються одночасно.

■ Умови змагання між клієнтом і сервером виникають, коли клієнт і сервер можуть здійснювати різні переходи, закінчуючи станом, з якого вони більше не можуть взаємодіяти. Помилки для умов змагання клієнта виникають, коли клієнт потрапляє в стан, коли він більше не може обробити повідомлення, отримане від сервера, і навпаки для умов змагання сервера. Засіб перевірки моделі також видає попередження - коли клієнт і сервер переходять у інший стан або коли клієнт використовує параметр, який сервер не може обробити.

■ Проста плутанина виникає, коли той самий сигнал або команда використовується для кількох переходів у стані. Для прикладу, якщо сервер отримує таку подію, він може недетерміновано вибрати тіло для переходу, відмінне від тіла переходу на стороні клієнта, звідки спочатку було надіслано сигнал або команду.

■ Загальна плутанина також припускає, що повідомлення на каналі не отримуються стороною, що спілкується, не в тому порядку, в якому вони були надіслані. Для прикладу, клієнт може надіслати два сигнали, отримані на сервері в неправильному порядку.

ComMA створює інтерактивний звіт про перевірку, який можна використовувати для перевірки результатів для властивостей, а також діаграми, які ілюструють контрприклади.

*2.2.2. Обмеження поточної методології верифікації*

Обмеження поточної методології верифікації полягають у наступному:

■ Перевірка може бути виконана лише в обмеженому порядку.

Створення графіка доступності залежить від глибини дослідження, яку встановлює користувач. Генерація припиняється або коли

1) досягнуто глибини дослідження,

2) коли всі місця HLPN відвідано один раз.

Якщо глибина дослідження досягнута, створений звіт про перевірку повідомить користувача, що властивості не були перевірені, оскільки модель не була охоплена. Якщо глибина дослідження достатньо велика, щоб усі місця з моделі HLPN були відвідані один раз, перевірки будуть виконані, а результати будуть передані користувачеві. Однак це обмежує, оскільки весь простір станів HLPN повинен враховувати всі можливі доступні позначки. Важливим наслідком тут є те, що результати перевірки не є справді надійними, навіть якщо ComMA каже, що вся специфікація була покрита (оскільки насправді це могло бути не так).

- Перевірку моделі не можна застосувати до моделей компонентів ComMA із функціональними обмеженнями.

Простори станів компонентів ComMA мають навіть більшу складність, ніж окремі інтерфейси. Це тому, що компонент містить кілька інтерфейсів і підкомпонентів, кожна з яких має власну поведінку. Семантика виконання компонента полягає в тому, що кожна підчастина може розвиватися незалежно, а функціональні обмеження використовуються для забезпечення порядку між певними подіями, які надходять з різних частин, що також може збільшити складність простору станів.

- Перевірка нових типів властивостей вимагає впровадження нових процедур перевірки моделі.

Усі поточні властивості мають власну процедуру перевірки моделі, яка працює на графіку досяжності. Таким чином, користувачі не можуть визначати власні властивості.

Таким чином, наявність максимальної глибини дослідження для моделей інтерфейсу, неможливість перевірити узгодженість моделей компонентів із функціональними обмеженнями та нездатність визначити нові типи властивостей представляють значні обмеження поточного підходу. Якщо ці обмеження буде подолано за допомогою найсучасніших технологій, кінцевий користувач ComMA може отримати значні переваги.

Ці переваги випливають із можливості перевіряти нові типи властивостей у всьому просторі станів для більш складних моделей (тобто моделей компонентів). Ці переваги підвищать впевненість користувача в тому, що модель справді має очікувану поведінку. Таким чином, переваги, які надходять від таких методів, як моніторинг і генерація тестів, які допомагають у процесі оновлення та підтримки інтерфейсів компонентів для дуже складних систем програмного забезпечення, можуть бути впевнені, оскільки може бути гарантована узгодженість моделей генераторів.

### 2.3. Підходи до інтеграції засобів перевірки моделей

PNML служить стандартом для опису всіх типів мереж Петрі, включаючи HLPN, на які семантику ComMA було зіставлено за допомогою . Численні засоби перевірки моделей приймають документи PNML нативно, і тому потенційне відображення в PNML може дозволити інтегрувати кілька засобів перевірки моделей у простий спосіб (тобто ITS-інструменти, TAPAAL, TINA та SMPT підтримують HLPN у форматі PNML, тоді як LTSmin підтримує прості мережі Петрі у форматі PNML).

Існують також інші інструменти (наприклад, для візуалізації, маніпулювання графіками та аналізу), які приймають документи PNML, і, отже, дозвіл ComMA експортувати такі документи може дозволити численні інші взаємодії та робочі процеси, які можуть представляти майбутню роботу. Таким чином, були вивчені інструменти, які дозволяють створювати та маніпулювати документами PNML:

■ PNML Framework — це API з відкритим вихідним кодом на основі Eclipse, який дозволяє створювати документи PNML. Його творці беруть участь у розробці стандарту ISO/IEC 15909 і прагнуть підтримувати найновішу бібліотеку, що відповідає стандартам.

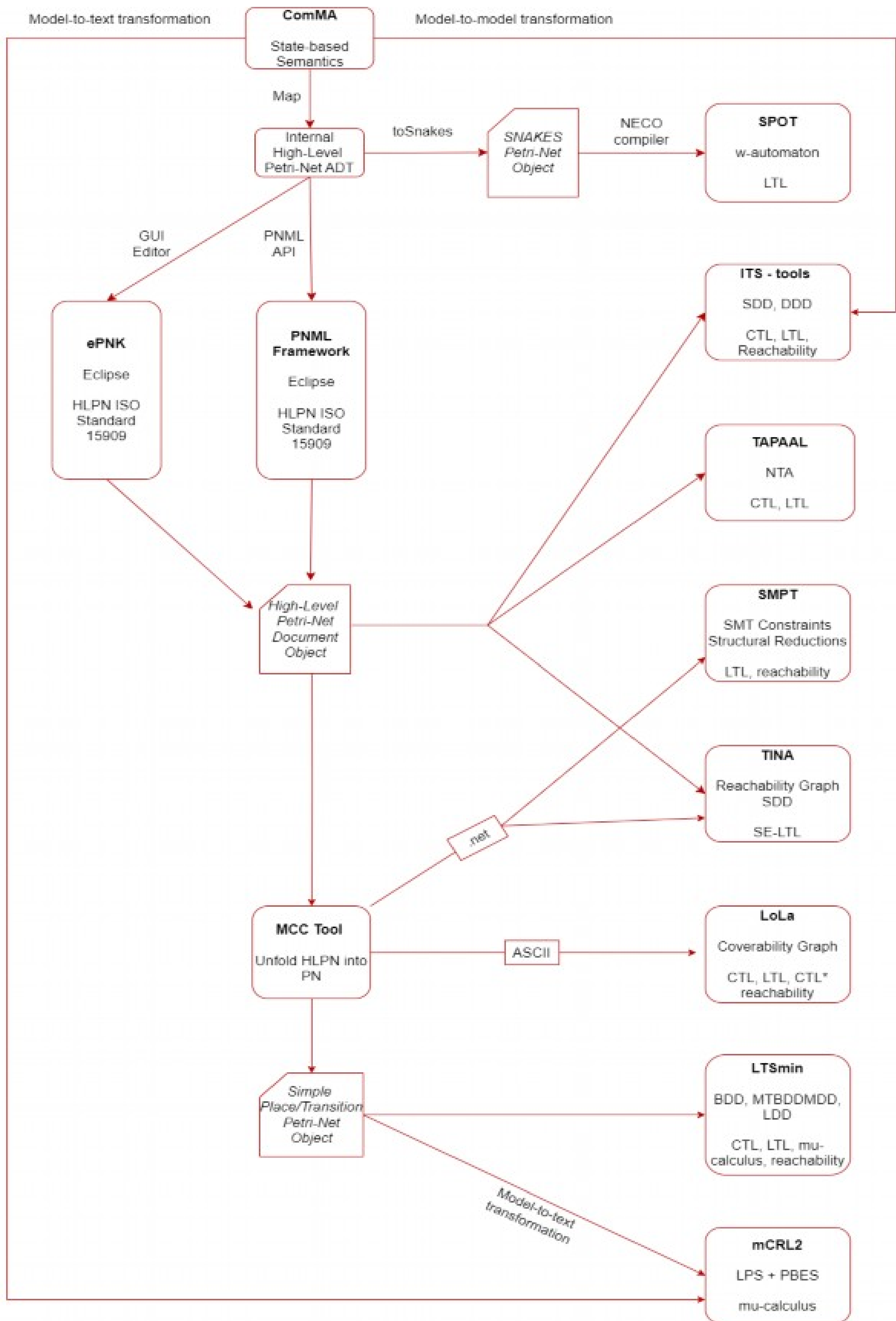


Рис.2.2. Підходи до інтеграції засобів перевірки моделей із моделями ComMA

- ePNK — це ще одна бібліотека Eclipse, яка має на меті побудувати поверх PNML Framework шляхом додавання GUI, за допомогою якого можна моделювати мережі Петрі. Він також використовує технологію EMF, щоб дозволити розробникам створювати нові типи мереж Петрі, які потім можна зіставляти з HLPN.
- Інструмент MCC приймає як вхід високорівневу мережу Петрі у форматі PNML, розгортає її (тобто видаляє всі функції високого рівня) і може виводити просту мережу Петрі у будь-якому PNML, яку можна імпортувати. LTSmin, формат .net, прийнятий TINA, або власний формат ASCII LoLa.

Однак також можливо, що PNML надає деякі обмеження на підмножину ознак, які можуть бути виражені. Якщо це так, слід виконати пряме відображення на прийнятну мову специфікації засобу перевірки моделі. На рисунку 2.2 представлені всі варіанти інтеграції, які були представлені досі.

## 2.4. Відображення моделей з використанням мережі Петрі

Документація щодо цієї внутрішньої трансформації недоступна, тому було вивчено вихідний код і отримано результати трансформації з різних інтерфейсів ComMA, які зосереджені на різних простих мовних функціях. З цих інтерфейсів було згенеровано відповідний код Python, який використовувався для створення візуалізацій отриманих мереж Петрі за допомогою бібліотеки GraphViz. Реверсивна інженерія цього перетворення призвела до відкриття різних закономірностей. Найзагальніші аспекти цього перетворення детально описані в решті розділу.

Нижче наведено різні структури Python, які використовуються як маркери в мережі Петрі:

- g — це структура Python, яка містить усі глобальні змінні, визначені в інтерфейсі ComMA. Один токен типу g спочатку створюється в

місці V. Наступні токени типу g створюються лише після того, як у мережі Петрі змодельовано весь перехід ComMA.

- l — це структура Python, яка містить єдиний параметр, який походить від запущеного переходу. Спочатку всі місця типу P, пов'язані з різними ініційованими переходами, можуть містити один або більше маркерів типу l, залежно від кількості параметрів, наданих користувачем ComMA.
- t і ” є заповнювачами для простих токенів. Вони використовуються лише для відстеження поточного стану ComMA у мережі Петрі .
- gl — це структура Python, яка містить як g, так і l. Він створюється функцією g.combine(g, l), яка приймає як вхідні дані маркер g разом із поточним маркером l і повертає структуру gl. Для кожного можливого токена l кінцевий токен gl буде різним, і це має важливий вплив на перетворення до PNML. Іншими важливими функціями є gl.globals(), яка витягує частину g з елемента gl, щоб передати її до місця V після моделювання всього переходу ComMA. Нарешті, gl.e() може приймати як вхідні дані будь-який вираз Python над глобальними змінними, що містяться в частині g, і параметром із поточної частини l. Після обчислення виразів змінні в gl оновлюються.

Тепер, коли було пояснено типи токенів, наступна процедура описує, як конструкції ComMA зіставляються з конструкціями Мережі Петрі .

1. Створено місце V , яке містить маркер, який містить усі глобальні змінні зі специфікації інтерфейсу ComMA. Це місце приймає жетони типу g.
2. Для кожного стану ComMA створюється відповідне місце із префіксом S. Це місце приймає прості токени типу t.
3. Для кожного запущеного чи не запущеного переходу ComMA, для кожного стану ComMA створюється перехід із префіксом T.

a) У разі запущеного переходу створюється місце із префіксом P. Це місце містить кілька токенів, які відповідають різним значенням, які

враховуються для певного сигналу або команди ComMA, які вказані у файлі параметрів. Це місце приймає жетони типу l.

б) Перехід, що відповідає запущеному переходу ComMA, має вхідні дуги з V, відповідного місця P і відповідного місця S (рис. 2.3). Таким чином, для запуску маркер споживається з S (таким чином, стан ComMA, пов'язаний із S, має бути поточним станом кінцевої машини), маркер споживається з P (таким чином, змінна параметра може бути використовується в охороні під час переходу ), а токен споживається з V (отже, глобальні змінні можуть бути використані в охороні під час переходу).

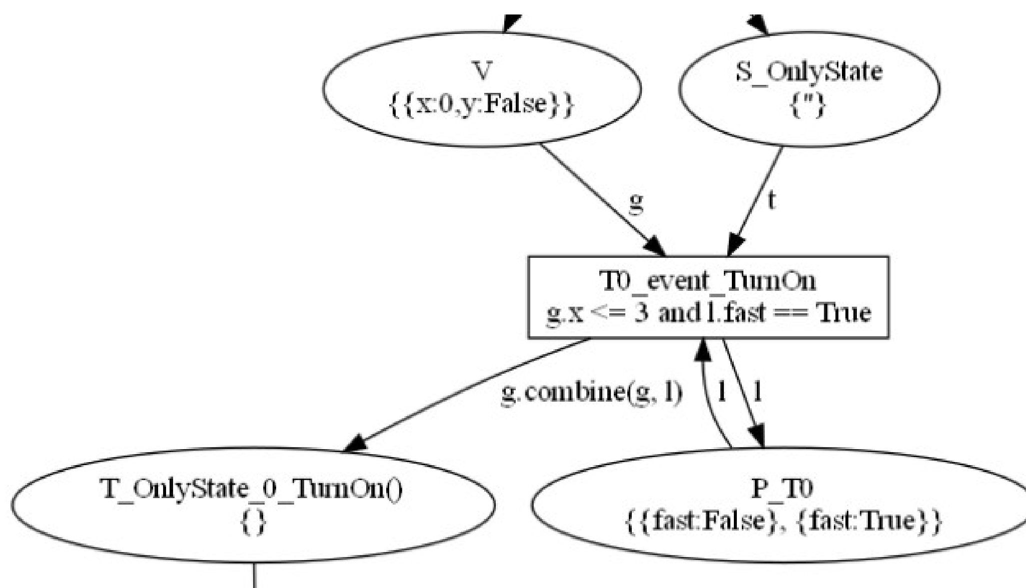


Рис. 2.3. Приклад шаблону, який використовується для відображення параметризованих тригерних переходів

в) Подібним чином перехід , що відповідає неініційованому переходу ComMA, має вхідні дуги з місця V, відповідного місця S, але не з місця P (оскільки ці переходи не потребують сигналу чи команди, а отже, немає параметрів і місця P створюється).

г) Якщо перехід ComMA містить охорону, вона буде перетворена на охорону ЗМІЙ усередині переходу ЗМІЇ. guard містить вираз, який включає глобальні змінні та потенційні параметри.

д) Перехід має вихідну дугу до відповідного місця з префіксом T, яке пояснюється далі.

5. Усі вирази та конструкції всередині тіла переходу ComMA відображаються на ланцюжок місць із префіксом C. Це місце приймає маркери типу gl. Між кожним місцем C створюються відповідні переходи з різними сценаріями, описаними нижче.

а) Вирази ComMA зіставляються з виразами на вихідній дузі між переходом і наступним місцем C.

б) if-else відображаються подібно до конструкції OR. З поточного місця C виконується кілька переходів із захисниками, що відповідають різним гілкам оператора if-else у ComMA. На вихідній дузі між таким переходом і наступним місцем C вирази ComMA, що відповідають поточній гілці if-else, можуть бути відображені як вирази . Вихідні дуги різних переходів з'єднані з одним місцем C, яке об'єднує різні гілки виконання. На рисунку 2.4 наведено приклад.

в) відповіді відображаються на беззахисні переходи між двома місцями C. Якщо відповідь у специфікації ComMA має параметр, це не буде оброблено під час перетворення.

г) Сповіщення зіставляються з беззахисним переходом між двома місцями C. Якщо сповіщення в специфікації ComMA має параметр, це не буде оброблено під час перетворення.

і. Шаблони сповіщень транслюються на комбінацію циклів переходу з охоронцями та лічильниками. Шаблон принаймні один раз (+) транслюється в цикл переходу, який підраховує кількість разів, коли цей перехід виконується. З того самого місця C виконується ще один перехід до наступного місця C , яке має охорону, яка контролює, що цикл було виконано принаймні один раз (і коли вказано фіксовану кількість разів, охорона відображатиме це натомість). Подібним чином шаблон можливості (?) перетворюється на петлю переходу ЗМІЇ, яку можна здійснити щонайбільше один раз, і додатковий беззахисний перехід до наступного місця C ЗМІЇ, який

можна зробити завжди. Нуль або більше (*) перетворюється на беззахисний цикл переходу , а додатковий беззахисний перехід на наступне місце С.

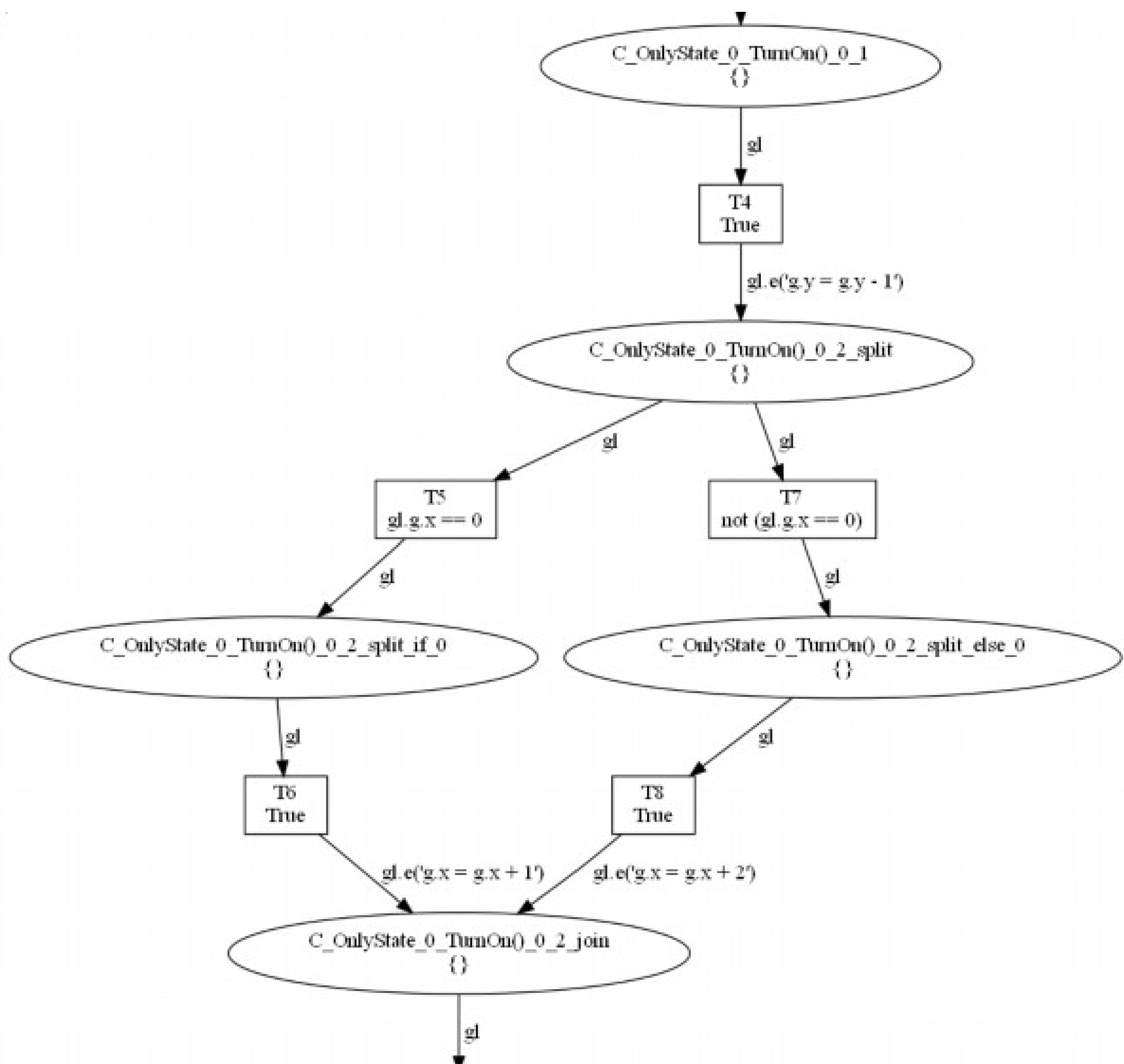


Рис. 2.4. Приклад шаблону, який використовується для відображення дій if-else

ii. Шаблон будь-якого порядку перетворюється шляхом створення переходу , який має вихідні дуги до кількох місць С, кожне з яких відповідає сповіщенню в конструкції будь-якого порядку. Створюючи кілька вихідних дуг з одного переходу, маркери створюються в кожному місці С . Після цього для кожного місця С створюється перехід, пов’язаний із певним сповіщенням ComMA, який пов’язується з відповідним місцем С. Створюється останній

додатковий перехід , який має вхідну дугу з кожного місця C для об’єднання потоку керування. Ця конструкція дозволяє виконувати сповіщення в будь-якому порядку, але всі вони мають відбутися перед продовженням. Фрагменти відображаються подібно до шаблону будь-якого порядку.

6. Після того, як усі вирази всередині тіла переходу будуть зіставлені, конструкція остаточного наступного стану всередині тіла переходу ComMA обробляється наступним чином: з останнього місця C створюється перехід, який має дві вихідні дуги. Одна з вихідних дуг з’єднана з правильною позицією S, яка стосується наступного відповідного стану ComMA, а інша дуга з’єднана з унікальною позицією V. Таким чином, після завершення повного переходу ComMA глобальні змінні оновлюються.

Наведені вище деталі перетворення пояснюють, як створюється мережа Петрі, анотації дуг і умови переходу. Оскільки Petri Nets можуть використовувати довільні вирази Python, оголошення не обробляються, оскільки операції не потребують попереднього.

## 2.5. Особливості, компоненти та архітектура ITS-інструментів верифікації символьної моделі

Мова захищених дій (GAL) є основним формалізмом, який нативно підтримується ITS-інструментами перевірки символьної моделі. GAL є метою трансформації моделі в модель від ComMA, що є одним із головних внесків цієї дисертації. У цій главі спочатку будуть представлені особливості ITS-інструментів, їх високорівневі компоненти та архітектура, а також короткий опис їх серверної технології та алгоритмів. Однак головна увага в цьому розділі — деталізація GAL, особливо щодо його синтаксису та семантики для простого типу GAL і для складеного GAL, а також різних типів властивостей, які можна виразити та перевірити.

ITS-tools пропонується як окремий інструмент на основі Eclipse (однак його алгоритми написані на C/C++), який пропонує символічну перевірку

мультиформалізму. Він пропонує можливості перевірки моделей для різноманітних властивостей (тобто доступності, CTL, LTL) у великих одночасних специфікаціях. Розробники інструменту визначили фокус як перевірку великих глобально асинхронних – локально синхронних специфікацій, ідеального контексту для одночасних компонентних систем. Ці специфікації можуть бути виражені в широкому спектрі підтримуваних формалізмів введення: мови, орієнтовані на комунікаційні процеси (наприклад, мова DVE DiVinE; мова Promela SPIN), мережі Петрі високого рівня (тобто описані в стандарті PNML) , специфікації дискретного часу (тобто автомати з часом, як визначено UPPAAL; часові мережі Петрі, як визначено TINA); специфікації з використанням семантики символічної структури Kripke (тобто вбудованого символічного формату ETF LTSmin). Основним спеціальним формалізмом загального призначення, який нативно пропонується ITS-інструментами, є Guarded Action Language (GAL). GAL був спеціально розроблений, щоб бути гнучким для вираження паралельної семантики. Внутрішньо більшість прийнятих сторонніми формалізмами відображаються на семантику GAL, яка потім використовується як вхідні дані для перевірки моделі. Винятком є формат ETF, який має нативний адаптер.

ITS-інструменти містять різноманітні бібліотеки, які також можна використовувати як окремі інструменти. libDDD описується як символічне ядро, яке охоплює представлення станів і переходів. Це бібліотека C++, розроблена для ефективного керування діаграмами рішень. Ця бібліотека підтримує гнучке та потужне кодування операцій для діаграм прийняття рішень (DDD) із цілочисельними значеннями, а також для ієрархічних діаграм прийняття рішень (SDD). Він також підтримує гомоморфізми, які можуть кодувати складні високорівневі вирази відношення переходу, які також можна розглядати як символічні переходи. Гомоморфізми можуть бути ефективно застосовані до великих наборів станів.

libITS — це бібліотека C++, яка пропонує інтуїтивно зрозумілий API, який можна використовувати для визначення масштабованих алгоритмів перевірки символьної моделі, які використовують ефективні представлення діаграми рішень і гомоморфізми libDDD. libITS дозволяє створити ієрархічну структуру систем, яку можна охопити за допомогою системи миттєвого переходу (ITS) [2]), яка описана нижче. Ця бібліотека може вводити різні формалізми, які автоматично кодуються в символьному представленні з пов'язаним символічним наступним відношенням. Таким чином, libITS дозволяє швидко створювати прототипи нових алгоритмів перевірки моделі, надаючи розробникам потужні гомоморфізми та операції з фіксованою точкою, одночасно піклуючись про низькорівневе символьне кодування.

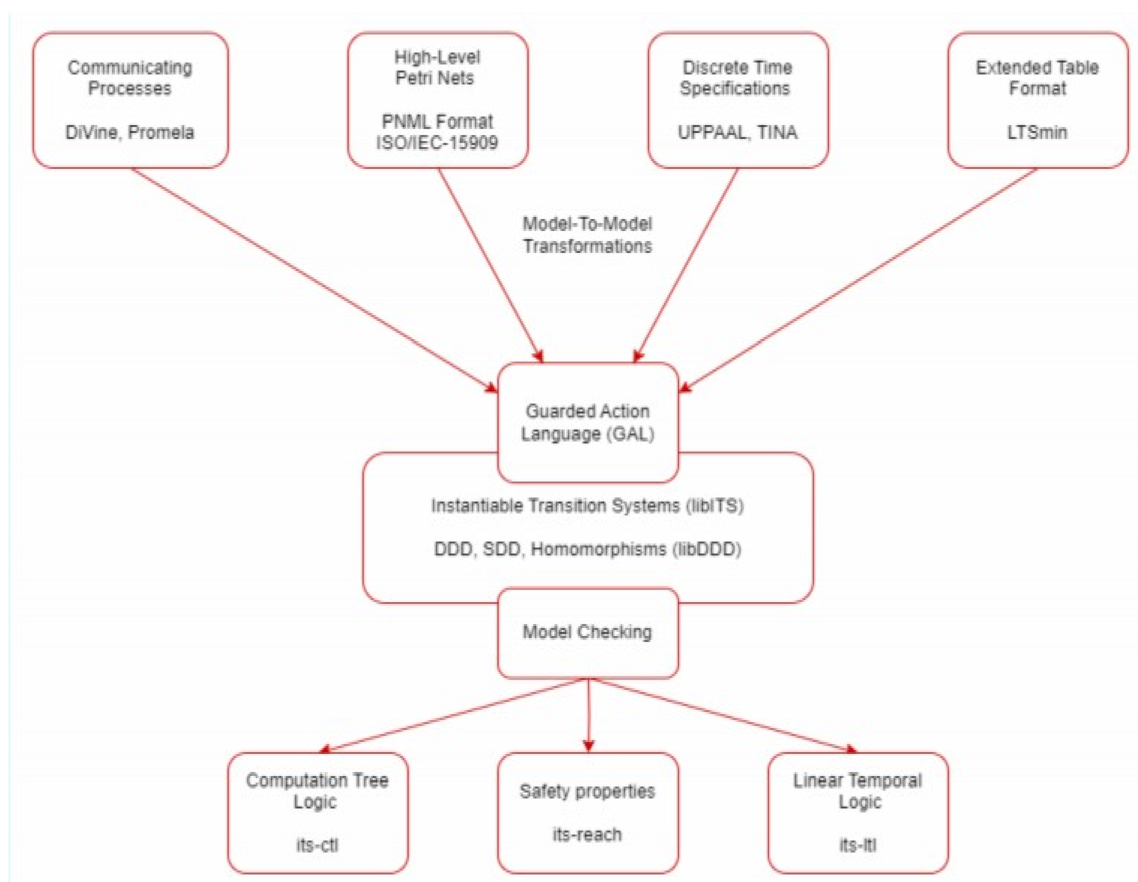


Рис. 2.5. Архітектура ITS-інструментів

Використовуючи API, запропонований libITS, було розроблено та включено три незалежні інструменти перевірки моделі. its-reach обчислює

досяжні стани, а також один або декілька шляхів-свідків до цільових станів, які задовольняють булевий предикат. Для формалізмів із дискретним часовим контекстом інструмент також може обчислювати найкращі та найгірші часові межі. Нарешті, він може виконувати перевірку обмеженої моделі властивостей досяжності. its-ctl виконує перевірку властивостей CTL, за винятком обмежень справедливості. its-ltl виконує гібридну, а також повністю символічну перевірку властивостей LTL і використовує бібліотеку SPOT.

ITS-інструменти містять багаторівневу архітектуру. Верхній рівень представлений прийнятими мовними введеннями з різних сторонніх інструментів. Перетворення від моделі до моделі використовуються для створення екземплярів вхідних моделей мовою GAL. Символьне ядро використовує операції діаграми рішень, запропоновані libDDD, а також API, визначений libITS, і пропонує можливість розробки ефективних алгоритмів перевірки. Нарешті, три різні інструменти перевірки моделі використовуються для перевірки безпеки, CTL і LTL формул.

## Висновки до розділу

У другому розділі було розглянуто методи моделювання, аналізу та верифікації компонентів і моделей, що становить основу для забезпечення якості програмного забезпечення на різних етапах його розробки. Описано основні принципи створення інтерфейсу, який забезпечує ефективну роботу з моделями компонентів, а також аналіз їхньої структури та функціональності. Досліджено процес створення артефактів, необхідних для верифікації, і досліджує можливості, які надають сучасні платформи для автоматизованої перевірки та верифікації моделей. Водночас було підкреслено обмеження поточної методології, які можуть впливати на повноту верифікації.

Розглянуто різні методи об'єднання інструментів для перевірки моделей, що дозволяють досягти більш високої гнучкості та точності

верифікації. Також було досліджено застосування мережі Петрі для відображення моделей. Цей метод дозволяє формалізувати поведінкові аспекти системи та забезпечити наочне представлення процесів, що особливо корисно для верифікації складних моделей.

На завершення розділу розглянуто особливості, компоненти та архітектуру ITS-інструментів, що використовуються для верифікації символьних моделей. Описано їхню структуру та функціональність, що є важливими для підтримки формальної перевірки моделей у реальних проєктах.

# РОЗДІЛ 3. МОДЕЛІ ТА МЕТОДИ ПІДВИЩЕННЯ МОЖЛИВОСТЕЙ ВЕРИФІКАЦІЇ В КОМП'ЮТЕРИЗОВАНИХ СИСТЕМАХ ЧЕРЕЗ ПЕРЕТВОРЕННЯ МОДЕЛІ

## 3.1. Відображення для моделей інтерфейсу

У цьому розділі представлено важливий результат: концептуальне відображення між ComMA та GAL. Правила перетворення представлені для моделей інтерфейсу та компонентів із функціональними обмеженнями, а також для властивостей, які можна перевірити. Наводяться аргументи щодо того, як відображення підтримує семантику ComMA. Пропонується новий погляд на те, як можна зрозуміти та використовувати функціональні обмеження в налаштуваннях перевірки.

У цьому розділі показано, як складові елементи моделей інтерфейсу ComMA зіставляються з конструкціями GAL типів GAL. По-перше, пояснюється загальна структура метамоделі інтерфейсів ComMA. Після цього для кожного відповідного складового блоку пояснюються еквівалентні конструкції GAL, наводяться приклади та наводяться аргументи щодо того, як підтримується семантика ComMA.

Рисунок 3.1 представлено спрощену версію метамоделі ComMA. У цьому розділі підкреслюється загальна структура моделі. На найвищому рівні модель інтерфейсу може містити змінні разом з їхньою ініціалізацією та одним або кількома кінцевими автоматами. Кінцеві автомати містять набір станів, де один позначається як початковий стан і необов'язковий блок усіх станів, який можна використовувати для визначення переходів, що запускаються з кількох станів (етап попередньої обробки можна виконати на вхідної моделі, щоб абстрагуватися від цієї конструкції, додаючи переходи, що містяться в блоці, до їх правильного стану). Стани містять набір переходів, які запускаються (командою чи сигналом) або не запускаються. У разі параметризованих ініційованих переходів конкретні значення параметрів

наводяться в окремій моделі параметрів. Переходи містять одне або кілька речень, розділених конструкцією АБО. Речення можуть містити перелік дій. Існує п’ять основних типів дій: дії призначення для змінних, виклики подій, які посилаються на сповіщення, відповіді на тригери, шаблон будь-якого порядку, а також дія if-else, яка містить список дій для гілки if, і необов'язкова гілка else-branch, яка також пов'язана зі списком дій. Решта глави представить правила трансформації для елементів ComMA, дотримуючись порядку зверху вниз, заданого ієрархією метамоделі.

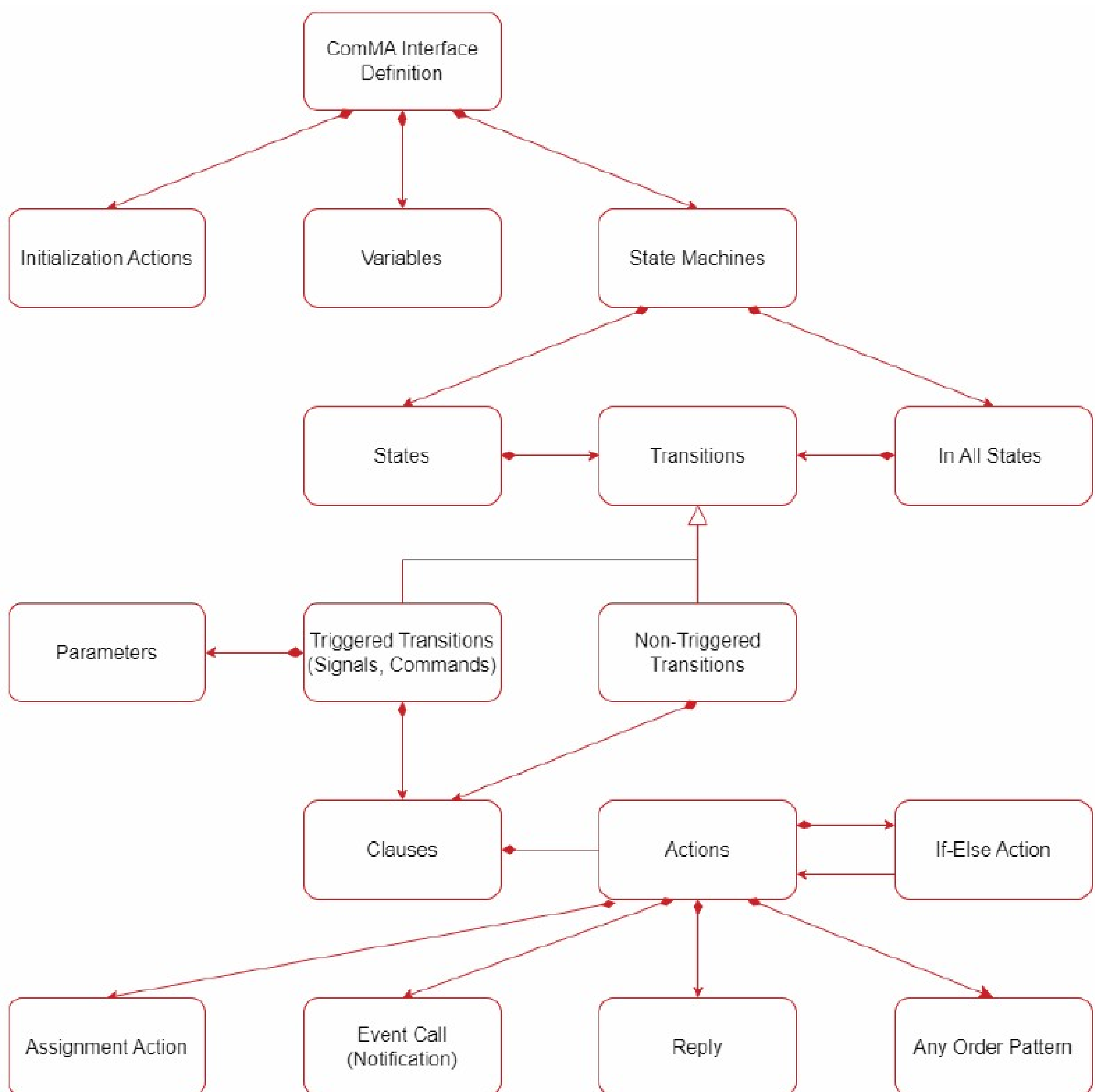


Рис. 3.1. Спрощена структура метамоделі інтерфейсу

*3.1.1. Попередньо визначені типи*

Константи та змінні ComMA можуть мати такі типи даних: int, bool, real, string і bulkdata. Також підтримуються такі складні типи: перерахування, вектори, записи та карти. З іншого боку, GAL забезпечує підтримку цілочисельних змінних і обмежених цілочисельних масивів. Таким чином, основні типи даних із ComMA зіставляються таким чином:

■ Int: усі цілочисельні константи та цілі вирази можна відобразити безпосередньо в GAL. Цілочисельну змінну з ComMA можна зіставити з цілочисельною змінною з GAL. Ініціалізація в ComMA дозволяє присвоювати початкові значення змінних виразам, які містять інші змінні. GAL не дозволяє ініціалізацію змінної залежати від інших змінних, щоб уникнути циклів залежностей. Однак це можна виправити, додавши перехід ініціалізації в GAL, який оновлює змінні, щоб мати правильні початкові значення.

■ Bool: усі булеві вирази від захисників переходу та дій if-else можна зіставляти з логічними виразами в GAL. Однак GAL не підтримує власну підтримку логічних змінних. Таким чином, логічну змінну з ComMA можна зіставити з цілочисельною змінною з GAL із двома значеннями: 1, що відповідає істині, і 0, що відповідає хибності. GAL надає оболонку , яка дозволяє булевим виразам бути піднятими до цілих виразів, оточуючи логічний вираз круглими дужками. Таким чином, призначення, які використовують логічні вирази для змінних у ComMA, можуть бути безпосередньо зіставлені з призначеннями, які використовують цілочисельні вирази в GAL.

■ Real: усі дійсні константи та дійсні вирази можна відобразити в цілих виразах, якщо всі дійсні значення, наявні в моделі, помножити на постійний коефіцієнт (наприклад, ступінь десяти), щоб видалити десяткову частину. У промислових моделях реальні значення зазвичай використовуються з точністю до трьох цифр, тому множення на 10 3 є адекватним. Крім того, механізм трансформації міг дозволити налаштувати

цей фактор. Справжню змінну з ComMA можна потім зіставити з цілочисельною змінною з GAL. Оскільки ComMA не дозволяє виразам містити змінні різного типу (наприклад, використання як дійсних, так і цілих змінних у виразі), дійсні змінні використовуються лише у виразах із дійсними числами. Таким чином, множення всіх дійсних констант на коефіцієнт зберігає однакову поведінку для захисників переходу та захисників структур if-else у GAL (Примітка: якби і дійсні, і цілі константи могли бути використані в одному виразі захисника, множення дійсного константи на коефіцієнт могли змінити результуюче значення істинності захисника у виході GAL).

■ String:: Єдиними операціями, які підтримуються для рядків, є призначення та порівняння між рядками. Тому кожному рядковому літералу в моделі можна призначити унікальний ідентифікатор. Рядкову змінну з ComMA можна зіставити з цілочисельною змінною з GAL, значення якої дорівнює унікальному ідентифікатору рядкового значення. Таким чином, порівняння унікальних ідентифікаторів у GAL матиме той самий результат, що й порівняння оригінальних рядків у ComMA.

■ Bulkdata: значення змінних bulkdata є неінтерпретованими масивами байтів, і вони зазвичай використовуються як параметри подій, і єдиною підтримуваною операцією є отримання довжини масиву. Змінну bulkdata можна зіставити з цілочисельною змінною з GAL, значення якої дорівнює розміру масиву bulkdata. Якщо розмір не вказано, можна використовувати значення за замовчуванням 0.

■ User-defined basic types: вони засновані на попередньо визначених типах даних (наприклад, тип Pixel на основі int). Вони зіставляються з GAL за допомогою правил для типів даних, на яких вони базуються.

### *3.1.2. Перерахування, вектори, записи та карти*

Перерахування фактично є фіксованим набором літералів перерахування, які зазвичай використовуються для параметрів подій. Ці

літерали перерахування можуть бути присвоєні невід'ємним цілим числам, і, якщо не задано явно, перший літерал приймає значення 0, а решта літералів збільшують це значення. Якщо змінна типу перерахування не ініціалізована, значення за замовчуванням задається першим літералом переліку. Тип перерахування можна зіставити з цілочисельною змінною GAL зі значеннями, рівними цілим значенням літералів перерахування з ComMA. Використання літералу перерахування в логічному виразі або призначення змінної в ComMA перетворюється на відповідне ціле число GAL, яке використовується. Рисунок 3.2 відображає приклад оголошення та призначення.

```
1  enum Mode {
2      //if not explicitly given, the first
       literal assumes value 0
3      updating = 1
4
5      //if not given, the value will increment
       the previous one, i.e. 2
6      working
7
8      //user may give a value greater than the
       increment of the previous value
9      sleeping = 4
10 }
11 variables
12 Mode m
13 Mode n
14
15 init
16 m := Mode::sleeping
```

```
1 //n_Mode and m_Mode accept values 1, 2 and 4
2
3 //n_Mode takes the default value
4 int n_mode = 1;
5
6 //m_Mode is initialized to sleeping
7 int m_Mode = 4;
```

Рис. 3.2. Відображення для перерахувань

Вектори будуються за допомогою базового типу та кількох вимірів. Для кожного розміру можна вказати додатковий розмір. Можна встановити повне значення вектора, а також динамічно додавати та видаляти елементи. Неможливо змінити значення однієї клітинки у векторному типі. GAL допускає лише масиви з фіксованим розміром, тому динамічне додавання та видалення не підтримується зіставленням. Однак це обмеження можна подолати в моделях, де динамічне додавання та видалення відсутні. Виконуючи статичний аналіз вхідної моделі, можна знайти верхню межу векторних розмірів (тобто перевіряючи кількість елементів кожного різного призначення векторній змінній і беручи максимум для довжини конкретного

розміру) . Для прикладу на рис 3.3, вектор myArray не має фіксованого розміру. Однак присвоєння в рядку 14 використовує чотири цілі числа. Якщо для цього вектора є кілька призначень і немає динамічних оновлень, максимальний розмір, який використовується у призначеннях, можна вибрати як верхню межу та, таким чином, як розмір масиву GAL.

```
1  //declare vector types
2  vector Matrix = int[2][3]
3  ...
4  //declare vector variables
5  Matrix m
6  int[] myArray
7  bool[3][3][3][3] largeSpace
8  ...
9  //assign a value to m
10 m := <Matrix>[<int[3]>[1, 2, 3],
11     <int[3]>[3, 4, 5]]
12
13 //assign a value to myArray
14 myArray := <int[]>[9, 10, 11, 12]
```

```
1  //a GAL array for each row
2  array [3] m_row_0_Matrix = (1, 2, 3);
3  array [3] m_row_1_Matrix = (3, 4, 5);
4
5  //a GAL array for a one-dimensional vector
6  array[4] myArray = (9, 10, 11, 12)
7
8  //option 1: a GAL variable for each cell
9  int largeSpace_0_0_0_0 = 0;
10 int largeSpace_0_0_0_1 = 0;
11 ...
12 int largeSpace_2_2_2_2 = 0;
13
14 //option 2: a GAL array with all values
15 array [81] largeSpace = (0, 0,...);
```

Рис. 3.3. Відображення для векторів

Вектори з одним виміром можуть бути зіставлені в масив GAL. Вектори з двома вимірами можуть бути зіставлені з серією масивів, де кожен векторний рядок відповідає одній змінній масиву GAL. Для трьох вимірів і більше є два варіанти. Для першого варіанту кожна клітинка у векторі може бути зіставлена з однією змінною GAL. Для другого варіанту можна створити єдиний масив GAL, який містить усі елементи (наприклад, у рядку 15 генерується масив largeSpace, який містить 3*3*3*3 = 81 елемент). До механізму перетворення можна додати функцію, яка відображає чотири індекси з ComMA на правильний елемент у відповідному масиві GAL. Порівнюючи параметри, окремі змінні легше читаються, але не дозволяють отримати доступ до індексу. На відміну від цього, другий варіант дозволяє індексувати, але менш інтуїтивно зрозуміло дізнатися, яка комірка масиву GAL відповідає конкретній комірці з вектора ComMA. Рисунок 3.3 відображає, як відображаються вектори з одним, двома та кількома вимірами. Примітка. Якщо до ComMA додано опцію оновлення значення окремої клітинки вектора, цю функцію можна відобразити в GAL шляхом

призначення змінної комірці масиву GAL (тобто для одно- або двовимірних векторів, і для другого варіанту відображення для багатовимірних векторів).

Записи — це набір інших змінних основних або складних типів даних. Тип запису також може розширювати інший уже визначений. Змінні можна присвоїти кожному елементу окремо або встановити значення всієї змінної запису. Відображення в GAL використовує спеціальні правила перетворення типу даних кожної змінної, що міститься в записі, а також схему іменування, яка вказує ім’я елемента та ім’я запису змінної.

```
1  record CurrentStatus {
2      string Name
3      int[3] Position
4      Mode m
5  }
6  ...
7  variables
8  CurrentStatus d
9
10 init
11 d := CurrentStatus{name = "Asset-1",
12     Position = <int[3]>[10, 11, 12],
13     m := Mode:working}
14 ...
15 d.Position = <int[3]>[10, 14, 15]
```

```
1  //unique string ID
2  int Name_d_CurrentStatus = 123
3
4  int [3] Position_d_CurrentStatus = (10, 11, 12)
5
6  //Mode:working is mapped to 2
7  int m_d_CurrentStatus = 2
8
9  Position_d_CurrentStatus = (10, 14, 15)
```

Рис. 3.4. Відображення для записів

Карти створюються шляхом визначення типів ключів і значень. Тип ключа має бути базовим типом або переліком. ComMA дозволяє динамічно додавати та видаляти пари ключ-значення з карт. Цю функцію не можна зіставити з GAL, де кількість змінних має бути фіксованою під час виконання. Однак у промислових випадках карти зазвичай використовуються для зберігання статичної інформації, головним чином для передачі як параметрів подій. Статичні карти можна зіставити з GAL, створивши цілочисельний масив GAL, що відповідає ключам. Також можна перевірити, чи міститься ключ у карті. Це можна відобразити в GAL, створивши цикл for, який повторює масив, щоб знайти ключ, або створивши логічний вираз, який містить список диз’юнкцій. Якщо типи значень карти також є перерахуваннями або базовими типами, можна використовувати другий цілочисельний масив GAL. Після цього елементи, розташовані за

однаковими індексами масивів, пов’язуються на карті. Якщо типи значень карт є більш складними структурами, такими як записи або вектори, масив GAL більше не можна використовувати для зберігання значень карти. Відповідні правила перетворення необхідно застосовувати для кожного значення карти окремо.

```
MapInt2Enum = map<int, mode>
...
variables
MapInt2Enum myMap
bool check

init
myMap := <MapInt2Enum>{10 -> mode:working, 11
     -> mode:sleeping, 12 -> mode:updating}
check := false
...
check := hasKey(myMap, "11")
```

```
//the keys
int [3] keys_myMap_MapInt2Enum = (10, 11, 12);
//the values
int [3] values_myMap_MapInt2Enum = (2, 4, 1);

check = 0;

//Checking for a key
typedef indexes = 0..2
for ($i: indexes) {
    if (keys_myMap_MapInt2Enum[$i] == 11) {
        check = 1;
    }
}
```

Рис. 3.5. Відображення для карт

### 3.2. Відображення для компонентних моделей

У цьому розділі показано, як елементи ComMA моделей компонентів із функціональними обмеженнями відображаються на серію типів GAL і один складений GAL. По-перше, для кожної моделі інтерфейсу створюється тип GAL за правилами з попереднього розділу. Після цього кожне функціональне обмеження на основі стану відображається на тип GAL. Загальні правила цього відображення подібні до тих, що представлені в попередньому розділі, оскільки функціональні обмеження на основі стану також використовують подібну семантику для свого визначення. Типи GAL моделей інтерфейсу доповнюються мітками залежно від подій із блоку подій використання функціонального обмеження на основі стану. Нарешті, створюється складений GAL, який забезпечує синхронізацію між типами GAL моделей інтерфейсу та типами GAL функціональних обмежень на основі стану. Цей складений GAL представляє поведінку компонента з урахуванням усіх інтерфейсів і функціональних обмежень на основі стану. Нарешті, у цьому представленні компонента можна перевірити різні властивості, які

неможливо було перевірити за допомогою поточних можливостей перевірки. Однак, перш ніж пояснювати відображення, представлено новий погляд на те, як функціональні обмеження використовуються в налаштуваннях перевірки.

### *3.2.1. Функціональні обмеження для верифікації*

Компоненти ComMA використовуються для визначення того, як різні інтерфейси ComMA на його портах можуть взаємодіяти один з одним. Моделі інтерфейсу ComMA вказують дозволені послідовності, що охоплюють типи подій, включені в їхній власний підпис, тоді як моделі компонентів ComMA вказують дозволені послідовності, що охоплюють події, які спостерігаються в контексті компонента (тобто всі події, які надсилаються до або з портів компонента). Конструкція, яка використовується для визначення цього порядку, називається функціональним обмеженням. Є два типи функціональних обмежень: функціональні обмеження на основі стану, які представляють аспект поведінки компонента за допомогою кінцевого автомата, і предикатні обмеження, які є виразами, які мають виконуватися для кожної спостережуваної події в контексті компонента. Рисунок 3.6 представляє спрощену метамодель визначень компонентів ComMA.

Кожне функціональне обмеження на основі стану визначає власний блок подій використання, який може містити будь-яку подію (тобто сигнал, команду, сповіщення або відповідь) з будь-якої моделі інтерфейсу, що міститься в компоненті. Події, які не містяться в блоці подій використання, не обмежуються функціональним обмеженням на основі стану, пов’язаним із цим блоком. Дозволеними тригерами переходу є команди та сигнали на наданих портах, а також відповіді та сповіщення на необхідних портах. І навпаки, дозволеними діями під час переходів є відповіді та сповіщення на наданих портах, а також команди та сигнали на необхідних портах.

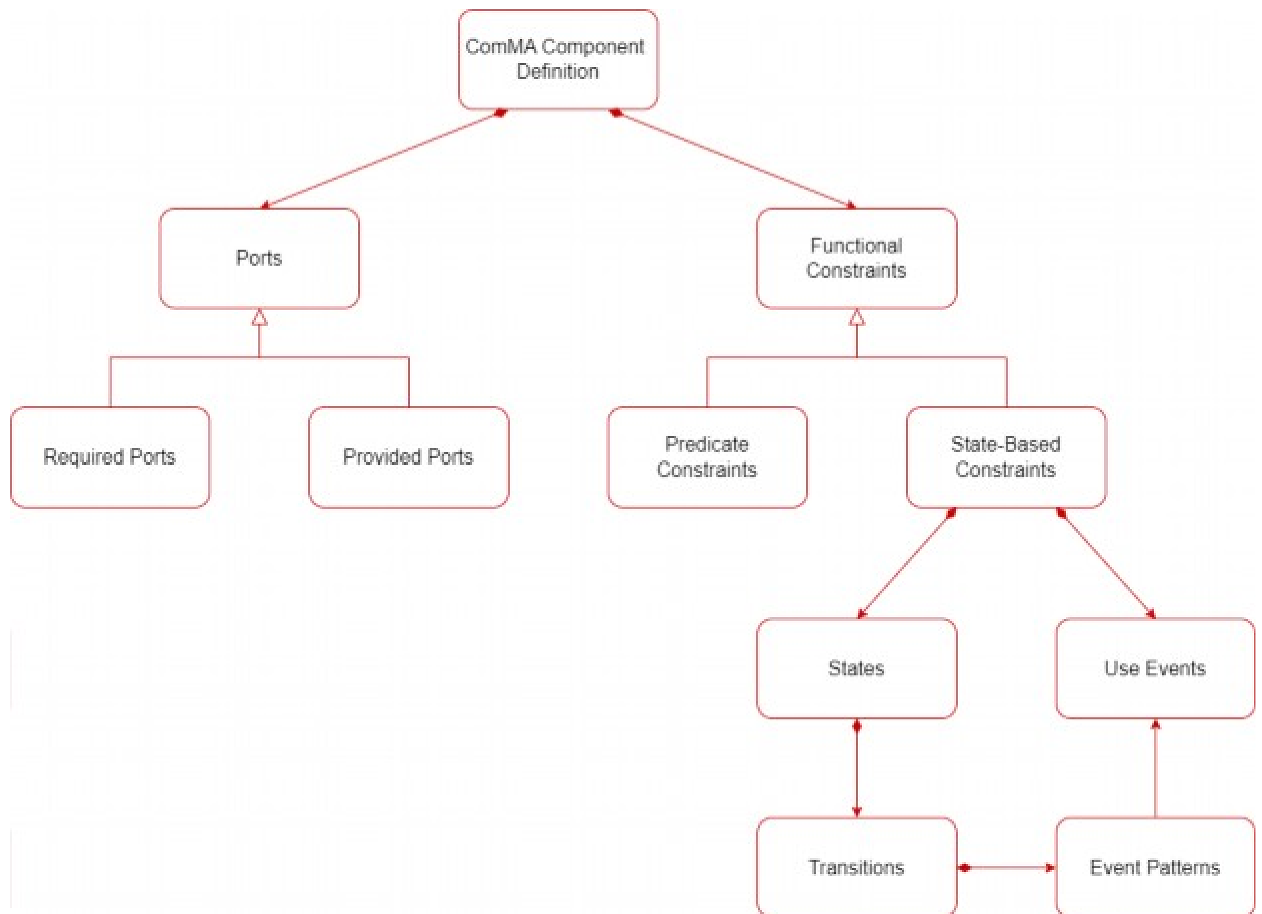


Рис. 3.6. Спрощена структура метамоделі компонента

Компонент із функціональними обмеженнями на основі стану та моделями інтерфейсу порту визначає набір трас подій, які йому відповідають. Трасування подій у цьому наборі має враховувати дві властивості: i) для кожного порту в компоненті, трасування, отримане шляхом збереження лише подій на цьому порту, задовольняє кінцевий автомат інтерфейсу та ii) для кожного функціонального обмеження на основі стану в компоненті , траса, отримана шляхом збереження лише подій із блоку подій використання цього конкретного обмеження, задовольняє функціональне обмеження на основі стану.

*3.2.2. Відображення для функціональних обмежень на основі стану*

Кожне функціональне обмеження на основі стану відображається на тип GAL. Правила цього відображення подібні до відображення

інтерфейсних моделей. Лістинг 3.1 представляє функціональне обмеження на основі стану, яке буде використано для довідки під час пояснення деталей відображення.

Лістинг 3.1. Приклад функціонального обмеження на основі стану

```
1  functional constraints
2  simple constraint {
3      use events
4      command camera::takePicture
5      camera::reply to command takePicture
6      command vacuum::VacuumOn
7
8      initial state OnlyState {
9          transition trigger: camera::takePicture
10             guard: vacuum in NoVacuum or vacuum in Evacuating
11                 camera::reply(CommandStatus::Interrupted) to command takePicture
12                 vacuum::VacuumOn
13             next state: OnlyState
14
15         transition trigger: camera::takePicture
16             guard: vacuum in Vacuum
17                 camera::reply(CommandStatus::Successful)
18             next state: OnlyState
19     }
20 }
```

Правила відображення представлені далі.

■ Створюється єдина змінна кінцевого автомата GAL, яка містить поточну конструкцію ComMA. Ця змінна використовується в щитах і тілах переходів GAL для імітації семантики запуску.

■ Генеруються параметри GAL, пов'язані з конструкціями функціонального обмеження на основі стану. Вони використовуються як можливі значення змінної кінцевого автомата GAL. Це такі конструкції:

□ Стан: Змінна кінцевого автомата GAL може дорівнювати параметру GAL, який відповідає стану кінцевого автомата. Для наступного значення змінної кінцевого автомата GAL робиться недетермінований вибір між параметрами GAL, що відповідають переходам цього стану.

□ Перехід: Змінна кінцевого автомата GAL може дорівнювати параметру GAL, який відповідає переходу стану. Охорона може включати умову щодо стану, в якому може перебувати один із інтерфейсів порту. Ця охорона абстрагується від типу GAL функціонального обмеження на основі стану. Наступним значенням змінної кінцевого автомата GAL буде параметр GAL, відповідний першій події, що міститься в переході.

□ Подія: Змінна кінцевого автомата GAL може дорівнювати параметру GAL, що відповідає події, що міститься в переході стану. Наступне значення змінної кінцевого автомата GAL залежить від того, чи є подія останньою в переході. Якщо це так, наступне значення змінної кінцевого автомата GAL є параметром GAL, що відповідає стану, що міститься в наступному блоці стану. Якщо подія не була останньою в серії, наступним значенням змінної кінцевого автомата GAL є параметр GAL, що відповідає наступній події.

Також можна мати нижчий рівень деталізації та абстрагуватися від станів. Якщо це буде зроблено, властивості щодо самих станів більше не можна буде сформулювати. Однак найважливішим аспектом є те, що всі події мають власний перехід GAL, щоб мати можливість синхронізувати їх у складеному GAL.

■ Існує чотири сценарії генерації переходів GAL, які представлені в таблиці 3.1. Для кожного сценарію захист переходу повинен перевірити, чи поточна змінна кінцевого автомата GAL вказує на параметр GAL, описаний у стовпці Guard , а тіло переходу оновлює змінну кінцевого автомата GAL, щоб вказати на параметр GAL, описаний у стовпці Оновлення.

Таблиця 3.1.

Сценарії генерації переходів GAL для функціональних обмежень на основі стану

| | Scenario | Guard | Update | Rationale |
|---|---|---|---|---|
| 1 | State → Transition | Current State | Chosen Transition | Choice between the transitions of a state |
| 2 | Transition → Event | Current Transition | First Event | Point to the first event in the block |
| 3 | Event → Event | Current Event | Next Event | The event is not the last one in the transition and there are still events left in the transition<br>Point to the next event in the block |
| 4 | Event → State | Last Event | Next State | Last event of the transition<br>Point to the next state |

■ Нарешті, усі переходи, пов’язані з подіями (тобто сценарії 2, 3 і 4 із таблиці), повинні мати репрезентативну мітку, яка вказує назву події. Ця

мітка використовуватиметься в синхронізаціях із складеного GAL, пов'язаного з компонентом ComMA.

Таким чином, використовуючи представлені правила, функціональне обмеження на основі стану з лістингу 3.1 відображається на тип GAL, присутній у лістингу 3.2.

Лістинг 3.2. Приклад типу GAL для функціонального обмеження на основі стану

```
1  //Parameters associated with execution states
2  $OnlyState = 0;
3  $OnlyState_trigger_takePicture0 = 1;
4  $OnlyState_trigger_takePicture0_reply = 2;
5  $OnlyState_trigger_takePicture0_VacuumOn = 3;
6  $OnlyState_trigger_takePicture1 = 4;
7  $OnlyState_trigger_takePicture1_reply = 5;
8
9  gal simpleConstraint {
10     //GAL state machine variable
11     int executionState = $OnlyState;
12
13     //Scenario 1
14     transition uniqueID_0 [executionState == $OnlyState] {
15         executionState = $OnlyState_trigger_takePicture0;
16     }
17
18     //Scenario 2
19     transition uniqueID_1 [executionState == $OnlyState_trigger_takePicture0] label "takePicture" {
20         executionState = $OnlyState_trigger_takePicture0_reply;
21     }
22
23     //Scenario 3
24     transition uniqueID_2 [executionState == $OnlyState_trigger_takePicture0_reply] label "reply" {
25         executionState = $OnlyState_trigger_takePicture0_VacuumOn;
26     }
27
28     //Scenario 4
29     transition uniqueID_3
30     [executionState == $OnlyState_trigger_takePicture0_VacuumOn] label "VacuumOn"{
31         executionState = $OnlyState;
32     }
33
34     //Scenario 1
35     transition uniqueID_4 [executionState == $OnlyState] {
36         executionState = $OnlyState_trigger_takePicture1;
37     }
38
39     //Scenario 2
40     transition uniqueID_5 [executionState == $OnlyState_trigger_takePicture1] label "takePicture" {
41         executionState = $OnlyState_trigger_takePicture1_reply;
42     }
43
44     //Scenario 4
45     transition uniqueID_6 [executionState == $OnlyState_trigger_takePicture1_reply] label "reply" {
46         executionState = $OnlyState;
47     }
48 }
```

### *3.2.3. Вкладені компоненти*

Аспект компонентів ComMA, який досі не згадувався в цьому розділі, це можливість додавання частин до компонентів. Частина — це екземпляр уже визначеного компонента, який може бути вкладений в інший компонент (тобто підкомпонент). Порти, пов'язані з компонентом, називаються

граничними портами, тоді як порти, пов’язані з частинами, називаються портами частини. Граничні порти можуть бути з'єднані з портами частин за допомогою з'єднань, які є абстракцією, що представляє канал для передачі подій. Підключення мають такі обмеження:

■ Частини підключення мають однакову модель інтерфейсу.

■ Існує щонайбільше одне підключення для будь-якої пари портів.

■ Наданий граничний порт можна підключити лише до одного наданого порту частини. Порт частини не може брати участь в інших підключеннях.

■ Необхідний порт частини можна підключити лише до одного наданого порту.

■ Порти частин у з’єднанні мають належати до різних частин.

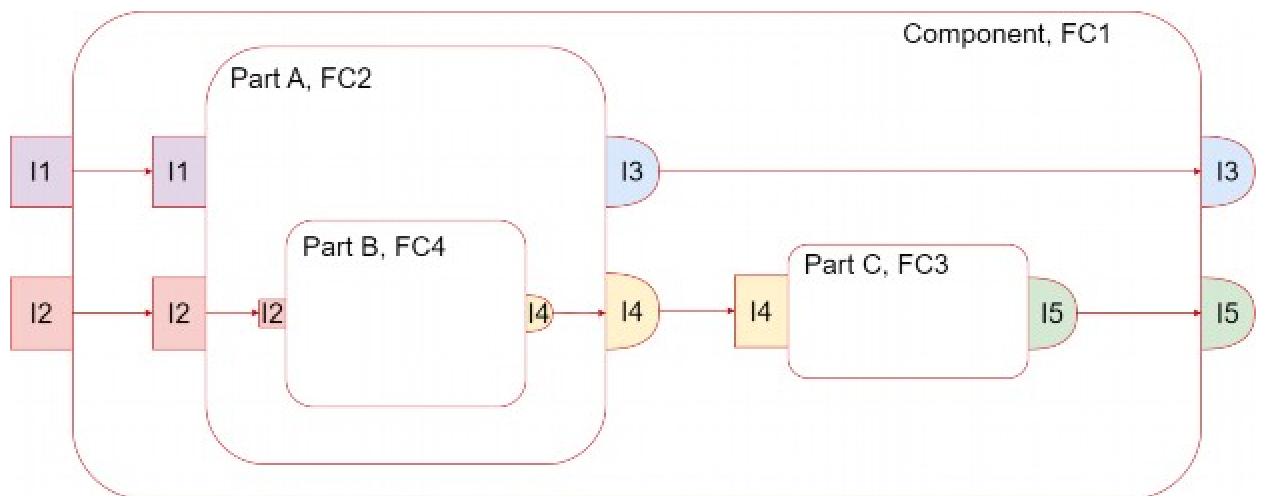

Рис. 3.7. Складений компонент із вкладеними частинами

Рисунок 3.7 представлено структуру прикладу складеної компонентної моделі, яка містить дві частини (тобто частину A та частину C), а одна з частин містить саму частину (тобто частина A містить частину B). Квадрати позначають передбачені порти, а півкола – необхідні порти. Існує п’ять типів інтерфейсів, що містяться в моделі, які також мають кольорове кодування (тобто I1, I2, I3, I4 та I5). Компонент і кожна частина мають власний набір функціональних обмежень (тобто FC1, FC2, FC3, FC4).

Проводиться паралель із моніторингом, щоб зробити висновок про деякі аспекти, які можна використовувати для справді вкладених компонентів. Єдиний монітор інтерфейсу призначається всім портам, підключеним або безпосередньо, або ланцюгом з'єднань. Таким чином, у цьому прикладі інтерфейси портів, які мають один і той же тип інтерфейсу, завжди знаходяться в тому самому стані. Що стосується моніторингу компонентів, порт інтерфейсу повинен відповідати функціональним обмеженням компонента, для якого він є портом. Оскільки порти в з'єднанні мають однаковий стан, це означає, що інтерфейс повинен відповідати об'єднанню функціональних обмежень (тобто тип інтерфейсу I1 відповідає до FC1 і FC2; тип інтерфейсу I2 відповідає FC1, FC2 і FC4; тип інтерфейсу I3 відповідає FC2 і FC1; тип інтерфейсу I4 відповідає FC4, FC2 і FC3; тип інтерфейсу I5 відповідає FC3 і FC1). Ці факти означають життєво важливий результат, який можна використовувати для відображення: частини можна повністю абстрагувати. Ця ідея не була детально досліджена і, як така, могла б представляти майбутню роботу. Проте тут наведено пропозицію щодо відображення. Для кожного типу інтерфейсу в моделі можна створити один тип GAL, що відповідає моделі інтерфейсу. Це тому, що всі підключені порти одного типу мають однаковий стан. Єдиним доповненням до відображення є те, що правильні набори функціональних обмежень повинні бути виведені для кожного типу інтерфейсу. Це відображається таким чином: для кожного типу GAL, що відповідає інтерфейсу, для кожного обмеження на основі стану, якому має відповідати модель інтерфейсу, виконуються синхронізації для відповідних подій із блоків подій використання з усіх на основі стану обмеження, що містяться в об'єднанні.

### *3.2.4. Синхронізаційні моделі інтерфейсу та функціональні обмеження на основі стану*

До цього часу були представлені відображення для моделей інтерфейсу та функціональних обмежень на основі стану. Тепер отримані типи GAL

потрібно синхронізувати за допомогою складеного GAL. Перший крок, який має відбутися, це додавання міток до відповідних переходів GAL від типів GAL моделей інтерфейсу, залежно від того, які події з блоку подій використання з функціональних обмежень на основі стану походять від конкретної моделі інтерфейсу.

Єдині розглянуті переходи GAL пов'язані з ініційованими переходами (тобто для команд і сигналів), діями викликів подій (тобто для сповіщень) і діями відповідей. Існує кілька правил додавання цих міток:

- Якщо блок подій використання містить команду або сигнал і цей тригер використовується кілька разів у моделі інтерфейсу, тоді кожен пов'язаний перехід GAL доповнюється тією самою міткою.
- Якщо блок подій використання містить відповідь на тригер, тоді кожен пов'язаний перехід GAL доповнюється тією самою міткою. Це включає відповіді з різних положень одного переходу та різних ініційованих переходів.
- Якщо блок подій використання містить сповіщення, кожен пов'язаний перехід GAL з усіх пунктів доповнюється тією самою міткою.
- Припустімо, що та сама подія використовується в окремих блоках подій використання з різних функціональних обмежень на основі стану. У цьому випадку мітки додаються лише один раз (тобто одна й та сама мітка може використовуватися для прогресу кількох типів GAL, пов'язаних із функціональними обмеженнями на основі стану).

Після того, як тип GAL моделі інтерфейсу було доповнено правильними мітками, можна створити складений GAL, пов'язаний із компонентом ComMA. Цей складений GAL містить екземпляр для кожного типу GAL, пов'язаного з моделями інтерфейсу та функціональними обмеженнями на основі стану.

Після цього генерується одна синхронізація для кожної події, присутньої в блоці подій використання. Синхронізація викликає мітки, пов'язані з цими подіями, із типу GAL інтерфейсу, з якого походить подія, а

також із типів GAL будь-яких функціональних обмежень на основі стану, які використовують цю подію. Лістинг 3.3 представляє загальну структуру виготовленого композиту GAL.

Лістинг 3.3. Структура композиту GAL, пов'язаного з компонентом ComMA

```
composite generalComposite {

    //GAL types associated with interfaces
    GALType_interface1 interface1;
    GALType_interface2 interface2;
    GALType_interface3 interface3;
    ...
    //GAL types associated with functional constraints
    GALType_FC1 fc1;
    GALType_FC2 fc2;
    GALType_FC3 fc3;
    ...

    //One synchronization per event
    synchronization eventA_interface1 {
        interface1."eventA";
        //Any constraints that use eventA
        fc1."eventA";
        fc2."eventA";
        ...
    }
    ...
}
```

### 3.3. Перевірка компонентів із функціональними обмеженнями

Нова методологія дозволяє перевіряти властивості моделей компонентів із функціональними обмеженнями. У цьому розділі представлено приклад застосування та результати нового рішення. Оцінка базувалася на стандартному прикладі компонентної моделі, з якою ComMA постачається. Цей приклад представляє реалістичне використання моделей компонентів, які раніше використовувалися для демонстрації функцій компонентного монітора.

Компонентна модель Imaging ComMA і відповідний композит GAL наведено в лістингу 3.4. Він спрямований на фіксацію ситуацій, які спостерігаються на практиці. Ці машини працюють за допомогою надпровідних магнітів і охолоджуються при надзвичайно низьких температурах, які можна підтримувати лише у вакуумі. Таким чином, можна ідентифікувати п'ять відмінних частин, кожна з яких має окрему модель інтерфейсу ComMA. Блок керування зображеннями надсилає інструкції

системі отримання зображень. Один пристрій контролює температуру, а інший - стан вакууму. Нарешті, для створення вакууму використовується насос.

Лістинг 3.4. Фрагмент Imaging компоненти

```
import "ITemperature/ITemperature.interface"
import "IVacuum/IVacuum.interface"
import "IImage/IImage.interface"
import "IPump/IPump.interface"
import "IAcq/IAcq.interface"

component Supervision

provided port ITemperature iTempPort
provided port IVacuum iVacPort
provided port IImage iImagPort
required port IPump iPumpPort
required port IAcq iAcqPort

functional constraints

StartImagingConstraint {

/*
 * Imaging can start only if vacuum is present and temperature level is reached
 */
 use events
  command iImagPort::StartImaging
  iImagPort::reply to command StartImaging

  initial state Status {
    command iImagPort::StartImaging where iVacPort in Vacuum and iTempPort in TemperatureSet
    iImagPort::reply(1) to command StartImaging
    next state: Status

    command iImagPort::StartImaging where not iVacPort in Vacuum or not iTempPort in TemperatureSet
    iImagPort::reply(0) to command StartImaging
    next state: Status
  }
}

ImagingConstraint

/*
 * If an image is being acquired, vacuum is always present and temperature is set.
 * Compare to the previous constraint: it does not prevent reset temperature or vacuum off after
     imaging has started
 */
always
not iImagPort in Imaging or (iVacPort in Vacuum and iTempPort in TemperatureSet)


TemperatureConstraint {

/*
 * Setting the temperature level is allowed only if vacuum is reached
 */
 use events
  command iTempPort::SetTemperature

  initial state CheckTempCondition {
    command iTempPort::SetTemperature where iVacPort in Vacuum
    next state: CheckTempCondition
  }
```

Таким чином, компонент Imaging містить три надані порти (тобто, за допомогою яких клієнти надсилають команди до компонента): ITemperature (тобто вхід ComMA та вихід GAL, які містяться в лістинг 3.5), IVacuum (лістинг 3.6) та IImage (лістнг 3.7) і два необхідні порти (тобто, які компонент

використовує від імені клієнтів): IPump (лістинг 3.8) і IAcq (лістинг 3.9). На рисунку 3.8 представлена схема компонента.

Лістинг 3.5. Інтерфейс ITemperature

```
import "ITemperature.signature"

interface ITemperature version "1.0"

machine TemperatureMachine {

  initial state BasicTemperature {

    transition trigger: SetTemperature(int t)
      do:
      reply(*)
      next state: SettingTemperature
  }

  state SettingTemperature {

    transition
      do: TemperatureOK
      next state: TemperatureSet
  }

  state TemperatureSet {

    transition trigger: ResetTemperature
      do:
      reply(*)
      next state: BasicTemperature
  }
}
```

Лістинг 3.6. Інтерфейс IVacuum

```
import "IVacuum.signature"

interface IVacuum version "1.0"

machine VacuumMachine {

  initial state NoVacuum {

    transition trigger: VacuumOn
      do:
      reply(*)
      next state: Evacuating
  }

  state Evacuating {

    transition
      do: VacuumOK
      next state: Vacuum
  }

  state Vacuum {

    transition trigger: VacuumOff
      do:
      reply(*)
      next state: NoVacuum
  }
}
```

Лістинг 3.7. Інтерфейс IImage

```
import "IImage.signature"

interface IImage version "1.0"

machine ImagingMachine {

  initial state Initial {

    transition trigger: Prepare(int setting)
        do:
        reply(1)
        next state: Prepared
      OR
        do:
        reply(0)
        next state: Initial

    transition
      do: Error
      next state: Initial
  }

  state Prepared {

    transition trigger: StartImaging
        do:
        reply(1)
        next state: Imaging
      OR
        do:
        reply(0)
        next state: Initial

    transition trigger: Unprepare
      do:
      reply(*)
      next state: Initial

    transition
      do: Error
      next state: Initial
  }

  state Imaging {
    transition trigger: StopImaging
      next state: Prepared

    transition
      do: Error
      next state: Initial
  }
}
```

Лістинг 3.8. Інтерфейс IPump

```
import "IPump.signature"

interface IPump version "1.0"

machine PumpMachine {

  initial state PumpOff {

    transition trigger: On
        do:
        reply(*)
        next state: PumpOn
  }

  state PumpOn {

    transition trigger: Off
        do:
        reply(*)
        next state: PumpOff
  }
}
```

Лістинг 3.9. Інтерфейс IAcq

```
1  import "IAcq.signature"
2
3  interface IAcq version "1.0"
4
5  machine ImagingMachine {
6
7    initial state AcqInitial {
8
9      transition trigger: PrepareAcquisition(int
       setting)
10         do:
11         reply(1)
12         next state: AcqPrepared
13       OR
14         do:
15         reply(0)
16         next state: AcqInitial
17
18
19     transition
20       do: Error
21       next state: AcqInitial
22   }
23
24   state AcqPrepared {
25
26     transition trigger: StartAcquisition
27         do:
28         reply(1)
29         next state: AcqImaging
30       OR
31         do:
32         reply(0)
33         next state: AcqInitial
34
35     transition trigger: UnprepareAcquisition
36       do:
37       reply(*)
38       next state: AcqInitial
39
40     transition
41       do: Error
42       next state: AcqInitial
43   }
44
45   state AcqImaging {
46
47     transition trigger: StopAcquisition
48       do:
49       reply(*)
50       next state: AcqPrepared
51
52     transition
53       do: Error
54       next state: AcqInitial
55   }
56 }
```

Компонент зображення також містить різні обмеження (тобто функціональні обмеження на основі стану):

■ Обмеження значення Prepare: обмеження даних визначає, що параметр, переданий для команди підготовки моделі зображення, використовується в команді підготовки отримання в моделі отримання зображення (тобто зіставляється з GAL).

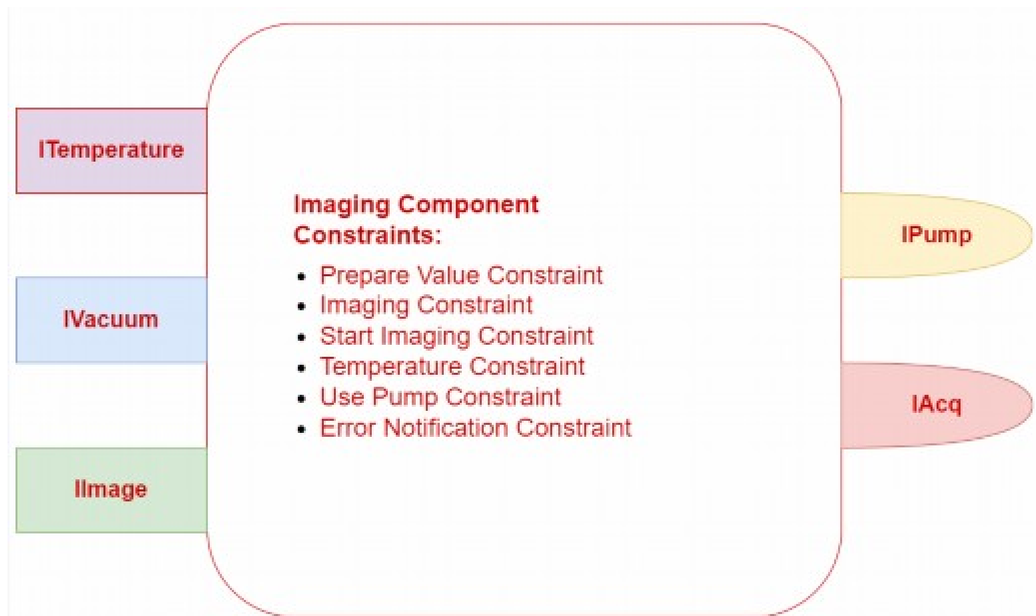


Рис. 3.8. Візуалізація компоненти верифікації

■ Обмеження зображення: предикатне обмеження стверджує, що якщо зображення отримується, вакуум завжди присутній і температура встановлена.

■ Обмеження запуску зображення: функціональне обмеження на основі стану стверджує, що створення зображення може розпочатися лише за наявності вакууму та досягнення правильного рівня температури.

■ Температурне обмеження: функціональне обмеження, засноване на стані, стверджує, що встановлення рівня температури дозволено лише за умови досягнення вакууму.

■ Обмеження використання насоса: функціональне обмеження на основі стану вказує, що для евакуації використовується насос, який можна зупинити лише після досягнення вакууму.

■ Обмеження сповіщень про помилки: функціональне обмеження на основі стану стверджує, що помилка зображення виникає тоді і лише тоді, коли виникає помилка в отриманні зображення.

Етапи, виконані в оцінці, були наступними.

1. Типи GAL для ITemperature, IVacuum, IImage, IPump і IAcq були створені за допомогою реалізації.

2. Згенеровані властивості (тобто стани доступності, взаємоблокування, блокування в реальному часі, стани приймача та стани вибору) перевірялися для кожного типу GAL. Властивості, які відповідають кожній моделі інтерфейсу, також включені в додатки.

3. Зведений тип GAL, що містить змінні та переходи від усіх чотирьох функціональних обмежень на основі стану, було реалізовано вручну шляхом об'єднання елементів автоматично згенерованих типів GAL.

4. Чотири типи GAL, що відповідають чотирьом функціональним обмеженням на основі стану, було реалізовано вручну.

5. Переходи GAL, що відповідають подіям із блоків використання чотирьох функціональних обмежень на основі стану, були вручну доповнені мітками (тобто як для незалежних типів GAL, так і для зведеного типу GAL)

6. Дві складені моделі GAL були реалізовані вручну. Такий, який реалізує стратегію, де кожен інтерфейс має свій власний тип, а також зведену стратегію. До обох композитів додаються синхронізації, що відповідають усім подіям.

7. Властивості були перевірені для обох композитів GAL. Властивості включали ті, що стосувалися моделей інтерфейсу, згаданих вище, і ті, що стосувалися функціональних обмежень на основі стану.

Було прийнято рішення порівняти результати перевірки для двох стратегій відображення (тобто стандартного відображення та зведеного відображення). Причиною цього є той факт, що ITS-інструменти були обрані як сервер верифікації через семантичну схожість, яку вони мають із ComMA. Перевірка компонентів із функціональними обмеженнями була важливою метою дослідження, а композити GAL, які можуть містити екземпляри типів GAL, є конструкцією, яка сильно нагадує стиль моделювання на основі компонентів, який також використовується в ComMA. У стандартному відображенні використовується більше типів GAL, і, таким чином, очікується, що цей більш детальний поділ дозволить ITS-інструментам краще використовувати базову ієрархію, присутню в складеному GAL. Таким

чином, усі властивості (тобто, що стосуються як типів моделей інтерфейсу GAL, так і функціональних обмежень на основі стану) було перевірено на обох композитах GAL, пов'язаних із двома різними стратегіями відображення. Таблиця 3.2 представлені результати її охоплення, тоді як у таблиці 3.3 представлені результати its-ctl.

Таблиця 3.2.

Результати його охоплення на компонентній моделі Imaging

| Strategy | State Space | Fin. SDD | Peak SDD | Fin. DDD | Peak DDD | Time |
|---|---|---|---|---|---|---|
| Standard | 67336272 | 38 | 443 | 34 | 433 | 0.018 |
| Flattened | 1.22265e+08 | 27 | 443 | 316 | 8804 | 1.087 |

Таблиця 3.3.

Результати its-ctl на моделі компонента Imaging

| Strategy | State Space | Fin. SDD | Peak SDD | Fin. DDD | Peak DDD | Time |
|---|---|---|---|---|---|---|
| Standard | 1.44196e+08 | 39 | 359 | 35 | 421 | 0.038 |
| Flattened | 1.44196e+08 | 27 | 429 | 316 | 9573 | 0.078 |

Як показують результати, дійсно, кодування стандартного відображення перевіряється швидше. Більше того, ті самі спостереження, які були зроблені раніше, все ще справедливі за деякими винятками: its-ctl вимагає більше часу (тобто, але не для згладженої моделі), і перевіряє більший простір станів, ніж its-reach. Більше того, цікаво, що кінцевий SDD для стандартної моделі більший, ніж для сплющеної моделі. Навпаки, кінцевий розмір DDD зведеної моделі значно більший, ніж у стандартної моделі.

## Висновки до розділу

Третій розділ присвячено розробці моделей і методів, спрямованих на підвищення можливостей верифікації комп'ютеризованих систем через перетворення моделі, що є важливим аспектом забезпечення надійності

складних систем. Розглянуто техніки відображення, які дозволяють адаптувати інтерфейси моделей для зручності верифікації. Було описано використання попередньо визначених типів даних, а також типів, таких як перерахування, вектори, записи та карти, що спрощує процес створення чітких і формалізованих моделей інтерфейсу.

Показано підходи до побудови компонентних моделей із урахуванням функціональних обмежень для верифікації. Це включає відображення функціональних обмежень на основі стану, роботу з вкладеними компонентами, а також використання синхронізаційних моделей для визначення функціональних обмежень інтерфейсів на рівні станів. Ці методи є важливими для забезпечення точності перевірки поведінкових аспектів компонентів системи.

Описано процес перевірки компонентів із врахуванням функціональних обмежень. Підходи, розглянуті в цьому розділі, спрямовані на покращення можливостей верифікації шляхом оптимізації моделей, що дозволяє ефективніше виявляти потенційні проблеми та аномалії на етапі проєктування.

Цей розділ сформував підґрунтя для покращення процесів верифікації через перетворення моделей, що є важливим кроком у підвищенні якості та надійності комп’ютеризованих систем.

# ВИСНОВКИ

У магістерській роботі досліджено процеси моделювання, аналізу та верифікації в комп'ютеризованих системах, зокрема у контексті предметно-спеціальних моделей.

У першому розділі проведено аналіз предметної області верифікації з акцентом на модельно-орієнтовану інженерію, де значну увагу приділено предметно-орієнтованим мовам, метамоделям і трансформаціям. Було визначено важливість платформи Eclipse як інструмента для моделювання та автоматизації процесів верифікації, а також досліджено інструменти перевірки, що підвищують точність і ефективність процесу.

Другий розділ присвячено методам аналізу та верифікації компонентів моделей, зокрема через вивчення синтаксису та семантики інтерфейсів, що дозволяє формалізувати процеси перевірки. Розглянуто можливості платформ для створення артефактів верифікації, а також їхні обмеження, що вказує на потенціал подальшого розвитку методології. Було досліджено методи інтеграції інструментів верифікації, використання мережі Петрі та особливості архітектури ITS-інструментів для перевірки символьних моделей. У третьому розділі запропоновано підходи до підвищення можливостей верифікації через перетворення моделей. Було розглянуто техніки відображення інтерфейсів та компонентних моделей з використанням типів даних, перерахувань і синхронізаційних моделей. Особлива увага приділена функціональним обмеженням та відображенням компонентів на основі станів, що дозволяє поліпшити точність верифікації.

У результаті виконаного дослідження розроблено рекомендації для покращення процесів моделювання та верифікації у комп'ютеризованих системах, що дозволяє підвищити їхню надійність та відповідність заданим вимогам. Запропоновані моделі та методи можуть бути використані для подальшого вдосконалення технологій верифікації та інтегровані в процеси розробки складних програмних рішень.

## ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Y. Thierry Mieg, “From Symbolic Verification To Domain Specific Languages,” Habilitation `a diriger des recherches, Sorbonne Universit´e , UPMC ; Laboratoire d’informatique de Paris 6 [LIP6], Dec. 2016. [Online]. Available: https://hal.science/tel-02104341
2. B.-J. Hilbrands, “Verification of inter-dependent interfaces in component-based architectures,” 2022.
3. G. Booch, J. Rumbaugh, and I. Jacobson, “Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series),” J. Database Manag., vol. 10, 01 1999.
4. G. T. Heineman and W. T. Councill, Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc., 2001.
5. B. Akesson, “Model-based specification, verification, and adaptation of software interfaces - the Dynamics approach,” ESI-TNO. [Online]. Available: https://esi.nl/news/blogs/the-dynamics-approach
6. R. van Beusekom, B. de Jonge, P. F. Hoogendijk, and J. Nieuwenhuizen, “Dezyne: Paving the way to practical formal software engineering,” in F-IDE@NFM, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:236909400
7. D. C. Schmidt et al., “Model-driven engineering,” Computer-IEEE Computer Society, vol. 39, no. 2, p. 25, 2006.
8. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, EMF: Eclipse Modeling Framework. Pearson Education, 2008.
9. I. Kurtev and J. Hooman, “Runtime verification of Compound Components with ComMA,” Lecture Notes in Computer Science, p. 382–402, 2022.
10. Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). Model Checking. MIT Press.
11. Baier, C., & Katoen, J. P. (2008). Principles of Model Checking. MIT Press.

12. Holzmann, G. J. (2003). The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley.
13. Broy, M., Jonsson, B., Katoen, J. P., Leucker, M., & Pretschner, A. (Eds.). (2005). Model-Based Testing of Reactive Systems. Springer.
14. Clarke, E. M., & Kroening, D. (2004). Model Checking with Binary Decision Diagrams. Springer.
15. Huth, M., & Ryan, M. D. (2004). Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press.
16. Fitzgerald, J., Larsen, P. G., & Verhoef, M. (2014). Collaborative Design for Embedded Systems: Co-modelling and Co-simulation. Springer.
17. Alur, R. (2015). Principles of Cyber-Physical Systems. MIT Press.
18. Anderson, R. J. (2008). Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley.
19. D'Souza, D., & Wadhawan, S. (2017). Formal Verification of Concurrent Programs. Springer.
20. Jhala, R., & Majumdar, R. (2009). Software Model Checking. ACM Computing Surveys, 41(4), 1-54.
21. Clarke, E. M., & Wing, J. M. (1996). Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, 28(4), 626-643.
22. McMillan, K. L. (1993). Symbolic Model Checking: An Approach to the State Explosion Problem. Springer.
23. Bengtsson, J., & Yi, W. (2003). Timed Automata: Semantics, Algorithms and Tools. Springer.
24. Gordon, M. J. C., & Melham, T. F. (Eds.). (1993). Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press.
25. O’Hearn, P. W. (2018). Continuous Reasoning: Scaling the Impact of Formal Methods. Communications of the ACM, 61(6), 56-65.
26. Milner, R. (1989). Communication and Concurrency. Prentice Hall.

27. Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems (TOPLAS), 8(2), 244-263.
28. Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. Communications of the ACM, 12(10), 576-580.
29. Harel, D., & Pnueli, A. (1985). On the Development of Reactive Systems. Springer.
30. Luckham, D. C. (2001). The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley.
31. Visser, W. (2004). Model Checking Programs. Automated Software Engineering, 10(2), 203-232.
32. Barrett, C., & Tinelli, C. (2018). Satisfiability Modulo Theories. Springer.
33. Jackson, D. (2006). Software Abstractions: Logic, Language, and Analysis. MIT Press.
34. Rocha, C., & Rocha, H. (2019). Formal Methods and Hybrid Systems Verification. Springer.
35. Vardi, M. Y. (1985). A Temporal Fixpoint Calculus. Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
36. Burch, J. R., Clarke, E. M., & McMillan, K. L. (1992). Symbolic Model Checking: 10^20 States and Beyond. Information and Computation, 98(2), 142-170.
37. Lutz, R. R., & Mikulski, I. C. (1995). Automated Requirements Analysis for a Reusable Spacecraft Guidance, Navigation, and Control System. IEEE Transactions on Software Engineering, 21(3), 209-223.
38. Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing. John Wiley & Sons.
39. Robinson, H., & Voorhees, J. (2008). Foundations of Security Analysis and Design. Springer.

40. Rushby, J. (1994). Formal Verification of Safety-Critical Software. ACM Computing Surveys, 28(4es), 731-736.
41. Wing, J. M. (1990). A Specifier's Introduction to Formal Methods. Computer, 23(9), 8-22.
42. Leveson, N. G. (1995). Safeware: System Safety and Computers. Addison-Wesley
43. J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, "Symbolic model checking for sequential circuit verification," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 4, pp. 401–424, 1994.
44. N. Roos, "ComMA interfaces open the door to reliable high-tech systems," Sep 2020. [Online]. Available: https://bits-chips.nl/artikel/comma-interfaces-open-the-door-to-reliable-high-tech-systems/
45. V. Pech, A. Shatalin, and M. V¨olter, "JetBrains MPS as a tool for extending Java," 09 2013, pp. 165–168.
46. J. Vinju, M. Hills, P. Klint, A. van der Ploeg, A. Izmaylova, and S. T, "The Rascal meta-programming language - a lab for software analysis, transformation, generation visualization," 11 2011.
47. OMG, OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, Object Management Group Publication, Rev. 2.4.1, Jun. 2013. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1
48. L. M. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Tr`eves, "The Petri net markup language and ISO/IEC 15909-2," in CPN Workshop, 2009.
49. Y. Thierry-Mieg, "Symbolic model-checking using its-tools," in Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. Springer, 2015, pp. 231–237.

50. T. van Dijk, J. Meijer, and J. van de Pol, “Multi-core on-the-fly saturation,” Tools and Algorithms for the Construction and Analysis of Systems, p. 58–75, 2019.
51. S. Brand, T. B¨ack, and A. Laarman, “A decision diagram operation for reachability,” 12 2022.
52. Lamport, L. (2002). Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley.
53. Cousot, P., & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.
54. T. van Dijk and J. Pol, “Multi-core symbolic bisimulation minimisation,” International Journal on Software Tools for Technology Transfer, vol. 20, pp. 1–21, 04 2018.
55. A. Laarman and A. Wijs, “Partial-order reduction for multi-core ltl model checking,” in Hardware and Software: Verification and Testing, E. Yahav, Ed. Cham: Springer International Publishing, 2014, pp. 267–283.