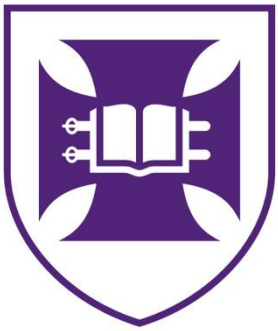

THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

CREATE CHANGE

FORMAL LANGUAGES FOR SECURITY PROTOCOL

Mayank Kumar

Bachelor of Software Engineering (Honours)

The University of Queensland in 2021

Student No. 45936995

The University of Queensland

Acknowledgment

The researcher wishes to express his heartfelt gratitude and warm appreciation to the thesis supervisor, Dr. Naipeng Dong, for constantly making appropriate comments and assisting throughout the process to improve the study and shape and reshape the work to achieve the intended outcome.

Table of Contents

Table of Figures

Abstract

This paper contains two languages, CSP sharp and Applied Pi Calculus which are used to make models for security protocols in tools ProVerif and PAT.

The aim of this project is to bridge the gap between the languages, i.e., to be able to translate one language from another such that a model which has been written in one language can be converted to the other so that it can be verified in the corresponding tool.

This paper discusses numerous strategies for doing this utilizing the methods given in the article. The meaning and translation of code from one language to another have been thoroughly examined.

The translation was examined by independently assessing all the items required for translation in both languages and their corresponding grammatical rules.

The benefit of doing this would be that the model could be checked more extensively as both PAT and ProVerif would check the protocols for different attributes.

Introduction

This paper mainly deals with two tools (ProVerif and PAT) and the languages which are used to verify protocols in those tools. Verification of the protocols utilizing these tools would increase trust in the particular protocols. Both tools check the model for different properties.

PAT is used to check the model for properties like deadlock freeness, divergence freeness, reachability, LTL properties, etc. [3], whereas ProVerif is used to prove properties like secrecy, authentication of the model, and equivalence between process [1].

PAT uses CSP sharp language to model the security protocols, while ProVerif uses applied pi-calculus.

This project aims to bridge the gap between these two languages such that the models could be verified in both tools. So, a translator is to be created, which would translate applied pi-calculus to CSP sharp so that the models written in one language could be translated into another for verification.

PROVERIF ← TRANSLATOR → PAT

C is the language in which the translator is written, and it is written by considering the application of each function, variable, and term, as well as their grammatical rules, and then translating them accordingly.

Because PAT and ProVerif both verify protocols for various characteristics, it would be advantageous if the protocols could be verified in both tools. A model has been translated and validated to see if the translation based on the specific technique works or not and what can be improved in the implemented approach.

Problem Statement

The problem at hand is quite intricate, and it is connected to the tools ProVerif and PAT.
As both the tools have different properties, they can check the protocols for different security aspects. Using both the tools on the same security protocol would extensively find the security protocols' vulnerabilities.

So, the best way to achieve that would be to develop an algorithm that would analyze the grammar rules of a language and then transform it into another language which would not change the meaning of the protocol while preserving the structure of the protocol.

Motivation

There are various driving aspects for this research, including the fact that it deals with two languages in which security protocols are validated.

To create a parser that can convert one language to another, everything about both languages (from meaning to grammatical rules) must be grasped.

It was an excellent motivator for the project because there were a lot of new things to learn, and the end result would be learning how to develop a parser that could translate a working protocol from one language to another functioning protocol.

Background

Security protocol verification

Security protocol verification was an active field of research since the 1990s. For several reasons, this subject is fascinating. Security protocols are all-embracing: for e-commerce, wireless networks, credit cards, and e-voting, etc. Security protocols are known to be faulty. Functional testing cannot uncover security flaws unless an attacker is present. These mistakes may also have profound implications. Therefore, it is particularly desirable that protocols be verified or proven.

Electronic systems are generally designed with an ad-hoc approach currently. Nevertheless, given the quick rise in systems size and complexity, this strategy does not meet the industry's expectations, namely, to minimize the design stage and simultaneously meet all functional, accurate, and performance demands. Formal methods, which are well-known in software engineering, can help address electronic systems with such difficulties [15].

There are several reasons why formal approaches are applied [15]:

• **Clearness:** Formal languages have formal syntax and semantics. Therefore models, including non-deterministic and stochastic behavior, are defined unequivocally. In addition, precision refining and code generation can be implemented.

• **Strict technical analysis**: Formal semantics enable rigorous reasoning for models, such as model control, theorem proofing, and designed algorithms .

These features enable the detection of design defects at the early stage of the design, so that time and resources can be saved, which are needed in the later phases to resolve difficulties.

This project comprises two tools (ProVerif and PAT) and two languages (CSP sharp and Applied pi Calculus). Along with tools and languages, many other tools are used to implement the approaches taken in this project.

ProVerif

It is an automatic cryptographic protocol verifier. It is a tool that can analyze whether a cryptographic protocol is secure or not.
Many different cryptographic primitives can be handled by it, which includes hash functions, symmetric and asymmetric encryption, etc. [1].
It does not have a bound-on no. of sessions of the protocol, i.e., protocols can run parallelly, resulting in getting false attacks sometimes. However, if it verifies and provides the result that the property satisfies some property, then there is no doubt, and it can be assumed that the property is satisfied [1].

Modeling Protocols - A protocol ProVerif model can be separated into three pieces, written in the input language for the tool (the typed pi-calculus). The statements formalize the behavior of primitive cryptography. Macros can be constructed to facilitate the development of sub-processes; and lastly, a primary process can be represented using macros for the protocol itself.

The ProVerif tool can demonstrate accessibility, correspondence, and equivalency.
They can be explained in the following way:

- **Reachability**: Proving accessibility features is the fundamental capacity of ProVerif. This tool allows for the investigation of the terms that an attacker can use, and therefore (syntactic) words secrecy may be assessed in relation to a model. The following queries are added in the input file prior to the primary procedure to test the secrecy of the word M in the model: query attacker (M) - where M is a ground term that contains free names without destructors (potentially private and so not known to an attacker initially) [1].

- **Authentication**: Authentication implies that if participant A seems to run the protocol with participant B, then B seems to run the protocol with A, and vice versa. It is frequently required that A and B have the same protocol parameter values.
  Generally, authentication is formalized by correspondence characteristics of the following form: if A performs a given event e1 (for example, A ends the

protocol with B), then B has performed a particular event e2 (for instance, B started a session of the protocol with A). There are various variations on these qualities.

- **Secrecy**: Secrecy means that protocol-altered data cannot be obtained by an opponent. In two ways, secrecy can be officialized [11]:

    1. Secrecy most typically means that the opponent cannot accurately calculate the facts considered. This attribute is simply labeled a secret in this survey or syntactic secrecy when the emphasis is required.

    2. Sometimes a more vital idea is used, strong secrecy, which means the opponent cannot discover a change in secret value. This means that the enemy has no knowledge of the significance of secrecy at all.

Applications of ProVerif

There is various applicability of ProVerif [1]:

Needham-Schroeder protocol (original version and the updated version) is discussed later in the paper has been successfully analyzed using ProVerif. Many more protocols have been appropriately examined using this tool.

There are several other applications of ProVerif like correspondence assertions to verify email protocol [1] which was explained before.

The resistance to the attacks due to password guessing was also demonstrated for the protocols based on passwords.

Results interpretation in ProVerif

It is crucial to grasp the distinction between the derivation of an attack and the trace of an assault to grasp the results appropriately. The attack derivative shows the actions of the attacker in the internal representation of ProVerif to break the security property. Due to the use of abstractions in this internal representation, in fact, the derivative cannot always be executed. For example, it may need repeats of certain acts which cannot, in fact, be repeated, for example, because they are not replicated. In contrast, the attack trace refers to the semantics of the calculus applied and is always an executable sign of the process under consideration.

Three kinds of results are displayed in ProVerif [1]:

- **[Query] RESULT is true:** the query has been proven; no attack has taken place. ProVerif does not display an attack derivative or an attack trail in this scenario.
- **[Query] RESULT is false:** ProVerif has found an attack on the targeted security property. The query is incorrect. The attack trace is indicated shortly before the outcome (and there is also an attack derivative but focussed on the attack trace because it depicts the actual attack). This trace is shown.
- **[Query] Unable to prove RESULT**: That is an answer that you do not know. ProVerif was unable to prove that the query is valid or to develop an attack that would indicate that the query is incorrect. As the problem of protocol verification for an unlimited number of meetings is unclear, this is an inevitable situation. Nevertheless, ProVerif provides some additional information that can be valuable if the query is valid.

```
--------------------------------------------------------------
Verification summary:

Query not attacker(secretANa[]) is true.

Query not attacker(secretANb[]) is true.

Query not attacker(secretBNa[]) is true.

Query not attacker(secretBNb[]) is true.

Query inj-event(endBparam(x,y)) ==> inj-event(beginBparam(x,y)) is true.

Query inj-event(endBfull(x1,x2,x3,x4,x5,x6)) ==> inj-event(beginBfull(x1,x2,x3,x4,x5,x6)) is true.

Query inj-event(endAparam(x,y)) ==> inj-event(beginAparam(x,y)) is true.

Query inj-event(endAfull(x1,x2,x3,x4,x5,x6)) ==> inj-event(beginAfull(x1,x2,x3,x4,x5,x6)) is true.

--------------------------------------------------------------
```

Figure 1 Needham Schroeder protocol verification result in Proverif

Applied Pi-Calculus

The Dolev-Yao paradigm underpins Applied Pi-Calculus. Concurrent processes and their interactions are depicted using this language. It is a more recent form of pi-calculus. Communication and concurrency are the constructs that are inherited into the applied pi-calculus by pi-calculus.

The syntax of applied pi-calculus is defined in Figure 2. There are various names a, b, c, k, s which signify the data items. The data items represented by this are keys and nonces. The x, y, z depicts an infinite set of variables. Function symbols are used to represent constructors like (f) and destructors (g). Constructors in applied pi-calculus build new terms in the following way f(M1,…., Mn).
Since there are many representations of the terms M, N, i.e., it can represent a variable, name, or it can even represent a constructor application f(M1,.., Mn).

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\mathsf{out}(M, N).P$	output
$\mathsf{in}(M, x).P$	input
$\mathbf{0}$	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q$	destructor application
$\mathsf{let}\ pat = M\ \mathsf{in}\ P\ \mathsf{else}\ Q$	pattern-matching
$\mathsf{event}(e(M)).P$	event
$P + Q$	internal choice

Figure 2: Syntax of the process calculus [2]

The terms in processes are manipulated by the destructors in the following manner [2]:

- Out(M, N). P executes as follows – message N is outputted on channel M then P is executed.
- In(M,x) . P executes as follows - a message is inputted on channel M then P is executed where the input messages are bound by x.
- The 0 represents a null process, i.e., it does nothing.
- P|Q is the parallel composition of P and Q.
- !P represents replication, i.e., infinite copies of P in parallel.
- (va)P represents the creation of a new name a, and then P is executed.
- let x = g(M1, . . . , Mn) in P else Q: - when this is executed, it tries to evaluate the first term g(M1, . . . , Mn), and if that is successful, then x would be bound to the result, and then P would execute otherwise Q is executed. Data structures and cryptographic operations could be represented using constructors and destructors.
- let pat = M in P else Q:- when this is executed, it matches M with the pattern pat, and if the matching is successful, then it executes P. Otherwise, it executes Q if the matching is unsuccessful. The pattern pat can be used in either way, i.e., like a variable x of a data constructor application f(pat1,…..,patn). As pattern pat is linear, it doesn't contain many occurrences of the same variable.
- ProVerif represents authentication as a set of correspondence statements, such as "if event e(x) has occurred, then event e′(x) has occurred." The process calculus provides a command for carrying out such events: process event(e(M)). The event e(M) is executed by P, and then P is executed.
- An internal choice build that was not contained in [5] was added: the P + Q process is non-deterministically either as P or Q. This build will help to define our list extension.

RELATED WORK: -

In this related work, a variant of applied pi calculus is used to model protocols with a list of unbounded lengths. A formal meaning is provided, and it translates it automatically to the Horn clause and proves that the translation done is correct [2].

PAT (Process Analysis Toolkit)

PAT supports the composition, simulation, and reasoning of concurrent, actual-time systems and other conceivable domain names within a self-contained framework. It has user-friendly interfaces, an animated simulator, and a model editor.

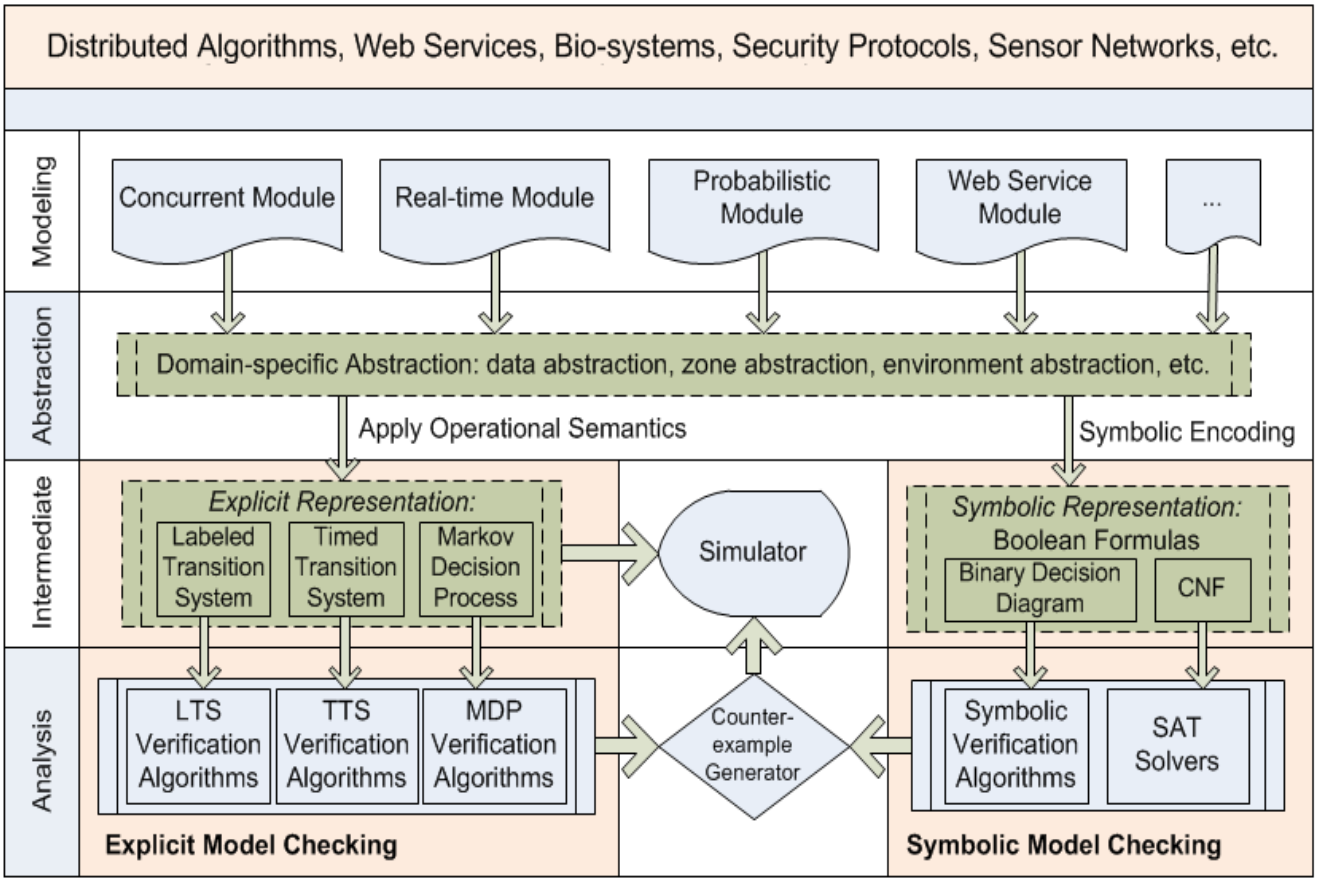

Figure 3 PAT Framework [10]

This framework enables the development of new languages with easy syntax rules and semantics. So far, eleven modules, namely CSP, Real-Time System Module, Probability CSP Module, RTS Module Probability, LASS Module, Timed Automata Module, NesC Module, ORC Module, State Flow(MDL), Security Module, and Web Service module(W.S.), are being built. So far, 11 modules have been produced. The targeted systems will, in the future, incorporate distributed systems, UML (state chart and sequence diagrams) [10].

Above all, PAT is carrying out many model controls for various characteristics, including deadlock-freeness. A deadlock occurs when there is at least one process waiting for resources to be released by another process to finish a task correctly. Hence, it checks that a deadlock does not occur, divergence-freeness, accessibility, fairness-assumed LTL features, smooth control, and probabilistic model monitoring. Advanced optimization techniques, such as partial order decreases, symmetry reduction, counter abstract processes, parallel model control, are used in PAT to obtain good performance.

The PAT modeling language (i.e., the input language of PAT) is, of course, communicating the process algebra-style sequential processes (CSP), which combines high-level modeling-operators like (conditional or non-deterministic), parallel composition interruption, (alphabetically) hiding, asynchronous message transmission channel, etc. Because of these factors, PAT offers a wide variety of models [6].

PAT consists of an easy-to-use editing environment with input models CSP style, powerful simulator, and models. PAT is to be designed as an overall framework that can be expanded readily to include additional languages of modeling or new languages of affirmation (as well as specific algorithms of verification) [6].

The key features of PAT are as follows [10]:

- Friendly editing environment for introducing models for multi-document, multilingual, I18N GUI, and extensive editing features are some of the key features.
- User-friendly simulator for system behavior; random simulation, user-guided step-by-stage simulation, complete graph construction in the state, trace reproduction, counterexample viewing, etc., makes it better than various other verification tools.
- Easy checks for deadlock-free analysis, accessibility analysis, linear status/event-time logic checks (with or without fairness), and refinement checks.
- A wide variety of integrated examples ranges from benchmarks to the algorithms/protocols newly developed.

Special features of PAT:

In this part, certain distinct features of PAT are compared to other existing PAT tools.

To verify liveness traits, certain aspects such as fairness are typically necessary for the fair resolution of non-determinism. PAT supports several forms of fairness reclassification to the level of events/processes and weak/substantial equity to meet varied requirements.

- **Parallel verification:** which uses the current typical computer multi-core architecture, provides efficient techniques to resolving sequential algorithms challenges, as well as a technique to incorporate fairness with existing parallel models.

- **Infinite Systems Verification:** Instead of checking the satisfaction of each process that makes the problem undecidable, PAT utilizes the abstract counter-techniques process to group many similar processes together and control their properties as an abstract process group. It is also recommended that the checking of fairness characteristics is crucial for the new approach for checking these systems against the formulae of Linear Temporal Logic (LTL).

- **Linearizability verification:** It is a critical accuracy criterion for standard items. Operations without linearization on shared objects can make the system inconsistent, which can lead to mistakes, including catastrophes. Existing methods to verify linearization demand that users have unique expertise and cannot be automatic about linearization points. PAT uses a novel method for monitoring linearity based on the refining of these disadvantages.

CSP# (Communicating Sequential Programs)

The original CSP is named entirely on the integrated syntax restriction that processes be part of the language's sequential subset. CSP passed the time test. The design of many current programming and specification languages has been generally accepted and influential. Nevertheless, it is still challenging to represent systems that use languages like CSP entirely with non-trivial data structures and functional characteristics.

CSP facilitates communication between processes by transferring messages but not sharing memory, e.g., via sharing variables. It has been recognized that a variable can be modeled in parallel with its process for a long time. The user processes then read from the CSP communication variable or write to it. While feasible, this is difficult for non-trivial data structure systems (for example, arrays) and operations (e.g., array sorting). 'Syntactic sugars' are therefore most often welcome, such as shared variables [9].

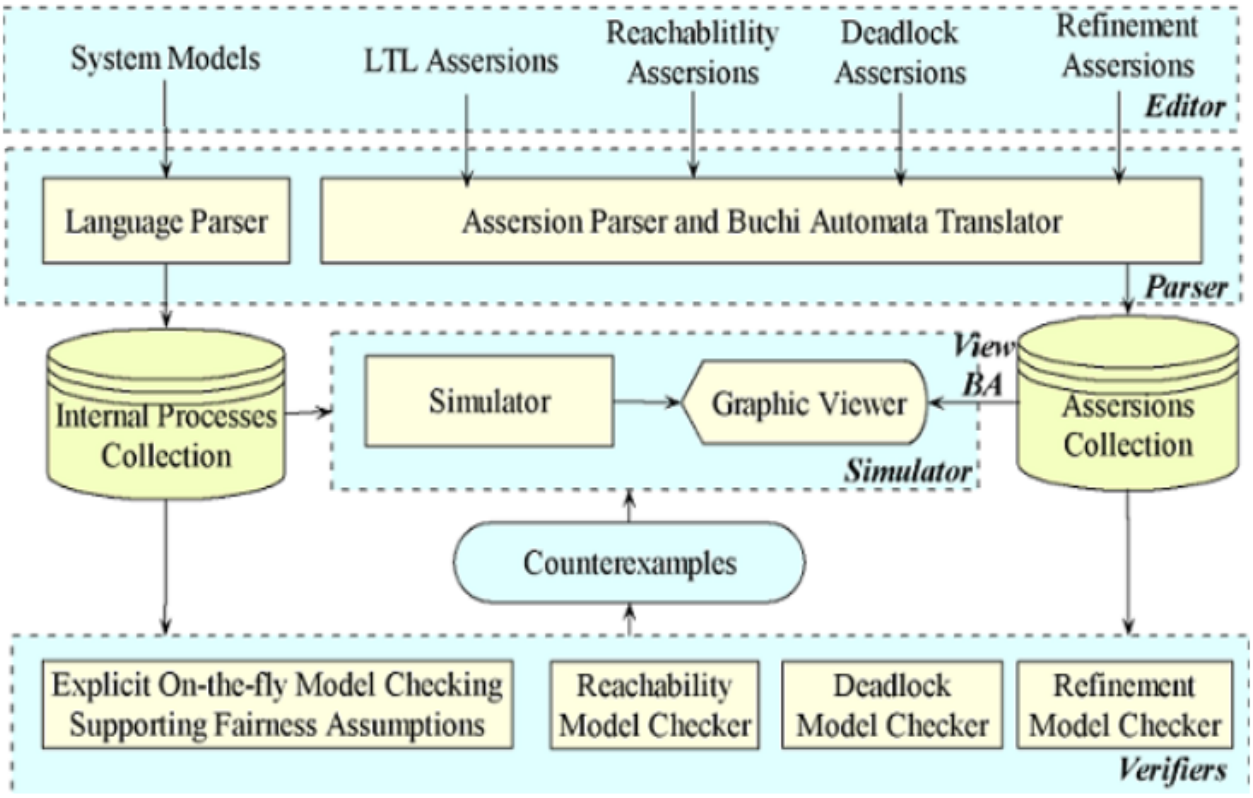

Figure 4 CSP MODULE'S WORKFLOW [10]

The above shows the CSP Module's workflow. In PAT system analysis, simulation or model controls are supported twofold. The graphical simulator enables the users to play with their models interactively, selecting one of the actions enabled and enabling the computer to alter system traces or even construct the full state chart (given it is not very large). The PAT-embedded model control systems are developed for the use of state-of-the-art methods for systemic analysis.

Hoare's traditional communications sequence process (CSP) has been an event-oriented modeling language for decades, which was quite successful. CSP has enhanced formal approaches in many respects in its theoretical development. Its distinctive features, such as alphabetized parallel composition, have proven valuable in designing a variety of systems.

CSP# models with full operational semantics can be executed and are subject, above all, to system simulation and completely automatic methods of system verification, such as model verification. CSP# module of PAT supports a rich CSP# modeling literature which combines high-level modeling operators such as (conditional or non-deterministic) options, interrupt parallel composition, (alphabetical) interleaving, hiding or transmission of a channel, etc., with low-level programmer-favored constructs such as variables, it offers significant flexibility in the way your systems are designed. The CSP# design principle is to retain as a sub-language of the original CSP# as a maximum while connecting the data states and executable data operations [9].

CSP# language constructs may be divided into the following categories.

- The first category consists of the primary subset of CSP operators, which includes event prefixing, internal/external options, alphabetized lock-step synchronization, conditional branching, interrupt, recursion, and so on.

- The second category comprises language features that might be considered "syntactic sugar" (according to CSP), such as global shared variables and asynchronous channels. CSP has long been renowned for its ability to describe shared variables or asynchronous channels as processes. However, the

specialized language structures provide excellent usability and may improve verification efficiency.

- The third group consists of a collection of event annotations. It is well understood that process algebra, such as CSP or CCS, merely specifies safety. Using an event-based compositional language, the event annotations provide a versatile means of expressing fairness.

- The last group is the language used to express assertions, which may then be automatically validated using the built-in verifiers.

SYNTAX

Process Definition	chan!expr, chan?expr	Channel
	e->P	Event Prefix

	case {
	cond1: P1
	default: P
	}
		Case

	atomic{P}	Atomic Sequence
	if (cond) { P } else { Q }	Conditional Choice
	ifa (cond) { P } else { Q }	Atomic Conditional Choices
	ifb (cond) { P }	Blocking Conditional Choices
	[cond]P	Guarded Process
	P;Q	Sequential Composition
	P\{e1,..,en}	Hiding
	P interrupt Q	Interrupt
	P[*]Q	External Choice
	P<>Q	Internal Choice
	P[]Q	General Choice
	P||Q	Parallel Composition
	P|||Q	Interleaving
	P(x1, x2, ..., xn) = Exp;	Definition

Figure 5 Operator precedence [10]

The operators at the top of the table bind more firmly than the operators at the bottom [10].

Below is a complete study of the syntax [10]: -

Global Definitions

#Define max 5;

#Define is a keyword that can be used for a variety of things. It defines a global constant named max with the value five here. The semi-colon signifies the conclusion of the sentence. It should be noted that the constant value can only be an integer (both positive and negative) or a Boolean value (true or false).

var knight = 0;

var is a keyword used to define a variable, and the knight is the variable's name. Knight has a value of 0 at first.

Var<Type> x; (USER DEFINED VARIABLE DECLARATION)

The Type class's default function Object () { [native code] } will be invoked.

Var<Type> x; new Type(1, 2);

A function Object () {[native code]} with two int arguments will be invoked.

PAT allows users to design any data structures and utilize them in PAT models to make modeling easier.

channel c 5; (channel)

the channel is a keyword used while defining channels, c is the channel name, and 5 is the channel buffer size; the channel buffer size must be larger than or equal to 0. A channel with a buffer size of 0 broadcasts and receives messages simultaneously. This is used to simulate pair-wise synchronization between two parties.

if (goal) { P } else { Q }; (if else condition)

If x is 0, perform P; otherwise, perform Q.

#include "c:\example.csp"; (used to insert libraries as well)

If the model is too large, it may be divided into separate files and included in the main model using the include keyword.

Process Definitions

P(x1, x2, ..., xn) = Exp;

P is the name of the process, x1,..., xn is a list of possible process parameters, and Exp is a process expression. The computational logic of the process is determined by the process expression, i.e., Exp.

Processes can communicate with one another using channels. Channel input/output is written in the same manner as primary event prefixing.

```
c!a.b -> P          -- channel output

c?x.y -> P          -- channel input

c?1 -> P            -- channel input with expected value
```

Figure 6 Channel input and output [10]

C denotes a channel, a and b are expressions that evaluate values (at run time), and x and y are (local) variables that accept input values. Before using a channel, it must be defined. If the channel buffer is not yet complete, the compound value calculated from a and b is placed in it. The procedure waits if the buffer is full. If the buffer is not empty for channel input c?x.y, the top element on the buffer is fetched and allocated to local free variables x and y. Otherwise, it sits and waits. If the buffer is not empty and the top element value is 1, the top element on the buffer is retrieved for channel input c?1. Otherwise, it sits and waits.

e -> P

A simple event is a term for a type of observation. Given a process P, the following defines a process that first executes e and then acts as process P specifies.

if (cond) { P } else { Q }

if (cond1) { P } else if (cond2) { Q } else { M }

cond is a Boolean expression. If cond is true, P is executed; otherwise, Q is executed. It is worth noting that the else-part is optional. The process if(false) P behaves identically to the process Skip.

case { (Case)

cond1: P1

cond2: P2

default: P

}

The essential term is the case, and the Boolean formulas cond1, cond2 are used. P1 is executed if cond1 is true. Otherwise, P2 if cond2 is true. Furthermore, if both cond1 and cond2 are false, P is executed by default. One by one, the conditions are reviewed until a true one is identified. If no condition is met, the default process is run.

[cond] P (guarded process)

P is a process, and cond is a Boolean formula. P performs if cond is true. In contrast to conditional choice, if cond is false, the entire process will wait until cond is valid before executing P.

P* || *Q (parallel composition)

Parallel composition is denoted by the symbol ||. In contrast to interleaving, P and Q can conduct lock-step synchronization, which means they can both conduct an event at the same time.

***P [] Q* (General choice)**

The choice operator [] specifies that P or Q may execute. If P is the first to conduct an event, P gains control. Otherwise, Q assumes command.

***P [*] Q* (External choice)**

The choice operator [*] indicates that either P or Q can be executed. If P first conducts a visible event, P gains control. If Q initially conducts a visible event, Q gains control. Otherwise, the option is still available.

***P <> Q* (Internal Choice)**

P or Q may carry out the execution. The decision is determined internally and in a non-deterministic manner.

P ||| Q (Interleaving)

Interleaving is denoted by the symbol |||. Except for termination events, both P and Q can do their local activities without syncing with one other. If just one process creates a termination event, that termination event cannot be immediately performed unless all processes emit a termination event.

These were some of the main definitions used in PAT, which are necessary for the translation, which has been talked about later in the paper.

Let us now examine how all of the previously supplied information is employed in the creation of the translator.

Approach

A parser that could translate the code from applied pi calculus to CSP crisp might be created using various ways.

Even though there are numerous approaches, the most crucial component of creating a parser is understanding how the grammar rules in both languages function and how to translate it from one language to another without affecting the message.

To understand how the grammatical rules functioned, a protocol confirmed in both languages was used, namely the Needham Schroeder public-key protocol.
The primary technique used throughout may be summarised as follows:

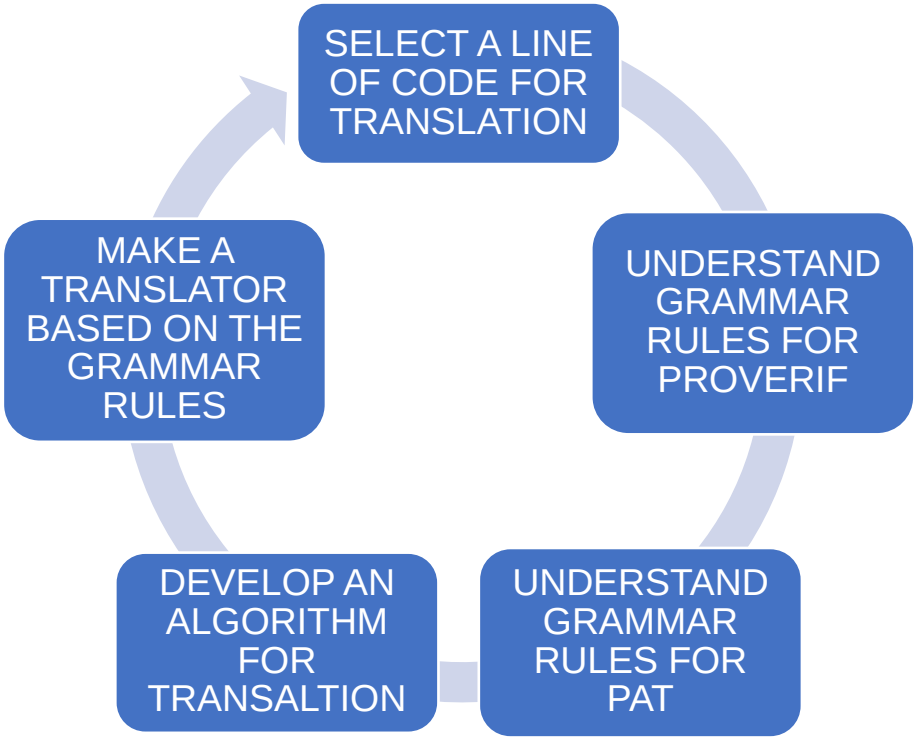

Figure 7 Process of approach.

The steps taken in this process were necessary to be able to understand all the rules for the implementation. The figure below states the essential translations and elements which were used from the library “Lib_v0+equal+know”.

ProVerif	Method	PAT3	#import "Lib_v0+equal+know"; (elements used from lib in pat)

type <ident>
<options>
Types
Free c : channel
Channels
	var<Type> x ; //default
	constructor of Type
	class will be called.
		var<Constant> A = new Constant();
		var<Nonce> Na=new Nonce();
		var<SKey> SKa=new SKey();
		
		channel ca 0;
			Constantequal
			Constantnonce

in(c, (xA: host, hostX: host));	Library - "Pair" which inherits Bitstring (a pair is a Bitstring)	X = new Pair (xA, hostX); channelS?X->	IsSEnc
(!processInitiator())| Processes	once started, the sequential program continues to execute until it finished without being interrupted.	P() = { } -> P();	Pair

let processInitiator() =
(Processes)
If then, input, output,
Using conditionCheck
	If (cond) { if condition is met then true} ->
	c!a.b -> P -- channel output
	c?x.y -> P -- channel input
		//Process initiator
		PIni() =
		//Event beginparam and if
		InibeginparamAB { if (A.Constantequal(sender) && B.Constantequal(receiver)) {
		InibeginparamAB = true; } } ->
		//channel input
		channelS!sender.receiver ->
		//channel output
		channelS?[conditionCheck.IsSEnc()==true]conditionCheck ->
			PEnc
			PDec

out(c, encrypt((Na, xA), pkX)); Encryption	Using SEnc, PDec, SDec , Constantequal. PEnc, PRes, result from the library.	ServerEnc=new SEnc(new Pair(Na,xA), pkX); ChannelA!ServerEnc ->	getfirst getsecond result

Figure 8 Important translations and library elements.

PROVERIF

- x,y, z variable
- f (M1,..., Ml) function application
- P,Q, R ::= processes (or plain processes)
- 0 null process
- P | Q parallel composition
- !P replication
- vn.P name restriction (“new”)
- if M = N then P else Q conditional
- N <x>.P message input
- $\overline{N}\langle M\rangle.P$ message output

PAT

- var x;
- Id '(' msg ')' //function declare
- A process is defined as an equation in the following syntax, *P(x1, x2, ..., xn) = Exp;*
- ESC(escape)
- *P || Q*
- P () { } -> p() (replication)
- var<Constant> host = new Constant();
  var<Nonce> Na=new Nonce();
- *if (cond) { P } else { Q }*
- *c!a.b -> P* -- channel output
- *c?x.y -> P* -- channel input

Figure 9 Syntax and Informal Semantics used in ProVerif and PAT

Implementing the approach

As figures 8, 9 depict, there were numerous things that were to be translated from ProVerif to PAT. This was achieved using the following procedure:

Please keep in mind that the translation technique was created from the many meanings gathered in the background of both applied pi calculus and CSP#.

Detailed translation approach

Applied Pi-Calculus	**Type host**
CSP#	var<Constant> host = new Constant();
Library usage **Lib_v0+equal+know**	The data type assigned to the host is constant, which is a bitstring.
Method	PAT allows users to design any data structures and utilize them in PAT models to make modeling easier. *var<Constant> x = new Constant();*
Challenges	The issues encountered during the translation of the host were that there is no specific data type named host in the PAT library, thus in order for the host to fulfill the requirements, its data type was transformed to bitstring in PAT.
Limitations	There were no restrictions because the hosts were turned into bitstrings. However, there are certain limits because hosts are not a data type in PAT.

Applied Pi-Calculus	Type nonce
CSP#	var<nonce> Na = new nonce();
Library usage Lib_v0+equal+know	The data type assigned to nonce is nonce which inherits a bitstring.
Method	PAT allows users to create any data structure and use it in PAT models to make modeling easier. var<nonce> x = new nonce(); Thus, this line declares a variable x of type nonce that inherits a bitstring.
Challenges	The issues encountered during the translation of the nonce were that there is no specific data type named nonce in the PAT library. Thus in order for the nonce to fulfill the requirements, its data type was transformed to bitstring in PAT.
Limitations	There were no restrictions because the nonces were turned into bitstrings. However, there are certain limits because the nonce is not a data type in PAT.

Applied Pi-Calculus	**free c: channel.**
CSP#	channel c 0;
Library usage **Lib_v0+equal+know**	N.A.
Method	For channel c; the channel is a keyword used exclusively for defining channels, c is the channel name, and the channel's buffer size is one. Hence in channel c 0; The buffer size would be zero.
Challenges	There were not many challenges faced during the translation of the channel from Proverif to PAT as it already had a data type for channels.
Limitations	There were no limitations for the translation of channels from ProVeirf to PAT.

Applied Pi-Calculus	**type skey.**
CSP#	var<SKey> Sk = new SKey();
Library usage **Lib_v0+equal+know**	The data type assigned to Skey is Skey which inherits a Key.
Method	PAT enables users to create any data structure and use it in PAT models to make modeling easier. varSKey> SK = new Skey(); This line so generates a variable S.K. of type SKey that inherits a Key, i.e.; It is a key.
Challenges	The problems noticed during the translation of the skey were that the PAT library does not include a particular data type named SKey. As a result, in order for the skey to meet the criteria, its data type was changed to Key in PAT.
Limitations	Skey is a secret key in applied pi calculus, but in CSP sharp, a secret key does not exist, so it is translated as a key.

Applied Pi-Calculus	**type pkey.**
CSP#	var<PKey> Pk = new PKey();
Library usage **Lib_v0+equal+know**	The data type assigned to Pkey is Pkey which inherits a Key.
Method	PAT allows users to design any data structures and utilize them in PAT models to make modeling easier. *var<PKey> PK = new Pkey(Sk);* Thus, this line creates a variable P.K., which is of type PKey that inherits a Key, i.e., It is a key. Sk is the corresponding private key of the public key.
Challenges	The issues encountered during the translation of the pkey were that there is no specific data type named PKey in the PAT library and how it will be able to use its corresponding private key. Thus in order for the pkey to fulfill the requirements, its data type was transformed to Key in PAT.
Limitations	Pkey is a public key in applied pi calculus, but in CSP sharp, public and private keys do not exist, so it is translated as a key and used to perform their specified tasks like private and public keys.

Applied Pi-Calculus	**Let process() = P**
CSP#	Process() =
Library usage **Lib_v0+equal+know**	N.A.
Method	Macros can be used to specify sub-processes in declarations in applied pi calculus. let R(x1 : t1, . . . , xn : tn) = P . R is the name of the macro, P is the name of the sub-process being defined, and the free variables of P are x1,..., xn of type t1,..., tn, respectively. In CSP sharp process is defined as follows: - *P(x1, x2, ..., xn) = Exp;* In this P is the process name, the information inside () are the process parameters, and EXP depicts the process expression.
Challenges	There were various issues encountered during the translation of processes as there are many different process definitions, and every one of them had to be translated individually so that the implied meaning of the code does not change.
Limitations	Not all the process definitions could be translated correctly, but the general application of the process could be translated.

Applied Pi-Calculus	**encrypt(nonce_to_bitstring(NX2), pkX)**
CSP#	var<PEnc> ProbabilisticEnc;
Library usage **Lib_v0+equal+know**	"Probabilistic Encryption - PEnc," which is derived from bitstring
Method	The technique employed here is relatively straightforward, as the input data for encryption is a bitstring and a key, and it returns a bitstring. encrypt(nonce to bitstring(NX2), pkX), which may be translated as follows when the new bitstring data type is declared: abcd = new PEnc(nonce to bitstring(NX2), pkX) the argument, which can be of any kind, the nonce, and the public key is used to encrypt it.
Challenges	There are many challenges in the translation of encryption, as in ProVerif, and there are two types of encryption, i.e., symmetric encryption and asymmetric encryption.
Limitations	As there are symmetric and asymmetric encryptions in ProVerif, it was not possible to get the functionality of both the encryptions, and the closest was to use bitstring as the ciphertext is in bitstring when encrypted.

Applied Pi-Calculus	**sdecrypt(sencrypt(x,y),y) = x.**
CSP#	var<PDec> ProcessDec;
Library usage **Lib_v0+equal+know**	"Decryption - PDec," it returns the output as a bitstring.
Method	The method used here is quite simple as for decrypting the input data is bitstring (which is an encrypted ciphertext) and a key, and it returns a bitstring. sdecrypt(nonce_to_bitstring(NX2), pkX) which could be translated as follows after the declaration of the new bitstring data type:- x = new DEnc(sencrypt(x,y),y) the parameters should be an encrypted bitstring, and the other parameters should be a key that is used to decrypt the encrypted data.
Challenges	As mentioned previously in encryption ProVerif, there are two types of decryption, i.e., symmetric decryption and asymmetric decryption.
Limitations	As there are symmetric and asymmetric decryptions in ProVerif, it was not possible to get the functionality of both the decryptions, and the closest was to use bitstring as the ciphertext is in bitstring when decrypted.

Applied Pi-Calculus	**in(x, y); P**
CSP#	*c?x.y -> P*
Library usage **Lib_v0+equal+know**	Pair - it inherits a bitstring There are two elements in a pair in which either can be of any type.
Method	Applied Pi-Calculus (ProVerif) In this case (x,y). P runs as follows: a message is sent through channel x, and then P is run where the input messages are constrained by y. CSP# (PAT) If the buffer is not empty for channel input c?x.y, the top element on the buffer is fetched and allocated to local free variables x and y.
Challenges	There were a few challenges faced in the input of messages. If there were more than one message to be inputted, then the method used was to pair them and then send the paired message as a bitstring.
Limitations	There were no limitations except for the sending of numerous messages at once in the translation of output messages.

Applied Pi-Calculus	**out(x, y); P**
CSP#	*C!x.y -> P*
Library usage **Lib_v0+equal+know**	Pair - it inherits a bitstring There are two elements in a pair in which either can be of any type.
Method	Applied Pi-Calculus (ProVerif) In this case (x,y). P runs as follows: P is prepared to send y on channel x, followed by P. When P is 0, we may omit it in both of these circumstances. CSP# (PAT) If the channel buffer is not yet complete, the compound value computed from a and b is stored in the channel buffer (FIFO queue). The procedure waits if the buffer is full.
Challenges	A few issues arose throughout the message output process. If more than one message needed to be sent, the method utilized was to pair them and then transmit the paired message as a bitstring.
Limitations	Except for the transmission of several messages at once in the translation of output messages, there were no restrictions.

Applied Pi-Calculus	**!P (Replication)**
CSP#	P() { …. } -> P()
Library usage **Lib_v0+equal+know**	N.A.
Method	Applied Pi-Calculus (ProVerif) Replication !P is the infinite composition P | P |..., which is frequently used to represent an unlimited number of sessions [1]. CSP# (PAT) P () { } -> P(); In PAT, the process instructs itself to provide the same outcome as replication in Proveif.
Challenges	There were a few challenges faced in the replication of processes, i.e., ProVerif does not have a bound on no of sessions, but PAT does have a bound on the no. of sessions running.
Limitations	ProVerif does not have a limit on the number of sessions, but PAT has a limit on the number of running sessions.

Applied Pi-Calculus	**if M then P else Q**
CSP#	*if (cond) {* *P* *} else {* *Q* *}*
Library usage **Lib_v0+equal+know**	N.A.
Method	Applied Pi-Calculus (ProVerif) If M, then P; else, Q is a normal conditional: When the boolean term M evaluates to true, it executes P; otherwise, it executes Q. When the term M fails, it does nothing. CSP# (PAT) where cond is a Boolean expression If cond is true, P is executed; otherwise, Q is executed.
Challenges	There were few difficulties encountered during the translation of conditional arguments because both PAT and ProVerif use them in a similar manner.
Limitations	N.A.

These were the direct translations necessary to convert a model from ProVerif to PAT. Apart from the obstacles indicated in the translations above, there were some additional obstacles that proved to be significant, such as the translation of an attacker from ProVerif to PAT, which is not achievable in most circumstances, making it one of the constraints of translating.

There were a few more crucial functions from the library that were used to make the translation as accurate as feasible. Some of the functions and their meanings are as follows:

Lib_v0+equal+know	**Meaning**
constantequal	determines the equality of two constants
constantnonce	obtains the equivalence of two nonces
IsSEnc	check to see whether it is an encrypted string
Getfirst	the first element is returned
getsecond	the second element is returned
Result	returns the outcome

The most significant aspect of the endeavor was translation. After the translation was completed, the next stage in translating from ProVerif to PAT was to study the grammatical rules of both tools and then employ them appropriately to translate effectively.

Figure 9 shows the sequence of translation that takes place when a user enters a ProVerif file for Translation into PAT.

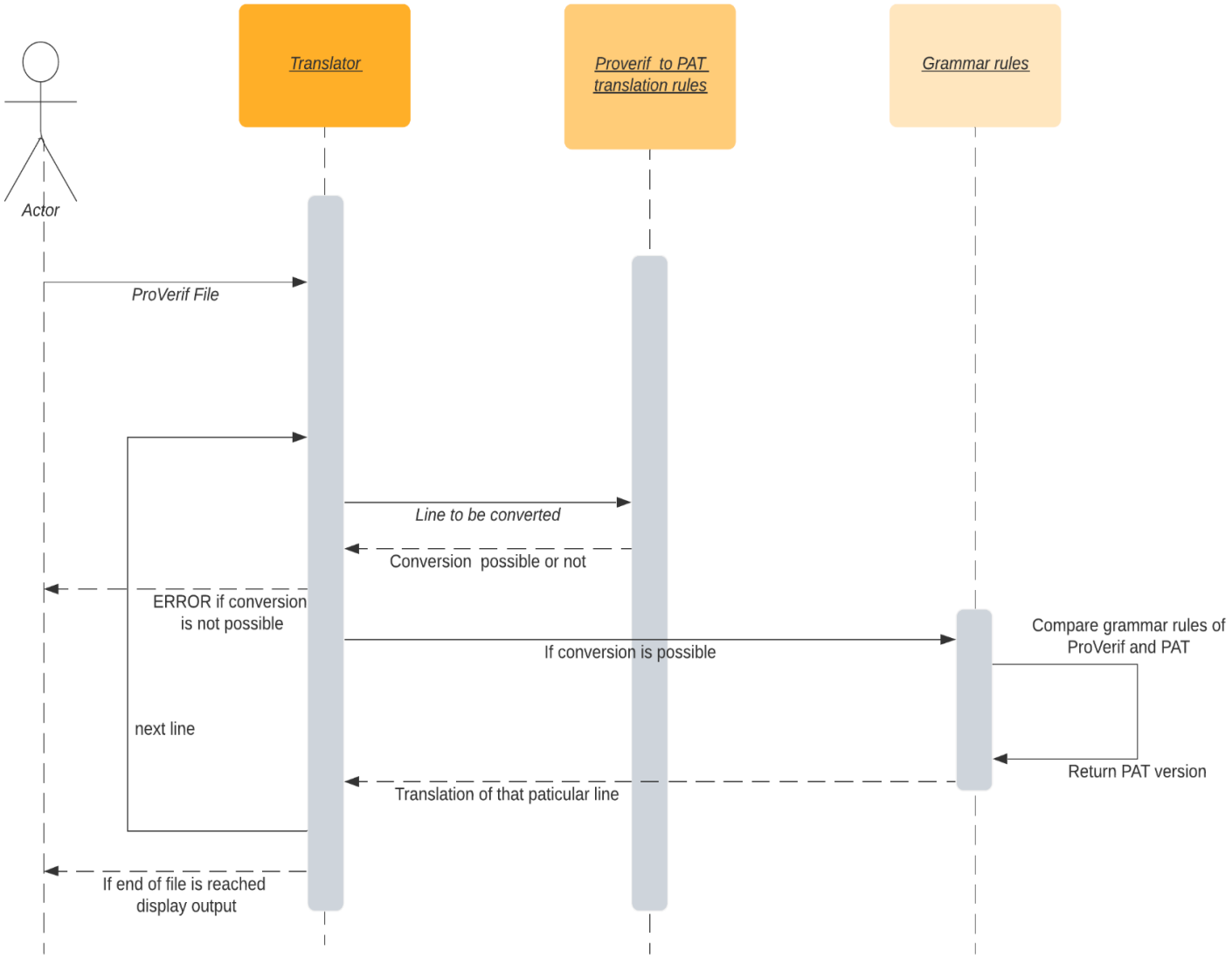

Figure 10 Sequence diagram of the translation

Following the translation rules from ProVerif to PAT, the grammar for both languages is compared and translated accordingly, as shown in the sequence diagram.

C is used as the language in which the translator was created, and it used the grammatical principles listed below to translate:

NOTE: The most significant grammatical rules are presented here. The remaining grammatical rules will be given in the appendix.

For	Grammar Rule Proverif	Grammar rule PAT
Type	type <indent> <options>	'var' ID ('<' datatype=ID '>')?variableRange? ('=' expression)? ';'*//user defined datatype is supported using<type>*

The technique described here is utilized for **type** translations from ProVerif to PAT. Similarly, some of the other grammar rules are listed below: -

For	Grammar Rule Proverif	Grammar rule PAT
Channel	channel seq+ <ident>.	'channel' ID ('[' expression ']')? ';'

Channel

For	Grammar Rule Proverif	Grammar rule PAT
Condition (if else)	if <pterm> then <process> [else <process>]	'if' '(' conditionalOrExpression ')' '{' interleaveExpr '}' ('else' '{' interleaveExpr '}')?

Condition (if-else)

For	Grammar Rule Proverif	Grammar rule PAT
input	in(<pterm>,<pattern>) <options> [; <process>]	name=ID '?' msg=ID ('.' conditionalOrExpression)* '->' assignmentExpr

Input messages

For	Grammar Rule Proverif	Grammar rule PAT
output	out(<pterm>,<pterm>)) [; <process>]	name=ID '!' msg=ID ('.' conditionalOrExpression)* '->' assignmentExpr

Output messages

For	Grammar Rule Proverif	Grammar rule PAT
process	let <pattern> = <pterm> [in <process> [else <process>]] | <ident>[:<typeid>] <- <pterm> [;<process>] [1]	ID ('(' (parameter(',' parameter)*)? ')')? '=' interleaveExpr ';'

Process declaration

After understanding the essential grammatical rules, let's look at why this method is chosen over others.

Translating using this approach was one of the best approaches that were available. Other approaches could have been used (but the basic approach of translation of the terms remains the same), like using ANTLR by feeding both the grammar rules as ANTLR= supports grammar inheritance as a technique for constructing a new grammar class based on a base class. Both the grammatical structure and the grammar-related behaviors can be changed independently.

Then rules are to be generated such that the translations work as required. The rules in it are usually in the form:

```
rulename
: alternative_1
| alternative_2
 ...
| alternative_n
;
```

If necessary, parameters may be added to the rules, and it would also be an excellent way to utilize ANTLR. However, the overall element of the translation would remain the same, i.e., and the translation rules would remain the same.

As a result, regardless of the tool or language used to create the translator, the core strategy of translating all the components and applying grammar rules to translate correctly would stay the same, making this the only practical solution.

Evaluation

The process used to evaluate the generated result is as follows: -

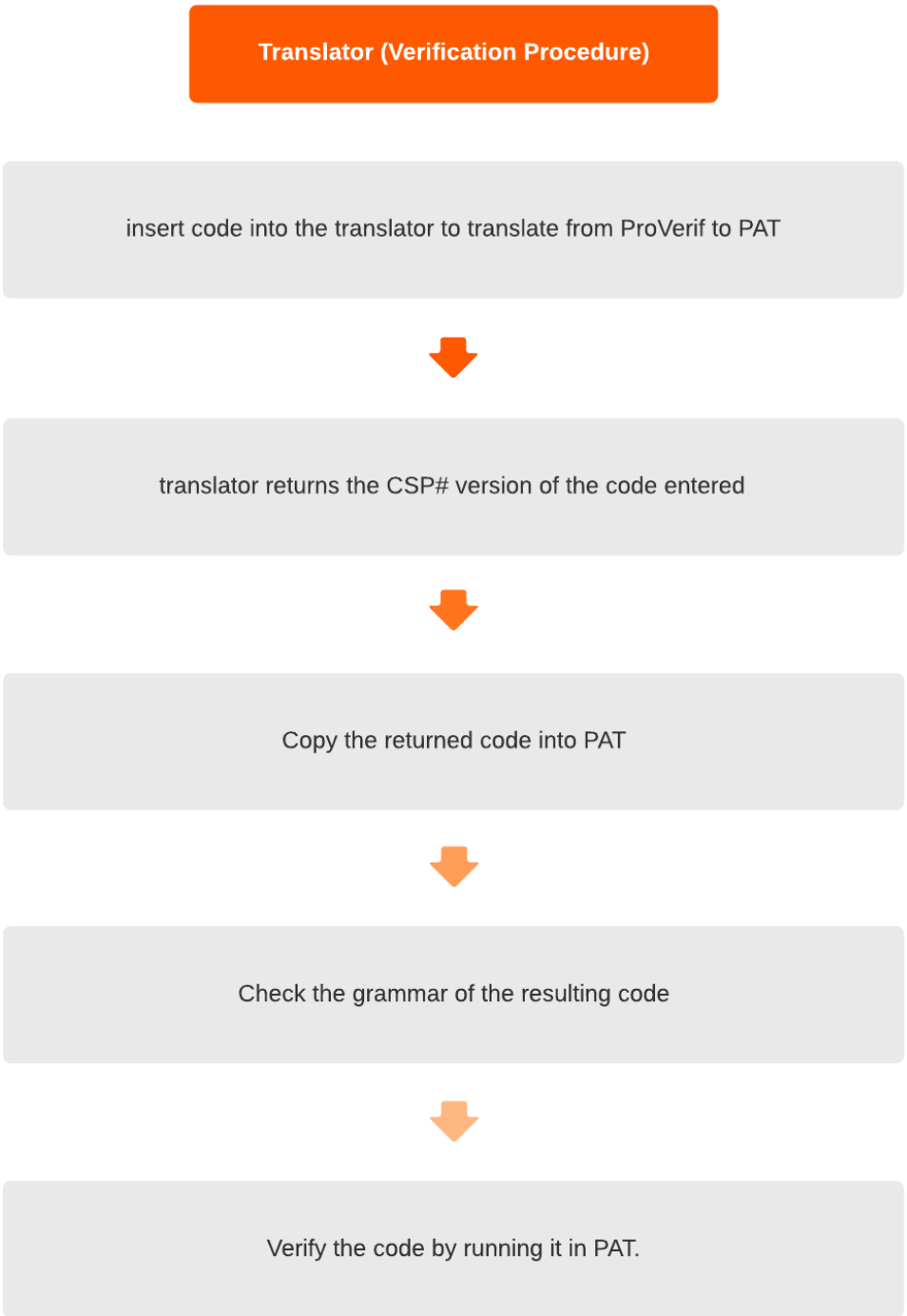

Figure 11 Verification Procedure.

Except for the code verification, every procedure has already been addressed; now, let us look at the outcome of the methodology used.

********Verification Result********
The Assertion (Protocol() |= [](([]! iniCommitAB)||(! iniCommitAB U resbeginparamAB))) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:105
Total Transitions:230
Time Used:0.0063249s
Estimated Memory Used:43917.464KB

********Verification Result********
The Assertion (Protocol() |= [](([]! resCommitAB)||(! resCommitAB U inibeginparamAB))) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:62
Total Transitions:94
Time Used:0.0084657s
Estimated Memory Used:42823.488KB

Figure 10 Result acquired by translation.

********Verification Result********
The Assertion (Protocol() |= [](([]! iniCommitAB)||(! iniCommitAB U resRunningAB))) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:3159
Total Transitions:8418
Time Used:0.1082428s
Estimated Memory Used:45499.32KB

********Verification Result********
The Assertion (Protocol() |= [](([]! resCommitAB)||(! resCommitAB U iniRunningAB))) is **VALID**.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:3349
Total Transitions:9328
Time Used:0.1269338s
Estimated Memory Used:43229.784KB

Figure 10 Result of the actual Needham Schroeder protocol in PAT

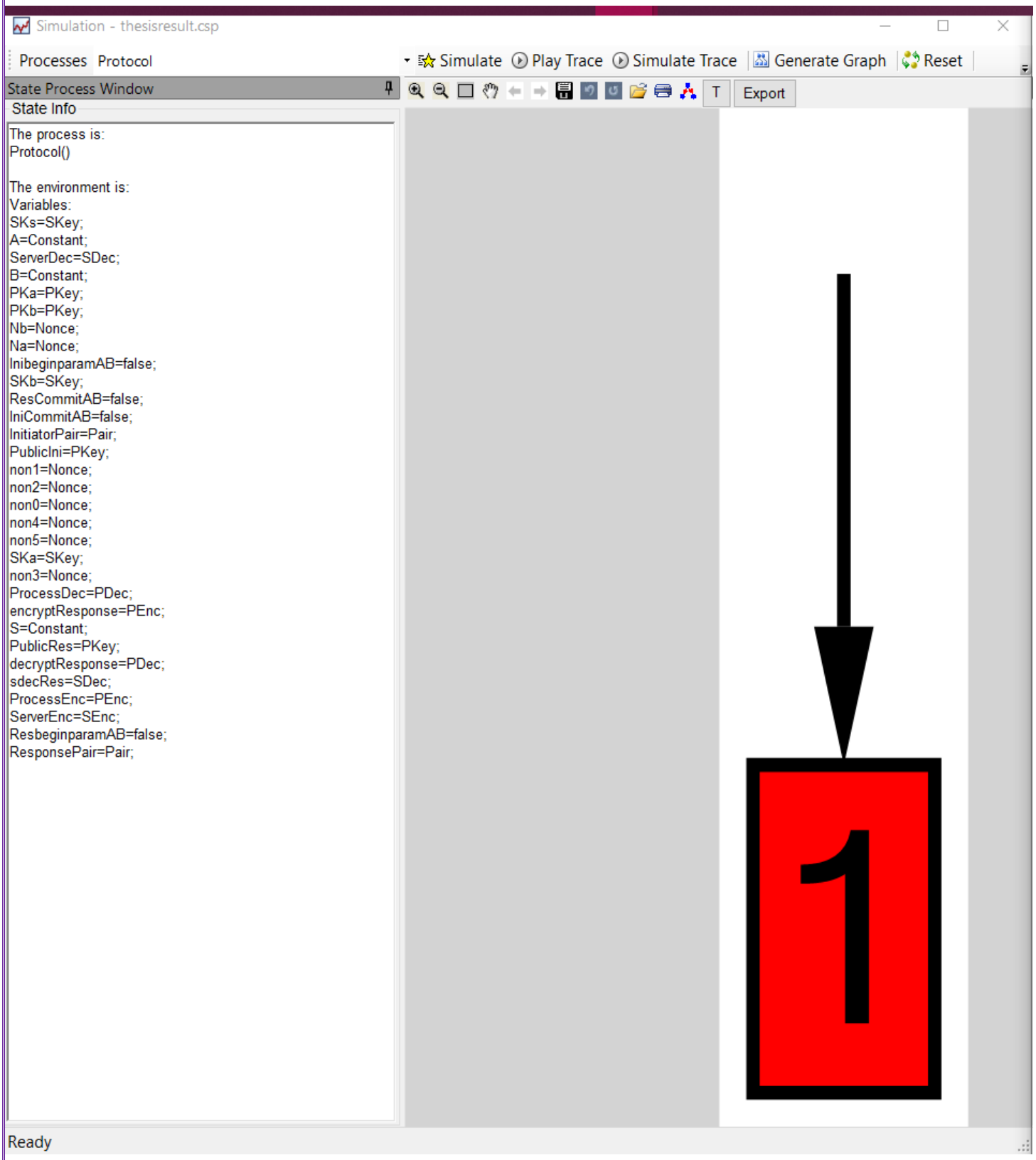

Figure 12 Simulation of the translated version

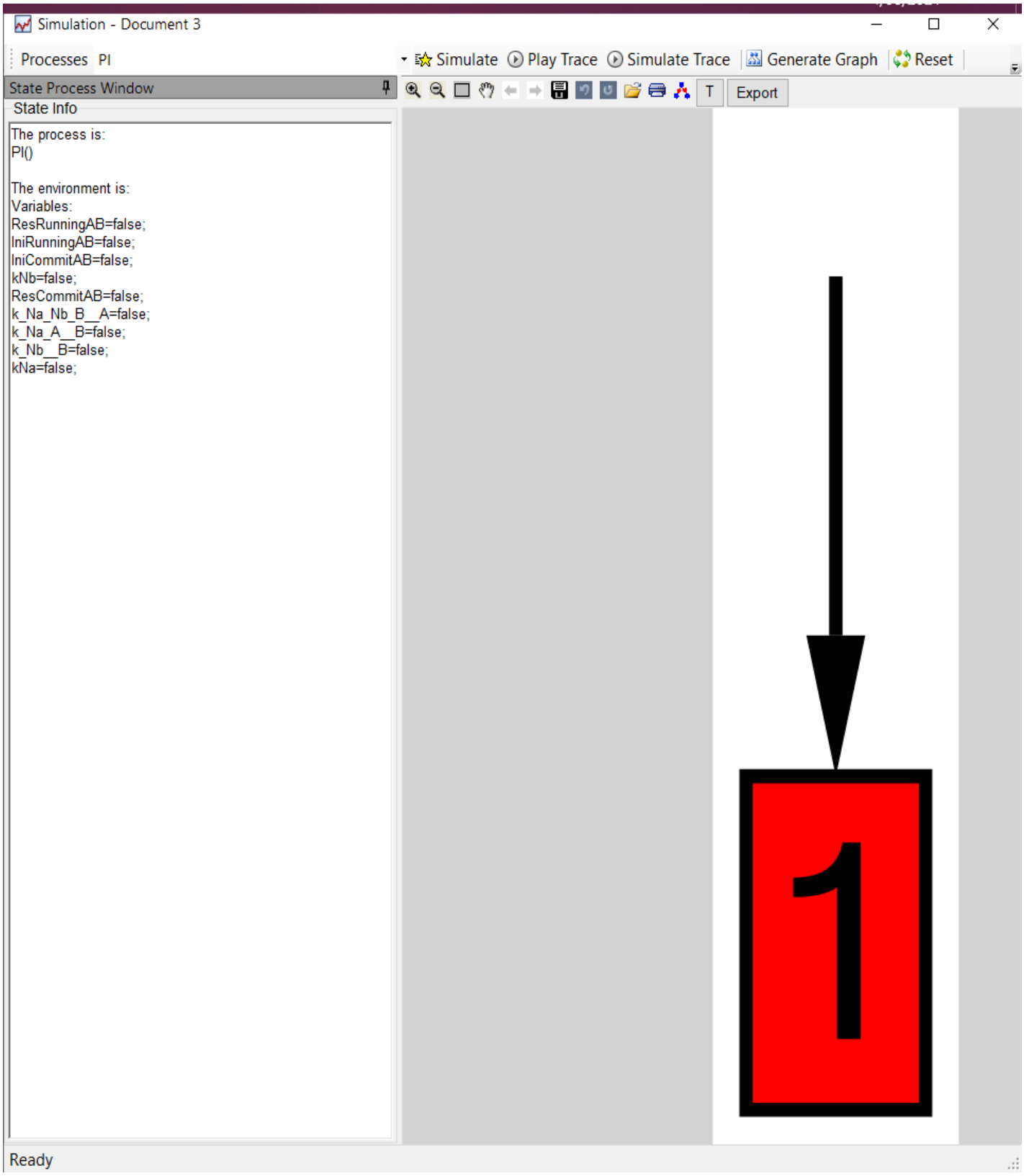

Figure 13 Simulation of the Needham Schroeder example in PAT

Discussion based on the results acquired

This protocol determines if the initiator A commits to a session with B only if B has participated in a protocol run with A and its converse.

The verification shows that there is a discrepancy in the number of states visited and the overall transition in the code that was contained in PAT and the translated version from ProVerif. Because the translated version visits fewer states, it is projected to use less time and memory than the original version in PAT. One cannot tell whether the translated protocol is valid based just on the verification findings.

So, Figure 12 (which represents the simulation of the translated version of the Needham Schroeder protocol) and Figure 13 (represents an example of the Needham Schroeder protocol in PAT) show that both protocols are identical, implying that the translation is allegedly accurate for this protocol.

It is also clear from the environments of both figures that the same code was not utilised to execute the following protocol.

There are several things that may be done to enhance the existing outcome, which are addressed in the next section.

Conclusion and Future Work

This study is primarily concerned with two tools (ProVerif and PAT) and the languages used to verify protocols in those tools. The use of these tools for protocol verification would boost trust in the specific protocols. Both tools inspect the model for various features.

The goal of this project was to bridge the gap between these two languages so that models may be validated in both tools. As a result, a translator was developed to convert applied pi-calculus to CSP sharp, allowing models defined in one language to be converted into another for verification.

The translator is written in C, and it is written by taking into account the application of each function, variable, and term, as well as their grammatical rules, and then translating them properly. It was evident from the results obtained that the translation was successful for the Needham Schroeder protocol but there are several tasks that could be completed to improve the current translator and make it more user friendly, such as: -

- As the existing translator does not follow all grammar rules, it may be enhanced by following all grammar rules so that anything from one language may be translated into another.

- Another item that can be improved in the present translator is the creation of a GUI for the existing translator so that the user can easily translate a file.

- Many of the current problems in the translator might be solved if libraries with functionality could be written in PAT. As a result, libraries for specific functions might be supplied to ease the migrations.

- Another essential item that might be built from this study is a system that can create the combined verification result of a protocol in both PAT and ProVerif by only putting the protocol into the system in one language. This would greatly aid in the improvement of the security protocols in the future.

References

[1] (2021). Retrieved 26 May 2021, from https://prosecco.gforge.inria.fr/personal/bblanche/proverif/

[2] Paiola, M., & Blanchet, B. (2015). *From the Applied Pi Calculus to Horn Clauses for Protocols with Lists* (Doctoral dissertation, Inria).

[3] Sun, J., Liu, Y., Dong, J. S., & Chen, C. (2009, July). Integrating specification and programs for system modeling and verification. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering* (pp. 127-135). IEEE.

[4] Ryan, M. D., & Smyth, B. (2011). Applied pi calculus. In *Formal Models and Techniques for Analyzing Security Protocols* (pp. 112-142). Ios Press.

[5] Bruno Blanchet. Automatic verification of correspondences for security protocols. Journal of Computer Security, 17(4):363–434, July 2009.

[6] Man, K. L., Krilavicius, T., & Leung, H. L. (2009, November). Case studies with process analysis toolkit (PAT). In *2009 International SoC Design Conference (ISOCC)* (pp. 133-136). IEEE.

[7] Roscoe, A.W.: Model-checking CSP, pp. 353–378 (1994)

[8] Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 1020 Dining Philosophers for Deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)

[9] Sun, J., Liu, Y., Dong, J. S., & Chen, C. (2009, July). Integrating specification and programs for system modeling and verification. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering* (pp. 127-135). IEEE.

[10] Process Analysis Toolkit (PAT) 3.5 User Manual. (2021). Retrieved 31 May 2021, from https://pat.comp.nus.edu.sg/wpsource/resources/OnlineHelp/htm/index.htm

[11] Blanchet, B. (2016). Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, *1*(1-2), 1-135.

[12] Parr, T. J. (1995). Language translation using PCCTS and C++(a reference guide). *Parr Research Corporation*.

[13] (2021). Retrieved 2 June 2021, from https://github.com/naipengdong/PAT_lib/blob/master/Lib_v0%2Bequal%2Bknow.cs
[14] Ling Shi, Yan Liu. "Modeling and Verification of Transmission Protocols: A Case Study on CSMA/CD Protocol" , 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion, 2010
[15] K. L. Man, T. Krilavicius and H. L. Leung, "Case studies with Process Analysis Toolkit (PAT)," 2009 International SoC Design Conference (ISOCC), 2009, pp. 133-136, doi: 10.1109/SOCDC.2009.5423889.

Appendix A

$$
\begin{aligned}
\langle term\rangle ::= & \ \langle ident\rangle \\
| & \ \langle nat\rangle \\
| & \ \texttt{(}\text{seq}\langle term\rangle\texttt{)} \\
| & \ \langle ident\rangle\texttt{(}\text{seq}\langle term\rangle\texttt{)} \\
| & \ \langle term\rangle\ (\texttt{+} \mid \texttt{-})\ \langle nat\rangle \\
| & \ \langle nat\rangle\ \texttt{+}\ \langle term\rangle \\
| & \ \langle term\rangle\ \langle infix\rangle\ \langle term\rangle \\
| & \ \texttt{not (}\langle term\rangle\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
\langle pattern\rangle ::= & \ \langle ident\rangle[\texttt{:}\ \langle typeid\rangle] \\
| & \ \langle nat\rangle \\
| & \ \langle pattern\rangle\ \texttt{+}\ \langle nat\rangle \\
| & \ \langle nat\rangle\ \texttt{+}\ \langle pattern\rangle \\
| & \ \texttt{(}\text{seq}\langle pattern\rangle\texttt{)} \\
| & \ \langle ident\rangle\texttt{(}\text{seq}\langle pattern\rangle\texttt{)} \\
| & \ \texttt{=}\langle pterm\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle mayfailterm\rangle ::= & \ \langle term\rangle \\
| & \ \texttt{fail}
\end{aligned}
$$

$$
\langle typedecl\rangle ::= \text{seq}^{+}\langle ident\rangle\texttt{:}\ \langle typeid\rangle[\texttt{,}\langle typedecl\rangle]
$$

$$
\langle failtypedecl\rangle ::= \text{seq}^{+}\langle ident\rangle\texttt{:}\ \langle typeid\rangle[\texttt{or fail}][\texttt{,}\langle failtypedecl\rangle]
$$

$$
\begin{aligned}
\langle decl\rangle ::= & \ \texttt{type}\ \langle ident\rangle\ \langle options\rangle\texttt{.} \\
| & \ \texttt{channel}\ \text{seq}^{+}\langle ident\rangle\texttt{.} \\
| & \ \texttt{free}\ \text{seq}^{+}\langle ident\rangle\texttt{:}\ \langle typeid\rangle\ \langle options\rangle\texttt{.} \\
| & \ \texttt{const}\ \text{seq}^{+}\langle ident\rangle\texttt{:}\ \langle typeid\rangle\ \langle options\rangle\texttt{.} \\
| & \ \texttt{fun}\ \langle ident\rangle\texttt{(}\text{seq}\langle typeid\rangle\texttt{):}\ \langle typeid\rangle\ \langle options\rangle\texttt{.} \\
| & \ \texttt{letfun}\ \langle ident\rangle[\texttt{(}[\langle typedecl\rangle]\texttt{)}]\ \texttt{=}\ \langle pterm\rangle\texttt{.}
\end{aligned}
$$

⟨*query*⟩ ::= ⟨*gterm*⟩ [**public_vars** seq$^+$⟨*ident*⟩] [; ⟨*query*⟩]
| **secret** ⟨*ident*⟩ [**public_vars** seq$^+$⟨*ident*⟩] ⟨*options*⟩ [; ⟨*query*⟩]
| **putbegin event:**seq$^+$⟨*ident*⟩ [; ⟨*query*⟩]
| **putbegin inj-event:**seq$^+$⟨*ident*⟩ [; ⟨*query*⟩]

H ::=	hypothesis
F	fact
M=N	equality
M<>N	disequality
M>N	greater
M<N	smaller
M>=N	greater or equal
M<=N	smaller or equal
is_nat (M)	M is a natural number
H && H	conjunction
H || H	disjunction
false	constant false
F ==> H	nested correspondence

M, N ::=	term
x, a, c	variable, free name, or constant
$0, 1, \ldots$	natural numbers
$f(M_1, \ldots, M_n)$	constructor application
$(M_1, \ldots, M_n)$	tuple
$M + i$	addition, $i \in \mathbb{N}$
$i + M$	addition, $i \in \mathbb{N}$
$M - i$	subtraction, $i \in \mathbb{N}$

⟨*process*⟩ ::= 0
| **yield**
| ⟨*ident*⟩[(seq⟨*pterm*⟩)]
| (⟨*process*⟩)
| ⟨*process*⟩ | ⟨*process*⟩
| !⟨*process*⟩
| ! ⟨*ident*⟩ <= ⟨*ident*⟩ ⟨*process*⟩
| **foreach** ⟨*ident*⟩ <= ⟨*ident*⟩ **do** ⟨*process*⟩
| **new** ⟨*ident*⟩[[seq⟨*ident*⟩]]: ⟨*typeid*⟩ [; ⟨*process*⟩]
| ⟨*ident*⟩ <-R ⟨*typeid*⟩ [; ⟨*process*⟩]
| **if** ⟨*pterm*⟩ **then** ⟨*process*⟩ [**else** ⟨*process*⟩]
| **in**(⟨*pterm*⟩,⟨*pattern*⟩) ⟨*options*⟩ [; ⟨*process*⟩]
| **out**(⟨*pterm*⟩,⟨*pterm*⟩)) [; ⟨*process*⟩]
| **let** ⟨*pattern*⟩ = ⟨*pterm*⟩ [**in** ⟨*process*⟩ [**else** ⟨*process*⟩]]
| ⟨*ident*⟩[:⟨*typeid*⟩] <- ⟨*pterm*⟩ [;⟨*process*⟩]

Appendix B

```
specification
        : (specBody)*
        ;

specBody
        : library
        | letDefintion
        | definition
        | assertion
        | alphabet
        | define
        | channel
        ;

library
        : '#' 'import' STRING ';' //import the library by full dll path or DLL name under the Lib folder
        | '#' 'include' STRING ';' //include the other models by full path or file name if under the same folder of current model
        ;

channel
        : 'channel' ID ('[' additiveExpression ']')? additiveExpression ';'
        ;

assertion
        : '#' 'assert' definitionRef
        (
          ( '|=' ( '(' | ')' | '[]' | '<>' | ID | STRING | '!' | '?' | '&&' | '||' | 'xor' | '->' | '<->' | '/\\' | '\\/' | '.' | INT )+ )
                | 'deadlockfree'
                | 'nonterminating'
                | 'divergencefree'
                | 'deterministic'
                | 'reaches' ID withClause?
                | 'refines' definitionRef
                | 'refines' '<F>' definitionRef
                | 'refines' '<FD>' definitionRef
          )
        '.'
        ;

withClause
        : 'with' ('min' | 'max') '(' expression ')'
        ;

definitionRef
        : ID ('(' (argumentExpression(',' argumentExpression)*)? ')')?
        ;
```

```
alphabet
        : '#' 'alphabet' ID '{' eventList (',' eventList )* '}' ';'
        ;

define
        : '#' 'define' ID  '-'? INT ';'
        | '#' 'define' ID  ('true' ';'
                           | 'false' ';')
        | 'enum' '{' a=ID(',' b=ID)* '}'  ';'
        | '#' 'define' ID dparameter? dstatement ';'
        ;

dparameter
        : '(' ID (',' ID )* ')'
        ;

dstatement
        : block
        | expression
        ;

block
        : '{' (s=statement)* (e=expression)? '}' //At least a statement or expression has to be specified, i.e. s and e cannot be both null.
        ;

statement
        : block
        | localVariableDeclaration
        | ifExpression
        | whileExpression
        | expression ';'
        | ';'
        ;

//local variable that can be used in the block
localVariableDeclaration
        : 'var' ID ('=' expression)? ';'
        | 'var' ID '=' recordExpression ';'
        | 'var' ID ('[' expression ']')+ ('=' recordExpression)? ';'
        ;

expression
        : conditionalOrExpression ('=' expression)?
        ;

conditionalOrExpression
        : '||' indexedExpression
        | conditionalAndExpression ( '||' conditionalAndExpression )*
        ;
```

```
conditionalXorExpression
        : 'xor' indexedExpression
        | bitwiseLogicExpression ( 'xor' bitwiseLogicExpression)*
        ;

indexedExpression
        : (paralDef (';' paralDef)*) '@' expression
        ;

bitwiseLogicExpression
        : equalityExpression ( ( '&' | '|' | '^' ) equalityExpression)*
        ;

equalityExpression
        : relationalExpression ( ('=='|'!=') relationalExpression)*
        ;

relationalExpression
        : additiveExpression ( ('<' | '>' | '<=' | '>=') additiveExpression)*
        ;

additiveExpression
        : multiplicativeExpression ( ('+' | '-') multiplicativeExpression)*
        ;
multiplicativeExpression
        : unaryExpression ( ('*' | '/' | '%' ) unaryExpression)*
        ;

letDefintion
    : ('var'|'hvar') ('<' userType=ID '>')? name=ID varaibleRange? ('=' (expression|'*') )? ';' //user defined datatype is supported using <type>
    | ('var'|'hvar') ID variableRange? '=' recordExpression ';'
    | ('var'|'hvar') ID ('[' expression ']')+ variableRange? ('=' (recordExpression|'*') )? ';' //multi-dimensional array is supported
    ;

varaibleRange
    : ':' '{' (additiveExpression)? '..' (additiveExpression)? '}'
    ;

argumentExpression
    : conditionalOrExpression
    | recordExpression
    ;

//if definition
ifExpression
    : 'if' '(' expression ')' statement ('else' statement)?
    ;

whileExpression
    : 'while' '(' expression ')' statement
    ;

recordExpression
    : '[' recordElement (',' recordElement)* ']'
    ;

recordElement
    : e1=expression ('(' e2=expression ')')? //e2 means the number of e1, by default it's 1
    | e1=expression '..' e2=expression //e1 to e2 gives a range of constants
    ;

//process definitions
definition
        : ID ('(' (parameter(',' parameter)*)? ')')? '=' interleaveExpr ';'
        ;

parameter
        : ID varaibleRange?
        ;
```

```
X ::=                                          (Variable)
        Identifier

D ::=                                          (Declaration)
        val P = E                              (value declaration)
      | def X( P , ... , P ) = E               (function declaration)

P ::=                                          (Pattern)
        X                                      (variable)
      | C                                      (constant)
      | _                                      (wildcard)
      | ( P , ... , P )                        (tuple pattern)
      | [ P , ... , P ]                        (list pattern)
      | P : P                                  (cons pattern)

Define:=                                       (Condition Definition)
        #define X (E)

Assert:=                                       (Assertion)
        #assert System
        (deadlockfree                          (deadlock-freeness check)
        | reaches X                            (reachability check)
        | |= LTL                               (ltl check)
        | nrsdeadlockfree                      (nrsdeadlock-freeness check)
        | nrscyclefree)                        (nrscycle-freeness check)

LTL:=                                          (LTL Expression)
        "C"                                    (publish value)
      | X                                      (predefined condition)
      |[] LTL                                  (always)
      |<> LTL                                  (eventually)
      |X LTL                                   (next)
      |LTL U LTL                               (until)
      |LTL R LTL                               (release)
```

Program ::= '#' 'Variables'

Variables declare

'#' 'Initial'

Initial declare

'#' 'Protocol description'

Protocol declare

'#' 'System'

System declare

('#' 'Intruder'

Intruder declare)?

('#' 'Verification'

Verification declare)?