arXiv:2512.14514v1 [cs.LO] 16 Dec 2025

嵌入式系统相关 HAL 接口要求

Manuel Bentele[1,3],Andreas Podelski[1],
Axel Sikora[2,3],和 Bernd Westphal[4]

[1] 德国弗赖堡大学，弗赖堡[2] 德国奥芬堡应用科学大学，奥芬堡[3] 德国哈恩-席卡德研究所，菲林根-施文宁根[4] 德国航空航天中心，奥尔登堡

Abstract. [Context & Motivation] Embedded applications often use a Hardware Abstraction Layer (HAL) to access hardware. Improper use of the HAL can lead to incorrect hardware operations, resulting in system failure and potentially serious damage to the hardware. **[Question & Problem]** The question is how one can obtain prioritize, among a possibly large set of HAL interface requirements, those that are indisputably relevant for preventing this kind of system failure. **[Ideas & Results]** In this paper, we introduce a formal notion of relevance. This allows us to leverage a formal method, i.e., software model checking, to produce a mathematical proof that a requirement is indisputably relevant. We propose an approach to extract provably relevant requirements from issue reports on system failures. We present a case study to demonstrate that the approach is feasible in principle. The case study uses three examples of issue reports on embedded applications that use the SPI bus via the *spidev* HAL. **[Contribution]** The overall contribution of this paper is to pave the way for the study of approaches to a new kind of prioritization aimed at preventing a specific kind of system failure.

关键词： 需求优先级排序 · 嵌入式系统 · 硬件抽象层（HAL）· HAL 接口需求 · 串行外设接口（SPI）· 形式化方法 · 软件模型检验。

1 引言

开发使用 HAL 访问嵌入式系统硬件的应用程序代码，会带来特定的需求工程挑战。除了通常在项目启动时定义的系统和软件需求外，HAL 接口需求往往被忽视，直到项目后期才予以考虑，甚至可能为时已晚：不当使用 HAL 可能导致硬件操作错误，进而造成系统故障，甚至可能严重损坏硬件。

Unlike system requirements, which vary depending on the specific application, interface requirements are determined by the HAL and can be reused across multiple system designs. There are, however, many of those. The fact that the

number of HAL interface requirements can be large and overwhelming, may be one reason why they are often ignored until their violation has indeed caused a system failure. This raises naturally the question how one can prioritize, among a possibly large set of HAL interface requirements, those that are indisputably relevant for preventing this kind of system failure that can lead to potentially serious hardware damage.

需求优先级排序在需求工程中发挥着重要作用 [13,6,19]。优先级排序需要一个可用于选择需求的标准。问题在于，人们往往对标准缺乏清晰的理解，导致优先级排序带有主观性且前后不一致 [13,6,19]。

In this paper, we address exactly this problem and introduce a novel criterion which can be defined precisely and unambiguously. The criterion of relevance can be used to prioritize requirements that are indisputably relevant for preventing system failures. We call this criterion *relevant*, for short. The definition of relevance allows us to leverage a formal method, i.e., software model checking [10], to produce a mathematical proof that a requirement is relevant (and, in this sense, indisputably relevant).

We propose an approach to extract provably relevant requirements from issue reports on system failures. The approach comes in two versions. In the first version, one infers a candidate requirement from the issue report and then applies software model checking to check whether the candidate requirement is a relevant requirement, or not. In the second version, one assumes that one is given a (possibly large) set of HAL interface requirements for, say, a specific HAL. The approach is then to take the whole set as a set of candidate requirements and apply software model checking to each of them. In the second version, the approach relies on the fact that the application of software model checking as a formal method is fully automatic.

我们通过一个初步案例研究来证明该方法原则上可行。案例研究使用了三个涉及嵌入式应用的问题报告示例，这些应用通过 *spidev* HAL 使用 SPI 总线。对于该方法的第二个版本，案例研究采用了一组针对 *spidev* HAL 的 26 项接口要求（并对三个应用程序各自的两个版本逐一检查这些要求）。这些要求必须手动提取。它们构成了 *spidev* HAL 所有特定形式的接口要求集合。

本文的贡献在于定义了一项精确且明确的标准，可用于确定优先级；提出一种运用该标准的方法；并通过案例研究证明该方法在原则上是可行的。总体而言，本文或许为研究一种新型优先级确定方法铺平了道路，该方法旨在预防特定类型的系统故障。

The paper is structured as follows: In Section 2, we introduce our approach. In Section 3, we present the experiments to evaluate the approach. In Section 4, we discuss related work. Section 5 is the conclusion section.

2 一种基于相关性的方法

In this section, we introduce a formal notion of relevance and an approach. Given an issue report on a system failure, we present two variants of the approach: one that extracts a single interface requirement from the issue and checks its relevance, and the other checks a given set of candidate requirements for relevance.

2.1 相关性的概念

直观而言，如果某项需求的有效性对于防止系统故障至关重要，我们就认为该需求是相关的。定义 1 精确地描述了这一直觉。

定义 1（相关性）。如果存在一个具有两个版本的系统，则要求 R 是相关的：一个是有故障的版本，即会出现系统故障的系统；另一个是修复后的版本，即不再出现该系统故障的系统，并且原始应用程序在有故障的系统中违反 R，而新的应用程序在修复后的系统中满足 R。

对于定义 1，我们有两点说明：（1）该定义基于存在性条件。也就是说，我们有一个可用于确认相关性的判据，但无法据此断定某项接口需求永远不会变得相关。（2）为了使该定义在实践中有用，我们需要将其应用于现有的系统故障。从理论上讲，构造一个人工系统来证明给定接口需求的相关性，或许始终是可能的。

2.2 提取并检查单条候选需求的相关性

The approach involves several stakeholders. Hardware manufacturers are only partly involved. They provide the HAL, particularly its documentation and occasionally the source code. However, the extent and quality of the HAL documentation are often moderate. Development companies are the primary users of the approach. Their embedded software developers are especially interested in ensuring that the application code interacts correctly with the HAL. The goal here is to prioritize interface requirements in order to single out the relevant ones. Developers can then concentrate on the relevant interface requirements to write more reliable application code for embedded systems. Figure 1 illustrates the approach embodied as a process chart. The approach comprises two parts. Initially, we extract a candidate requirement from a given issue report and subsequently, we formally prove the relevance of the candidate requirement. A detailed description of each part is provided below to explain the steps involved.

从问题报告中提取候选需求。在流程的第一部分，开发人员从给定的问题报告中提取候选需求。该报告描述了使用 HAL 的故障嵌入式系统中观察到的系统故障（见图 1 中的蓝色方框）。提取出的需求作为可能导致系统故障的候选项，即违反该需求表明发生了故障，而满足该需求则证明故障不存在。

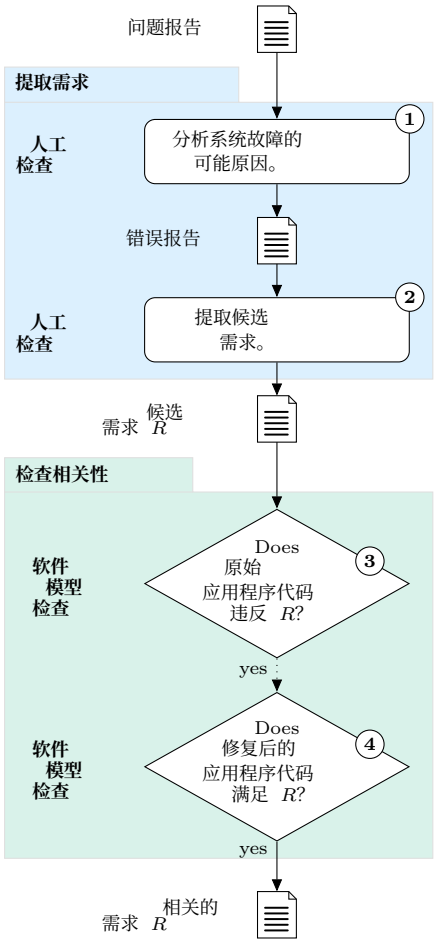


图1. 总体方法。应用程序代码有两个版本：原始版本会导致系统故障，修复后的版本不会。

Inspect issue report. In Activity 1 (Manual inspection), the developer carefully analyzes the cause of the system failure from the issue report in the faulty system to create an error report. Such a report should identify and describe the error, its underlying cause, and include details about the current configuration and state at failure. If the issue report already contains these details, we can skip the activity and continue with Activity 2. Finally, the activity produces a comprehensive error report.

提取候选需求。在活动 2（人工检查）中，开发人员根据错误报告中提供的线索提取候选需求，尤其关注可能表明 HAL 使用不当的根本原因。

To prevent improper HAL usage, the HAL functions must follow specific sequences to operate the hardware correctly. These sequences can be prescribed by dependencies between HAL functions where certain functions depend on the successful execution of others. To model this relationship, we use a binary relation representing a dependency between two HAL functions. For a pair of HAL functions f_1 and f_2, the dependency $f_1()\triangleleft f_2()$ ("f_2 depends on f_1") expresses that a call of f_1 must

在嵌入式系统的应用程序代码中，f_2 的调用之前必须有另一调用。我们将这样一对 HAL 函数称为时间依赖关系。

开发人员仔细调查报告中的原因，以确定 HAL 函数之间是否存在可能与错误相关的依赖关系。为此，开发人员查阅 HAL 文档，并在可能的情况下查看 HAL 实现，以确定是否存在依赖关系。根据此类依赖关系，开发人员提取一个时间依赖关系作为候选项。最后，该活动生成候选需求 R，供相关性检查使用。

检查提取出的候选项是否相关。相关性检查是该流程的第二部分（见图 1 中的绿色框），需要以下内容作为输入：

候选需求 R。该检查包含两个决策（在图 1 中以菱形表示），用于根据定义 1 证明 R 的相关性。对于这两个决策，我们采用一种形式化方法，具体来说是软件模型检测（例如，参见 [10]），以形式化证明 R 的相关性。为清晰起见，我们将在下文说明相关性检查中的这两个决策。

相关性检查的第一个决策。在相关性检查的第一步中，我们对故障系统的原始应用程序代码进行软件模型检查，预期检测到对 R 的违反。如果决策结果为“是”，则表示发现了违反。该违反表明 R 仍是相关性的候选项。反之，如果未检测到违反，则意味着 R 对问题报告中所述的系统故障不负有责任。

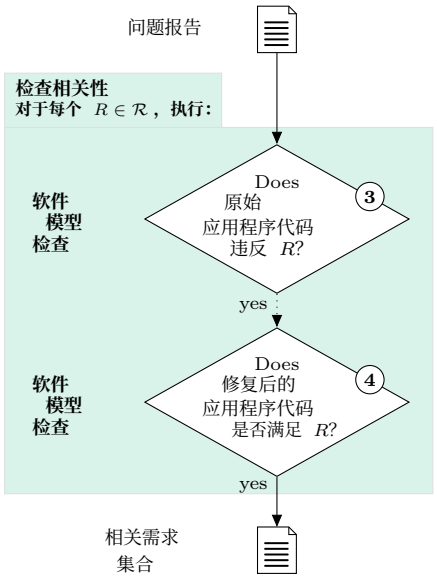


图 2. 替代方法。对于给定的 HAL 接口，候选需求有限集合 $\mathcal{R}$ 是固定的。应用代码同样有两个版本：原始版本会导致系统故障，修复后的版本则不会。

Second decision of relevance check. In the second step of the relevance check, we again apply software model checking, this time to the application code of the system in the repaired version. Hereby, we aim to confirm that R is no longer violated. Consequently, we expect that the application code in the repaired version satisfies R. If the decision returns the result ‘yes’, the relevance check witnesses that R is indeed a relevant requirement. Otherwise, if the decision returns the result ‘no’, R may be bogus.

2.3 检查一组候选需求的相关性

我们提出了一种获取相关接口需求的替代方法，其流程图如图2所示。这种相关性检查的替代方法非常适合在工业环境中使用，尤其是那些通过内部问题跟踪系统收集问题的公司。大多数问题都已经得到解决，即，相应的故障系统已得到修复，报告的系统故障也不再发生。但哪些需求与防止系统故障相关，仍不清楚。

Here, the alternative approach can be applied retrospectively to single out relevant requirements. Unlike the overall approach, this alternative does not require the manual extraction of a candidate requirement from a given issue

report. Instead, the alternative lifts the relevance check for a single candidate requirement (as shown in Figure 1) to a finite set of candidate requirements $\mathcal{R}$. The set $\mathcal{R}$ is fixed for a given HAL interface and can be inferred from the HAL documentation and, if possible, from the HAL implementation.

我们对相关性的形式化定义使我们能够利用软件模型检测，同时证明每项需求 $R \in \mathcal{R}$ 的相关性，如第 2.2 节所述。如果证明 R 相关（即两个判定结果均为“是”），我们就将其加入有限的相关需求集合。理想情况下，该集合恰好包含一项相关需求，以凸显相关性的重要意义。存在多项相关需求则表明，某个系统故障被其他故障掩盖。

3 案例研究

本节介绍一个案例研究，我们将第 2 节提出的方法应用于三份系统故障问题报告。我们的目标是评估该方法。具体而言，我们旨在回答以下研究问题：

RQ1：从系统故障问题报告中提取相关 HAL 接口需求的方法在原则上是否可行？

RQ2：对于每种系统故障，是否只有一项需求与之相关？

RQ2 背后的动机是，我们认为，当两个需求都与同一系统故障相关时，相关性作为筛选需求的一种手段，其重要性会有所减弱。为了至少能初步回答 RQ2，我们通过将需求集限定为可针对给定 HAL 推断出的时序依赖关系集来缩小问题范围。这样一来，需求数量有限，我们便可以针对每个系统的两个版本逐一检查。

3.1 嵌入式系统中的系统故障

系统故障源于一些近期在新闻组中受到关注的问题，相关讨论见多个地方，包括 [15], [16], 和 [18]。问题报告涉及嵌入式系统，并提供了系统结构描述，但完全没有明确说明接口要求。

Overview of systems. The embedded systems share a common structure, as illustrated in Figure 3, which specifically depicts the embedded system in [16]. While this schematic representation focuses on that particular example, it is important to note that the other two systems are similarly structured.

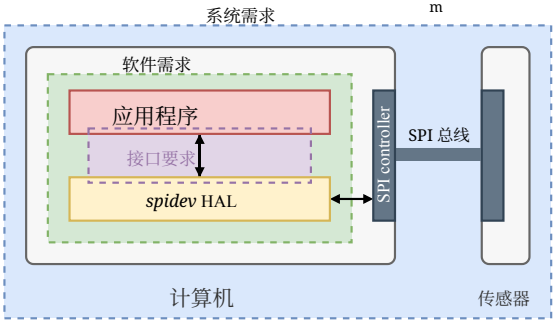


图 3. [16] 中问题报告所述的嵌入式系统架构。

SPI bus and spidev *HAL.* The embedded systems use the Serial Peripheral Interface (SPI) [12] as a communication bus to connect the system's hardware, in particular a computer or microcontroller, to a peripheral device. The peripheral device is specifically an actuator like a display controller in [15], a sensor in [16], or an unspecified device in [18]. The SPI bus is operated in a master/slave configuration where the integrated SPI controller of the computer or microcontroller serves as master to control the bus communication. Conversely, the SPI controller of the peripheral device serves as slave. The computer or microcontroller runs an application program that uses the *spidev* HAL[5] to operate the SPI bus via the master SPI controller (see Figure 3). The *spidev* HAL comprises several software layers to support SPI controllers from various manufacturers and uses POSIX system calls to provide a HAL interface. Proper operation of the SPI controller by the embedded application requires the use of HAL functions such as `open` to initialize the SPI controller. Configuration and data transfers are then performed through HAL functions like `ioctl`, `read`, and `write`. The `ioctl` function uses specific SPI constants[6] to set parameters such as mode, speed, and bit order for SPI data transfers.

Interface requirements. Interface requirements between the application program and the *spidev* HAL (see dashed violate box in Figure 3) solely depend on the HAL. These requirements have to be satisfied during the runtime of the system. Otherwise runtime errors can occur that may lead to serious system failures. Currently, such interface requirements do not exist for the *spidev* HAL. To address RQ1, we extract a candidate requirement for each of the three system failures according to our overall approach from Section 2.2. To address RQ2, we systematically infer a set of requirements for the *spidev* HAL in order to apply our alternative approach from Section 2.3 retrospectively on the three systems.

Replicated systems and corresponding failures. While the reports offer some insights into the issues and the used embedded systems, they do not provide

[5] *spidev* HAL 及其用户空间 API 是 Linux 内核代码库的一部分，可在 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/spi/spidev?h=v5.3[6] 中获取。为提高可读性，我们省略了 *spidev* HAL 中所有 SPI 常量的前缀 SPI_IOC_，并将 SPI 常量 MESSAGE 缩写为 MSG（另请参见表 1）。

the complete application code or sufficient details of the hardware setup to clone each system. We rebuild the embedded systems with the hardware to replicate system failures corresponding to those reported in the issue reports. With the rebuild, we make sure that an error in each system lead to a system failure from the issue reports, i.e., that it is a real error (as opposed to a *theoretical* error which cannot appear in a practical environment). We create three applications that correspond to the applications from the issue reports: control an actuator (cf. [15]), read sensor measurements (cf. [16]), and test communication (cf. [18]). In the following, we describe the three embedded systems and their observations that align with the ones from the issue reports.

嵌入式系统 1：I/O 扩展器

系统描述：系统"I/O 扩展器"旨在控制 Microchip MCP23S17 输入/输出扩展器，该扩展器提供 16 个可配置的输入/输出引脚。此系统配置包括计算机上的主 SPI 控制器和 MCP23S17 中的从 SPI 控制器。应用程序使用 *spidev* HAL 与 MCP23S17 通信，并按指定顺序执行初始化和控制命令。

故障观察：程序执行后，观察到系统故障，MCP23S17 的 A0 引脚未能按预期激活。此故障与 [15], 中报告的问题相对应：执行器（显示控制器）无法得到可靠控制。

嵌入式系统 2：加速度计

系统描述：应用程序"加速度计"向 Analog Devices ADXL345 加速度计发送控制命令，该加速度计可测量三个轴向的加速度。该系统采用类似的主从配置，其中计算机充当 SPI 主设备，ADXL345 充当 SPI 从设备。应用程序通过 *spidev* HAL 传输命令来获取传感器数据，执行方式与原始系统中的方式类似。

故障观察：运行此应用程序时，我们观察到一个故障，即无法从 ADXL345 接收到加速度数据。这一观察结果与 [16]，的问题报告中描述的系统故障一致，即传感器无法提供测量数据。

嵌入式系统 3：*spidev*-测试

System description: The application program '*spidev*-Test' is generally used by developers to test the functionality of newly developed SPI components. In this case, we applied this application to test the communication with the ADXL345 accelerometer. The system uses the same SPI mas-

ter/slave configuration, with the computer configured to check the connectivity and data flow through the ADXL345.

故障观察：运行此应用程序时，通信测试未能返回有效的设备 ID，导致发生故障。这与 [18]，中报告的问题一致：与外围设备通信的健全性检查未能通过。

3.2 从系统故障中提取并检查候选需求

我们将第 2.2 节中的整体方法应用于这三份问题报告，以回答 RQ1。我们从复现的系统中获取原始应用程序代码，这些系统均出现了故障。

Inspect issue reports. The three issue reports describe system failures observed during operation and related to the SPI bus. While the application code is syntactically correct and can be compiled and run, the issue reports indicate that the errors are not trivial. Manual inspection of the system failures reveals that each application code properly initializes the SPI controller using the HAL function `open`, but omits essential configuration calls, causing the controller to operate with default settings. This misconfiguration leads to communication failures, as there is no universal agreement on a default configuration for SPI controllers. In the 'I/O Expander' code, the missing `ioctl(WR_MAX_SPEED_HZ)` call causes clock speed mismatches. In the 'Accelerometer' code, the absent `ioctl(WR_MODE32)` call results in polarity and phase mismatches. In the '*spidev*-Test' code, the missing `ioctl(WR_BITS_PER_WORD)` call causes word size mismatches in data transfers between the master and slave SPI controller.

提取候选项并检查相关性。 按照整体方法，我们根据检查所得的洞见，为每个问题提取一项候选需求。具体而言，我们提取了时间依赖关系 δ_{26} （针对'I/O Expander'）、δ_{17} （针对'Accelerometer'）和 δ_{23} （针对'*spidev*-Test'）作为候选项（详见表 1）。我们对这三项候选项进行相关性检查，并使用 0.3.0 版的验证工具 Ultimate Automizer [9] 执行软件模型检查。[7]

时间依赖关系的有效性。 相关性检查的软件模型检测会验证从使用 HAL 中 *spidev* 函数的 C 程序中提取的需求是否有效。我们通过在程序的注释版本中添加断言来编码这种有效性。

The annotation, shown schematically in Listing 1, adds auxiliary statements of two kinds: update statements that only modify auxiliary variables, and assert

[7] Ultimate Automizer 是开源程序分析框架 Ultimate 的一部分，可通过 https://www.ultimate-pa.org 获取。

```
1  /*@幽灵 int状态_d1 = 0; */
2
3  int open(const char*路径, int oflag, ...)
4  {
5      整型 ret = ...;
6
7      /*@ 幽灵 状态_d1 = 1; */
8      return 返回;
9  }
10
11 ssize_t 读取(int 文件描述符, void*缓冲区, size_t 字节数)
12 {
13     /*@断言(state_d1 == 1); */
14
15     return ...;
16 }
```

列表 1. 通过应用程序注释中的断言有效性，编码应用程序中的时序依赖关系。对于表 1 中的时序依赖关系 δ_1 : open()◁read()，注释包括向 *spidev*函数 open 和 read 中添加幽灵代码（以绿色显示）。此处的代码片段引用 HAL 中 *spidev*函数的原始实现。除所示位置外，该注释不会修改应用程序。

statements. For each temporal dependency of the form $\delta : f_1() \triangleleft f_2()$, we insert an update statement in the code for the implementation of function f_1 and an assert statement in the function f_2, in addition to the declaration of an auxiliary variable for the temporal dependency δ. In each execution, the value of the auxiliary variable (the *ghost state*) flags whether the call of the function f_1 has (or has not yet) taken place. We use the ANSI/ISO C Specification Language (ACSL), an annotation language for C, supported by a variety of verification tools [4]. The encoding of the validity is independent of the particular application code and solely depends on the HAL.

Experimental results. In the first step of the relevance check, we apply Ultimate Automizer on each of the three application programs to verify the validity of the corresponding candidate temporal dependency. The verification results reveal that one assert statement is violated in each faulty application program. We inspect each violation (using the failure path from the verification result) to witness that the violated assert statements are indeed part of the program annotation. Based on the program annotation, we do a reverse-lookup to locate the temporal dependencies, to which these violated assert statements belong. The verification results witness that all three candidate temporal dependencies are violated in each corresponding application program.

In the second step of the relevance check, we again apply Ultimate Automizer, this time to each of the application programs in the repaired version. We obtain the repaired version by manual repair of the error in each faulty application program. The failure path from each verification result can guide us to localize the errors, so that the repaired application programs no longer cause the observed system failures. From the check, we obtain the verification results, which witness that all assertions in each of the three application programs are

satisfied. Thus, all three candidate temporal dependencies are satisfied and indeed relevant. Our overall approach to obtain a relevant interface requirement succeeds in the three cases and is feasible in principle (**RQ1**).

3.3 检查一组*spidev* HAL 接口要求的相关性

我们将第 2.3 节提出的替代方法追溯应用于这三份问题报告，以回答 RQ2。我们从复现系统中获取原始的故障版本应用程序代码，并从回答 RQ1 的实验中获取修复后的正确版本代码。

推断一组候选项。 该方法需要一组接口需求，我们从 *spidev* HAL 中推断时间依赖关系。我们采用系统化的方法推断时间依赖关系，逐一考察所有 HAL 函数对。对于每一对函数，我们查阅 HAL 文档，并在可能的情况下检查 HAL 实现，以确定是否存在依赖关系。经确认的依赖关系将作为时间依赖关系添加到集合中。最终，我们得到表 1 所示的时间依赖关系集合。

表 1。 从*spidev* HAL 推断出的候选时间依赖关系。

δ_1 : open()	◁ 读取()
δ_2 : open()	◁ write()
δ_3 : open()	◁ ioctl(MSG)
δ_4 : open()	◁ close()
δ_5 : open()	◁ ioctl(RD_模式)
δ_6 : open()	◁ ioctl(WR_模式)
δ_7 : open()	◁ ioctl(RD_MODE32)
δ_8 : open()	◁ ioctl(WR_MODE32)
δ_9 : open()	◁ ioctl(RD_LSB_优先)
δ_{10} : open()	◁ ioctl(WR_LSB_优先}
δ_{11} : open()	◁ ioctl(RD_BITS_PER_WORD)
δ_{12} : open()	◁ ioctl(WR_BITS_PER_WORD)
δ_{13} : open()	◁ ioctl(RD_MAX_SPEED_HZ)
δ_{14} : open()	◁ ioctl(WR_MAX_SPEED_HZ)
δ_{15} : ioctl(WR_MODE32)	◁ read()
δ_{16} : ioctl(WR_MODE32)	◁ write()
δ_{17} : ioctl(WR_MODE32)	◁ ioctl(MSG)
δ_{18} : ioctl(WR_LSB_FIRST)	◁ read()
δ_{19} : ioctl(WR_LSB_FIRST)	◁ write()
δ_{20} : ioctl(WR_LSB_FIRST)	◁ ioctl(MSG)
δ_{21} : ioctl(WR_BITS_PER_WORD)	◁read()
δ_{22} : ioctl(WR_BITS_PER_WORD)	◁write()
δ_{23} : ioctl(WR_BITS_PER_WORD)	◁ioctl(MSG)
δ_{24} : ioctl(WR_MAX_SPEED_HZ)	◁read()
δ_{25} : ioctl(WR_MAX_SPEED_HZ)	◁写入()
δ_{26} : ioctl(WR_MAX_SPEED_HZ)	◁ioctl(MSG)

检查候选项的相关性。对于三个问题报告中的每一个，我们对表 1 中的时间依赖关系集合应用相关性检查。我们再次使用 Ultimate Automizer，这次同时验证该集合中所有时间依赖关系的有效性。这要求我们将所有时间依赖关系的有效性编码为每个应用程序两个版本中的注释。

实验结果。 表 2 展示了相关性检查的实验结果。在总计 26 个候选时序依赖项中，有三个（即 δ_{17}、δ_{23} 和 δ_{26}）被正式证明具有相关性。这三项需求分别导致了三个问题报告中记录的系统故障。对于每个系统故障，我们都恰好找到了一个相关的接口需求（**RQ2**）。这一发现凸显了相关性在本案例研究中的重要意义。

3.4 讨论

我们的结果表明，使用整体方法和替代相关性检查，我们得到的相关接口要求相同。此结论-

表 2. 针对表 1 中 *spidev* HAL 推断出的时间依赖关系进行相关性检查所得的实验结果。“结果”下列出的集合，正是该检查判定为违反的接口要求集合。也就是说，其补集包含该检查判定为正确的要求。

应用程序 name	版本	size [LOC]	结果 错误	验证 time [s]	mem. [MB]
I/O 扩展器	故障	172	$\{\delta_{26}\}$	6.24	264
	已修复	180	$\emptyset$	8.10	263
加速度计	故障的	284	$\{\delta_{17}\}$	10.43	294
	已修复	292	$\emptyset$	12.36	345
spidev-测试	故障的	764	$\{\delta_{23}\}$	990.35	1129
	已修复	768	$\emptyset$	925.58	1335

sistency provides a strong indication that our approach is feasible in principle. In our case study, we did not encounter the scenario where one system failure masks other ones, which indicates that several requirements might be relevant. Masking failures could occur if a developer forgets to insert not only the HAL function calls for the configuration of the master SPI controller but also the call to initialize the SPI controller, leading to several observable system failures that may mask one another. While this case could potentially arise, our alternative relevance check from Section 2.3 can obtain several relevant requirements even in the presence of masking failures.

表 2 的实验结果还表明，对时间依赖关系有效性本身的验证并不是本工作中的挑战（参见表 2 的最后两列）。即使是更复杂的应用程序“*spidev*- 测试”，其修复版本包含 766 行代码（包括注释），也能在 17 分钟内完成验证，且内存消耗从未超过 1.4 GB。[8]因此，评估其他软件模型检查工具或形式化验证方法的效率似乎没有意义。

Several factors could affect the validity of our findings from the experimental evaluation. First, the representativeness of the observed system failures is crucial, as it influences the relevance of the interface requirements. The lack of case studies on system failures and the often imprecise descriptions limit our understanding of potential issues. Due to the unavailability of the original systems, we replicated comparable embedded systems to reproduce similar system failures. Consequently, the findings from these replicated systems cannot be directly applied to the original systems; instead, they provide evidence that the independently identified interface requirements may be relevant for failures resembling those in the issue reports. In industrial practice, the availability of the actual systems would reduce the importance of system replication to obtain application code. More research is needed to confirm the generality of our results.

[8]所有使用 Ultimate Automizer 的验证运行均已完成，并在一台配备 2.6 GHz 四核 CPU 和 8 GB 内存的普通台式计算机上进行测量，该计算机运行 Arch Linux，Linux 内核版本为 6.12.47。

4 相关工作

本节介绍与本文主题相关的工作。我们重点关注需求优先级排序、接口需求以及需求推导方法等领域。

Requirement prioritization. Recent surveys [6,13,14,19], provide an overview of requirements prioritization techniques, with guidelines to choose a suitable prioritization technique [7]. While most techniques rely on manual effort, our formal notion of relevance allows us to leverage a formal method to automatically check relevance. A related review study [17] focuses on the most important aspect of single out requirements, the prioritization criteria. However, most criteria focus on importance, cost-benefit, and stakeholder satisfaction and leave system reliability and safety out of consideration. In addition, many criteria are ambiguous and not clearly understandable leading to subjective and inconsistent prioritization decisions. In contrast, this work introduces a formal relevance criterion specifically designed to address system reliability and safety. Its formal nature ensures unambiguity and clarity, which helps to single out important interface requirements that are essential for preventing system failures.

时序依赖关系。我们用于定义这组 26 项候选需求的时序依赖关系类别，对应于 [8] 层次结构中的优先顺序需求模式。该类别的限制足够严格，能够系统地筛选候选需求；同时又具有足够的表达能力，能够涵盖相关需求，正如我们的案例研究所示。

Deriving requirements. Previous work [5], building upon the work in [1] and [11], proposes different algorithms to automatically derive interface requirements from an existing library implementation for its safe and permissive use. These approaches require the presence of explicit safety requirements (e.g., invariants) within the library implementation, which serve as a basis for the automatic inference of a possibly comprehensive set of interface requirements. In contrast, our approach does not depend on safety requirements defined a priori. Instead, the primary objective of our approach is to derive a concise set of HAL interface requirements by prioritizing candidate requirements to single out the relevant ones that are essential for preventing system failures. A similar line of research is the specification mining based on statistical or stochastic methods where interface requirements are automatically derived from a set of error-free application programs that properly interact with the interface, e.g., see [2,3]. The mined specifications are utilized to detect so-called API (mis)uses. The existing approaches to automatically mine interface requirements may be contrasted with our approach to derive relevant HAL interface requirements: the former relies on programs that properly use a given interface, whereas the latter relies in the existence of programs that do not. Exploring how one can exploit this apparent complementariness remains an open area for future research.

5 结论

在本研究中，我们提出了一种基于系统故障的需求优先级排序相关性准则的形式化概念。该准则使我们能够找出对于防止系统故障至关重要的需求。其形式化定义确保了对该准则的清晰理解，使优先级排序保持一致、明确，且不受任何主观评估的影响。

我们以 HAL 接口需求为例，介绍了一种通过在定义明确的流程中形式化证明需求相关性，从而确定需求优先级的方法。HAL 接口需求规定了嵌入式系统应用程序中正确使用 HAL 接口的方式。我们获取相关 HAL 接口需求的方法，首先是分析提及可观测系统故障的问题报告。随后，我们根据这些报告推断出正确使用 HAL 接口的候选需求。接着，我们借助软件模型检测，对每项候选需求的相关性进行形式化证明。相关性检查是一个至关重要的流程步骤，因为它证明了某项接口需求确实是导致所观察到的系统故障的原因。

我们针对三个问题报告示例开展了案例研究，以评估我们的方法。这些报告涉及通过*spidev* HAL 使用 SPI 总线的嵌入式应用程序。实验结果证明了我们方法的实际可行性。对于三份问题报告中提到的每一种系统故障，我们都能找出一项相关的接口需求。每一项相关接口需求对于防止可观察到的系统故障都至关重要。嵌入式软件开发人员应重点关注此类相关需求，以防止特定类型的系统故障，例如问题报告中提到的故障。

致谢。 本研究的部分经费由德国研究联合会（DFG）资助，项目编号为 503812980。

参考资料

1. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. SIGPLAN Notices **40**(1), 98–109 (2005). https://doi.org/10.1145/1047659.1040314
2. Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N., Mezini, M.: MUBench: a benchmark for API-misuse detectors. In: Proceedings of the International Conference on Mining Software Repositories. MSR, ACM (2016). https://doi.org/10.1145/2901739.2903506
3. Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M.: A systematic evaluation of static API-misuse detectors. Transactions on Software Engineering **45**(12), 1170–1188 (2019). https://doi.org/10.1109/tse.2018.2827384
4. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Frama-C (2024), https://frama-c.com/download/acsl-1.20.pdf, version 1.20
5. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 4–19. CAV, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_4

6. Bukhsh, F.A., Bukhsh, Z.A., Daneva, M.: A systematic literature review on requirement prioritization techniques and their empirical evaluation. Computer Standards & Interfaces **69**, 103389 (Mar 2020). https://doi.org/10.1016/j.csi.2019.103389
7. Castillo-Valdivieso, P.A., Alhenawi, E., Awawdeh, S., Abu Khurma, R., García-Arenas, M., Hudaib, A.: Choosing a suitable requirement prioritization method: A survey. Journal of Computer Science & Technology **24**(1), e04 (Apr 2024). https://doi.org/10.24215/16666038.24.e04
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the International Conference on Software Engineering. pp. 411–420. ICSE, ACM (1999). https://doi.org/10.1145/302405.302672
9. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 447–451. TACAS, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
10. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Computer Aided Verification. pp. 36–52. CAV, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
11. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. Software Engineering Notes **30**(5), 31–40 (2005). https://doi.org/10.1145/1095430.1081713
12. Hill, S.C., Jelemensky, J., Heene, M.R., Groves, S.E., DeBrito, D.N.: Queued serial peripheral interface for use in a data processing system. Patent US-4958277 (Sep 1990), https://image-ppubs.uspto.gov/dirsearch-public/print/downloadPdf/4958277
13. Hujainah, F., Bakar, R.B.A., Abdulgabber, M.A., Zamli, K.Z.: Software requirements prioritisation: A systematic literature review on significance, stakeholders, techniques and challenges. IEEE Access **6**, 71497–71523 (2018). https://doi.org/10.1109/access.2018.2881755
14. Pacheco, C., García, I., Reyes, M.: Requirements elicitation techniques: a systematic literature review based on the maturity of the techniques. IET Software **12**(4), 365–378 (Aug 2018). https://doi.org/10.1049/iet-sen.2017.0144
15. SPI connection between RPi and ATmega32. Raspberry Pi Forum (2016), https://forums.raspberrypi.com/viewtopic.php?p=881605
16. Reading values from MCP3002 not working [...]. Raspberry Pi Forum (2019), https://forums.raspberrypi.com/viewtopic.php?t=230569
17. Riegel, N., Doerr, J.: A systematic literature review of requirements prioritization criteria. In: Requirements Engineering: Foundation for Software Quality. pp. 300–317. REFSQ, Springer (2015). https://doi.org/10.1007/978-3-319-16101-3_22
18. *spidev* sanity check not working. Stack Exchange (2019), https://raspberrypi.stackexchange.com/questions/102286/spidev-sanity-check-not-working
19. Yaseen, M., Mehmood, W., Hameed, F., Nauman, M.: Scalability and limitations of existing software requirements prioritization techniques: A systematic literature review. Journal of Software: Evolution and Process **37**(8) (Aug 2025). https://doi.org/10.1002/smr.70039