CSharPN: A Code-First Framework for Coloured Petri Net Modelling with C# as Inscription Language

Simon Tjell[1]

[1]Independent Researcher, Denmark

Abstract

We present *CSharPN*, a framework for modelling, simulating, and visualising Coloured Petri Nets (CPNs) in which C# serves as the inscription language. Rather than defining a domain-specific modelling language, CSharPN embeds CPN semantics directly into the host language: colour sets are ordinary C# types, arc expressions and guards are lambda expressions, and a model is a plain class inheriting from `CpnModel`. The result is a *code-first* workflow in which models enjoy full IntelliSense support, compile-time type checking, and the entire .NET ecosystem. An interactive Blazor-based visualiser provides animation and step-by-step simulation directly inside VS Code. The framework supports untimed, timed, and hierarchical CPNs. We demonstrate the approach on an OAuth2 authorisation protocol model that exercises all three features, including concurrent users, shared resources, timed operations, session expiry, and hierarchical decomposition into browser and server sub-pages.

Keywords

Coloured Petri Nets, C#, code-first modelling, inscription language, VS Code, simulation, Blazor

1. Introduction

Coloured Petri Nets [1, 2] extend classical Petri nets with data: tokens carry colours (values), and arc inscriptions as well as transition guards are written in a formal inscription language. The expressive power of CPNs makes them suitable for modelling and verifying concurrent systems, communication protocols, and business processes.

Existing CPN tools such as CPN Tools [3] and the CPN IDE [4] are mature environments, but they rely on *Standard ML* as their inscription language and require a dedicated graphical editor. A practitioner who is already working in a modern IDE with a mainstream programming language must either learn an additional formalism and toolset or forego formal modelling altogether.

This paper introduces *CSharPN*[1], a .NET framework that takes a different approach: C# itself is used for both the inscription language and the model definition. A CPN model is an ordinary C# class; colour sets are C# types; arc inscriptions and guards are C# lambda expressions. No separate modelling tool is required. The developer writes, refines, and tests models in the same editor — VS Code — used for all other software development. An interactive Blazor-based[2] visualiser running alongside the model provides an animated view of the net structure and the evolving marking, accessible directly from the editor via VS Code's built-in Simple Browser panel.

The main contributions of this paper are:

- A design for embedding CPN semantics into C# through a fluent API with type-safe places, arc variables, and transition builders.
- Support for untimed, timed, and hierarchical CPNs within a single coherent framework.
- A Blazor-based interactive visualiser with hierarchical page navigation integrated with VS Code.
- A demonstration of the approach on an OAuth2 authorisation code flow that exercises colour sets with domain logic, timed channels, resource sharing, and hierarchical decomposition.

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

simon@simplesystemer.dk (S. Tjell)

[1]CSharPN is open source under the MIT licence and available at https://github.com/simontjell/CSharPN.public.

[2]Blazor is Microsoft's framework for building interactive web user interfaces in C#: https://learn.microsoft.com/aspnet/core/blazor/.

The paper assumes the reader is familiar with the basic concepts of Coloured Petri Nets and with the C# language. Section 2 briefly recalls the CPN concepts that the framework builds on.

2. Background: Coloured Petri Nets

A Coloured Petri Net [2] is a tuple $\mathrm{CPN} = (\Sigma, P, T, A, N, C, G, E, M_0)$ where Σ is the set of *colour sets*, P a finite set of *places*, T a finite set of *transitions*, A a finite set of *arcs*, $N : A \to (P \times T) \cup (T \times P)$ maps each arc to its endpoints, $C : P \to \Sigma$ assigns a colour set to each place, G assigns *guards* to transitions, E assigns *arc expressions* to arcs, and M_0 is the *initial marking*.

Tokens are multisets of values drawn from the colour set of their place. A transition is *enabled* under a binding (an assignment of values to its arc variables) if (i) each input place holds at least the tokens demanded by the corresponding input arc expression under that binding, and (ii) the guard evaluates to `true`. Firing the transition removes the demanded tokens from the input places and deposits the tokens computed by the output arc expressions into the output places.

Timed CPNs [2] extend this model with a global clock and timestamps on tokens. A timed token $v@t$ is only available for consumption when the clock has reached or passed t. The clock advances to the earliest future token time when no transition is immediately enabled.

Hierarchical CPNs [2] introduce substitution transitions and port places. A substitution transition represents an entire sub-page; port places (*In*, *Out*, *I/O*) define the interface between parent and sub-page.

3. The CSharPN Framework

CSharPN is implemented as a .NET class library. Its core components closely mirror the CPN formalism. We introduce them through a single running example: an OAuth2 authorisation code flow. Figures 1–3 show this model as rendered by the CSharPN visualiser. They give the graphical reading of the very net that the listings in this section construct, and the reader is encouraged to consult them alongside the code that follows.

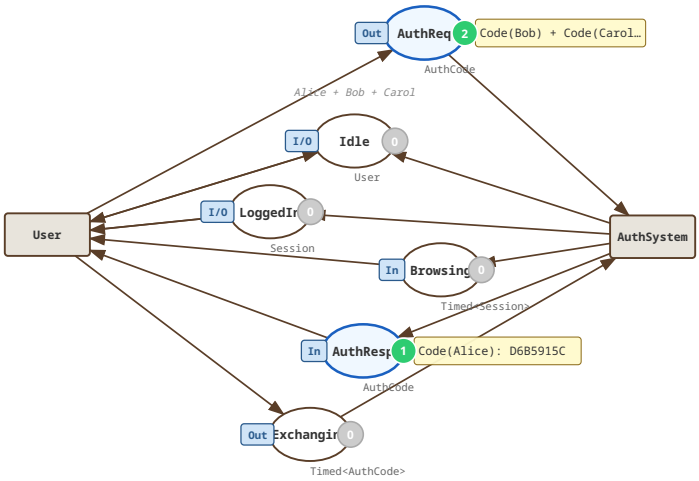

Figure 1: Top-level page view. Shared places connect to substitution transitions for the User and AuthSystem sub-pages.

3.1. Colour Sets, Places, and Multisets

Colour sets are ordinary C# types. Any type satisfying `notnull`, `IEquatable<T>` can be used — primitive types, enumerations, records, or user-defined classes. `Multiset<T>` is an immutable, strongly

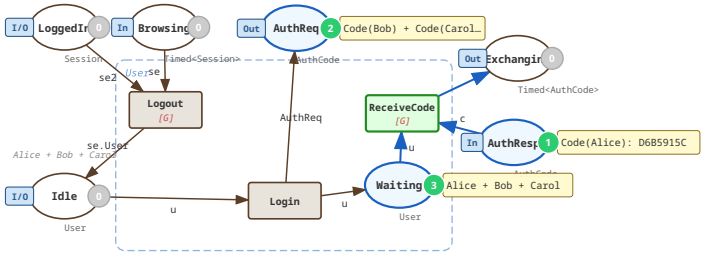

Figure 2: User sub-page (browser-side). Port direction tags (In, Out, I/O) label the shared places. Dashed lines indicate page boundaries. [G] denotes guards. The Login transition is enabled.

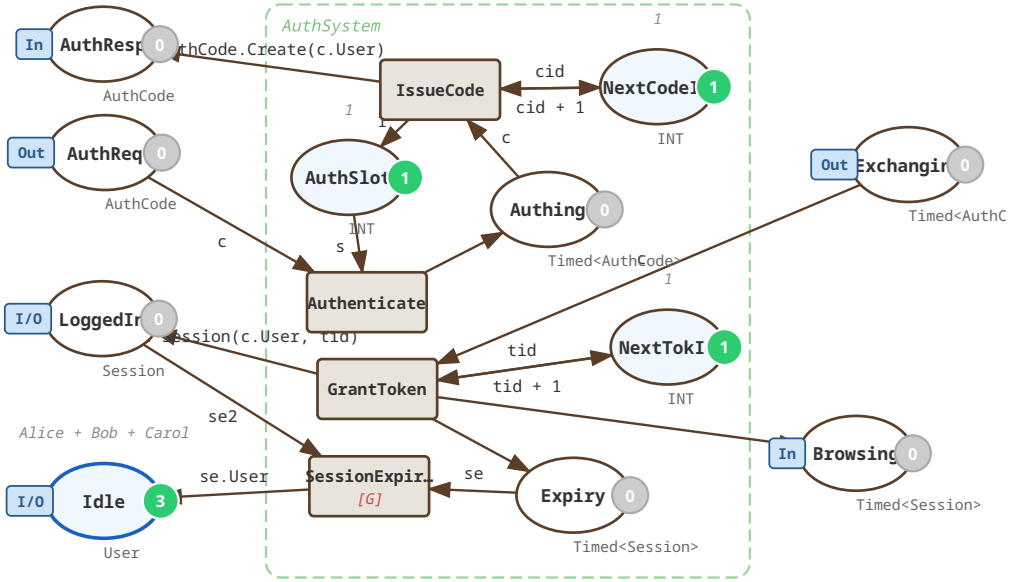

Figure 3: AuthSystem sub-page (server-side). `AuthSlot` limits concurrency; `Expiry` delays session timeout.

typed bag. `Place<T>` holds an initial and a current marking, both of type `Multiset<T>`; the type parameter *is* the colour set.

Listing 1 shows the colour sets from our running example: an OAuth2 authorisation code flow. The `User` record carries domain logic (CSRF state, browse time); `AuthCode` has a validation method and a factory; `Session` pairs a user with a token ID.

Listing 1: Colour sets as C# records with domain logic.

```
public record User(int Id, string Name) {
  public string CsrfState  => $"s{Id}";
  public int    BrowseTime => Id * 5;
}
public record AuthCode(User User, string Code, string State)
{
    public bool ValidCsrf(User u) => State == u.CsrfState;
    public override string ToString() => string.IsNullOrEmpty(Code) ? $"Code({User})" :
        $"Code({User}): {Code}";

    public static AuthCode Create(User u, int seq) {
        var hash = System.Security.Cryptography.SHA256
            .HashData(System.Text.Encoding.UTF8.GetBytes($"{u.Id}:{seq}"));
        return new(u, Convert.ToHexString(hash)[..8], u.CsrfState);
    }
}
public record Session(User User, int TokenId);
```

Places themselves are declared in the model constructor with `AddPlace`: the generic argument fixes the colour set, and an optional `Multiset` supplies the initial marking (Listing 2). `AddTimedPlace` declares a place of timed tokens (Section 3, timed CPNs).

Listing 2: Declaring places, with and without an initial marking.

```
// places of the OAuth2 model, declared in its constructor
var idle    = AddPlace("Idle", Multiset.Of(
                new User(1,"Alice"), new User(2,"Bob"), new User(3,"Carol")));
var authReq = AddPlace<AuthCode>("AuthReq");         // empty initial marking
var exchg   = AddTimedPlace<AuthCode>("Exchanging"); // timed place
```

3.2. Arc Variables and Transition Builder

A binding variable `Var<T>` stores the value assigned during binding enumeration. Transitions are constructed through a fluent `TransitionBuilder`, obtained from `AddTransition(name)`. All model elements are created through such `Add...` methods — `AddPlace`, `AddTransition`, `AddSubPage`: the `Add` prefix signals that the call both constructs the element and registers it with the enclosing model, following the naming convention of .NET collection APIs. A bare `Transition(...)` constructor would build a transition without attaching it to the net.

A transition is connected to places solely through the builder: each `Input(p, ...)` call adds an input arc from place `p` to the transition, and each `Output(p, ...)` call adds an output arc from the transition to place `p`. The set of these calls *is* the transition's arc set; there is no separate arc-declaration step. Output arcs support three styles: variable shorthand (`Output(place, var)`), expression with auto-derived inscription (`Output(place, () => expr)`), and explicit label. Listing 3 shows representative transitions from the OAuth2 model.

Listing 3: Transition builder: login and code exchange.

```
AddTransition("Login")
  .Input(idle, u).Output(waiting, u)
  .Output(authReq, () => AuthCode.Create(u), "AuthReq")
  .Build();
AddTransition("ReceiveCode")
  .Input(authResp, c).Input(waiting, u)
  .Guard(() => c.Val.ValidCsrf(u.Val))
  .TimedOutput(exchg, () => c.Val, Config.ExchangeTime)
  .Build();
```

All arc expressions and guards are C# lambdas that capture `Var<T>` objects by reference. They are validated by the C# compiler at build time, not at simulation time.

Finding enabled bindings. An input arc carries one of two inscriptions: a `Var<T>` (optionally with a multiplicity), which binds the variable to a token drawn from the place; or a multiset expression `() => Multiset<T>`, which may reference `Vars` already bound by earlier input arcs and is checked against the place's marking. CSharPN evaluates input arcs in declaration order: each `Var` arc enumerates the candidate tokens of its place, and each expression arc is then evaluated against the current binding and its result subtracted from the place's available tokens. The guard is checked last. Bindings are thus enumerated by forward evaluation over place contents — never by solving an arc inscription backwards for a variable.

Another limitation is that CSharPN cannot express a *free variable* — one that appears on no input arc and is instead ranged over its colour set, as CPN Tools does when that set is small[3]. The canonical case is a Boolean carried only on an output arc to model the success or failure of a step [6]; in CSharPN such

[3]CPN Tools classifies a colour set as *small* when it is small enough to enumerate — by default, at most 100 elements — and *large* otherwise [5].

a choice must be materialised as tokens in a place and consumed like any other value. In return, binding enumeration is a predictable iteration over place contents that is fully type-checked by the C# compiler.

3.3. Timed CPNs

`TimedCpnModel` maintains a global integer clock. `AddTimedPlace<T>()` creates a place holding `Timed<T>` tokens. Timed arcs are specified via `TimedInput` and `TimedOutput` on the transition builder. The timed simulator automatically advances the clock when no transition is immediately enabled.

In the OAuth2 model, authentication takes 2 time units, token exchange takes 1 time unit, session expiry fires after 12 time units, and each user's browsing time varies (Listing 4).

Listing 4: Timed transitions: auth server and session management.

```
AddTransition("Authenticate")
  .Input(authReq, c).Input(authSlot, s)
  .TimedOutput(authing, () => c.Val, Config.AuthTime)
  .Build();
AddTransition("GrantToken")
  .TimedInput(exchg, c).Input(nextTokId, tid)
  .Output(nextTokId, () => tid.Val + 1)
  .Output(loggedIn, () => new Session(c.Val.User, tid.Val))
  .TimedOutput(expiry, () => new Session(c.Val.User, tid.Val), Config.SessionTTL)
  .TimedOutput(browsing, () => new Session(c.Val.User, tid.Val), () => c.Val.User.BrowseTime)
  .Build();
```

The `GrantToken` transition produces three tokens simultaneously: the session (available immediately), an expiry timer (ready after 12 time units), and a browse timer (ready after the user's individual browse time). `Logout` and `SessionExpire` compete to consume the session — whichever timer fires first wins.

3.4. Hierarchical CPNs

`CpnPage` declares port places using `In()`, `Out()`, and `InOut()` (following CPN Tools conventions). `HierarchicalCpnModel` inherits from `TimedCpnModel`, so sub-pages can use timed arcs with the parent's shared global clock. `AddSubPage()` injects the clock, merges local places and transitions, and registers timed-place inspectors for clock advancement.

The OAuth2 model decomposes into two sub-pages (Listing 5): a `UserPage` modelling browser-side behaviour (login, code receipt, logout) and an `AuthSystemPage` modelling server-side behaviour (authentication, code generation, session management). Shared places — `Idle`, `AuthReq`, `AuthResp`, `Exchanging`, `LoggedIn`, `Browsing` — serve as the communication channels between the two pages.

Listing 5: Hierarchical decomposition: UserPage ports and assembly.

```
class UserPage : CpnPage {
  public UserPage(Place<User> idle, Place<AuthCode> authReq,
    Place<AuthCode> authResp, Place<Timed<AuthCode>> exchg,
    Place<Session> loggedIn, Place<Timed<Session>> browsing)
    : base("User") {
    InOut(idle); Out(authReq); In(authResp);
    Out(exchg);  InOut(loggedIn); In(browsing);
    var waiting = AddPlace<User>("Waiting");
    // ... transitions: Login, ReceiveCode, Logout
  }
}
public class OAuth2 : HierarchicalCpnModel {
  public OAuth2() : base("OAuth2") {
    var idle     = AddPlace("Idle", Multiset.Of(
      new User(1,"Alice"), new User(2,"Bob"), new User(3,"Carol")));
    var authReq  = AddPlace<AuthCode>("AuthReq");
```

```
    var authResp = AddPlace<AuthCode>("AuthResp");
    var exchg    = AddTimedPlace<AuthCode>("Exchanging");
    var loggedIn = AddPlace<Session>("LoggedIn");
    var browsing = AddTimedPlace<Session>("Browsing");
    AddSubPage(new UserPage(idle, authReq, authResp, exchg, loggedIn, browsing), "User");
    AddSubPage(new AuthSystemPage(idle, authReq, authResp, exchg, loggedIn, browsing),
        "AuthSystem");
  }
}
```

4. Visualisation and VS Code Integration

CSharpN includes a Blazor-based visualiser that renders the net structure, overlays the current marking, and supports interactive step-by-step simulation. The visualiser is a web application that can be embedded in VS Code's Simple Browser panel, giving a self-contained modelling environment within the editor (Figure 4).

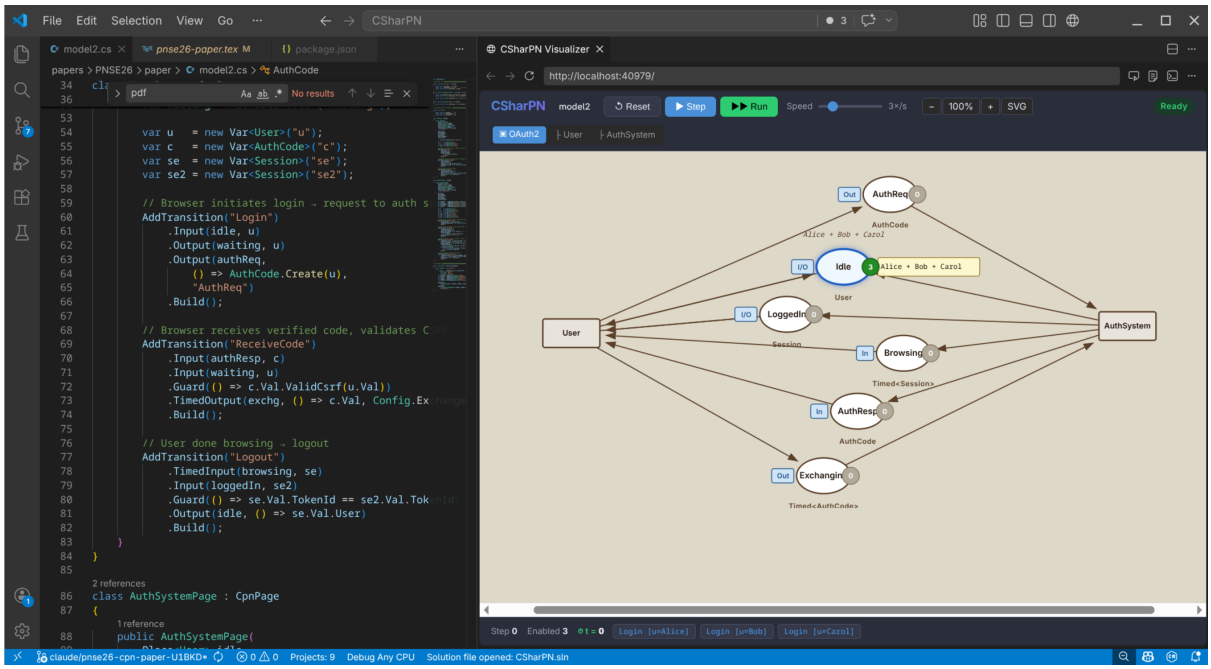

Figure 4: Code-first workflow in VS Code: model source (left) and interactive visualiser (right) side by side. The status bar shows the global clock and enabled bindings; marking detail boxes are toggled via the token badge.

For hierarchical models, the visualiser provides page-level navigation: the top page shows shared places connected to substitution transitions (Figure 1); clicking a substitution transition drills into the sub-page (Figures 2 and 3). A page tree in the toolbar allows direct navigation. Port places carry direction labels (*In*, *Out*, *I/O*) shown as small coloured tags.

The toolbar displays the global clock, step count, and a list of currently enabled bindings. Each binding can be fired individually by clicking it, or the *Step* button fires a random one. Clicking the green token-count badge on a place toggles a detail box showing the current marking values. Double-clicking a place or transition navigates the editor to the corresponding line in the model source code, bridging the visual and textual representations.

Node positions are persisted in a `.layout` file alongside the model source (e.g., `model.cs.layout`). Positions are stored per page and survive model reloads and page switches. A file watcher enables hot-reload: saving the model source triggers recompilation and immediate reload in the visualiser, preserving the current marking where possible through automatic marking migration.

5. Related Work

We discuss how CSharPN relates to existing tools and libraries for Petri net modelling. Most approaches employ dedicated graphical environments or domain-specific inscription languages. CSharPN takes a different approach: it leverages C# as the inscription language, targeting practitioners already embedded in the .NET ecosystem.

CPN Tools / CPN IDE. The dominant tool for CPNs is CPN Tools [3], later succeeded by the CPN IDE [4]. Both use ML as the inscription language and a dedicated graphical editor. CSharPN differs in making a mainstream language (C#) the inscription language and placing the programmer's IDE rather than a dedicated tool at the centre of the workflow.

Snoopy [7]. Snoopy is a multi-formalism editor for various Petri net classes. It provides a graphical-first workflow; CSharPN takes the opposite code-first stance.

SNAKES [8]. SNAKES is the most closely related work: like CSharPN, it is a library that embeds Petri nets in a general-purpose language rather than offering a separate modelling tool, and it uses host-language expressions as annotations. The substantive differences follow from the choice of host language and the level of abstraction. SNAKES annotations are Python expressions, dynamically typed and checked only when the net is executed; CSharPN inscriptions are C# lambdas that the compiler type-checks before any simulation runs, and colour sets are declared static types rather than arbitrary Python values. A SNAKES place carries an optional, dynamically checked type constraint — none by default, so any Python object may serve as a token — whereas every CSharPN place carries a mandatory, statically known colour set, as the Coloured Petri Net formalism requires. Neither choice dominates: SNAKES' dynamic, untyped tokens make it remarkably flexible and lightweight — a real strength for rapid prototyping and for models whose token structure is irregular or still evolving — while CSharPN deliberately trades that flexibility for guarantees the compiler can establish ahead of any simulation. CSharPN further provides timed and hierarchical CPNs with CPN Tools-style port places as first-class constructs, which in SNAKES are available only through plugins or ad-hoc encoding. Finally, SNAKES is a pure library whose drawing support produces static diagrams, while CSharPN couples the library to an interactive visualiser with step-by-step simulation and bidirectional navigation between diagram and source. In short, the two share the embed-in-a-language philosophy but differ in static versus dynamic typing, in committing to the statically-typed CPN formalism (declared colour sets per place) versus SNAKES' dynamically Python-coloured nets, and in the degree of tooling integration.

6. Discussion: Models as Programs

A fundamental consequence of using C# as the inscription language is that a CSharPN model is not merely a *description* of a system — it is a *program* that can be compiled, executed, tested, and deployed using standard software-engineering infrastructure. This blurs the traditional boundary between model and implementation in several ways.

Version control and code review. Because a model is a plain `.cs` file, it lives alongside production code in the same Git repository. Changes to the model appear in pull requests, are reviewed by colleagues, and are tracked in the commit history. Branching and merging work exactly as for any other source file. In contrast, binary or XML-based model formats used by graphical CPN tools resist meaningful textual diffing and review.

Refactoring and IDE support. Renaming a colour set, extracting a sub-page into its own class, or changing a guard expression can be performed with the IDE's built-in refactoring tools. The compiler immediately reports type errors introduced by the change. IntelliSense auto-completes record fields in arc expressions, discovers available methods on colour-set types, and shows parameter documentation for the transition builder API.

Unit testing. A model can be instantiated in an xUnit or NUnit test, stepped through a known sequence of bindings, and asserted against expected markings. This makes it possible to write regression tests that verify model properties after every code change — the same continuous-integration discipline

used for production software. The OAuth2 model, for instance, could be tested to ensure that Carol's session always expires before she logs out.

Reuse of domain libraries. Arc expressions and guards have access to the entire .NET ecosystem. A model of a payment system can call actual currency-conversion libraries; a protocol model can use real cryptographic primitives in its colour sets. This eliminates the mismatch between the formalisms used during modelling and the libraries used in the final implementation.

Composition with production code. A CSharPN model can be packaged as a NuGet library and referenced by a web service, a desktop application, or a test suite. The same `CpnModel` class that was simulated and visualised during design can drive runtime behaviour — firing transitions in response to HTTP requests, database events, or message-queue messages. The formal semantics of the CPN then serve not only as documentation but as the actual state-machine core of the deployed system.

Eliminating the code-generation step. A recurring theme in CPN research is the *generation* of executable code from CPN models. Mortensen [9] demonstrated automatic code generation from Design/CPN models for an industrial access-control system. Kristensen and Westergaard [10] introduced Process-Partitioned CPNs (PP-CPNs) and generated Erlang implementations of routing protocols. Simonsen and Kristensen [11] explicitly addressed the reconciliation problem: keeping a CPN model and its generated implementation synchronised as either evolves.

All these approaches share a fundamental tension. The model is written in one language (ML-based CPN inscriptions), the generated code is in another (Erlang, Java, C), and any change to the model requires re-running the generator — with the risk that manual edits to the generated code are overwritten, or that the generator does not support a construct the modeller needs. The model and the implementation are two artefacts that must be kept in sync, and experience shows [11] that this synchronisation is a significant engineering burden.

CSharPN sidesteps this problem entirely. Because the model *is* the program, there is no translation step, no generated artefact, and no synchronisation to maintain. The colour sets used during simulation are the same types used at runtime; the guard that was verified during analysis is the same predicate that executes in production. The gap between model and implementation is zero by construction.

When the deployment target is not .NET. The preceding argument assumes that the system being built is itself a .NET application into which a `CpnModel` can be linked. When it is not — the production code is written in another language, or runs on a platform that cannot host the .NET runtime — the model-is-program identity no longer holds end to end. CSharPN still yields an executable, testable, version-controlled model, but the model and the deployed system are once again two artefacts, and the code-generation and reconciliation problems discussed above re-emerge, now across a language boundary. The approach is therefore most compelling for teams already working within the .NET ecosystem; we make no claim that it removes the model/implementation gap for arbitrary target platforms. For a heterogeneous deployment the realistic benefit is narrower: an executable reference model against which production code in another language is validated.

The pull of the host language. Embedding the model in a general-purpose language is double-edged. Because arbitrary C# is available inside colour sets, guards, and arc expressions, a modeller can express behaviour as ordinary imperative code instead of as net structure — pushing logic out of the places and transitions that analysis can reason about and into opaque method bodies. CSharPN does not prevent this: keeping control flow in the net is a matter of modelling discipline, much as coding standards rather than the compiler keep a codebase idiomatic. We regard the natural boundary as follows — data modelling belongs in C# types, control flow belongs in the net structure — and tooling that flags transitions with unusually large guard or arc bodies is a plausible way to make that boundary visible. A related cost is the entry barrier: a modeller fluent in CPNs but not in C# faces a steeper start than with an ML-based tool, and the approach is correspondingly most accessible to practitioners for whom C# is already a daily language — the audience the framework explicitly targets.

7. Future Work

Several directions are planned.

Models as deployable software components. Because a CSharPN model is a regular .NET assembly, it can be embedded directly into a production application as a state-machine core. Transitions could be exposed as HTTP endpoints, allowing CPN models to serve as backends for RESTful services.

Testing existing C# libraries. A CPN model can import and call any .NET library from its arc expressions and guards. This opens a novel testing pattern: the model *is* the test harness, exploring interleavings while assertions in guards verify expected behaviour.

State-space analysis. The immutable `CpnState` snapshot together with deterministic multiset hashing forms the foundation for reachability analysis, liveness checking, and deadlock detection.

Empirical evaluation. This paper argues the code-first approach conceptually and demonstrates its feasibility on a single worked example; it does not yet evaluate the approach empirically. Two forms of evaluation are planned: a usability study comparing how practitioners — with and without prior CPN experience — build and modify models in CSharPN versus a graphical CPN tool, and a performance comparison of the simulator against CPN Tools on a common set of benchmark models.

8. Conclusion

We have presented CSharPN, a framework that brings Coloured Petri Net modelling into the mainstream software-development ecosystem by using C# as the inscription language. The code-first approach means that models are ordinary C# programs: they compile with the standard toolchain, are tested by standard unit-test frameworks, and are edited in VS Code with full IntelliSense and refactoring support.

We demonstrated the framework on an OAuth2 authorisation code flow — a protocol with concurrent users, shared resources, timed operations, and session management. The model uses C# records with domain methods as colour sets, timed arcs for channel delays and session expiry, and hierarchical decomposition into browser and server sub-pages. All features — colour sets, timing, hierarchy, port declarations, and interactive visualisation — work together within a single coherent API.

The central insight is that when the inscription language is C#, the full power of a mature programming language — its type system, its libraries, its tooling — is available to the modeller at no extra cost. This lowers the barrier to adoption for practitioners and opens new directions in which CPN models play an active role in the deployed software systems they describe.

Acknowledgments

The author would like to thank the Petri nets community for the foundational work on Coloured Petri Nets that made this framework possible.

Declaration on Generative AI

During the preparation of this work, the author used Claude (Anthropic) in order to: assist with drafting and structuring the paper text, to support implementation of the framework, and to develop the example models. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] K. Jensen, Coloured petri nets, in: W. Brauer, W. Reisig, G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties, Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, pp. 248–299. doi:10.1007/BFb0046842.

[2] K. Jensen, L. M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, Springer, 2009. doi:10.1007/b95112.

[3] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, K. Jensen, CPN Tools for editing, simulating, and analysing coloured Petri Nets, in: Applications and Theory of Petri Nets 2003, volume 2679 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 450–462. doi:10.1007/3-540-44919-1_28.

[4] CPN IDE Project, CPN IDE – an integrated development environment for coloured Petri Nets, 2024. URL: https://cpnide.org/.

[5] CPN Tools Project, Size and complexity of color sets – CPN Tools documentation, 2018. URL: https://cpntools.org/2018/01/09/size-and-complexity-of-color-sets/.

[6] K. Jensen, L. M. Kristensen, L. Wells, Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (2007) 213–254. doi:10.1007/s10009-007-0038-x.

[7] M. Heiner, M. Herajy, F. Liu, C. Rohr, M. Schwarick, Snoopy – a unifying petri net tool, in: S. Haddad, L. Pomello (Eds.), Application and Theory of Petri Nets, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 398–407. doi:10.1007/978-3-642-31131-4_22.

[8] F. Pommereau, SNAKES: A flexible high-level Petri nets library (tool paper), Lecture Notes in Computer Science 9115 (2015) 254–265. doi:10.1007/978-3-319-19488-2_13.

[9] K. H. Mortensen, Automatic code generation method based on coloured Petri net models applied on an access control system, in: Application and Theory of Petri Nets 2000, volume 1825 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 367–386. doi:10.1007/3-540-44988-4_21.

[10] L. M. Kristensen, M. Westergaard, Automatic structure-based code generation from coloured Petri nets: A proof of concept, in: Formal Methods for Industrial Critical Systems (FMICS), volume 6371 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 215–230. doi:10.1007/978-3-642-15898-8_14.

[11] K. I. F. Simonsen, L. M. Kristensen, Towards a CPN-based modelling approach for reconciling verification and implementation of protocol models, in: Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), volume 7706 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 106–125. doi:10.1007/978-3-642-38209-3_7.