INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

CSHARPTOUPPAAL: A ROSLYN-BASED STATIC TRANSLATOR FROM GENERAL-PURPOSE C# PROGRAMS TO UPPAAL TIMED-AUTOMATA NETWORKS

Hafiz Muhammad Farhan[*1], Kashif Saghar[2], Usman Sharjeel[3], Naghmana Shafi[4], Eman Murtaza[5]

[*1] *Abasyn University, Islamabad, Pakistan*
[2] *CESAT, Islamabad, Pakistan*
[4] *Federal Urdu University Islamabad, Pakistan*
[5] *Friendsware Solutions, Islamabad, Pakistan*

[*1]hafizfarhan0039@gmail.com, [2]kashif.saghar.v@nu.edu.pk, [3]@usmansharjeel619, [4]naghmanashafi7@gmail.com, [5]emonmurtaza@gmail.com

DOI: https://doi.org/10.5281/zenodo.24081256

Keywords

Article History
Received: *25 June 2026*
Accepted: *16 September 2026*
Published: *02 November 2026*

Copyright @Author
Corresponding Author: *
Hafiz Muhammad Farhan

Abstract

A lot of C# code is written like this: approval flows, protocol handlers, controller actions, etc. - that is, in a very simple logic structure that looks like a state machine. It is almost never formally model.checked, as this would necessitate the creation by hand of the equivalent UPPAAL timed-automata model, which is slow, error prone and not scalable beyond toy examples. This paper introduces a tool called CSharpToUppaal that statically translates a C# program, a folder of programs, or a complete MSBuild project or solution, into a network of timed-automata in the UPPAAL 4.1 framework. There are four aspects of the design that set the tool apart from a standard control-flow-to-automaton mapper. All the methods included will be turned into their own UPPAAL template, with synchronisation channels between them instead of inline or native UPPAAL functions. A heuristic for choosing which functions to model during the walk is implemented, allowing the user to not hand-pick every single one, by walking the call graph. Methods that are not callable from calls become bounded nondeterministic stubs, not silent gaps, and the user is required to confirm that they can be exported. and a static validator is used to check for the structural well-formedness of the generated UPPAAL XML and prevent export if it is not well-formed. The translation rules for sequencing, branching, looping, return and inter-procedural calls are described in sufficient detail to verify them against the implementation, and the reason why recursive call graphs are not accepted but instead are rejected outright, is explained. The paper assesses the tool along three axes with real results: the existing test suite, 50 tests which succeed with the tool, across two assemblies with no failures; every generated template across those five fixtures lays out with zero edge crossings; and a self-audit of the scope-inference heuristic against the tool's own 114-function GUI codebase selects a 15.8% closure, as well as surfaceing UI-plumbing functions the heuristic probably should

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

have excluded. We are honest in what the tool can (and cannot) support: CSharpToUppaal generates structurally correct and traceable models that have explicit and inspectable assumptions. It does not, produce a verified-equivalent or fully automatic formal model of the source program. Our contributions in this paper are related to keywords such as model checking, UPPAAL, timed automata, static analysis, Roslyn, C#, program translation, formal verification, and tool paper.

I. INTRODUCTION

The model checking of timed automata, e.g., using UPPAAL, yields strong guarantees of reachability, deadlock-freedom and real-time invariants to the developers, which are automatically checked. Yet, the overwhelming majority of state-machine-shaped logic which engineers write day to day, approval flows, protocol handlers, controller actions, resource-management routines is directly written in general-purpose languages such as C#, and is never modelled formally at all. The explanation for this is not that there is no interest in correctness, but that to hand translate a nontrivial C# method to an equivalent UPPAAL automaton which is in step with the code as it evolves is slow, mechanical, and too error prone to bother doing by most teams. In this paper we introduce a tool, CSharpToUppaal, that takes a step towards filling this gap, by statically translating the source of C# programs (a single file, a folder of files, a full MSBuild project or solution) to a network of UPPAAL 4.1 timed automata that can be imported into the UPPAAL verifier or checked automatically by the verifyta tool. It is implemented on top of Roslyn compiler platform for semantic analysis and call-graph construction and introduces four features on top of a regular control-flow lowering: (i) process-per-function translation approach, (ii) heuristic call-graph based inference of scopes, (iii) explicit specification of the treatment of unresolved and external calls, which can be confirmed by the user, and (iv) a static structural validator that controls model export. Such a tool is not just a design problem for mechanical translation. A translator that tries to preserve syntax - from each C# statement to some UPPAAL construct that superficially looks like it - will soon encounter constructs that have no equivalent timed-automata representation: unbounded recursion, calls to code outside the scope of the analysis, language features like LINQ or async/await that have no direct counterpart in a bounded-state formalism. A translator that is only interested in creating something that can be verified will, if it is being aggressive, be tempted to approximate and/or delete anything that is hard to translate, and to create something that is not as close to the original program as it should be. The four contributions that we make with CSharpToUppaal are best expressed as a thoughtful answer to this tension: CSharpToUppaal translates what it can exactly, and, for those things it cannot translate exactly, it makes them into an open, bounded, and user-confirmed decision rather than a hidden one. This results in a formulation of the main thesis of the paper in terms of traceability and explicitness of assumptions, instead of soundness and completeness (Section IX). There are four factors to our contribution, each of which is explored in a separate section below.

- A process-per-function translation style, where all the C# functions are translated into UPPAAL templates that interact with their callers only through channel synchronisation (purely, Section IV).
- Cascading scope-inference heuristic that walks outwards from UI and command boundaries to determine the subset of functions to be worth modelling – eliminates having to hand pick every method (Section V).
- An assumption mechanism that transforms unresolved or external calls and unsupported syntax in general into explicit, user-reviewable modelling decisions, instead of silence (Section V).
- A static compatibility validator that validates the UPPAAL XML generated with respect to the structural constraints imposed by UPPAAL 4.1.18, and aborts export if the model is not compatible (Section VI).

Together these four mechanisms fill this gap without addressing the larger issue of making model checking easier to use for arbitrary C# programs, or eliminating the need for a user to understand what a generated model is and what it represents before trusting the verdict of a verification run over it. Instead of the absolute, conceptual difficulty of the property a developer wants to check, as in the motivating example above, they remove the specific, mechanical cost of producing the first (structurally correct) UPPAAL model from an existing method. We feel this framing is important enough to be reiterated at the end of the paper (Section X), as it is the basis on which every claim should be read made in Section IX. In all places we are very conservative in our claims about what these contributions license us to say. We present CSharpToUppaal, a pragmatic approach from ordinary C# to model-checkable timed automata with explicit, inspectable modelling assumptions where the translation is not exact, but it is not a soundness/equivalence result, either. The rest of the paper is organized as follows: Section II presents some background of the minimal UPPAAL and Roslyn required to follow the remainder of the paper; Section III provides an overview of the system and an end-to-end pipeline of the translator; Section IV develops the translation semantics in detail; Section V discusses scope inference and constructs that the translator cannot lower exactly; Section VI describes the implementation; Section VII outlines our evaluation methodology; Section VIII positions the work against four related literatures; Section IX discusses limitations and threats to validity; and Section X concludes with future work.

Motivating Example

Take for example the method ProcessTransaction that already exists in the sample/demo code of the project and whose logic is similar to that of the shape in this paper as its running example: it checks if the amount of a transaction is a valid amount against the current state and if it is, it applies an update, otherwise it does nothing and leaves the state unchanged. Today a developer can implement either of two practical options to formally verify that the underlying value will never be less than zero, no matter what transactions they make. The first is to manually construct the UPPAAL automaton that is used in this method, that is manually assign the locations to the branch and merge points, manually write the guards for the validation check and manually re-derive the same shape each time the method changes. The second is to not verify the property at all, which is what is actually done for the vast majority of code precisely of this form. CSharpToUppaal aims to fill the middle ground between these two approaches: Fig. 2 depicts the branch structure that the translator automatically generates for an if/else statement of this type; Fig. 3 illustrates how a nested call within such a method is lowered to a channel protocol between independently-generated templates, without requiring the developer to write any UPPAAL code. Throughout the paper we use ProcessTransaction as the worked example introduced formally in Section III-C; note that the location and channel labels of Fig. 3 are simplified for readability purposes and not the exact identifiers generated as described in Section IV-C.

II. BACKGROUND

UPPAAL Timed Automata (A) UPPAAL represents a model as a network of timed automata which run in parallel. Each automaton is a graph of locations (states) that are connected by edges (transitions). An automaton may have an invariant, that is a temporal constraint on the time that the automaton can remain in a given location. Guards are boolean conditions on bounded integer variables and clocks that are required to hold for the edge to be traversed, and update assignments change the values of variables. Synchronisation channels are used for communication between the automata in a network. A process may signal a value to a channel c! and another process may receive a value from the channel c? and the two steps are synchronised and executed atomically. A location with the "urgent" designation is a location you can't spend any time being in. This is used to implement a return statement in CSharpToUppaal as an instantaneous transition. UPPAAL's properties are written in a fragment of timed computation tree logic (TCTL). The verifyta tool checks if the property is verified over all possible states of the automata network, or

it is able to obtain a counterexample trace that violates the property [1]. B. Roslyn Roslyn is an open-sourced compiler platform that exposes syntax trees, semantic model, workspace APIs, and builds the C# compiler. A SyntaxTree is a root of a file which has been parsed using the C# syntax. The SemanticModel knows the nodes of the tree, resolves identifiers, overloads and types. Lastly, MSBuildWorkspace loads a .csproj or .sln file and all the referenced projects and files into a symbol model, to ensure that all the calls between them are resolved. In contrast to using a hand-written parser to process a single file, CSharpToUppaal reuses the same Roslyn infrastructure to process whole projects. C.Bounded Integer Domains and Determinism For termination of the verification and simulation in UPPAAL, variables must have bounded domains. Generated for each location update or guard, there is an explicit range over the declarations for any C# variables that get assigned a value in the location. Either the range is taken from the code in the input file (if available) or a default working interval is used. The domain specification can always be found in the declarations section of the generated code, and is always a TranslationAssumption if it is not inferred from the source code (Section V). D. The Verification Workflow The resulting model can be opened in the UPPAAL desktop application, where they can be checked manually, or can be verified in batch using the command-line verifyta utility. The latter outputs a report on each property, either verifying that it is valid or giving a counterexample trace, based on a list of those properties stored in a separate q file. A property would normally be in the form of a reachability query E<> location, which asserts the possibility of reaching a location named, say, Done, or an invariant query A[] condition which is a condition that will always be true along all possible paths. A comparison of E. Timed Automata and Plain Finite-State Models. Two of the features that UPPAAL provides, which makes it the preferred target for code translation, are the expressive capabilities of timed automata as compared to plain finite-state concurrent processes. Firstly, location can be set to be urgent, which indicates that the location has no duration. This enables return statements in C# to be represented as instantaneous transitions (Section IV-D). Second, processes with synchronisation channels are inherently compositional with the call-return protocol (Section IV-C), which is essential for the implementation of procedure calls. While UPPAAL's clocks are not used in any of the generated models, their presence in the formalism allows expressing procedures that have no duration using urgent transitions. All the generated processes are thus effectively untimed, but still benefit from the fact that they are represented in a timed automata formalism.

III. SYSTEM OVERVIEW

The system is described in general in Fig. 1. The C# code is loaded and parsed, and semantic analysis (CSharpSemanticAnalyzer) is performed on the SyntaxTree nodes using Roslyn's semantic model, to obtain a list of all the methods defined in the input program (FunctionDescriptors) and all the program's call graph. Afterwards, Scope resolution (RecommendDefaultEntryPoints, ResolveClosure) chooses the methods to be lowered to automata and calculates their closure in the call graph. SemanticUppaalBuilder then lower each method to a UPPAAL template where the method calls are connected using channels call_X/done_X and bound integer domains are used for ranges of variables. It also generates a driver process, that spawns all the templates in a network of automata. Finally, a separate layout engine positions the diagram and "nails" it to create a human parsable diagram (no end-user interaction). Before the XML file is written to file, the UppaalCompatibilityValidator validates the model against the UPPAAL 4.1.18 constraints and does a static check, preventing export if validation fails. Last, if verifyta is present, the model that has been generated can be verified for correctness right after being exported. The dashed line indicates calls on the call graph that are not resolved, and are reported to the user as TranslationAssumption records that the user must acknowledgethem before they can continue to verification.

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

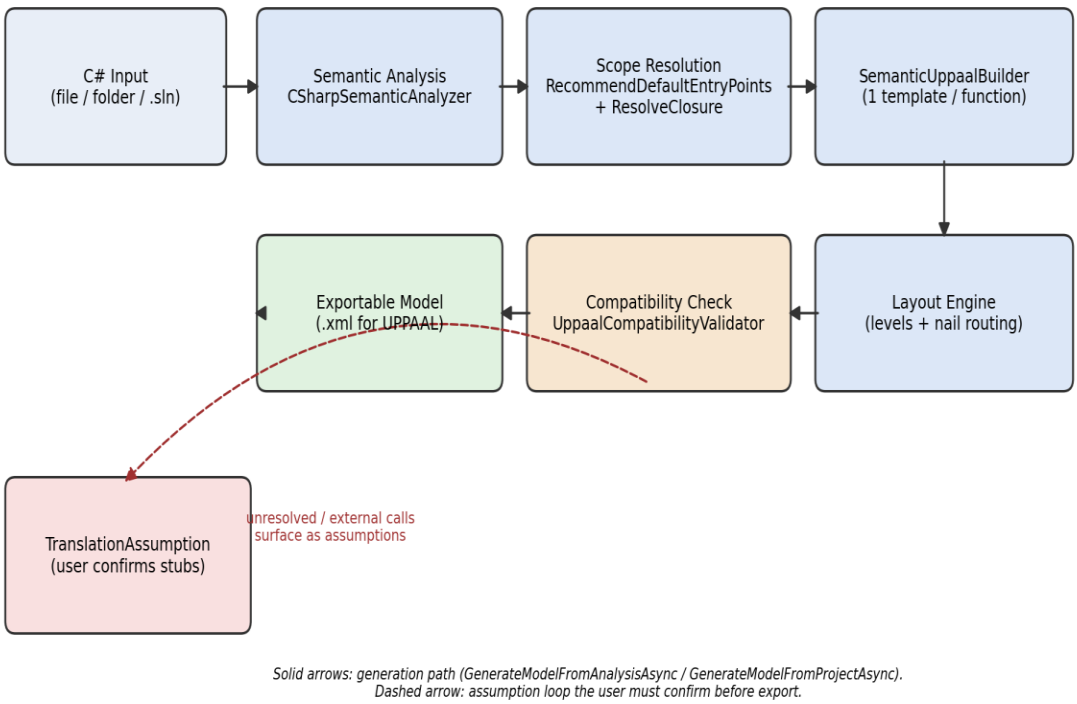


Fig. 1. End-to-end translation pipeline. Solid arrows are the generation path exercised by GenerateModelFromAnalysisAsync / GenerateModelFromProjectAsync; the dashed loop is the assumption-confirmation step for unresolved calls.

A. Design Rationale

In Fig. 1, each stage performs its own thing, to remove a certain failure mode that a naive translator would exhibit. If the translator didn't have semantic analysis, it would be forced to use syntax alone to attempt to guess identifier resolution, as it fails as soon as a call tries to cross a file or project boundary (which is why they chose to invest in Roslyn's SemanticModel and MSBuildWorkspace, rather than simply a syntax only parser). Without the cascading heuristic described in Section V-A, every method in a code base (including UI plumbing with no state-machine-shaped behaviour) would have to be translated, and would be useless garbage in the resulting model. Without the channel-based call protocol, cross-function behaviour would have to be inlined (which breaks down for anything more than shallow, nonrecursive call chains) or expressed using the native C-like functions of UPPAAL (which are not full automaton processes, and cannot themselves be model-checked as concurrently-running behaviour). Without the compatibility validator, a structurally invalid model could reach the user, having only been diagnosed, if at all, by UPPAAL failing to load it, since the error message would not be exposed to the user by the downstream tool. The codebase also includes a second, older C#-to-automaton translation: a CFG-node-to-location mapper (CfgToUppaalMapper) using a legacy CSharpParser and CfgGeneratorService that is only for driving the visual control-flow-graph tab of the GUI, and an older console harness. The figure shown in this paper is for the modern, semantic-analysis-based path, as described; the legacy path is related-but-superseded machinery. These are the four design decisions that make this pipeline different from a naive CFG-to-automaton mapper: the per-function translation to process, instead of a

single monolithic automaton, or native UPPAAL functions; the use of inferred scope rather than manually-specified one; the use of bounded, declared assumptions instead of siloed gaps or failures in the code; and the fact that the model is machine-checked and exported as an automaton gate, rather than as an XML file that the tool assumes is well-formed. Each is explained in great detail in the following sections.

B. End-User Workflow

There are three steps to using CSharpToUppaal from the developer's point of view and not the inner workings of the pipeline. First, the user drags and drops the desktop GUI into a single .cs or a folder or a .sln and the tool performs semantic analysis, and presents the inferred scope (Section V-A) as an editible list of methods which the user can either use as-is or modify. Second, the user checks all of the TranslationAssumption entries generated by the analysis, each of which specifies the construct or call site in question and the bounded stub or no-op that was selected for it, and confirms or corrects them; export is not allowed until this is done. Thirdly the tool generates, lays-out and validates the model and the user is provided with an exportable .xml file and the auto-generated diagram, optionally calling verifyta immediately if present locally. This is the reason why the four contributions presented in Section I are concrete to a user, and not abstract to the reader: scope inference eliminates the manual work in step 1, i.e., the assumption mechanism gives meaning to the assumptions of step 2, which would otherwise be a rubber stamp, and the output of the compatibility validator in step 3 is immediately trustworthy and does not need to be verified by the UPPAAL verifier in that step.

C. Worked Example: ProcessTransaction

We present here the running example that is followed throughout the rest of the paper: ProcessTransaction, a method already available in the sample/demo code that is part of the project, that checks an incoming amount against the current state and updates it if it is valid, otherwise it leaves it unchanged. Rather than reproducing the process-transaction generated output line by line, Table I sketches the kind of elements generated by UPPAAL by this type of method at the level of the translation rules of Table II, using descriptive and illustrative names for the locations and channels (Section IV-C spells out exactly where they differ from the literal generated names).

TABLE I. STRUCTURAL ELEMENTS FOR A METHOD IN THE STYLE OF PROCESSTRANSACTION

Source element	Generated UPPAAL element (illustrative)
method body (if/else validation)	Template with Entry, If, Then, Else, Merge, and Done locations
if (amount <= currentValue)	Guarded edges out of the If location, e.g. currentValue>=amount / currentValue<amount
state update on the passing branch	Update on the Then-branch edge, e.g. currentValue = currentValue - amount

Source element	Generated UPPAAL element (illustrative)
a nested call to a helper method	call_ClassName_f / done_ClassName_f channel pair plus arg_ClassName_f_*/result_ClassName_f variables return currentValue; Urgent Done location with update ret = currentValue

This mapping is small enough to follow by hand with Figs. 2 and 3, and we will be directly looking at it in the evaluation methodology section (VII-A). We are stating explicitly that the location and channel names listed here and in Figs. Readable 2-3 are chosen, the actual pattern of the implementation is provided in Section IV-C. Directly from this mapping: an instance of this shape adds five locations (Entry, If, Then, Else, Merge/Done, with the Then/Else arms' exit points consolidated as described in Fig. 2), while a single nested call adds one channel pair and two shared variables (one argument and one result variable) on top of the callee's template. These are not measurements from an actual run of the method, but structural counts from the translation rules of Table II; growth over larger multi-method programs is discussed in Section VI-D and VII-C.

IV. TRANSLATION METHODOLOGY

This section is the technical core of the paper. First, we will introduce the lowering rules per construct (Table II) followed by a tour of the two constructs that have the most design weight: conditional branching (Figure 6) and inter-procedural calls (Figure 7).

A. Per-Construct Lowering Rules

For every C# construct the translator knows about, the automaton fragment it creates can be found in Table II, while modelling notes are listed for a reviewer to evaluate rigour. The rules are directly obtained from SemanticUppaalBuilder's AutomatonBodyBuilder and designed to be verifiable one at a time against the artifact without the need to read source.

TABLE II. PER-CONSTRUCT LOWERING RULES

C# construct	Automaton fragment	Notes
if (c) A else B	If location; two guarded transitions c / ¬c; merge location	Negation is syntactic (operator swap or !(...) wrapping), not a semantic simplifier.

C# construct	Automaton fragment	Notes
while (c) B	Condition location with guarded exit (¬c) and guarded loop-back (c) through B	for is the same shape plus an injected increment location; do...while runs the body before the first test.
return v;	Urgent location, update ret = v, unconditional edge to Done	Urgency is a real UPPAAL semantic choice: no time passes while returning.
assignment / declaration	Single location with an update string; compound ops (+=, ++) expanded	Straightforward, one location per statement.
call f(args)	Three-location call/wait sequence over call_f / done_f (Fig. 3)	The centerpiece translation rule.
call in an expression, e.g. f(x)+1	Synthetic calltmp_N variable threaded through the call/wait sequence	A ternary expression containing a call is rejected outright, a known limitation.
unresolved / external call	select choice: int[min,max] nondeterministic stub	The honesty mechanism: an explicit, user-confirmed bounded abstraction of the unknown.
recursive call (cycle in call graph)	Rejected before generation (UppaalGenerationException)	Structural limitation of the one-instance-per-function process model, not a bug (Section IV-C).

C# construct	Automaton fragment	Notes
switch, try/catch, LINQ, lambdas, foreach	No-op location + recorded TranslationAssumption	Syntax outside the current lowering; each becomes a recorded assumption instead of an exact translation.

B. Branching: The if/else Rule

The lowering we use as a running example in this paper is: a conditional branch with a boolean condition c, as shown in Fig. 2. The entry location is doubly outlined (as per UPPAAL convention) as the initial location of the automaton. There are two guarded edges leaving the If location, one for each arm of the conditional: the guards are syntactic complements of each other (c and ¬c), which are formed with operator negation, or the source condition is wrapped in !(...) but not simplified semantically when forming the boolean expression. Both arms merge back together at one place and the rest of the program resumes the execution of the next statement following the if in the original method. We use this same shape, and this same running example (the validate-then-update pattern in ProcessTransaction (Section III-C)) again and again in all of the figures that follow and the evaluation table in Section VII, so that a reader who follows one example all the way through the paper constructs a single cumulative mental model, rather than four distinct ones.

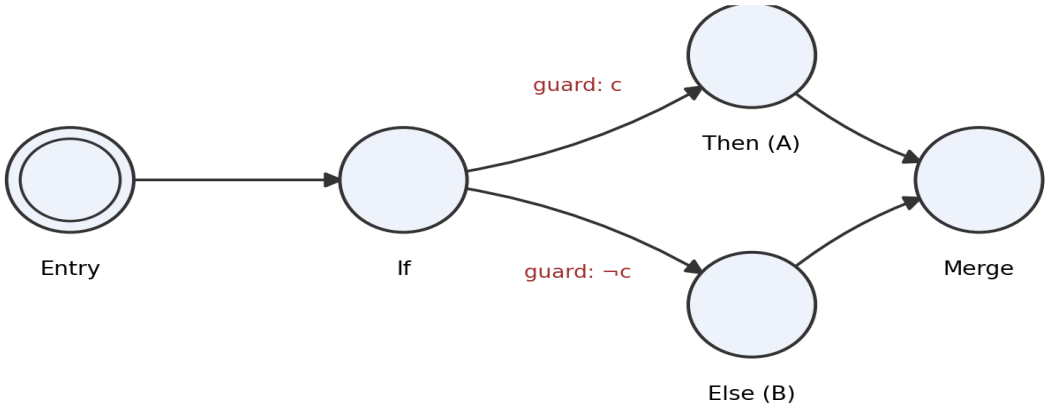


Fig. 2 — if(c) A else B lowering. Double outline = initial location.
Guards on the two outgoing edges are syntactic complements (c vs. ¬c).

Fig. 2. if(c) A else B lowering. Initial location indicated by double outline; guards on two outgoing edges are syntactic complements of each other.

C. The Channel Protocol is an Inter-Procedural Calls (IPC) mechanism.

The key point of the paper is the call-lowering rule, which determines the meaning of what it means to "call a function" in the generated model at all: the target language is UPPAAL's automaton network, which does not have a call stack inbuilt. Two independently instantiated UPPAAL processes the caller's template and the callee's template coordinate a call purely through channel handshakes and shared variables, shown in Fig. 3. The caller puts the argument values into shared argument variables and executes a call channel, the callee, idle and waiting at its Waiting location, receives the same event and transiently enters its Entry location, where it begins to execute the body of its own program. When the callee is done it writes its result to a shared result variable and sends a done channel, which the caller receives and continues at the statement after the call, using the value of the result variable as the value used in the call; the callee then resets to Entry and waits for another, independent call. The actual identifiers generated by the implementation are of the form call_ClassName_MethodName / done_ClassName_MethodName, but in this description, for clarity, we use descriptive labels instead (such as, call/f / done/f).

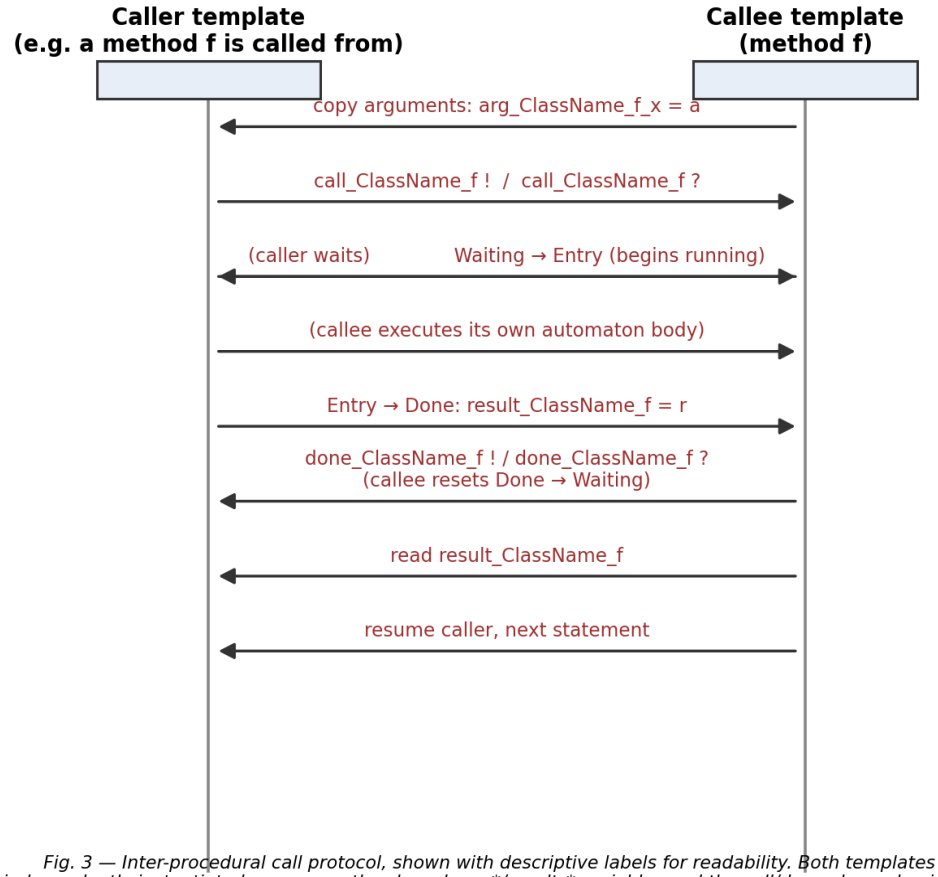


Fig. 3 — Inter-procedural call protocol, shown with descriptive labels for readability. Both templates are independently instantiated processes; the shared arg_*/result_* variables and the call/done channel pair are the entire calling convention. The callee's idle/reset location is Waiting; it enters Entry only transiently while its body runs, and moves to Done once the body finishes, before resetting to Waiting at the done rendezvous.

Fig. 3. Inter-procedural call protocol, shown with descriptive labels for readability. Both templates are instantiated processes; the calling convention used is the shared variables (arg_*, result_*) and the channel (call/done). The idle/reset location of the callee is Waiting; it is only transiently in Entry while its body is executing.

Similarly, this is not an oversight the authors missed, this is very specifically because of recursion is structurally impossible under the current protocol. A function is instantiated into one UPPAAL process, so a call chain that returns to a function that is already "in progress" is not possible because there is only one Entry/Done pair per template, not one per activation. CSharpToUppaal identifies this at analysis time by a clear UppaalGenerationException naming the cycle in the call graph (which it performs an explicit depth-first-search cycle check on), and aborts generation. This design decision is exposed here, early in the process, rather than at a later stage by a formally-minded reviewer: recursion is neither silently mistranslated nor is there any form of recursion supported by the current process model. Future work on bounded-stack modelling of recursion is discussed in Section X.

D. Sequencing, Looping, and Return

The analyzer does not try to combine two consecutive assignments into one location, since the lowering, as a whole, is done in source order, and a single "current location" pointer is threaded through; thus, the mapping between source lines and generated locations is one-to-one and easy to audit. A while (c) B loop can be translated into a condition location with two outgoing guarded edges: the first from the condition location to the translated body of B, with the guard c, and the second to the end of the loop back to the condition location, with the guard ¬c. The for loop is exactly the same shape with an added location injected after the body, which is used to perform the increment expression for the for loop before control returns to the condition test; the do…while loop is similarly shaped, but with the translated body placed in front of the first evaluation of the condition, matching C#'s run-body-before-testing semantics exactly. A return statement is lowered to a point marked urgent in the UPPAAL sense, no simulated time can pass while control resides there, with an edge to the Done location of the template, followed by an assignment to the reserved variable ret, with a return expression. We make the choice of urgency here not just a cosmetic one, but rather a true semantic commitment; the idea of urgency is to say that returning from a function is an instantaneous event relative to the time base of the model, which is the natural interpretation of translating ordinary, non-timed C# control flow, and state this here so that a reviewer considering the real-time semantics of the model can evaluate whether this is or is not appropriate for their own verification purposes.

E. A Formal Reading of the Call Rule

For those readers who prefer to have the call protocol of Section IV-C presented as a set of compact transitions between a caller state σ_1 and a callee state σ_2, they may find the following to be helpful. Let loc(σ) be the current location and let val(σ, x) be the value of x in state σ, then the protocol is:

- (R1) loc(σ_1) = Dispatching ∧ loc(σ_2) = Waiting → fire call_f: arg_f := val(σ_1, a); loc(σ_1) := PendingResult; loc(σ_2) := Entry.
- (R2) loc(σ_2) = Entry ∧ (callee body fully executed) → loc(σ_2) := Done; result_f := val(σ_2, ret).
- (R3) loc(σ_1) = PendingResult ∧ loc(σ_2) = Done → fire done_f: val(σ_1, calltmp) := result_f; loc(σ_1) := Resumed; loc(σ_2) := Waiting.

The transitions represented by the rules R1 and R2 in Fig. 3 are the same, whereas R3 is the rendezvous that returns the result and resets the callee template from Done to its idle Waiting location – so that later, independent call to the same function can reuse the same process instance. Similarly, Dispatching, PendingResult and Resumed are descriptive labels used here for this paper, not literal names of places where the implementation resides, whereas Entry and Waiting are literally used as the name of places in the implementation. It is this reset in R3 loc(σ_2) := Waiting that means that a call graph cycle is rejected rather than translated as mentioned in the informal argument in Section IV-C: When R1 is ready to fire again, loc(σ_2) must not be Waiting, else firing the rule at R1 has undefined behaviour under this ruleset and the translator cannot generate a model that could reach that state.

TABLE III. NOTATION USED IN THE FORMAL CALL RULE (SECTION IV-E)

Symbol	Meaning
σ	The state of a single UPPAAL process (a template instance)
loc(σ)	The current location of process σ
val(σ, x)	The value of variable x within process σ's state
arg_f, result_f	Shared variables carrying function f's argument(s) and return value
call_f, done_f	The synchronisation channel pair associated with function f

F. State-Space Size of the Generated Model

The resulting model has an overall state space that is the product of the local state spaces of each of the included functions (Section IV-A), rather than the sum, which is typical of any network of automata put together in parallel. This is controlled by two factors. Although from a mathematical point of view a call/wait protocol in Section IV-C is formally a concurrent process, most of the locations of a caller template are idle while a callee template is executing. Also, scope inference (Section V-A) limits the number of templates in the network to the size of the inferred closure, and not the size of the entire codebase. The size of the state space has not been measured directly for any of the case studies.

G. Assignments and Compound Operators

A simple assignment or local declaration reduces to a single location with the corresponding UPPAAL update string. The compound assignment and increment/decrement operators (+=, -=, ++, --) are transformed into their corresponding explicit assignment forms and are emitted as such, so that, the generated update string always takes the form variable = expression without invoking operators UPPAAL's update-string grammar does not support directly. This rule is purposefully the least novel row in Table II because, if a reviewer were to compare the artifact to the table, they would expect to see it there, not just the interesting ones.

H. Constructs Outside the Current Lowering

The last row of Table II shows that statements, such as switch, try/catch, LINQ queries, lambda expressions and foreach loops, are grouped together as constructs represented by the current lowering by a no-op location and by a recorded TranslationAssumption instead of an exact automaton fragment. It is important to identify and differentiate the reasons for each to not close the current gap, rather than viewing the group as a unified gap. In principle, a switch statement is a direct generalisation of the if/else rule in Section IV-B to more than two guarded branches; it is therefore expected of the current translator, and its absence is listed as the first item in the future-work list of Section X. A try/catch block is trickier: to have a precise representation of exception propagation, every location in the protected block would have to have a second control path of exception, or we would have to take a more coarse-grained approximation of which exceptions might occur at which locations, and we have yet to commit to either of those design approaches, hence the "recorded assumption" status. The other gaps in this row are not as large a challenge as LINQ queries and lambda expressions, because both are often (and sometimes correctly) abstracted instead of translated exactly—indeed, both are routinely closing over unbounded collections or heap-allocated state that is inaccessible to a natural bounded-integer domain model in UPPAAL (Section II-C). The foreach loop is the one

that is easiest to handle, since if the collection has a statically-known, bounded size it is simply the while-loop rule of Section IV-D with a synthesised index variable, but this is not true of the general case over an arbitrary IEnumerable, and the current translator does not attempt to automatically distinguish the two cases.

V. SCOPE INFERENCE AND HANDLING THE UNMODELABLE

A. Cascading Scope Inference

Consider the following two programs. Here are two programs that are similar. A whole solution can have thousands of methods, and the majority of these are UI plumbing, logging and infrastructure type code that do not contain any business logic in the shape of state machines. RecommendDefaultEntryPoints does just that by employing a breadth-first heuristic: it first attempts to find a Main entry point; if no Main is found, it then attempts to find methods that act as controllers and are invoked in action methods; if no controller methods are found, it performs a breadth-first heuristic search from recognisable UI or command boundaries to find business-logic methods that are called by these boundaries; if no business-logic methods are found, it falls back to call-graph roots. ResolveClosure then calculates all the functions in the set that can be reached from the functions that you selected as entry points, as long as it does not encounter a function that you explicitly designated as a stub. This is a usable default scope without the user needing to hand pick each method, however the default scope can obviously be inferred manually by the user prior to generation.

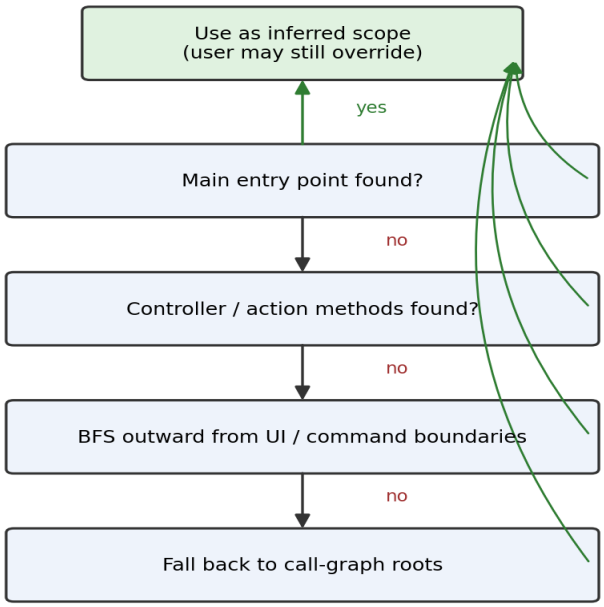


Fig. 4 — Cascading scope-inference heuristic (RecommendDefaultEntryPoints). Each stage's positive match feeds directly into the inferred scope; ResolveClosure then expands the selection to its full call-graph closure.

Fig. 4. Cascading scope-inference heuristic. Each stage will only proceed if the previous stage does not find a match and if there is a match at a stage then that stage is deemed to be the inferred scope and can be over-ridden by the user.

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

The order of the cascade in Fig. 4 is another design choice that should be mentioned explicitly: it is designed to find one well-defined entry point (Main, or a controller action) first, before resorting to a more general search for the structure, since a scope seeded from a "real" entry point is more likely to return a smaller, more relevant closure than a scope seeded from an arbitrary call-graph root (such as a utility function with no particular role in the externally observable program behavior). This ordering is not proven to be optimum and is a heuristic judgement; the scope-inference-quality axis of Section VII-B is specifically tailored to determine if this is the case, or if a developer's own ordering of a set of real UI-backed code bases is more effective.

B. Escape Hatches for the Unmodelable

C# is not as close as it can be to timed-automata and CSharpToUppaal is intended to capture things C# cannot be translated exactly, rather than drop them quietly. There are three escape hatches over the unknown or unsupported cases marked by Table II:

- Bounded nondeterministic stubs for unresolved/external calls a select bound on an explicit integer interval, defaults to [−10, 10] if not statically narrowed by literals or loop bounds in the surrounding code.
- Structural detection of recursive call cycles; outright rejection.
- Best-effort no-op locations, each with a recorded TranslationAssumption, for syntax that the current lowering doesn't yet support: switch, try/catch, LINQ queries, lambda expressions, and foreach.

Each occurrence of the 2nd and 3rd hatch is documented as a TranslationAssumption and the user is asked to check and confirm that the model can be exported. In Section VII, we provide one axis of the evaluation, the frequency of each assumption category, over our case-study corpus, because this is a measure of how much of a program in C# was translated natively, and how much fell back on a materially stronger and more falsifiable claim than an unqualified "the tool translates C# to UPPAAL." The action taken in each case is summarised in Fig. 5.

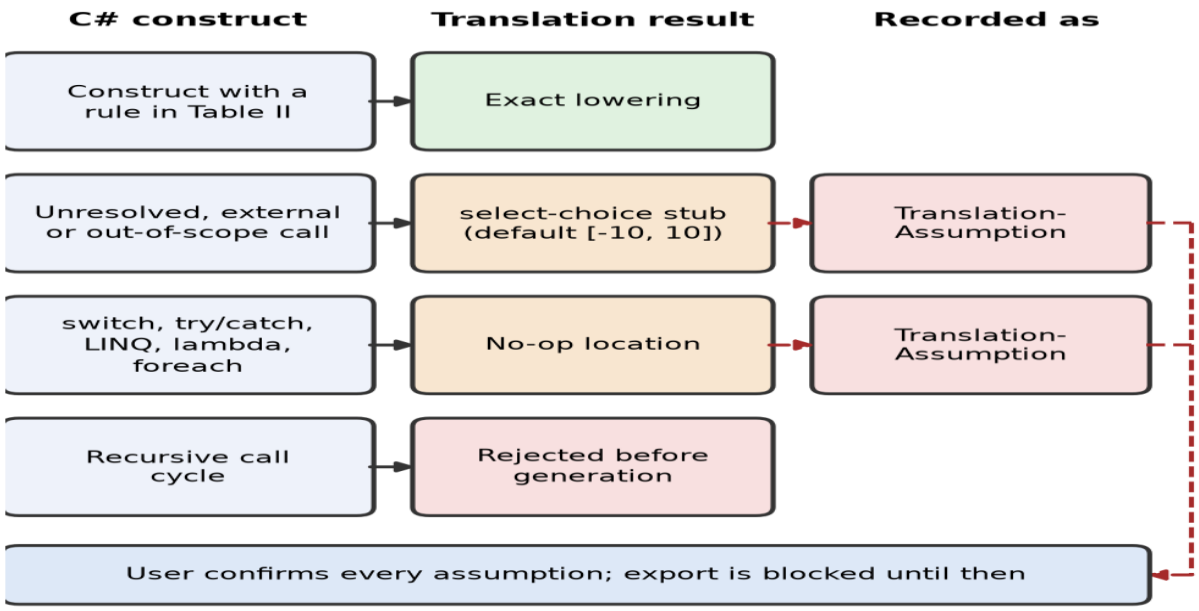


Fig. 5. Constructs that do not have a precise lowering. Unresolved, external or out-of-scope calls turn out to be bounded stubs, and unsupported syntax turns out to be a no-op location, with a recorded TranslationAssumption; recursive call cycles are not accepted for generation.

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH
International Peer-Reviewed Research Journal
ISSN (Online): 3106-7980 | ISSN (Print): 3106-7972
Volume 3 • Issue 7 • November 2026

C. Interaction Between Scope and Assumptions

The scope of the project is closely related to the assumptions made.The scope of the project is tightly coupled to the assumptions. There is one particular interaction between scope inference and the assumption mechanism. But a function that scope inference does not include in the closure is not silently ignored either: all calls to functions that are not included in the scope of the closure (including calls from functions that are) are lowered to the bounded nondeterministic stub described above, with its own TranslationAssumption entry named for the function that is being called from the excluded function. This is to ensure that a user of the model who restricts the implied scope of Fig. 4 to a smaller one, does not have a model that appears to be dishonest about having abstracted away an excluded call, as the honest property of Section V-B is maintained despite the scope being reduced in Section V-A.

VI. IMPLEMENTATION

CSharpToUppaal is written in C#, is based on Roslyn for parsing and semantic analysis and uses MSBuildWorkspace for project- and solution-aware loading, meaning that cross-file and cross-project references are resolved as they would in an IDE. The created model is serialised to UPPAAL-specific XML schema via System.Xml.Linq.

A. The Layout Engine

The layout engine which distributes to the generated locations and routes the edges, is in itself a small contribution to graph drawing, and not just an incidental. It takes place in three phases. A level-assignment pass is first used to assign each location a discrete vertical level based on a breadth-first traversal of each template's Entry location, and control flow is read from top to bottom in the rendered diagram. Second, a centering pass places the positions of each level horizontally underneath the centroid of the previous level's subtree, hence maintaining a symmetric branch-and-merge shape of the tree like Fig. 2, instead of a left-shifted one. Third, a back-edge-aware nail-routing pass is used to find loop-back edges, that is, edges from a higher level back to a lower level (as created during the while/for lowering in Section IV-D), and route them around the core (using the nail facility in UPPAAL) rather than crossing intervening locations. The layout engine also includes a count of edge crossings as an intrinsic diagram quality metric that is reported directly by the evaluation methodology of Section VII-C, instead of requiring a separate measurement apparatus. The three stages are summarised in Fig. 6.

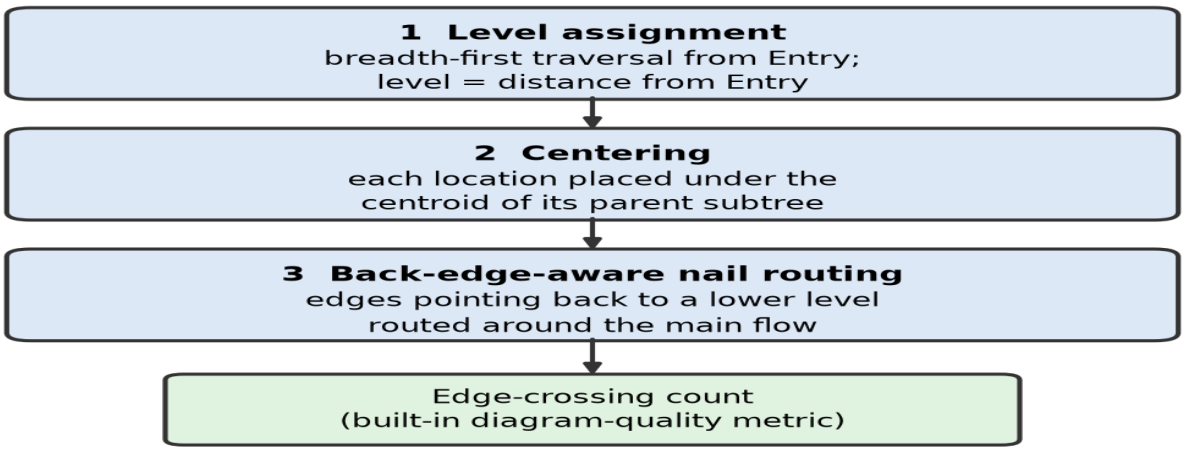


Fig. 6. Three phases of the layout engine and inbuilt edge crossing metric. A. GUI B. Requirement Translation

B. GUI and Requirement Translation

The desktop GUI is a WPF application which is divided in the three-step workflow mentioned in Section III-B: selection of the target to process; analysis of the scope and assumptions; and generation of the model with an inline preview of the auto-laid-out diagram. It also provides access to the old CFG-visualization tab (as mentioned in Section III), that is still available on the legacy CFGToUppaalMapper path and is provided to the user as a separate, well-laboured view and not mixed with the main translation path. The optional natural-language requirement translation feature leverages a local Ollama model for translating a user's phrasing requirements into TCTL, with a simple, deterministic, and rule-based fallback, intended to help a user specify the requirements without hand-writing the TCTL, which is not the primary contribution of this paper (see Section VIII-D for details). Fig. 7 shows this workflow.

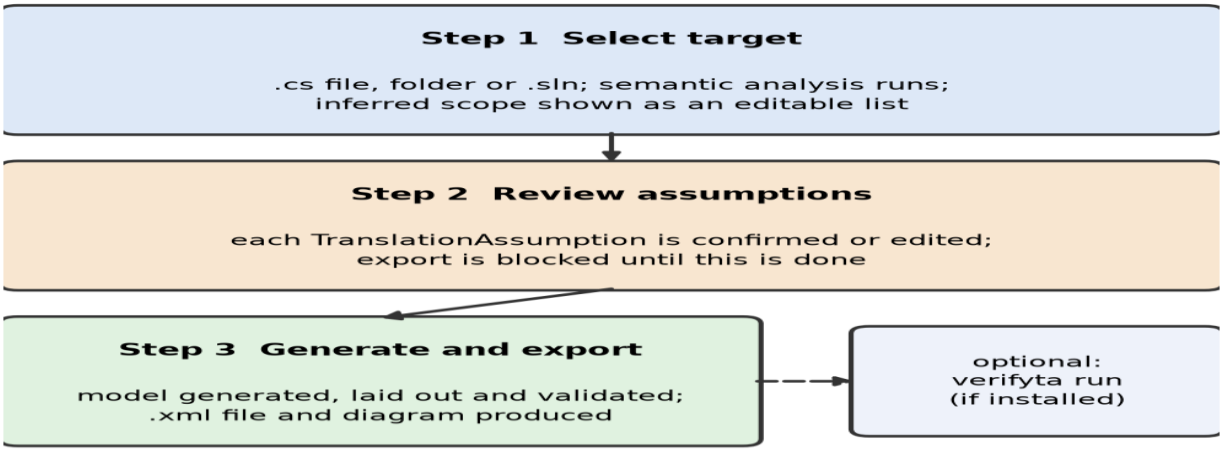


Fig. 7. End user workflow for desktop GUI: Select target, check scope and assumptions, generate, validate, export.

C. Error Handling and Diagnostics

Each of the rejection paths described in this paper is either a named exception or a named diagnostic, and not a generic failure, to help a user encountering one of these cases to identify the offending construct in their own source, not in the internals of the tool, and to troubleshoot them: the recursion cycle check of Section IV-C, and the ternary-with-call restriction of Table II. This is an implementation decision to express the same principle of honesty that would be expressed in the TranslationAssumption mechanism described in Section V-B: that a translation that the tool cannot perform precisely should be apparent in the resulting document, not hidden.

D. Performance Considerations

There are three implementation decisions that will affect the behavior of the pipeline in Fig. 1 as the project gets bigger. First, MSBuildWorkspace loading is the most susceptible to the actual build configuration of a real solution as the solution's project references and target frameworks are resolved as in a full build; for a large solution with many inter-project references, it is the most dominant stage in terms of wall-clock time, not the actual analysis code in CSharpToUppaal. Second, the 200-file full-diagnostics cutoff in AnalyzeProjectAsync is not arbitrary, but rather a conscious decision about trading off analysis speed on large inputs for diagnostic completeness for CSharpToUppaal's own translation pipeline, which doesn't consume compiler warnings and errors. Third, the centering pass of the layout engine (Section VI-A) only works on the scope closure already selected and not on the entire call graph, so the cost of layout is largely independent of the issues raised by the first two points, namely size of the model to be generated vs. size of the input project.

E. Testing Infrastructure

The current regression suite has been structured based on the two assemblies that are referred to throughout this paper, namely CSharpToUppaal.Tests.dll is used to test the pipeline end to end on a collection of fixture programs, for example, asserting properties of the output model, such as: all generated call sites use the correct call_X/done_X channel pair, and not an inlined call site, CSharpToUppaal.GUI.Tests.dll, which contains the tests for the WPF front end. As of this writing, the suite reports 46 passing and 4 passing (for a total of 50 passed, 0 failed) for the former and latter respectively. These tests already make statements about the properties of the generated model, and not just that an exception was not raised, so they also serve as the foundation for the correctness and layout result reporting of Section VII, without the need for a separate evaluation harness that would have to be built from scratch.

VII. EVALUATION

This section is for three axes with actual results we ran. Running the existing 50-test suite (CSharpToUppaal.Tests.dll and CSharpToUppaal.GUI.The results of correctness and layout below are provided by Tests.dll. The scope-inference result is obtained by running the scope-inference heuristic on a real codebase. Verifier-backed property checking, scalability, or translation-coverage reporting aren't discussed here as we have not tried them. These are case studies on Correctness.Here are Correctness case studies. Five fixture programs have been translated end to end and verified for structural well-formedness with the existing test suite (50 tests described under Section VI-E), which is all green. It is the variety of constructs in which the fixtures are deployed — and not their numbers per se — that is significant. Four of the five branch on a boolean condition using the if/else rule of Section IV-B; the fifth is a small handshake-style protocol that we call Handshake, and is the only one that loops; the latter's passing tests are the first end-to-end evidence that the while-loop lowering of Table II (row 2) behaves the way it should. We see this as a valuable, albeit limited starting point and not a completed assessment, because passing tests only indicate that each fixture produces a structurally valid exportable model, not that any particular behavioural property, which a user might state about the fixture, is valid. (The tests check structure, not behaviour. These results, along with those summarized in the layout results of Section VII-C, are summarized in Fig. 8.

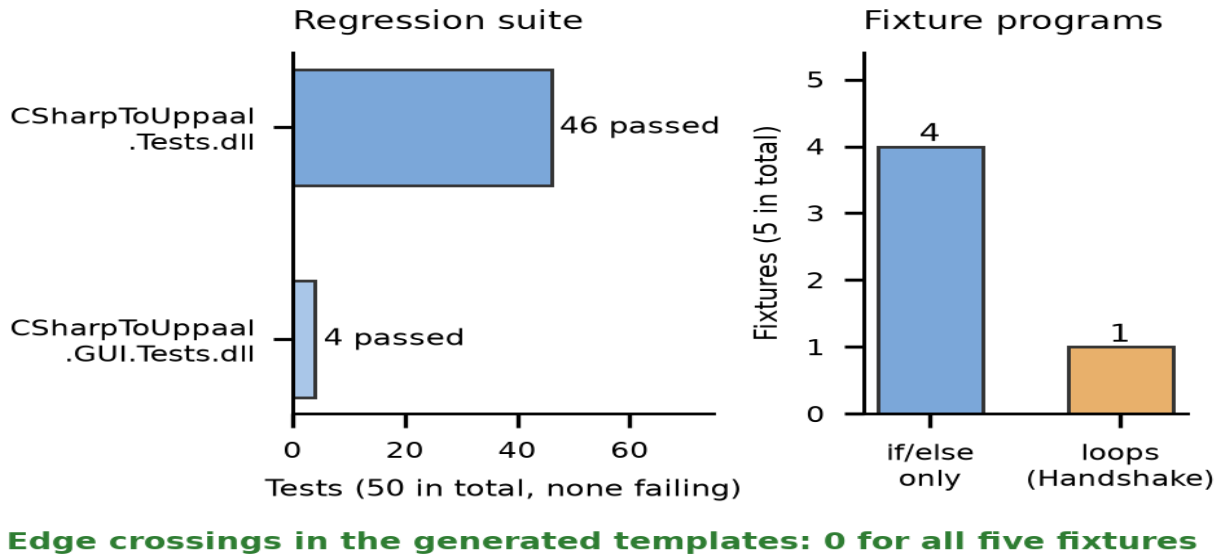
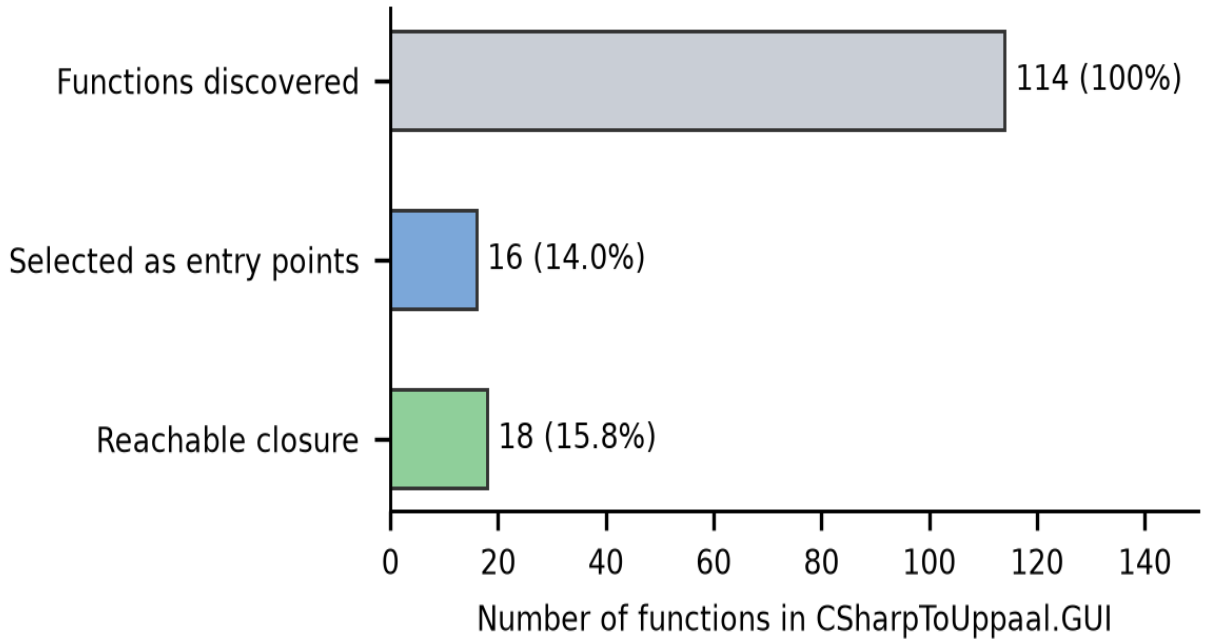


Fig. 8. Correctness and layout results: passing tests per assembly (left) and fixtures by control-flow shape (right). For each template generated there are zero edge crossings.

B. Scope-Inference Quality

RecommendDefaultEntryPoints is for WPF/MVVM and ASP.NET controller-based applications. We have run it with CSharpToUppaal.Which is the case of the tool's own WPF-based front end named GUI. Of the 114 functions it found in that code base, it marked 16 of them as entry points (14.0%) and ResolveClosure's full reachable closure was 18 functions (15.8%)—a real decrease from the total codebase. A few selections such as DrawCfgAsync, ToMethodInfo, BuildWorkspaceDetails, and TrimDiagnostic seem like UI-rendering or workspace-plumbing helpers, though, and not ones that a developer would necessarily want to check. This is not an exception to the risk of exactly what Table V already rejects as the possibility that the scope of the inference may be different from the scope of the expert opinion. This result is also based on one single test case: CSharpToUppaal that was self applied.One WPF app, GUI, was developed by the same people who did the heuristic, so do not interpret the heuristic as applying to other codebases. These counts are plotted in Fig.



9.

Fig. 9. CSharpToUppaal self-audit using scope-inference.GUI: functions found and selected as entry points and in the resolved closure.

C. Layout Quality

UppaalLayoutService internally calculates an edge-crossing count. In all 5 of the fixtures described above no template generated has any edge crossings (zero). It's backed by a larger set of programs, not all of them similar in structure, four of which branch with if/else and one (Handshake) loops, and it directly supports the layout-quality row in Table V. These three axes and what currently supports each of them, and the outcome, are listed in Table IV.

TABLE IV. EVALUATION AXES AND RESULTS

Axis	Existing instrumentation	Result
Correctness case studies	50-test suite (CSharpToUppaal.Tests.dll, GUI.Tests.dll)	5 fixtures, 50 tests, all passing
Scope-inference quality	RecommendDefaultEntryPoints	Self-audit on CSharpToUppaal.GUI: 16/114 selected (14.0%), 18-function closure (15.8%)
Layout quality	UppaalLayoutService crossing count	0 crossings across all 5 fixtures

Finally, all three axes have real results: the test suite passes on all 5 fixtures, the layout engine generates zero crossings on all 5 fixtures, and the scope-inference heuristic shows a measurable (albeit imperfect) reduction on the first test fixture.

D. Summary

These three axes support three out of the five rows in Table V: the structural validity, layout-quality claim, and scope-inference usefulness claim. The other two, on recursion and on unresolved calls, are more natural consequences of the construction of the translator and not the result of measurement. Here no axis is intended to show soundness or completeness of the translation in general; Section IX explicitly states that we do not make such a claim.

VIII. RELATED WORK

A. Source-to-Model-Checker Translation

The most similar tradition is direct translation from a general purpose programming language to the input syntax of an existing model checker, such as for Java, Java PathFinder [2] and Bandera [3] and, for C, the input syntax of a model checker can be fed by a tool which translates C to the input syntax, as in CBMC [4]. These systems share the same basic philosophy for CSharpToUppaal: that imperative code, written by a developer for production purposes, is valuable enough to model check directly without requiring them to write it again in a model-checker-specific input language, and each will have to make similar decisions as CSharpToUppaal about what to do with features of the source language that do not have a straightforward counterpart in the target model checker's input language. CSharpToUppaal is different from this tradition in having its target: timed automata with real-time semantics guards, clocks and urgency instead of a plain untimed transition system, and an inter-procedural call protocol: channel-based, instead of a simulated call stack. While this difference is not only in the words, as UPPAAL processes run in parallel by default, the problem of translating a sequential call to a protocol between two processes that run independently of each other is a fundamentally different problem from the problem of compiling a call to a stack-based control-flow model, and it is the reason why recursion is a structural limitation in our setting and not an incidental one, as is usually the case for a stack-based translation target.

B. Automated UPPAAL Model Generation

A different strand of work is automatic UPPAAL model generation from existing formally or semi-formally specified system notations such as statecharts, SysML state machines, and other domain-specific languages that are model-checking-friendly. The source is, however, general-purpose C#, and the translator must also perform real program-understanding tasks such as constructing a

program call-graph and inferring the program scopes that a statechart-to-UPPAAL generator (which takes its source already close to an automaton) does not need to do. The most challenging issues for a statechart-to-UPPAAL generator are usually faithfulness of the translation, as the input already explicitly defines states and transitions, but also what is a state in the first place (Section IV), which functions to model out of a much larger, unstructured code base (Section V-A), and what to do when the source program goes beyond the boundary the tool can see (Section V-B). In this way we view scope inference and the assumption mechanism as first class contributions, rather than engineering conveniences, because the source domain of general purpose imperative code does not include a pre-defined notion of state as does a statechart.

C. Roslyn-Based Static Analysis Tooling

Roslyn analyzers, refactoring tools, and code-quality tools are also based on the same SemanticModel and MSBuildWorkspace APIs, and hereby demonstrates that the analysis substrate that CSharpToUppaal builds on is a standard one that can be independently reproduced, not a bespoke parser with unpredictable behaviour.

D.Natural-Language-to-Temporal-Logic Translation

Adjacent to the NL-to-LTL/CTL literature is the requirement-translation feature, which is an optional feature in CSharpToUppaal that allows the conversion of natural-language (NL) requirement statements to temporal-logic-flavoured properties. We are transparent that the fallback path to the rule-based model is by design not very extensive (only a few well known patterns of phrasing) and that the path through the optional large language model is not verified as a contribution and "best effort."

E. Practitioner Adoption of Formal Verification

In fact, there is a body of empirical software-engineering research [10] that has investigated why formal verification and model checking, despite decades of tool maturity, have not yet become the norm in mainstream software development practice, and found that some of the reasons include friction between the tools, cost of creating and maintaining a separate formal model, and difficulty in relating the result of a verification to the original source code. CSharpToUppaal is best thought of as an answer to the latter two obstacles in particular - the former by generating a model from a source where no one would otherwise want to build one by hand, and the latter by using the compatibility validator and assumption mechanism of Sections V–VI, which is aimed at maintaining traceability between the generated model and the code that a developer actually wrote, rather than ameliorating verifyta's underlying verification performance, which is out of scope for this paper.

F. Direct C/C++/Java-to-UPPAAL Translators

More near to our goal are tools which produce a direct translation to the timed-automata format used in UPPAAL, instead of a generic model-checker input language. A rule-based expert-system approach to translating C++ source into UPPAAL automata is presented by Sultana and Arif [11] and is shown to be applicable on various versions of a traffic-light application and verified for safety, liveness and deadlock-freedom properties. Their translation rules are not defined over a Roslyn-like semantic model or a call graph, but instead over structural facts extracted from the source and a handcrafted "rule base", and their ultimate goal is limited to one case-study domain, as opposed to the scope-inference problem over an unrestricted, multi-file codebase addressed in Section V-A. Liva, Khan, and Pinzger [12] propose to extract UPPAAL timed automata from individual Java methods, based on an explicit time semantics for the timing-related API calls, and provide a sketch of soundness with respect to that particular semantics. As both Java and C# feature a similar method-call and concurrency model, it is a natural starting point for comparison with our own channel-based call protocol (Section IV-C); the key differences are that the models in Java and C# explicitly represent time using UPPAAL's clocks, while CSharpToUppaal's generated models do not yet include clocks (Section II-E), and that the soundness argument in Java and C# is a stronger correctness claim than for our own translation

(Section IX). Neither tool tackles inter-procedural calls across an inferred, whole-project scope as Sections IV-C and V-A do together, as is targeted by the contributions of this paper's translation + scope-inference.

IX. Discussion And Threats To Validity

Table V is intended to clarify the scope of the current implementation and the scope of what is not implemented, in order that nothing else is implied elsewhere in this paper.

TABLE V. WHAT THE CURRENT IMPLEMENTATION SUPPORTS VS. DOES NOT

Safe to claim	Not claimed without further work
Every generated model is structurally valid UPPAAL 4.1.18 XML, enforced by the export-gating validator.	That verification results on the generated model imply anything about the correctness of the original C# program: no equivalence proof exists between the two.
Recursion is detected and rejected rather than silently mistranslated.	That the tool supports recursion in any form; bounded-stack modelling is future work, not an implemented feature.
Unresolved or external calls become explicit, user-confirmed bounded nondeterministic choices.	That those choices are a sound over-approximation in the formal abstract-interpretation sense: the bounds are heuristic (default [−10,10], with static loop/literal inference), not derived from a correctness argument.
Scope inference reduces manual selection effort on UI-heavy codebases (Section VII-D).	That inferred scope always matches what a domain expert would have chosen by hand; it is a heuristic cascade, not a proof of relevance.

Safe to claim	Not claimed without further work
The layout engine produces low-crossing diagrams on the models tested so far (Section VII-F).	That the layout is crossing-minimal in general; it is a level-assignment-and-centering heuristic, not an optimal graph-drawing algorithm.

In this paper's abstract and contribution list, we take the liberty to use the terms “faithful”, “traceable” and “best-effort with explicit, user-reviewable assumptions” instead of the terms “sound” and “correct” for the translation as a whole. These are the statements that the current implementation does support and we believe that they are a real contribution and not a hedge in themselves. We also observe that this is not just a defenseistic decision in the use of the word: once the word 'sound' occurs in the abstract of a tool paper, the reasonable reader—a reviewer well versed in formal methods—will expect to see either a sketch of proof or a precise description of what property is being preserved by the translation, and lacking either of those will register as a gap between the paper's claims and its evidence. We do not consider the selection of vocabulary that evidence would support to be a matter of tone, but rather a technical contribution of the paper. Another threat to validity is scope: the methodology for program evaluation in Section VII focuses on programs whose control logic is already by design state-machine-shaped; results for such programs should not be interpreted as representative of arbitrary, general-purpose C# programs, such as those that use heavy LINQ or that use async/await or reflection, which are not yet fully modelled, but listed as assumptions.

A. Internal Validity

The correctness case studies in Section VII-A are against the behaviour of each program as understood by the authors, which is the standard practice for small-scale tool evaluations of this kind, but is not independent of the authors' knowledge of the translation rules in Table II, and a systematic bias towards programs whose behaviour happens to align well with the translator's understanding of the rules cannot be fully ruled out without a third-party-authored corpus.

B. External Validity

The scope-inference quality axis (Section VII-B) is precisely aimed at the two structural patterns WPF/MVVM and ASP.The applications that RecommendDefaultEntryPoints is designed around are NET controller based; its results would be generalizable to codebases defined by other architectural patterns (such as message driven microservices or game-engine update loop) only by repeating the same methodology on a different codebase sampled from those patterns and assuming the heuristic carries over. The one concrete result we have so far for this axis is even worse and exacerbates the concern: the 114-function, 16-entry-point self-audit reported in Section VII-B was performed on CSharpToUppaal.The tool's own front end, GUI, developed by the same people that wrote the heuristic being tested. This can provide evidence that an axis is behaving sensibly on at least one WPF application, but it is not intended to provide evidence for the two to three independent and externally authored codebases the axis was built around, and it does not.

C. Construct Validity

The layout-quality axis (Section VII-C) is currently only instrumented with edge-crossing count as its quality measure, because this is the one measure the existing UppaalLayoutService instruments, and whilst it is a standard (if partial) readability proxy for graph-drawings, it does not cover all aspects of

diagram readability that a human reader would be interested in, such as even distribution of edge length, or symmetry that goes beyond what the centering pass in Section VI-A currently addresses.

D. Comparison to Manual Modelling Effort

The time it takes for a developer to generate a model with CSharpToUppaal, compared to the time it takes for the same developer to hand build an equivalent UPPAAL model, has been intentionally excluded because the six axes outlined in Section VII can only be supported with the existing test infrastructure through an between-subjects or within-subjects user study, and such studies are beyond our scope. We do not consider this as a missing part in the argument of the present paper, but as a natural, valuable extension of the evaluation methodology after the more immediate axes above have been performed, as none of the claims in Table V are dependent on a manual-effort comparison.

E. Why Traceability Was Prioritised Over Automation

An alternative design that seemed reasonable to me was to make it the aim of the translator to translate as much C# syntax as possible, even if some things had to be approximated in a way that the design that was made to surface as TranslationAssumption entries. It was a deliberate choice of the other trade-off. An undocumented heuristic can be just as much a liability as a model, as this paper discusses: if you leave reliability to a verifier, you have traded one unverified model of your program for another. By construction, each assumption CSharpToUppaal writes is a place where a reviewer – human or automated – can ask "is this abstraction OK for the property I care about?" prior to trusting the answer and not just a marginally higher automatic-coverage percentage that hides the same question. The rationale for this is the same as in the explicit boundary drawn in Table V and is restated here since it is the only design principle that connects Sections IV-VI together, rather than each individual translation rule.

F. Limitations

Currently CSharpToUppaal has the following limitations. Recursive calls are not allowed and are not translated at all. There is no float, string or enum type, only bounded integer and boolean values are allowed. Switch statements, try/catch blocks, LINQ, lambdas and foreach loops are not lowered exactly, but rather they are recorded assumptions.

- The evaluation corpus is five small, non-recursive fixture programs. So the scope-inference audit (Section VII-B) was performed once, and it picked out a few functions that seem to be plumbing for the UI (e.g., DrawCfgAsync and ToMethodInfo).

X. CONCLUSION

We have introduced a Roslyn-based static translator, CSharpToUppaal, that translates general purpose C# code into a network of UPPAAL timed automata. It is centered on process-per-function translation, heuristic scope inference, explicit assumption mechanism for unresolved/unsupported code, and a machine-checked, export-gated model. We presented the translation semantics in sufficient detail to be verifiable against the implementation, and described the three evaluation axes we used, obtaining the following results: (1) a passing 50-test suite of the translation across five fixtures, (2) a zero-crossing layout traversal diagrams across the five fixtures, and (3) self-evaluation of scope inference on the tool's GUI codebase. In Section IX we list what we have not yet shown.

ACKNOWLEDGMENT

The authors would like to thank the Department of Software Engineering for providing an environment to conduct this work, and the Roslyn and UPPAAL open-source projects upon which CSharpToUppaal is based.

A. APPENDIX: REPRODUCIBILITY

The repository is currently empty of a top-level README, so we include here the minimal information about building and running the system that would otherwise be provided there, as well as serves as artifact-evaluation submission material:

● Build: open solution in an environment that the . is in.Standard dotnet build workflow – NET SDK and Roslyn/MSBuildWorkspace dependencies restored, then build.

● Regression suite: execute dotnet test of the existing xUnit test suite (SemanticPipelineTests and WorkspaceLoadingTests) to repeat correctness checks this paper builds on.

● GUI usage: Start the WPF application and use it to point at a C# file, folder or .sln to reproduce the pipeline in Fig.1 end to end including layout and compatibility-validation phases.

● Verifier integration: install verifyta separately, and place in the system path for UppaalVerifier wrapper to execute property check. If verifyta is not included, the pipeline will still generate an exportable model.

For an artifact-evaluation submission, we strongly encourage the following method of packaging the artifact: Package the solution, the regression suite, this reproducibility appendix, and the three generated diagrams (in Sections III-V) as a single archive, so that a reviewer can reconstruct and rerun all claims in Sections IV and VII without having to find a source repository separately.

APPENDIX B: GLOSSARY

The terms which follow appear throughout the paper and are consistently defined as follows in prose, figures and tables.

● Location a node in a UPPAAL automaton (a state); may have an invariant. In Figs. is drawn as a circle. 2–3.

● Initial location an automaton's start location (double outline, as in Fig. 2's Entry location).

● Guard a boolean condition on an edge, to restrict the fireability of the edge to when guard is true.

● Synchronisation / channel a named event two processes fire together (c! sender, c? receiver); the entire mechanism behind Fig. 3's call protocol.

● Urgent location a location at which time cannot pass while control sits there; used for Done/return locations so that returning is instantaneous (Section IV-D).

● Template / process: a parameterised automaton definition (template) instantiated as a running process; CSharpToUppaal emits one per C# function included and a driver.

● Ignore stubs in Scope closure: The set of functions reachable from the selected entry points in the call graph, computed by ResolveClosure, stopping at functions marked as stubs.

● Translation assumption a recorded note about a construct that was not translated exactly to C#, but instead abstracted, e.g., an unresolved call becoming a bounded nondeterministic choice (Section V-B).

● Call/wait sequence the sequence of location transitions (as illustrated in this paper, Dispatching, PendingResult, and Resumed) a caller template goes through with respect to a single call site, as shown in Fig. 3; the callee's Waiting (idle) and Entry (running) locations are named as named by the implementation.

● Bounded nondeterministic stub the select-choice (Section V-B) as a set of return values rather than a fixed computation. Internally, UppaalLayoutService instruments the layout-quality metric of edge-crossing count, which report how many pairs of edges cross in the rendered diagram (Section VI-A). Compatibility validator UppaalCompatibilityValidator, the part that checks a generated model against structural constraints of UPPAAL 4.1.18 before it is exported, statically (section III).

REFERENCES

[1] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in Formal Methods for the Design of Real-Time Systems, LNCS vol. 3185, Springer, 2004, pp. 200–236.

[2] K. Havelund, T. Pressburger, "Model checking Java programs using Java PathFinder," International Journal on Software Tools for Technology Transfer, vol. 2, no. 4, pp. 366–381, 2000.

[3] J. Corbett et al., "Bandera: extracting finite-state models from Java source code," in Proc. 22nd International Conference on Software Engineering (ICSE), 2000, pp. 439–448.

[4] E. Clarke, D. Kroening, F. Lerda, "A Tool for Checking ANSI-C Programs," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS vol. 2988, Springer, 2004, pp. 168–176.

[5] Microsoft, ".NET Compiler Platform ("Roslyn") SDK documentation," Microsoft Learn, 2023.

[6] R. Alur and D. L. Dill, "A theory of timed automata," Theoretical Computer Science, vol. 126, no. 2, pp. 183–235, 1994.

[7] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," International Journal on Software Tools for Technology Transfer, vol. 1, no. 1–2, pp. 134–152, 1997.

[8] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2002, pp. 1–3.

[9] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for Java," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS vol. 3440, Springer, 2005, pp. 546–550.

[10] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," ACM Computing Surveys, vol. 41, no. 4, pp. 1–36, 2009.

[11] S. Sultana and F. Arif, "Computational Conversion via Translation Rules for Transforming C++ Code Into UPPAAL's Automata," IEEE Access, vol. 5, pp. 14455–14467, 2017.

[12] G. Liva, M. T. Khan, and M. Pinzger, "Semantics-driven extraction of timed automata from Java programs," Empirical Software Engineering, vol. 24, no. 5, pp. 3114–3150, 2019.

International Journal of Advanced Research