软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(5):1711−1735 [doi: 10.13328/j.cnki.jos.006557]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于场景模型的 DDS 架构一体化舰船任务系统测试*

钱 巨[1], 王 寅[1], 程 浩[1], 韦正现[2]

[1](南京航空航天大学 计算机科学与技术学院, 江苏 南京 211106)

[2](中国船舶工业系统工程研究院, 北京 100036)

通信作者: 钱巨, E-mail: jqian@nuaa.edu.cn

摘　要: 以数据分发服务 (data-distribution service, DDS) 为基础架构的新型一体化舰船任务系统在研发模式、结构和应用方面具有特殊特点, 使得其测试相当具有挑战. 基于模型的测试 (model-based testing, MBT) 是工业系统测试的一种有效方法. 然而, 对于类舰船任务系统, 由于其自身的高度复杂性和协同开发方式, 传统需要建立完整模型以表达系统内部行为的 MBT 技术极难应用. 为此, 提出了一种基于场景模型的类舰船任务系统 MBT 方法. 方法从外部角度构建模型来表达 DDS 架构系统中的交互场景. 模型使用扩展正则表达式来建模交互序列, 使用基本数据元素限制、约束公式和计算函数来建模交互数据, 能够在保留抽象性的同时便捷并相对完整地表达交互过程. 基于场景模型, 进一步提出算法生成直接可执行的测试用例. 在某真实舰船任务系统上的实验表明, 方法能够测试从一族舰船任务系统历史失效中总结出的 21 种常见风险场景, 对类舰船任务系统的 DDS 架构工业系统测试具有潜在应用价值.

关键词: 基于模型的测试; 分布式系统; 场景; 正则表达式

中图法分类号: TP311

中文引用格式: 钱巨, 王寅, 程浩, 韦正现. 基于场景模型的DDS架构一体化舰船任务系统测试. 软件学报, 2022, 33(5): 1711–1735. http://www.jos.org.cn/1000-9825/6557.htm

英文引用格式: Qian J, Wang Y, Cheng H, Wei ZX. Scenario Model Based Testing of Integrated DDS-based Naval Mission Systems. Ruan Jian Xue Bao/Journal of Software, 2022, 33(5): 1711–1735 (in Chinese). http://www.jos.org.cn/1000-9825/6557.htm

Scenario Model Based Testing of Integrated DDS-based Naval Mission Systems

QIAN Ju[1], WANG Yin[1], CHENG Hao[1], WEI Zheng-Xian[2]

[1](College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

[2](Systems Engineering Research Institute, Beijing 100036, China)

Abstract: Modern integrated naval mission systems (NMS) built on data-distribution service (DDS) have special characteristics in development, structure, and application which, in combination, make their testing challenging. Model-based testing (MBT) is considered a promising technique for testing such systems. However, for NMS-like systems under test, due to their high complexity and cooperative ways of development, traditional MBT techniques requiring a complete model of the system internals are difficult to be used. This paper presents a scenario-based MBT approach for NMS-like systems. The approach builds scenario models to express the interaction scenarios in a DDS-based system from the external perspective. A scenario model uses an extended form of regular expression to model interaction sequences and uses basic data element restrictions (e.g., ranges and enumerations), constraints, and calculation functions to model interaction data. It can express the interaction processes in an abstract, convenient, and relatively comprehensive way. On the models, algorithms are proposed to generate directly executable test cases for testing. Experiments on a real NMS show that the approach can be used to test 21 kinds of common risky scenarios identified from historical failures reported during the development of a family of NMS. This indicates that the approach might be helpful for testing NMS-like DDS-based industrial systems.

Key words: model-based testing (MBT); distributed system; scenario; regular expression

---

* 基金项目: 国防基础科研计划 (JCKY2016206B001, JCKY2014206C002); 中央高校基本科研业务费专项资金 (NS2021066)
本文由“领域软件工程”专题特约编辑汤恩义副教授、江贺教授、陈俊洁副教授、李必信教授以及唐滨副教授推荐.
收稿时间: 2021-08-10; 修改时间: 2021-10-09; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

舰船任务系统是一种大型关键工业系统, 控制舰艇上通信、作战等各种任务, 需谨慎测试以确保其质量. 在DDG-1000 驱逐舰、CVN 78 航母等现代舰船中, 任务系统常封装为一系列分布式构件, 以开放式架构一体化集成于全舰计算环境中[1]. 实际部署前, 需要在分布式构件[2,3]的接口进行大量测试以尽可能多发现故障[4,5]. 因为一旦部署上线, 系统通常只能通过有限的外部物理接口来访问, 检测和修复错误的成本将极为高昂.

自动生成经过构件接口的消息序列来触发和检验舰船任务系统的执行, 可以提高测试效率. 尽管如此, 新型一体化舰船任务系统常常以发布-订阅通信模式的数据分发服务 (data-distribution service, DDS)[6]为基础架构来实现动态、松耦合的集成. 如表 1 所列, 系统在研发模式、结构和应用方面有许多特殊特点, 使得现有测试技术[7]难以完全适用. 由于特点 1, 白盒测试生成难以用于舰船任务系统整体测试. 在黑盒测试中, 基于模型的测试 (model-based testing, MBT)[8−10]是一种广泛有效的测试技术, 为被测系统构建模型, 并从模型导出测试用例. 然而, 许多MBT 方法需要深入理解被测系统如何工作, 以便创建完整的系统内部模型[8,9]. 在实践中, 如表 1 特点 3 所述, 对于测试人员而言, 获取足够的知识以为舰船任务系统构建此类模型相当具有挑战. 要求开发者提供详细工作模型是一种选择, 但是, 由于系统通常由许多不同机构联合研制 (特点 1), 该方法实践中很难操作.

表 1　基于 DDS 的一体化舰船任务系统的一些特征

类别	特征
研发模式	1. 系统通常由许多不同机构联合开发, 由于商业秘密等原因, 总体测试人员很难获得系统源代码
结构	2. 系统事件驱动, 采用无连接DDS通信. DDS消息有接口描述, 为测试数据生成提供了参照
应用	3. 系统极为庞大、复杂, 测试人员很难清楚了解系统是如何实现的
	4. 系统具有一些非确定性, 一个输出在不同的环境下可能会不同, 即使是由相同的输入序列触发
	5. 系统有许多已知失效模式, 例如与时间、重复、并发相关的失效, 这些都需仔细测试

对于舰船任务系统这类大型工业系统, 更可行的 MBT 测试方法是使用从外部视角构建的基于场景的模型来生成测试用例[11−17]. 场景只关注系统的部分行为, 且许多测试人员能够从外部输入/输出角度描述场景如何工作. 因此, 此类场景模型更容易获得. 通过控制场景模型的抽象粒度, 可以用单个场景模型来表达一簇系统执行, 仅从少量场景模型就能生成众多测试用例.

在基于场景的方法中, 一种选择是使用图形化模型, 如用例图[13]、UML 顺序图[14,15]、活动图[18]、消息序列图[16]等来表达场景中的交互[17]. 一些工作还使用文本化的正则表达式 (RE) 来描述交互场景[19−25]. 正则表达式为开发人员所熟知, 其抽象表达能力等价于有限状态机. 与图形化模型相比, 正则表达式可以通过文本编辑器轻松输入, 这使其成为舰船任务系统测试中快速创建和编辑大量场景模型的良好选择.

尽管便捷易用, 但现有基于正则表达式的场景建模方法仍存在一些问题, 阻碍了其在真实舰船任务系统中的应用. 首先, 已有正则表达式的表达能力仍需提升. 在舰船任务系统工作场景中, 存在许多可变选择. 这些选择虽然也可以通过正则表达式的选择运算符 ('|') 来表示, 但当存在较多候选, 且不同选择相互关联时, 正则表达式模型可能会太过复杂. 现有正则表达式也缺乏灵活的机制来表达存在复杂顺序或并发出现次数限制的序列, 这影响了其在稳定性、性能等问题测试中的使用. 除此以外, 正则表达式主要用于对交互序列的结构进行建模, 为生成完整可执行的测试用例, 还需要一些补充机制来建模交互数据, 特别是那些存在复杂限制的数据.

针对上述问题, 本文扩展了正则表达式的语法, 以此为基础, 提出一种可用于类舰船任务系统的基于场景的MBT 方法. 为构建该方法, 论文首先从一族舰船任务系统的历史失效记录总结了 21 种常见风险场景. 从这些风险场景出发, 确定了测试过程对场景模型和测试生成算法的需求, 以使所提出方法能够支持新舰船任务系统或其新版本中风险场景的测试. 基于具体需求, 进一步提出一种场景建模技术, 结合扩展正则表达式 (ERE)、基本数据元素限制 (如范围和枚举)、一阶逻辑约束和计算函数, 从外部 DDS 消息的角度建模系统执行场景. 在场景模型上, 论文提出了自动生成包含交互序列和交互数据的可执行测试用例的算法. 整个测试方法考虑了表 1 中的所有特征. 在某真实舰船任务系统上开展的实验表明, 该方法能够支持所总结出的常见类型风险场景的建模. 其生成的测试用例成功揭示了 23 个根据待测舰船任务系统家族故障历史注入到系统中的失效. 本文的方法并不局限于舰船

任务系统, 其提出原因来源于此, 但也适用于其他具有类似结构的基于 DDS 的分布式系统.

本文的主要创新点包括:

(1) 提出了一种基于扩展正则表达式的场景模型, 能够便捷且相对完整地表达舰船任务系统等 DDS 架构分布式系统上的交互过程. 模型引入变量化拓展的事件模式来表达可变事件选择, 引入受约束的量词来描述关于序列顺序和并发出现的限制, 能够支持灵活的交互序列建模. 模型还定义基于基本数据元素限制、事件内和事件间约束、以及计算函数的事件数据模式来建模交互数据, 定义错误模式来支持健壮性测试. 该模型支持无连接通信 (表 1 特征 2), 除了功能性问题外, 还可以用于测试时间和性能相关的问题 (表 1 特征 5).

(2) 提出了一种从场景模型生成可执行测试用例的方法. 方法将状态机遍历、随机/边界数据生成、约束求解、函数计算相结合, 能够生成可直接在基于 DDS 的舰船任务系统上执行的测试用例. 测试生成主要采用离线方式, 但也支持在线测试数据补充, 以应对分布式系统中一些可能存在的不确定性 (表 1 特征 4).

1 背景知识

DDS 是一种用于可伸缩、实时、可靠、高性能发布-订阅模式通信的中间件标准[6], 广泛应用于航空航天、国防、智能电网、机器人等诸多工业领域[26]. 在基于 DDS 的通信中, 发布者创建一个主题并在该主题下发布消息. DDS 将消息传递给主题订阅者, 以支持发布者和订阅者之间的数据交换 (图 1). 主题由名称、类型和一组服务质量 (QoS) 策略定义. 类型的描述包含在 DDS 接口对应的.idl 文件中. DDS 还支持使用域来区分通信范围.

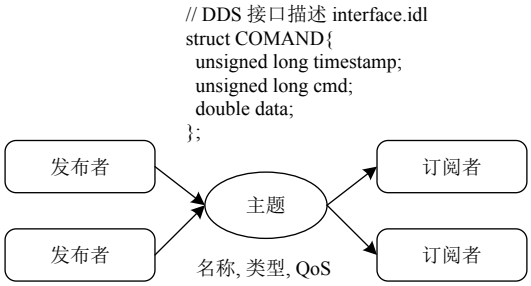


图 1 DDS 通信示例

图 2 展示了一个基于 DDS 的舰船任务系统的结构示意图. 该系统由一组通过 DDS 协议通信的软硬件构件组成. 构件行为主要由 DDS 消息触发. 在接收到传入消息时, 构件可以向外部发布或不发布 DDS 消息. 发布的消息可以被多个订阅者接收. 某些软件构件可能部署在同一计算机上, 共享并竞争相同的计算资源.

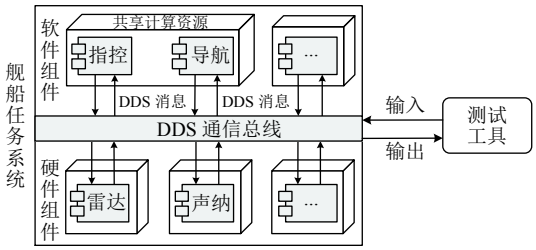


图 2 舰船任务系统结构示意

2 相关工作

2.1 基于正则表达式的测试

本文以正则表达式作为测试生成的一种直接基础, 而不是某种辅助性匹配技术. 在此方面, 除了构造字符串,

正则表达式也常被用来生成交互序列[19−25]. 正则表达式易于理解, 便于编辑, 也具备用一个表达式描述一组不同序列的抽象能力. 在基于正则表达式的交互序列生成中, 文献 [12,20−23] 用正则表达式表达操作、方法调用、GUI 事件等的出现模式来生成测试集. 为扩展表达能力, 文献 [19,24,25] 还在序列建模中应用形如 $a \,||\, b$ 的运算符来表达 a、b 序列的并发出现. Belli 等在正则表达式中添加了层次模型来表达复杂系统行为模式[24].

正则表达式用在系统建模中的一个问题是缺乏灵活机制来表达复杂可变选择. 从范围{a, b}进行选择通常通过运算 $a \mid b$ 表达. 当有很多候选选项, 且不同选择存在相关性时 (例如多个位置总是要求作出同样的选择), 表达式很容易变得过于复杂. 此外, 普通正则表达式中的限定量词只能是常量, 例如表达式 $a^m b^n$ 中的 m 和 n 为常量. 很难表达 a 和 b 的出现次数大于 10 这样的限制, 即 $m + n$>10. 这导致正则表达式不容易建模具有复杂重复或并发结构的序列, 不便用于一些稳定性和性能相关的测试. 为解决这些问题, 本文使用事件模式扩展正则表达式语法, 来使其更易于事件选择的表达; 允许限定量词为约束变量, 以更好地控制正则表达式中的重复和并发.

与正则表达式类似, 其他文本化的表达, 如线性时序逻辑 (LTL) 公式[9]和上下文无关文法 (CFG)[27]也可用于测试生成. 然而, LTL 需要更多的数学基础来掌握. CFG 比正则表达式表达能力更强, 但从 CFG 表达式很难直接看出序列结构. 本文认为相较于 LTL 和 CFG, 正则表达式具有更为易用的优势.

2.2 交互数据建模

除了建模交互序列的结构, 还需要机制来描述进而获得交互数据, 从而使生成的交互序列具有实际的可执行性. 一种生成交互数据的方法是提供一个系统内部行为模型, 然后使用符号执行等技术在模型上模拟交互过程来推演交互数据[12,20]. 这种方法较难用在舰船任务系统的测试中, 因为如引言所讨论, 对于复杂工业系统而言, 系统内部行为模型常常难以获取. 另一种方法是给定候选参数数据池, 然后使用组合测试等技术从池中选择值来构造交互数据[21−23]. 定义数据池相对繁琐, 并且在交互序列中可能存在对不同参数的约束, 这使得不容易通过组合测试获得合理的组合数据. 第 3 种方法是使用像 OCL[28,29]这样的约束语言来直接建模交互数据, 求解约束以生成具体取值. 然而, OCL 仅表达单个事件上的取值约束, 不支持表达交互序列中前后不同事件上取值之间的关联限制. 文献 [30,31] 在 OCL 中添加了事件间约束, 但这些约束用于限制事件之间的时序, 而不是表达不同事件中数据之间的联系. 实践中, 一些复杂的取值限制难以用可求解的约束公式表示, 这也限制了约束对测试数据生成的作用.

如何对工业系统交互序列中的各种交互数据进行建模和生成仍然具有挑战. 本文引入一种组合使用范围、枚举、字符串模式、事件内约束、事件间约束以及计算函数的方法, 来实现交互序列中的数据建模. 该方法支持生成真实工业系统中经常涉及的各种数据. 除了建模有效测试输入, 本文还允许在场景模型中添加错误模式以生成用于健壮性测试的无效输入.

2.3 DDS 分布式系统测试

关于基于 DDS 的分布式系统测试, Michlmayr 等人[32]提出了一个用于测试发布-订阅程序的单元测试框架, 但该工作的重点是测试框架, 而不是测试生成算法. Piel 等人[4]提出一种发布-订阅系统集成测试方法, 但他们更关注方法论而不是具体的测试技术. Cotroneo 等人[33]和 Grace 等人[34]提出了支持用 Petri 网和有限状态机来验证 DDS 架构系统性能和互操作性的技术. 他们使用模型来检查在测试期间是否到达了某些状态, 而不是进行测试生成.

DDS 构件在一些方面与 Web 服务相接近. 它们都有接口描述可用于测试生成. 但 DDS 使用无连接通信. 输入和输出并不存在一一对应关系. 这与基于 SOAP 的服务或 RESTful 微服务非常不同. 此外, 在诸如舰船任务系统的 DDS 架构系统中, 软件通常是事件驱动, 也没有 BPEL 等语言来编排任务流程. 因此, 现有针对 Web 服务的测试技术[35]并不直接适用于基于 DDS 的分布式系统, 需要为基于 DDS 的分布式系统研究新的测试技术.

3 测试方法概述

本文测试基于 DDS 的构件化分布式系统. 被测对象可以是包含所有构件的整个系统, 也可以是仅含部分构件的子系统. 论文构建测试工具来模拟外部环境和不属于被测范围的构件的行为. 测试用例主要是 DDS 消息序列.

测试工具将 DDS 消息发送给被测对象以触发其执行, 订阅 DDS 通信总线上的消息确定被测对象行为是否符合预期 (图 2). 称测试工具发布给被测系统的 DDS 消息为输入消息, 称从被测系统订阅到的消息为输出消息. 除 DDS 消息之外, 测试工具还支持弹出对话框来提示完成交互序列所需的少量手工操作 (如按下硬件按钮).

3.1 测试过程

所提出的测试方法包括 4 个主要步骤, 如图 3 所示.

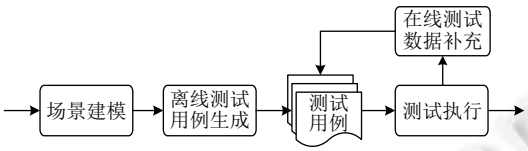


图 3 测试过程

(1) 场景建模. 测试人员找出应仔细测试的系统风险执行场景. 然后, 对这些场景进行建模.

(2) 离线测试用例生成. 从每个场景模型生成测试用例. 测试用例是场景的实例化, 激发系统一种行为表现.

(3) 测试执行. 测试工具执行生成的测试用例. 用例中的预期输出将用于帮助确认测试执行是否成功.

(4) 在线测试数据补充. 在舰船任务系统中可能存在非确定性. 给定的输入序列下, 可能无法事先确定输出内容, 导致依赖交互序列中前置输出的后续输入难以合理地生成取值. 例如, 输入消息中的安全令牌通常需要与之前根据环境动态生成并在某输出中返回的内容相同. 在获得输出的安全令牌前, 无法为输入生成令牌内容. 为应对此种情况, 本文支持在线测试数据生成. 在离线生成时, 可以部分地生成测试用例, 保留一些输入数据到运行时确定. 执行测试用例时, 测试生成算法在线补充测试数据, 以使生成的输入可以与运行时环境一致.

3.2 对场景建模语言和测试生成算法的要求

本文的目标是希望所提出的方法能够支持舰船任务系统中常见风险场景的建模和测试. 为理清对场景建模语言和测试生成算法的需求, 论文首先从一族基于 DDS 的舰船任务系统开发过程中报告的 400 余个可访问历史失效记录中总结出了 21 种常见的失效模式 (表 2). 这些模式可分为 3 类: 无法接受正确的输入序列, 无法容错无效输入并因此进入意料之外的状态, 以及无法容错问题环境. 在分布式系统中可能会发生各种异常, 后两种类型的存在是因为任务系统通常需要在构件接口级实现防御编程, 并避免在遇到异常时进入崩溃、停止响应等不可恢复状态. 失效模式对应于在开发新任务系统或其版本时应该仔细测试的危险场景.

表 3 列出了对场景建模语言和测试生成算法的详细需求. 一些需求来自于失效模式, 而一些则来自舰船任务系统自身特性. 后续的场景模型构建和测试生成算法设计将以这些需求为出发点, 提出解决途径来满足需求.

4 场景建模

本文在输入/输出级建模场景. 场景模型结构如图 4 所示, 表达一组由特定输入和预期输出构成的交互序列. 序列中每个元素称为一个事件. 输入事件将由算法生成, 输出事件用于帮助校验测试结果. 模型不仅可以表达合理的交互序列, 还可以描述包含无效输入数据或无效事件顺序的异常序列, 用于测试被测系统的健壮性.

定义 1. 场景模型. 场景模型是给定待测系统上一组交互序列的输入/输出级描述, 由事件序列模式、事件数据模式和错误模式几部分构成.

4.1 事件序列建模

表 3 中的 R1–R5 表达了对事件序列的建模需求, 这些需求很难直接用现有正则表达式满足. 在文本化表达、便捷而又不失直观地创建模型这一理念驱动下, 本文提出基于扩展正则表达式的事件序列模式来建模一个场景中的事件序列. 其中, 事件可以是 DDS 输入、预期 DDS 输出、手工动作或时间延迟. 一些系统行为无法通过编程方式激活, 因此需要手工动作. 还需要特殊的时间延迟来建模场景中与时间相关的行为. 各个 DDS 消息和手工动作的类型 (包括名称、参数、DDS 主题和域等) 在场景建模前定义.

表 2 基于 DDS 的舰船任务系统中常见失效模式

类别	子类别	模式编号	描述
正常输入序列下的失效	普通任务流	FP1	一个任务流程无法在正确的输入DDS消息或手工动作序列下完成
	批量输入	FP2	不能正常处理大量连续输入消息
		FP3	不能正常处理大量并行输入消息
	非寻常时间间隔	FP4	任务流由于输入消息之间的时间间隔太长或太短而失败.
	复杂输入数据	FP5	包含大数据块(例如大数组)的输入报文导致内存耗尽、缓冲区溢出等失效
		FP6	包含可变长度数据(例如序列类型数据)的输入消息导致系统异常
		FP7	具有复杂嵌套数据类型的输入消息导致系统无法工作
		FP8	带有浮点数成员的输入消息导致系统无法工作
		FP9	带有复杂对齐数据成员的输入消息导致系统无法工作
系统无法容错无效输入而出现异常	时序错误的报文	FP10	系统无法容错顺序异常的报文序列
		FP11	系统无法容错报文丢失
		FP12	系统无法容错报文重复
	异常的周期	FP13	系统无法容错频率过高或过短的异常周期性输入
	异常的报文数据	FP14	系统无法容错含越界数据项的报文
		FP15	系统无法容错含未初始化数据项的报文
		FP16	无法容错单个输入消息的数据成员违反取值约束, 例如, 宽度大于长度的潜艇
		FP17	无法容错不同消息的数据成员间违反取值约束, 如时间戳在消息序列中错误地减小
	错误的DDS设置	FP18	系统无法容错主题设置错误的DDS消息, 例如空主题、冲突主题或未初始化主题
		FP19	系统无法容错域设置错误的DDS消息, 如域设置不一致
无法容错问题平台环境	停止响应	FP20	在异常的环境下, 如部分构件丢失连接或崩溃, 整个系统停止响应
	无法恢复	FP21	系统无法从异常的环境(如断网、硬件崩溃、CPU/内存/网络资源耗尽等)中恢复

表 3 对场景模型和测试生成算法的需求

类别	需求编码	要求	原因	来源	解决途径
事件序列建模	R1	表达无连接DDS通信	无连接DDS通信下输入和输出不一一对应	系统特征	ERE 事件序列
	R2	表达手工动作	一些失效场景涉及少量手工动作	失效模式	ERE 事件模式
	R3	抽象类似的任务流	舰船任务系统中的许多任务流类似	系统特征	ERE 变量机制
	R4	表达时间相关的行为	许多失效场景与时间有关	失效模式	ERE 事件模式
	R5	表达复杂重复和并发行为	许多失效因顺序或并行消息重复造成	失效模式	ERE 事件序列
事件数据建模	R6	对单个数据元素上的取值限制进行建模	DDS消息中的数据元素可能存在取值限制, 例如, 取值范围	系统特征 失效模式	基本数据元素限制
	R7	对单个消息中数据之间的关联性限制进行建模	单个DDS消息中的不同数据元素之间可能存在关联性取值限制	系统特征 失效模式	事件内约束限制 事件内计算限制
	R8	对不同消息中数据之间的关联性限制进行建模	不同DDS消息中的数据元素之间可能存在关联性取值限制	系统特征 失效模式	事件间约束限制 事件间计算限制
测试生成算法	R9	建模和处理非确定性	输入可能依赖于内容不确定的输出	系统特征	在线测试生成
	R10	建模并生成异常场景	需要在异常条件下评估系统鲁棒性	失效模式	错误植入

定义 2. 事件序列模式. 事件序列模式是建立在事件模式上的扩展正则表达式 (ERE), 对场景中事件之间的顺序进行建模.

事件序列模式中使用事件模式从预定义的池中选择固定事件或可变事件构成其序列, 使用正则表达式运算符和量词对序列结构进行描述. 以下是一个在舰船任务系统中建模设备初始化场景的事件序列模式示例.

$$\langle \text{i1 : dev1\_init} \rangle (\langle \text{i2 : dev1\_query} \rangle \langle \text{o3 : dev1\_status} \rangle)\{1,5\} \quad (1)$$

该示例中, <i1: dev1_init>、<i2: dev1_query> 和<o3: dev1_status> 是 3 个事件模式, 每个对应正则表达式中一个符号.i1、i2、o3 是事件标识符. 前缀 i 和 o 分别表示事件是一个输入或输出 DDS 消息. 一个输入消息后可以跟随零条、一条或多条输出消息. 其中 dev1_init、dev1_query、dev1_status 为 DDS 消息名, 在事件序列模式中的不同位置可重复出现. 每个名称与一种 DDS 消息类型、主题和域绑定. 输出事件<o3: dev1_status> 用于校验设备初始化是否成功. 整个事件序列模式表达了一个由单次 i1 和 1–5 次重复的 i2 和 o3 事件组成的设备初始化过程.

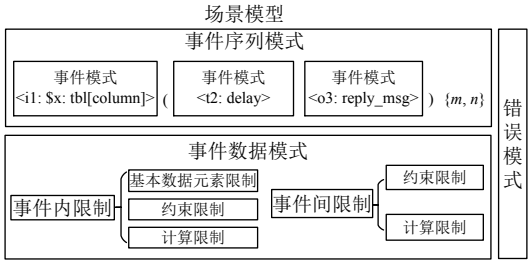


图 4 场景模型的结构

事件序列模式的具体语法如下.

event_seq ::= event | ‘(’ event_seq ‘)’ | event_seq event_seq | event_seq ‘|’ event_seq
| event_seq ‘||’ event_seq | ‘(’ event_seq ‘)’ quantifier

event ::= ‘<’ eid ‘: ’ content ‘>’ eid ::= prefix Nat prefix ::= ‘i’ | ‘o’ | ‘m’ | ‘t’

quantifier ::= ‘{’ [‘//’] num [‘, ’ num] ‘}’ | common_RE_quantifier num ::= variable | Nat

其中, event_seq 代表事件序列模式. event 为标识单个事件的事件模式, 其中 eid 是事件标识符, 由{i, o, m, t}中的一个前缀字符和一个自然数 (Nat) 组成, 前缀标记事件类型.content 保存事件的信息, 格式由事件类型决定 (见表 4). 在不同的事件之间可以有连接、选择 (' | ') 和并发 (' || ') 关系. 例如, A || B 表示事件 A 和事件 B 并行发生.

表 4 事件模式

前缀	事件类型	内容content	描述
i	输入DDS消息	(message | message_variable) [‘, ’ period]	message: 固定DDS消息名; message_variable: DDS消息变量; period: 供订阅者接收和处理DDS消息的短暂停止时间; timeout: 等待特定输出的最长超时时间.
o	输出DDS消息	(message | message_variable) [‘, ’ period [‘, ’ timeout]]	
m	手工操作	action | action_variable	action: 固定手工操作; action_variable: 手工操作的变量.
t	时间延迟	variable | Nat	变量或常量时间值(ms), 如<t1: delay>, <t1: 1000>.

量词 quantifier 表达事件的重复. ERE 扩展了普通正则表达式中的量词语法, 允许量词中出现约束变量, 以表达存在复杂限制的事件出现. 在该语法下, 通过表达式 A {x} B {y}和约束 x + y = 10 很容易表达事件 A 和事件 B 的出现次数相关, 且它们的总出现次数为 10. 带有符号‘//’的量词用于描述并发负载. 例如, A {//10}表示事件 A 并发发生 10 次. 这些并发负载可用于性能测试.

在事件模式中, 为简化选择、特别是相互关联选择的表达, 本文在 DDS 消息和手工动作上引入了变量标记 $name: table[column]. 该标记允许从预定义的表中选择 DDS 消息或手工动作, 其中 name 是变量名, table 是绑定的候选表, column 为供选择值的表列. 当从表的第一列中选择值时, 可以省略 column. 两个事件可以共享相同的变量名和表名. 这种情况下, 该变量是一个场景级的全局变量, 意味着两个事件相关, 它们将始终从相同的表中选择相同的行. 变量名也可以省略. 标记$table[column] 表示变量是事件内局部变量,它将独立从候选表中选取值, 而与其他事件无关. 变量表示法为场景模型增加了灵活性. 它允许模型轻松地抽象类似的任务流 (需求 R3).

对于公式 (1) 中设备初始化的事件序列模式, 一个用变量替换固定 DDS 消息的更一般化形式为:

$$\langle i1:\$x:dev[init]\rangle(\langle i2:\$x:dev[query]\rangle\langle o3:\$x:dev[status]\rangle)\{k\} \tag{2}$$

其中, $x:dev[init], $x:dev[query], $x:dev[status] 分别表示从预定义表 dev(表 5) 的列 init, query, status 中选择 DDS 消息. 表 dev 每行对应一个不同的设备, 3 列分别对应给定设备的初始化、状态查询和状态响应消息. 事件 i1、i2 和 o3 共享同一个变量名 x, 因此这些事件相关, 它们是关于相同设备的消息 (来自相同的表行). 使用指定的表列, 可以很容易地为同一设备上的 3 个事件选择不同的消息. 公式 (2) 还将关于事件出现的量词{1, 5}推广为变量 k, 通过对 k 施加不同的约束, 可以设置对 i2、o3 出现次数的不同限制.

表 5 事件变量候选表的示例

#	init	query	status
1	dev1_init	dev1_query	dev1_status
2	dev2_init	dev2_ query	dev2_status

4.2 事件数据建模

事件序列模式仅对场景的框架进行建模. 为使事件序列可执行, 还需确定事件数据, 包括 DDS 消息的基本数据、人工动作的参数和具体的延迟时间. 以表 3 中的需求 R6–R8 为出发点, 本文提出事件数据模式来对事件数据进行建模. 对于 DDS 消息, DDS 的主题和域由事件序列模式确定. 在事件数据建模中, 只关注消息体. 如图 1 所示, 消息体就像 C 语言中的结构. 对于其他事件, 其参数也可视为结构体. 与结构类似, 事件数据由整数、浮点数、字符串等基本类型元素构成. 真实系统中, 这些基本元素的取值往往受到限制. 任意为元素设定取值可能会导致不合理的交互数据, 无法用于测试目标系统功能. 因此, 需要模型来描述对这些数据的限制. 在本文的方法中, 模型包含了事件内限制和事件间限制两种对于数据取值的描述机制.

4.2.1 事件内限制

事件内限制指单个事件上的数据取值要求, 无论事件处于哪个场景下. 考虑到仅用范围、约束等无法全面表达真实系统中的各种取值限制, 本文引入了计算限制的概念, 共支持对于事件数据的三种限制模式. 这些限制相当于事件上的取值不变式, 可在不同场景下复用.

- 基本数据元素限制: 限定基本类型数据的合理取值范围、可选枚举值、以及字符串的模式.
- 约束限制: 用 SMT 约束求解器[36]支持的一阶逻辑公式表达一个或多个数据元素上的取值约束. 在实际系统中, 限于业务逻辑或物理规则, 不同数据之间经常存在关联. 一些关联可以用约束来表达.
- 计算限制: 用计算函数来表达某些不能通过基本数据元素限制和一阶逻辑公式描述的复杂数据元素取值. 例如, 有些声纳数据很难通过约束条件来建模, 但可以通过模拟声纳行为的计算函数来限定.

基本数据元素限制与 XML Schema 中的含义相同. 本文使用 XML 中的.xsd 文件来存储此类限制. 例如, 图 5 展示了一个.xsd 文件, 它将 DDS 消息 DEVICE1_INIT 的宽度元素 width 限制在 [100, 200] 范围内. 其中还扩展了 XML Schema 来表示约束和计算限制 (为方便扩展、统一约束形式, 没有直接使用 XML 提供的 assert 约束机制).

约束由原子条件或其与或非构成. 原子条件是建立在约束变量和常数基础上的线性方程或不等式, 例如:

$$value(eventId/length) > value(eventId/width) \tag{3}$$

表示设备的长度总是大于宽度. 其中 value(eventId/length) 和 value(eventId/width) 是两个约束变量. value 是表示标识符取值的变量类型. eventId 用于区分不同事件, 对于事件内限制, 一般设为占位符'*'. /length 是到 DDS 消息中基本类型元素的访问路径.

计算限制使用预定义或用户定义的函数来计算某些事件数据的取值. 例如, 可以用预定义函数 select(query, query_arguments) 来查询过去从 DDS 总线上侦听的历史报文数据库中的数据. 通过 select 函数, 可以从 DDS 消息池中选择值来构造测试数据, 而不用仔细地对数据进行建模并应用算法进行数据生成. 在图 5 中, 计算限制:

$$value(*/area) = multiply(value(*/width), value(*/length)) \tag{4}$$

描述面积 area 是宽度和长度的乘积. 这样的面积值比较容易计算, 但不容易通过约束求解获得.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=…>
<xs:complexType name="DEVICE1_INIT">
 <xs:sequence>
    <xs:element name="width">
       <xs:simpleType>
          <xs:restriction base="xs:int">
            <xs:minInclusive value="100"/>
            <xs:maxInclusive value="200"/>
          </xs:restriction>
       </xs:simpleType>
    </xs:element>
    ……
 </xs:sequence>
 <constraints>
  <and name="shape_constraint">
   <constraint> <![CDATA[ value(*/length) > value(*/width) ]]>
   </constraint>
    <constraint> <![CDATA[ value(*/length) < 1500]]> </constraint>
  </and>
 </constraints>
 <calculation>
    <calculator> <![CDATA[ value(*/area) =
       multiply(value(*/width), value(*/length)) ]]> </calculator>
  </calculation>
</xs:complexType>
</xs:schema>
```

图 5　事件内限制描述 dev1_init.xsd

4.2.2　事件间限制

事件内限制只能表达单个事件上的取值要求, 无法描述前后不同事件间的关联. 为解决该问题, 本文提出了事件间限制的概念, 描述特定场景下不同事件间的取值要求. 事件间限制包括事件间约束限制和事件间计算限制, 作用在两个或更多顺序出现的事件上.

事件间约束从当前事件的角度, 表达该事件与交互序列中历史事件在数据上的相关性. 在事件间约束中, 对于每个如 value(eventId/accesspath) 的约束变量, 都需指定 eventId. eventId 可以是类似 i2[current] 的形式, 其中 i2 是事件序列模式中的事件标识符 (如公式 (1) 中的 i2), current 表示当前出现的 i2. 方括号中的部分也可以替换为 prev (表示事件 i2 的之前最近一次出现) 或-N(如-2, 表示 i2 的前第 N 次出现).

例如, 对于由多个 i2 所标识事件构成的连续 DDS 消息序列, 后一个消息中的时间戳值必须大于前一个消息的时间戳; 否则, 时间戳数据将不合理. 该约束可以表达为:

$$\text{value}(\text{i2}[\text{current}]/\text{time}) > \text{value}(\text{i2}[\text{prev}]/\text{time}) \tag{5}$$

如果没有事件间约束, 在生成多个事件的测试数据时, 很容易得到不合理的测试数据.

事件间计算限制表达当前事件的某些数据是从以前的事件数据通过计算获得. 其描述与事件内计算限制类似, 只是也需要指定 eventId. 例如, 公式 (6) 表示当前 i2 事件中的传感器取值是从 i2 的数据池中选择的值, 所选择的记录必须与前一个 i1 事件中的 mode 值相同. 在公式中, type(i2[current]) 表示当前 i2 事件的 DDS 结构名, 而 select() 函数执行数据库查询:

$$\text{value}(\text{i2}[\text{current}]/\text{sensor}) = \text{select}(\text{“sensor from \%s where mode = ‘\%s’”}, \text{type}(\text{i2}[\text{current}]), \text{value}(\text{i1}[\text{prev}]/\text{mode})) \tag{6}$$

4.3　错误建模

为了满足表 3 中的需求 R10, 本文使用错误模式对场景中发生的异常进行建模. 通过错误模式, 可以从有效的测试输入派生出特定无效测试输入, 测试舰船任务系统对于常见异常输入的健壮性.

定义 3. 错误模式. 错误模式是一个二元组<P, F>, 其中 P 指定错误发生的位置, F 指定错误类型.

错误位置 P 用 i1[first]/width 形式表示. i1 是事件标识符.first 关键字表示事件 i1 在整个场景中的第一次出现. i1 和 first 都可以用 any 来代替, 表示任何可能的事件. /width 是一个访问路径, 指定发生错误的数据元素. 如果错误不限于特定的数据元素, 则访问路径可以省略. 错误类型 F 可以是消息丢失、重复或乱序 (message_lost, message_duplicated, message_disorder)、事件中断 (interrupt[param])、数据溢出 (overflow)、未初始化数据

(uninitialized)、时间越界 (time_out_of_boundary)、违反约束 (constraint_violated, constraint_violated[inter])、空 DDS 主题 (empty_topic)、错误的 DDS 域 (wrong_domain) 等舰船任务系统中可能发生的异常情况. 错误类型也可以带有参数, 例如 interrupt[action] 表示插入由参数 action 配置的事件来中断事件流.

4.4 场景模型示例

一个完整的场景模型由 3 个部分组成: 基本设置、事件内限制和场景模型文件. 基本设置独立于特定场景, 包括 DDS 消息和手工动作的类型定义、变量候选值表等. 其中, DDS 消息抽取自 DDS 的接口描述.idl 文件, 手工动作由用户根据需要定义, 变量候选值表根据变量的概念内涵设置, 保存于 Excel 文件. 事件内限制存储在扩展的 XML Schema 文件中. 每个场景定义一个场景模型文件来保存特定于场景的信息. 图 6 展示了为第 4.1 节中介绍的设备初始化场景定义的示例场景模型文件. 该文件存储事件序列模式 (元素 event_sequence)、关于事件序列模式中量词变量的约束 (元素 sequence_constraints)、事件间约束 (元素 data_constraints)、事件间计算限制 (元素 calculators) 和错误模式 (元素 fault_pattern). 除了模型名称和事件序列模式, 其他部分为可选内容.

```
{  "name": "device initialization check",
   "event_sequence": "<i1: $x:dev[init ]>
          (<i2: $x:dev[query]> <o3: $x:dev[status]>){k}",
   "sequence_constraints":{
        "and":{
            "k_lower_bound": "k >= 1",
            "k_upper_bound": "k <= 5"
          }},
   "data_constraints":{
     "i2": {
        "and":{
          "i1_to_i2_time":
              "value(i2[current]/time) > value(i1[prev]/time)",
          "i2_to_i2_time":
              "value(i2[current]/time) > value(i2[prev]/time)"
     }}},
   "calculators":{
        "sensor": "value(i2[current]/sensor) = select(\"sensor
                    from %s where mode=\'%s\'\",
                    type(i2[current]), value(i1[prev]/mode))"
   },
   "fault_pattern": {
         "position": "i1[first]/width",
         "fault": "overflow"
   }}
```

图 6 场景模型文件的示例

5 测试用例生成

从场景模型可以生成测试用例. 生成过程包括 3 个关键步骤: (1) 生成事件序列; (2) 生成事件数据; (3) 错误植入, 如图 7 所示. 步骤 (1) 首先从场景模型中的事件序列模式构造出不含事件参数的事件序列. 步骤 (2) 融合约束求解、函数计算等技术来为序列中的每个事件构造其具体参数取值. 步骤 (3) 根据错误模式将异常注入到事件序列中. 本文定义了覆盖准则来指导如何在生成空间中进行高覆盖度取值. 为便于说明, 本章先假定生成空间中所有可选项都任意选取, 在第 5.1–5.3 节介绍每一生成步骤的基本方法, 然后在第 5.4 节介绍如何根据覆盖准则在候选值空间中做出选择, 以生成满足特定充分性的测试集.

5.1 生成事件序列

场景模型中的序列模式包含受约束量词变量、正则表达式、事件变量等多种抽象性实体, 本文组合约束求解、有限状态机遍历和表选择, 来将抽象实体实例化, 得到具体的事件序列. 算法关键步骤如图 8 所示.

第 1 步, 根据约束, 求解事件序列模式中的量词变量, 将其替换为具体值. 例如, 对于公式 (2) 中的模式, k 是个量词变量, 如图 6, 其约束为 ($k \geqslant 1 \wedge k \leqslant 5$). 通过求解约束, 可得 k 的一个值 2. 由此, 可将事件序列模式部分实

例化为:

$$< i1 : \$x : dev[init] > (< i2 : \$x : dev[query] >< o3 : \$x : dev[status] >)\{2\} \quad (7)$$

第 2 步, 通过状态机遍历, 从事件序列模式的扩展正则表达式, 实例化出一个抽象事件序列. 根据自动机理论, 一个匹配正则表达式的序列对应等价于正则表达式的有限状态机中一个从入口状态到终结状态的路径. 因此, 为生成事件序列, 首先将扩展正则表达式转化为一个普通正则表达式. 每个事件模式, 例如<i1: $x: dev(init)>, 将映射到一个 Unicode 字符. 公式 (7) 可以映射为 a(bc){2}. 然后, 通过在等价的有限状态机中遍历一条从入口状态到终止状态的路径, 可以从普通正则表达式生成一个字符串. 该字符串将被映射回抽象事件序列. 例如, 从正则表达式 a(bc){2}中, 可以得到一个字符串 abcbc. 然后该字符串可以被映射回:

$$< i1 : \$x : dev[init] >< i2 : \$x : dev[query] >< o3 : \$x : dev[status] >< i2 : \$x : dev[query] >< o3 : \$x : dev[status] > \quad (8)$$

所获得的事件序列仍然包含事件变量, 它被称为抽象事件序列.

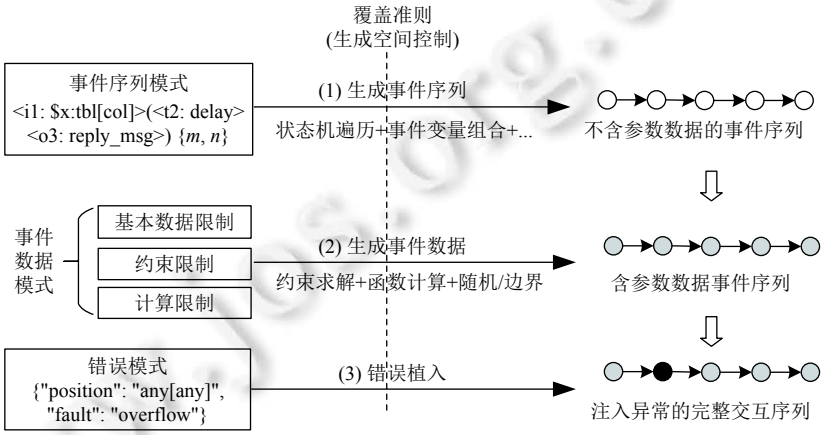


图 7 测试用例生成步骤

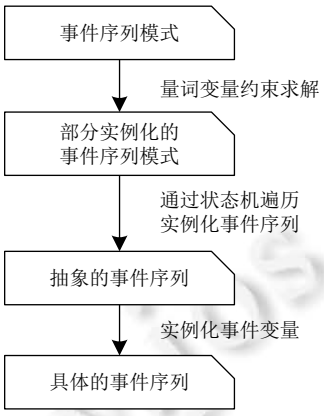


图 8 生成具体事件序列的步骤

在正则表达式的实例化中, 如果扩展正则表达式中存在并行结构, 将为每个并行部分分别生成事件序列, 然后使用并行流将其衔接. 例如, 给定一个具有并行结构 Y || Z 的扩展正则表达式 X (Y || Z). 首先, 视 Y || Z 为整体, 用符号 W 代替. 然后分别生成 XW、Y 和 Z 对应的序列. 设所生成的序列分别为 S_{XW}, S_Y, S_Z, 则最终为 X (Y || Z) 生成的序列是从 S_{XW} 中将 W 对应部分替换为 S_Y 和 S_Z 之间并行流的结果. 如果扩展正则表达式中存在类似(A){//k}的并行结构, 且并行负载 k 为固定值, 则为 A 生成 k 个序列, 对应于 (A){//k}的序列就是这 k 个序列的并

行流. 如果并行负载 k 是一个范围, 则首先从该范围选择一个值, 然后生成过程与固定的并行负载相同.

图 9 展示了从不同扩展正则表达式的生成的实例化序列示例.

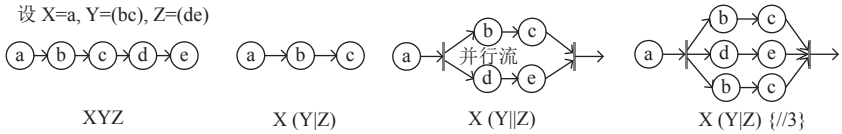


图 9 不同扩展正则表达式的实例化序列

下一步是在候选表中为事件变量确定具体值, 将包含事件变量的抽象事件序列最终转化为具体事件序列. 对于每个事件变量, 从绑定候选表的一行中选择一个具体的 DDS 消息或人工动作来替换该变量. 若为局部变量, 每个事件独立地为变量选择一个候选值. 若为全局变量, 具有相同名称的变量将绑定到候选表的同一行. 例如, 对于事件<i1: $x:dev[init]>、<i2: $x:dev[query]> 和<o3: $x:dev[status]>, 相同的变量名 x 意味着它们在候选表中共享同一行. 因此, 首先从候选表 dev(表 5) 中选择一行. 然后, 根据变量标记中的列名, 将不同事件中的变量映射到不同的具体 DDS 消息. 对于公式 (8) 中的抽象事件序列, 最终可以实例化得到以下具体事件序列:

$$< i1 : dev1\_init >< i2 : dev1\_query >< o3 : dev1\_status >< i2 : dev1\_query >< o3 : dev1\_status > \tag{9}$$

其中, 变量 x 被映射到候选表 dev 的第 1 行. 基于该具体的事件序列, 可生成可执行的测试用例.

5.2 生成事件数据

场景模型中约束公式和计算函数限制的组合使用, 为测试数据的生成带来了困难, 无法直接通过约束求解或者函数计算生成测试数据. 为此, 本文提出算法结合随机/边界数据生成、约束求解和函数计算来获得测试用例中的事件数据. 事件数据主要为离线生成. 为应对一些可能的不确定性, 部分测试数据也可能需要在线生成, 而不是在测试执行之前确定.

生成过程的第一步是找出事件序列中的所有基本数据元素及其关联关系. 将为事件序列中的每个事件出现分配一个类似 e_i 的标识符. 用类似/v 的访问路径表示某次事件出现中的一个基本类型元素, 则事件序列中的所有基本类型数据都可以用 e_i/v 的形式表示. 场景模型中的所有数据限制都将转化为对基本数据元素的限制.

5.2.1 构建数据元素依赖关系图

确定事件序列中的基本数据元素后, 将为它们分配标签{I, O, C, F}, 并根据场景模型计算它们之间的依赖关系. 标签分配规则如下.

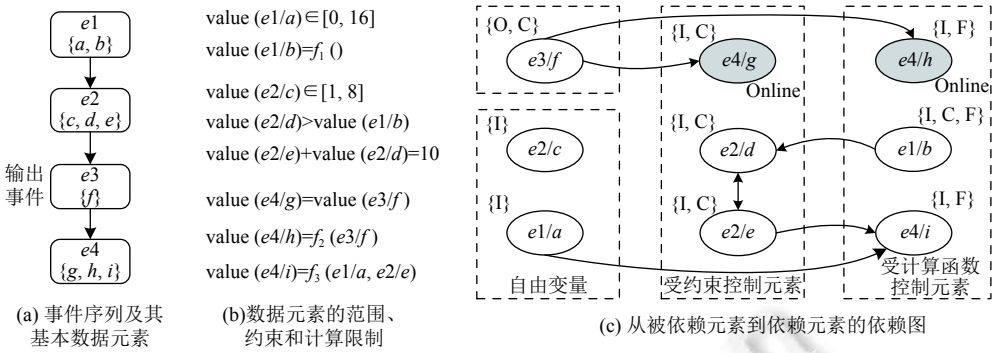
- 标签 I: 输入数据元素 (由算法生成其取值);
- 标签 O: 保存预期输出的数据元素 (其值由被测系统提供);
- 标签 C: 出现在约束限制中的数据元素;
- 标签 F: 出现在计算限制中的数据元素.

标签 I 和 O 互斥. 标签 C 和 F 可以附加在同一个基本数据元素上. 标签 F 只能附加在带标签 I 的输入上.

将生成一个依赖图来反映不同数据元素之间的依赖关系. 如果两个数据元素出现在同一事件内约束中, 将在这两个元素之间建立一个双向依赖边. 对于事件间约束, 从事件序列中每个事件出现的角度建立依赖边. 对于事件间约束中形如 value(i1[prev]/v) 的约束变量, 标识符 i1[prev] 将被解析为事件序列中具体的事件出现, i1[prev]/v 将被解析为具体数据元素. 然后, 在相关数据元素上建立双向依赖边. 对于包含在一个约束中的两个数据元素, 如果一个带标签 O 对应某输出, 或者带标签 F 对应某个由计算函数确定的值, 则依赖边只能从该元素开始到其他元素, 说明元素的值不是由约束决定. 对于计算限制, 如果一个数据元素被用作另一个元素的计算参数, 则建立一个从参数元素到被计算元素的依赖边. 图 10 展示了一个示例事件序列, 以及其中的数据元素、元素标签和这些元素间的依赖关系图.

5.2.2 离线生成测试数据

根据数据元素的标签和元素间依赖关系, 可以确定不同元素的测试数据生成方法和生成顺序.



(a) 事件序列及其基本数据元素　(b)数据元素的范围、约束和计算限制　(c) 从被依赖元素到依赖元素的依赖图

图 10　数据元素及其依赖图

- 带标签 I 不带标签 C、F 的元素 (自由变量): 首先采用随机或边界值方法为这些元素独立生成数据;
- 带标签 I、C 不带标签 F 的元素 (受约束控制): 作为整体, 通过 SMT 约束求解器生成取值;
- 带标签 I、F 不带标签 C 的元素 (受计算函数控制): 与约束无关, 约束求解后通过函数计算生成其取值;
- 带标签 C 和 F 的元素 (受计算函数和约束双重控制): 视计算函数的优先级高于约束, 在约束求解之前进行函数计算, 生成数据元素取值.

在对受计算函数控制的元素用函数计算方式生成数据时, 根据依赖图上的拓扑排序逐个处理数据元素. 在一个元素所依赖的所有其他元素被处理之后, 为其生成测试数据. 如果生成顺序上存在冲突, 则要求用户设法改变计算函数或数据限制表达方式, 更新场景模型以消解冲突.

对于图 10 中的例子, *e*1/*a* 和 *e*2/*c* 既不包含在任何约束中, 也不受计算函数的限制, 可视为自由变量. 将根据其数据范围、枚举和字符串模式限制, 使用随机或边界值法为其生成数据. *e*1/*b* 同时具有标签 C 和标签 F, 值由计算函数决定, 并通过约束影响其他数据元素的取值. 因此, 需要在约束求解前, 先通过计算函数 f_1 生成 *e1/b* 的值. 元素 *e2/d* 和 *e2/e* 包含在约束中, 且它们的值不由计算函数决定, 将通过约束求解来生成其数据. 在求解约束前, 约束所涉及的元素 *e1/b* 会被计算函数生成的值代替. 在约束求解过程中, *e2/d* 和 *e2/e* 上的所有数据范围限制、枚举限制、事件内与事件间约束将合并成一个大约束, 用于测试数据求解. *e4/i* 有一个 F 标签, 但没有 C 标签. 它应该在约束求解后生成. *e4/i* 的计算以受约束的元素 *e2/e* 为参数. 这也要求在约束求解后生成 *e4/i*.

5.2.3　在线补充测试数据

上节未处理依赖于输出 (带标签 O) 的输入元素. 对于这些输入, 由于所依赖对象的取值在运行时才可获知, 它们的数据很难离线生成. 本文使用一个在线算法为其生成数据. 在线生成前, 先识别依赖图上从输出元素可达的所有输入元素. 这些依赖于输出的元素将被附加 online 标记, 而不离线生成数据. 当执行一个测试用例时, 一旦获得一个输出, 将查找之后所有带 online 标记且不再从任何未获知的输出可达的输入元素. 这些元素的值当前已可确定, 将根据事件数据模式, 在线生成其数据. 在线测试数据的生成与离线几乎相同, 只是在运行时, 仅需要生成小部分未确定的输入数据. 生成后, 带有新获得数据的输入元素上的 online 标记将被删除.

在图 10 的例子中, 数据元素 *e4/g* 和 *e4/h*(灰色节点) 依赖于输出元素 *e3/f*, 在依赖图上从 *e3/f* 可达. 因此, 它们的数据将在线生成. 在得到输出 *e3/f* 后, 提取对 *e4/g* 的约束和 *e4/h* 的计算函数, 将已经确定的数据元素用它们的具体值替换, 并使用约束求解和函数计算生成 *e4/g* 和 *e4/h* 的值.

通过离线和在线测试数据生成, 可以得到完整且可执行的测试用例. 在这些用例中, 输入事件的内容由算法生成. 对于输出事件, 保留数据限制. 将利用这些限制来帮助校验运行时输出.

5.3　错误植入

在获得有效的事件序列后, 如果场景模型中存在指定的错误模式, 则根据错误模式将该有效序列转换为错误序列. 错误序列生成首先在事件序列中定位注入错误的目标位置. 如果位置以 any[any] 的形式表示, 则将错误注入到事件序列中任一个可能的事件出现中.

定位目标位置后, 根据错误类型实施注入. 对于消息乱序 (message_disorder), 交换目标位置和事件序列中的随机位置上的消息. 对于未初始化数据 (uninitialized), 在指定数据各字节设置对应内存状态 0xCC/0xCD 的值 (未初始化内存常出现这些值). 对于违反事件内约束 (constraint_violated) 和违反事件间约束 (constraint_violated [inter]), 取反指定事件上的事件内/事件间约束, 求解新的数据替换原事件数据. 对于事件中断 (interrupt[param]), 根据参数 param 将事件插入到目标位置. param 可以指定一个常量事件或一个变量事件, 用语法$x:table[column] 表示 (参见第 4.1 节). 其他错误的注入相对直白, 限于篇幅不再赘述.

5.4 生成测试集

本文的方法支持随机生成测试用例, 以及生成满足特定覆盖准则的测试集. 论文引入一组覆盖准则, 如表 6 所示. 这些准则涉及场景模型的不同方面, 可以以不同方式组合使用, 以确保所得测试集的充分性. 覆盖准则 RE-S, EV, ED-FR, ED-C, ET_R 和 FT_P 默认启用, 其他准则下覆盖内容更多, 但生成测试集也更大.

表 6 覆盖准则

关注方面	准则名称	覆盖要求
事件序列	RE-S	覆盖场景模型中扩展正则表达式对应的有限状态机中的状态
	RE-T	覆盖场景模型中扩展正则表达式对应的有限状态机中的状态变迁
	RE-QB	覆盖场景模型扩展正则表达式中量词变量的典型边界值
	EV	覆盖单个事件变量的所有可能选择
	EV-C	覆盖所有事件变量的每一个可能选择组合
事件数据	ED-FR	在事件数据中覆盖每个自由变量的至少一个随机值
	ED-FB	在事件数据中覆盖自由变量的边界值
	ED-C	实现关于事件内和事件间约束的条件覆盖
	ET_R	覆盖时间延迟事件中各时间变量的至少一个随机值
	ET-B	覆盖时间延迟事件中各时间变量的边界值
错误	FT_P	在生成错误时, 覆盖错误模式指定的所有可能的事件位置
	FT_D	在生成错误时, 覆盖错误模式指定的每个可能的数据元素
	FT_C	在生成错误时, 覆盖所有可能条件的否定

为了实现关于正则表达式的 RE-S 和 RE-T 覆盖, 当遍历一个扩展正则表达式对应的有限状态机来生成事件序列时, 首先进行随机遍历, 得到一组覆盖状态机中所有状态或状态迁移的路径. 然后, 将这些路径映射为事件序列以构造测试集. 为了实现 RE-QB 覆盖, 对于扩展正则表达式中的自由量词变量, 生成典型的边界值, 例如 0 和 1000 这样的大值, 来实例化量词变量. 对于有范围约束的量词变量, 根据指定的范围为其生成边界值.

一个实际的事件序列极少出现所有数据元素取值均在边界上的情况. 因此, 通常只为自由数据元素生成随机值. 如果指定了 ED-FB 或 ET-B 覆盖, 那么每次从事件序列的所有数据元素中选择一个数据元素, 为其生成边界值来实例化序列. 重复此过程, 直到所有数据元素都使用了一次边界值.

在求解约束生成事件数据时, 本文支持一种条件覆盖 (ED-C 覆盖). 对于约束 C 中的每个原子条件 c_a 中, 假设 $(C \wedge c_a)$ 和 $(C \wedge \neg c_a)$ 都可解, 如果测试集中满足 c_a 和满足 $\neg c_a$ 的测试数据都存在, 则称测试生成实现了约束条件 C 上的条件覆盖. 该条件覆盖可使生成的数据集覆盖满足约束 C 的不同情况. 为实现条件覆盖, 在求解约束 C 时, 将对每个原子条件 c_a 求解 $(C \wedge c_a)$ 和 $(C \wedge \neg c_a)$, 而不是依赖约束求解器自身的随机性来生成测试数据. 当求解扩展正则表达式中的受约束量词变量的取值来生成的事件序列时, 默认也会启用条件覆盖.

本文的测试集通过迭代器生成. 迭代器每总体前进一步生成一个新测试用例. 总体迭代由一组子迭代器实现, 每个子迭代器控制影响测试覆盖的一个方面. 共有 5 个子迭代器, I_{qv}, I_{re}, I_{ev}, I_{cd} 和 I_{bf}, 如图 11, 分别遍历正则表达式中量词变量的取值, 正则表达式对应有限状态机中的路径, 从事件变量到具体事件的映射, 约束控制的事件数据, 和注入到交互序列中的边界值与异常. 后一迭代器依赖前一迭代器所得结果进行测试构造. 将所有迭代器生成的数据组合并在组合过程中补充随机值等必要信息即可获得一个测试用例.

实现覆盖的测试集通过完整地遍历一轮迭代器获得. 遍历过程中, 从依赖源到被依赖的迭代器, 即图 11 从右向左, 每步推进一个子迭代器, 右侧迭代完迭代左侧. 当所有迭代器均移动到末尾时测试生成终止.

该基于迭代器的生成的一个优点是无需事先对每个影响因素都枚举出其全部取值, 所有数据按需构造, 即使一个覆盖准则下测试数据组合空间巨大, 也可以快速逐步构造用例, 并可随时终止生成.

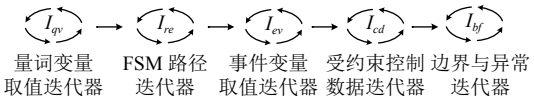


图 11 基于迭代器的测试生成

6 实验分析

本文在所研制的测试工具 DDSTest[37]中实现了论文提出的基于模型的测试方法. DDSTest 基于 Java 和 C++ 开发, 使用 C++发布和订阅 DDS 消息, 在 Java 端开展模型分析和测试生成, 利用 automaton 库[38]将正则表达式转换为有限状态机, 使用 Z3 SMT 约束求解器[36]求解约束, 支持 RTI[5]等多种广泛使用的 DDS 中间件.

为论证所提出的方法, 本文在一个真实的基于分布式构件的舰船任务系统上进行了实验. 实验用所提出的方法在构件接口级开展面向风险场景的测试. 被测任务系统部署于集成试验现场, 由 15 个以上不同供应商提供的主要构件通过 DDS 连接组成. 系统支持舰艇上导航、目标跟踪和武器控制等多种任务, 其庞大规模使得仅仅是集成和联调整个系统就耗费数百研发人员近一年时间. 系统的典型构件包括指控软件、导航软件、远程通信软件、武器管理软件等软构件, 以及声纳设备、雷达设备、时间同步设备、武器装置等硬构件. 软件构件部署在 5 个带显控的嵌入式计算机上. 硬件构件向软件构件提供信息, 并由软件控制. 各构件的大部分功能可由 DDS 消息触发. 系统中有 300 种以上 DDS 消息, 本实验涉及其中 62 种.

实验尝试回答以下几个问题.

RQ1: 场景建模语言是否具有测试表 2 中所列出常见失效模式所需的表达能力?

RQ2: 测试用例生成算法是否能够生成可揭示失效的测试用例集?

RQ3: 测试用例生成算法的时间效率如何?

RQ4: 本文方法相对人工测试与其他方法在测试构造代价、建模与测试生成能力等方面表现如何?

6.1 实验设置

从常见失效模式 (表 2), 可知新研制舰船任务系统极可能存在的许多风险场景, 但目前还不清楚问题是否确实存在, 又在哪些具体的交互序列下会出现失效. 在手工测试中, 测试人员对失效做出假设, 并编写测试用例, 试图揭示潜在故障. 在本文的方法下, 测试人员为有风险的场景编写场景模型, 使用场景模型来表达一组怀疑可能导致失效的交互序列, 并从模型自动生成批量测试用例.

为了检验场景建模语言的表达能力和测试用例生成算法的有效性, 实验首先根据常见失效模式, 从被测舰船任务系统上选取了一组研发人员建议的代表性风险场景作为建模对象. 由于整个任务系统庞大、复杂、专业性强, 即使是详细功能也短时间难以理清, 因此本文场景选择主要聚焦在武器控制、目标跟踪、设备管理 3 个方面. 研究的场景如表 7 所示. 这些场景或者在待测舰船任务系统家族历史中曾经出现过失效, 或者在类似的其他子系统上出现过异常, 具有高风险. 例如, 场景 S1、S2、S3 虽然都对应正常输入下的失效 (失效模式 FP1), 但在它们之下曾多次出现缺陷, 是同类任务系统评估报告中多次提到的重点问题场景.

实验用所提出的建模语言为表 7 中的风险场景构建场景模型, 检查场景模型是否能够表达场景中的主要可变因素, 以及是否能够从场景模型生成可执行的测试用例来回答 RQ1. 如果场景建模语言能够成功描述场景中的各种可变因素, 表达生成可执行测试用例所需的各种信息, 则认为它具有良好的表达能力.

表 7 中的每个风险场景对应于一系列具体的交互序列, 但其中只有少数可能真正触发失效. 有效的测试用例生成算法应能够生成可揭示失效的交互序列. 实验将仅通过危险场景下特定输入才能触发的失效注入到目标任务

系统, 并检查生成的测试集是否能够揭示这些失效, 来回答 RQ2. 失效按照待测舰船任务系统家族历史版本整体集成时实际出现过的情况注入. 一个风险场景注入一个失效, 所注入失效如附录所示. 一些失效 (F1, F2, F3, F4, F6, F23) 曾原样发生在待测系统历史版本的相关构件中, 其他失效发生在舰船任务系统的其他子系统上, 为便于开展实验, 被迁移到本实验所关注的子系统上, 但失效机理相同.

表 7 实验研究的风险场景

序号	失效模式	风险场景	主要可变因素
S1	FP1	发射和暂停武器	不同的武器类型
S2	FP1	系统任务模式切换	不同任务模式切换方式
S3	FP1	发射过程中的武器类型切换	不同武器类型之间的切换
S4	FP2	多组连续出现的目标探测消息	不同数量和类型的顺序出现消息
S5	FP3	多组并行出现的目标探测消息	不同数量和类型的并行出现消息
S6	FP4	非寻常报文间隔的武器发射过程	不同武器类型, 报文之间不同的时间间隔
S7	FP5	目标探测子系统包含大数据块的输入消息	不同的消息类型和数据
S8	FP6	目标探测子系统包含可变长度数据的输入消息	不同的消息类型和数据
S9	FP7	武器跟踪控制中包含复杂嵌套数据类型的输入消息	不同的消息类型和数据
S10	FP8	武器跟踪控制中包含浮点数数据成员的输入消息	不同的消息类型和数据
S11	FP9	武器跟踪控制中具有对齐数据成员的输入消息	不同的消息类型和数据
S12	FP10	武器发射控制消息乱序	不同的武器类型, 不同的乱序情况
S13	FP11	武器发射中的丢失消息	不同的武器类型, 不同的丢失消息
S14	FP12	武器发射中消息重复	不同的武器类型, 不同的重复信息
S15	FP13	异常的目标探测消息时间周期	不同目标侦测设备的消息和时间周期
S16	FP14	实时目标探测消息中出现数据溢出	不同DDS消息, 不同有溢出数据元素
S17	FP15	实时目标探测消息中包含未初始化数据	不同DDS消息, 不同未初始化数据元素
S18	FP16	武器控制中未满足事件内约束的输入消息	不同的受约束控制消息
S19	FP17	周期出现的目标探测消息不满足事件间约束	不同消息类型, 不同未满足事件间约束
S20	FP18	设备管理中出现DDS主题异常	不同DDS消息中的主题异常
S21	FP19	设备管理中出现DDS域设置异常	不同DDS消息中的域设置异常
S22	FP20	向武器管理软件查询状态时, 武器设备丢失连接	不同的武器设备, 不同的连接丢失时间
S23	FP21	目标探测子系统设备崩溃并尝试从崩溃中恢复	不同设备、不同时间的崩溃

实验在 DDS 消息接口监听报文并手工构造匹配目标失效的错误以注入失效. 尽管在真实任务系统构件中注入失效是更好的选择, 但被测系统的真实构件是硬件或嵌入式软件, 且由不同供应商提供. 一旦部署完毕, 很难将错误重新注入到其中. 因此, 实验仅在真实构件对应的模拟器中实施失效注入, 通过修改模拟器源代码植入错误, 产生失效表现. 实验重点是论证所提出方法生成的测试输入是否可以触发失效. 最终触发的失效类型与论证目标并不十分相关. 因此, 尽管在真实系统中可能存在各种最终失效类型, 本实验注入的失效主要是系统崩溃和无响应两种. 如果观察到上述异常, 则认为失效可以由测试用例揭示. 在测试期间, 对于植入错误的构件, 应用被测系统提供的构件使能工具将真实构件离线, 并激活其模拟器作为替代; 其他构件仍使用真实部署版本.

为回答 RQ3, 实验收集了测试用例生成算法所消耗的时间, 以评估所提出算法的效率.

对于 RQ4, 实验一方面收集了场景建模的大致开销, 并与手工测试进行对比; 另一方面, 与已有基于正则表达式和基于 UML 场景模型的测试生成进行了逐场景对照分析, 展示本文方法在解决领域问题方面的优势.

6.2 实验结果

6.2.1 RQ1 (模型表达能力) 结果

本文为表 7 中的目标场景构建了 23 个场景模型. 表 8 列出了场景模型的一些信息. 限于篇幅, 表中仅给出了模型的事件序列模式和错误模式, 更多细节可查看文献 [37](出于敏感性原因, 模型已擦除一些非核心信息, 仅保

留关键结构). 表 9 展示了场景模型的一些特征. 该表中, 列“#相关构件”统计每个场景模型中涉及的相关构件数量, 其他列标注了模型中用到的描述机制. 如表 9 所示, 本文所有描述机制的引入都是为了对风险场景进行建模, 实验几乎用到了第 4 节提出的所有建模技术.

表 8 场景模型

模型	场景	事件序列模式	错误模式
M1	S1	<i1:$x:weapon[warm]> <o2: $x:weapon[status], to> <i3: $x:weapon[prepare]> <o4: $x:weapon[status], to> <i5: $x:weapon[launch]> <o6: $x: weapon[status], to> <i7: $x:weapon[suspend]> < o8: $x: weapon[status], to>	N/A
M2	S2	(<m1: AZC> | <m2: BZC> | <m3: CTC> | <m4: DKZ>) <m5: Login> (<m6: I> | <m7: W> | <m8: Z>) <m9: OK>	N/A
M3	S3	<i1: $w1:weapon[load_query]> <o1: $w1:weapon[load_info], to1> <i2: $w1:weapon[warm]> <o2: $w1:weapon[status], to2> <t1: ti1> <i3: $w1:weapon[prepare]> <o3: $w1:weapon[status], to2> <t2: ti2> <i4: $w1:weapon[suspend]> <o4: $w1:weapon[status], to2> <t3: ti2> <i5: $w2:weapon[load_query]> <o5: $w2:weapon[load_info], to1> <t4: ti3> <i6: $w2:weapon[warm]><o6: $w2:weapon[status], to2> <t5: ti2> <i7: $w2:weapon[prepare]> <o7: $w2:weapon[status], to2 >	N/A
M4	S4	(<i1:$detect_dev[realtime_info]> \| (<i2:$detect_dev[period_info]><t1:period>)){n}	N/A
M5	S5	(<i1:$detect_dev[realtime_info]> \|\| <i2:$detect_dev[realtime_info]>){//n}	N/A
M6	S6	<i1:$x:weapon[warm]> <o2: $x:weapon[status], to> <t1:ti> <i3: $x:weapon[prepare]><o4: $x:weapon[status], to> <t2:ti> <i5: $x:weapon[launch]> <o6: $x: weapon[status], to>	N/A
M7	S7	<i1: $big_block>	N/A
M8	S8	(<i1: $var_length>){k}	N/A
M9	S9	<i1: $nested>	N/A
M10	S10	<i1: $float> | <i2: $double>	N/A
M11	S11	<i1: $align>	N/A
M12	S12	<i1:$x:weapon[warm]> <t1:ti1> <i2: $x:weapon[prepare]> <t2:ti2> <i3: $x:weapon[launch]> <t3:ti2> <i4: $x:weapon[suspend]> <o5: $x: weapon[status], to>	{"position": "any[any]", "fault": "message_disorder"}
M13	S13		{"position": "any[any]", "fault": "message_lost"}
M14	S14		{"position": "any[any]", "fault":"message_duplicated"}
M15	S15	(<i1:$x:detect_dev[period_info]> <t1: $period>){n}	{"position": "any[any]", "fault": "time_out_of_boundary"}
M16	S16	<i1: $detect_dev[realtime_info]>	{"position": "any[any]", "fault": "overflow"}
M17	S17		{"position": "any[any]", "fault": "uninitialized"}
M18	S18	(<i1: $x:constrainted><t1: ti>){k}	{"position": "any[any]", "fault": "constraint_violated"}
M19	S19	(<i1:$x:detect_dev[period_info]> <t1: $period>){n}	{"position": "any[any]", "fault": "constraint_violated[inter]"}
M20	S20	<i1: ZTCX> <i2:$x:controlled_dev[command]> <o3: $x:controlled_dev[response]>	{"position": "any[any]", "fault": "empty_topic"}
M21	S21	<i1:$x:controlled_dev[command]> <o2: $x:controlled_dev[response]>	{"position": "any[any]", "fault": "wrong_domain"}
M22	S22	<o1:BJXX, to1> (<i1:ZCX> (<o2:BZT, to2> || <o3:BQCSL, to2>) <t1: ti>) {k}	{"position": "any[any]", "fault": "interrupt[$weapon[disconnect]]"}
M23	S23	(<i1:$x:controlled_dev[command]> <o2: $x:controlled_dev[response], to>){k}	{"position": "any[any]", "fault": "interrupt[$x:controlled_dev[reboot]]"}

所编写的场景模型可以表达目标场景中的可变因素. 例如, 图 12 给出了武器发射中消息重复异常场景 S14 的详细模型文件. 该模型包括事件序列模式 (元素 event_sequence)、事件间约束 (元素 data_constraints) 和错误模式 (元素 fault_pattern). 其对应的正常事件序列包含 4 条 DDS 输入消息, 用于编排武器发射任务流, 可用于测试武器控制器子系统. 模型使用如<i1:$x:weapon[warm]> 的事件变量表达所发射武器类型的可变性, 并使用错误位置 any[any] 来表达被重复消息的可变性. 待测任务系统在输入的 DDS 消息中使用特定的数据成员 order 来区分不同的命令, 如 warm、prepare、launch、suspend. 在不同命令中, 将 order 数据的不同位设置为 1. 不同 DDS 消息中数据成员 order 之间的关系可以建模为 value(i2[current]/order) = value(i1[prev]/order)*2 以表达位移. 数据成员 info 和 num 保存要发射的武器的类型和数量. 同一个任务流中的不同 DDS 消息必须具有相同的 info 数据. 当系统成功处理命令消息时, 输出结果中有一个关系 value(o5[current]/status) = value(i4[prev]/order) + 1, 输出在 order 数据上附加了一位标志. 模型使用错误模式{"position": "any[any]", "fault": "message_duplicate "}来创建可能在任何位置重复输入消息的异常, 以测试关于消息重复的容错性.

表 9 场景模型的特征

场景模型	#相关构件	事件内约束	事件间约束	时间延迟事件	人工动作事件	事件变量	计算限制	错误模式	场景模型	#相关构件	事件内约束	事件间约束	时间延迟事件	人工动作事件	事件变量	计算限制	错误模式
M1	4	–	√	√	–	√	–	–	M13	4	–	√	√	–	√	–	√
M2	4	–	–	–	√	–	–	–	M14	4	–	√	√	–	√	–	√
M3	3	–	√	√	–	√	–	–	M15	4	–	–	√	–	√	√	√
M4	3	–	–	√	–	√	–	–	M16	3	–	–	–	–	√	–	√
M5	5	–	–	–	–	√	–	–	M17	3	–	–	–	–	√	–	√
M6	4	–	√	√	–	√	–	–	M18	4	√	–	√	–	√	–	√
M7	4	–	–	–	–	√	–	–	M19	4	–	√	√	–	√	√	√
M8	4	√	–	–	–	√	–	–	M20	6	√	–	–	–	√	√	√
M9	6	√	–	–	–	√	–	–	M21	6	√	–	–	–	√	√	√
M10	5	√	–	–	–	√	–	–	M22	3	–	√	√	√	–	–	√
M11	5	√	–	–	–	√	–	–	M23	6	√	–	√	√	√	√	√
M12	4	–	√	√	–	√	–	√	–	–	–	–	–	–	–	–	–

```
{   "name": "S14",
    "event_sequence": "<i1:$x:weapon[warm]> <t1:ti1> <i2: $x:weapon[prepare]> <t2:ti2>
            <i3: $x:weapon[launch]> <t3:ti2> <i4: $x:weapon[suspend]> <o5: $x: weapon[status], to>",
    "data_constraints": {
      "ti1":  {  "&&": {  "lower": "time(ti1)>6000", "upper": "time(ti1)<8000"  }  },
      "ti2":  {  "&&": {  "lower": "time(ti2)>3000", "upper": "time(ti2)<5000"  }  },
      "to":   {  "&&": {  "lower": "time(to)>6000", "upper": "time(to)<8000"  }  },
      "i1":   {  "warm": "value(i1[current]/order)=2"  },
      "i2":   {  "&&":  {
                  "order": "value(i2[current]/order)=value(i1[prev]/order)*2",
                  "info": "value(i2[current]/info)=value(i1[prev]/info)",
                  "num": "value(i2[current]/num)=value(i1[prev]/num)"  }  },
      "i3":   {  "&&":  {
                  "order": "value(i3[current]/order)=value(i2[prev]/order)*2",
                  "info": "value(i3[current]/info)=value(i2[prev]/info)",
                  "num": "value(i3[current]/num)=value(i2[prev]/num)"  }  },
      "i4":   {  "&&":  {
                  "order": "value(i4[current]/order)=value(i3[prev]/order)*2",
                  "info": "value(i4[current]/info)=value(i3[prev]/info)",
                  "num": "value(i4[current]/num)=value(i3[prev]/num)"  }  },
      "o5":   {  "success": "value(o5[current]/status)=value(i4[prev]/order)+1"  }
    },
    "fault_pattern": {  "position": "any[any]",  "fault": "message_duplicated"  }
}
```

图 12 场景 S14 的模型文件

实验从表 8 的场景模型生成测试用例. 表 10 列出了不同场景下的测试用例生成结果. 该表中, 列“典型具体事件序列”给出了各场景模型下生成的典型具体化事件序列 (真实事件名略敏感, 这里使用了无意义的名称作为替代). 列“测试集大小”列出了在每个场景模型和相应覆盖准则下生成的测试用例数量. 列“平均事件数”展示了测试集中每个测试用例所含的平均事件数量. 实验在目标舰船任务系统上运行所生成的测试用例. 这些用例中的输入 DDS 消息可以有效地调动软硬件构件、驱动任务流程的执行. 因此, 测试用例是可执行的.

由于使用所提出的场景建模语言, 可以构建模型来表达表 7 风险场景中的主要可变因素, 从场景模型也能生成可执行的测试用例, 因此, 实验认为该建模语言具有良好表达能力, 可服务舰船任务系统风险场景的测试需要.

表 10 测试用例生成结果

场景模型	典型具体事件序列	测试集大小	平均事件数	单用例生成时间(s)	覆盖准则
M1	<i1: AZB> <o2: AZT, 5000> <i3: AYB> <o4: AZT, 5000> <i5: AFS> <o6: AZT, 8000> < i7: AZZ> <o8: AZT, 8000>	3	8	0.034	EV
M2	<m2: BZC> <m5: Login> <m8: Z> <m9: OK>	4	4	0.009	RE-T
M3	<i1: ZCX> <o1: BZZXX, 2000> <i2: BZB> <o2: BZT, 5000> <t1: 5000> <i3: BYB> <o3: BZT, 5000> <t2: 5000> <i4: BZZ> <o4: BZT, 5000> <t3: 5000> <i5: ZZCX> <o5: AZZXX, 5000> <t4: 5000> <i6: AZB> <o6: AZT, 5000> <t5: 5000> <i7: AYB> <o7: AZT, 5000>	9	19	0.038	EV-C
M4	(<i2: BTCXX><t1:1000>){1000}	91	1268	0.832	RE-QB
M5	(<i1:AZCXX> || <i2: BZCXX:>){//1000}	28	501	0.007	RE-QB
M6	<i1: CZB> <o2:CZT, 500> <t1: 20 000> <i3: CYB> <o4: CZT, 500> <t2: 20 000> <i5: CFS> <o6: CZT, 500>	9	8	0.033	ET-B
M7	<i1: BMBP>	5	1	0.039	EV
M8	<i1: QBW> <i1: RMXX>	985	2	0.006	ED-FB
M9	<i1: QBW>	22	1	0.011	EV
M10	<i1: DIXX>	55	1	0.012	RE-T
M11	<i1: STSJ>	54	1	0.008	EV
M12	<i1: AZB> <t1: 7000> <i4: AZZ> <t2: 4000> <i3: AFS> <t3: 4000> <i2: AYB> <o5: AZT>	12	8	0.012	FT_P
M13	<i1: AZB> <t1: 7000> <t2: 4000> <i3: AFS> <t3: 4000> <i4: AZZ> <o5: AZT>	12	7	0.010	FT_P
M14	<i1: AZB> <t1: 7000> <i2: AYB> <t2: 4000> <i3: AFS> <i3: AFS> <t3: 4000> <i4: AZZ> <o5: AZT>	12	9	0.011	FT_P
M15	(<i1: DIXX><t2: 20 000>){1000}	42	668	0.202	RE-QB, FT_D
M16	<i1: LTCXX>	99	1	0.008	FT_D
M17	<i1: RMXX>	99	1	0.007	FT_D
M18	(<i1: BSDXX> < t1: 2000>){2}	268	4	0.169	FT_C
M19	(<i1: DIIXX> <t1: 500>){2}	49	4	0.601	FT_C
M20	<i1: ZTCX> <i2: AFK> <o3: AZT>	14	3	0.287	FT_P
M21	<i1: STCX> <o2: ZSTSJ>	7	2	0.195	FT_P
M22	<o1: BJXX, 10 000> <i1: ZCX> (<o2: BZT, 4000> || <o3: BQCSL, 4000>) <t1: 5000> <m1: A_disconnect> <i1: ZCX> (<o4: BZT, 4000> || <o5: BQCSL, 4000>) <t1: 5000>	2	10	0.014	FT_P
M23	<i1: ZZCX> <o2: AZZXX, 6000> <m2: A_reboot> <i1: ZZCX> <o2: AZZXX, 6000>	14	5	0.010	FT_P

6.2.2 RQ2 (失效揭示能力) 结果

在默认的覆盖准则下 (见第 5.4 节), 相对容易控制所生成测试集的大小, 但可能无法揭示一些失效. 本文还提出了 RE-T、EV-C、ET-B 等非默认覆盖准则. 这些准则在被测系统行为的某些方面具有更好的失效揭示能力.

实验尝试在本文引入的不同覆盖准则下生成测试用例. 其结果表明, 通过使用表 10 列“覆盖准则”中给出的覆盖准则以及关于模型其他方面的默认覆盖准则, 所生成的测试用例能够成功地揭示了注入到待测任务系统中的失效 (见附件). 例如, 在场景 S6 中, 过短或过长的 DDS 消息间隔存在导致系统无法正常工作的风险. 实验中注入了一

个失效, 如果在某给定消息和它的前置消息之间的时间间隔超过一个限度, 系统将停止响应, 以重现在该场景下曾经发生的历史失效. 通过场景模型 M6 和覆盖准则 ET-B, 可以生成具有不同时间间隔的消息序列. 这些消息序列可以触发注入的失效.

能够成功在所编写的场景模型和覆盖准则下揭示注入的失效, 这表明所提出的测试用例生成算法在生成具有失效揭示能力的测试集方面是有效的.

6.2.3 RQ3 (测试生成效率) 结果

在表 10 中, 列“单用例生成时间”列出了在给定场景下用于生成单个测试用例的平均时间. 从该表可见, 平均而言, 一个测试用例可以在不到 1 s 的时间内生成 (0.007–0.832 s). 对于测试集, 在一个场景模型下, 整个测试集通常可以在几分钟内生成 (最大为 75.7 s, 在场景模型 M4 下). 这些结果表明, 所提出的测试用例生成算法具有较高的效率. 之所以测试生成如此快速, 是因为在如表 8 所示的黑盒场景模型下, 需要求解的约束、需要确定的自由数据元素、以及需要运算的计算函数规模往往不是很大. 在实验中, 生成的测试集相对较小, 这也使得测试生成速度可以较快.

6.2.4 RQ4 (方法比较) 结果

(1) 与人工测试的代价比较

实验耗费一个研究生约 2 个月驻集成试验现场时间来完成 (和被测系统研发同步开展, 受研发进度和现场资源限制, 难以投入更多人员和时间). 实验过程涉及 4 个主要任务: 学习舰船任务系统详细功能、结构和 DDS 接口; 熟悉构件的模拟器功能和代码; 构建场景模型并生成测试用例; 实施故障注入并收集实验数据.

整个实验过程共编写 102 个场景模型, 限于安全策略, 本文仅讨论其中具有代表性的 23 个场景及其模型. 粗略统计, 约 1/3 实验时间 (20 天) 耗费于建立场景模型. 表 11 列出了实验所涉及场景的建模时间. 每个场景模型平均约耗费 1.85 小时构建, 包括编写场景模型描述文件以及调试模型直到成功生成有效的可执行测试用例, 越复杂的模型耗费的构建时间越多. 考虑到模型编写者并不十分熟悉舰船任务系统, 建模代价可以接受.

实验测算, 手工新编写一个待测系统测试用例一般需要 0.5 小时以上. 编写一个场景模型的时间高于编写一个普通测试用例, 但考虑到从一个场景模型出发, 可以用算法在几分钟内衍生大量不同测试用例 (实验中 2-985 个), 总体测试构造效率并不低于手工编写测试用例.

(2) 与现有基于正则表达式和 UML 场景模型的生成方法比较

目前基于正则表达式的测试生成主要关注序列结构构造. 一些方法采用组合测试技术生成事件数据[23], 但其需要构造良好的参数数据池, 目前尚无法用于本实验的舰船任务系统. 序列生成方面, 序列结构表达能力较强的是带并发机制的正则表达式[19,24,25], 本实验将就各个场景案例与其比较表达能力.

基于 UML 顺序图的测试生成是典型的基于场景模型的测试生成方法[14,15], 结合 OCL 约束公式, 也可以从事件内角度描述交互数据, 但现有方法或者依靠手工构造事件数据[15], 或者需要建模系统内部计算过程, 从而采用符号执行来构造完整事件数据[14], 同样难以用于本实验的舰船任务系统. 考虑到对于复杂系统建模内部计算过程的困难, 本实验对已有方法进行简化, 假定有一个从顺序图出发来构造事件序列, 基于事件内随机和边界值生成及 OCL 约束求解构造事件数据的方法, 以此作为比较对象.

表 11 右侧列出了现有方法与本文方法相比在处理 23 个场景方面的不足. P1–P10 是各场景上已有方法存在的问题. 其中, 问题 P1 是指现有建模方法不支持用类似本文的全局事件变量方式表达交互序列中多个位置的关联性事件选择, 需要在正则表达式或顺序图中显式枚举不同变量选择组合下的事件序列, 导致模型构建繁琐复杂. P2 是指对于非关联的局部事件变量, 也需要在正则表达式或场景图中枚举列出所有选项.

总体来看, 对于实验所列场景, 不考虑事件参数时, 许多情况用已有正则表达式也能建模 (S1–S11), 但缺乏类似测试数据参数化的变量机制, 导致必须用更复杂表达式才能描述同样的事件序列 (P1, P2); 不支持变量性的量词 (P3), 也限制了灵活性. 本研究的突出改进是将正则表达式序列模型和交互序列中各类事件以及事件数据的建模融为一体, 使得测试生成不再如已有工作局限于交互顺序的构造, 而是可以获得完整可执行的测试用例.

相对于 UML 顺序图, 从表达能力来看, 本文引入了变量化事件机制, 较之顺序图上的 alt 等机制更容易表达关联或非关联的多组事件选择 (P1, P2); 支持表达并发压力 (P7)、异常输入 (P5), 这是顺序图模型所缺乏的; 本文

还支持直接方式表达事件间的数据关联 (P6), 较之建模系统内部计算过程来建立前后事件间的联系更适合舰船任务系统这样的复杂系统. 从模型编写来看, 本文的文本化模型只需简单工具即可快速编制, 而如果要绘制数十上百顺序图, 则往往代价更高. 就测试生成算法而言, 对于不含事件间数据关联 (P6)、并发压力 (P7)、计算函数 (P9)、后继输入依赖前置输出而需要在线生成 (P10)、异常输入 (P5) 的情况, 采用现有基于顺序图和 OCL 的技术在更复杂的场景模型描述下也能生成同样的测试用例 (场景 S2, S4, S7–S11). 相较于现有方法, 本文优势在于通过基于函数计算和约束求解的混合生成机制、在线测试生成等, 能够处理更多场景 (S1, S3, S5, S6, S12–S23), 更适合领域需要. 当然, UML 本身设计完善, 标准化程度高, 本文方法也仅在所针对的测试问题方面具有更好的适用性.

表 11　与现有基于正则表达式和 UML 顺序图的测试生成的比较

场景	场景建模时间	与基于正则表达式的测试比较		与基于UML顺序图的测试比较		问题
		场景建模	测试生成	场景建模	测试生成	
S1	2.0	P1, P4	可在更复杂的正则表达式下生成事件序列; 无法生成事件参数	P1, P6	P8	P1. 需显式枚举关联性选择路径; P2. 需要枚举变量选择; P3. 不支持变量性量词; P4. 不支持交互数据建模; P5. 不支持异常输入表达; P6. 不支持事件间数据关联的显式表达; P7. 不支持描述并发压力; P8. 建模不完整, 导致测试难以生成; P9. 算法不能处理计算函数, 导致测试无法生成; P10. 不支持在线生成算法, 导致测试无法生成
S2	1.6	可以表达		可以表达	可以生成	
S3	3.0	P1, P4		P1, P6	P8	
S4	1.5	P2, P3, P4		P2	可以生成	
S5	1.5	P2, P3, P4		P2, P7	P8	
S6	2.0	P1, P4		P6	P8	
S7	0.5	P2, P4		P2	可以生成	
S8	0.5	P2, P3, P4		P2	可以生成	
S9	0.5	P2, P4		P2	可以生成	
S10	0.6	P2, P4		P2	可以生成	
S11	0.5	P2, P4		P2	可以生成	
S12	2.2	P1, P4, P5		P1, P5, P6	P8	
S13	2.2	P1, P4, P5		P1, P5, P6	P8	
S14	2.2	P1, P4, P5		P1, P5, P6	P8	
S15	1.7	P1, P3, P4, P5		P1, P5, P6	P8, P9	
S16	1.8	P2, P4, P5		P2, P5	P8	
S17	1.8	P2, P4, P5		P2, P5	P8	
S18	4.0	P1, P3, P4, P5		P1, P5	P8	
S19	4.5	P1, P3, P4, P5		P1, P5, P6	P8, P9	
S20	1.6	P1, P4, P5		P1, P5	P8, P9	
S21	1.6	P1, P4, P5		P1, P5	P8, P9	
S22	2.5	P3, P4, P5		P5, P6	P8, P10	
S23	2.2	P1, P3, P4, P5		P1, P5	P8, P9	

6.3 有效性威胁

实验中有几个有效性威胁. 首先, 所提出的方法仅在单个舰船任务系统上进行了论证. 尽管该系统具有代表性, 但在更多案例上进行论证, 有助于增强实验结论的通用性. 然而, 真正的舰船任务系统很难有机会访问. 未来计划在类似的工业系统上进行更多案例研究, 不断加强实验论证.

其次, 本文仅检验了 23 个风险场景上的场景建模和测试用例生成. 尽管如此, 这些场景覆盖了论文针对的常见失效模式. 相信这些场景的成功建模和测试可以展示所提方法的有效性. 建模舰船任务系统这类大型工业系统上的场景, 需要大量领域知识, 未来也考虑培训来自行业的测试人员来描述更多场景并编写场景模型.

另外, 实验将失效注入到真实构件的模拟器中来检验测试生成算法的失效揭示能力. 这些模拟器与真实构件具有相同的 DDS 接口, 仅内部逻辑被简化. 相同的接口, 意味着一个测试用例能在模拟器上触发某 DDS 交互场景下的失效, 同样也能在真实系统上触发相同交互场景下的失效. 在模拟器上进行实验, 并不影响本文输入输出级失效模式检测能力的评估. 由于模拟器仅有简单实现, 且本文主要关注黑盒级失效模式, 因此实验并未系统地在源代码中植入错误. 未来也计划编写一些实现代码与真实构件接近的模拟器, 依托代码开展更系统的错误注入, 以进一

步评估测试用例生成的失效揭示能力.

6.4 讨 论

本文的方法本质上是人工测试过程在某种程度上的自动化. 在人工测试中, 测试人员设计具体的事件序列来构建测试用例. 本文将具体事件序列抽象为基于扩展正则表达式的场景模型. 一个场景模型可以表达许多不同序列, 这可避免大代价地逐个编写具体事件序列. 人工测试中, 在测试人员对潜在失效作出新的假设后, 需要重新设计测试用例来验证设想. 而使用本文的方法, 一旦一个场景模型被构建, 类似的测试用例就可以自动生成. 覆盖引导的测试生成使得可以比较容易地、系统化地检查场景下的多种可能情况. 人工测试需要编写底层通信程序来触发事件序列的执行, 而使用本文的方法, 事件序列只需在抽象层面表达, 无需任何底层编程. 上述特点为基于 DDS 的分布式系统的测试带来了方便.

尽管所提出的方法展现了其价值, 但也存在一些局限.

(1) 该测试方法主要针对表 2 从历史经验中总结出的部分失效模式, 虽然已可用于检测许多常见质量风险, 但无法保证对更多未知失效模式的支持, 且检错也依赖有效的场景建模. 方法基于输入输出, 不支持与输入模式无关的缺陷的测试. 本文主要关注测试输入的生成, 对于测试执行结果的判别, 一些失效无法从场景模型中给出的预期 DDS 输出判定. 针对这类情况, 可能需要人工检查或采用看门狗等技术帮助确定失效.

(2) 场景模型中, 扩展正则表达式的表达能力仍然有限. 例如, 正则表达式本质上表达的是一种上下文无关语言, 尽管作了扩展, 但一些后续动作序列由前置动作决定的上下文相关交互序列仍不能由扩展正则表达式建模.

(3) 该方法难以测试一些与复杂领域数据 (如雷达/声纳数据) 相关的问题. 这是因为对这些数据的限制很难用一阶逻辑公式甚至简单的计算函数来表达. 实验中, 基于场景的方法主要用于测试整体业务逻辑. 过于复杂的领域数据通过随机生成或从监控到的真实数据中选择来构造, 所得结果难以系统地测试领域数据相关问题.

(4) 本文方法只能处理有限形式的非确定性, 即后续输入的内容依赖于确定性事件序列中前置非确定输出结果的情况. 对于更复杂的非确定性事件序列, 仍需研究其他方法加以解决.

将不断改进所提出的方法, 更多克服上述限制, 增强其可用性.

7 小 结

本文从新型舰船任务系统的测试需要出发, 提出了一种基于场景模型的 DDS 架构分布式系统测试方法. 首先基于扩展正则表达式等, 构建场景模型来描述可触发特定任务场景的测试输入以及相应系统输出. 然后, 设计算法从场景模型生成测试用例. 在某真实舰船任务系统上的实验证实了方法的有效性, 其场景建模语言具有良好的表达能力, 所生成的测试用例可揭示系统中的多种常见失效模式. 方法适用于具有特殊研发和应用特征的被测对象, 但对 DDS 架构系统的结构并无特殊要求, 除舰船任务系统外, 也可用于其他基于 DDS 的分布式系统.

References:

[1] Wang D, Zuo YJ, Guo J. New generation surface warship combat system of the US navy system architecture. Command Control & Simulation, 2018, 40(1): 132–140. [doi: 10.3969/j.issn.1673-3819.2018.01.026]

[2] Song M, Wei ZX, Yin GS. Evolution analysis of data flow oriented internetware service. Ruan Jian Xue Bao/Journal of Software, 2013,24(12):2797–2813 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4396.htm [doi: 10.3724/SP.J.1001.2013.04396]

[3] Tao CQ, Li BX, Gao J. Complexity metrics for regression testing of component-based software. Ruan Jian Xue Bao/Journal of Software, 2015, 26(12): 3043–3061 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4876.htm [doi: 10.13328/j.cnki.jos.004876]

[4] Piel EAB, González A, Gross HG. Automating integration testing of large scale publish/subscribe systems. Principles and Applications of Distributed Event-based Systems. 2010. 140–163. [doi: 10.4018/978-1-60566-697-6.ch007]

[5] Köksal Ö, Tekinerdogan B. Obstacles in data distribution service middleware: A systematic review. Future Generation Computer Systems, 2017, 68: 191–210. [doi: 10.1016/j.future.2016.09.020]

[6] OMG. A data distribution service for real-time systems. 2015. https://www.omg.org/spec/DDS/1.4

[7] Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P. An orchestrated survey of

methodologies for automated software test case generation. Journal of Systems and Software, 2013, 86(8): 1978–2001. [doi: 10.1016/j.jss.2013.02.061]

[8] Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 2012, 22(5): 297–312. [doi: 10.1002/stvr.456]

[9] Gurbuz HG, Tekinerdogan B. Model-based testing for software safety: A systematic mapping study. Software Quality Journal, 2018, 26(4): 1327–1372. [doi: 10.1007/s11219-017-9386-2]

[10] Li YS, Pierce BC, Zdancewic S. Model-based testing of networked applications. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). Virtual: ACM, 2021. 529–539. [doi: 10.1145/3460319.3464798]

[11] Nebut C, Fleurey F, Le Traon Y, Jezequel JM. Automatic test generation: A use case driven approach. IEEE Trans. on Software Engineering, 2006, 32(3): 140–155. [doi: 10.1109/TSE.2006.22]

[12] Dadeau F, Castillos KC, Tissot R. Scenario-based testing using symbolic animation of B models. Software Testing, Verification and Reliability, 2012, 22(6): 407–434. [doi: 10.1002/stvr.1467]

[13] Kesserwan N, Dssouli R, Bentahar J, Stepien B, Labrèche P. From use case maps to executable test procedures: A scenario-based approach. Software & Systems Modeling, 2019, 18(2): 1543–1570. [doi: 10.1007/s10270-017-0620-y]

[14] Bandyopadhyay A, Ghosh S. Test input generation using UML sequence and state machines models. In: Proc. of the 2009 Int'l Conf. on Software Testing Verification and Validation (ICST). Denver: IEEE, 2009. 121–130. [doi: 10.1109/ICST.2009.23]

[15] Rocha M, Simão A, Sousa T. Model-based test case generation from UML sequence diagrams using extended finite state machines. Software Quality Journal, 2021, 29(3): 597–627. [doi: 10.1007/s11219-020-09531-0]

[16] Dan HT, Hierons RM. Conformance testing from message sequence charts. In: Proc. of the 4th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). Berlin: IEEE, 2011. 279–288. [doi: 10.1109/ICST.2011.29]

[17] Minhas NM, Masood S, Petersen K, Nadeem A. A systematic mapping of test case generation techniques using UML interaction diagrams. Journal of Software: Evolution and Process, 2020, 32(6): e2235. [doi: 10.1002/smr.2235]

[18] Wang LZ, Yuan JS, Yu XF, Hu J, Li XD, Zheng GL. Generating test cases from UML activity diagram based on gray-box method. In: Proc. of the 11th Asia-Pacific Software Engineering Conf. Busan: IEEE, 2004. 284–291. [doi: 10.1109/APSEC.2004.55]

[19] Grieskamp W. Multi-paradigmatic model-based testing. In: Proc. of the 1st Combined Int'l Workshops on Formal Approaches to Software Testing and Runtime Verification. Seattle: Springer, 2006. 1–19. [doi: 10.1007/11940197_1]

[20] Masson PA, Julliand J, Plessis JC, Jaffuel E, Debois G. Automatic generation of model based tests for a class of security properties. In: Proc. of the 3rd Int'l Workshop on Advances in Model-based Testing. London: ACM, 2007. 12–22. [doi: 10.1145/1291535.1291537]

[21] Larsen PG, Lausdahl K, Battle N. Combinatorial testing for VDM. In: Proc. of the 8th Int'l Conf. on Software Engineering and Formal Methods (SEFM). Pisa: IEEE, 2010. 278–285. [doi: 10.1109/SEFM.2010.32]

[22] Ledru Y, du Bousquet L, Maury O, Bontron P. Filtering TOBIAS combinatorial test suites. In: Proc. of the 7th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE). Barcelona: Springer, 2004. 281–294. [doi: 10.1007/978-3-540-24721-0_21]

[23] Polo M, Pedreira O, Places ÁS, de Guzmán IGR. Automated generation of oracled test cases with regular expressions and combinatorial techniques. Journal of Software: Evolution and Process, 2020, 32(12): e2273. [doi: 10.1002/smr.2273]

[24] Belli F, Budnik CJ, Hollmann A. A holistic approach to testing of interactive systems using statecharts. In: Proc. of the 2nd South-east European Workshop on Formal Methods.2005. 59–73.

[25] Liu P, Miao HK. Theory of test modeling based on regular expressions. In: Proc. of the 3rd Int'l Workshop on Structured Object-oriented Formal Language and Method. Queenstown: Springer, 2013. 17–31. [doi: 10.1007/978-3-319-04915-1_2]

[26] RTI in aerospace and defense. https://www.rti.com/docs/RTI_for_Defense.pdf

[27] Hoffman DM, Ly-Gagnon D, Strooper P, Wang HY. Grammar-based test generation with YouGen. Software: Practice and Experience, 2011, 41(4): 427–447. [doi: 10.1002/spe.1017]

[28] Ali S, Iqbal MZ, Khalid M, Arcuri A. Improving the performance of OCL constraint solving with novel heuristics for logical operations: A search-based approach. Empirical Software Engineering, 2016, 21(6): 2459–2502. [doi: 10.1007/s10664-015-9392-6]

[29] Sartaj H, Iqbal MZ, Khan MU. Testing cockpit display systems of aircraft using a model-based approach. Software and Systems Modeling, 2021, 20(6): 1977–2002. [doi: 10.1007/s10270-020-00844-z]

[30] Dadeau F, Fourneret E, Bouchelaghem A. Temporal property patterns for model-based testing from UML/OCL. Software & Systems Modeling, 2019, 18(2): 865–888. [doi: 10.1007/s10270-017-0635-4]

[31] Kanso B, Taha S. Specification of temporal properties with OCL. Science of Computer Programming, 2014, 96: 527–551. [doi: 10.1016/j.scico.2014.02.029]

[32] Michlmayr A, Fenkam P, Dustdar S. Architecting a testing framework for publish/subscribe applications. In: Proc. of the 30th Annual Int'l

Computer Software and Applications Conference (COMPSAC'06). Chicaco: IEEE, 2006. 467–474. [doi: 10.1109/COMPSAC.2006.28]

[33] Cotroneo D, Natella R, Russo S, Scippacercola F. State-driven testing of distributed systems. In: Proc. of the 17th Int'l Conf. on Principles of Distributed Systems. Nice: Springer, 2013. 114–128. [doi: 10.1007/978-3-319-03850-6_9]

[34] Grace P, Barbosa J, Pickering B, Surridge M. Taming the interoperability challenges of complex IoT systems. In: Proc. of the 1st ACM Workshop on Middleware for Context-aware Applications in the IoT. Bordeaux: ACM, 2014. 1–6. [doi: 10.1145/2676743.2676744]

[35] Bozkurt M, Harman M, Hassoun Y. Testing and verification in service-oriented architecture: A survey. Software Testing, Verification and Reliability, 2013, 23(4): 261–313. [doi: 10.1002/stvr.1470]

[36] de Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Budapest: Springer, 2008. 337–340. [doi: 10.1007/978-3-540-78800-3_24]

[37] DDSTest. https://github.com/juqian/ddstest

[38] automaton. http://www.brics.dk/automaton/

附中文参考文献:

[1] 王达, 左艳军, 郭俊. 美国海军新一代水面舰艇作战系统体系架构. 指挥控制与仿真, 2018, 40(1): 132–140. [doi: 10.3969/j.issn.1673-3819.2018.01.026]

[2] 宋敏, 韦正现, 印桂生. 面向数据流的网构软件服务动态演化分析. 软件学报, 2013, 24(12): 2797–2813. http://www.jos.org.cn/1000-9825/4396.htm [doi: 10.3724/SP.J.1001.2013.04396]

[3] 陶传奇, 李必信, Gao J. 构件软件的回归测试复杂性度量. 软件学报, 2015, 26(12): 3043–3061. http://www.jos.org.cn/1000-9825/4876.htm [doi: 10.13328/j.cnki.jos.004876]

附录 A

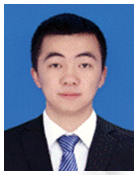
表 A.1 注入的失效

序号	失效描述	序号	失效描述
F1	武器发射和暂停过程在正确的输入消息序列下失败. 输入: 特定X型武器的发射和暂停命令 预期: 武器X成功暂停并返回新状态 失效: 武器X没有响应	F13	当武器发射过程中丢失消息时, 武器系统失效. 输入: 武器X的发射和暂停命令序列, 且丢失某个特定的命令 预期: 武器X报告错误信息 失效: 武器X没有响应
F2	任务模式切换在显控台上正确的手工动作顺序下失败. 输入: 从任务模式X切换到任务模式Y的手工界面动作 预期: 任务模式成功改变(界面布局随之改变) 失效: 任务模式保持不变	F14	当武器发射过程中出现重复消息时, 武器系统失效. 输入: 武器X的发射和暂停命令序列, 且包含了某个特定命令的重复 预期: 武器X报告错误信息 失效: 武器X没有响应
F3	武器发射过程中的武器类型切换失败. 输入: 暂停武器X并切换到武器Y的消息 预期: 武器Y正常工作并返回新状态 失效: 武器Y没有响应	F15	系统无法容错周期性目标探测消息的周期过长. 输入: 周期过长的目标探测消息 预期: 指控软件正确处理目标探测消息或报告错误 失效: 指控软件崩溃
F4	指控软件无法响应大量连续出现的目标探测消息. 输入: 连续出现数量超过N的目标探测消息 预期: 指控软件正确处理输入消息 失效: 指控软件崩溃	F16	系统无法容错实时目标探测消息中出现的数据溢出. 输入: 特定数据元素出现数据溢出的目标探测消息 预期: 指控软件立即或在稍后的执行中报告错误 失效: 指控软件崩溃
F5	指控软件无法响应大量接近并行发生的目标探测消息. 输入: 并行出现数量超过N的目标探测消息 预期: 指控软件正确处理输入消息 失效: 指控软件崩溃	F17	系统无法容错实时目标探测消息中包含未初始化的数据. 输入: 带有特定未初始化数据元素的目标探测消息 预期: 指控软件立即或在稍后的执行中报告错误 失效: 指控软件崩溃
F6	武器发射任务流由于命令消息间隔时间过长而失败. 输入: 以T为时间间隔的武器X发射命令序列 预期: 武器X正常处理处理命令, 或返回错误消息 失效: 武器X没有响应	F18	武器管理软件在收到数据成员不满足约束的鱼雷设备状态消息时发生异常. 输入: 数据成员不满足特定值约束的鱼雷设备状态消息 预期: 武器管理软件立即或在随后的执行中报告错误 失效: 武器管理软件崩溃

表 A.1 注入的失效 (续)

序号	失效描述	序号	失效描述
F7	来自声纳的包含大块数组的消息导致指控软件崩溃. 输入: 包含大块浮点数数组的某声纳消息 预期: 指控软件正确处理输入消息 失效: 指控软件崩溃	F19	指控软件无法处理时间戳不满足消息间约束的周期出现目标探测消息序列. 输入: 不满足消息间时间戳约束的目标探测消息序列 预期: 指控软件立即或在稍后的执行中报告错误 失效: 指控软件崩溃
F8	来自雷达的带边界大小可变长数据的消息导致指控软件崩溃. 输入: 来自雷达的某带边界大小可变长数据的消息 预期: 指控软件正常处理输入消息 失效: 指控软件崩溃	F20	无法容错设备状态查询中主题为空的DDS消息. 输入: 空主题的设备X状态查询命令 预期: 设备X忽略该消息 失效: 设备X停止工作
F9	声纳软件无法处理来自指控软件的包含复杂嵌套数据类型的消息. 输入: 指控软件包含复杂嵌套数据类型的特定消息 预期: 声纳软件正确处理输入消息 失效: 声纳软件崩溃	F21	无法容错设备状态查询中域值错误的DDS消息. 输入: 域值异常的设备X的状态查询命令 预期: 设备X忽略该消息 失效: 设备X停止工作
F10	指控软件无法处理导航软件报告的带浮点数数据的消息. 输入: 导航软件报告的带浮点数数据的特定消息 预期: 指控软件正确处理输入消息 失效: 指控软件崩溃	F22	当武器设备丢失连接后, 武器管理软件崩溃. 输入: 多组武器状态查询指令, 查询过程中武器X突然丢失连接 预期: 武器管理软件立即或稍后检测到武器设备的丢失 失效: 武器管理软件崩溃
F11	鱼雷软件无法处理来自指控软件的包含复杂对齐数据成员的命令消息. 输入: 来自指控软件的包含对齐数据的特定命令消息 预期: 鱼雷软件正确处理消息 失效: 鱼雷软件崩溃	F23	声纳子系统无法从设备崩溃并重启中恢复. 输入: 插入了声纳设备突然关机和手动重启(模拟崩溃和重启)的声纳状态查询命令 预期: 声纳子系统重启后成功恢复 失效: 声纳软件功能无法恢复, 无法获得查询结果
F12	当武器发射过程中出现乱序消息时, 武器系统失效. 输入: 武器X的发射和暂停命令序列, 且两个特定消息之间的顺序不正确 预期: 武器X报告错误信息 失效: 武器X没有响应		

钱巨(1981－), 男, 博士, 副教授, 硕士生导师, CCF 专业会员, 主要研究领域为软件分析与测试.



程浩(1994－), 男, 硕士生, 主要研究领域为软件分析与测试.

王寅(1999－), 女, 硕士生, 主要研究领域为软件分析与测试.

韦正现(1977－), 男, 研究员, 主要研究领域为软件体系结构, 软件构件.