Embedded Technoloy and Intelligent Systems 嵌入式技术与智能系统, 2025, 2(4), 223-231
Published Online August 2025 in Hans. https://www.hanspub.org/journal/etis
https://doi.org/10.12677/etis.2025.24020

嵌入式软件语言及开发新范式

陈铭松，焦跃军，杨永淏

华东师范大学软硬件协同设计技术与应用教育部工程研究中心，上海

收稿日期：2025年10月28日；录用日期：2025年11月20日；发布日期：2025年12月1日

摘　要

嵌入式系统作为针对各类特定用途的计算平台，已被广泛应用到航空航天、轨道交通、汽车电子以及智慧城市等领域。随着嵌入式系统复杂度增加，传统嵌入式软件语言及开发面临性能瓶颈、资源受限、安全要求严苛及开发效率低下等多重挑战。传统嵌入式软件语言内存管理机制复杂不仅提高了学习门槛，也增加了维护难度。嵌入式软件开发层次多、复杂性高、软硬件耦合紧密导致上市时间长难以快速迭代。近年来兴起的新型嵌入式开发语言、低代码平台、智能开发辅助工具以及虚拟仿真平台有效提升了嵌入式开发效率。文章将围绕这些新兴的嵌入式软件语言和开发范式，综述国内外研究现状与发展趋势，为嵌入式软件开发提供参考。

关键词

嵌入式软件，软件开发语言，低代码，虚拟仿真

New Paradigms for Embedded Software Languages and Development

Mingsong Chen, Yuejun Jiao, Yonghao Yang

MoE Engineering Research Center of Software/Hardware Co-Design Technology and Application, East China Normal University, Shanghai

Received: October 28, 2025; accepted: November 20, 2025; published: December 1, 2025

Abstract

Embedded systems are application-specific computing platforms that have been widely used in various safety-critical domains, including aerospace, rail transit, automotive electronics, and smart cities. As the complexity of embedded systems increases, traditional embedded software languages and development processes face multiple challenges, including performance bottlenecks, limited resources, strict security requirements, and low development efficiency. The complex memory

文章引用: 陈铭松, 焦跃军, 杨永淏. 嵌入式软件语言及开发新范式[J]. 嵌入式技术与智能系统, 2025, 2(4): 223-231.
DOI: 10.12677/etis.2025.24020

management mechanisms of traditional embedded languages not only raise the learning threshold but also increase maintenance difficulty. The multiple layers, high complexity, and tight hardware-software coupling of embedded software development lead to long time-to-market and difficulty in rapid iteration. In recent years, emerging embedded development languages, low-code platforms, intelligent development tools, and virtual simulation platforms have been reshaping the paradigms for embedded software languages and development. This survey focuses on these emerging embedded software languages and development paradigms, reviews the current research status and development trends at home and abroad, and provides references for embedded software development.

Keywords

Embedded Software, Software Development Language, Low-Code, Virtual Simulation

1. 引言

嵌入式系统作为面向特定应用场景的计算平台，已广泛应用于航空航天、轨道交通、汽车电子以及智慧安防等领域。随着系统功能的持续扩展与集成度提升，嵌入式软件的开发语言与方法面临新的挑战。传统嵌入式开发语言虽然在执行性能和资源占用方面具有显著优势，但其内存管理机制复杂、语法门槛高，难以适应嵌入式项目的快速迭代需求[1]-[3]。同时，嵌入式系统的开发流程层次多、实现复杂度高，这导致一致性维护与软件验证成为开发效率瓶颈。此外，嵌入式开发中软硬件紧密耦合，从前期的系统规约、软硬件划分到具体的软硬件实现，持续对软件开发效率和优化迭代造成影响，从而限制了应用场景的扩展并延长了项目研发周期。

为了应对传统方法在嵌入式软件开发过程中展现的弊端，围绕新型嵌入式开发语言、低代码平台、智能开发辅助工具以及虚拟仿真平台的研究正成为业界与学术界关注的重点。本文将系统剖析当前嵌入式开发所面临的关键技术挑战，探讨这些嵌入式软件语言与开发范式在嵌入式领域的核心机制、成功案例与工业实践，为嵌入式软件开发提供参考。

2. 嵌入式系统开发挑战

嵌入式系统开发面临在有限时间内实现高性能与高可靠性的双重压力[4]。随着市场竞争推动上市时间[5] (Time-to-Market)要求不断缩短，传统的嵌入式软件开发方法逐渐暴露出在复杂系统中难以兼顾开发效率与产品质量的局限性。本章将分析嵌入式开发过程中面临的挑战，从开发语言的设计局限与开发流程的层次复杂两个方面展开探讨。

2.1. 嵌入式开发语言设计局限多

嵌入式系统开发领域存在语言种类繁多、学习周期长等问题。由于不同应用场景往往采用特定的 DSL (Domain-Specific Language，领域特定语言) 或平台相关框架，开发者不仅需要掌握底层硬件接口与实时操作系统机制，还必须熟悉多种语言特性与工具链，从而显著延长了学习与开发周期。在实际工程中，传统通用语言(如 C/C++)虽然在性能优化与硬件控制方面具备优势，但其语法允许程序直接操作内存地址，缺乏访问约束与内存安全机制，极易引发缓冲区越界与指针错误，对系统可靠性构成长期威胁[6] [7]。

此外，随着嵌入式软件代码规模普遍达到百万行级别，这些问题的检测与修复难度急剧增加。传统事后审查的排错方式需要根据故障现象排查漏洞。由于嵌入式系统运行环境复杂，运行时暴露的问题往往难以稳定复现，使故障定位与修复过程更加困难。

2.2. 嵌入式开发流程复杂度高

嵌入式系统开发是一个贯穿方案设计、系统实现到产品维护的多阶段复杂工程，其流程层次多、环节复杂导致一致性维护难度大。嵌入式系统开发的典型流程如图 1 所示。在方案设计阶段，系统架构师与软件工程师往往因专业背景差异产生认知偏差，导致设计意图与技术实现间存在落差[8] [9]。在项目实现阶段，开发团队成员能力与习惯差异可能降低代码质量并引入潜在漏洞[10]。同时，受限于硬件资源与实现复杂度，开发目标常被迫调整为简化方案，从而削弱系统性能优化空间[11]。在维护阶段，复杂的技术文档使问题定位与修复过程高度依赖开发人员，延长了纠错周期。

此外，嵌入式软件与硬件之间的强耦合关系极大延长了设计周期和开发周期。在开发前期，若目标板卡或外设尚未就绪，驱动程序、BSP (Board Support Package，板级支持包)及底层固件等模块无法在真实环境中进行验证与调优，团队只能依赖功能受限的模拟环境或延迟实现[12] [13]。而在后期集成阶段，软件测试往往需等待硬件原型完成后才能开展，且硬件验证(从印制电路板制造到原型调试)周期长、成本高，进一步压缩了软件测试与优化的空间。这种软硬件紧密耦合的串行开发模式导致开发流程缺乏灵活性与自动化支持，使嵌入式系统难以实现快速迭代与持续优化。

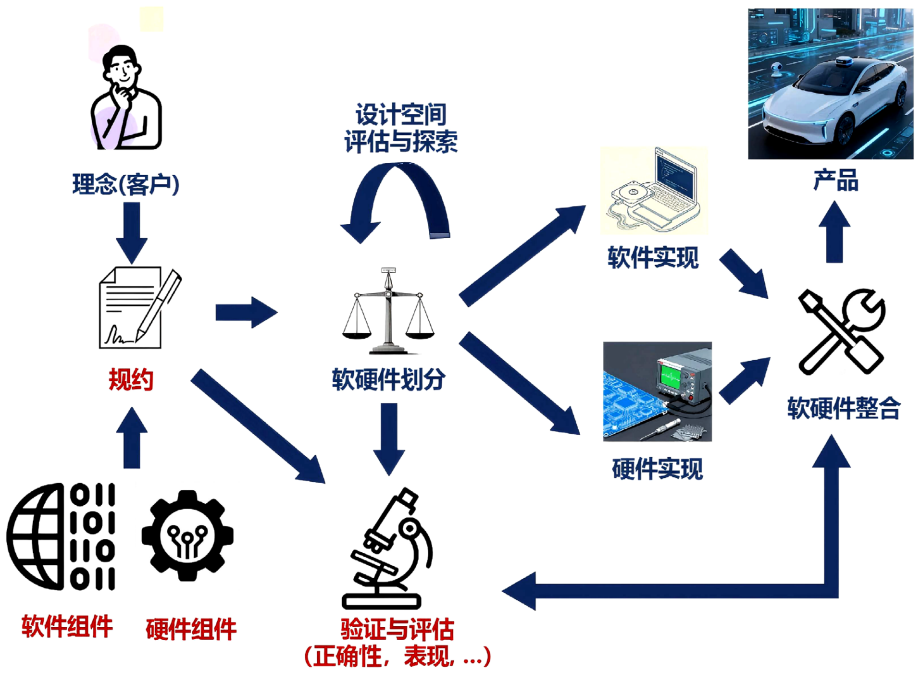

Figure 1. General development process of embedded systems
图 1. 嵌入式系统开发基本流程

3. 嵌入式软件语言创新趋势

以 C/C++为代表的传统嵌入式软件语言在其性能与泛用性上已经得到了充分验证，但基于传统软件语言的工业项目在长期运行中暴露出了安全隐患与漏洞。此类项目的代码修复需要从运行中暴露出的问

题出发，通过问题复现与事后代码分析来进行问题定位。不适应当前“高精度，高实时，高可信”的嵌入式开发要求。目前新型嵌入式软件语言的特征主要集中在以下几个方面。其一是运行前消除，在编译过程中通过合理安全规则检出代码潜在漏洞。其二是运行时安全，通过运行时异常处理策略来避免漏洞损失。其三是保证代码的运行效率，在保证系统安全的基础上尽力贴近 C/C++的运行效率。本章将列举分析 Ada、Go 与 Rust 等典型新型嵌入式语言的特性与进一步发展方向，并结合实际案例，说明其如何在实际应用中构建高可信的嵌入式开发项目。

3.1. 新型嵌入式语言

新型嵌入式软件语言的核心设计理念是将传统由开发者负责的内存安全保障，转变为由语言规范与编译器强制执行的约束条件。其中一种设计是利用先进的类型系统、所有权模型和生命周期分析等技术，在编译阶段识别潜在的内存错误，从源头上消除安全隐患。另一种解决方案是在运行时采用垃圾回收或智能指针等技术，自动化管理内存分配与释放。在嵌入式系统开发中，不同编程语言在内存安全与执行效率的平衡上呈现出差异化特征。目前工业项目中较常见的新型嵌入式软件语言包括 Ada，Go 及 Rust。

Ada 语言在嵌入式领域积累了深厚的应用基础。它的强类型系统、契约编程(含前置与后置条件)及并发任务模型，使其在航空航天、国防等对可靠性要求极高的场景中被广泛采用。通过 SPARK 子集及配套工具(如 SPARK Examiner、CodePeer)，Ada [14] [15]可实现静态分析，对缓冲区越界、空/悬挂指针等潜在运行时错误进行验证，显著增强系统的可靠性与容错能力。

Go 语言则通过垃圾回收机制保障内存安全，并借助其 Go routine 轻量线程与 channel 工具简化并发实现，在快速开发、系统原型验证及非硬实时系统中具有优势。然而，Go 语言的垃圾回收机制引入的不可预测暂停使其难以适配硬实时任务[16]。已有 TinyGo 项目[17]裁剪 Go 运行时检测机制进行优化，目前已应用于驱动开发、IO 控制、OTA 客户端等轻量级任务，在部分场景中满足了基本需求。但在严格硬实时的安全关键系统中，Go 语言垃圾回收机制导致的延迟与内存开销仍是未完全解决的限制[18]。

Figure 2. Memory error cases and their Rust solutions
图 2. 内存错误案例及其 Rust 解决方案

Rust 语言低运行成本的安全管理使其在嵌入式领域的关注度显著提升。Rust 核心优势在于通过独特

的所有权系统与借用检查机制，在编译阶段时保障了内存安全与无数据竞争的并发执行，因此在无需依赖垃圾回收机制的情况下，实现了安全性与性能间的更优平衡[19]。相较于传统 C/C++，Rust 在保留底层硬件控制能力的同时，从语言层面消除了多数悬挂指针、缓冲区越界等内存错误。图 2 中展示了 Rust 编译器可检出的常见内存漏洞。多项研究[20]-[22]结果显示，在资源密集型基准测试中，Rust 程序的执行时间相比于同功能 C++程序仅增加个位数百分比，且内存峰值消耗更低。这种在内存安全与效率上的双重优势，使得 Rust 目前已在嵌入式系统开发中获得较为广泛的应用。

总体而言，当前新型嵌入式编程语言在嵌入式系统的安全与性能平衡方面已取得了优异表现，但 Ada 的复杂语法，Go 的运行时开销，Rust 的复杂编译等问题仍限制了更广泛的应用。新兴开发语言正试图从以下方面取得突破，一是通过语义简化与模块化设计以提升编译效率与开发体验，二是在静态安全性与可用性之间寻求平衡，以降低入门门槛，三是在垃圾回收机制上实现可配置或轻量化设计，以适应边缘计算与云端快速迭代需求。当前嵌入式开发语言的研究趋向于在安全性、可扩展性与性能表现之间实现动态平衡，以满足更复杂的嵌入式系统需求。

3.2. 工业应用案例

已有大量新型嵌入式开发语言应用于开源项目或大型工业项目。如表 1 所示，新型嵌入式开发语言在嵌入式软件开发中广泛应用。例如 Ada 语言在航空航天、轨道交通等安全关键型系统中依旧保持稳健地位。代表性实现包括 SPARK/Ada [23]及 Ravenscar Profile [24]，通过形式化验证与任务调度约束，为波音飞控系统等高安全标准项目提供了长期可靠的技术支撑。Go 语言在嵌入式领域的探索更多集中于边缘计算与云端协同类场景。典型项目如 TinyGo [17]与 Periph.io [25]，前者通过轻量化编译链实现对 ARM Cortex-M、RISC-V 等平台的原生支持，已被应用于物联网终端、工业监测节点等场景。后者则通过统一的硬件抽象层(HAL)接口，简化了基于 Go 的设备驱动与通信协议开发。

与此同时，Rust 语言因其高效且安全的特性，对嵌入式操作系统有较大应用价值。TockOS [26]是由 Rust 实现的嵌入式实时操作系统，保证必要的不安全操作(如直接硬件寄存器访问)被明确标记并封装于受信任的驱动模块中，缩小了潜在危险代码范围，其具体内核安全区划分如图 3 所示。清华大学团队以 Rust 为基础语言研发的 ArceOS [27]可以快速定制适配不同场景的内核。ArceOS 仅需新增几千行代码，即可实现功能完善的宏内核 StarryOS [28]，而替换插件为 vCPU 管理模块后，又能快速形成 Hypervisor AxVisor [29]，其应用 Rust 进行的开发设计兼顾了内核的性能、安全性与可移植性。

总体来看，Go 与 Ada 在其特长领域表现稳定，而 Rust 凭借其编译期内存安全机制与较低运行时损耗，正在成为当前嵌入式开发语言中最受重视、增长最迅速的方向。

Table 1. Large-scale industrial projects/open source projects developed using memory-safe languages
表 1. 使用内存安全型语言开发的大型工业项目/开源项目

使用语言	项目名称	核心特征
GO	TinyGo/Embedded Go	TinyGo 允许将 Go 代码编译用于资源受限的嵌入式设备。
	Mender OTA 客户端	Mender 团队使用 Go 语言编写嵌入式客户端应用。
Ada	Boeing 777 飞控系统	飞机飞控系统采用 Ada 编写，确保高可靠性与容错能力。
Rust	Tock OS	面向微控制器的 Rust 安全嵌入式实时操作系统。
	Ariel OS	支持多核抢占式调度的 Rust 嵌入式操作系统。
	蓝河 OS	国内行业首个全栈 Rust 语言编写的操作系统。

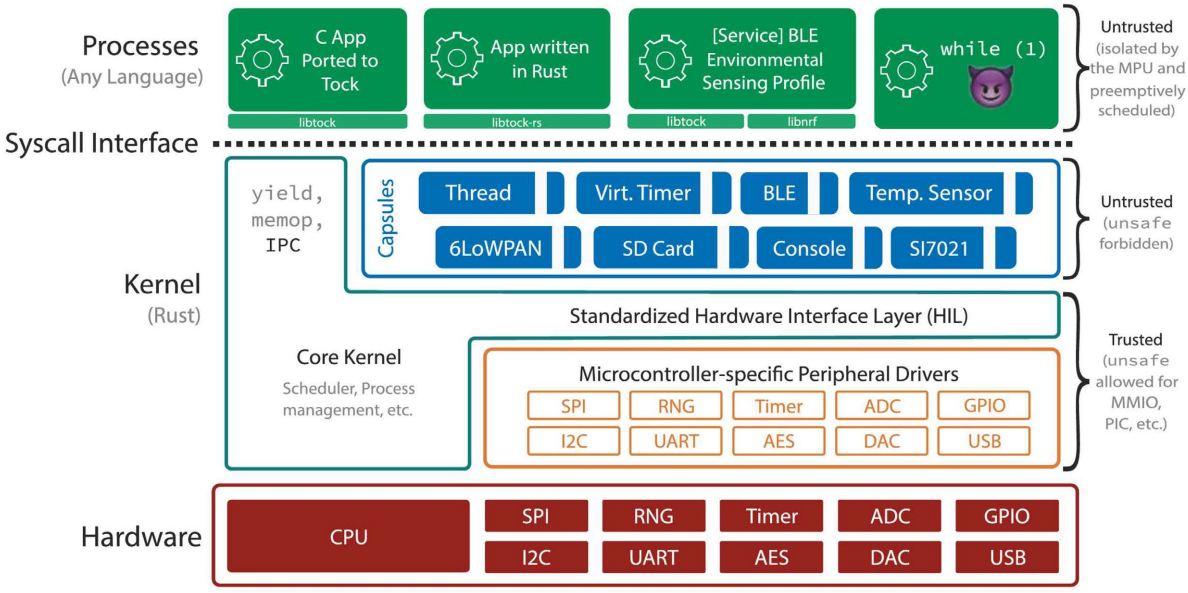

Figure 3. The architecture of TockOS [26]
图 3. TockOS 的基本架构[26]

4. 嵌入式软件开发演进趋势

随着嵌入式系统功能复杂度的不断提升，传统开发模式在性能优化与开发效率之间的矛盾愈发突出。嵌入式软件开发虽然允许底层代码对硬件进行精细控制，从而充分发挥有限资源的潜力、满足实时性与可靠性要求，但其学习门槛高、开发层次多、自动化工具少及软硬件耦合紧密等问题已成为制约创新效率的瓶颈。为应对上述挑战，在嵌入式软件开发领域已涌现出低代码平台、智能开发辅助工具以及虚拟仿真平台的应用案例。低代码平台通过可视化编程显著降低开发门槛，促进跨领域协作。智能辅助开发工具利用大模型与智能体技术，在软件实现与测试评估环节提供持续决策支持。虚拟仿真平台则通过解耦软硬件开发流程，使软件得以在硬件未就绪阶段完成基本功能验证与性能优化，从根本上突破了传统串行开发的限制。本章将系统性地阐述这三类技术的特征与优势，分析它们如何共同推动嵌入式系统开发流程向智能化、自动化与协同化方向演进。

4.1. 嵌入式低代码平台

嵌入式低代码平台通过可视化编程显著降低了开发门槛，使多领域专家能够直接参与应用逻辑的构建过程。嵌入式低代码平台核心技术优势主要体现在三个方面。一是通过拖拽式操作定义功能模块与业务流程，构建可视化开发环境，使开发者能够专注于业务逻辑而非实现细节。二是将实际编码量大幅降低，提升开发效率。三是通过可视化的业务流清晰表达设计意图，有效减少了与专业软件开发团队之间的理解偏差，显著降低了沟通成本。在架构设计上，嵌入式低代码平台通常以节点编程为核心，将应用逻辑转化为由功能节点及其连接关系所构成的数据流图来进行系统定义与实现。每个功能节点代表独立的处理模块。开发者通过连接线定义节点间的数据流向与控制逻辑。

在嵌入式低代码开发领域，已涌现出一批具有代表性的技术平台。国际新兴平台(如 Node-RED、XOD、Visuino 等)普遍侧重于通过节点式可视化编程实现快速的产品原型构建。例如 Node-RED [30]采用基于流(Flow)的可视化编程模型，将应用组织为由输入、逻辑处理与输出节点构成的流程结构。数据在节点之间沿既定路径依次传递与处理，使复杂的底层实现被封装于节点内部。开发者只需通过节点组合与流程编

排即可完成功能构建，从而显著提升开发的直观性与效率。国内相关平台在具体业务应用上有较多实践，其通常与某一类专用硬件设备或某类具体任务紧密结合。例如中科创达 ModelFarm 平台基于其公司设备，专注图像识别任务的低代码开发。

4.2. 嵌入式智能开发辅助工具

嵌入式智能开发辅助工具正逐步成为提升嵌入式系统开发效率与质量的重要手段。这类工具不同于一般的代码补全与生成工具，它们通过集成大模型推理与多智能体协作机制，能够从代码开发到运维管理进行长周期的决策与执行支持。

在开发阶段，基于大模型的工具(如 GitHub Copilot、Tabnine 与 OpenAI Codex)能够理解嵌入式代码上下文，并在特定语义层面生成 RTOS 调度逻辑、硬件驱动模板及中断服务框架。其中，Codex [31]不仅能进行上下文感知代码生成，还可自动构造测试场景以提高覆盖率与系统稳定性。进一步的研究如 AutoDev [32]框架展示了“人工智能代理 + 沙箱环境”的模式。智能体可在受控环境中自动执行编译、调试及优化任务，从而减少人工循环迭代成本。在运维阶段，智能体的作用从开发支持延伸至系统生命周期管理。联通数科已实现设备管理智能体，它可以进行基于自然语言的设备状态查询、故障定位与策略调整，使嵌入式系统的运维由被动监控转向主动诊断与自优化。总体而言，这类嵌入式智能辅助开发工具显著提升了嵌入式系统的开发效率、代码质量与可维护性。

4.3. 虚拟仿真平台

虚拟仿真平台是解决嵌入式系统开发中硬件资源依赖问题的关键技术。其核心思想是通过在软件开发阶段构建虚拟硬件环境，实现软硬件开发流程的解耦，使软件模块能够在硬件尚未到位时独立完成编码与测试。该技术的应用极大缩短了系统集成周期，并减少了硬件迭代带来的开发延误与成本开销。

嵌入式系统的仿真平台经历了从 RTL (Register Transfer Level, 寄存器传输级)仿真到 ESL (Electronic System Level, 电子系统级)仿真的演化过程。RTL 仿真面向硬件逻辑验证，能够准确反映信号变化，但由于仿真粒度细和效率低的问题，导致其难以支撑大规模系统的快速验证。ESL 建模的提出标志着从硬件行为模拟到系统行为模拟的层次提升，它通过 TLM [33] (Transaction Level Modeling，事务级建模)将硬件通信抽象为高层交互事件，在保证功能一致性的同时显著提升仿真效率，允许软件开发脱离硬件设计进度进行前期软件开发、系统集成验证与虚拟原型(Virtual Prototype)构建。

现代虚拟仿真平台普遍采用接口抽象与模块化设计，以提升系统的可扩展性与移植性。例如，VCML [34] (Virtual Component Modeling Library，虚拟组件模型库)基于 SystemC/TLM 框架构建，提供标准化的组件接口与时序控制机制。图 4 演示了使用 VCML 组件进行虚拟平台构建。在汽车电子领域，SystemC/TLM 技术支持开发者对多 ECU (Electronic Control Unit，电子控制单元)单元与车辆动力学模型进行并发仿真[35]，实现并行测试与快速迭代开发。AIoTML 框架[36]通过平台无关建模与模块化编译器架构，实现了面向 AIoT 系统的虚拟化仿真与控制模型统一抽象。Renode [37]则通过可插拔接口支持多架构 CPU 与外设的快速集成，加速嵌入式系统验证。深圳航天科技创新研究院[38]通过高精度元器件模型与实时数据接口，实现软硬件闭环验证。这些平台通过统一的事务接口和协同仿真机制，使多核处理器、通信总线、外设模型等组件能够在同一环境中高效协作。

总体而言，虚拟仿真平台的发展通过实现软硬件解耦，使嵌入式系统开发模式从依赖硬件的串行流程向软硬件并行协同演进，从而显著提升了开发效率。其高抽象层次的系统建模方法，不仅为复杂嵌入式系统的前期快速验证提供了可行途径，也为智能化开发工具和自动化验证体系的构建奠定了技术基础。

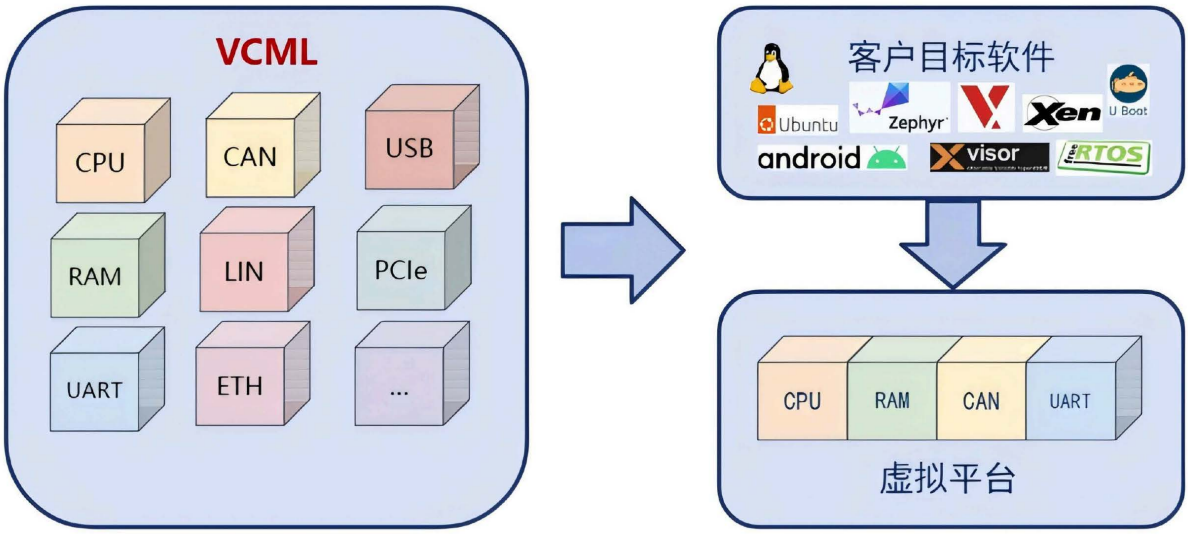

Figure 4. Illustration of virtual platform construction with VCML [34]
图 4. 基于 VCML 的虚拟平台构建示意图[34]

5. 结束语

随着嵌入式系统向“人–机–物”系统演化，嵌入式系统开发语言及开发新范式正以“安全、泛在、高效”为目标加速演进。各类创新技术的协同融合持续推动嵌入式开发向更高效、更可信的阶段跨越。具体来说，新型嵌入式软件语言通过运行前消除机制与运行时安全策略，从根源防范内存泄漏、数据竞争等底层风险。低代码平台降低了嵌入式开发的入门门槛，吸引更多跨领域群体参与嵌入式创新。智能开发辅助工具为专业开发者提供支持，帮助他们突破复杂场景的开发限制。虚拟仿真平台通过软硬件解耦，打破了软件开发对物理硬件依赖的效率瓶颈。随着技术融合的持续深化，嵌入式系统将更广泛地适配特定应用领域的需求。无论是关键行业的核心控制场景，还是民生领域的智能应用场景，都将依托更成熟的开发技术实现创新突破。嵌入式软件语言及开发新范式将加快推动嵌入式系统开发完成从技术迭代到价值落地的全面跃升，为各行业数字化转型提供更坚实的底层支撑。

参考文献

[1] CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. https://cwe.mitre.org/data/definitions/119.html

[2] Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition). https://www.sei.cmu.edu/library/sei-cert-c-coding-standard-rules-for-developing-safe-reliable-and-secure-systems-2016-edition

[3] Secure by Design Alert: Eliminating Buffer Overflow Vulnerabilities. https://www.cisa.gov/resources-tools/resources/secure-design-alert-eliminating-buffer-overflow-vulnerabilities

[4] Sas, D. and Avgeriou, P. (2020) Quality Attribute Trade-Offs in the Embedded Systems Industry: An Exploratory Case Study. *Software Quality Journal*, **28**, 505-534. https://doi.org/10.1007/s11219-019-09478-x

[5] van der Spek, P. and Verhoef, C. (2014) Balancing Time-to-Market and Quality in Embedded Systems. *Systems Engineering*, **17**, 166-192. https://doi.org/10.1002/sys.21261

[6] Cimpanu, C. (2019) Microsoft: 70 Percent of All Security Bugs Are Memory Safety Issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues

[7] Eliminating Memory Safety Vulnerabilities at the Source. https://sechub.in/view/2949364

[8] Mohanani, R., Salman, I., Turhan, B., Rodriguez, P. and Ralph, P. (2018) Cognitive Biases in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering*, **46**, 1318-1339.

https://doi.org/10.1109/tse.2018.2877759

[9] McDermott, T.A., Folds, D.J. and Hallo, L. (2020) Addressing Cognitive Bias in Systems Engineering Teams. *INCOSE International Symposium*, **30**, 257-271. https://doi.org/10.1002/j.2334-5837.2020.00721.x

[10] Tornhill, A. and Borg, M. (2022) Code Red: The Business Impact of Code Quality—A Quantitative Study of 39 Proprietary Production Codebases. *Proceedings of the International Conference on Technical Debt*, Pittsburgh, 16-18 May 2022, 11-20. https://doi.org/10.1145/3524843.3528091

[11] Kaisti, M., Rantala, V., Mujunen, T., Hyrynsalmi, S., Könnölä, K., Mäkilä, T., *et al.* (2013) Agile Methods for Embedded Systems Development—A Literature Review and a Mapping Study. *EURASIP Journal on Embedded Systems*, **2013**, Article No. 15. https://doi.org/10.1186/1687-3963-2013-15

[12] Herdt, V. and Drechsler, R. (2022) Advanced Virtual Prototyping for Cyber-Physical Systems Using RISC-V: Implementation, Verification and Challenges. *Science China Information Sciences*, **65**, Article 110201. https://doi.org/10.1007/s11432-020-3308-4

[13] Liukkunen, K., Eteläperä, M., Oivo, M., Soininen, J. and Pellikka, M. (2008) Virtual Prototypes in Developing Mobile Software Applications and Devices. In: Jedlitschka, A. and Salo, O., Eds., *International Conference on Product Focused Software Process Improvement*, Springer, 174-188. https://doi.org/10.1007/978-3-540-69566-0_16

[14] JJaloyan, G.A., Moy, Y. and Paskevich, A. (2018) Borrowing Safe Pointers from Rust in Spark. arXiv:1805.05576.

[15] GNAT Pro Assurance: Product for Safety-Critical Systems. https://www.adacore.com/gnatpro/assurance

[16] Adapting Go for Embedded/Real-Time Use. https://groups.google.com/g/golang-nuts/c/6MPAzFgoS7M

[17] Microcontroller Targets, TinyGo Documentation. https://tinygo.org/docs/concepts/compiler-internals/microcontrollers

[18] TinyGo: Driver Design and Development. https://tinygo.org/docs/guides/driver-design

[19] Sharma, A., Sharma, S., Tanksalkar, S.R., Torres-Arias, S. and Machiry, A. (2024) Rust for Embedded Systems: Current State and Open Problems. *Proceedings of the* 2024 *on ACM SIGSAC Conference on Computer and Communications Security*, Salt Lake, 14-18 October 2024, 2296-2310. https://doi.org/10.1145/3658644.3690275

[20] Rust vs C Speed and Zero-Cost Abstractions. https://kornel.ski/rust-c-speed

[21] Martins, E.M., Faé, L.G., Hoffmann, R.B., *et al.* (2025) NPB-Rust: NAS Parallel Benchmarks in Rust. arXiv:2502.15536.

[22] Benchmarks: Rust vs C/C++ Performance Comparisons. https://benchmarksgame-team.pages.debian.net/benchmarksgame

[23] SPARK/Ada: Formal Verification for High-Integrity Software. https://www.adacore.com/sparkpro

[24] Ravenscar Profile (Ada Real-Time Subset). https://en.wikipedia.org/wiki/Ravenscar_profile

[25] Peripheral I/O in Go. https://periph.io

[26] Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P., *et al.* (2017) Multiprogramming a 64kb Computer Safely and Efficiently. *Proceedings of the* 26*th Symposium on Operating Systems Principles*, Shanghai, 28 October 2017, 234-251. https://doi.org/10.1145/3132747.3132786

[27] ArceOS: An Experimental Modular OS Written in Rust. https://github.com/arceos-org/arceos

[28] Starry Tutorial Book. https://azure-stars.github.io/Starry-Tutorial-Book

[29] AxVisor Tutorial Book. https://arceos-hypervisor.github.io/doc/architecture/axvisor.html

[30] Low-Code Programming for Event-Driven Applications. https://nodered.org

[31] Chen, M., Tworek, J., Jun, H., *et al.* (2021) Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

[32] Tufano, M., Agarwal, A., Jang, J., *et al.* (2024) AutoDev: Automated AI-Driven Development. arXiv:2403.08299.

[33] Mahmoudi, A., Nešković, A., Thermann, C., Sehm, R., Hübner, C., Plattenteich, T., *et al.* (2025) A Systematic Mapping Study on Systemc/TLM Modeling Capabilities in New Research Domains. *ACM Transactions on Design Automation of Electronic Systems*, **30**, 1-41. https://doi.org/10.1145/3735641

[34] Virtual Components Modeling Library. https://github.com/machineware-gmbh/vcml

[35] Kim, H., Kwak, J. and Cho, J. (2024) Autosar-Compatible Level-4 Virtual ECU for the Verification of the Target Binary for Cloud-Native Development. *Electronics*, **13**, Article 3704. https://doi.org/10.3390/electronics13183704

[36] Hu, M., Cao, E., Huang, H., Zhang, M., Chen, X. and Chen, M. (2023) AIoTML: A Unified Modeling Language for AIoT-Based Cyber-Physical Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **42**, 3545-3558. https://doi.org/10.1109/tcad.2023.3264786

[37] Antmicro Platform Renode. https://antmicro.com/platforms/renode

[38] 深圳航天科技创新研究院[EB/OL]. https://app.puliedu.com/#/portals, 2025-10-10.