项目用户手册

陈锦赐

2020 年 6 月

目　录

Part 1. 项目适用场景与基本功能

1.1 适用场景

上下文感知程序 (context-aware application) 已经在我们的生活中得到越来越广泛的应用，这类程序可以利用各种从环境中收集的信息，如 GPS 数据、环境温度、氧气浓度等来帮助程序提供更智能的服务，这部分程序感兴趣的环境信息也被称为程序的上下文 (context)。上下信息通常是通过程序关联的各类传感器进行感知，但是由于传感器感知过程容易受到噪声等不可控因素的影响，最终感知到的上下文信息很有可能不准确、不完整甚至相互冲突，造成上下文不一致 (inconsistency) 问题，不一致的上下文信息会影响程序的正常功能，甚至会造成一些不可预期的严重后果。因此，如何对于上下文信息的一致性进一个预先的检测判断至关重要。

例如，一个上下文感知程序为监测城市 X 和 Y 的人员流动的程序 P，如图1.1所示，它所关心的环境上下文为城市 X 和 Y 的人员流动情况，有人员离开或进入某个城市会使得上下文信息发生改变，在时刻 3，程序 P 得到到“Sanji enters city Y”的动作发生的信息后，程序 P 可能会发生错误，因为 Sanji 早在 0 时刻已经进入了城市 X，并且在时刻 3 之前程序 P 并未得到“Sanji leaves city X”的动作信息，程序 P 会认为 Sanji 在时刻 3 既城市 X 又在城市 Y，即它认为 Sanji 在同一时刻出现在了不同的物理空间，这显然是不可能发生的，由此影响了程序 P 的正常功能。我们的项目适用于对这样的上下文不一致问题做出预先的检测和报告，为后续上下文信息的修复或直接丢弃提供指导作用。

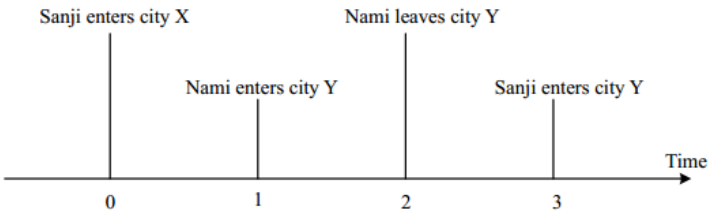

图 1.1: 城市 X 和 Y 的人员流动情况

1.2 基本功能

由于直接判定上下文信息的一致性十分困难，所以我们项目的解决方式是通过根据上下文感知程序的应用场景预定义一系列不可违反的一致性规则

(consistency constraint)，并根据这些规则来帮助检测上下文信息的一致性，一旦发现任何违反预定义规则的情况就认为此时的上下文信息存在不一致。

还是以图1.1为例，我们定义一致性规则为“同一个人不能同时出现在不同的地方”，因此，在时刻 3，“Sanji enters city Y”的动作发生后，由于一致性规则的存在，这个不一致问题会被系统报告，从而避免程序 P 发生不可预测的错误。

1.3 项目手册组织结构

本项目手册的组织结构如下：

- Part 1. 介绍项目的适用场景与基本功能，简述本项目的主要工作；
- Part 2. 介绍项目的环境需求与安装部署，简述如何配置项目运行的软硬件依赖，以及如何获取和安装部署本项目，此部分是项目运行的基础，需要详细阅读；
- Part 3. 介绍项目的总体架构与主要组件构成，简述项目系统的设计与实现，主要展示系统的设计思路和实现细节，此部分对于非开发者用户来说，仅需了解项目系统中所涉及的技术名称，以配置系统运行，无需关注设计细节；对开发者用户来说，需要参照项目系统架构与各大组件设计细节，结合相应源码理解设计的思路，以便根据自身应用场景需求，定制项目系统；
- Part 4. 介绍项目的运行与 Demo 分析，简介项目系统如何配置与运行，结合 Demo 展示配置与运行的细节，此部分对于非开发者来说，仅需了解项目的输入、输出，以及根据 Demo 介绍运行 Demo 即可，无需关注需要代码定制的部分；对于开发者用户来说，需要以现有 Demo 为参照，根据标注的需要根据自身应用场景定制的部分修改项目系统源码，适配自身应用场景。

Part 2. 项目环境需求与安装部署

2.1 硬件环境需求

2.1.1 处理器

项目的正常运行对处理器的型号和性能没有特殊要求，使用自己现有的处理器即可，但使用较高性能的处理器，可以获得更好的软件性能，要进一步了解自己的处理器性能可根据处理器型号参看处理器天梯图。

2.1.2 主存

项目运行要求内存至少 8GB。

2.1.3 显卡 (可选)

项目运行 GAIN[4] 技术需要一块支持CUDA 编程的 NVIDIA 系列显卡，这是一种基于 GPU 的加速技术，后续在 Part 3. 中会给出更多说明，若不需要可以跳过本小节。

关于支持 CUDA 编程的部分显卡信息如图2.1所示,更多信息可以从NVIDIA开发者官网进一步查询。

GeForce and TITAN Products

GPU	Compute Capability
NVIDIA TITAN RTX	7.5
Geforce RTX 2080 Ti	7.5
Geforce RTX 2080	7.5

GeForce Notebook Products

GPU	Compute Capability
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5

图 2.1: 支持 CUDA 的部分显卡信息

2.2 软件环境需求

2.2.1 操作系统

项目可运行在 Linux 或 Windows 操作系统之上，Linux 推荐使用Ubuntu 16.04 LTS版本，Windows 推荐使用Windows 10版本。

2.2.2 Java 运行环境

项目运行要求 Java 运行环境为 Oracle Java 8 或以上的版本。

2.2.3 显卡驱动与 CUDA Tool Kit (可选)

若跳过 2.1.3 小节，则可跳过本小节。

显卡驱动与 CUDA Tool Kit 的安装与具体的操作系统和使用的显卡型号相关，这里以 Ubuntu 16.04 LTS 系统和 NVIDIA GeForce GTX 980 显卡为例。

1. 显卡驱动

   需要根据操作系统版本及显卡型号从NVIDIA 驱动官网下载相应的版本进行下载安装。

   (a) 如图2.2所示，选择正确的版本进行下载。

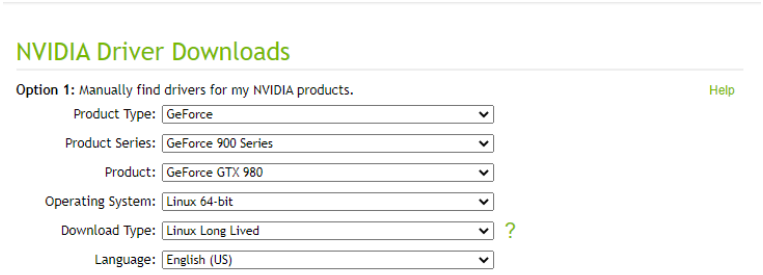

图 2.2: 选择驱动版本

   (b) 安装参考驱动安装教程。

2. CUDA Tool Kit

   需要根据操作系统版本等信息从NVIDIA 开发者官网下载相应版本，并根据提示进行相应安装。

   (a) 如图2.3所示，选择正确的版本进行下载。

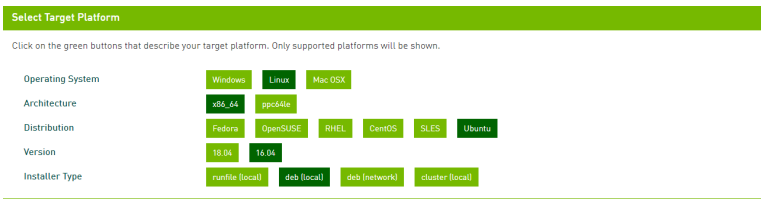

图 2.3: 选择 CUDA Tool Kit 版本

   (b) 如图2.4所示，进行安装。

```
$ sudo dpkg -i cuda-repo-ubuntu1604-10-2-local-10.2.89-440.33.01_1.0-1_amd64.deb
$ sudo apt-key add /var/cuda-repo-10-2-local-10.2.89-440.33.01/7fa2af80.pub
$ sudo apt-get update
$ sudo apt-get -y install cuda
```

图 2.4: CUDA Tool Kit 安装命令

2.2.4 第三方库

以下这些 Java 第三方包在开发时均可通过Maven进行下载管理，无需单独下载安装：

1. dom4j-1.6.1.jar : 用于 XML 文件的处理和解析；
2. jcuda-0.8.0.jar : 用于桥接 Java 代码与 CUDA 代码；
3. jcudaUtils-0.0.4.jar: 对 jcuda 的进一步封装，提供更为简易的 API。

需要注意的是，无论是否选择跳过 2.1.3 及 2.2.3 小节，上述有关 CUDA 的第三方包 (2 和 3) 都是需要的，否则项目无法正常编译。

2.3 项目源码的获取、编译与安装

2.3.1 源码获取

项目源码仓库地址为：https://github.com/njucjc/graduation-project，如图2.5所示，可直接通过网页下载，或通过 git clone 命令进行下载。

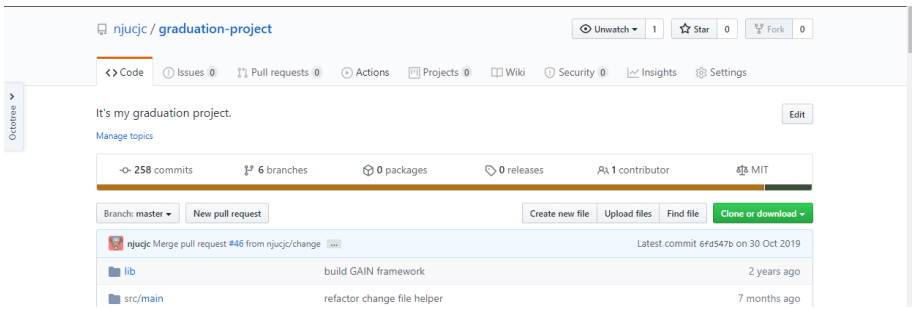

图 2.5: 项目源码仓库

2.3.2 编译

项目采用 IntelliJ IDEA 集成开发环境进行开发编译，可从进行官方网站下载安装，下载时选择 Community 的免费版本即可。

1. 打开 IntelliJ IDEA 集成开发环境，如图2.6所示，选择 Import Project 定位到下载的源码目录，选择导入 Maven 项目即可；

图 2.6: 导入项目

2. 等待 Maven 下载完 2.2.4 节中的第三方库依赖后，如图2.7所示，点击 Build 选择 Build Project 编译源码。

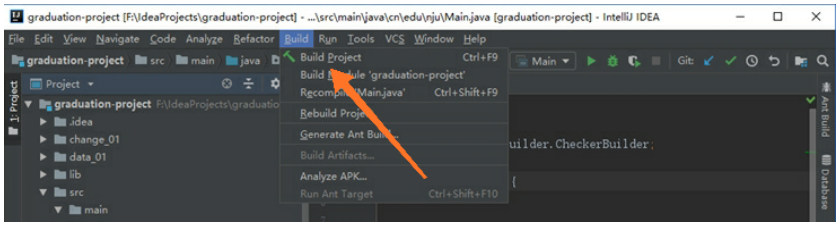

图 2.7: 编译源码

3. 若跳过 2.1.3 及 2.2.3 小节，则可跳过本步骤。
   (a) 打开 Terminal 定位到项目的 src/main/kernel 目录；
   (b) 在 Terminal 输入：nvcc -m32 -O3 -ptx kernel.cu -o kernel.ptx；
   (c) 成功生成 kernel.ptx 文件即可。

2.3.3 安装

除了每次从源码编译运行，项目也支持生成 JAR 包来独立安装运行，我们可以利用 IntelliJ IDEA 集成开发环境生成 JAR 包，步骤如下：

1. 如图2.8所示，打开项目，右键 File 选中 Project Struture；

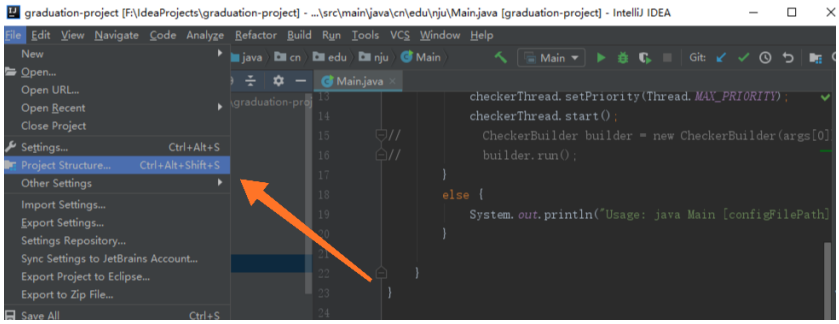

图 **2.8**: 生成 JAR 步骤 1

2. 如图2.9所示，选中 Artifacts，点击“+”号，选择 JAR，再选下面的 from modules with dependencies 即可；

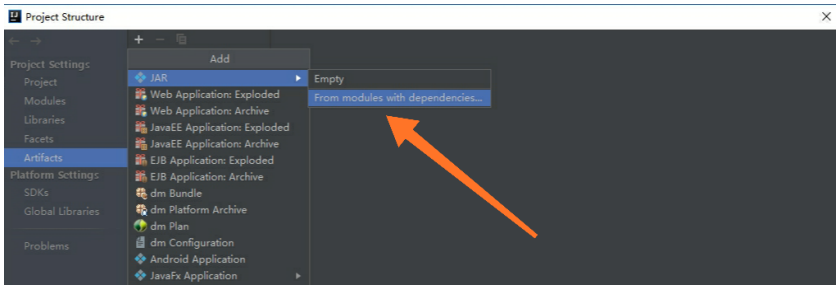

图 **2.9**: 生成 JAR 步骤 2

3. 如图2.10所示，点击 Main Class 选中项目主类，可供选择的主类包括Main、Server和Client类 (它们的区别详见 4.2.3 小节)，这里以Main类为主类举例，点击 OK 进入下一步；

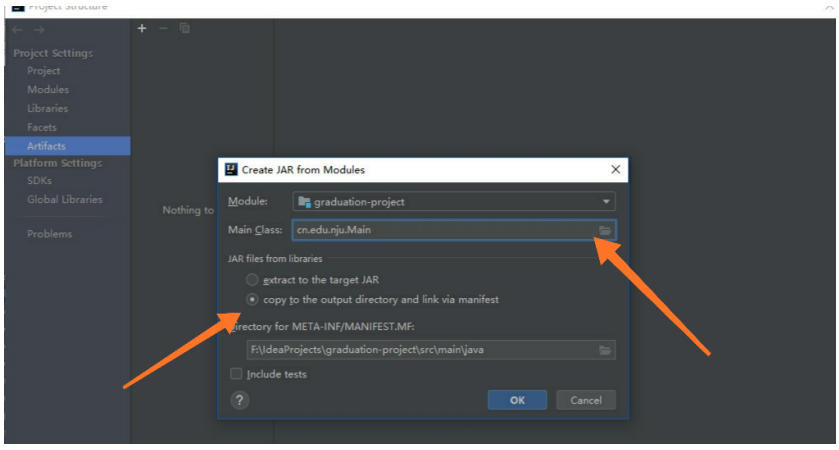

图 **2.10**: 生成 JAR 步骤 3

4. 如图2.11所示，配置图中的四个地方，按照编号分别表示：

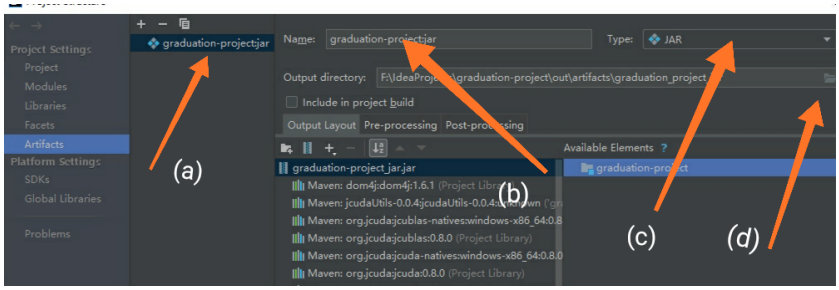

图 **2.11**: 生成 JAR 步骤 4

(a) 对于需要针对哪个 JAR 去打包；

(b) 打出的 JAR 包的名字，可以随意命名；

(c) Type 就是你打出的文件类型，这里选择 JAR；

(d) 打出的 JAR 的生成路径，可以选择你需要放置的任何位置；

5. 如图2.12所示，最后点击 Build，选择 Build Artifacts 即可在 4(d) 中选择的路径生成 JAR 包。

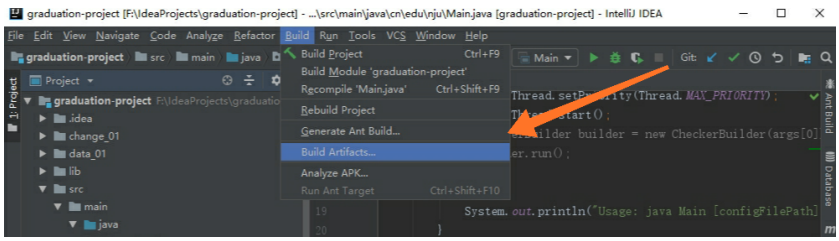

图 **2.12**: 生成 JAR 步骤 5

Part 3. 项目架构与主要组件介绍

3.1 总体架构

项目的总体软件架构如图3.1所示：

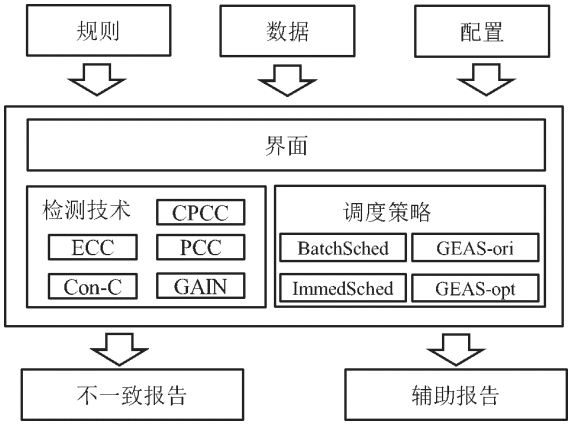

图 3.1: 项目总体架构

主要输入三项信息，这里仅简单说明，更具体的格式等信息将在下个部分 Part 4. 中结合 Demo 说明：

1. 数据：感知程序所关心的上下文信息数据流；
2. 规则：结合感知程序的应用场景人工自定义的一致性规则信息；
3. 配置：用于配置项目中集成的一致性检测技术和调度策略等信息。

主要输出两项信息，这里仅简单说明，更具体的格式等信息将在下个部分 Part 4. 中结合 Demo 说明：

1. 不一致报告：报告数据中的违反规则的不一致信息；
2. 其他辅助报告：报告检测时间等额外信息。

3.2 主要组件

项目在实现层面上，主要包含以下四大核心组件，本节将重点介绍它们结构和作用：

1. ContextParser：上下文信息解析器；
2. Checker：一致性检测技术算法接口；

3. Scheduler：调度策略算法接口；
4. AbstractBuilder：抽象构造器，解析配置。

3.2.1 ContextParser

ContextParser 的主要工作是从文件中读出的原始数据进行解析和加工，它的接口结构如图3.2所示，它的结构十分简单，只含有一个对外接口 parserContext, 这个方法接收一个表示上下文信息的字符串数据，然后将这个字符串的各个域解析成一个Context 结构，这个结构完全表示了原本字符串中所表示的信息，对于具体的感知程序所关心的上下文信息，Context 及 ContextParser 都需要根据给定的字符串数据进行重新构造，在后续的 Part 4. 中会给出 Demo。

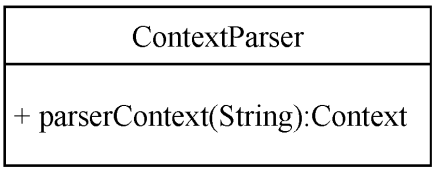

图 3.2: ContextParser **接口结构**

3.2.2 Checker

项目支持多种检测技术如图3.3(a) 所示，虽然它们之间的内部算法各不相同，但是它们对外表现出来的功能是一致的，因此，从它们对外的共性中能够抽象出一组对外的接口，然后根据它们内部算法的差异各自实现这组接口，如下：

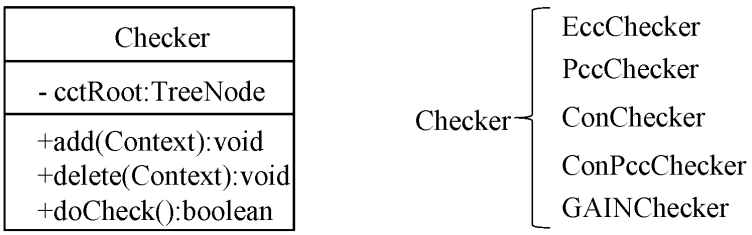

图 3.3: (a) Checker **接口结构** (b) **项目支持的** Checker

1. add：添加一个 Context，并调整算法内部数据结构；
2. delete：删除一个 Context，并调整算法内部数据结构；
3. doCheck：根据算法调度一次一致性检测。

如图3.3(b) 所示，项目共实现以下五种检测技术：

1. 完全检测 (ECC)：对应EccChecker类；
2. 增量部分检测 (PCC[1])：对应PccChecker类；
3. CPU 并发检测 (Con-C[3])：对应ConChecker类；
4. CPU 并发的增量部分检测 (CPCC)：对应ConPCCchecker类；
5. GPU 并行检测 (GAIN[4])：对应GAINChecker类。

3.2.3 Scheduler

如图3.4(a) 所示，与 Checker 的设计思路类似，系统同样需要支持多中调度策略，它们的行为都是根据上下文改变的情况，为是否要调度一次上下文一致性检测做出决策。它支持的接口如下：

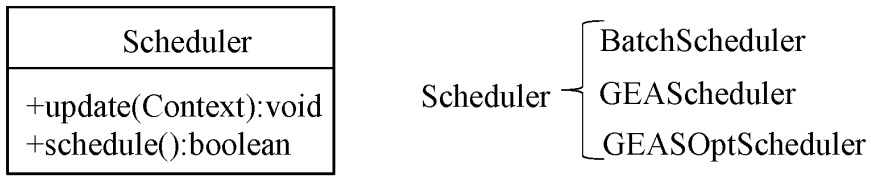

图 3.4: (a) Scheduler **接口结构** (b) **项目支持的** Scheduler

1. update：负责记录上下文信息改变状态；
2. schedule：根据当前状态返回一个布尔值，为 true 表示此时应该调度一次一致性检测，为 false 表示不调度一致性检测。

如图3.4(b) 所示，项目共支持以下四种调度策略：

1. 固定批量调度 (BatchSched)：按照窗口的大小，每 n ($n \geq 2$) 条上下文信息改变进行一次一致性检测，对应BatchScheduler类；
2. 即时调度 (ImmedSched)：窗口大小 1 的 BatchSched，对应BatchScheduler类；
3. 自适应批量检测 (GEAS-ori[5])：窗口大小自适应，对应GEAScheduler类；
4. 自适应批量检测优化版 (GEAS-opt[6])：窗口大小自适应 GEAS-ori 的优化升级，对应GEASOptScheduler类。

3.2.4 AbstractBuilder

AbstractBuilder 作用就是分析用户给定的配置文件，并根据配置文件实例化用户指定的 Checker 和 Scheduler 等配置，然后启动上下文一致性检测过程。

系统的核心组件之间的关系如图3.5所示，AbstractBuilder 通过 parseConfigFile 接口在运行时根据配置文件的指示分别获取 Checker 和 Scheduler 的实例对象，这样设计的好处是能够解耦一个复杂系统，能够很好地解决系统维护和拓展的问题，当我们需要添加一种检测技术和调度策略时，只需要根据 Checker 和 Scheduler 的接口要求实现具体的算法即可。

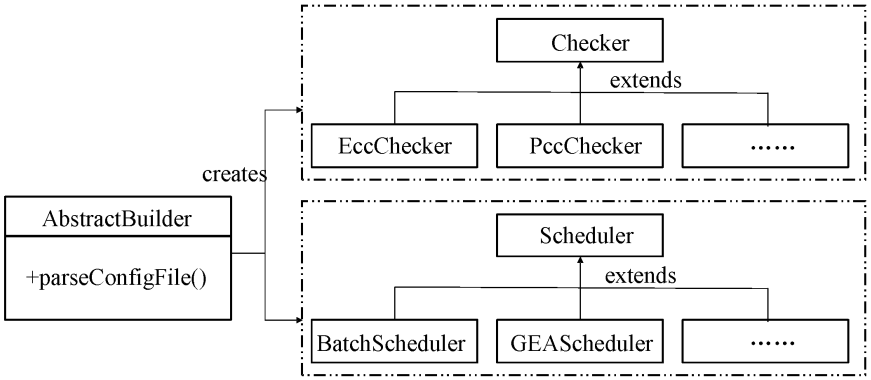

图 3.5: AbstractBuilder 与 Checker 和 Scheduler 的关系示意图

3.2.5 组件间的协作关系

项目各个组件间的协作关系如图3.6所示，首先是由 AbstractBuider 模块根据配置实例化特定的 Scheduler 和 Checker 模块，其中 Checker 中包含了用户自定义的一致性规则，之后数据通过 ContextParser 模块转化为 Context 结构体传入 Checker，最后 Checker 在 Scheduler 的调度下进行一致性检测。

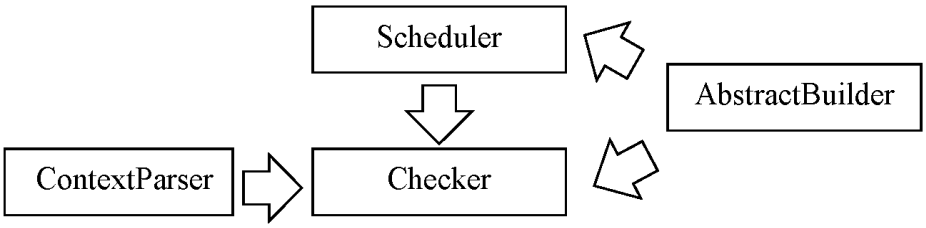

图 3.6: 组件间的协作关系

3.3 静态检测与动态检测

如图3.7所示，项目支持静态检测与动态检测两种运行方式：

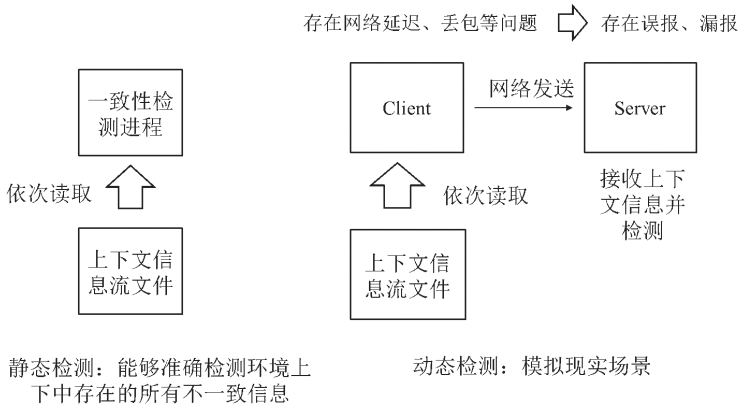

图 3.7: **静态检测和动态检测**

1. 静态检测：如图3.7(a) 所示，系统不断地读取本地上下文信息数据流文件，并根据读到的数据依次检测上下文信息一致性；
2. 动态检测：如图3.7(b) 所示，用 C/S 架构模拟现实中上下文信息的收集和检测的过程，即由 Client 端按照数据文件中上下文信息的时间戳间隔将上下文信息通过网络发送给 Server，并由 Server 对从 Client 端接收到的上下文信息进行一致性约束检测。

静态检测的优势是能够准确检测环境上下中存在的所有不一致信息，但是，它存在的问题是它不能处理一些现实生活中存在的实际情况，例如，在现实中，上下文感知程序通常是 C/S 架构的，程序所需要的上下文信息大多是由 Client 端的传感器感知并通过无线传输发送给在 Server 端上下文感知程序，并在使用这些信息之前进行一致性约束检测。若某个时刻发送到 Server 端的上下文信息过多 (工作负载大)，而一致性检测的处理的过程很慢，导致接收到的上下文信息不能够被及时检测，就有可能造成某些上下文信息的丢失 (网络丢包)，进而导致一致性错误的误报和漏报。静态检测无法模拟这种场景，所以它给出的检测结果不足以反映在现实情景下某种检测技术或调度策略对检测效率及准确度的提升。因此，我们在系统中添加了动态检测方案，它的优势是能够完全模拟现实生活中感知程序的工作方式，包括在工作负载过大时，网络的丢包、延迟等情况对上下文一致性约束检测的影响。同时采用静态检测和动态检测的方式，

能够将发挥它们各自的优势，又能进一步弥补各自的缺陷。用户可根据自身实际应用场景进行选择，至于如何配置选择将在 Part 4. 中结合 Demo 说明。

Part 4. 项目运行与 Demo 分析

4.1 项目的输入与输出

项目的正常运行需要用户提供以下四个文件：

1. rules.xml：描述一系列自定义的一致性规则，将其写成 XML 格式；
2. patterns.xml：描述一系列数据集合，将集合的特性写成 XML 格式；
3. data.txt/data_changes.txt：表示数据流文件，支持两种格式分别对应这两个文件，用户仅需提供其一，区别后续结合 Demo 说明；
4. config.properties：配置文件。

项目正常运行后输出一个 LOG 文件，主要包含以下两项信息：

1. 数据文件中的数据违反的规则情况；
2. 规则违反的总个数；
3. 本次检测的检测时间。

有关于上述输入、输出的具体格式，我将在 4.2 小节中结合具体 Demo 给出详细说明，用户可根据自己的实际应用需求做出相应的准备和修改。

4.2 Demo 1：出租车场景

4.2.1 场景描述

假设纬度在 [20, 30] 之间的某城市中有一个出租车打车程序利用出租车的实时位置等数据为用户提供打车服务，为了给用户提供更加良好的打车服务，出租车打车程序在使用从各个出租车传达的实时位置等信息时，需要判定其合理性，以免影响程序的正常运行。显然，打车程序所需要的合理的出租车数据是它所处的经度必须在这个城市的范围内，否则，打车程序无法对其进行正确分配客流。为此需要我们的项目对传入打车程序的出租车数据进行一个预先的一致性检测[2]。

[2]本 Demo 对应项目的 master 分支，可在项目中使用 git checkout master 命令进行切换。

4.2.2 运行准备

根据 4.2.1 中描述的场景和 4.1 中的项目运行输入需求我们需要依次准备如下 4 个文件：

1. rules.xml：即写成 XML 格式的由一阶逻辑语言描述的一致性规则文件，从出租车场景中我们可以定义一个一致性规则为“所有出租车必须处于纬度 [20, 30] 之间”，将其翻译成一阶逻辑语言为 $\forall v_1 \in pat\_000\,(sz\_loc\_range(v_1))$，将其写成 XML 格式如图4.1所示，每个 <rule> 元素共有 2 个子元素：

```xml
<rule>
    <id>rule_00</id>
    <formula>
        <forall var = "v1" in = "pat_000">
            <bfunction name = "sz_loc_range">
                <param pos = "1" var = "v1" field = "object" />
            </bfunction>
        </forall>
    </formula>
</rule>
```

图 4.1: 规则的 XML 描述示例

(a) <id>：其文本内容表示该 rule 的名称，可任意修改，但不能出现重复名称；

(b) <formula>：其包含的是用一阶逻辑描述的一致性规则的 XML 形式，它的一般描述方式如下；

i. 对于全程命题“$\forall v \in pattern\, P(v)$”，XML 形式如图4.2所示；

```xml
<forall var="v" in="pattern">
    <!-- pattern是由patterns.xml描述的特定数据集合 -->
    P(v)的XML形式
</forall>
```

图 4.2: 全称命题的 XML 形式

ii. 对于存在命题“$\exists v \in pattern\, P(v)$”，XML 形式如图4.3所示；

```xml
<exists var="v" in="pattern">
    <!-- pattern是由patterns.xml描述的特定数据集合 -->
    P(v)的XML形式
</exists>
```

图 4.3: 存在命题的 XML 形式

iii. 对于一阶公式“$\neg P(v)$”，XML 形式如图4.4所示；

```
<not>
    P(v)的XML形式
</not>

```

图 4.4: ¬ 的 XML 形式

iv. 对于一阶公式“$P(x) \wedge Q(y)$”，XML 形式如图4.5所示；

```
<and>
    P(x)的XML形式
    Q(y)的XML形式
</and>
```

图 4.5: ∧ 的 XML 形式

v. 对于一阶公式“$P(x) \vee Q(y)$”，XML 形式如图4.6所示；

```
<or>
    P(x)的XML形式
    Q(y)的XML形式
</or>
```

图 4.6: ∨ 的 XML 形式

vi. 对于一阶公式“$P(x) \rightarrow Q(y)$”，XML 形式如图4.7所示；

```
<implies>
    P(x)的XML形式
    Q(y)的XML形式
</implies >
```

图 4.7: → 的 XML 形式

vii. 对于布尔函数 $sz\_loc\_range(v_1)$，XML 形式如图4.8所示，它必须根据具体场景实际构造[1]；

```
<bfunction id=" sz_loc_range">
    <!-- id表示函数名 -->
    <param pos="1" var="v1" field = "object">
</bfunction>
```

图 4.8: 布尔函数 $sz\_loc\_range(v_1)$ 的 XML 形式

2. patterns.xml：写成 XML 格式的描述数据集合特征的文件，它所描述的集合供 rules.xml 中的全程命题和存在命题使用，其物理意义是某些规则仅

[1]可参照 Demo 修改BFuncHelper的 bfunc 方法构造布尔函数。

针对特定的数据进行检测，因此我们需要这个文件描述规则所关心的数据，并把它们过滤出来供规则使用，因此 Pattern 的含义是需要自定义的[1]，例如规则 $\forall v_1 \in pat\_000\,(sz\_loc\_range(v_1))$ 中的 pat_000 的形式如图4.9所示，每个 <pattern> 元素共有 7 个子元素：

```
<pattern>
    <id>pat_000</id>
    <freshness>2000</freshness>  <!-- ms -->
    <category>location</category>
    <subject>any</subject>
    <predicate>any</predicate>
    <object>any</object>
    <site>any</site>
</pattern>

```

图 4.9: Pattern 的 XML 描述示例

(a) <id>：其文本内容是该 pattern 的名称，可任意修改，但不能出现重复名称；

(b) <freshness>：其文本内容表示数据加入 pattern 表示的数据集合的时间限制，如取值为 2000 表示集合中数据的有效时间为 2000 毫秒；

(c) <category>：元素表示 pattern 的类型，可以任意修改，只作为注释用；

(d) <subject>：在本例中元素文本内容默认为 any，表示对 pattern 描述的数据集合中对数据主体不作限制，可根据具体情况修改；

(e) <predicate>：在本例中元素文本内容可取值为 run_with_service、run_without_service 和 any，分别表示有载客、无载客和任意（有载客和无载客皆可），在特定场景中可以被赋予其它意义，表示数据主体的行动；

(f) <object>：在本例中元素文本内容默认为 any，示对 pattern 描述的数据集合中对数据客体不作限制，可根据具体情况修改；

(g) <site>：在本例中元素文本内容可取值为 sutpc_x（x 为 0~9 的数字）和 any，分别表示尾号为 x 的车辆和任意车辆，用户可根据具体应用场景重新定义其含义。

3. data.txt/data_changes.txt：待检测的数据文件，二者取其一；

(a) data.txt：为 time-based 格式，在该出租车场景中它可以是如图4.10所示的格式；

[1]可参照 Demo 修改Pattern类的 isBelong 方法，解析判断 Context 是否属于 Pattern。

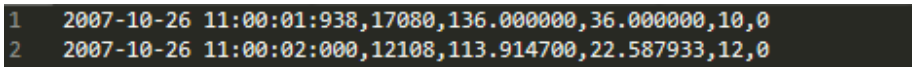

图 4.10: Time-based 格式出租车数据示例

i. 其中每一行表示单独一条出租车数据，每个数据域之间以英文逗号分隔，从左到右依次表示“时间戳、车牌号、经度、纬度、速度、载客量”；

ii. 每一行数据可以由 ContextParser 解析成一个 Context 对象传入 Checker 进行检测，每个 Context 对象包含每行数据的所有属性，若应用场景不同，则 ContextParser 和 Context 需要根据实际情况加以修改[1]，本例中数据域与Context类域的对应关系如表4.1所示 。

表 4.1: 数据域与 Context 域对应关系

数据域	Context 类域
无对应，从整个文件每一行从 0 开始递增	id
时间戳	timestamp
车牌号	plateNumber
经度	longitude
纬度	latitude
速度	speed
载客量	status

(b) data_changes.txt：如图4.11所示为图4.10中 time-based 格式数据对应的 change-based 格式数据。

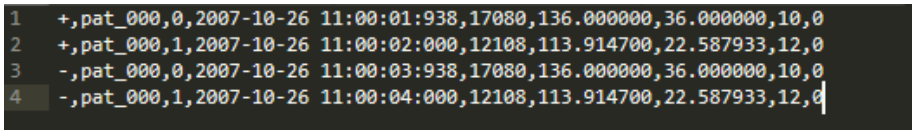

图 4.11: Change-based 格式出租车数据示例

i. 每一行表示对一个 pattern 集合的增删操作，每个数据域间以英文逗号分隔，从左到右依次表示“增/删、集合名称、Context 编号(从 0 开始)、时间戳、车牌号、经度、纬度、速度、载客量”；

[1]可参照 Demo 的ContextParser类的 parseContext 方法解析数据和Context类属性定义。

ii. 根据每条 time-based 数据所属的 pattern 集合产生一个增操作的 change-based 数据，由于图4.10中有 2 条 time-based 格式数据，因此在图4.11中产生了行号为 1 和 2 的 change-based 数据；

iii. 再根据 pattern 描述的 freshness 信息产生一条删操作的 change-based 格式数据，由于 pat_000 的 freshness 是 2000 毫秒，所以依次产生了图4.11中行号为 3 和 4 的 change-based 数据；

iv. 总的来说，change-based 格式数据根据 pattern 描述的信息显式写明了数据添加进和被删除出 pattern 集合的时机。

4. config.peoperties：项目配置文件，它的形式如图4.12所示，共有 9 个配置项。

```
ruleFilePath = rules.xml
patternFilePath = patterns.xml
dataFilePath = data.txt
changeFilePath = data_changes.txt
logFilePath = output.log
technique = PCC
schedule = GEAS
taskNum = 4
changeHandlerType = static-change-based
```

图 4.12: 配置文件示例

(a) ruleFileParh：rules.xml 文件的路径；

(b) patternFilePath：patterns.xml 文件的路径；

(c) dataFilePath：data.txt 即 time-based 文件的的路径，它与 changeFilePath 其中一个有效即可；

(d) changeFilePath：data_change.txt 即 change-based 文件的路径，它与 dataFilePath 其中一个有效即可；

(e) logFilePath：输出文件的路径，可以为任意有效路径；

(f) technique：配置检测技术；

i. 可以是 3.2.2 小节中 Checker 的任意一种，即 ECC、PCC、Con-C、GAIN 或 CPCC；

ii. 需要注意的是这里取 CPCC 时，仅能在 (g) 中选择 GEAS-ori/opt，否则运行会产生错误；

(g) schedule：配置调度策略；

i. 可以是 3.2.3 小节中 Scheduler 的任意一种，即正整数 n (取 $n = 1$ 表示 ImmedSched，取 $n \geq 2$ 表示 BatchSched 的窗口大小)、GEAS-ori/opt；

ii. 需要注意的是这里取 GEAS-ori/opt 时，仅能在 (i) 中选择 change-based，否则运行会产生错误；

(h) taskNum：任意正整数，表示 technique 配置为 Con-C 或 CPCC 下的并发粒度；

(i) changeHandlerType：配置运行方式；

i. 它可以取值为 (static/dynamic)_(change/time)_based，括号中二选一；

ii. 其中 static 表示运行静态检测，dynamic 表示运行动态检测，它们的区别见 3.3 小节；change-based 表示读取 changeFilePath 数据，time-based 表示读取 dataFilePath 中的数据。

另外，对于配置中各个“检测技术 + 调度策略”的组合中，它们的大致性能排名如表4.2所示，名次越小性能越高；它们的资源开销排名如表4.3所示，名次越小开销越大，需要说明的是由于 BatchSched 的性能与开销和具体窗口大小的选择相关，窗口越大性能越高资源开销也越大，故这里不单独列出，表中的“—”表示该组合不适用。

表 4.2: 各个技术组合大致性能排名

	ECC	**Con-C**	**GAIN**	**PCC**	**CPCC**
ImmedSched	14	13	12	8	——
GEAS-ori	11	10	6	5	3
GEAS-opt	9	7	4	2	1

表 4.3: 各个技术大致资源开销排名

	ECC	**Con-C**	**GAIN**	**PCC**	**CPCC**
ImmedSched	14	11	10	9	——
GEAS-ori	13	8	6	4	2
GEAS-opt	12	7	5	3	1

4.2.3 运行与结果分析

注备好上述 4 个文件后就可以开始运行项目(这里仅展示从源码中编译启动，也可从生成的 JAR 中启动，运行参数皆为 config.properties 的文件路径，这里不再赘述，JAR 包生成参看 2.3.3 小节)，并分析最终的 LOG 结果：

1. 运行静态检测：静态检测的主类是Main类，如图4.13所示，它以 config.properties 文件路径作为参数，若无错误信息输出则表示正常运行；

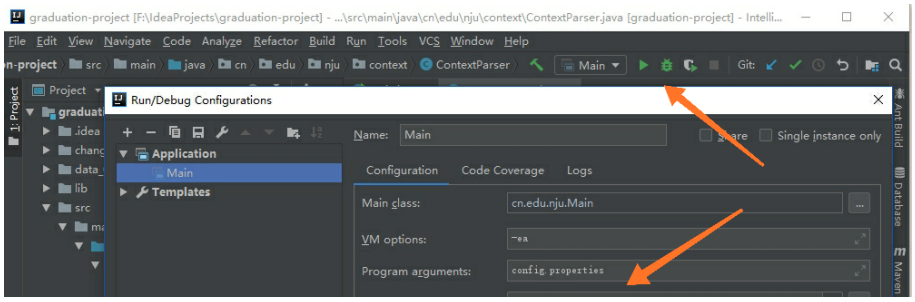

图 4.13: 启动静态检测

2. 运行动态检测：动态检测需要先如图4.14所示，以 config.properties 文件路径作为参数，启动Server，待控制台提示成功启动 Server，再如图4.15所示，以 config.properties 文件路径作为参数，启动Client，若无错误信息输出则表示正常运行 (在同一项目里依次以 Server 和 Client 为主类启动即可)；

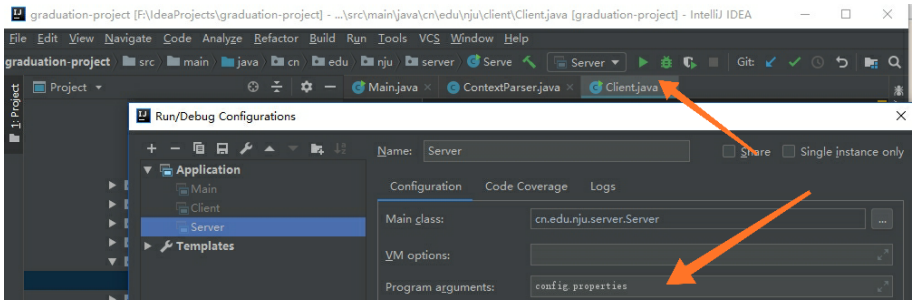

图 4.14: 启动 Server

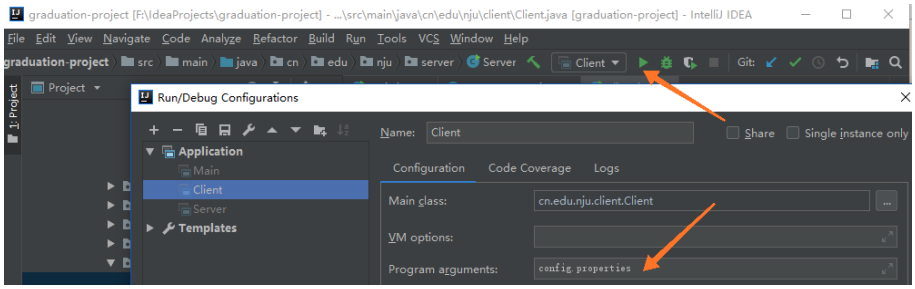

图 4.15: 启动 Client

3. 结果分析：无论是运行静态检测还是动态检测，正确运行后都会在 config.properties 中的 logFilePath (用户自主配置) 指定的路径中产生一个 LOG 文件，如图4.16所示；

```
1  rule_00 ctx_0
2  Total INC: 1
3  run time: 10 ms
```

图 4.16: LOG 文件示例

(a) 第 1 行“rule_00 ctx_0”表示编号为 0 的 Context 违反了 rule_00，也即图4.10中的第 1 行 (Context 编号为 0) 出租车数据违反了车辆应在纬度 [20, 30] 之间的一致性约束，类似地除了最后一行其余所有行都是在当前规则下所有的数据违反，在这个例子中只有一个违反，用户可根据实际应用场景来类比解读；

(b) 倒数第二行表示本次检测中总共检测到的不一致报告数量，这里仅有“rule_00 ctx_0”这一个，所以值为 1；

(c) 最后一行表示整个检测运行时间为 10 ms，用于对项目的性能分析。

4.3 Demo 2：高速收费场景

4.3.1 场景描述

假设某高速公路如图4.17所示，车辆从 Z0 进入该高速公路领取磁卡，经过门架 3、4、5、6 到达出站口 Z1，正常情况下，磁卡上记录的路径信息应该为“Z0-3-4-5-6-Z1”，但是由于门架不能区分车辆方向，在经过 4 和 5 号门架时，其反向的门架 D 和 E 先与车辆磁卡建立通信进行了交易，导致磁卡最终记录的路径信息变成了“Z0-3-D-E-6-Z1”，此时在收费过程的路径还原时，由于没有修复标记的反向门架 D 和 E 的信息，而是认为车辆从远端可以绕回的路径上经过了

门架 D 和 E，重新绕回了正常路径上，最终路径还原结果为“Z0-3-4-5-6-7-8-H-G-F-E-D-C-B-A-1-2-3-4-5-6-7-8-H-G-F-E-D-C-B-A-1-2-3-4-5-6-Z1”，与正确的路径相差甚远，导致高速收费异常。为此，我们需要制定一致性规则，预先检测和修复磁卡上记录的门架路径信息[1]。

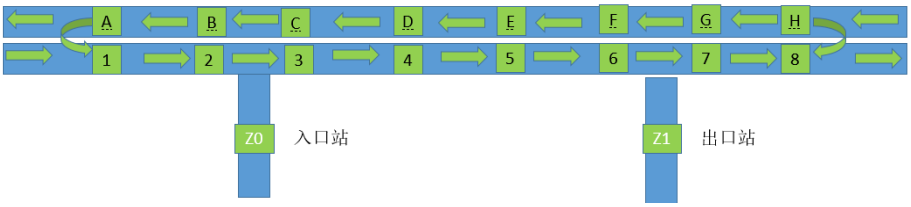

图 4.17: 高速收费示例

4.3.2 运行准备

与 4.2.2 中的 Demo 1 类似，我们需要准备 4 个输入文件：

1. rules.xml：针对上述反向门架的场景，我们可以制定一个一致性规则为“若门架 v1 记录在门架 v2 之前，则 v1 和 v2 之间存在路径长度为 1 的路径”，若违反规则则说明 v1 和 v2 都是反向门架，其 XML 格式如图4.18所示，如何写成 XML 格式不再赘述；

```xml
<rule>
    <id>rule_00</id>
    <formula>
        <not>
            <exists var = "v1" in = "pat_0">
                <exists var = "v2" in = "pat_1">
                    <and>
                        <bfunction name = "before">
                            <param pos = "1" var = "v1" field = "id" op = "" default_value = ""/>
                            <param pos = "2" var = "v2" field = "id" op = "" default_value = ""/>
                            <param pos = "3" var = ""   field = ""   op = "" default_value = "2"/>
                        </bfunction>
                        <bfunction name = "conn">
                            <param pos = "2" var = "v1" field = "code" op = "" default_value = ""/>
                            <param pos = "1" var = "v2" field = "code" op = "" default_value = ""/>
                            <param pos = "3" var = ""   field = ""     op = "" default_value = "1" />
                        </bfunction>
                    </and>
                </exists>
            </exists>
        </not>
    </formula>
</rule>
```

图 4.18: rule_00 的 XML 格式

2. patterns.xml：如图以上述规则中的 pat_0 为例；

[1]本 Demo 对应项目的 highway 分支，可在项目中使用 git checkout highway 命令进行切换

```xml
<pattern>
    <id>pat_0</id>
    <freshness>3</freshness>
    <category>highway</category>
    <subject>any</subject>
    <predicate>any</predicate>
    <object>any</object>
    <site>any</site>
</pattern>
```

图 4.19: rule_00 的 XML 格式

(a) 这里仅需关注 freshness，它的值表示 pattern 集合最多可容纳的 Context 数目，这里取值为 3 表示 pat_0 最多只含 3 个 Context，在第 4 个 Context 进入集合前需要先删除一个最先进入集合的 Context；

(b) 其余属性含义与 Demo 1 中类似，这里不需要使用。

3. 数据文件：本例中的 time-based 数据格式，可以如图4.20所示；

```
Z0,3
3,0
D,0
E,0
6,0
Z1,3
```

图 4.20: 高速收费 time-based 格式数据示例

(a) 每一行数据代表一个门架信息，每个域以逗号分隔，依次表示“门架编号、门架类型 (0-普通门架，3-收费站)”；

(b) 每一行数据可以由 ContextParser 解析成一个 Context 对象，本例中数据域与Context类域的对应关系如表4.4所示

表 4.4: 数据域与 Context 域对应关系

数据域	Context 类域
无对应，从整个文件每一行从 0 开始递增	id
无对应，取相对时间，从 0 开始递增	timestamp
门架编号	code
门架类型	type

(c) 可以依据 pattern 信息，相应转换成 change-based 格式，如图4.21所示展示其中一部分；

```
+,pat_0,0,0,Z0,3
+,pat_1,0,0,Z0,3
+,pat_0,1,1,3,0
+,pat_1,1,1,3,0
+,pat_0,2,2,D,0
+,pat_1,2,2,D,0
-,pat_0,0,3,Z0,3
-,pat_1,0,3,Z0,3
+,pat_0,3,3,E,0
+,pat_1,3,3,E,0
```

图 4.21: 高速收费 change-based 格式数据示例

i. 每一行代表一个对 pattern 集合的操作，每个数据域间以英文逗号分隔，依次表示” 增/删、集合名称、Context 编号 (从 0 开始)、时间戳、门架编码、门架类型 (0-普通门架，3-收费站)”。

4. config.properties：配置方式与 Demo 1 类似，这里不再赘述。

4.3.3 运行检测、结果分析与上下文修复

1. 运行检测：运行方式与 Demo 1 类似，这里不再赘述；
2. 结果分析：配置运行结束后，LOG 信息如图4.22所示，显示门架 D (ctx_2) 与门架 E (ctx_3) 违反了我们制定的规则 (rule_00)，即门架 D 记录在门架 E 之前，但门架 D 和 E 间不存在路径长度为 1 的路径；

```
rule_00 ctx_2 ctx_3
Total INC: 1
run time: 5 ms

```

图 4.22: LOG 文件示例

3. 上下文修复：根据规则 rule_00 的语义，我们知道门架 D 与门架 E 是反向门架，因此修复过程就是把门架 D 和 E 都更新为其反向门架 4 和 5，更多修复细节参见代码Repair类。
   (a) 注意这里的修复是根据规则的语义来进行的，不同的规则可以自定义不同的修复方式，例如违反规则 rule_00 一定是反向门架，故只需在检测出 INC 后将对应的门架记录取反即可。

参考文献

[1] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," ACM Transaction on Software Engineering and Methodology (TOSEM), vol. 19, no. 3, pp. 9:1–9:61, 2010.

[2] C. Xu, S. C. Cheung, and W. K. Chan, "Incremental consistency checking for pervasive context," in Proceedings of the 28th international conference on Software engineering (ICSE). ACM, 2006, pp. 292–301.

[3] C. Xu, Y. Liu, S. C. Cheung, C. Cao, and J. Lv, "Towards context consistency by concurrent checking for internetware applications," Science China Information Sciences, vol. 56, no. 8, pp. 1–20, 2013.

[4] J. Sui, C. Xu, S. C. Cheung, W. Xi, Y. Jiang, C. Cao, X. Ma, and J. Lv, "Hybrid CPU–GPU constraint checking: Towards efficient context consistency," Information and Software Technology (IST), vol. 74, pp. 230–242, 2016.

[5] B. Guo, H. Wang, C. Xu, and J. Lv, "GEAS: Generic adaptive scheduling for highefficiency context inconsistency detection," in Proceedings of International Conference on Software Maintenance and Evolution (ICSME), pp. 137–147, 2017.

[6] H. Wang, C. Xu, B. Guo, X. Ma, and J. Lv, "Generic adaptive scheduling for efficient context inconsistency detection," IEEE Transactions on Software Engineering (TSE), 2020.