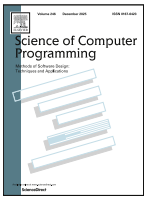
Science of Computer Programming 250 (2026) 103398

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico



# Design and Evaluation of Coconut: Typestates for C++

Arwa Hameed Alsubhi*, Ornela Dardha, Simon J. Gay

*School of Computing Science, University of Glasgow, Glasgow, United Kingdom*

ARTICLE INFO

*Keywords:*
Typestate
C++
Embedded Systems

ABSTRACT

This paper introduces Coconut, a C++ tool that uses templates for defining object behaviours and validates them with typestate checking. Coconut employs the GIMPLE intermediate representation (IR) from the GCC compiler's middle-end phase for static checks, ensuring objects follow valid state transitions as defined in typestate templates. It supports features like branching, recursion, aliasing, inheritance, and typestate visualisation. We illustrate Coconut's application in embedded systems, validating their behaviour pre-deployment. We present an experimental study, showing that Coconut improves performance and reduces code complexity wrt the original code, highlighting the benefits of typestate-based verification.

## 1. Motivation and Significance

Typestate analysis is a program analysis technique that models software components—such as sensor modules, communication interfaces, hardware timers, or peripheral controllers—as finite state machines. A component operates in one of several well-defined states, and each state permits only certain operations [1]. These rules specify the valid sequences of operations in a component's lifecycle. When an operation is invoked, the component transitions to a new state. Typestate analysis [2] checks whether these transitions follow the rules, ensuring the component is used correctly.

This state-dependent discipline is particularly valuable in embedded systems, where components must follow strict operational sequences. For instance, a temperature sensor [3] must be **initialised** before it can **collect** data, and the collected data must be **processed** and **validated** before being **transmitted** to other devices. Executing these steps out of order—such as transmitting data before validation—can lead to data corruption, communication failures, or system instability. Typestate analysis detects such violations at compile time, which helps manage the complexity of large systems and reduces development cost by avoiding failures later in testing or deployment.

Support for such checking exists in several languages and tools that detect incorrect object usage by enforcing valid state transitions during development. Representative examples include Vault [2], Fugue [4], and Obsidian [5], as well as tools such as Mungo and JaTyC for Java [6,7] and Papaya for Scala [8]. While these efforts demonstrate the benefits of typestate, they predominantly target managed runtime environments (e.g., Java, Scala) with garbage collection and relatively large memory footprints. These characteristics are acceptable in desktop contexts but often impose overheads that are impractical on resource-constrained embedded platforms. In embedded robotics, for example, Java implementations have been observed to use more memory and run more slowly than equivalent C++ systems [9]. Consequently, C and C++ remain dominant for embedded development, and typestate tooling built for managed languages is often ill-suited to this setting.

In parallel, safety-critical embedded domains (e.g., healthcare and automotive) often employ formal methods to model and verify behaviour. Two common formalisms are Extended Finite State Machines (EFSMs) [10], which add variables and guards to state

* Corresponding author.
*E-mail addresses:* a.alsubhi.1@research.gla.ac.uk (A.H. Alsubhi), ornela.dardha@glasgow.ac.uk (O. Dardha), simon.gay@glasgow.ac.uk (S.J. Gay).

https://doi.org/10.1016/j.scico.2025.103398
Received 14 December 2024; Received in revised form 19 September 2025; Accepted 7 October 2025
Available online 25 October 2025

**Table 1**
Code metadata.

| Nr. | Code metadata description | |
|---|---|---|
| C1 | Current code version | V-2.0.0 |
| C2 | Permanent link to code/repository used for this code version | https://github.com/CoLab-Glasgow/Coconut |
| C3 | Permanent link to Reproducible Capsule | https://doi.org/10.5281/zenodo.14478714 |
| C4 | Legal Code License | BSD 3-Clause License |
| C5 | Code versioning system used | C++20 , gcc-13, cmake-3.27.1 |
| C6 | Software code languages, tools, and services used | C++, Bash |
| C7 | Compilation requirements, operating environments and dependencies | Linux, macOS, Windows |
| C8 | If available, link to developer documentation/manual | https://github.com/CoLab-Glasgow/Coconut/blob/main/Documentation.md |
| C9 | Support Email for questions | a.alsubhi.1@research.gla.ac.uk |

machines, and Event-B [11], a refinement-based method for mathematically precise system models. While these methods provide strong guarantees, they also introduce practical challenges, as a variety of examples have shown [12–16]. Many of these systems require developers to define specifications separately using dedicated modelling languages and tools. This adds overhead, particularly when system requirements change and both specification and code must be updated and kept consistent. Tools such as Asmeta [17] and IBM Rational Rhapsody [18] can generate large portions of an implementation from the specification, but application-specific code (e.g., hardware integration or custom logic) still need to be written manually and preserved across regenerations. In these implementations, correct usage of components is often enforced at runtime with flags, conditionals, and state variables. This approach may work for small systems, but as systems grow, the state-handling logic becomes dispersed, harder to maintain, and more error-prone. Even minor oversights, such as omitting a single check, can introduce subtle bugs that are difficult to detect through testing or runtime checks.

In contrast, Coconut bridges this gap with a lightweight approach: it integrates typestate checking directly into C++ code using compiler infrastructure, rather than relying on separate modelling languages or tools. By performing the checks at compile time, Coconut prevents incorrect usage before the program is run, reducing reliance on runtime checks and making errors easier to detect early. The first version, Coconut v1 [19], realised this idea using C++ templates. While this enabled compile-time checking, it also introduced two key limitations: long compile times from repeated template instantiations, and limited scalability, since templates could not track usage across functions or modules.

To address these limitations, this paper introduces Coconut v2 [1] (GIMPLE-based): a complete re-engineering of the tool as a plugin for the GCC (GNU Compiler Collection) compiler [20–22]. GCC is one of the most widely used compilers for C and C++ in embedded systems. It compiles code through multiple internal stages, one of which is called GIMPLE IR (Intermediate Representation)—a simplified, structured form of the program used for optimisations and analysis [20–22]. Coconut v2 moves typestate checking directly into this internal compiler stage, improving efficiency by performing checks in a single compiler pass and eliminating the overhead of repeated template instantiations. It also leverages GCC's internals to statically track object state transitions more accurately, including across function boundaries. The code metadata for Coconut v2 can be found in (Table 1).

**Remark 1.** In the rest of this paper, we refer to the original template-based design as Coconut v1 and the new GIMPLE-based design as Coconut v2. Unless otherwise stated (e.g., in Section 2.4), "Coconut" refers to v2 for all techniques, examples, and evaluations.

### *1.1. Contributions*

The paper makes the following contributions:

- **GCC-IR-Based Typestate Checking:** Coconut extends GCC with a plugin that performs typestate checking on the GIMPLE intermediate representation (see Section 2).
- **Interprocedural Alias Analysis:** Coconut introduces context-sensitive interprocedural typestate checking by extending the Papaya algorithm with SSA-based reasoning to support typestate tracking across function boundaries, including in the presence of aliasing (see Section 3).
- **Reproducibility of Timing Measurements:** Coconut is evaluated using a controlled setup that ensures consistent and accurate timing measurements (see Section 4).

## 2. Software Framework

Coconut brings typestate analysis to C++ by providing lightweight templates for declaring protocols. A typestate checker then verifies conformance primarily at compile time and flags any protocol violations. The tool applies to general-purpose C++ development, with embedded systems being particularly relevant because (i) C/C++ dominate this domain and typically incur lower overhead than managed languages (e.g., Java, Scala), where most prior typestate tools were developed; (ii) many embedded toolchains are GCC-

[1] Artefact avalible at: 10.5281/zenodo.14478714.

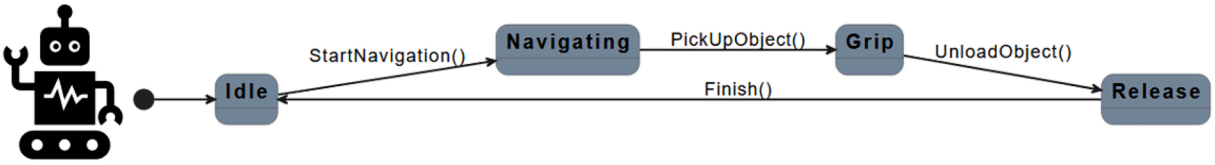



**Fig. 1.** Robot State Machine.

or GCC-derived [23], allowing Coconut's plugin to run within standard builds; and (iii) embedded components often follow strict operational sequences that typestate naturally enforces.

This section explains how protocols are declared and enforced in Coconut, using a warehouse robot controller as a running example—a typical embedded component with a strict operational sequence. It then outlines the GIMPLE-SSA analysis and the targeted runtime fallback, and finally contrasts Coconut v1 (template-based) with Coconut v2 (GIMPLE-based).

### 2.1. Running Example: Warehouse Robot

To illustrate Coconut's workflow, we use a warehouse robot controller that performs a fixed sequence of actions: navigate to a source location, pick up an object, deliver it to a destination, and return to the idle state. This behaviour is implemented by the `Robot` class shown in Listing 1.

```
1  class Robot {
2  public:
3   void StartNavigation(const char* source) {
4      printf("Start navigation to %s\n", source);}
5   void PickUpObject(const char* object) {
6      printf("Arrived at source. Picking up object %s\n", object); }
7   void ReleaseObject(const char* destination, const char* object) {
8      printf("Arrived at destination %s and release %s\n", destination,
       object); }
9   void Finish() {
10     printf("Task complete. Returning to Idle...\n");}};
```

**Listing 1.** Robot Class

To function safely and correctly, the robot must follow a strict order. The robot: **(1)** Starts in an `Idle` state and transitions to `Navigating` state to move to a source location. **(2)** Once the robot reaches the source, it transitions to `Grip` state to hold an object from the source location. **(3)** The robot transitions to `Release` state, where it moves to the destination and releases the object. **(4)** After completing its task, the robot returns to `Idle` and waits for the next move command. This sequence is represented in Fig. 1.

This order reflects physical and safety-critical requirements. For instance, if `ReleaseObject` is called before the robot has picked up anything, the system may try to open an empty gripper in mid-movement, potentially damaging the hardware or confusing downstream processes [24]. Likewise, calling `PickUpObject` before the robot has reached its destination may cause it to activate the arm in the wrong location, risking a collision or mechanical failure. The C++ type system does not prevent these errors: all public methods remain callable at all times. As a result, incorrect sequences often go undetected until late testing or deployment phases, where faults are more costly to identify and fix. To address these issues, Coconut enforces the protocol at compile time, preventing violations and catching misuse early.

### 2.2. Defining and Enforcing Protocols with Coconut

To enforce the correct usage of the `Robot` class, Coconut provides two main components:

1. **A header-only library,** which allows developers to declare typestate protocols via templates.
2. **A typestate checker,** which builds an internal finite-state machine (FSM) from those templates and then analyses the program, checking each method call against the declared typestate transitions.

***Defining Typestate Rules With templates.*** The Coconut library provides three core templates that are used to define typestate protocols. The first is `State`, which describes a single transition: it specifies a current state, a method that is allowed in that state, and the resulting state if the method is called. In the format:

```
State<Current_State, Allowable_Method, Next_State>
```

The second is `Typestate_Template`, which collects multiple `State` templates into a complete protocol that outlines the valid operational behaviour of a class. The `Robot_Typestate` in Listing 2 encompasses all transitions for the `Robot`, with line 2 representing the first state transition and lines 3, 4, and 5 corresponding to the second, third, and fourth transitions, respectively.

```
using Robot_Typestate = Typestate_Template<
State<RobotState::Idle,&Robot::StartNavigation,RobotState::Navigating>,
State<RobotState::Navigating,&Robot::PickUpObject,RobotState::Grip>,
State<RobotState::Grip,&Robot::ReleaseObject,RobotState::Release>,
State<RobotState::Release,&Robot::Finish,RobotState::Idle>>;
```

**Listing 2.** Robot Typestate Specfications

The third template, `TypestateClassConnector`, links a class to its protocol and flags it for typestate checking during compilation, as shown in Listing 3.

```
TypestateClassConnector<Robot, Robot_Typestate> Robot_Flag;
```

**Listing 3.** Robot Typestate Flag

***Two-Stage Typestate Enforcement.*** Coconut enforces typestate protocols in two main stages during compilation.

**Template Instantiation Phase.** In the first stage, Coconut hooks into the C++ compiler when template instantiation occurs—that is, when the compiler takes the typestate template and fills in the actual class names, method pointers, and states to create concrete type information. At this point, Coconut extracts the full typestate specification and builds an internal finite-state machine (FSM) model. This model records, for each state, which methods are allowed and how each method call updates the object's state. The FSM is then used to guide the program-wide typestate checking in the next analysis phase.

**Program GIMPLE Analysis Phase.** Once the typestate model is built, Coconut performs the second stage of enforcement during the middle-end phase of the GCC compilation process. At this point, the C++ source code has already been parsed into an abstract syntax tree (AST) and lowered into GIMPLE, a simplified intermediate representation used by GCC for optimisation and analysis. GIMPLE expresses code in three-address form and uses Static Single Assignment (SSA), where each variable has exactly one definition. This representation makes it easier to track how values flow through a program. Coconut checks typestate usage at this level by examining each method call and determining which object the call applies to. The analysis proceeds differently depending on how the object is referenced:

(i) **Simple cases.** When a call is made directly on an object or through a single temporary variable, Coconut statically verifies that the call is valid in the current typestate, updates the state accordingly, or reports a violation. To illustrate, consider `Robot` in the client code in Listing 4.

```
1 int main() {
2     Robot robot;   // Idle
3     robot.StartNavigation("shelf03901"); // Navigating
4     robot.PickUpObject("Box1837"); // Grip
5     robot.ReleaseObject("shelf04759", "Box1837"); //  Release
6     robot.Finish(); // Idle
7     return 0;
8 }
```

**Listing 4.** Robot Client Code

When compiled with the Coconut plugin enabled, GCC lowers the C++ to GIMPLE-SSA. Conceptually, a call such as line 3 in Listing 4 appears as a temporary holding the address of the receiver and a `GIMPLE_CALL` (`gcall`) that takes that pointer as its first argument, for example:

```
D.2 = &robot;
gcall StartNavigation(D.2, "shelf03901");
```

Here, the freshly constructed `Robot` object is named `robot`, and Coconut records its initial typestate as `Idle`. Coconut then scans each `gcall` statement in the GIMPLE-SSA. The first statement loads the address of `robot` into the temporary `D.2`; the second invokes `StartNavigation` with that address. Coconut traces `D.2` back to `robot` to identify the receiver and its current state (`Idle`). It then consults the finite-state model built in the first phase—a table of rules of the form (current state, method) → next state. For this call, the model contains (`Idle`, `StartNavigation`) → `Navigating`, so the call is valid. Coconut accepts it and updates the tracked state to `Navigating`. If no rule exists—for example, if lines 3 and 4 in Listing 4 are swapped so the program attempts (`Idle`, `ReleaseObject`)—the checker rejects the call and emits a compile-time error. This illustrates the simple case, where calls are made directly on the object or through a single temporary.

(ii) **Intermediate cases.** When objects are reassigned, copied into temporaries, or passed through functions, SSA makes the analysis simpler because each temporary has only one definition, which allows value flows to be traced back to their origin without ambiguity in these cases. Coconut uses GIMPLE's SSA internal APIs to follow these links, check statically that each call is legal in the current protocol state, update the state, or stop compilation on violation.

(iii) **Complex cases.** When complex situations occur, such as control flow branches and later rejoins, for example, after an `if` statement. At these join points, SSA introduces $\phi$ nodes that merge values coming from different paths. As a result, a single variable may now refer to several possible objects; this situation is known as a may-alias. Coconut conservatively approximates these cases by computing alias sets and attempting to statically verify typestate compliance across all possible targets. If this analysis cannot prove safety, Coconut does not reject the program outright. Instead, it issues a warning and defers typestate checking to runtime for the affected execution. This hybrid strategy maintains enforcement soundness while avoiding unnecessary rejections caused by over-approximation.

In short, SSA makes value tracking straightforward in simple and intermediate cases. For complex cases with aliasing or complex joins, Coconut uses conservative approximation and inserts targeted runtime checks when static proof is inconclusive, preserving overall enforcement soundness. These intermediate and complex cases are explained with examples in Section 3.

### 2.3. Dynamic Typestate Compliance Checking

Coconut enforces typestate statically by default, as it is computationally cheaper and introduces less overhead. When static verification is infeasible or overly conservative, we provide a fallback mechanism that triggers runtime typestate monitoring to preserve enforcement soundness without blocking progress.

**Sequential deterministic code.** For regular control flow (e.g., Listing 4 or with branches, loops, and interprocedural calls), Coconut's static analysis explores all relevant paths and proves that typestate violations cannot occur. Thus, no runtime checks are emitted. When the analysis cannot establish safety — for example, due to complex aliasing where an object reference may be reassigned at runtime (rare in embedded systems) — Coconut issues a warning and enables a runtime monitor for the affected execution. This monitor dynamically tracks the object's state and checks typestate conditions at runtime, once the actual reference is resolved. This ensures that violations are caught precisely when they occur, even if static analysis could not guarantee safety. The fallback mechanism allows the program to compile and run while maintaining typestate correctness during execution. Because runtime monitoring is activated only where necessary, the overhead remains minimal.

**Concurrent (nondeterministic) code.** In multithreaded programs, enumerating all possible thread interleavings statically is generally intractable. Different threads may invoke state-changing methods in different orders, and the concrete interleaving is unknown at compile time. Consider Listing 5: both threads operate on the same `robot1`. If `StartNavigation` executes before `PickUpObject`, the typestate specification is respected; if `PickUpObject` executes first, it is a violation. Because Coconut does not attempt to prove typestate correctness for all interleavings statically, it activates a runtime monitor that detects violations during execution.

```
1  int main() {
2      Robot robot1;
3      std::thread t1([&]() {
4          std::cout<<"T1 ..\n";
5          robot1.StartNavigation("shelf03901");
6      });
7      std::thread t2([&]() {
8            std::cout<<"T2 ..\n";
9          robot1.PickUpObject("Box1837");
10     });
11     t1.join();
12     t2.join();
13     return 0;
14 }
```

**Listing 5.** Robot in Threads

When a runtime monitor is needed, Coconut maintains a lightweight mapping from object identities to their current typestate. On each invocation of a method that affects or depends on typestate, the monitor consults this mapping to verify that the receiver object is in a permitted state. If the check passes, the method proceeds and the typestate is updated; if it fails, the monitor reports the violation and halts execution. In this example, if `t1` runs first and completes, the check passes; if `t2` runs first, the monitor reports a violation and terminates execution.

To summarise, when a typestate property is verified statically, no runtime check is emitted (no duplication). When static analysis cannot determine safety, Coconut enables monitoring during execution. This preserves soundness while avoiding redundant checking and keeping overhead minimal by activating monitoring only when needed.

### 2.4. Template-based (Coconut v1) vs GIMPLE-based (Coconut v2) Designs

This section sets the design contrast between Coconut v1 (template-based) and Coconut v2 (GIMPLE-based), highlighting what is new in v2.

***Coconut v1 (template-based).*** In Coconut v1 [19], all typestate logic— typestate rules, and enforcement—is implemented using C++ templates. When a method is called on an object governed by a typestate protocol, Coconut injects compile-time checks via

specialised templates that are instantiated by the compiler. These templates form part of Coconut's library and implement the logic to validate whether a given method is permitted in the object's current state. This is done through **template instantiation**—a process where the compiler generates concrete code from a template by substituting specific types or values. For example, in Listing 4, line 3, a call to `robot.StartNavigation()` triggers the compiler to instantiate a new version of a checker template. This checker compares the object's current state and the method being invoked against the typestate protocol's declared transitions. If a matching rule is found, the transition is allowed, and the next state is computed at compile time; otherwise, the compiler generates an error. Internally, Coconut causes the compiler to perform logic conceptually similar to Listing 6.

```
1 Typestate_Check<RobotState::Idle,&Robot::StartNavigation>::Validate();
2     using result = is_valid_transition<
3     RobotState::Idle, &Robot::StartNavigation,
4     State<...>, State<...>, State<...>, State<...>
5 >::result;
```

**Listing 6.** Compiler Logic

This means that every usage of a typestate-controlled method triggers a new compile-time check. If an object goes through multiple states, the compiler must instantiate a new checker template for each transition along that path. And if those transitions appear in multiple parts of the code, each context generates additional template instantiations. The compiler must repeatedly re-evaluate template logic, perform type comparisons, and instantiate checkers — all of which can lead to significantly longer compile times in larger programs. Furthermore, Coconut v1 performs static typestate checking locally at each method call during template instantiation. However, it does not retain an object's typestate once the object is passed into another function. As a result, typestate protocols cannot be enforced across function boundaries or file scopes. This limits Coconut's ability to analyse modular or large-scale programs, where object usage is often distributed across multiple functions and components.

***Coconut v2 (GIMPLE-based).*** Coconut v2 addresses this limitation by shifting typestate checking from C++ templates to GCC's GIMPLE intermediate representation. This choice was motivated by GIMPLE's ability to represent the entire program in a simplified, SSA-based format, enabling accurate tracking of object lifecycles across functions, files, and control-flow paths. Unlike templates, which are expanded locally and lack interprocedural context, GIMPLE allows Coconut to perform typestate analysis efficiently in a single compiler pass, reduce redundancy from template expansion, and scale to larger, modular codebases—capabilities essential for real-world embedded systems. Moving beyond v1's local, per-function checking, v2 performs context-sensitive interprocedural analysis, ensuring that each method call respects the expected protocol even when operations are spread across different functions or modules.

## 3. Interprocedural Analysis

Typestate analysis ensures that objects are used according to a predefined protocol. For example, a robot must follow the rules defined in Listing 2: it must initiate navigation before picking up an object, release it after picking, and terminate at the end. However, in real-world programs, protocol-relevant operations are often distributed across multiple functions. An object might be created in one function, passed to another, modified indirectly, and eventually returned or finalised elsewhere. This section describes how Coconut addresses this challenge through context-sensitive interprocedural analysis that tracks protocol state across function boundaries.

### *3.1. Example: Coordinating Robot Behaviour via Operator*

In many object-oriented designs, one class manages or coordinates the behaviour of another by invoking methods on an object it receives from elsewhere. This structure appears in the following example, where a `Robot` object is controlled by a separate `Operator` class.

```
1 class Operator{
2     public:
3     void automatePickingReleasing(Robot& r){
4         r.PickUpObject("Box1837");
5         r.ReleaseObject("shelf03901","Box1837");}};
```

**Listing 7.** Operator Class

As shown in Listing 7, the `Operator` does not create the `Robot`, but receives a reference to it and performs operations that represent key transitions in the robot's usage protocol.

This class alone does not reveal the full context of the object's lifecycle. That context is provided by the client code in Listing 8. In the example, the `robot` instance is created in `main()`, passed by reference to the `Operator`'s `automatePickingReleasing()` method, and later finalised with a call to `Finish()`. Crucially, protocol-relevant state changes—such as picking up and releasing an object—occur within the callee, not the caller. A sound typestate analysis must recognise that references like `r1` and `r2` alias the same `robot` instance, and that operations in one function affect the object's state in others. To support this level of interprocedural reasoning,

Coconut propagates typestate transitions across function boundaries and uses SSA-based analysis to simplify alias resolution and reference tracking. The next section outlines how this mechanism works in detail.

```
int main() {
   Robot robot;
   Robot& r1 = robot;
   robot.StartNavigation("shelf03901");
   Operator operator1;
   operator1.automatePickingRereleasing(r1);
   Robot& r2 = robot;
   operator1.automatePickingReleasing(r2);
   robot.Finish();
   return 0;}
```

**Listing 8.** Robot-Operator Client Code

### *3.2. Handling Protocol Transitions Across Calls*

Coconut performs interprocedural typestate analysis to ensure that objects follow their usage protocols even when accessed across function boundaries. It tracks the state of each object throughout the program, including when objects are passed as parameters, returned from functions, or modified in callees. The analysis operates globally, combining typestate tracking with SSA-based alias reasoning to maintain a context-sensitive view of how each object evolves. To carry out its analysis, Coconut maintains four main data structures for each object o:

1. `State`[$o$]: The object's current typestate; read before a typestate-relevant call, update after a valid transition.
2. `Aliases`[$o$]: List of SSA names that definitely refer to $o$; whenever a new must-alias appears (assignment, parameter, return), prepend it so it becomes the head; remove names when leaving scope. Used to name the current receiver and build context.
3. `AliasSet`[$v$]: Possible target objects for SSA var $v$ used when the receiver is not unique; enumerate at the call site.
4. `Summary`: Global memorisation table that maps a 4-part context key to the post-call typestate (or error). Used only for interprocedural calls.

Coconut analyses the program incrementally, updating its data structures whenever an object is used. When a new object is created, it is assigned an identity, an initial state, and its first alias from the SSA variable at creation. If another variable later refers to the same object, SSA form introduces a fresh variable, which Coconut places at the head of the alias list so the current reference is always available. For direct method calls, Coconut checks the current state and applies the transition immediately if valid.

Interprocedural analysis occurs when an object is passed into another function. Coconut builds a context key from the object's identity, the callee, the most recent alias and the current state. If the key is already in `Summary`, the stored result is reused; otherwise, the callee is analysed, state changes are applied, and the outcome is recorded. On entry of the callee, the parameter name (a fresh SSA variable) is added to the alias list; on exit, it is removed, keeping alias information consistent across calls.

**Robot example (cont'd).** We illustrate these steps with the `Robot` example in Listing 8. At each stage, we show how Coconut uses and updates its data structures and how it determines whether transitions are valid or invalid across function boundaries.

We start with object creation: when a new object is declared, Coconut assigns it an ID, sets its initial state, and records its first alias.

**Line 2:** `Robot robot;`
→ New object created and assign ID $o_1$
→ Updated data structures:
`State`[$o_1$] = Idle, `Aliases`[$o_1$] = [robot]
Summary: not used

Next, when another SSA name is introduced for the same object (via assignment, parameter, or return value), that name is added to the front of `Aliases`.

**Line 3:** `Robot& r1 = robot;`
→ New alias created, then add `r1` to alias list
`Aliases`[$o_1$] = [r1,robot],
State[$o_1$] unchanged (Idle), Summary: not used

Then, when typestate-object calls a method directly, Coconut checks if it is valid in the current state. If valid, the state is updated immediately. No summary lookup is needed.

**Line 4:** `robot.StartNavigation(...)`
→ Direct call (no context key needed)
→ Check `State`[$o_1$] = Idle → valid transition
`State`[$o_1$] = Navigating
Aliases[$o_1$] unchanged, Summary: not used

Line 5: `operator1` is created as a non-typestate object, serving as the receiver in Line 6. Its allocation does not affect any data structures for $o_1$.
Finally, when a typestate-object is passed into another function, Coconut builds a *context key* with four parts: (object ID, function name, most recent alias, current state,). It then checks the `Summary` table: If the key is found, Coconut reuses the stored result. If not, it analyses the function body in that context, updates the object's state, and records the result in `Summary`. This allows typestate information to be propagated soundly across function calls while avoiding redundant computation.

**Line 6:** `operator1.automatePickingReleasing(r1)`
→ Cross-function call, key not in Summary
→ Inside function: parameter `r` is added
`Aliases`[$o_1$] = [r, r1, robot]
Operations: PickUpObject (Navigating → Grip),
ReleaseObject (Grip → Release)
→ After call: `State`[$o_1$] = Release,
`Summary`[($o_1$,`automatePickingReleasing`,`r1`,`Navigating`)] = Release
→ Exit function, remove parameter `r`
`Aliases`[$o_1$] = [r1, robot]

Later, another alias can be introduced for the same object. Coconut simply adds this new name to the alias list while the state and summary remain unchanged.

**Line 7:** `Robot& r2 = robot;`
→ Add new alias, r2 is now head
`Aliases`[$o_1$] = [r2,r1,robot]
State[$o_1$] unchanged (Release), Summary unchanged

After that, the object is passed into the same function again, but now its state is `Release`. Coconut builds a new context key. Since this key is not in `Summary`, it analyses the function body. The first method inside the callee, `PickUpObject()`, is invalid in the `Release` state. At this point, Coconut raises an error and stops the compilation. On exit, the temporary parameter alias is removed, and the alias list returns to its previous form. Both interprocedural calls are depicted in Fig. 2.

**Line 8:** `operator1.automatePickingReleasing(r2);`
→ Cross-function call, key not in Summary
→ Inside function: parameter `r` is added
`Aliases`[$o_1$] = [r, r2, r1, robot]
First call PickUpObject — invalid in Release (Error reported)
→ Exit function, remove parameter `r`
`Aliases`[$o_1$] = [r2, r1, robot]

**Remark 2.** This approach builds on the Papaya global typestate tracking algorithm [8], which uses a simpler context based only on object identity and method name. Coconut refines this model in two main ways: *i*) by combining SSA form with must-alias analysis [25] to improve tracking of object references; *ii*) by using a richer context key that includes the object's current typestate and alias head, along with the object identity and callee method. Consequently, Coconut performs a context-sensitive interprocedural analysis.

### 3.3. Alias Resolution in Coconut

A **must-alias** relationship occurs when two variables are guaranteed to refer to the same object instance, and this can be determined statically. Coconut records this information in per-object lists: for each object $o$, `Aliases[o]` contains the SSA variables that definitely refer to $o$. These must-alias lists provide the current reference to $o$ and are used when building context keys for interprocedural summaries, as illustrated in the previous example.

In contrast, a **may-alias** relationship arises when a variable might refer to one of several possible objects, but the exact target cannot be resolved statically. This situation commonly occurs due to branching control flow, conditional assignments, or indirect access through pointers. Listing 9 shows a case where the variable `ptr` is conditionally assigned to either `&robot1` or `&robot2`, depending on the result of a runtime sensor reading. At compile time, the exact target of `ptr` cannot be resolved statically, so Coconut conservatively treats this as a may-alias scenario.

When Coconut analyses this program in SSA form, each branch assignment is given a fresh SSA variable: `ptr_1` in the first branch and `ptr_2` in the second. At the merge point, SSA introduces a fresh $\phi$ variable that unifies the two definitions:$ptr_\phi$ =

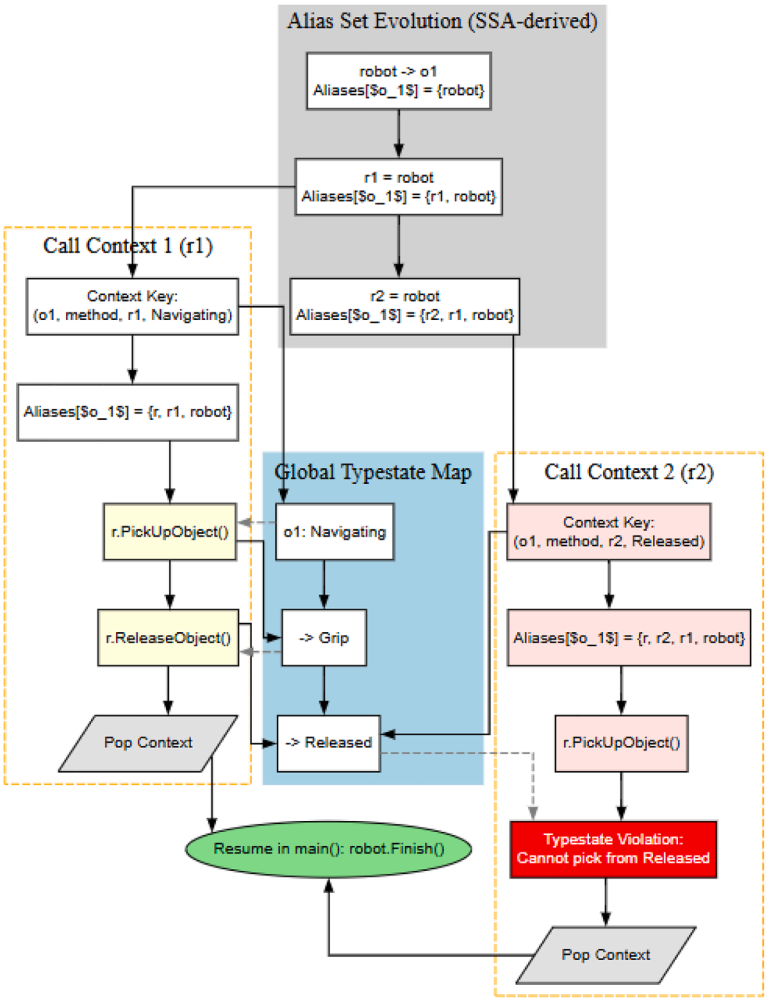



**Fig. 2.** IPA Analysis.

```
1  int main(){
2  Robot robot1, robot2;
3  robot1.StartNavigation("shelf03901");
4  Robot* ptr;
5  int currentZone = getSensorReading();  // runtime sensor check
6  if (currentZone ==1) {
7  ptr = &robot1; }
8  else {
9  ptr = &robot2; }
10 ptr->PickUpObject("Box1837");
11 return 0;}
```

**Listing 9.** Client Code with May-Alias

$\phi$(ptr_1, ptr_2). Here, ptr_1 refers to robot1 and ptr_2 refers to robot2, based on the assignments in their respective branches. Coconut therefore determines that $ptr_\phi$ may alias either robot1 or robot2, and records this uncertainty in a new dataset:AliasSet($ptr_\phi$) = {robot1, robot2}.

At the method call, SSA represents line 10 as $ptr_\phi$->PickUpObject("box"). Coconut consults AliasSet($ptr_\phi$) to enumerate the possible targets and checks each object's State. If the operation is valid for all possible targets, the transition is applied at compile time (which is not the case in this example). However, if the analysis detects that the call may be invalid for one but not the other potential aliases (e.g., it is invalid for robot2 here), Coconut does not reject the program. Instead, it issues a warning and defers the

check to runtime. During execution, the system monitors the actual target and applies the typestate check dynamically, once the alias is resolved.

**Call:** $ptr_\phi$->PickUpObject("box")
Check AliasSet($ptr_\phi$). For each object:
→ robot1: Navigating → Grip (valid)
→ State[robot1] = Grip
→ robot2: Idle → invalid
Warning: Typestate violation detected;
fallback to runtime monitoring

In this example, Coconut only uses the `AliasSet[v]` dataset, which ensures soundness when a variable may point to multiple objects. The per-object `Aliases[o]` are not involved here, since no must-alias relationship can be established for $ptr_\phi$. Similarly, the interprocedural `Summary` is not consulted because the call is direct. By conservatively accounting for all possible targets and deferring checks when necessary, Coconut preserves enforcement soundness even when the exact alias cannot be determined statically.

## 4. Comparative Analysis Study

In this section, we evaluate Coconut v2 on three embedded C++ applications, each reflecting a widely used strategy for enforcing behavioural correctness under hardware constraints: (1) manual control-flow logic, (2) runtime enforcement with model-based state machines, and (3) compile-time checking with a macro-based typestate tool.

Embedded systems must run predictably on limited CPU, memory, and power [26], so the choice of verification method directly affects both performance and maintainability. Measuring these effects is important for assessing which techniques are practical under such constraints. By re-implementing each case study with Coconut's static typestate system, we compare it against manual, dynamic, and static approaches that are currently used in practice. Specifically, we pursue two objectives:

Objective 1: Assess whether Coconut improves performance in embedded system development compared to existing methods.
Objective 2: Assess whether Coconut reduces code complexity in embedded system development compared to existing methods.

### 4.1. Benchmarks

Each of the three selected case studies is a previously published/reference implementation that represents a common way of enforcing behavioural correctness in embedded C++ systems. While their enforcement strategies differ—manual, runtime, or static—the underlying goal is the same: to ensure that usage protocols and state transitions are respected. We describe the original design and verification method for each benchmark and then explain how it was re-implemented with Coconut. Only the Coconut variants were re-engineered—moving enforcement to the Coconut typestate checker—while the prior implementations were left unchanged. In these benchmarks, all call sites were proved safe statically, so Coconut emitted no runtime checks and public APIs/externally observable behaviour were preserved.

**LightSwitch** is a simple embedded application that controls a digital light (switch on/off) through a microcontroller [27]. In the original implementation, correctness is enforced manually: the LightSwitch object exposes methods like `switchOn()` and `switchOff()`, but the code relies on if/else checks around every call to prevent illegal sequences (e.g., calling `switchOff()` twice in a row). These are runtime checks, because the program evaluates the object's state at execution time before deciding which branch to take. No static guarantee exists, so errors can only be caught (or ignored) when the program runs.

In the Coconut implementation, the same behaviour is re-expressed using a typestate specification template, which declares legal states and transitions directly. Coconut's static checker enforces these rules at compile time, ensuring that invalid method sequences are rejected before the program can even run, which eliminates the need for runtime conditionals around object usage that exist in the original.

**PillBox** is a medication dispenser application that alerts patients to take medicine at scheduled times [12]. The original `PillBox` is specified using Abstract State Machines (ASMs) in the Asmeta framework. Its behaviour includes scheduling doses and moving between states such as `Idle`, `Activating`, and `Dispensing`. These ASM models were translated into C++ with the Asm2C++ tool, yielding an object-oriented implementation where `PillBox` is defined as a class with three drawers, but its methods enforce correctness through long `if/else` and `switch` blocks. In this setup, state validity is not checked statically; instead, every operation is guarded dynamically at runtime.

In contrast, Coconut `PillBox` is re-implemented as a class with three drawers with the same public API, but its legal states and transitions are declared in a typestate specification template using the Coconut typestate library rather than being embedded in control flow. For example, transitions such as `Idle` → `Activating` → `Dispensing` are expressed declaratively in the typestate template. Coconut's checker plugin, following the same approach described in Section 2, enforces these constraints at compile time, ensuring correctness without runtime guards or long conditional structures.

**HTTP-Connection** is a networking utility that constructs an HTTP request by enforcing a valid sequence of method calls—such as setting headers and body—before sending. It was implemented in C++ as a class and verified using the typestate analysis tool ProtEnc [28]. ProtEnc encodes usage protocols with C++ macros. A macro is a preprocessor directive that expands into extra code before compilation. In ProtEnc, each macro inserts additional functions and hidden state variables around the original methods. Instead of

calling the method directly, the program calls the generated function, which updates the protocol state and then forwards the call. This mechanism is validated at *compile time* and it does not rely on runtime checks, but the extra functions and state-tracking logic remain in the compiled binary.

In the Coconut implementation, we keep the same class and public methods. The protocol is declared directly in a typestate template and validated at compile time with the Coconut typestate checker plugin pass, which extensively analyses the semantics of the whole program in GIMPLE form without generating intermediate functions in the binary.

### 4.2. Metrics and Evaluation Setup

We designed the evaluation using predefined benchmarks, selected performance and complexity metrics, and a controlled test environment to ensure reliable and reproducible results. The evaluation follows the ISO/IEC 25010 software quality standard [29], which outlines how to assess attributes, including performance efficiency and maintainability.

**Performance metrics.** To evaluate Objective 1 (performance), we measured compile time, runtime, memory usage, and CPU utilisation. These metrics were chosen because they directly impact the efficiency and resource use of embedded systems. Compile time reflects the cost of static analysis during build, while runtime indicates execution responsiveness. Memory and CPU usage provide insights into how well the system performs under resource constraints. These metrics are standard in embedded systems evaluation and widely adopted in prior studies [30,31].

**Complexity metrics.** To evaluate Objective 1 (Complexity), we used three common measures. Cyclomatic complexity shows how many independent paths exist in the code, which helps estimate its logical complexity. Non-comment lines of code (NLOC) count the number of actual code lines, excluding comments and blanks, to reflect code size. Token count refers to the total number of basic code elements, such as keywords and variables, reflecting how detailed or dense the code is. These metrics are commonly used to assess code maintainability, structural complexity, and the effort required to test and analyse a program [32,33].

**Experimental Environment and Setup.** All tests were run in a VirtualBox virtual machine using Ubuntu 22.04 (64-bit), with 2 virtual CPU cores and 4 GB of RAM. The virtual machine is used to provide a clean, isolated, and repeatable environment for all experiments. The benchmarks and scripts also run natively outside the VM, but all results reported in Section 4.3 were collected inside the VM.

We started by collecting performance data using Bash scripts that called standard Linux tools: `time` (to measure how long it takes to compile and run the program), `top` (run in batch mode to sample process CPU%), and `ps` (to measure memory usage). These tools were chosen because they provide reliable, low-overhead system measurements and are easy to automate. Each performance metric for each program was measured over **100 runs** to reduce the effect of random variation and ensure the consistency of the average results. We report the arithmetic mean to represent the typical outcome, along with the standard deviation (SD), which quantifies how much the values varied across runs. A smaller SD indicates that the results were more consistent, thereby increasing the reliability of the measurements. This setup differs from our earlier work [19], which timed runs by recording timestamps immediately *before* starting the program and *after* it finished (the "before/after" method). Because that includes a small amount of start-up and wait time, it slightly overstates very short runs and increases variability. In this paper, we rely on system tools that measure the program itself—`time` for duration, and `top`/`ps` for CPU and RSS—thereby avoiding that extra cost and improving consistency and reproducibility.

After measuring performance, we evaluated code complexity using standard command-line tools in the same virtual machine. Cyclomatic complexity was measured using `pmccabe`, a lightweight tool that analyses C/C++ source code. NLOC were computed using `grep` and `wc`, filtering out blank lines and comment syntax. Token count was estimated using `wc -w` to count lexical tokens directly from the source files. Unlike performance tests, these complexity metrics are based on static code analysis and produce deterministic results. we use minimal Bash scripts that directly invoke the same system-level tools described in the experimental environment above.

### 4.3. Results and Conclusion

Now, we present the outcomes of our comparative evaluation. The results indicate that Coconut generally improves runtime performance and reduces resource consumption compared to existing implementations.

**Dynamic baselines (LightSwitch, PillBox).** In these benchmarks, the prior versions enforce correctness at runtime via per-call `if/else` guards or `switch(state)` dispatch. The Coconut versions declare the protocol in typestate templates and check it at compile time using Coconut's compiler pass. As expected, moving checks out of execution lowers runtime and memory use but increases compile time. For example, in `PillBox` as shown in Table 2, runtime drops from 13 ms to 7 ms and memory from 1900 KB to 1700 KB, while compile time rises from 120 ms to 200 ms. `LightSwitch` shows smaller runtime gains (10 ms to 8 ms) and a more visible compile-time increase (66 ms to 150 ms), reflecting its small size.

**Static baseline (HTTP-Connection with ProtEnc).** Both Coconut and ProtEnc enforce protocols at compile time, but by different mechanisms. ProtEnc encodes protocols with C++ macros that expand during preprocessing into additional functions and hidden state variables that remain in the binary. Coconut validates typestate specifications in a compiler pass after parsing and does not generate intermediate functions. This explains the trade-off observed in Table 2: Coconut's compile time is higher (260 ms vs. 170 ms) because of the analysis pass which extensively analyses the whole semantic of the program, but runtime and memory use are lower (5 ms vs. 12 ms; 1500 KB vs. 2000 KB) because no extra call layers or state keeping are compiled into the executable.

**Variability and reliability.** SDs are small across metrics, indicating consistent measurements over 100 runs. For example, `PillBox` memory under Coconut averages 1700 KB with SD 10 KB ($\approx 0.6\,\%$), and `HTTP-Connection` Coconut compile time averages 260 ms

**Table 2**
Comparison of performance metrics: Existing formal approaches vs. Coconut v2. each benchmark was executed 100 times, and all reported metrics are the arithmetic mean across those runs.

| Metrics | LightSwitch | | PillBox | | HTTP-Connection | |
|---|---|---|---|---|---|---|
| | Prior [27] | Coconut | Prior [12] | Coconut | Prior [28] | Coconut |
| Compile-Time(ms) | 66 ▼ | 150 ▲ | 120 ▼ | 200 ▲ | 170 ▼ | 260 ▲ |
| ± SD | 3.5 | 4.3 | 2.8 | 3.2 | 4.5 | 4.8 |
| Run-Time(ms) | 10 ▲ | 8 ▼ | 13 ▲ | 7 ▼ | 12 ▲ | 5 ▼ |
| ± SD | 0.3 | 0.8 | 0.5 | 0.6 | 0.7 | 0.5 |
| Memory-Usage(KB) | 30 ▲ | 27 ▼ | 1900 ▲ | 1700 ▼ | 2000 ▲ | 1500 ▼ |
| ± SD | 1.4 | 1.7 | 12 | 10 | 13 | 10 |
| CPU(%) | 1.1 ▲ | 0.9 ▼ | 1.97 ▲ | 1.65 ▼ | 3.3 ▲ | 2.7 ▼ |
| ± SD | 0.05 | 0.07 | 0.1 | 0.08 | 0.2 | 0.15 |

**Table 3**
Comparison of code complexity: Existing formal approaches vs. Coconut v2.

| Metrics | LightSwitch | | PillBox | | HTTP-Connection | |
|---|---|---|---|---|---|---|
| | Prior [27] | Coconut | Prior [12] | Coconut | Prior [28] | Coconut |
| NLOC | 36 ▲ | 35 ▼ | 389 ▲ | 140 ▼ | 84 ▲ | 45 ▼ |
| Tokens | 143 ▲ | 68 ▼ | 1404 ▲ | 484 ▼ | 812 ▲ | 145 ▼ |
| Cyclomatic-Complexity | 7 ▲ | 4 ▼ | 90 ▲ | 22 ▼ | 5 ▲ | 4 ▼ |

with SD 4.8 ms (≈ 1.8 %). `LightSwitch` runtimes have SDs below 1 ms for both variants. In general, coefficients of variation are under 2 % for compile-time and CPU, and under 1 % for memory, supporting the stability of the observed differences.

**Code complexity.** Table 3 shows that Coconut reduces NLOC, token count, and cyclomatic complexity, with the largest effects in the larger benchmarks. This reduction occurs because the prior implementations often encode protocols with nested `if/else` or `switch` statements, which increases control-flow paths and code size. Coconut lets developers describe allowed states and transitions in typestate templates (Section 2) and its compiler pass enforces these rules at compile time. In the dynamic baselines, this removes embedded runtime guards; in the static baseline, it avoids macro-expanded protocol code. Concretely, `PillBox` NLOC falls from 389 to 129 and cyclomatic complexity from 90 to 22; in `HTTP-Connection`, token count drops from 812 to 145. As a result, the code is simpler, with fewer execution paths to consider, making it easier to understand, test, and maintain.

To summarise, Coconut enhances performance and simplifies code. These improvements are more noticeable and beneficial in larger, complex applications than in smaller ones.

## 5. Conclusion and Future Work

We presented Coconut, a GCC/GIMPLE-based typestate system for embedded C++ that lets developers declare usage protocols in templates and have them enforced by a compiler pass. Coconut builds an FSM from those declarations, performs context-sensitive interprocedural analysis over GIMPLE with SSA-based alias reasoning, and falls back to selective runtime checks only when static proof is inconclusive. We evaluated Coconut on three programs representative of common enforcement strategies—ad-hoc runtime guards, model-driven code with runtime checks, and a fully static macro tool—and found that Coconut reduced runtime and memory for the dynamic baselines and outperformed the static macro approach at the cost of higher compile time from whole-program analysis. It also reduced code complexity, simplifying maintenance. For future work, we plan to move beyond function-based verification to validate objects by time and functional behaviour, integrate Coconut, and evaluate systems on actual microcontroller hardware.

## CRediT authorship contribution statement

**Arwa Hameed Alsubhi:** Writing – original draft, Writing – review & editing; **Ornela Dardha:** Writing – review & editing, Supervision, Writing – original draft; **Simon J. Gay:** Writing – review & editing, Supervision, Writing – original draft.

## Acknowledgements

This article is supported by the UK EPSRC New Investigator Award grant EP/X027309/1 “Uni-pi: safety, adaptability and resilience in distributed ecosystems, by construction”. Additionally, this work has partial support from the Royal Embassy of Saudi Arabia Cultural Bureau.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References


[1] J. Wang, W. Tepfenhart, Formal Methods in Computer Science, New York, 2019.
[2] R. DeLine, M. Fähndrich, Enforcing High-level Protocols in Low-level Software, SIGPLAN 36 (5) (2001) 59–69. https://doi.org/10.1145/381694.378811.
[3] A. Devaraju, S. Hoh, M. Hartley, A context gathering framework for context-aware mobile solutions, in: International Conference on Mobile Technology, Applications, and Systems and the 1st International Symposium on Computer Human Interaction in Mobile Technology, Mobility Conference 2007, Singapore, September 10–12, 2007, ACM, 2007, pp. 39–46. https://doi.org/10.1145/1378063.1378070
[4] R. DeLine, M. Fähndrich, Typestates for Objects, in: Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP), Springer, 2004, pp. 465–490. https://doi.org/10.1007/978-3-540-24851-4_21.
[5] M. Coblenz, R. Oei, T. Etzel, P. Koronkevich, M. Baker, Y. Bloem, B.A. Myers, J. Sunshine, J. Aldrich, Obsidian: Typestate and Assets for Safer Blockchain Programming, ACM Trans. Program. Lang. Syst. 42 (3) (2020). https://doi.org/10.1145/3417516.
[6] D. Kouzapas, O. Dardha, R. Perera, S.J. Gay, Typechecking Protocols with Mungo and StMungo, in: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP), PPDP '16, ACM, New York, NY, USA, 2016, pp. 146–159. https://doi.org/10.1145/2967973.2968595.
[7] J. Mota, M. Giunti, A. Ravara, Java Typestate Checker, in: Proceedings of the 23rd IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION), Springer, 2021, pp. 121–133. https://doi.org/10.1007/978-3-030-78142-2_8.
[8] M. Jakobsen, A. Ravier, O. Dardha, Papaya: Global Typestate Analysis of Aliased Objects, in: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP), ACM, 2021. https://doi.org/10.1145/3479394.3479414.
[9] L. Gherardi, D. Brugali, D. Comotti, A Java vs. C++ Performance Evaluation: A 3D Modeling Benchmark, in: Simulation, Modeling, and Programming for Autonomous Robots, Springer, Berlin, Heidelberg, 2012, pp. 161–172.
[10] V.S. Alagar, K. Periyasamy, Extended Finite State Machine, in: Specification of Software Systems, Springer, 2011. https://doi.org/10.1007/978-0-85729-277-3_7.
[11] J.-R. Abrial, Modeling in Event-B: System and Software Design, Cambridge University Press, 2010. https://doi.org/10.1017/CBO9781139195881.
[12] A. Bombarda, S. Bonfanti, A. Gargantini, Developing Medical Devices from Abstract State Machines to Embedded Systems: a Smart Pill Box Case Study, in: Proceedings of the 51st International Conference on Software Technology: Methods and Tools (TOOLS), Springer, 2019, pp. 89–103. https://doi.org/10.1007/978-3-030-29852-4_7.
[13] S. Bonfanti, A. Gargantini, A. Mashkoor, A systematic literature review of the use of formal methods in medical software systems, J. Softw. Evol. Process 30 (5) (2018) e1943. https://doi.org/10.1002/smr.1943.
[14] T.S. Hoang, C. Snook, L. Ladenberger, M. Butler, Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation, in: Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ), Springer, 2016, pp. 360–375. https://doi.org/10.1007/978-3-319-33600-8_31.
[15] Y. Hu, L. Yan, J. Zhan, F. Yan, Z. Yin, F. Peng, Y. Wu, Decision-making System based on Finite State Machine for Low-speed Autonomous Vehicles in the Park, in: IEEE International Conference on Real-time Computing and Robotics (RCAR), 2022, pp. 721–726. https://doi.org/10.1109/RCAR54675.2022.9872208.
[16] S. Hwang, K. Lee, H. Jeon, D. Kum, Autonomous Vehicle Cut-in Algorithm for Lane-merging Scenarios via Policy-based Reinforcement Learning Nested Within Finite-State Machine, IEEE Trans. Intell. Transp. Syst. 23 (10) (2022) 17594–17606. https://doi.org/10.1109/TITS.2022.3153848.
[17] Asmeta, Asmeta Framework, 2022, Formal Methods and SE Laboratory University of Milan and Formal Methods and Software Engineering Lab University of Bergamo, https://asmeta.github.io/. Accessed: 2023-05-30.
[18] IBM, IBM Engineering Rhapsody. https://www.ibm.com/products/engineering-rhapsody. Accessed: 2025-09-04.
[19] A.H. Alsubhi, O. Dardha, Coconut: Typestates for Embedded Systems, in: Coordination Models and Languages, Springer Nature, Cham, Switzerland, 2024, pp. 219–238. https://doi.org/10.1007/978-3-031-62697-5_12.
[20] GNU Compiler Collection (GCC), 2024, Accessed: 2024-12-04.
[21] D. Novillo, GCC—An Architectural Overview, Current Status, and Future Directions, 2010. https://api.semanticscholar.org/CorpusID:49334934.
[22] J. Merrill, Generic and gimple: A new tree represen-tation for entire functions, 2003. https://api.semanticscholar.org/CorpusID:58211542.
[23] Microchip Technology Inc, GCC Compilers for AVR and Arm-Based MCUs and MPUs, 2025, https://www.microchip.com/en-us/tools-resources/develop/microchip-studio/gcc-compilers. Accessed: 2025-09-07.
[24] C.D. Reese, J.V. Eidson, Handbook of OSHA Construction Safety and Health, CRC Press, Boca Raton, 2nd edition, 2006. https://doi.org/10.1201/9781420006230
[25] N.A. Naeem, O. Lhoták, Efficient alias set analysis using SSA form, in: Proceedings of the 2009 International Symposium on Memory Management, ISMM '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 79–88. https://doi.org/10.1145/1542431.1542443
[26] E. White, Making embedded systems, O'Reilly Media, Inc., 2024.
[27] M. Barr, Programming Embedded Systems in C and C++, O'Reilly, 1999.
[28] V. Tolmer, ProtEnc library, 2019, https://github.com/nitnelave/ProtEnc.
[29] M.O. Normalizacyjna, Systems and Software Engineering-Systems and Software Quality requirements and evaluation (SQuaRE), ISO, 2011.
[30] T. Sherman, Quality Attributes for Embedded Systems, in: Advances in Computer and Information Sciences and Engineering, Springer, 2008, pp. 536–539. https://doi.org/10.1007/978-1-4020-8741-7_95.
[31] M.F.S. Oliveira, R.M. Redin, L. Carro, L.d.C. Lamb, F.R. Wagner, Software Quality Metrics and their Impact on Embedded Software, in: Proceedings of the 5th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), IEEE, 2008, pp. 68–77. https://doi.org/10.1109/MOMPES.2008.11.
[32] D.S. Chawla, G. Kaur, Comparative Study of the Software Metrics for the complexity and Maintainability of Software Development, Intern. J. Adv. Comput. Sci. Appl. 4 (9) (2013). https://doi.org/10.14569/IJACSA.2013.040925
[33] S. Chowdhury, R. Holmes, A. Zaidman, R. Kazman, Revisiting the debate: Are code metrics useful for measuring maintenance effort?, Empirical Softw. Engg. 27 (6) (2022). https://doi.org/10.1007/s10664-022-10193-8