arXiv:2301.00835v1 [cs.SE] 2 Jan 2023

用于 Simulink 模型的定时基于模型的变异算子

Jian Chen* [1] **Manar H. Alalfi** [2] **Thomas R. Dean** [3]

[1] 电气与计算机工程系，女王大学，金斯顿，安大略省，加拿大 电子邮箱：*jian.chen@queensu.ca*[2] 计算机科学系，瑞尔森大学，多伦多，安大略省，加拿大 电子邮箱：*manar.alalfi@cs.ryerson.ca*[3] 电气与计算机工程系，女王大学，金斯顿，安大略省，加拿大 电子邮箱：*tom.dean@queensu.ca*

摘要

基于模型的变异分析是一个新兴研究领域，实时系统测试可从使用模型变异体中获益。基于模型的变异测试（MBMT）是基于模型测试的一个分支。它利用变异算子生成模型的故障版本，以评估和改进测试用例。变异测试是确保软件正确性的有效方法，已应用于多个应用领域。Simulink 是实时系统的重要建模语言。本文引入 Simulink 模型变异分析，以改进模型在环（MIL）测试。我们基于 AUTOSAR 提出一组 Simulink 变异算子，用于反映 Simulink 模型映射到操作系统任务时的时间正确性。我们实现了一个变异框架，可为隐式时钟 Simulink 模型生成变异体。最后，我们展示该框架如何通过生成变异体，在仿真中揭示任务干扰问题。我们的工作将 Simulink 模型与定时系统相结合，以更好地支持变异测试自动化。

关键词：变异测试、基于模型的测试、基于模型的变异测试、变异算子、Simulink、实时系统、调度、AUTOSAR

1. 引言

如今，汽车配备了许多以前不存在的先进技术，例如自动紧急制动（AEB）、自适应巡航控制（ACC）、车道偏离预警/车道保持以及自动驾驶。所有这些功能都依赖软件来实现复杂的控制算法。通常，此类软件是在定时系统的背景下开发的，在这种系统中，系统的正确性不仅取决于软件所实现功能的正确性，还取决于系统是否满足时间约束。许多因素都可能影响系统在目标平台上运行时的执行时间。任务干扰等问题可能导致任务执行延迟。在此类安全关键应用中，软件质量发挥着至关重要的作用。

基于模型的测试（MBT）是一种很有前景的技术，可用于对定时系

统进行自动化测试。模型表示软件的行为，通常根据实时规范抽象而来。然而，一些建模环境仅在硬件在环（HIL）仿真测试中支持这一特性，而不支持模型在环（MIL）测试。例如，Matlab/Simulink (ML/SL) 仿真假定各个模块的行为几乎在零执行时间内完成，而实际执行需要有限的执行时间，这可能导致故障。ML/SL 模型基于同步反应式（SR）模型 [23]，该模型可能假定任务执行时间为零。如果模型在模型在环阶段没有进行显式的实时执行，其中的错误可能不会显现。通常，Simulink模型在模型在环测试中可以得到良好的仿真，但在实时环境中可能存在错误。

因此，基于模型的测试（MBT）需要扩展以适应实时上下文，这包括通过定时形式化方法对系统进行建模，并检查实现是否符合其规范。传统上，这通过一致性检查[35]来完成。近年来，已有多种工具被提出，用于在模型在环（MIL）中模拟 ML/SL 模型的实时执行效果，例如 TrueTime 21、TRES [12],时序感知模块 27 和 SimSched [9]。SimSched 通过模型转换将调度集成到模型中，以便在仿真期间验证实时上下文。为评估 SimSched，我们采用变异测试，并借助变异分析来辅助评估 SimSched 工具。

本文提出一组基于定时任务模型的变异算子。该任务模型基于汽车开放系统架构（AUTOSAR），能够反映 Simulink 模型在实时上下文中映射到实时操作系统（RTOS）任务时的时间正确性。

本文结构如下：第 2 节介绍背景信息。第 3 节介绍针对 Simulink 模型提出的定时变异算子集合。第 4 节说明定时变异算子的用法。第 5 节介绍验证实验及结果。第 6 节总结基于模型的测试（MBT）相关研究。最后，第 7 节总结本文结论并展望未来工作。

2. 背景

本节概述解释本文工作所需的背景知识。首先，我们简要介绍变异测试、Simulink 和 AUTOSAR；然后介绍我们的定时任务模型。

2.1. 变异测试

变异测试于 20 世纪 70 年代被提出 [17,13,22]，并被证明是发现软件故障的一种有效方法 [32]。这是一种基于故障的软件测试技术，数十年来一直受到广泛研究和应用。它为软件测试提供了各种方法、工具和可靠的研究成果。变异测试旨在找出有效的测试用例，并发现程序中的实际错误。

基于模型的变异测试（MBMT）兼具基于模型的测试和变异测试的优势，已广泛应用于多种类型的模型，例如特征模型 20、基于状态图的模型 41，1、定时自动机 3，2 以及 Simulink [8,26,38]。然而，在实时系统开发中，逻辑正确性和时间正确性对于系统功能的正确实现都至关重要。时间正确性取决于每个任务的时序假设。定时自动机（TA）[4] 是一种用于对实时系统进行建模和验证的常见形式化方法，可用于检验设计是否满足时间要求。Aich- ernig *etal*.[3] 提出了一种针对定时自动机的基于模型的变异测试技术，该技术应用于输入/输出 定时自动机（TAIO）模型。Nilsson*et al.* [28] 在 TA 形式化方法中加入了任务模型扩展，其变异算子侧重于及时性。Simulink*广泛用于汽车行业的软件模型驱动开发。从属性角度来看，针对 Simulink 模型提出的大多数变异算子属于运行时或设计时变异，例如信号修改、算术替换或模块更改 [18,38,36,43]。其中一些已提出的变异测试旨在为 Simulink 模型生成测试用例 [8,19]。然而，目前还没有针对 Simulink 的、具有显式时钟模型的变异算子。

* https://www.mathworks.com/products/simulink.html

2.2. Simulink

Simulink 是用于对动力系统建模的最流行建模语言之一，而 MATLAB 提供了用于执行系统仿真的图形化编程环境。Simulink 模型由图形块和连线组成，输入端口与输出端口之间通过信号连接。Simulink 仿真引擎在每次仿真执行前，根据模块之间的数据依赖关系确定模块的执行顺序。Simulink 定义了两种类型的模块，即直接馈通模块和非直接馈通模块，以确保仿真中的正确数据依赖关系。Simulink 使用以下两条基本规则[25] 确定排序后的执行顺序：模块必须在其驱动的所有直接馈通端口所属模块之前执行；不包含直接馈通输入的模块可以按任意顺序执行，只要它们先于其驱动的直接馈通输入所属模块即可。所有模块均按排序后的顺序调度，并按照顺序依次执行。Simulink 引擎维护一个虚拟时钟，以便在每个虚拟时间执行每个排序后的模块。

Simulink Coder[†]支持代码生成，并提供了一个框架，可在实时环境中执行生成的代码。Simulink Coder 可以为周期任务生成代码，既可以采用单任务实现，也可以采用多任务实现。单任务实现中，生成的代码由单线程中的简单调度器调用，不会发生抢占，因此可以在仿真过程中保持语义不变。对于多任务实现，生成的代码由多线程实时操作系统环境中的速率单调（RM） [24] 调度器调用，其中每个任务都分配有一个优先级，并且任务之间会发生抢占。由于抢占和调度，多速率系统中的实现语义可能与模型语义发生冲突。

2.3. AUTOSAR

AUTOSAR 是一项开放的行业标准，旨在满足未来汽车开发的需求。AUTOSAR de-

† https://www.mathworks.com/products/simulink-coder.html

‡ https://www.mathworks.com/products/simulink.html

定义了三个主要层：应用层、运行时环境（RTE）和基础软件（BSW）层 [40]。应用层中的功能由软件组件实现，这些组件封装了部分或全部汽车电子功能，如图 1 所示。组件通过虚拟功能总线（VFB）进行通信，它抽象了 AUTOSAR 的所有通信机制。工程师利用 VFB 抽象软件组件的通信细节。一组可运行实体表示软件组件的内部行为，而可运行实体是可单独调度的最小可执行代码单元，可由定时器或事件触发。最后，可运行实体需要映射到目标平台的一组任务中，并且映射必须保留顺序关系和因果依赖关系。自 R2006a 版本起，Simulink 已支持符合 AUTOSAR 标准的代码生成[‡]。所有 AUTOSAR 概念都可以通过 Simulink 模块表示，现有的 Simulink 模块也可以轻松应用于 AUTOSAR 开发流程。表 1 [39] 展示了部分 AUTOSAR 概念与 Simulink 概念之间的映射关系。

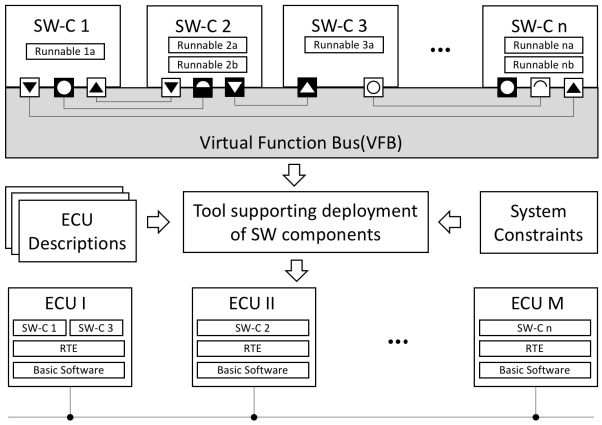


图 1. AUTOSAR 组件、接口和可运行实体。（改编自 [5]）

表 1. ML/SL 与 AUTOSAR 概念映射示例。

ML/SL	AUTOSAR
子系统	原子软件组件
函数调用子系统	可运行实体
函数调用	RTE 事件

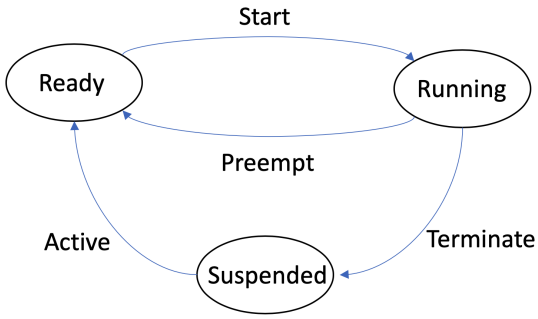


图2. 任务模型的任务状态和状态转换。

图 2 展示了基于 OSEK 基本任务状态模型的任务模型状态图及状态转换图。任务模型包含三种状态：挂起、就绪和运行；以及四种状态转换：激活、启动、抢占和终止。这些状态转换表示激活、启动、抢占或终止任务的操作。

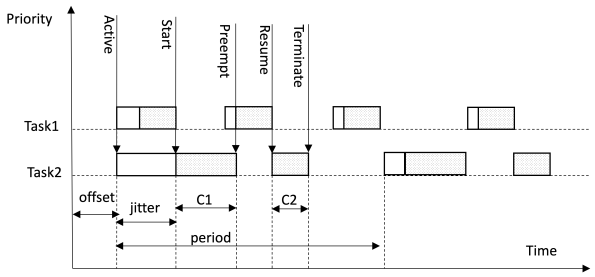


图3. 甘特图中显示的任务时间参数（均与Task2相关）。

图3展示了任务模型的时间参数，不同的时间参数可能会改变系统内应用程序的实时行为。

2.4. 任务模型

在汽车软件中，Simulink 模型通常根据实时规范绘制，并实现为一组在实时操作系统上运行的任务。为了在模型在环阶段更好地测试此类软件，基于模型的测试需要扩展到实时环境，这需要使用定时形式化方法对符合实时需求的被测系统进行建模。我们定义了一种任务模型，用于描述 Simulink 环境中任务的时序属性，并将应用建模为一组周期性任务。

任务模型 T 表示为一个元组$\{\phi,\rho,c\ \gamma,$ $prect,precr,prio,jitter\}$，其中 ϕ是任务的偏移量，ρ 是任务的周期，c 是任务的最坏情况执行时间（WCET），γ是属于该任务的可运行实体列表，*prect* 是任务的优先约束，*precr* 是任务内可运行实体之间的优先约束，*prio*是与任务关联的优先级，*jitter* 是任务偏离周期性释放时间的程度。每个任务都有隐式截止时间，这意味着任务的截止时间等于 ρ。偏移量 ϕ 指的是周期任务第一个实例到达与其释放时间之间的延迟。WCET 是各可运行实体执行时间的总和。优先约束 *prect* 是指定任务执行顺序的任务列表，而 *precr* 是任务内可运行实体的列表。

3. Simulink模型的变异算子

本节介绍一种变异分析方法，用于在仿真过程中验证实时上下文。变异算子是变异测试的关键要素。基于模型的变异测试是一种向模型中注入故障，以检查测试通过还是失败，从而验证软件的方法。注入的故障即为变异算子。在将变异算子应用于模型之前，我们需要先识别这些算子，也就是

了解哪些类型的错误会导致失败。我们提出了以下任务相关变异算子。

3.1. 偏移变异算子

任务释放偏移是影响任务干扰计算结果的因素之一。为了在分析中考虑偏移量，我们引入了一种偏移量变异算子。对于已知偏移量 ϕ，任务现在可以在其周期开始后经过 ϕ 个时间单位执行。任务的执行时间保持不变，即在下一周期开始前执行 c 个时间单位。

3.1.1. mITO

该算子会在当前偏移量上增加 δ 个时间单位。对于给定任务 $\tau_i\{\phi_i, \rho_i, c_i, \gamma_i, prect_i, precr_i, prio_i, 抖动_i\} \in T$，此变异算子会将偏移量 ϕ_i 更改为$\phi_i + \delta$。

3.1.2. mDTO

此算子会从当前偏移量中减去 δ 时间。对于给定的任务 $\tau_i\{\phi_i, \rho_i, c_i, \gamma_i, prect_i, precr_i, prio_i, jitter_i\} \in T$，此变异算子将偏移量 ϕ_i 更改为$\phi_i - \delta$。

3.2. 周期变异算子

实时操作系统通常采用抢占式多任务调度算法来确定任务的执行顺序，其中最常用的算法是固定优先级调度（FPS）。该算法为每个任务分配一个静态优先级。实时操作系统调度器会从就绪任务队列中执行优先级最高的任务。Simulink Coder 支持速率单调调度器，其中任务的优先级与其周期相关：任务的周期越短，优先级越高。此外，在执行过程中，低优先级任务可能会被更高优先级的任务抢占。

3.2.1. mITPER

该算子会延长任务的周期，使任务以较慢的速率运行。对于给定任务 $\tau_i\{\phi_i, \rho_i, c_i, \gamma_i, prect_i, precr_i, prio_i, 抖动_i\} \in T$，此变异算子将任务$i$的周期更改为 $\rho_i + \delta$。

3.2.2. mDTPER

该算子会缩短任务的周期，使任务以更快的速率运行。对于给定任务$\tau_i\{\phi_i, \rho_i, c_i, \gamma_i, prect_i, precr_i, prio_i, 抖动_i\} \in T$，该变异算子会将任务$i$的周期改为 $\rho_i - \delta$。

3.3. 执行时间变异算子

实时系统的正确性一方面取决于逻辑程序的计算结果，另一方面也与结果生成的时间严格相关。因此，在嵌入式系统的设计和验证过程中，执行时间分析至关重要。为此，我们提出了执行时间算子，可在可运行实体层面调整每个任务的执行时间，以模拟不同处理器速度下的运行时执行情况。任务执行时间延长可能导致低频率任务阻塞高频率任务，使其错过截止时间。

3.3.1. mITET

该算子会为每个可运行实体当前的执行时间增加 δ 时间，从而延长总执行时间。对于给定的任务$\tau_i\{\phi_i, \rho_i, c_i, \gamma_i, prect_i, precr_i, prio_i, jitter_i\} \in T$，该变异算子将执行时间从$c_i$ 更改为$c_i + \delta$。

3.3.2. mDTET

该算子从每个可运行实体的当前执行时间中减去 δ 时间，从而缩短总执行时间。对于给定的任务$\tau_i\{\phi_i, \rho_i, c_i, \gamma_i, 优先级_i, 优先级_i, 优先级_i, 抖动_i\} \in T$，该变异算子将执行时间$c_i$ 更改为$c_i - \delta$。

3.4. 执行优先级变异算子

实时操作系统调度器根据任务的优先级选择要执行的任务。不过，对于优先级相同的任务，生成顺序决定其执行顺序。先生成的任务会先被实现，并首先获得中央处理器运行。因此，在实现中，某对任务之间可能存在设计阶段中不存在的优先关系，或者实现中已有的优先关系可能丢失。不正确的优先关系可能导致任务执行顺序错误。因此，我们提出了优先关系变异算子，用于指定一对任务或可运行实体之间的优先关系。该算子通过为一组任务或可运行实体指定特定的执行顺序来创建变异体，以体现优先关系。

3.4.1. mATPREC

对于给定任务 $\tau_i\{\phi_i,\rho_i,c_i,\gamma_i,prect_i,precr_i,prio_i,jitter_i\}\in T$，对于每个任务 $\tau_j\in T(j\neq i)$，如果 $\tau_j\notin prect_i$，此变异算子会向 $prect_i$ 添加 τ_j。

3.4.2. mRTPREC

对于给定任务 $\tau_i\{\phi_i,\rho_i,c_i,\gamma_i,prect_i,precr_i,prio_i,jitter_i\}\in T$，对于每个任务 $\tau_j\in T(j\neq i)$，如果 $\tau_j\in prect_i$，此变异算子会从 $prect_i$ 中移除 τ_j。

3.4.3. mARPREC

对于给定任务 $\tau_i\{\phi_i,\rho_i,c_i,\ \gamma_i,\ prect_i,\ precr_i,\ prio_i,$ 抖动$_i\}\in T$，对于每个可运行实体 $\gamma_{im}\in\tau_i$，如果 $\gamma_{im}\notin precr_i$，此变异算子会将 γ_{im} 添加到 $precr_i$。

3.4.4. mRRPREC

对于给定任务 $\tau_i\{\phi_i,\rho_i,c_i,\ \gamma_i,\ prect_i,\ precr_i,\ prio_i,$ 抖动$_i\}\in T$，对于每个可运行实体 $\gamma_{im}\in\tau_i$，如果 $\gamma_{im}\in precr_i$，此变异算子会从 $precr_i$ 中移除 γ_{im}。

3.5. 优先级变异算子

在实时操作系统中，每个任务都会被分配一个相对优先级，这是一个用于标识任务重要程度的静态整数。优先级最高的任务一旦就绪并可运行，就会立即获得中央处理器。最常见的实时操作系统调度算法是抢占式调度。

3.5.1. mITPRI

此算子会提高任务的优先级。对于给定的任务 $\tau_i\{\phi_i,\rho_i,c_i,\ \gamma_i,\ prect_i,\ precr_i,\ prio_i,\ jitter_i\}\in T$，此变异算子会将任务的优先级 $prio_i$ 更改为 $proi_i+\delta$。

3.5.2. mDTPRI

此算子会降低任务的优先级。对于给定的任务 $\tau_i\{\phi_i,\rho_i,c_i,\ \gamma_i,\ prect_i,\ precr_i,\ prio_i,\ jitter_i\}\in T$，此变异算子会将任务的周期 i 更改为 $proi_i-\delta$。

3.6. 抖动变异算子

实时操作系统中存在时间抖动，它是给定任务连续两个周期之间的延迟。

3.6.1. mITJ

该算子会增加任务的抖动时间。对于给定的任务 $\tau_i\{\phi_i,\rho_i,c_i,\ \gamma_i,\ prect_i,\ precr_i,\ prio_i,\ jitter_i\}\in T$，此变异算子会将任务的优先级 $jitter_i$ 更改为 $jitter_i+\delta$。

3.6.2. mDTJ

该算子会减少任务的抖动时间。对于给定的任务 $\tau_i\{\phi_i,\rho_i,c_i,\ \gamma_i,\ prect_i,\ precr_i,\ prio_i,\ jitter_i\}\in T$，此变异算子会将任务的周期 $jitter_i$ 更改为 $jitter_i-\delta$。

3.7. 共享内存变异算子

实时操作系统中的任务通常通过共享内存（例如），全局变量、内存缓冲区或硬件寄存器交换数据或信息。如果在任何边界情况下忽略了合理的软件设计，共享内存很容易导致访问冲突。下面介绍一组变量变异算子。

3.7.1. mDSM

该算子会在某个任务中为全局变量赋予一个新值。如果任务读取该全局变量，则在读取发生之前为其赋予一个新值。

3.7.2. mUDSM

此算子会在任务中取消定义一个全局变量。如果某个任务写入此全局变量，则忽略该写入操作。

3.7.3. mRDSM

此算子会移除全局变量的定义。如果全局变量在任务中初始化，则不要对其进行初始化。

3.7.4. mRSM

此算子会为全局变量添加引用。

3.7.5. mRMSMR

此算子会移除对全局变量的引用。

3.7.6. mRSMR

该算子将一个全局变量的引用替换为另一个全局变量。

表 2. Simuilnk 变异算子

变异键	标题
mITO	增加任务偏移量
mDTO	减少任务偏移量
mITPER	增加任务周期
mDTPER	缩短任务周期
mITET	增加任务执行时间
mDTET	缩短任务执行时间
mATPREC	添加任务优先约束
mRTPREC	移除任务优先约束
mARPREC	添加可运行体优先约束
mRRPREC	移除可运行优先级
mITPRI	提高任务优先级
mDTPRI	降低任务优先级
mITJ	增加任务抖动
mDTJ	降低任务抖动
mDSM	定义共享内存
mUDSM	取消定义共享内存
mRDSM	移除定义的共享内存
mRSM	引用共享内存
mRMSMR	移除共享内存引用
mRSMR	替换共享内存引用

4. 变异算子演示

我们介绍了按七类划分的二十个变异算子，并对每一类变异进行了说明。变异算子总结于表 2 中。我们使用简单示例演示每个变异算子的用法。为演示我们的变异算子，我们使用 SimSched 修改以 Simulink模型实现的软件应用程序的属性。从表 1 可知，每个函数调用子系统代表一个 AUTOSAR 可运行实体。当函数调用事件信号到达时，可以有条件地执行函数调用子系统。S函数模块和 Stateflow模块都可以提供此类函数调用事件。SimSched 采用函数调用机制，使用 S函数生成函数调用事件，以调度每个可运行实体（函数调用子系统）。图4 展示了 SimSched 参数对话框，我们可以利用它调整时序属性，从而实现适用于 Simulink 模型的变异算子。

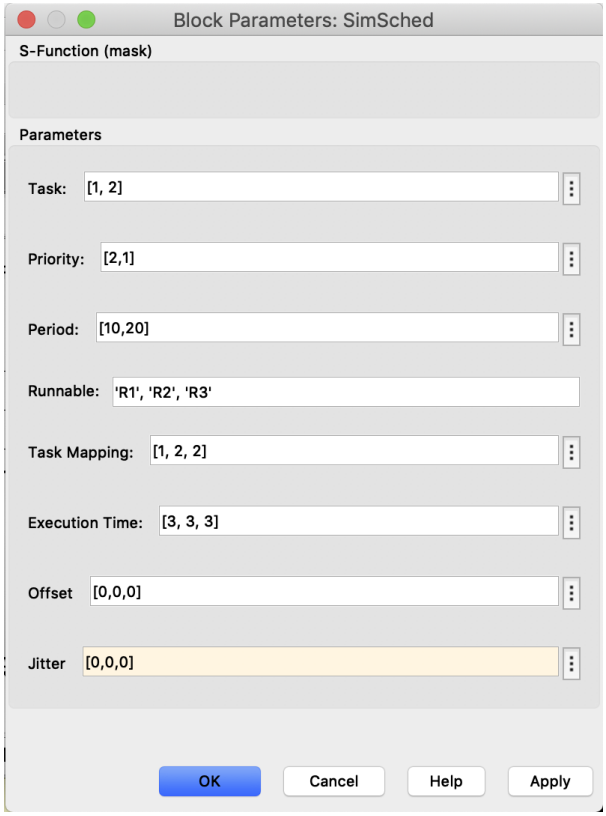


Fig. 4. SimSched 参数设置对话框 .

本节使用几个简单示例展示由我们的变异算子生成的变异体。图 5 说明了如何使用 SimSched 调度 Simulink模型。在此示例中，SimSched 位于左上角，用于调度三个可运行实体（R_1, R_2, R_3），它们被映射到两个任务（T_1, T_2）。可运行实体 R_1 被映射到 T_1,；T_1 的周期为 10毫秒，优先级为 2，R_1 的执行时间为 3毫秒。R_2 和 R_3 被映射到 T_2。T_2 的周期为 20 毫秒，优先级为 1，R_2 和 R_3 的执行时间分别为 3 毫秒和 3 毫秒。详细参数设置列于表 3 中。此示例中有一个名为 A 的数据存储器模块，它定义了一个共享数据存储区，即供具有相同数据存储名称的数据存储读取和数据存储写入模块使用的存储区域。R_1 将一个常数值写入全局变量 A。R_2 先读取 A，然后将 A 与其延迟值之和写入 A。R_3 读取 A，然后从 A 中减去其延迟值，并输出结果。

表 3. 简单示例设置

Task	周期 (ms)	执行时间（毫秒）	优先级	可运行实体
T_1	10	3	2	R_1
T_2	20	3	1	R_2
T_2	20	3	1	R_3

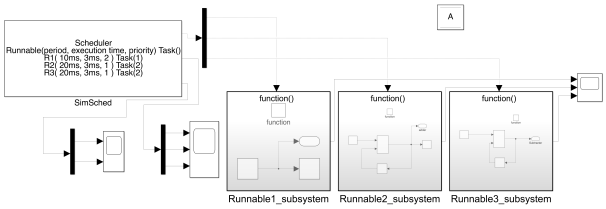


图 5. 使用 SimSched 调度 AUTOSAR 软件组件的简单示例。

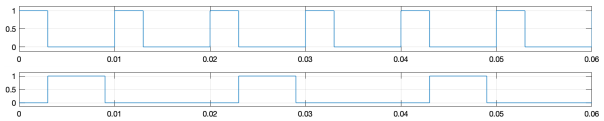


图 6. 运行示例的任务执行甘特图。

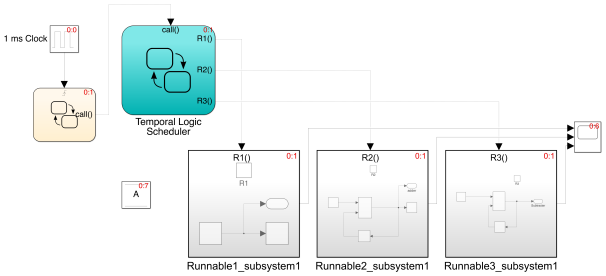


图 7. 使用 Stateflow 调度 AUTOSAR 软件组件的简单示例。

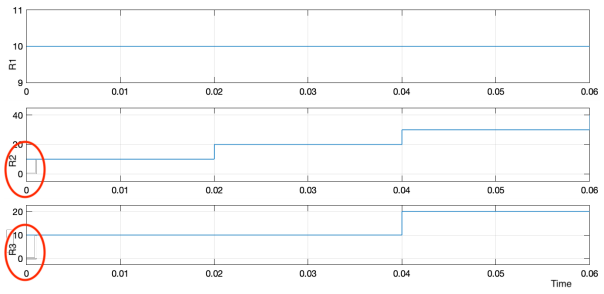


图 8. Stateflow 调度器仿真的简单示例输出。

我们使用图 7 所示简单示例的 Stateflow 调度器版本来展示典型的 Stateflow 调度器仿真结果，然后将其与 SimSched 调度器仿真结果进行比较。任务参数与表 3 中所示的完全相同。我们对 Stateflow 调度器和 SimSched 模型采用相同的任务配置进行仿真。图 8 展示了 Stateflow 调度器仿真结果，图 9 展示了 SimSched 仿真结果。结果图显示了各可运行实体的输出值。从图中可以看出，在图 8 中，R_1, R_2 和 R_3 均在时间 0 执行；在图 9 中，R_1，在时间 0 执行，R_2 在 3毫秒执行，R_3 在 6毫秒执行。R_2 和 R_3 的执行时间晚于图 8 中的 Stateflow 调度器仿真。这是因为 SimSched 会考虑执行时间，而且在单核平台上，每个任务都必须等到前一个任务完成后才能执行。

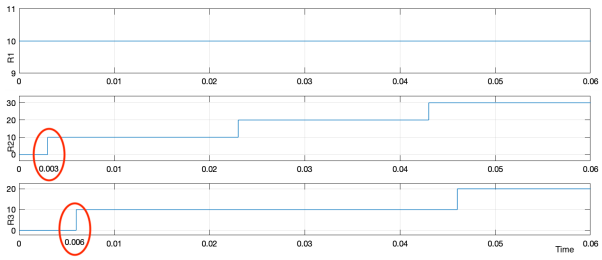


图9. 未应用任何变异算子的 SimSched 仿真简单示例输出。

4.1. 偏移变异算子

我们首先将 *mITO* 变异算子应用于运行示例，假设将 T_1 的 $\delta_1 = 3$毫秒增加，此时可得到图 10 所示的任务执行时间线。可以看到，T_2 在时间 0 首先执行，而在第一个周期中，由于偏移效应，T_1 在 3毫秒 时抢占了 T_2 。第一个周期之后，T_1 和 T_2之间没有发生抢占。接着，我们基于之前的设置应用 *mDTO* 变异算子。将 $\delta_1 = -1$毫秒 设置为 T_1 的偏移量后，T_1 的偏移量为 2毫秒。图 11 展示了任务执行时间线。在第一个周期的执行过程中，T_2 被T_1 抢占。与运行示例中无偏移量时的任务执行甘特图（见图 6）相比，我们可以清楚地看到抢占效应。

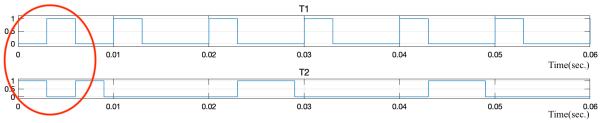


图10. 应用增加偏移变异算子后，运行示例的任务执行甘特图。

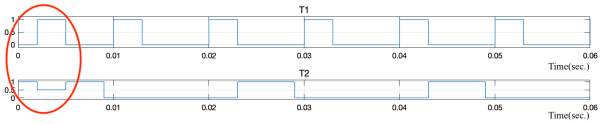


图11. 应用减少偏移变异算子后，运行示例的任务执行甘特图。

应用*mITO*变异算子后，运行示例的输出如图 12 所示，与图 9 不同。由于T_1 在首次实例执行时抢占了T_2 ，因此R_3 的输出从 0 增加到 10，随后又回到 0，再次上升，而不是始终递增。应用*mDTO*变异算子后，运行示例的输出与图 9 相同，因为偏移量算子只影响每个任务的首次执行，而抢占发生在R_2 实例完成首次执行之前。在我们的模型调度器程序中，我们会在每个执行时隙结束时触发各个子系统。从技术上讲，此示例的执行顺序仍为R_1,R_2, 和R_3 ，因此仿真的输出保持不变。

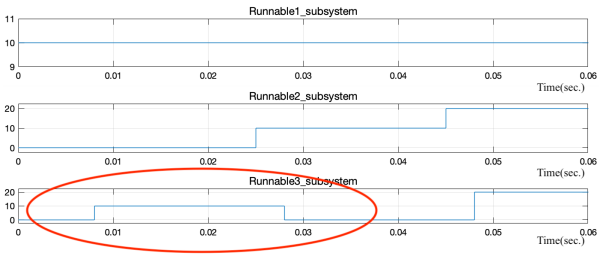


图12. 应用*mITO* 变异算子后的 SimSched 仿真简单示例输出。

4.2. 周期变异算子

我们对该示例应用*mITPER* 算子，以延长任务的周期。我们将 $\delta_1 = 1ms$ 设置为T_1，因此任务T_1 的周期现在为 $11ms$ 。图 13 显示，T_2 在 $22ms$时被抢占，而此次抢占导致仿真产生错误结果，如图 14 所示。R_3 的输出呈交替变化，而非递增。



图 13. 应用增加周期变异算子后运行示例的任务执行甘特图。

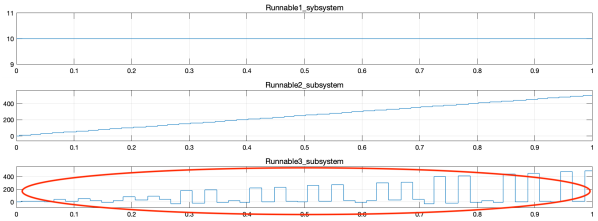


图 14. 应用*mITPER* 变异算子后 SimSched 仿真的简单示例输出。

我们将*mDTPER* 算子应用于此示例，以缩短一个任务的周期。我们将 $\delta_1 = -4$毫秒设为T_1 ，因此任务T_1 的周期现在为 6毫秒 。然后运行仿真，T_2 被T_1 抢占，如图 15 所示，并产生了图 16 所示的错误仿真结果。R_3 的输出不是递增值，而是在零和十之间变化。

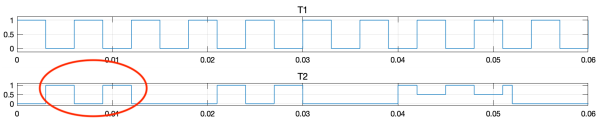


图 15. 应用减少周期变异算子后运行示例的任务执行甘特图。

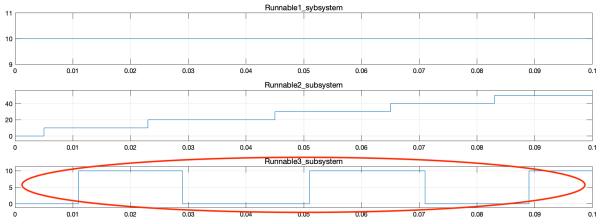


图 16. 应用 *mDTPER*变异算子后 SimSched 仿真的简单示例输出。

4.3. 执行时间变异算子

我们将*mITET* 算子应用于此示例，以增加一个任务的执行时间。我们可以指定任务中的任意可运行实体来增加其执行时间。例如，我们将T_2 中的 $\delta_1 = 4$毫秒 设为R_2 ，这样R_2 的执行时间为 7毫秒 ，而 T_2 现在需要 10 毫秒 才能执行。图 17 显示，T_2 在 10毫秒时被抢占，仿真也因这次抢占而产生错误结果。该错误结果与缩短任务周期示例中的结果相同。我们将*mDTET* 算子应用于此示例，以缩短一个任务的执行时间。我们将$\delta_1 = -1$毫秒 设为T_1 ，因此任务T_1的执行时间现在为 2毫秒 。然后运行仿真，这两个任务之间没有发生抢占，输出符合预期，与原始模型一致。

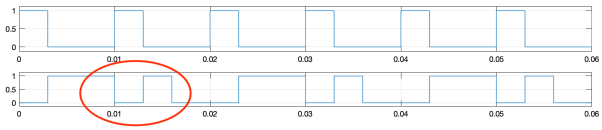


图 17. 应用增加执行时间变异算子后，运行示例的任务执行甘特图。

4.4. 执行优先级变异算子

我们将第二个示例作为表 4 引入，用于解释 *mATPREC* 和 *mRTPREC* 算子。图18展示了该示例的任务执行甘特图。从任务执行图中可以看出，任务的执行顺序为 T_1, T_2, T_1, T_3。

表 4. 简单示例设置

Task	周期 (ms)	执行 时间（毫秒）	优先级	可运行实体
T_1	5	1	3	R_1
T_2	10	4	2	R_2
T_3	10	3	1	R_3

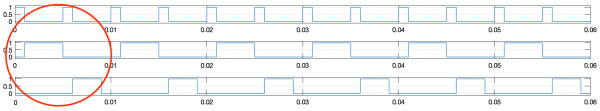


图18. 示例2的任务执行甘特图。

首先，我们假设任务之间不存在优先关系，因此使用 *mATPREC* 变异算子添加优先关系 τ_3 到 $prect_2$，，该关系规定，除非T_2 在T_2的上一个实例之后已执行，否则T_3 的新实例不能启动。因此，我们在设置对话框中设置执行顺序，使 T_3 在T_2 之前执行。图19展示了执行结果：T_2 被 T_1抢占。如果T_2不是可重入函数，那么此次抢占可能导致执行失败。

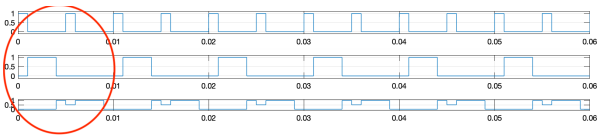


图19. 应用任务优先关系变异算子后的示例2任务执行甘特图。

然后，我们假设T_1 和T_3 之间存在优先关系，并且任务执行图与图18相同。我们应用 *mRTPREC*变异算子，从 τ_1中移除优先关系 $prect_3$ 。结果与图19所示相同。

我们在第一个示例中添加一个可运行实体R_4 ，并将其分配给T_1。新的任务配置如表5所示。R_4 向全局变量A写入的常数值与R_1 不同。我们对这个新示例应用*mARPREC*变异算子，将γ_1 添加到 $precr_4$。R_4 要求R_1 先执行，以便R_4 覆盖R_1写入的值。该算子改变了可运行实体的执行顺序。

表5. 可运行优先级变异算子的任务配置设置。

Task	周期 (ms)	执行 时间(毫秒)	优先级	可运行实体
T_1	10	2	2	R_1
T_2	20	2	1	R_2
T_3	20	2	1	R_3
T_1	10	2	1	R_4

在示例一中，T_2 有两个可运行实体R_2 和R_3，两者之间存在优先关系。我们应用*mRRPREC* 可运行实体移除优先关系变异算子，从$precr_3$中移除优先关系 γ_2 。我们安排R_3 在R_2 之前运行；由于没有优先约束，结果与原始仿真不同。R_3 的原始输出值会随着执行不断增加，而不是零值或固定值。原因是R_3 先执行，并在R_2 向A写入任何新值之前读取A 。图20中的最下方输出行展示了执行结果。

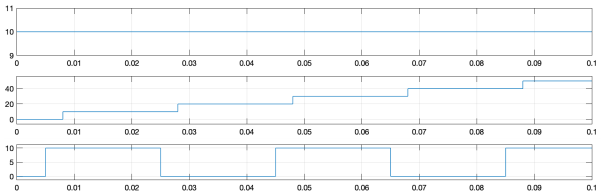


图 20. 示例一中三个可运行实体的输出。

4.5. 优先级变异算子

表 6 . 优先级变异算子示例设置 s

Task	周期 (ms)	执行 时间(毫秒)	优先级	可运行实体
T_1	10	1	4	R_1
T_2	10	2	3	R_2
T_3	10	3	2	R_3

我们将*mITPRI* 算子应用于表 6 中的示例，以提高T_3的优先级。该变异算子将$prio_3$ 的优先级更改为$proi_3 + 3$，因此在此示例中T_3 的优先级最高，为 5，从而使T_3 最先执行。图 21 显示，T_3 在任务执行甘特图中最先触发。此变异改变了任务执行顺序。

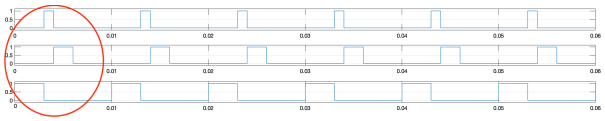


图 21. 应用提高任务优先级变异算子后的任务执行甘特图。

我们将*mDTPRI* 算子应用于T_1，以降低其优先级。该变异算子将$prio_i$ 的优先级更改为$proi_3$，因此在此示例中T_1 的优先级最低，为 1，从而使T_1 最后执行。任务执行甘特图如图 22 所示。

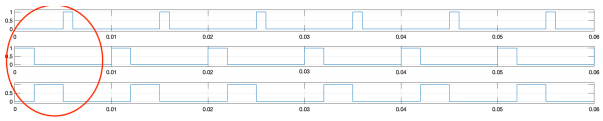


图 22. 应用降低任务优先级变异算子后的任务执行甘特图。

4.6. 抖动变异算子

我们应用 *mITJ* 算子来增加任务的抖动时间。例如，设 $\delta = 2$，该变异算子会将任务的实际释放时间更改为抖动$_1 = 0 + 2$。图 23 展示了由抖动导致T_2被T_1 抢占的执行过程。

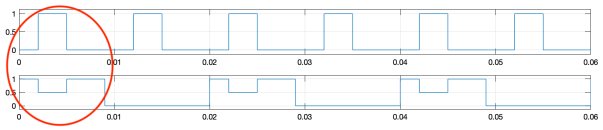


图 23. 应用增加抖动变异算子后的任务执行甘特图。

然后，*mDTJ* 变异算子会减少任务的抖动时间。我们将此算子应用于上述示例，并令 $\delta = -1$，使任务的抖动$_1 = 2 - 1$。在仿真阶段，T_2 被T_1 抢占。

4.7. 共享内存变异算子

在这个共享内存类别中，我们介绍五种变异算子。第一种是*mDSM*，该算子会在读取之前为内存存储赋予一个新值。以我们的示例为例，我们在数据存储读取执行之前紧邻其前添加一个数据存储写入模块，使数据存储写入模块为变量定义一个新值，并选择该变量的初始值作为默认的新值。使用 *mUDSM* 变异算子生成的变异体如图 24 所示，其中仅展示了 Runnable2 子系统的更改。我们在左上角添加了一个常量模块和一个数据存储写入模块。

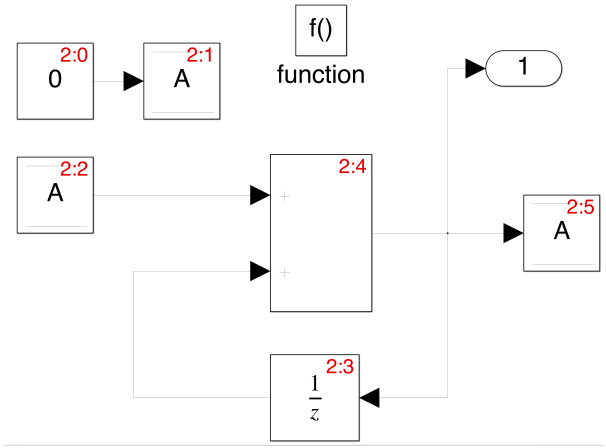


图 24. DSM 变异体的简单示例。

第二个变异算子是*mUDSM*，该算子会忽略对数据存储模块的写入。对于我们的示例，我们移除了数据存储写入模块。图 25 展示了*mUDSM* 变异体，其中数据存储写入模块已被移除。

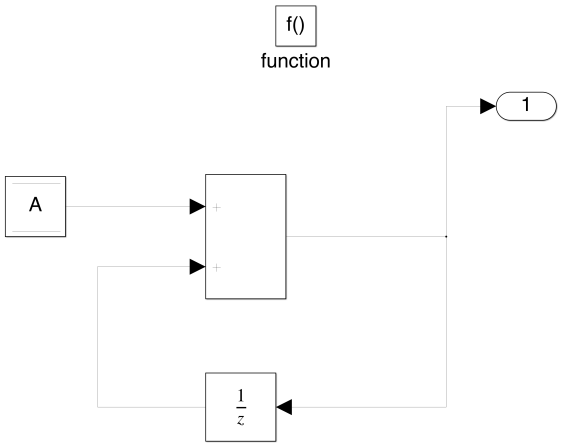


图 25. UDSM 变异体的简单示例。

第三个变异算子是 *mRDSM*，该算子会移除数据存储器的一个初始化值。在许多程序中，变量需要

先有初始值才能正常使用。对于我们的示例，Runnable1 子系统负责初始化数据存储 A；随后，我们移除 Runnable1 子系统中的数据存储写入模块。Simulink 模型仍然可以正常运行仿真，但仿真的输出只有一个值。

变异算子 *mRSM* 向共享内存添加一个新的引用。图 26 展示了一个 Simulink 模型的模块图，该模型包含三个子系统，并将它们映射到两个任务。该模型的根级系统中有一个 DSM 模块 A 。子系统*Task B*1 中有一个数据存储写入模块，*Task B*2 中有一个数据存储读取模块。*Task A* 的周期为 5毫秒，*TaskB* 的周期为 10毫秒。为实现 *mRSM*，我们在$Task_A$ 子系统中添加一个数据存储读取模块，如图 27 所示。在原始示例中，*Task A*先执行，然后*Task B*1 写入A ，而*Task B*2 读取 A。变异程序的执行顺序与原始模型相同。然而，当*Task A* 中的数据存储读取模块执行时，该模块读取的是未初始化的数据存储中的数据，或*Task B*1 先前时刻的数据，因为*Task B* 尚未执行，或者已经执行过。

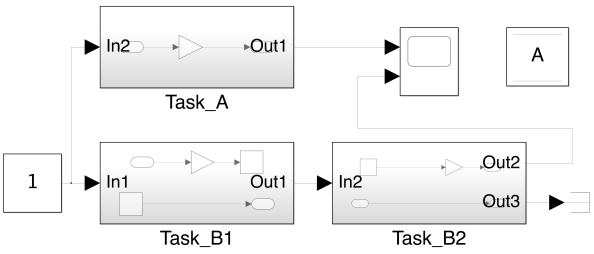


图 26. 一个简单的 Simulink 模型。

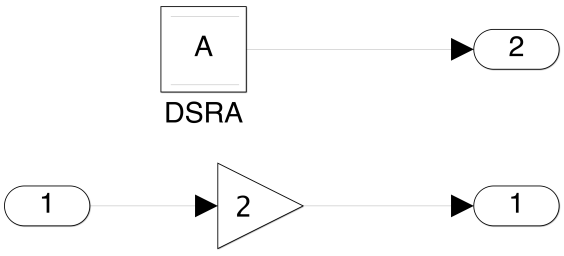


图 27. mRSM 变异算子的示例。向任务 A 模块添加一个数据存储读取模块。

变异算子 *mRMSMR* 删除对共享内存的引用。图 26 中，*Task B*2 引用了根级系统中的 DSM 模块A 。为实现*mRMSMR*，我们删除$Task_B2$ 子系统中的数据存储读取模块。在变异程序中，由于不存在引用，*Task B*2 的输出值恒为零。

5. 评估阶段

在上一节中，我们介绍了模型调度器 SimSched 如何在仿真过程中验证实时上下文，并利用变异测试评估 SimSched。本节中，我们开展实验，以展示如何使用我们的变异测试框架评估 SimSched 和 Stateflow 调度器在实时系统中调度任务的质量。

5.1. 评估流程

为验证所提出的变异算子，我们将其应用于 ML/SL 模型。根据 ML/SL 的能力，我们将评估过程分为基础部分和扩展部分。我们将一阶变异体（FOMs）[22] 应用于 ML/SL 模型以生成一个变异体，这意味着我们仅使用一次变异算子来生成变异体。

5.1.1. 基础情形

在基础情形中，我们检查原始模型、SimSched 模型及其变异体的仿真结果。原始模型 $\mathscr{M}$ 是由 Stateflow 调度器调度的 ML/SL 模型；SimSched 模型 $\mathscr{M}'$ 是由 SimSched 调度的原始模型；变异体（$\mathscr{M}_\mu$ 或 $\mathscr{M}'_\mu$）是经我们的某个变异算子变异后的原始模型或 SimSched 模型。图 28 展示了变异体生成流程的示意图。我们将 $\mathscr{M}$ 的仿真结果作为比较基准，然后将该基准与 $\mathscr{M}_\mu$和 $\mathscr{M}'_\mu$的其他所有仿真结果进行比较。我们检查比较结果，以判断模型仿真过程中是否出现判定失败。如果判定结果为失败，我们就称该变异体被杀死。

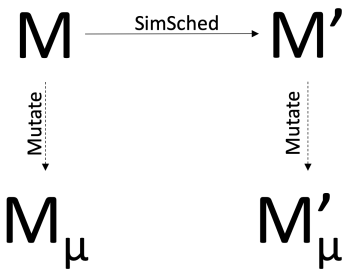


图 28. 模型变异体生成过程示意图。

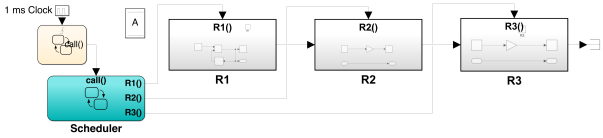


图 29. 由 Stateflow 调度器调度的简单求值示例。

我们用图 29 所示的一个简单示例来说明基础情形的评估流程。该示例是一个原始模型。我们将 Stateflow 调度器替换为 SimSched 调度器，从而构成一个 SimSched 模型。然后，我们分别使用特定的变异算子（例如 mDTPER）生成原始模型和 SimSched 模型的变异体，以缩短任务周期。接着，我们对这两个变异体运行仿真并分析结果，查看是否存在错误。如果 $\mathscr{M}_\mu$ 或 $\mathscr{M}'_\mu$ 的仿真结果与原始模型不同，且出现判定失败，我们就称该变异体被杀死。

在这个示例中，有三个可运行实体 R_1, R_2, R_3，它们被映射到两个任务 T_1, T_2。R_1 被映射到 T_1，R_2, R_3 被映射到 T_2。T_1 的周期为 3 毫秒，T_2 的周期为 6 毫秒。每个可运行实体的执行时间为 1 毫秒。$\mathscr{M}$ 的仿真结果如图 30 所示，图中每个可运行实体的输出都是一条上升的非交错折线。我们将 mDTPER 变异算子应用于原始模型和 SimSched 模型，分别将周期缩短 1 毫秒，以生成变异体。变异体中的任务 T_1 的周期为 2 毫秒。这些仿真的仿真结果如图 31 和图 32 所示。$\mathscr{M}'_\mu$ 的仿真结果与 $\mathscr{M}$ 的结果不同，图中 R_2 和 R_3 的输出是两条上升的交错折线，因为 SimSched 能够仿真执行时间和抢占。T_1 抢占了 T_2，在 SimSched 变异模型中产生了另一条执行轨迹，我们据此判定结果为失败。然而，$\mathscr{M}_\mu$ 的仿真结果与 $\mathscr{M}$ 的结果相似。因此，*mDTPER* 变异体对 $\mathscr{M}'_\mu$ 而言被杀死，对 $\mathscr{M}_\mu$ 而言则仍然存活。由于 ML/SL 的特性，我们无法将这种方法应用于所有变异算子。我们将这种方法与下面的方法结合起来评估变异算子。

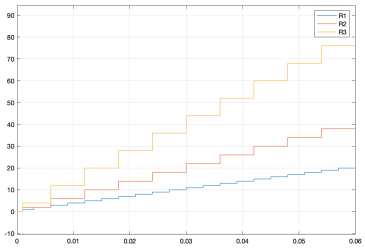


图 30. $\mathscr{M}$ 仿真结果。

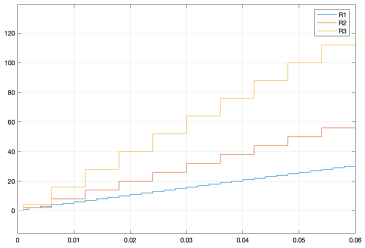


图 31. $\mathscr{M}_\mu$ 仿真结果。

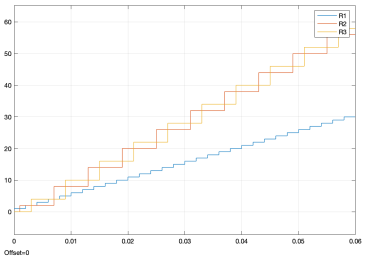


图 32. $\mathscr{M}'_\mu$ 仿真结果

5.1.2. 扩展

为了评估其余变异算子，我们实现了一个具备额外功能的变异生成器，以辅助验证过程。其中一项功能是在给定的任务配置下检查变异体模型的可调度性，以确定是否满足所有任务截止期限。另一项功能是

检查数据访问序列。如果变异模型中存在 DataStore 模块，则会记录对此 DataStore 模块的每次读写操作。随后，我们使用变异模型的数据访问序列与原始模型的数据访问序列进行比较。该变异生成器以 m 文件形式实现为 MATLAB 脚本。

验证过程以经过 Stateflow 调度的 ML/SL 模型和测试规范作为输入。测试规范指定要使用的变异算子。变异体生成器通过 SimSched 将指定的变异算子应用于 ML/SL 模型，并生成一个变异体。随后，变异体生成器利用用于分析仿真的附加功能，分别对原始模型和变异模型执行仿真。如果分析结果显示变异模型中至少有一个任务错过了截止时间，我们就称该变异体被杀死。或者，如果 DataStore 访问序列中至少有一个变量比较结果不匹配，我们也称该变异体被杀死；否则，我们报告该变异体是无害的。

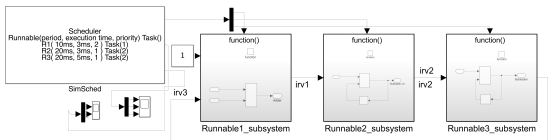


图 33. 使用模型调度器调度 AUTOSAR 软件组件的简单示例。

我们使用图 33 所示的示例来解释验证过程。该示例包含三个可运行实体，并映射到两个任务：R_1 映射到T_1，，R_2, 和R_3 映射到T_2。任务T_1 的周期为 $10ms$，而T_2 的周期为 $20ms$，每个可运行实体的执行时间均为 $3ms$。在该示例模型中，有一个名为A 的 DataStore 模块作为共享变量。如果对该模型应用周期变异算子$mDTPER\ \rho_i - \delta$，其中$i-1$和$\delta = 6$，以缩短T_1 的周期并生成变异体，然后运行该变异体。分析结果显示T_2 错过了截止时间，因此我们称该变异体被杀死。如果对该模型应用执行时间变异算子$mITET\, c_i + \delta$，其中$i\ = 1$ 和 $\delta = 3$，以增加T_1 的执行时间并生成变异体。原始模型的 DataStore 访问序列为$WRWR$模式，其中W表示对共享变量进行写入，而R 表示对共享变量进行读取。该变异体生成了不同的序列，即$WRWWR$。这是因为T_1 的执行时间比原始模型更长，并且在T_2执行期间抢占了T_2。因此，DataStore 访问序列中多了一个W。

5.2. 实验

我们使用两个示例来展示如何使用我们的变异测试框架。首先，我们详细介绍这两个示例。随后，我们将变异算子应用于分别由 Stateflow 调度器和 SimSched 调度的两个模型。

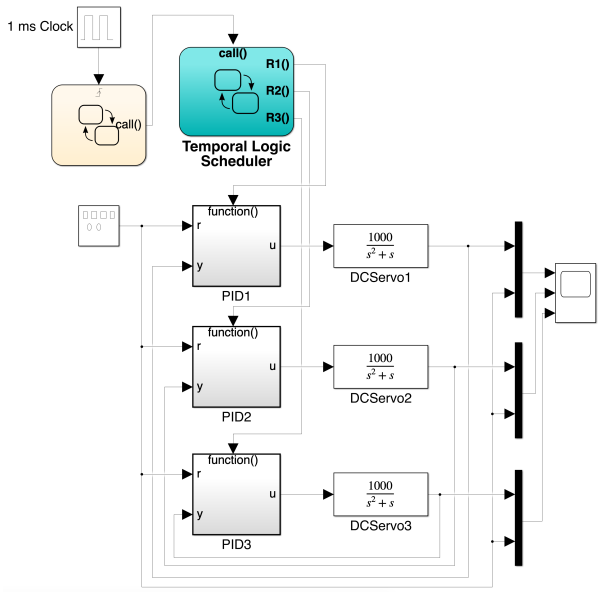


图 34. 改编自 [12] 并采用 Stateflow 调度器的三伺服示例。

5.2.1. 三伺服模型

我们改编了 TrueTime [21] 示例库中的一个示例，该示例展示了三伺服PID控制系统的一种可能实现。该示例如图 34 所示，其中使用了 Stateflow 调度器。在此示例中，三个直流伺服器由一个连续时间系统建模，三个 PID 控制器则实现为三个子系统。我们将三个控制器子系统映射到三个可运行实体$R_1, R_2,$ 和R_3，然后将它们映射到任务$T_1, T_2,$ 和 T_3。这些任务的周期分别为T_1=4 、T_2 = 5 和T_3 = 6 毫秒。每个任务的执行时间相同，为

1毫秒。任务设置如表 7 所示。根据上述任务设置得到的仿真结果如图 35 所示。三幅图展示了相应任务参数按要求设置后，使用三个 PID 控制器得到的电动机输出。在图中，方波是电动机的参考输入信号，未考虑计算延迟。三个 PID 控制器的输出信号均如预期般平滑。

表 7. 三伺服示例设置。

Task	周期 (ms)	执行 时间（毫秒）	优先级	可运行实体
T_1	4	1	3	R_1
T_2	5	1	2	R_2
T_3	6	1	1	R_3

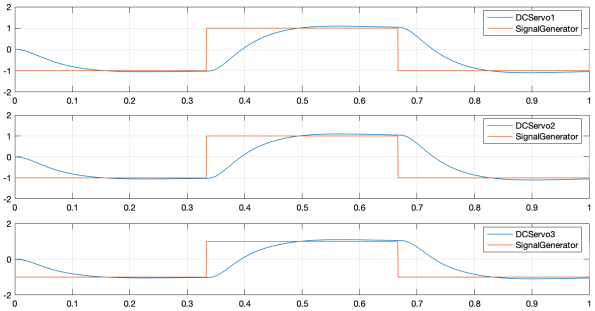


图 35. 使用 Stateflow 调度器的三个伺服器示例输出。

我们将 Stateflow 调度器替换为 Si m-Sched 调度器，更新后的示例如图 36 所示。在此示例中，三个直流伺服器的任务设置与 Stateflow 调度器示例相同。每个可运行实体的执行时间均为 1毫秒。根据上述任务设置，仿真结果与 Stateflow 调度器示例相同。没有任何任务错过截止时间，因此仿真结果显示每个任务都实现了平滑控制。图 37 显示了由 SimSched 生成的任务活动图。每个任务都在其自身的截止时间内执行。

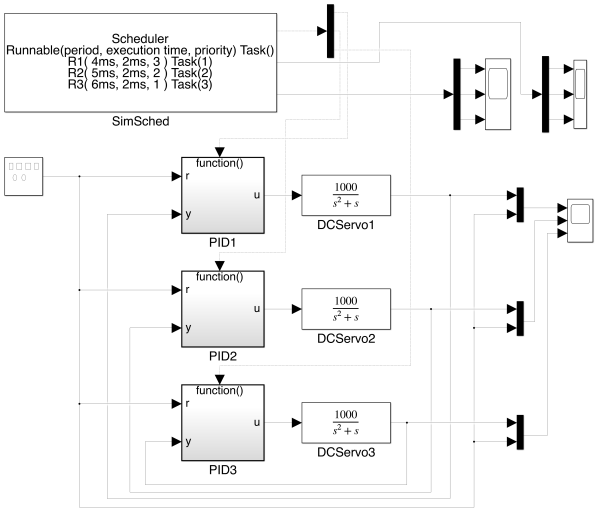


图 36. 改编自 [12]的三个伺服机构示例。

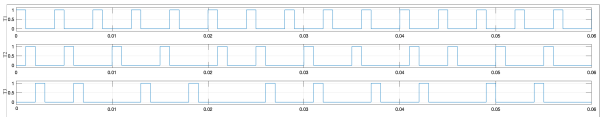


图 37. 由 SimSched 生成的三个伺服机构示例任务活动图。

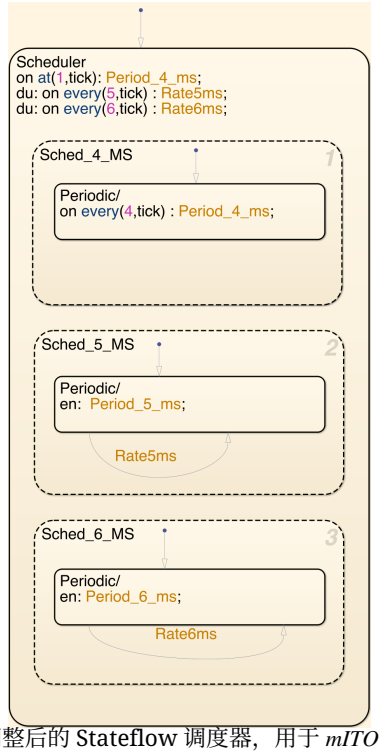


图 38. 调整后的 Stateflow 调度器，用于 *mITO* 突变算子，将 DCServo1 的$offset$增加为 1毫秒 。

接下来，我们将 *mITO*、*mDTO*、*mITPER*、*mDTPER*、*mARPREC*、*mRRPREC*、*mITJ*、*mDTJ*变异算子应用于 Stateflow 调度器和 SimShced 示例，使用相同的变异算子生成两个版本的变异体。为了应用部分变异算子来评估 Stateflow 调度器，我们需要调整该调度器，使其能够用于生成的变异体。图 38 展示了一个针对 偏移量 变异算子调整后的示例。此示例使用状态中的时序逻辑算子 *at* 来设置偏移量参数，从而为 PID1 生成一个变异体；在此示例中，PID1 的运行周期为 4毫秒 。该变异体将 PID1 控制的 DCServo1 的偏移量 增加了 1毫秒。通过我们的模型调度器 SimSiched，可以轻松生成 SimSched 版本的变异体。

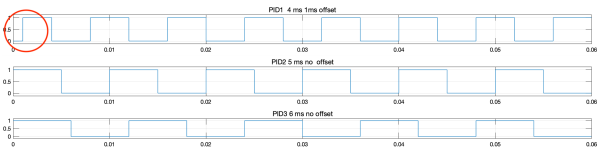


图 39. 对 DCServo1 应用 *mITO* 变异算子，将偏移量增加 1毫秒 后，Stateflow 调度的三伺服示例任务活动图。

我们对通过偏移量 变异算子生成的变异体的两个版本都进行了仿真。这两个版本中，三个伺服机构的输出相同，如图 35 所示。唯一的区别出现在仿真开始时，但不会影响直流伺服系统的平滑控制。我们可以通过以下对比看出差异。图 39 展示了对 DCServo1 应用*mITO* 变异算子，将偏移量 增加到 1毫秒 后，Stateflow 调度任务活动图的情况。应用变异算子之前，每个任务都在时间 0 释放。应用偏移量变异算子后，任务 1 延迟了 1毫秒 ，如图顶部所示。任务 2 和任务 3 都在时间 0 释放。图 40 展示了对 DCServo1 应用 *mITO* 变异算子，将偏移量 增加到 1毫秒 后，SimSched 调度的三个伺服器示例任务活动图。三个输出信号从上到下分别代表任务 T1、T2 和 T3。由于 T1 的偏移量为 1毫秒 ，任务 2 先执行，如图所示，第二行从时间 0 开始。由于 SimSched 调度器包含执行时间参数，任务 2 分别在时间 1 执行，任务 3 在时间 2 执行。

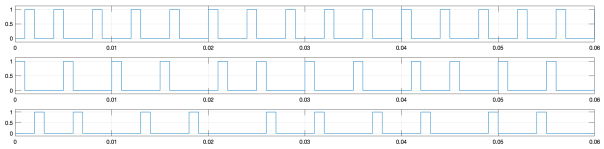


图 40. 对 DCServo1 应用 *mITO* 变异算子，将偏移量增加 1毫秒后，SimSched 调度的三个伺服器示例任务活动图。

我们采用类似的方法，将周期 变异算子应用于三伺服示例，并为 Stateflow 调度器模型和 SimSched 模型生成变异体。我们使用两种变异配置来展示这两种调度器的相似之处与差异。第一种配置为 [1,5,6]。这表示任务 1 的周期为 1毫秒 ，而任务 2 和任务 3 的周期分别保持为 5毫秒 和 6毫秒。图 41 显示了 Stateflow 调度器变异体的仿真结果。由于任务 1 的周期为 1 毫秒 ，它的计算次数过多，输出值超出了图表范围。任务 2 和任务 3 的输出与之前相同。图 42 显示了 SimSched 变异体的仿真结果。任务 1 的输出与 Stateflow 调度器变异体相同。然而，任务 2 和任务 3 的输出与 Stateflow 调度器变异体不同。由于 SimSched 将执行时间纳入考量，任务 1 的优先级最高，执行时间为 1毫秒，因此在整个仿真过程中始终运行。任务 2 和任务 3 的优先级低于任务 1，因此总是被任务 1 抢占。

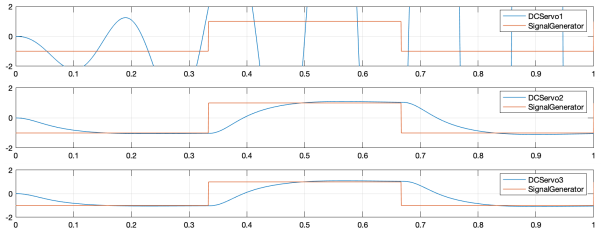


图 41. 对 DCServo1 应用 *mDTPER* 变异算子，将周期缩短至 1毫秒后，Stateflow 调度的三伺服示例输出。

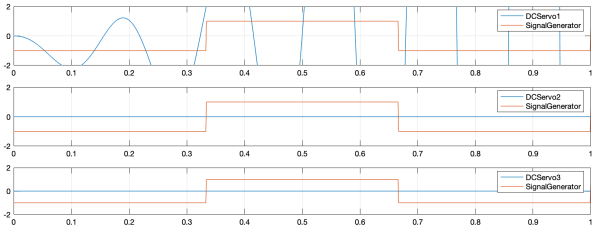


图42. 对 DCServo1 应用 *mDTPER* 变异算子，将周期 缩短至 1毫秒后，SimSched 调度的三个伺服机构示例输出。

第二种配置为 [13,5,6]。这表示任务 1 的周期为 13毫秒，而任务 2 和任务 3 的周期分别保持为 5毫秒 和 6毫秒。图 43 显示了 Stasflow 调度器变异体的仿真结果，SimSched 变异体的输出也与图中所示相同。由于任务 1 的周期为 13毫秒，其计算次数比原始模型少。尽管图中的输出行为看起来像是任务 1 错过了截止时间，但任务 1 的每次执行都满足截止时间要求，并且都是按计划执行的。

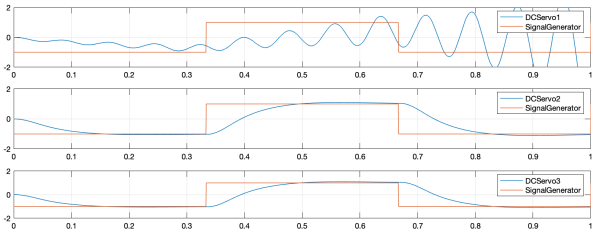


图43. 对 DCServo1 应用 *mITPER* 变异算子，将周期 延长至 13毫秒后，Stateflow 调度器的三个伺服机构示例输出。

我们通过设置 Stateflow 调度器模型中每个并行状态的执行顺序，并配置 SimSched 模型的参数和连接，为 *mARPREC* 和*mRRPREC* 算子生成变异体。这两个变异体的仿真结果与原始模型相同，但每个任务的执行顺序与原始模型不同。

我们通过调整模型中的 Stateflow 调度器，并配置 SimSched 模型中的参数，为 *mITJ*和 *mDTJ*运算符生成变异体。我们将配置设为 [1,0,0]。这表示只有任务 1 的抖动为 1毫秒。图44展示了 SimSched 调度器的三个伺服器示例

对任务活动图应用*mITJ* 变异运算符，将 DCServo1 的抖动 增加至 1毫秒 后。由于任务 1 有 1毫秒 的抖动，因此先执行任务 2，然后执行任务 1 和任务 3。

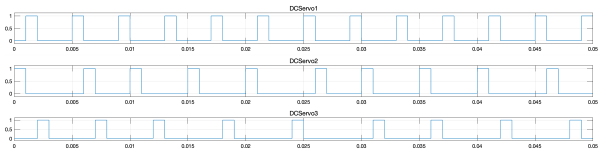


图44. 对 DCServo1 应用 *mITJ* 变异算子，将抖动 增加至 1毫秒后，SimSched 调度器的三个伺服机构示例任务活动图。

由于 Stateflow 调度器缺乏支持，我们仅对 SimSched 模型应用执行时间算子。图 45 展示了 *mITET* 变异算子的效果输出。我们使用 *mITET* 变异算子，将任务 1 的执行时间设置为 $c_1 = 3$毫秒，以生成一个变异体。任务 3 的输出（图底部的 DCservo 3）呈曲线波形。由于被T_1 抢占，且T_3 错过了截止时间，控制变得不稳定。图 46 展示了任务抢占的影响。任务 1 的执行时间为 3毫秒，任务 2 的执行时间为 1毫秒 。任务 1 和任务 2 执行完毕后，本应执行任务 3；但此时正是任务 1 被调度运行的时间。任务 1 的优先级较高，因此任务 3 被任务 1 抢占。任务 3 的第一个实例在 19毫秒 时才执行，因此任务 3 的控制信号输出不像其他任务那样平滑。尽管任务 2 发生了一些抢占，但它错过的截止时间不足以对输出造成显著影响。如图 45 的第二张图表所示，任务 2 的输出信号仍然平滑。

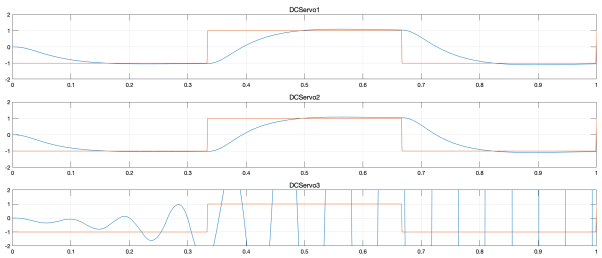


图45. 对 DCServo1 应用 *mITET* 变异算子，将执行时间增加至 3毫秒 后，SimSched 调度器的三个伺服机构示例信号输出。

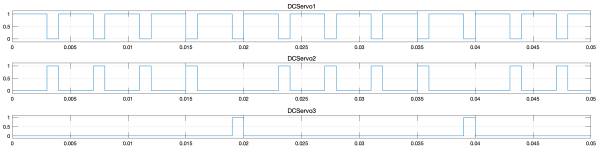


图 46. 对 SimSched 调度器的三个伺服机构示例任务活动图应用 *mITET* 变异算子，将 DCServo1 的执行时间增加至 3 毫秒 后的结果。

5.2.2. 节气门位置控制模型

我们采用了 MathWorks 的 AUTOSAR 软件组件 Simulink 模型，如图 47 所示。该模型为汽车实现节气门位置控制系统，包含三个传感器、一个监控器、一个控制器和一个执行器。这些组件通过六个子系统实现，并映射到六个可运行实体 *TPSSecondary*、监控器、控制器、执行器、*APPSnsr* 和 *TPSPrimary* ，随后映射到任务T_1, 和 T_2。任务周期分别为T_1=5毫秒 和T_2 = 10毫秒 。每个可运行实体的执行时间均为 1毫秒。任务设置见表 8。此示例使用七个 Data-StoreMemory 模块访问共享资源。

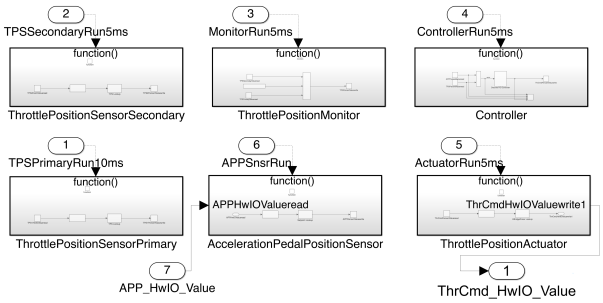


图 47. 节气门位置控制 Simulink 模型包含六个可运行实体。

表 8. 节气门控制示例设置。

Task	周期 (*ms*)	执行 时间（毫秒）	优先级	可运行实体
T_2	10	1	1	TPSPrimary
T_1	5	1	2	副节气门位置传感器
T_1	5	1	2	监控器
T_1	5	1	2	控制器
T_1	5	1	2	执行器
T_2	10	1	1	APPSnsr

图48展示了由 Stateflow 调度器生成的仿真结果。图中的方波是模拟的踏板输入，曲线波是节气门体的输出，表示当前的节气门位置。Stateflow 调度器很好地模拟了节气门控制器的运行过程，并模拟了整个控制过程。

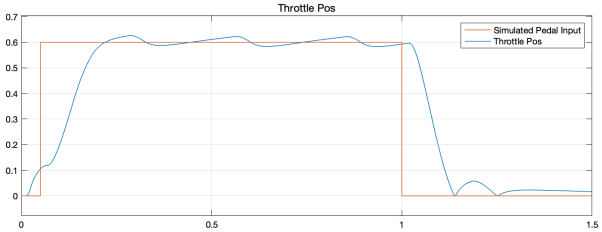


图48. 由 Stateflow 调度器调度的节气门位置控制模型的仿真节气门位置。

图49展示了由 Stateflow 调度器调度的可运行实体活动图表。所有可运行实体均在时间0调度并执行。可运行实体*TP-SPrimary* 和 *APPSensor* 映射到T_2，并且每隔10毫秒 调度并执行一次；图中顶部和底部显示的图表是它们的活动图表。可运行实体*TPPSSendary*、监控器、控制器和执行器 的活动图表是图中间的四个图表。它们每隔5毫秒调度并执行一次。

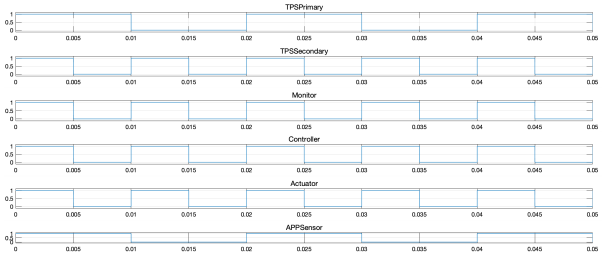


图49. 由 Stateflow 调度器调度的节气门位置控制模型的可运行实体活动图表。

我们将 SimSched 应用于 Stateflow 调度器模型，基于上述任务设置，可以获得与 Stateflow 调度器示例相似的仿真结果。图 50 展示了 SimSched 生成的任务活动图。每个任务都在其截止时间内执行完毕。任务 T_1 和T_2都在时间 0 被调度，但由于T_1 的优先级更高，只有它在

时间 0 执行。T_1 需要 4毫秒 才能运行完毕。T_1 的第一个实例执行完毕后，T_2 开始执行。T_1 的第二个实例在 5毫秒时到达，此时T_2正执行到一半。T_2被T_1 抢占，并在T_1的第二个实例执行完毕后恢复执行。

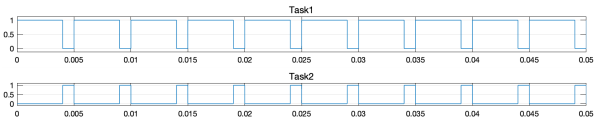


图 50. 由 SimSched 调度器调度的节气门位置控制模型的任务级活动图表。

图 51 展示了由 SimSched 调度器调度的节气门位置控制模型的可运行实体级活动图表。从图中可以清楚地看到各个可运行实体的活动情况。它准确地展示了各个可运行实体的执行顺序。可运行实体*TPPSSendary*、监控器、控制器和执行器 依次执行，随后执行*TPSPrimary*。可运行实体*APPSensor* 在T_1 的第二个实例之后执行，并且它是T_2的抢占点。

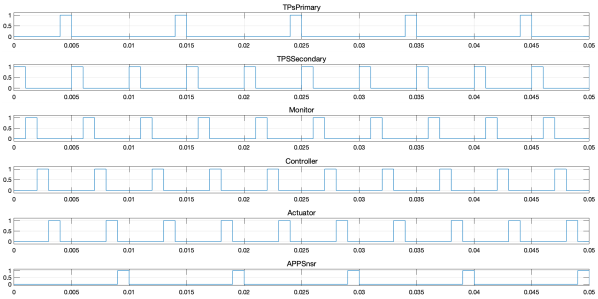


图 51. 由 SimSched 调度器调度的节气门位置控制模型的可运行实体活动图表。

我们采用应用于三个伺服机构示例的相同方法，将其应用于节气门位置控制模型。我们采用 Stateflow 调度，并将其替换为 SimSched，以便在实验中为 Stateflow 调度器和 SimSched 生成变异体。

图 52 展示了对节气门位置控制模型应用 *mTIO* 变异算子后，Sim- Sched 调度的可运行活动图表，其中对T_1增加了 2毫秒 的偏移量。图中底部第二个图表所示的可运行执行器活动图表缺少了首次执行。

T_1 需要 4毫秒 运行，且偏移量为 2毫秒，因此T_1 的总执行时间超过了其 5毫秒的周期。另一方面，由 Stateflow 调度器生成的变异体由于缺少执行时间仿真支持，无法仿真这种超限运行情况。

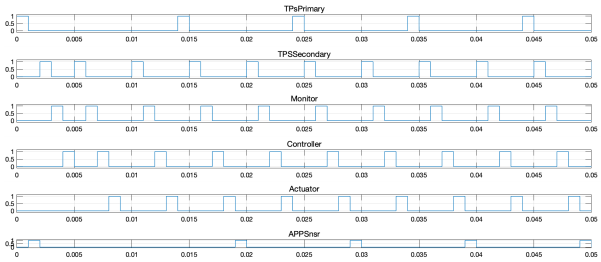


图 52. 对节气门位置控制模型应用*mTIO* 变异算子，将毫秒偏移量增加 2毫秒 ，并由 SimSched 调度后得到的可运行活动图表，适用于T_1。

图 53 展示了对*mITPER* 变异算子应用于T_1 并设为 100毫秒后，由 SimSched 调度的节气门位置控制模型的仿真节气门位置。Stateflow 调度器也能输出相同的图。*mITPER* 变异算子可以在相同时间内减少任务的计算时间，从而导致图中所示的不稳定控制。

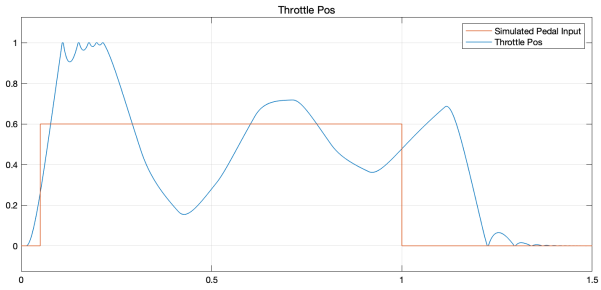


图 53. 对节气门位置控制模型应用*mITPER*变异算子后，由 Stateflow 调度器调度得到的仿真节气门位置。

图 54 显示了应用*mDTPER* 变异算子后，由 SimSched 调度的节气门位置控制模型的模拟节气门位置，该算子针对的是T_1 ，时间为 4毫秒。*mDTPER* 可能导致T_2 没有输出信号，因为高频率任务会增加计算时间，而低频率任务无法获得执行。在此示例中，我们使用*mDTPER* 变异算子将 $\rho_1 = 4$设为毫秒，

随后T_2 始终被T_1 抢占，因此没有机会执行。

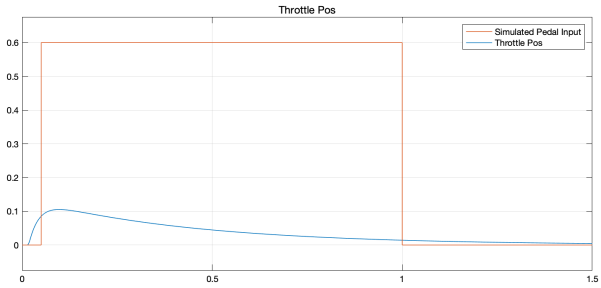


图 54. 应用*mDTPER* 变异算子后，由 SimSched 调度的节气门位置控制模型仿真的节气门位置。

我们仅对 SimSched 模型应用执行时间算子。我们使用$c_1 = 5$毫秒，通过*mITET*变异算子为T_1 生成一个变异体。该变异体的输出与图 54 所示的节气门位置相同。由于T_1 的执行时间增加了 1毫秒，它占用了其周期内的所有时间片。在仿真过程中，T_2 被T_1 抢占。

我们将*mARPREC* 和*mRRPREC* 变异算子应用于此节气门位置控制示例。首先，可运行实体*APPSnsr* 与*TPSPrimary*之间没有优先级关系，因此我们使用*mARPREC*变异算子添加优先级 $\gamma_{APPSnsr}$ 到$precr^{TPSPrimary}$，为这两种调度器生成变异体。可运行实体控制器 使用*APPSnsr* 和 *TPSPrimary* 产生的值，计算节气门执行器的节气门百分比值。两个变异体的仿真结果变化都很小。图 55 显示了原始模型与 SimSched 变异体的仿真结果对比。

其次，控制器与执行器之间存在优先级，且控制器 在执行器之前执行。我们移除这对可运行实体之间的优先级，使执行器 （目标）在控制器 （源）之前运行，从而改变数据依赖关系并延迟数据。仿真中的差异与图55类似。

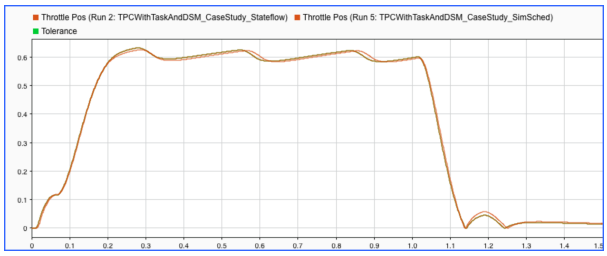


图 55. 由 SimSich 调度的 *mARPREC* 变异算子作用后，原始模型与变异体之间的仿真结果差异。

我们将*mDSM*、*mUDSM*、*mRDSM*、*mRSM*、*mRMSMR*和*mRSMR* 变异算子应用于此示例，并分别为 Stateflow 调度器和 SimSched 生成相应的变异体。有趣的是，由于共享内存变异算子未改变任务的时间属性，因此两类变异体的仿真结果一致。例如，我们将*mDSM*变异算子应用于变量 *ThrCmdPercentValus*，并将该变量设置为新的常数值。两类变异体的仿真结果相同，如图56所示。由于该变量是查找表的输入，因此始终存在一个与输入值匹配的输出，并产生相应的输出值。输入值是常数，因此节气门位置的输出是一条平滑曲线。

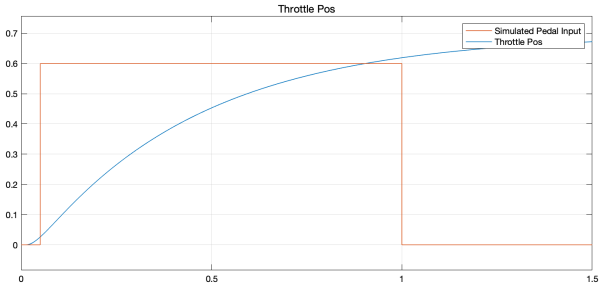


图 56. 对变量 *ThrCmdPercentValus* 应用由 SimSich 调度的 *mDSM*变异算子后的节气门位置输出。

5.3. 评估结果

我们将评估流程应用于示例模型，以研究变异算子杀死变异体的有效性。表 9 和表 10 总结了变异算子的效果，

其中每行给出了变异体数量以及我们提出的变异算子的变异得分。变异得分用于衡量被杀死的变异体占变异总数的百分比。

表 9. 三个服务示例的变异算子的变异分析。

	Stateflow 调度器		SimSched	
算子	变异体	杀死	变异体	杀死
偏移量	36	24	36	27
周期	39	25	39	25
执行 Time	N/A	N/A	36	31
优先级	5	0	5	0
优先级	18	0	18	0
抖动	36	24	36	27
变异得分	54.48%		64.71%	

对于三伺服示例，我们为 Stateflow 调度器模型生成了 134 个变异体，为 SimSched 模型生成了 170 个变异体。Stateflow 调度器模型的变异得分为 54.48%，SimSched 模型的变异得分为 64.71%。评估结果表明，时间相关变异算子对变异测试的影响最大，例如偏移量、周期、执行时间和抖动 变异算子。我们发现，优先级 和优先级 变异算子杀死的变异体数量为零，因为在此示例中，每个控制器都独立控制一个电机。它们之间没有关联，因此优先级的变化不会导致仿真发生变化。不过，三个控制器运行在同一个中央处理器上，一个任务执行时间的长短会影响其他任务。我们还发现，偏移量 变异算子只会影响每个任务的初始执行，并导致任务错过截止时间。尽管如此，每个变异体的仿真结果都表明，每个控制器都能稳定地控制各自的伺服电机。

表10. 节气门位置控制示例的变异算子变异分析。

	Stateflow 调度器		SimSched	
算子	变异体	杀死	变异体	杀死
偏移量	19	7	19	15
周期	30	6	30	19
执行 Time	N/A	N/A	34	34
优先级	23	10	23	10
优先级	11	0	11	0
抖动	19	12	19	15
共享内存	72	46	72	46
变异得分	49.39%		70.2%	

对于节气门位置控制示例，我们为 Stateflow 调度器模型生成了 164 个变异体，为 SimSched 模型生成了 198 个变异体。Stateflow 调度器模型的变异得分为 49.39%，SimSched 模型的变异得分为 70.2%。评估结果与前一个示例相似。时间相关变异算子对变异测试最为有效。我们发现，对于 $\mathscr{M}_\mu$ 和 $\mathscr{M}'_\mu$，共享内存 变异算子杀死的变异体数量相同。在此示例中，任务T_2 有两个可运行实体*TPSPrimary* 和 *APPSnsr*，每个实体都在每次执行时更新一个共享变量，二者之间没有直接关系。尽管 SimSched 可以模拟T_2 被抢占从而中断其执行的情况，但对于 $\mathscr{M}_\mu$ 和 $\mathscr{M}'_\mu$，共享内存变异体的模型行为相同。

从以上两个示例可以看出，变异算子依赖于应用，而 SimSched 在时间相关变异算子上的变异得分更高。例如，对于三伺服示例，优先级 变异算子未能杀死任何变异体；但对于节气门位置控制示例，无论使用 Stateflow 调度器还是 SimSched，该算子都能杀死十个变异体。三伺服示例完全不需要设置优先级；每个任务只控制自身。然而，节气门位置控制示例需要设置优先级。一个可运行实体会使用上一个可运行实体执行时产生的数据。如果改变优先级，执行顺序就会不同于原始模型中的执-

行顺序，从而产生不同的数据流。例如，对于周期变异算子，在三伺服示例中，Stateflow 调度器和 SimSched 杀死的变异体数量相同；但在节气门位置控制示例中，SimSched 杀死的变异体比 Stateflow 调度器更多。这是因为节气门位置控制示例在T_1 中有四个可运行实体，且每个可运行实体的毫秒 执行时间为 1。我们将*mDTPER* 应用于T_1 ，生成一个变异体，使T_1 的周期变为 4毫秒 ，而非 5毫秒，那么T_1 将占用所有执行时隙，而T_2 在 SimSched 中将不会执行。然而，Stateflow 调度器不考虑执行时间，因此T_1 和T_2 都会按计划执行。

6. 相关工作

我们的工作与基于模型的变异测试（MBMT）相契合，且已应用于各种模型。Trakhtenbrot [41] 提出了面向反应式系统基于状态图的模型的变异测试。这种方法主要处理状态图模型特定语义与模型实现之间的一致性问题。变异测试已应用于特征模型，以测试软件产品线 [20]。特征模型表示软件产品线和可配置系统的可变性。El-Fakih等人.[14] 开发了一种面向扩展有限状态机（EFSM）的基于变异的测试用例生成技术，用于检查被测 EFSM 是否符合用户定义的故障。Belli等人.[6] 对基于模型的变异测试方法进行了广泛综述，并详细介绍了该方法在基于图的模型中的应用，包括有向图、事件序列图、有限状态机和状态图。近期一篇变异测试综述[31] 介绍了最新的先进方法，这些方法利用变异体支持软件工程活动中的模型制品。

我们的工作也属于定时系统测试，这类测试需要实时环境。研究人员 [3,28,29] 利用研究最充分的形式化方法——时间自动机（TA）向定时系统注入故障，并揭示了随机生成的测试套件无法发现时间相关错误。Nilsson等人.[28] 首次提出了一组基于带任务的时间自动机（TAT） [30] 扩展而来的时间自动机变异算子，用于测试实时系统，这类系统依赖于各个任务的执行时间和执行顺序。这些变异算子关注任务能否及时满足其截止期限。Aichernig等人.[3] 为定时系统提出了一个变异测试框架，其中定义了八种用于变异模型的变异算子，其变异体以 Uppaal 规范格式中的时间自动机变体表示。作者还开发了一个面向实时系统的基于变异的测试用例生成框架，其中采用了符号有界模型检测技术和增量求解。Cornaglia等人.[11] 提出了自动化框架 MODELTime，该框架借助 MATLAB/Simulink 仿真，支持在嵌入式应用的基于模型的开发过程中研究目标平台相关的时序。

ML/SL 是用于建模和仿真嵌入式系统的最流行形式化方法之一，许多研究人员已探索应用于 ML/SL 模型的各种变异算子。Hanh等人.[18] 通过调查 ML/SL 模型中的常见故障，提出了一组变异算子，用于验证测试套件，并介绍了 ML/SL 模型的变异测试流程。他们提供了十二种变异算子，并将其分为五类。Stephan 等人.[38] 采用变异测试技术来比较 ML/SL 模型的模型克隆检测工具。他们提出了 ML/SL 的分类体系，并基于三种克隆类型提出了一组结构变异算子。这些变异算子用于评估模型克隆检测器。已有研究探讨了使用基于变异的技术自动为 ML/SL 模型生成测试用例 [8,19]。这些技术能够有效生成少量测试用例，并在一组来自汽车领域的 Simulink 模型上实现较高的覆盖率。近期的一项工作 SLEMI [10] 将变异技术应用于 Simulink 编译器，并使用工具为种子模型生成变异体，发现了 Simulink 模型中的 9 个已确认错误。

我们的工作旨在利用变异分析识别 ML/SL 模型中的潜在时间相关错误。Roy 和 Cordy [33,34] 首次提出使用变异分析来协助评估软件克隆检测工具。他们开发了一个基于变异的方法，用于测试代码克隆检测器，

以变异分析为基础。Stephan 等人.[37,36] 提出了一个框架，可利用变异分析对模型克隆检测器进行客观、定量的评估和比较。他们的工作基于一种针对 ML/SL 模型变异的结构变异方法。我们的变异算子基于定时系统任务模型；然而，目前尚无相关研究直接将定时系统中的 ML/SL 模型集成到模型在环（MIL）阶段。因此，我们开展了本文所述的工作。

协同仿真[16] 是基于模型的测试中广泛使用的一种技术，旨在尽可能验证系统各子系统的功能。组合各子仿真器的仿真，可以实现对耦合系统的联合仿真。在基于模型的设计领域，人们出于不同目的使用多种语言和工具，例如设计连续对象或离散控制器。近年来制定了一项开放标准——功能模型单元接口（FMI），用于以标准化格式交换仿真模型，并支持协同仿真。Gomes等人.[15] 提出了一种方法，以促进功能模型单元接口标准的实现。他们采用基于模型的测试方法来评估可导出功能模型单元（FMU）的工具。因此，他们能够找出其中的歧义，并提高对 FMI 标准的符合度。Garro等人.[7] 利用 FMI 进行协同仿真，并基于形式化需求建模语言和 Modelica语言验证系统需求。Zafar等人.[42] 提出了一种系统化、由工具支持的基于模型的测试工作流，以促进嵌入式系统的基于仿真的测试过程。该工作流涵盖从需求阶段、可执行测试脚本的生成，到在仿真层面执行生成的测试脚本。

7. 结论与未来工作

本文提出了一组针对 ML/SL 模型的定时变异算子，主要用于将定时任务模型集成到 ML/SL 模型中，以便更好地支持基于变异分析的模型在环（MIL）仿真。此外，在开发过程的早期阶段进行测试可以降低开发成本，因为较早进行更改和修复错误要容易得多。我们介绍了一种定时任务模型，并基于该任务模型提出了一组适用于 ML/SL 模型的变异算子。我们实现了一个变异分析框架，可将变异算子应用于简单的 ML/SL 模型。我们在若干 ML/SL 模型上展示了该方法。结果验证了变异分析能够揭示时间相关故障。我们计划在未来实现 ML/SL 环境下变异测试流程的自动化，并改进变异算子以暴露缺陷。我们还将进一步在更复杂的工业 ML/SL 模型集上验证我们的变异分析方法。

致谢

本研究部分得到了加拿大自然科学与工程研究委员会（加拿大自然科学与工程研究委员会）的资助，是 NECSIS 汽车合作项目的一部分。该项目的合作方包括通用汽车、IBM Canada 和 Malina Software Corp。

1. Bernhard K. Aichernig、Harald Brandl、Elisabeth J¨obstl、Willibald Krenn、Rupert Schlick 和 Stefan Tiran。基于模型的变异测试的击杀策略。软件测试、验证与可靠性，25(8):716–748，2015年12月。2. Bernhard K. Aichernig 和 Florian Lorber。基于确定化定时自动机的部分模型生成自适应测试用例。在*2015*年 *IEEE* 第八届软件测试、验证与确认国际会议研讨会（*ICSTW*），第1–6页。IEEE，2015年4月。3. Bernhard K. Aichernig、Florian Lorber 和 Dejan Niˇckovi´c。变异体的时机——基于定时自动机的模型变异测试。在计算机科学讲义（含人工智能讲义和生物信息学讲义子系列），第7942卷 LNCS，第20–38页。施普林格，柏林，海德堡，2013年。4. Rajeev Alur 和 David L. Dill。定时自动机理论。理论计算机科学，126(2):183–235，1994年4月。5. AUTOSAR。AUTOSAR开发合作伙伴关系。http: //www.autosar.org，2018年。6. Fevzi Belli、Christof J. Budnik、Axel Hollmann、Tugkan Tuglular 和 W. Eric Wong。基于模型的变异测试——方法与案例研究。《计算机编程科学》，120:25–48，2016年5月。

7. Daniel Bouskela、Alberto Falcone、Alfredo Garro、Audrey Jardin、Martin Otter、Nguyen Thuy 和 Andrea Tundis。面向信息物理系统工程的形式化需求建模：基于 FORM-L 和 Modelica 的集成解决方案。需求工程，第1–30页，2021年。8. Angelo Brillout、Nannan He、Michele Mazzucchi、Daniel Kroening、Mitra Purandare、Philipp R¨ummer 和 Georg Weissenbacher。面向 Simulink 模型的基于变异的测试用例生成。在计算机科学讲义（含人工智能讲义和生物信息学讲义子系列），第6286卷 LNCS，第208–227页，2010年。9. Jian Chen、Manar H Alalfi、Thomas R Dean 和 S Ramesh。在 Simulink 中对 AUTOSAR 实现进行建模。在欧洲建模基础与应用会议，第279–292页。施普林格，2018年。10. Shafiul Azam Chowdhury、Sohil Lal Shrestha、Taylor T. Johnson 和 Christoph Csallner。SLEMI：基于输入等价（EMI）的信息物理系统模型变异，用于发现 Simulink 中的编译器缺陷。在 *2020*年 *IEEE/ACM* 第*42*届国际软件工程会议（*ICSE*），第335–346页，2020年。11. Alessandro Cornaglia、Shakib Hasan、Alexander Viehl、Oliver Bringmann 和 Wolfgang Rosenstiel。MODELTime：面向嵌入式处理器的 Simulink 模型全自动时序探索。在*AmE 2019*——汽车与电子相遇；第*10*届 *GMM* 研讨会，第1–6页，2019年。12. Fabio Cremona、Matteo Morelli 和 Marco Di Natale。TRES：用于控制 Simulink 仿真的调度器、任务和消息的模块化表示。第*30*届 *ACM* 应用计算年度研讨会论文集，第1940–1947页，2015年。13. R.A. DeMillo、R.J. Lipton 和 F.G. Sayward。测试数据选择提示：为实务程序员提供帮助。*Computer*，11(4):34–41，1978年4月。14. Khaled El-Fakih、Anton Kolomeez、Svetlana Prokopenko 和 Nina Yevtushenko。由用户定义故障驱动的基于扩展有限状态机的测试推导。在*2008*年国际软件测试、验证与确认会议，第308–317页。IEEE，2008年4月。15. Cl´audio Gomes、Romain Franceschini、Nick Battle、Casper Thule、Kenneth Lausdahl、Hans Vangheluwe 和 Peter Gorm Larsen。基于模型的测试在功能模型单元动态评估中的应用。2020年。16. Cl´audio Gomes、Casper Thule、David Broman、Peter Gorm Larsen 和 Hans Vangheluwe。协同仿真：综述。*ACM* 计算调查（*CSUR*），51(3):1–33，2018年。

17. R.G. Hamlet. 借助编译器测试程序。*IEEE*《软件工程汇刊》，SE-3(4):279–290，1977年7月。18. Le Thi My Hanh 和 Nguyen Thanh Binh。Simulink 模型的变异算子。载于第四届知识与系统工程国际会议论文集（*KSE 2012*），第54–59页。IEEE，2012年8月。19. N. He、P. R¨ummer 和 D. Kroening。基于形式概念分析的嵌入式Simulink测试用例生成。载于*2011*年第*48*届*ACM/EDAC/IEEE*设计自动化会议（*DAC*），第224–229页，2011年。20. Christopher Henard、Mike Papadakis、Gilles Perrouin、Jacques Klein 和 Yves Le Traon。通过基于模型的变异评估软件产品线测试：相似性测试应用。载于*2013*年*IEEE*第六届软件测试、验证与确认国际会议研讨会，第188–197页。IEEE，2013年3月。21. Dan Henriksson、Anton Cervin 和 Karl-Erik ˚Arz´en。TrueTime：使用 MATLAB / Simulink进行实时控制系统仿真。北欧 *MATLAB*会议论文集，2003年。22. Yue Jia 和 Mark Harman。变异测试的发展分析与综述。*IEEE*《软件工程汇刊》，37(5):649–678，2011年9月。23. Edward A. Lee、Stephen Neuendorffer 和 Gang Zhou。同步反应式模型。载于 Claudius Ptolemaeus 编，《style id='16'>使用Ptolemy II进行系统设计、建模与仿真。Ptolemy.org，2014年。24. J. Lehoczky、L. Sha 和 Y. Ding。速率单调调度算法：精确特征及平均情况行为。 *[1989]* 实时系统研讨会论文集，第0–5页，1989年。25. MathWorks。Simulink用户指南，r2017b。http: //www.mathworks.com，2017年。26. Reza Matinnejad、Shiva Nejati、Lionel C. Briand 和 Thomas Bruckmann。时间连续Simulink模型的自动化测试套件生成。载于第*38*届软件工程国际会议论文集 - *ICSE '16*，第595–606页，美国纽约州纽约市，2016年。ACM出版社。27. Andreas Naderlinger。在Simulink中使用时序感知模块仿真抢占式调度。载于*2017*年欧洲设计、自动化与测试会议及展览会（*DATE*），第758–763页。IEEE，2017年3月。28. R. Nilsson、J. Offutt 和 S.F. Andler。面向及时性的基于变异的测试准则。载于第*28*届年度国际计算机软件与应用会议论文集，*2004*年。*COMPSAC 2004*。，第306–311页。IEEE，2004年。29. Robert Nilsson 和 Jeff Offutt。及时性的自动化测试：案例研究。载于国际软件工程会议论文集，2007年。

30. C. Norstrom、A. Wall 和 Wang Yi。定时自动机作为事件驱动系统的任务模型。载于第六届实时计算系统与应用国际会议论文集。*RTCSA'99*（目录号*PR00306*），第182–189页，1999年。31. Mike Papadakis、Marinos Kintis、Jie Zhang、Yue Jia、Yves Le Traon 和 Mark Harman。变异测试进展：分析与综述。载于《计算机进展》，第112卷，第275–378页。爱思唯尔，2019年。32. Mike Papadakis、Marinos Kintis、Jie Zhang、Yue Jia、Yves Le Traon 和 Mark Harman。变异测试进展：分析与综述。《计算机进展》，112:275–378，2019年1月。33. Chanchal K. Roy 和 James R. Cordy。一种基于变异/注入的自动化框架，用于评估代码克隆检测工具。载于*IEEE*软件测试、验证与确认国际会议研讨会（*ICSTW 2009*），第157–166页，2009年。34. Chanchal K. Roy、James R. Cordy 和 Rainer Koschke。代码克隆检测技术与工具的比较和评估：一种定性方法。《计算机编程科学》，74(7):470–495，2009年5月。35. Julien Schmaltz 和 Jan Tretmans。定时系统的一致性测试。载于《计算机科学讲义》（含《人工智能讲义》和《生物信息学讲义》子系列），第5215卷LNCS，第250–264页。施普林格，柏林，海德堡，2008年。36. Matthew Stephan。使用变异分析评估模型克隆检测器。载于第*30*届……论文集国际软件维护与演化会议，*ICSME 2014*，第633–638页。电气电子工程师学会公司，2014年12月。37. Matthew Stephan、Manar H. Alafi、Andrew Steven- son 和 James R. Cordy。使用变异分析进行模型克隆检测器比较框架研究。载于国际软件工程会议论文集，第1261–1264页，2013年。38. Matthew Stephan、Manar H Alalfi 和 James R Cordy。迈向 Simulink 模型变异分类体系。载于*IEEE* 第七届软件测试、验证与确认国际会议研讨会论文集，*ICSTW 2014*，第206–215页，2014年。39. AUTOSAR 联盟。将 Simulink 应用于 AUTOSAR，r3.1，2006年。40. AUTOSAR 联盟。AUTOSAR 标准，r4.3，2018年。41. Mark Trakhtenbrot。面向实现的状态图模型变异测试。载于*2010* 年第三届软件测试、验证与确认国际会议研讨会，第120–125页。IEEE，2010年4月。42. Muhammad Nouman Zafar、Wasif Afzal 和 Eduard Enoiu。迈向嵌入式系统基于模型的测试工作流。载于第*12*届测试用例设计、选择与评估自动化国际研讨会论文集，第33–40页，2021年。43. Yuan Zhan 和 John A. Clark。面向 Simulink 模型的基于搜索的变异测试。载于*GECCO 2005*——遗传与进化计算会议，第1061–1068页，美国纽约州纽约市，2005年。ACM出版社。