Transformations Between CSP# and C#

ZHU HUIQUAN

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

ACKNOWLEDGEMENTS

First and foremost, I feel deeply grateful to my supervisor, Dr. Dong Jin Song, for his guidance, advice and encouragements on my doctoral program. He has given me different supports on research and career fields.

I am grateful to Dr. Liu Yang, Dr. Sun Jun, Dr. Bimlesh Wadhwa and Dr. Lin Shang-Wei. They gave me a lot of good advice and help on my research works.

I would also like to express my great appreciation to Dr Zheng Manchun and Nguyen Truong Khanh. Thanks for giving me a lot of feedback and information for my research.

My special thanks are extended to the classmates and the staffs of School of Computing, National University of Singapore.

Contents

Summary

Concurrent software system contains multiple processes running in parallel. These processes synchronize with each other to perform collaborative tasks. Due to the complexity of concurrency, it is difficult to ensure the implemented system satisfying the desired concurrent properties. Formal mathematical models have been introduced to model the interaction between different processes in concurrent systems. Communicating Sequential Processes (CSP), as a formal language, models concurrent systems on event and channel communications. The concurrency related properties, represented as Linear Temporal Logic (LTL) formula, can be verified on the CSP model that represents the system.

After validating the properties, a concurrent model is ready to be implemented in the programming language used in target platform. Formal languages usually are in a high level of abstraction and they are quite different to the programming languages used in implementation. It is desirable to have a well-defined transformation from the abstract model to the low-level implementation. The transformation shall guarantee the implementation preserve the properties that have been verified on the formal model. On the other hand, there are situations when the systems are implemented without formal design documents. Or during maintenances, the program has become inconsistent to the original design documents. In these cases, the reverse transformation from the implemented program to formal model helps to verify the concurrent properties on the implemented program.

This thesis discusses the transformations and verification on CSP# models and multi-threaded C# programs. CSP# extends CSP to support shared variables and event-attached programs. These program-friendly features in CSP# enable the transformations to use flexible boundaries between formal models and the user-defined programs that are imported to the model.

Our first approach translates C# source code to CSP# models. The C# program's class inheritance relations and its fields are preserved as user-defined data structures. CSP# model imports these user-defined data structures as shared variables. The communications between threads are captured and represented as event and channel synchronizations in CSP#. For the features that are not supported in CSP#, such as thread creation, they are translated to processes based on their behaviors in the program.

The second approach performs Virtual Machine based verification on C# programs. We add a “modelchecking” mode in the Mono virtual machine. When running in this mode, it takes the multi-threaded C# program as a LTS system and communicates with PAT framework to traverse its state space. The tool allows different transition atomicity levels, such as IL (Intermediate Language) level and source code level. The tool does not change the programs’ assemblies and each transition is executed as its original behaviors on virtual machine. Deadlock-freeness and safety properties defined on the program data can be verified by our VM-based verification tool.

The synchronization between threads in C# is based on shared memory communication, which is different from the event and channel synchronization in CSP#. Our third approach first implemented the CSP# operators in a C# library “PAT.Runtime”. The event synchronization is based on the “Monitor” class in C#. The precondition layer and choice layer are added above the CSP event synchronization to support CSP# specific features. We also developed a code generation tool in PAT framework to transform CSP# models to multi-threaded C# programs, which use the CSP# operators in “PAT.Runtime” to communicate between threads. We proved that the generated C# program and original CSP# model are equivalent on the trace semantics. This equivalence guarantees the validated properties of the CSP# models preserve in the generated C# programs. Additionally, based on the existing implementation of choice operator, we redesign the synchronization mechanism to remove the unnecessary communications among these choice operators. The experiment results show the improved mechanism notably outperforms the JCSP library and our first version of “PAT.Runtime” library.

Key words: **Formal Verification, Model Checking, Concurrent Systems, CSP#, C#, Program Verification, Multi-threaded, Monitor**

List of Figures

List of Tables

Chapter 1

Introduction

Developers spend many efforts to ensure the correctness of concurrent systems. In design phase, concurrent systems are abstracted as formal models to describe the interaction between different components. The requirements of the system are represented as properties of these models, then the *model checker* can verify whether the models satisfy the properties. After verification, the validated models are implemented on the target platform with specific programming languages. The communications in the concurrent models are usually implemented using the programming languages' built-in concurrency mechanisms or other third-party concurrency libraries. In the implementation phase, one of the main concerns is to guarantee that the implemented program is consistent to the original design model, and the validated properties are preserved in the implemented program. In this thesis, we try to provide better transformations between the concurrent design models and the implemented programs.

1.1 Concurrent System

A concurrent software system contains multiple computational processes running in parallel. Each process performs a number of operations sequentially and they communicate with each other to collaborate on complex tasks. The design models of the concurrent systems usually describe how the processes communicate with each other. These communications restrict the behaviors of the processes in the system so that they do not perform the tasks in undesirable operation sequences.

The requirements on the system's concurrency are represented as the *properties* of the system. These properties define what kinds of operation sequences are allowed (or not allowed). Before implementation, the properties are verified on the design model manually or automatically.

Formal mathematical models, such as Process Calculus[4], Petri Nets[85] and Actor model [1], have been developed to describe the concurrent aspect of the systems. Developers can use the formal models as the abstractions of the system states and the communications between the processes. *Model checkers* can verify whether the system model satisfies the properties [5] by traversing the state space of the model.

CSP (Communicating Sequential Processes) [46], as a member of Process Calculus, is one of the popular formal languages to model concurrent systems since the 70s. It has been applied to practical projects including the embedded systems, protocols and concurrent systems [12, 97, 57, 75]. CSP# [102] is based on classic CSP. It extends CSP with programming features such as shared variables and event-attached program etc. In CSP#, the concurrent system is modeled as several processes communicating with each other via *events* and *channels*. The *trace* of a process is the finite sequence of the event and channel operations that the process has engaged. The system's concurrent behavior is represented as the possible traces that all the processes in the system can engage. PAT (Process Analysis Toolkit) [71], as a model checker, can validate whether the concurrent properties are satisfied on the CSP# model.

After the CSP# models are verified, they will be implemented in specific programming languages on the target platforms. The formal languages are quite different to the programming languages used in implementation. For example, the message passing communications in the process calculus are different from the shared memory communications in the programming languages like C# and Java. On the other hand, the object-related features in C# and Java, such as polymorphism and automatic garbage collecting, are not intuitively supported in the modeling languages such as CSP and CSP#. The concurrent models are usually in a high level of abstraction and they do not include the low level detail of the system. In implementation phase, the additional codes are added to the program to implement these low level functionalities. Well-defined transformation is needed from CSP# models to the implementations in object-related languages. The transformation needs to ensure the implementations preserve the verified properties of the CSP# models.

There are situations where we want to verify the concurrent properties on the implemented programs instead of the models. For example, sometimes the programs are implemented without detailed designs, or the design documents are inconsistent with the programs in maintenance. In these situations, one possible solution is to translate the programs back to abstract models and check the concurrent properties on the models. Another possible solution is to check the concurrent properties directly on the implemented programs, with customized Virtual Machine (VM) that traverses the program's state space on virtual machine.

We study both directions mentioned above, including transforming from models to programs and from programs to models. For the languages, we focus on the transformation between CSP# models and multi-threaded C# programs. As mentioned above, CSP# has extended classic CSP with procedural program features and it can import user-defined class libraries into the models. These program-friendly features allow CSP# to effectively model programs in object-oriented languages.

C# is a general-purpose, object-oriented programming language and it has been widely used in industries. Its specification has been standardized in 2003 and updated in 2005 by ECMA International [3]. C# language also has good built-in concurrency control, as the language features or as libraries. Choosing C# as the programming language shall make this thesis more readable for the industrial and academic readers. Although the approaches of the thesis are based on CSP# and C#, the methodology and technique are general for process calculus and object-oriented languages. The result of this thesis shall be applicable to other object-oriented programming languages and CSP-like formal languages.

1.2 Model and Program

Usually, a formal concurrent model contains not only the software system but also the user activities and the environment. The software system model will later be implemented in the software system. The user activities and the environment models help to create the simulation configuration and testing cases for the software system, but they are not included in the software system. In this thesis, the transformations are focusing on the software system model and its corresponding implementation program.

For better understanding of the transformations between the CSP# model of the concurrent system and its corresponding implementation in C#, we compare the abstract model and the implemented program to discuss what have been abstract away in the model and how to manage them in the implemented program.

The operational semantics of CSP# model can be expressed using a *Labeled Transition System* (LTS) [102]. A LTS is a tuple $(S, \Sigma, \rightarrow, s_0)$. It contains a set of states S and a set of labels Σ. The transition is a relation from one state to another, with a label associated with the transition, i.e. $\rightarrow \subseteq S \times \Sigma \times S$. We use $s \xrightarrow{\alpha} s'$ to represent a transition $(s, \alpha, s') \in \rightarrow$. A LTS system has an initial state s_0 and it may have finite or infinite states. A CSP# model, represented as a LTS system, is shown as (a) in Figure 1.1. Each edge represents a label transition and each vertex represents a state in LTS.

In a multi-threaded C# program, the execution of each statement in each thread changes the program state. For a specific thread, the method that it is executing can be represented as a flowchart. In the flowchart, each process step (in rectangle) represents executing a statement. The flowchart of a two-thread C# program is shown in (b) in Figure 1.1. For a single processor computer, the operating system sequentially executes the statements in "run()" methods of the two threads. When "T1" runs to the statement "send" and communicates with thread "T2", it may be blocked on the statement "send" if "T2" has not run to the statement "receive". Here the "send" and "receive" are the synchronized communication between threads. They do not return until the communication succeeds. If "T1" and "T2" have reached the "send" and "receive" blocks, they both can successfully

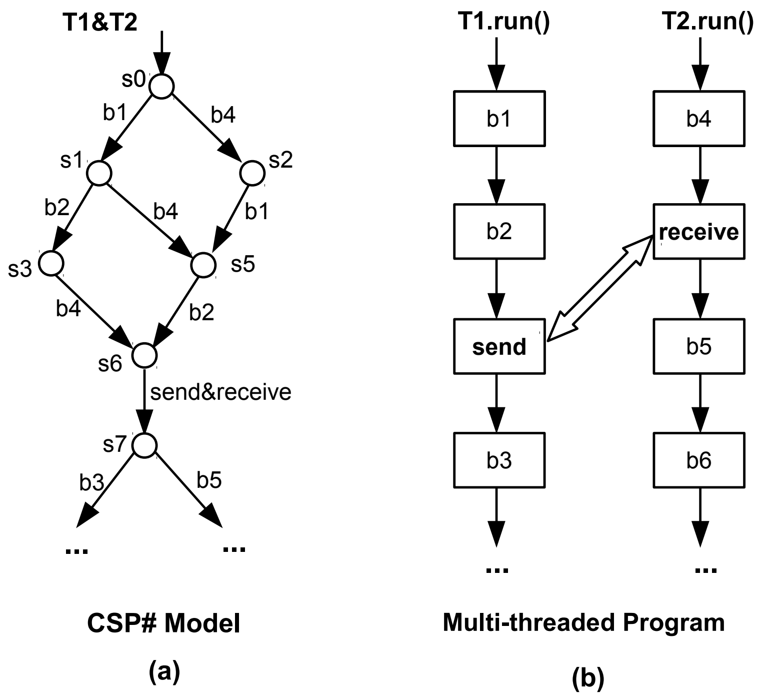

Figure 1.1: CSP# Model and C# Program

finish the communication and execute the following statements.

The LTS of (a) in Figure 1.1 is actually the CSP# model of the two-thread program in (b). The communication of "send" and "receive" from "T1" and "T2" are explicitly modeled in the CSP# model as a single event "send&receive". Other statements are also modeled as individual events in (a). These events may occur in multiple transitions in the LTS, starting from different program states. For example, there are three transitions labeled as the event "b4". As events in models are considered as instantaneous, the time of the operating system executing these statements are ignored.

The duration of the system staying in a state of LTS is ignored too. Suppose there is one statement "b7" between "b1" and "b2", and the CSP# model abstracts away this statement. The reason may be that "b7" is a simple output statement or it does not change the program state. As a result, the time that the system executes this statement may be happening in state "s1" or "s5". When the program is executing "b7", the event "b2" is not enabled although there are transitions labeled "b2" started from "s1" and "s5". From the analysis, we can know that the program can stay at any state in the LTS for a time interval. The length of this interval is decided by the operating system or environment. The next enabled events in the CSP# model may not be enabled until the end of this interval.

The CSP# model takes the scheduling as being non-deterministic. When certain properties are verified on the model, the algorithm of the model checker may take the *fairness* assumptions on the scheduling. For the multi-threaded C# program, it is the operating system that controls the scheduling. The fairness in the model shall be implemented in the communications between threads.

To implement a CSP# model, the developers need to keep in mind about these differences between the model and the program. The functionality codes that are not represented in the model shall be added to the program carefully. They shall not change the concurrent aspect of the program. Otherwise, the validated properties on the CSP# model may not preserve in the program.

1.3 Research Goals

In the previous section, we discussed the difference between the CSP# model and its implemented C# program. They shall be consistent on the concurrent behaviors that are defined in the CSP# model but there are always more functional behaviors in the program. When we want to verify the concurrent properties on the implemented program, stress testing [80, 105] cannot ensure the properties held on all situation. The transformation from C# program to CSP# model can exhaustively verify the properties. In the transformation, we shall focus on the concurrent behaviors and control the atomicity of the functional behaviors in resulted models. The other direction, to implement the CSP# model, we need to clearly define how the concurrent behaviors of the CSP# model are represented in the program. Based on the equivalence on the model and the program, the properties of the model can be preserved in the program. Additionally, we also need to ensure the functional codes do not break the verified properties.

It is not easy to achieve above goals. The differences between the modeling language CSP# and programming language C# need to be investigated carefully. The semantics of CSP# needs to be projected to the C# program, as well as the properties. Making good use the language features of CSP# helps the transformation on both directions, from the CSP# models to C# programs and reversely from programs to model. Balanced boundary between the CSP# model and the C# program not only increase the efficiency of the transformations but also make both the model and program more readable and easier to use in practice.

One way to verify the C# program is to translate the source code to a CSP# model. To balance the complexity of the CSP# model, the translation shall emphasize on the inter-thread communications, using finer atomicity on them. Other operations in the C# program can allow more flexible atomicity control. The properties are translated to formula and verified on the translated model. The translation can be done manually or with the help of automatic tools.

Execute the C# program and examine the program state on virtual machine is another way to verify the properties on implemented program. To traverse all the state space of the program, the virtual machine needs to take over the thread or process scheduling of the operating system. It also needs to take the snapshots of the program states at each transition.

Using appropriate abstractions in two approaches above can increase the efficiencies of the verifications. Depending on the application domain, the program may be divided into several layers or components. Not every layer or component needs to be exhaustively traversed in the model. The model checking algorithm may choose to completely traverse the state space on some components and filter out unnecessary traverses on the other components. To control the atomicity of the transition is another way to abstract the programs. Using larger transition atomicity can significantly reduce the state space of the programs. The algorithm can allow different types of atomicity on different segments in the program.

To implement a C# program designed with CSP# model, developers can use the built-in thread synchronization mechanism, which is based on share memory communication. This needs extra efforts to represent the CSP# semantics in the built-in synchronization. Developers also need to prove the implemented program is consistent with the design model, which is not easy and sometimes error-prone.

The changes on the software design may happen from time to time, even after implementation. The iterations on the design and implementation phases are sometimes unavoidable in practice. A closer relation between the model and the implemented program helps to improve the development efficiency. An automatic code generation tool to transform CSP# model to its implementation enables developers to test the changed model right after it is verified. We would try to provide better supporting facilities from CSP# model to the implemented C# program.

We summarize the research contributions of the thesis as follows.

- To enable the concurrent properties to be verified on the implemented C# program, we introduce appropriate transformation from C# program to CSP# model. The first approach translates the C# programs to CSP# models. The program data are stored in the shared variables and the methods in C# are translated to processes in CSP#.

- The C# program can be regarded as a LTS system at runtime. Our second approach verifies the properties of the C# program with PAT and a customized virtual machine (VM). The VM does not to modify the behavior of the C# program and traverse the state space of it with configurable atomicity.

- Define the CSP# semantics in the C# program. Based on this definition, we discuss the equivalence between the CSP# models and C# programs and the properties of the CSP# models that are preserved in the C# programs.
- Based on the equivalence on CSP# semantics, the CSP# operators can be implemented in a C# class library. We can provide a code generation tool to transform the CSP# models to C# programs with the verified properties preserved.

In Chapter 2 we review the background knowledge about CSP and CSP#. Chapter 3 introduces the existing works related to this thesis. Our translation-based and VM-based verification approaches are described in Chapter 4 and 5 respectively. We discuss the CSP# to C# approach and the proof of equivalence in Chapter 6. An improved implementation of the CSP and CSP# operators is described in Chapter 7. Chapter 3 compares our three approaches to other related works of software model checking. In Chapter 8, we compare our three approaches to other related works of software, summarize the contributions and discuss possible future works.

Chapter 2

Background

2.1 CSP and CSP#

CSP (Communicating Sequential Processes) was introduced by C. A. R. Hoare [48] in 1978. Since then it has evolved and became a popular modeling language. It is suitable for modeling systems or parts of the system where the communication and concurrency are the key concerns [65].

CSP model is composed of a set of sequential processes interacting with each other. The *primitives* in CSP include *events* and *primitive processes*. The events are basic communication units between processes in the system. Each event has a unique name. The set of all the event names in the system is the *alphabet* of the system. An event communication can have one or multiple processes synchronizing on an event in their alphabets. The processes can also communicate via *channels*. A *channel communication* always involves one process performing the *read* operation on the channel while the other process performing the *write* operation on the same channel. The primitive processes include *Skip* and *Stop*. *Skip* represents a successful termination of a process. *Stop* is a process that communicates nothing and it models the *deadlock*. The *process* is defined as the operators that combine event synchronizations, channel operations, primitive processes and subprocesses.

The event can be in simple form or compound form. An event in simple form uses a lower-case letter or word as its name, such as “a”, “b” and “enter”. A compound event has multiple letters or words linked with “.” in its name, such as “a.b.c” and “enter.room1”. The input or output operations to a channel are considered as channel events. A channel event name is composed with the channel name, input or output symbols and the inputted or outputted data. For a channel named “ch”, “ch?x” represents the event that read the data from “ch” and store it in “x”; “ch!3” represents the events that outputs “3” to channel “ch”. Events are considered as atomic and instantaneous. An event is enabled when the processes and its environment agree on this event and are ready to perform it.

A *trace* of a process is a finite sequence of the events’ names that the process has performed. The *traces* of a process is the set of all possible traces that the process can perform. The traces of a process P is denoted as $traces(P)$. For two processes P and Q, we say that Q *refines* P on traces, denoted as $P \sqsubseteq Q$, if the *traces* of process Q is a subset of the one of P, i.e. $traces(Q) \subseteq traces(P)$.

CSP# [102] is a modeling language based on classic CSP while it additionally offers rich operations on global shared variables in the models. It shares the principle ideas as TCOZ [73, 72] that integrates the state specifications of the components with the interact operations between themselves. The communication in CSP# is based on either the shared variables or the message passing communication as in CSP.

Definition 1 *The* process *definition in CSP# is formally defined as follows:*

$$\begin{aligned} P = \ & Stop \mid Skip \mid e \rightarrow P \mid e\{prog\} \rightarrow P \mid ch!x \rightarrow P \\ & \mid ch?x \rightarrow P \mid [b]P \mid if\, b\,\{P\}else\{Q\} \mid P;\ Q \mid P[]Q \\ & \mid P \parallel Q \mid P \mid\mid\mid Q \mid P \bigtriangleup Q \end{aligned}$$

Here P and Q are the processes. Stop and Skip are built-in primitive processes. The e is an event and ch is a channel (synchronous or asynchronous). x is either a simple or a complex expression (i.e. .x.y.z). b is a boolean expression. prog is an optional block of C# program attached on event e.

Let $\checkmark$ denote the special event of the successful termination of a process; τ denotes the invisible *event and αP denotes the alphabet set of process P. Here αP contains all the events in P excluding $\checkmark$. e_τ denotes any event excluding $\checkmark$. The Stop communicates nothing and Skip $= \checkmark \rightarrow Stop$. The* event prefix *$e \rightarrow P$ performs the event e and then performs as P. Likewise, the* data operation *$e\{prog\} \rightarrow P$ first performs the C# code of prog on the global shared variables then the process performs as P. The* channel output *$ch!x \rightarrow P$ evaluates the expression x on global shared variables, if the channel ch is not full, it sends the evaluated x to channel ch and behaves as P after that. Similarly, the* channel input *$ch?x \rightarrow P$ evaluates x and reads the evaluated x from channel ch and performs as P afterwards. The* guarded process *$[b]P$ is blocked until expression b becomes true and it performs as P after that. The* conditional choice *if b $\{P\}$ else $\{Q\}$ (also denoted as "IF" operator) evaluates b first. If its value is true, the process behaves as P; Otherwise, it behaves as Q. The* sequential composition *$P;\ Q$ behaves as P till its termination and behaves as Q. The* general choice *$P[]Q$ can perform as P or Q, depending on whose first visible event is engaged first. If P performs an event first, $P[]Q$ behaves as P afterwards, otherwise it behaves as Q. For the* parallel composition *$P \parallel Q$, P and Q run and synchronize on the events in $\alpha P \cap \alpha Q$ and they communicate through global shared variables and channels too. In* indexed interleaving *$P \ ||| \ Q$, P and Q run independently and only communicate through global shared variables and channels. The* interrupt *$P \bigtriangleup Q$ behaves as P until the first event of Q is engaged, then the process behaves as Q afterwards.*

CSP# supports *process parameters* [1] and *shared variables* used in the process definition. The conditional expressions can contain the process parameters and shared variables. Shared variables in CSP# can be read by conditional expressions and they can be read and written by event-attached programs. Because both the evaluations of expressions and the executions of event-attached programs are atomic in CSP# models, the shared variables in the CSP# model do not suffer the data race problem. The value changes on shared variables represent the shared memory communication in the CSP# models.

CSP# also supports the combinations of shared memory and message passing communications. Besides the general *conditional choice*, the *atomic conditional choice* operator (also denoted as

[1]The process accepts a set of parameters that can be used as read-only variable in the process. "Phil(i)" and "Fork(x)" in the Dining Philosopher Problem in 2.2 are the processes with parameters "i" and "x".

"IFA") in CSP# is defined as *ifa*(b) $\{P\}$ *else* $\{Q\}$. It requires b being *true* and the first event of P being engaged occurring atomically, or b being *false* and the first event of Q being engaged occurring atomically. On the contrary, the process *if*(b) $\{P\}$ *else* $\{Q\}$ can go to the branch $\{P\}$ at the time when b is true, but later when the first event of P engages, b may have become *false*. The *blocking conditional choice* operator *ifb*(b) $\{P\}$ (also denoted as "IFB") blocks the process until b becomes *true*, but it does not require the first event of P to be engaged atomically. It is considered as the complement of the *guarded process*.

Definition 2 *A* Labelled Transition System *(LTS) is a tuple* $L = (S, s_0, Act, T)$ *where:*

- *S is a set of states;*
- *$s_0 \in S$ is an initial state;*
- *Act is the alphabet of the transition labels;*
- *$T \subseteq S \times Act \times S$ is the labeled transition relation.*

For a transition $(s, e, s') \in T$ ($s, s' \in S$, $e \in Act$), we say that the system can reach state s from state s by performing transition e. For simplicity, we use $s \xrightarrow{e} s'$ to denote $(s, e, s') \in T$.

As mentioned in 1.2, the operational semantics of CSP# model can be expressed using a LTS [102].[2] The state of this LTS is the *system configuration* of the model. It is composed with three parts (P, V, C). P is the current process expression, V is the current valuation of the global variables and C is the current valuation of all the channels in the system. A *transition* is represented as $(P, V, C) \xrightarrow{e} (P', V', C')$ which means the model at state (P, V, C) can perform event e and go to state (P', V', C') .

The specific operational semantics for the CSP# model are defined as follows:

$$\frac{}{(Skip, V, C) \xrightarrow{\checkmark} (Stop, V, C)} \; [\, skip \,]$$

[2]In this thesis, we discuss finite-state CSP# models

$$\frac{}{(e \rightarrow P, V, C) \xrightarrow{e} (P, V, C)} \; [\, event \,]$$

$$\frac{}{(e\{prog\} \rightarrow P, V, C) \xrightarrow{e} (P, upd(V, prog), C)} \; [\, prog \,]$$

$$\frac{C(ch) \; is \; not \; full}{(ch!x \rightarrow P, V, C) \xrightarrow{ch!eva(V,x)} (P, V, C') \atop where \; C'(ch) = C(ch) \cup eva(V, x)} \; [\, out \,]$$

$$\frac{C(ch) \; is \; not \; empty}{(ch?x \rightarrow P, V, C) \xrightarrow{ch?C(ch).head} (P, V, C'), \atop C'(ch) = C(ch) \backslash C(ch).head} \; [\, in \,]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C'), \;\; V \models b}{([b]P, V, C) \xrightarrow{e} (P', V', C')} \; [\, guard \,]$$

$$\frac{V \models b}{(if \; b\{P\}else\{Q\}, V, C) \xrightarrow{\tau} (P, V, C)} \; [\, cond1 \,]$$

$$\frac{V \not\models b}{(if \; b\{P\}else\{Q\}, V, C) \xrightarrow{\tau} (Q, V, C)} \; [\, cond2 \,]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C')}{(P; \; Q, V, C) \xrightarrow{e} (P'; \; Q, V', C')} \; [\, seq1 \,]$$

$$\frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C')}{(P; \; Q, V, C) \xrightarrow{\checkmark} (Q, V', C')} \; [\, seq2 \,]$$

$$\frac{(P, V, C) \xrightarrow{e_\tau} (P', V', C')}{(P[]Q, V, C) \xrightarrow{e_\tau} (P', V', C')} \; [\, ch1 \,]$$

$$\frac{(Q,V,C) \xrightarrow{e_\tau} (Q',V',C')}{(P[]Q,V,C) \xrightarrow{e_\tau} (Q',V',C')}\ [\ ch2\ ]$$

$$\frac{(P,V,C) \xrightarrow{e} (P',V',C'),\ e \notin \alpha Q}{(P \parallel Q,V,C) \xrightarrow{e} (P' \parallel Q,V',C')}\ [\ par1\ ]$$

$$\frac{(Q,V,C) \xrightarrow{e} (Q',V',C'),\ e \notin \alpha P}{(P \parallel Q,V,C) \xrightarrow{e} (P \parallel Q',V',C')}\ [\ par2\ ]$$

$$\frac{(P,V,C) \xrightarrow{e} (P',V',C'),\ (Q,V,C) \xrightarrow{e} (Q',V',C')}{(P \parallel Q,V,C) \xrightarrow{e} (P' \parallel Q',V',C')}\ [\ par3\ ]$$

$$\frac{(P,V,C) \xrightarrow{\checkmark} (P',V',C'),\ (Q,V,C) \xrightarrow{\checkmark} (Q',V',C')}{(P \parallel Q,V,C) \xrightarrow{\checkmark} (P' \parallel Q',V',C')}\ [\ par4\ ]$$

$$\frac{(P,V,C) \xrightarrow{e} (P',V',C')}{(P \mid\mid\mid Q,V,C) \xrightarrow{e} (P' \mid\mid\mid Q,V',C')}\ [\ intl1\ ]$$

$$\frac{(Q,V,C) \xrightarrow{e} (Q',V',C')}{(P \mid\mid\mid Q,V,C) \xrightarrow{e} (P \mid\mid\mid Q',V',C')}\ [\ intl2\ ]$$

$$\frac{(P,V,C) \xrightarrow{\checkmark} (P',V',C'),\ (Q,V,C) \xrightarrow{\checkmark} (Q',V',C')}{(P \mid\mid\mid Q,V,C) \xrightarrow{\checkmark} (P' \mid\mid\mid Q',V',C')}\ [\ intl3\ ]$$

$$\frac{(P,V,C) \xrightarrow{e} (P',V',C')}{(P\ \triangle\ Q,V,C) \xrightarrow{e} (P'\ \triangle\ Q,V',C')}\ [\ intr1\ ]$$

$$\frac{(Q,V,C) \xrightarrow{e} (Q',V',C')}{(P\ \triangle\ Q,V,C) \xrightarrow{e} (Q',V',C')}\ [\ intr2\ ]$$

Given a process P in CSP#, we can verify whether P satisfies *deadlock-freeness*, *divergence-freeness*, *deterministic* or *nonterminating*. For the safety properties, the reachability assertion checks whether P can reach a state (P', V', C') where V' satisfies proposition *cond*, i.e. $eva(V', cond) = true$. PAT also supports the refinement / equivalence checking as FDR model checker [92] does.

The full set of *Linear Temporal Logic* (LTL[5]) formula can be verified on CSP# models.

Definition 3 *A LTL formula F on CSP# is defined as*

$$F = e \mid prop \mid \Box F \mid \Diamond F \mid X\,F \mid F_1\,U\,F_2 \mid F_1\,R\,F_2$$

Here e is an event or a channel input/output, prop is a proposition defined on V. $\Box F$ means F holds for entire subsequent paths; $\Diamond F$ means F eventually has to hold in the subsequent paths; $X\,F$ means F holds for the next state; $F_1\,U\,F_2$ means F_1 will hold at least until F_2 holds. $F_1\,R\,F_2$ means F_2 has to be true until and including the state where F_1 become true, or if F_1 never happens, F_2 must remain true.

The LTL can represent the *liveness* properties, which means something must eventually happen. When verifying liveness properties on model, *fairness* constraints are often adopted to rule out the unrealistic scheduling on the model's nondeterminism. Three levels of fairness are defined in [103].

Definition 4 *Let $E =< S_0, e_0, s_1, e_1, \ldots >$ be an execution of the model.*

- *E satisfies* weak fairness *if and only if for any e that eventually becomes enabled forever in E, there are infinitely many i that $e_i = e$;*
- *E satisfies* strong local fairness *if and only if for any e that is infinitely often enabled in E, there are infinitely many i that $e_i = e$;*
- *E satisfies* strong global fairness *if and only if for every transition $s \xrightarrow{e} s'$, if E has infinite many i that $s_i = s$, then there are infinite j that $s_j = s$, $e_j = e$ and $s_{j+1} = s'$.*

With the fairness applied, the model checking algorithms do not take the unrealistic schedulings on the nondeterminism in models. The liveness properties are usually verified with certain fairness constraint. When the model is implemented, the developer shall ensure the program's behavior conforms to the fairness constraint.

The case studies in this thesis use some concepts and techniques from *probabilistic model checking*. Below we introduce the basic definition on the probabilistic model based on Markov Decision Process (MDP) [13]. More information about probabilistic model checking can be found on [64] and [45].

Besides the correctness of the concurrent system, sometimes it is useful to verify the quantitative properties of the system. Adding the probabilistic aspect to the system modeling allows us to calculate the probability that the model satisfy the property. In [104], the module *PCSP* was developed in PAT framework to support *Probabilistic Model Checking*. It introduces the PCSP# language that extends CSP# to allow probabilistic choice in the models. The operational semantics of PCSP model can be expressed using Markov Decision Process.

Definition 5 *A Markov Decision Process is a tuple $M = (S, s_0, Act, Pr)$ where:*

- *S is a set of states;*
- *$s_0 \in S$ is an initial state;*
- *Act is the alphabet of the action labels;*
- *$Pr : S \times Act \times S \to [0,1]$ is the transition probability matrix and Σ such that the labeled transition relation.*

PCSP# supports all the process definitions in CSP#. It introduces the *Probabilistic Choice* operator "pcase" as follows.

$$P = \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k;\ \}$$

Here pr_i is a positive integer called *probability weight*. The process P has the probability $\frac{pr_i}{pr_0+pr_1+\cdots+pr_k}$ to behave as process P_i.

With the probabilistic extension, we can calculate the probability and maximum and minimum probability that a PCSP# model satisfies a property. If the *reward* is defined on the events, we can also use the probability model to calculate the estimated reward when it satisfies the property. [104] and [99] give more information about probabilistic model checking in PAT framework.

2.2 PAT and CSP# model

PAT (Process Analysis Toolkit)[71, **?**] is a general model checker framework which supports modeling, simulating and reasoning of concurrent, real-time and probabilistic systems. It provides a user-friendly simulator and model specific abstractions, such as process counter abstraction will group the identical processes to make the verification more efficient.

PAT adopts a layered design and supports both explicit and symbolic model checking. Different model checkers have been developed efficiently under this flexible framework, such as the PCSP module for the probabilistic CSP models [104, 99] and NesC for the sensor network programs [115, 114]. It has also introduced verification of linearizability, time refinement checking and parallel verification etc.[70, 101]. PAT supports different levels of fairness [101], including the strong and weak event fairness, the strong and weak process fairness, as well as the global fairness. PAT supports CSP# model to be verified on the properties such as deadlock-freeness, divergence-freeness, refinement and LTL properties etc.

An input file of the CSP# model is composed with three parts: the global definition, the process definition and the assertion. The *global definition* declares the global shared variables that can be accessed by all the processes in the model. The *process definition* describes how each process behaves and how these processes communicate with each other. The *assertion* part contains the properties to be validated on the model. We give two simple CSP# models in the remainder of this section. They will be referred to in the subsequent chapters.

Figure 2.1 shows the classic *dining philosopher problem* be modeled in CSP#. This model contains two philosophers and two forks. These two forks are shared by the two philosophers. In

```
1 : #define N 2;
2 : Phil(i) = get.i.(i + 1)%N → get.i.i
3 :        → eat.i → put.i.(i + 1)%N → put.i.i → Phil(i);
4 : Fork(x) = get.x.x → put.x.x → Fork(x)
5 :       []get.(x − 1)%N.x → put.(x − 1)%N.x → Fork(x);
6 : College() =|| x : {0..N − 1}@(Phil(x) || Fork(x));
7 : #assert College() deadlockfree;
8 : #assert College() |= [] <> eat.0;
```

Figure 2.1: CSP# Model: Dining Philosophers Example

order to eat the spaghetti, a philosopher needs to acquire both forks, on his left side and on his right side. As two philosophers sit facing each other, one's left hand fork is shared as the other's right hand fork. Both philosophers are using the strategy that first tries to get the left fork then the right one. The deadlock happens when each philosopher has grabbed the fork on his left hand and keeps trying to grab the right one. Starvation happens when one philosopher always get both forks before the other philosopher get the forks. In this example, we use the processes to model the "philosophers" and the "forks". When they synchronize on the "get.i.j", it means philosopher "i" has grabbed the fork "j". Similar meaning applies to the event "put.i.j". After a philosopher get two forks, the one on his left and the one on his right, he can perform the "eat.i" event, which means philosopher "i" is now enjoying his dinner.

In the model source file, line 1 defines a constant "N" that determines the number of the philosophers. Line 2 to 3 describe the behavior of a philosopher, modeled as process "Phil". It takes an "i" as parameter, which is used as the identification number of the philosopher. The behavior of fork is modeled as process "Fork" in line 4 to 5. It will either be acquired by philosopher "x" and be put down by the same philosopher, or be acquired by philosopher "(x-1)%N" and be put down. The process "College" on line 6 is composed of "N" philosophers and "N" forks synchronizing with each other by the *parallel operator* $||$. Line 7 and 8 define two assertions to be verified on the model. Specifically, line 7 verifies whether the model will go into deadlock status and line 8 verifies whether the philosopher "0" can always eventually have his dinner.

Another CSP# model is shown in Figure 2.2. This model defines a *readers-writer* lock that allows concurrent read and exclusive write on a shared queue data structure. This model uses the shared variables to constrain the behaviors of the reading and writing threads.

Line 1 to 3 declare a constant "M" and two variables "writing" and "noOfReading". The value of the Boolean variable "writing" denote whether there is a thread writing on the queue. Initially, it is set to "false". The integer variable "noOfReading" denotes how many threads are reading on the queue and it is set to "0" initially.

Line 4 to 12 are the process definitions ("Writer", "Reader" and "ReadersWriters"). Before starting to read or write, the processes use the *guarded process* to wait for specific condition. The write process (line 4 to 6) waits until number of reading threads drops to 0 and no thread is writing, i.e. "$noOfReading == 0 \&\& !writing$", then it atomically set the Boolean variable "writing" to *true*. After writing, the write thread set the variable "writing" back to *false* to allow other thread to read or write on the queue.

The read process (line 7 to 11) waits until the number of reading threads is less than the allowed maximum number of concurrent read threads and no thread is writing, i.e. "$noOfReading < M \&\& !writing$". When the condition turns *true*, the read process atomically increase the number of reading thread (i.e. "noOfReading") by 1. After the reading operation, it decrease the "noOfReading" by 1.

The whole system starts as the process "ReadersWriters" that contains 4 reading threads and 4 writing threads running concurrently. They are composed with the *interleave operator* $|||$. That is, they do not synchronize on any specific events, but they rely on the shared memory communication based on the shared variables ("writing" and "noOfReading").

Line 13 to 17 contains three properties to be verified on model process "ReadersWriters". This model is verified on deadlock-freeness, the safety property "exclusive" and the liveness property "[]<>someonewriting". Here the "exclusive" means the model shall not go to a situation that some thread is writing on the queue while there are some threads reading on the queue simultaneously, i.e. "$!(writing == true \&\& noOfReading > 0)$". The liveness property "[]<>someonewriting" means that there are always eventually some thread can enter the writing state (i.e. "$writing == true$").

```
#define M 2;
var writing = false;
var noOfReading = 0;

Writer() =
   [noOfReading == 0 && !writing]startwrite{writing = true; }
    → stopwrite{writing = false; } → Writer();
Reader() =
   [noOfReading < M && !writing]
      startread{noOfReading = noOfReading + 1; }
    → stopread{noOfReading = noOfReading − 1; }
    → Reader());
ReadersWriters() = ||| x : {0..3}@(Reader() ||| Writer());

#assert ReadersWriters() deadlockfree;
#define exclusive !(writing == true&&noOfReading > 0);
#assert ReadersWriters() |= []exclusive;
#define someonewriting writing == true;
#assert ReadersWriters() |= [] <> someonewriting;
```

Figure 2.2: CSP# Model: Readers and Writers

As we can see from the examples, CSP# is convenient to model imperative programs with shared memory communication. Although classic CSP can also represent shared variables and asynchronous channels, CSP# represents them in a natural way. Not only do these program-friendly features make modeling object-oriented programs easier, but also they reduce the state space significantly. Moreover, CSP# has other syntax sugars to simplify the modeling on data operations and flow control. Both designing and verifying in CSP# benefit from these features.

Chapter 3

Related Works

3.1 Model Checking Program

One of the important dialect of CSP is the machine-readable CSP (usually denoted as CSPm) [94]. It is developed to describe the parallel systems in CSP and it can be read and verified by automatic tools. CSPm supports the notation of CSP and it includes a functional programming language to enhance the extensiveness and expressiveness. It contains a wide range of data types including numbers, booleans, sequences, sets, tuples and user-defined types. Other features such as pattern matching and lambda terms are added to CSPm. Although the syntax of CSPm is advanced and powerful, its primary purpose is to define the processes in the system rather than the programs.

FDR (and its subsequence FDR2) is the first tool to support CSPm [90]. It is developed by Oxford University Computing Laboratory and licensed by the Formal Systems (Europe) Limited. Generally, FDR2 takes two CSPm models as inputs and verifies whether one of the processes is the refinement of the other. One of these models is the specification of the system and has the properties that the system shall satisfy. The other model is the detail design model of the system. With the refinement checking, we can verify the design model has fulfilled the specification. FDR2 have been applied to model and verify different kinds of protocols and concurrent systems [89, 57]. The ProBE tool [92] from the same research group as FDR provide a graphical simulator for the processes written in CSPm.

SPIN [49, 51, 14] is a successful model checker since the 80s. SPIN aims to provide an intuitive, program-like notation to specify the behavior of the processes and to validate the design model satisfy the correctness requirement in LTL. The model in SPIN is specified in the Promela language [53]. SPIN has a graphical front-end XSPIN for developing the model in Promela. With the Promela parser and LTL parser, the model and the properties are compiled to an on-the-fly verification C program. Running the verification program will give the result that the properties are satisfied, or give the counterexamples that violate the properties.

SPARK Ada [11] has a tool "Examiner" to perform static analysis and property verification. A *Ravenscar* profile [31, 2] is created on each task in the Ada program. The "Examiner" can verify the program's properties based on the Ravenscar profiles. FDR or other model checkers can be used to perform model checking on these properties based on the Ravenscar profiles [26].

Microsoft CHESS project [79, 6] is designed to systematically test concurrent program, but it checks the properties as model checkers do. CHESS controls the thread scheduling of the tested program but it does not store the program state. CHESS enumerates all possible thread schedules with the number of preemption bounded at user-defined number. Deadlock, livelock, data race and assertions can be detected by CHESS.

Usually the stateless approaches can only apply model checking on the terminating programs. In [78], the authors use an explicit fair scheduler on the CHESS tool. For a program that has finite state in a fair scheduler, the extended CHESS tool can verify the safety properties on the nonterminating program.

The C language is one of the most widely used programming languages. To check the C source code for embed system grants interest of the researchers [96]. These embedded systems are usually smaller and contain less states. BLAST [15, 43], SLAM [8, 9] and CBMC [22] are typical model checkers in this category. BLAST and SLAM abstract the data predicate on the program and convert it to boolean program[7] then use SAT solver [30] to verify the properties on them. Other approaches, such as FeaVer [52], AX[50] and FocusCheck[58], translate the C source code to the input of general model checkers. In [76] Mercer et al use the GNU debugger to verify the machine code of embedded systems.

Generally, for the object-oriented programming languages like C# and Java, their standard language syntax and semantics have been defined. The transformation of the program can be conducted on the intermediate language (IL) level or above. On the other hand, the programs of these high-level languages are larger in scale so they usually suffer more on the space explosion problem.

Early version of Java Path Finder (JPF) [41, 42] translates the Java programs to Promela, the input language of SPIN [14]. The classes in the Java program are represented as user-defined data types in Promela. Each object is a record of its type and a unique object ID is assigned. The model uses the object ID as reference to access the data. The methods of the class are translated to macros that use the object ID and the original parameters as the macro parameters. The Java program's properties are represented as methods of the "Verify" class provided by JPF, then translated to LTL formula in SPIN.

To address the dynamic features of Java, dSPIN [29, 27] and JCAT [28] add language extensions on Promela. The object references are represented as the left and right value of the pointers in dSPIN. The methods are represented as functions, which are also referenced by the pointers. The heap data of Java programs are represented in the model's global variables and the stack data are stored with the process data in Promela. With these dynamic data be managed, the models of Java programs can be verified by dSPIN.

Bogor[88, 32, 24] defines an intermediate language BIR to facilitate model checking on program language. The object-oriented programming languages like Java can be easily translated to BIR. On the other hand, it adopts the guarded command format for its control flow and this can be projected to the semantics of other modeling languages conveniently.

After the implemented program has been translated into BIR, Bogor provides abstraction on the program and checks the properties on the BIR level. The open framework approach of Bogor allows convenient extensions to other programming languages and application domains[33, 10, 87].

LLVM2CSP [62, 63] generates CSP model on the intermediate representation (IR) from LLVM compiler. LLVM2CSP translates the program to an application-specific model, and then combines it with the OS model and the hardware model. The combined model is in CSPm [94] and can be checked with FDR2 model checker. LLVM2CSP works on the IR level, which is closer to the OS and hardware, but as it adopts on the translation-based approach, it faces the similar difficulties in scalability as the earlier version of JPF.

Started from version 2, Java PathFinder applied the modified virtual machine approach to model checking Java program at byte-code level. It acts as another Java virtual machine above the native Java virtual machine and internally applies software instrumentation on the Java program running on it. The program state is the snapshot of the program stacks, local variables of all the classes and objects. After executing a Java bytecode, JPF checks whether the verifying property is satisfied. If not, it guides the program to continue traveling to n possible next states, given that there are n enabled threads. The backtracking happens when all the next states have been visited before or the program ends normally.

In JPF, most resources are taken up by the state storage. The state compression and symmetry reduction are applied to dynamic allocated objects and stack data in JPF [68]. In [86] the author applied symbolic execution [23, 59, 20] in Java PathFinder. Other researchers also extended Java PathFinder to wider industrial practices and testing [84, 107, 40, 83] etc.

MoonWalker [19] is inspired by Java PathFinder but targets on C# program on .NET framework. As .NET framework provides a layer that represents the program in its CIL (Common Intermediate Language) bytecode, MoonWalker instruments the CIL bytecode to traverse the state space of the program. The authors also implemented the Memoised Garbage Collector [81] and two partial order reduction techniques to improve the performance of MoonWalker.

3.2 Implementing Concurrent Models

JCSP[108, 109] provides CSP operators in Java. It hides the built-in Java concurrent features, such as *mutex* and *monitor*. Instead of using the explicit synchronizations, the Java program shall only rely on the JCSP library to communication between different components. JCSP supports nondeterminism and fairness concepts via the “Alternative” class. Furthermore, JCSP adds two concepts, *poison* and *immunity*, to channel. The Java programs can use the poison operation to chain-terminate the components that have communication with an already terminated component. The program can also assign different immunity levels on different channels to control how the poison spread in the program. The poison only spreads when the immunity level of the channel is less than the poison strength.

In [110] the authors proved that the operators of JCSP are equivalent to the ones in classic CSP. With these CSP operators, the developer can implement a CSP model in Java with ease. However, when these operators are used in the program, the operation atomicities of the program are hidden in the program structures, making it difficult to check whether the properties of the CSP model are preserved in the implemented program.

Similar to JCSP, CTJ [95, 44] provides the CSP operators in multi-threaded Java programs. It replaces the OS scheduler to provide more flexibility. JCSProB [112, 111] apply JCSP’s idea to provide the operators for the B+CSP model. JACK [36] is another CSP framework for Java program. C++CSP [16, 18, 17] provides CSP operators in C++ language. CSP.NET [67] implements the JCSP like operators in .NET framework.

Some modern program languages integrate CSP concepts, in the language itself or by external libraries. For example, Occam [56] provides *named channels*, *parallel* and *choice* operators for process communication. PyCSP [106] brings CSP to Python via external library. The Go language [100] also uses CSP style channels for synchronization.

In [60, 61] the author proposed an approach which assigns the user-defined functions to CSP events. Different from other approaches, it uses explicit simulator to manage the concurrent model. As the concurrency are separated from the program’s sequential part, this approach allow the model be verified on the concurrent aspect of the program.

CSP++[39, 38, 37] is a framework that uses CSPm in design phase and generates C++ source code from the CSPm model. The properties of the CSPm model are verified by FDR model checker.

The validated CSPm models can be automatically translated to C++ program as the concurrent control layer. After the functionality codes are implemented in C++, CSP++ framework weaves the control layer and the functionality codes into the final program. CSP++ implements a subset of CSP_M, but the validated properties of the CSP_M model preserves in the weaved program. However, the properties are based on the concurrent CSP_M model and do not access the functionality codes.

In [44], Hilderink proposed a graphical notation of CSP. Based on this notation, gCSP [34] provides a graphical tool to design model in CSP diagram. The design model in gCSP can also generate code in Occam and C languages.

Chapter 4

C# Program to CSP# Model

4.1 Overview

Integrating the formal method into the software development process is one of the important goals in software engineering. Verifying the equivalence between the system model and the implemented program is part of the development loop. However, this verification is not trivial, as the implemented program contains many functionality details. Manually converting the implemented program to a formal model and verifying the properties on the formal model can solve the problem but it takes a lot of time and efforts. The converting itself requires modeling specialists and even so, the manual converting may still introduce errors. A solid and automatic transformation from program to the modeling language helps to save these efforts and to avoid introducing errors. The automatic transformation tool is also easier to deploy in the development loop.

The automatic transformed programs shall be easy to understand and ready to be verified on properties. In order to do so, we choose some aspects of the program to be transformed to the model, while the other aspects be simplified or abstracted away to some extent. We shall choose a balanced boundary as it relates to the complexities of the models and the properties that we want to verify. The features of the programming language and the modeling language also constrain on what aspects need to be in the model while the others do not.

As introduced in Chapter 1, we choose CSP# as the modeling language and C# as the programing language. In this chapter, we provide an approach that translates C# source codes to CSP# models to verify concurrent properties. The targets are the multi-threaded C# programs that use C# built-in synchronization between threads. The next chapter will describe another approach based on the Mono virtual machine that executes C# programs. The virtual machine approach can reach the lower level of the program executables. However, the simplification and abstraction are more convenient to be applied on the source code level.

4.2 Analysis on C# and CSP#

Threading is the lightweight concurrent mechanism on today's systems with multi-core processors. Most modern programming languages have built-in supports for thread communication. They usually include the thread creation, mutual exclusion, waiting for specific event and thread interruption. In C# programming language, the "lock" statement and the namespace "System.Threading" provide the inter-thread communication based on monitor.

Figure 4.1 shows a multi-threaded Producer-consumer example in C#. The thread "Producer" constantly puts new entries to the shared queue "buffer" while the thread "Consumer" keeps getting these entries from the "buffer". To prevent the "Consumer" from reading an empty queue, it will wait on the monitor attached on the "buffer". Each time the "Producer" puts an entry in the "buffer", it sends a notification to that monitor. If the "Consumer" thread is waiting on the "buffer", it wakes up and consumes the entries in the shared queue. The program first start two threads, a "Producer" thread and a "Consumer" thread, by calling the "Start()" method of the thread object. The program will also wait on the two threads' terminations then exit the program. [1]

Based on the Producer-consumer example, we discuss the typical thread communication in C#. The object of "Thread" class is created with a method as parameter. This method designates the behavior of the thread when it is executing. Calling the "ThreadStart()" method of the thread object

[1]Here both the "Producer" and the "Consumer" do not terminate and the program will keep running.

```
public class ProducerConsumer
{
  private Queue<int> buffer = new Queue<int>();
  private int c = 0;
  public void Run() {
    Thread t1 = new Thread(Producer);
    Thread t2 = new Thread(Consumer);
    t1.Start(); t2.Start();
    t1.Join(); t2.Join();
  }
  private void Producer() {
    while(true) {
      lock (buffer) {
        buffer.Enqueue(c++);
        Monitor.Pulse(buffer);
      }
    }
  }
  private void Consumer() {
    while(true) {
      lock (buffer) {
        while (buffer.Count == 0)
          Monitor.Wait(buffer);
        Console.WriteLine("Consumed {0}", buffer.Dequeue());
      }
    }
  }
  public static void Main() {
    new ProducerConsumer().Run();
  }
}
```

Figure 4.1: Producer-consumer Example in C#

starts the thread's execution.

The "lock" statement provides the mutual exclusion control in C#. A "lock" statement can be represented as $lock(obj)\{\phi\}$. It accepts an object *obj* as argument. Multiple threads can try to lock on the same object, but at any time only one will get the lock and be executing the code in ϕ. Other threads are blocked on the beginning of the "lock" statement. When the thread runs to the end of the "lock" statement, it unlocks the *obj* and unblocks one of other threads that are trying to lock this object. This thread will be granted the lock of *obj* and it can execute ϕ in its "lock" statement.

Combining the use of the "lock" statement and the "Monitor" class, C# threads can wait on specific notifications attached on the shared objects. When a thread *A* is holding the lock of a specific object *obj*, it can use "Monitor.Wait(obj)" to release the lock of *obj* and block itself, waiting for notification on *obj*. When another thread *B* calls the "Monitor.Pulse(obj)" or "Monitor.PulseAll(obj)", it wake up one or all of the threads that are waiting on *obj*. These threads will wait to re-lock the *obj* and then resume its execution on ϕ, from the statement right after "Monitor.Wait(obj)".

In the C# programs, the statements related to thread communications are the main concerns of the translated concurrent models. In CSP# models, these communications shall be represented as the corresponding processes being blocked or unblocked on specific events. There are other statements in the C# programs that neither locks the "Monitor" objects nor sends notifications on them. These statements can still evoke communications between threads, given that they may read and write on variables that are shared by multiple threads. If the statements only read or write on the local variables of its own thread, they do not relate to thread communication. These statements can be put in the CSP# model without modifications.

As introduced in Chapter 2, CSP# allows shared variables be imported to the models. The processes may read the value of these variables in the conditional expressions, and read and write them in the event-attached programs. The example in Figure 4.2 demonstrates the communication between the event-attached program $e\{prog\}$ and the conditional operator $if(b)\{P\}else\{Q\}$.

If *Sys* chooses to perform the conditional operator of process *P* first, as the value of *x* does not equal to 1 at the beginning, process *P* will perform as $\{e_2 \rightarrow Skip\}$ after that. If *Sys* chooses to

$var\ x = 0;$
$P() = if(x == 1)\{e_1 \rightarrow Skip\}$
$\quad else\{e_2 \rightarrow Skip\};$
$Q() = e_q\{x = 1;\ \} \rightarrow Skip;$
$Sys() = P()\ |||\ Q();$

Figure 4.2: Communication via Shared Variable in CSP#

perform event $e_q\{x = 1;\ \}$ of process Q, the event-attached program sets the value of x to 1. Later when Sys perform process P, as the conditional expression $(x == 1)$ is *true* now, P will perform as $\{e_1 \rightarrow Skip\}$. The communication between process P and Q are through the variable x. The communications between C# threads can be conveniently represented as these communications via shared variables.

4.3 Translation Outline

To translate a C# program to a CSP# model, our approach puts the program data in the user-defined class library, which will be imported to the CSP# model as shared variables. The methods of the classes in the program are translated to processes in the CSP# model. With the concurrent properties provided by the user, PAT tool verifies whether the translated model satisfies these properties. An overview of our approach is shown in Figure 4.3.

The translator takes a C# program's source code ς as input and it outputs the CSP# model ϵ and the source code of a C# class library ι. The model ϵ imports the class library ι and creates the shared variables of the classes of ι in the CSP# model.

For each class c_s in the C# source code ς, there are two classes $\{c_d, c_l\}$ in class library ι to manage the data related to c_s. Class c_d contains all non-static fields of class c_s but it does not define the methods in c_s. Class c_l contains the static fields of class c_s. Furthermore, it has a list to store all objects of class c_d in the model. For an object of c_d, its index in this list and the class c_l info are combined to be the *object ID* in the CSP# model. All the c_d and c_l objects are maintained in a class

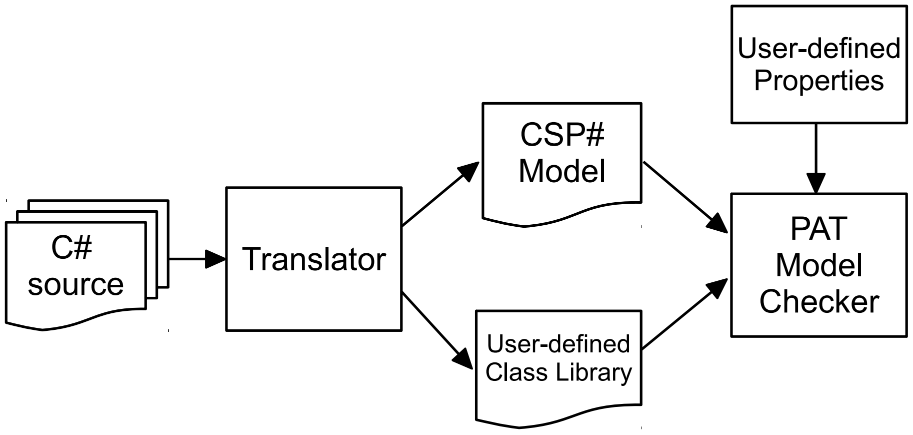

Figure 4.3: Process of Translation-Based Approach

"*Memory*". For a CSP# model ϵ, it defines one shared variable "memory" of class "*Memory*". In the CSP# model ϵ, all the objects are referenced on their object ID. The "*Memory*" class provides the "getter and setter" to access each field of the objects and classes. A *stack* structure is allocated for each thread in the "*Memory*" class.

For each method m in class, a process p_m is defined in the CSP# model. p_m accepts the object ID and the origin parameters of m as the process parameters. The local variables and the return value are stored in the *stack* structure allocated for this thread. For the arithmetic and logical operators in the expressions of the C# program, they stay intact as they are supported in the conditional expressions in CSP#. [2]

In general, each statement in method m is translated to one or more CSP# operators in the model. The assignment statements are put in the *prog* in the event-attached program "$\rightarrow e\{prog\} \rightarrow$". The selection statements (i.e. the "if" and "switch") are translated to the general "IF" process in CSP#. The iteration statements repeatedly execute a block of statements until the loop condition becoming "false". The statements in a loop in C# are put in a subprocess in CSP#, this subprocess repeats itself until the loop condition become "false".

[2]There are a few exceptions that the operators in CSP# expressions have different meaning compared to C#, for example, the remainder operator "%" in C# can result negative value while the "%" in CSP# always get positive value.

For the statements related to thread management, such as the “lock”, “Monitor.Wait()/Pulse()” and “Thread.Start()”, they are translated to the specific operations on the object and thread data in the “memory” object, or the channel operations for the whole program. The detailed translations of them are described in next section.

The execution of the C# program starts from the static “Main()” method. Accordingly, the CSP# model have a process “Main” to initialize the environment, create the objects and start the subprocesses. Besides the process “Main”, a “CNT” process is in charge of the creations of every new threads. When the C# program starts a new thread on the method *m* of object *obj*, the CSP# model send the new thread ID, the object ID and the method ID to process “CNT” process. Based on these IDs, “CNT” performs itself as $CNT = p_m \parallel CNT;$. As result, the new process p_m run in parallel with all other threads’ processes. The whole CSP# model is the parallel composition of the process “Main” and “CNT”.

$$Prog() = p_{Main} \parallel CNT;$$

After the translation, the model checker PAT exhaustively explores the state space of the translated CSP# model and verify the properties. The properties shall be the LTL formula as defined in 2.1. The proposition in the LTL formula can access the global shared variables in the CSP# model. [3] If the property is not valid on the model, PAT provides the trace of the counter example. The counterpart of this trace in the original C# program is the execution that leads to the violation of the property.

4.4 Translation for Specific C# Statements

In this section, we introduce the translation of the statements in C# methods. Table 4.1 summarizes the translation for each supported statement in C#. The first column lists the original C# statement. The second column shows the translated CSP# processes for each of the C# statements. For some C#

[3]The data of the original C# program can be accessed via the “memory” variable in its translated CSP# model.

C# statement	Translated CSP#	Additional Process Definition
υ=ϕ;	$\tau\{\upsilon$=$\varphi\}\rightarrow$Skip;	
obj = new OBJ()	obj = memory.OBJ_create();	OBJ_create() [4]
if(*cond*)$\{\phi_1\}$else$\{\phi_2\}$	if(*cond'*)$\{\varphi_1\}$ else $\{\varphi_2\}$;	

for(*init*, *cond*, *inc*)$\{\phi\}$	($\tau\{$*init*$\}\rightarrow$Skip);
	if(*cond'*)$\{L_1()\}$;
		L_1()= φ;
		($\tau\{$*inc'*$\}\rightarrow$Skip);
		if(*cond'*)$\{L_1()\}$;

while(*cond*)$\{\phi\}$	if(*cond'*)$\{L_2()\}$;	L_2()= φ; if(*cond'*)$\{L_2()\}$;
do$\{\phi\}$while(*cond*)	L_3();	L_3()= φ; if(*cond'*)$\{L_3()\}$;

lock(obj)$\{\phi\}$	OBJ_Lock(obj);
	φ;
	OBJ_Unlock(obj);
		OBJ_Lock()
		OBJ_Unlock()

Monitor.Wait(obj);	OBJ_Wait(obj);	OBJ_Wait(obj)
Monitor.Pulse(obj);	OBJ_Pulse(obj);	OBJ_Pulse(obj)
Monitor.PulseAll(obj);	OBJ_PulseAll(obj);	OBJ_PulseAll(obj)
thobj.Start();	Create_thread!mid.thobj	
return ϕ;	ret = φ;	

Table 4.1: CSP# Translation of C# statements

statements, such as the "for" or "lock", additional CSP# subprocesses are needed for the translation. The third column in the table lists the name of the additional subprocesses. Their specific definitions will be given when we discuss the detail of each statement.

As stated in the previous section, the assignment statements in the method are translated to the event-attached C# program. For an assignment statement $\phi \in m$, the subprocess $\tau\{\phi\} \rightarrow Skip$ is inserted to the process p_m at the ϕ's place. Here the event "τ" does not synchronize with other process, so the event-attached program ϕ can be executed anytime when p_m runs to this place.

The selection statements in C# language include "IF" and "switch" statements. Intuitively, both of them can be represented as CSP# general "IF" statement. For a C# "IF" statement ϕ in the form $if(cond)\{\phi_1\}else\{\phi_2\}$, the corresponding subprocess is $if(cond')\{\varphi_1\}else\{\varphi_2\}$ in translated CSP# model. Here *cond* is a boolean expression in C#. $cond'$ is the CSP# condition expression translated from *cond*, with proper substitutions on the object references. The φ_1 and φ_2 are the

[4]The "OBJ_create()" is the translated CSP# process of the constructor method of the C# class "OBJ"

translated subprocesses of ϕ_1 and ϕ_2. The C# "switch" statement can be represented as a set of C# "IF" statements and be translated to CSP# "IF" subprocesses as well.

The iteration statements of C# include "while", "do", "for" and "foreach" statements. Here we use the "for" statement as the general case to demonstrate the translation. A "for" statement ϕ in C# method *m* is defined as $for(init;\ cond;\ inc)\{\phi_1\}$. Here the *init* and *inc* are two statements. *init* is executed once at the beginning when the "for" statement is executed. *inc* is executed at the end of each iteration. The *cond* is a condition expression to decide when the iteration ends. At the beginning of each iteration, the program evaluate whether *cond* holds. If *cond* is *true*, the program continue the next iteration; if it is *false*, the "for" statement is finished and the program goes on by executing the statement that follows the "for" statement.

In CSP#, we use the tail recursion on the CSP# process to implement the "for" statement in C#. The C# method *m* is translated to process p_m. To represent the "for" statement ϕ in p_m, we create an extra process L_ϕ to represent the repeated behavior of the original "for" statement.

$$\begin{array}{rcl} L_\phi & = & \varphi_1'; \\ & & (\tau\{inc'\} \rightarrow Skip); \\ & & if(cond)\{L_\phi\}else\{Skip\}; \end{array}$$

Process L_ϕ first executes the translated body of the "for" statement φ_1 then it executes the "*inc*′" after it finished one loop. The last sub-process evaluates whether the looping condition *cond*′ is *true*. If not, the process terminates successfully. Otherwise it recurs back from the beginning of process L_ϕ. After defining this L_ϕ, the ϕ in method *m* is translated to

$$\begin{array}{l} \tau\{init'\} \rightarrow Skip; \\ if(cond)\{L_\phi\}else\{Skip\}; \end{array}$$

Here the *init*′ will be executed only once as the "for" statement is defined. Before going to subprocess L_ϕ, the looping condition *cond*′ is evaluated. If it is *true* the process p_m performs L_ϕ until *cond*′ become *false*. If it is evaluated to *false* at the beginning, process p_m does not perform L_ϕ and the looping block φ_1 is not executed at all.

The “System.Threading” namespace provides classes and interfaces for inter-thread communication. Here we focus on the use of the C# “lock” statement and the methods “Wait()”, “Pulse()” and “PulseAll()” from “Monitor” class. They are fundamental for thread communication in C# language and other thread communications can be translated in similar ways as they do.

In the translated user-defined library, two variables are attached to the class of “Obj”.

```
public class Obj {
   public int LOCK;
   public int WAITING;
   ...
}
```

A system channel “ch_notify” is defined in the translated CSP# model.

$$\textit{channel ch\_notify}\ 0;$$

The C# “lock” statement ϕ has the form $lock(o)\{\phi_1\}$. Before executing the ϕ_1, the program enters the monitor of the object o. After ϕ_1 it releases the acquired monitor of o. To represent the entering and leaving of the monitor of object o, two subprocesses are created in the translated CSP# model.

$$\begin{array}{l}
\textit{OBJ\_Lock}(tid, oid) = \\
\quad [0 == memory.\textit{OBJ\_Get\_LOCK}(oid)] \\
\quad (\tau\{memory.\textit{OBJ\_Set\_LOCK}(oid, tid);\ \} \rightarrow Skip); \\
\textit{OBJ\_Unlock}(tid, oid) = \\
\quad assert(memory.\textit{OBJ\_Get\_LOCK}(oid) == tid); \\
\quad (\tau\{memory.\textit{OBJ\_Set\_LOCK}(oid, 0);\ \} \rightarrow Skip);
\end{array}$$

Here the CSP# models use “memory.OBJ_Get_LOCK(oid)” [5] and “memory.OBJ_Set_LOCK(oid, val)” to read and write the field “LOCK” of object o. The oid is the object ID of o in CSP# model.

[5]The “OBJ_Get_LOCK” is the “getter” method defined for accessing the “LOCK” field of the object via the global shared variable “memory”. Similar case applies to the “OBJ_Set_LOCK”.

Process $\mathit{OBJ\_Lock}(\mathit{tid}, \mathit{oid})$ waits the object's "LOCK" become 0 and atomically set it to *tid*, the current thread ID. Process $\mathit{OBJ\_Unlock}(\mathit{tid}, \mathit{oid})$ asserts the object's "LOCK" equals to the current *tid* and set it to 0 to release the lock.

With above two sub-processes, the $\mathit{lock}(o)\{\phi_1\}$ is translated to

$$\mathit{OBJ\_Lock}(\mathit{tid}, \mathit{oid});\ \varphi_1;\ \mathit{OBJ\_Unlock}(\mathit{tid}, \mathit{oid})$$

The *wait* operation "Wait(o)" requires the thread already holds the monitor of o. Executing "Wait(o)" will unlock the monitor and block the thread until other thread *notifies* it. The translated CSP# process for the "Wait(o)" is shown as follows.

```
$\mathit{OBJ\_Wait}(\mathit{tid}, \mathit{oid}) = \mathit{atomic}\{$
    $\mathit{OBJ\_Unlock}(\mathit{tid}, \mathit{oid});$
    $(\tau\{\mathit{memory}.\mathit{OBJ\_Set\_WAITING}(\mathit{oid}, \mathit{memory}.\mathit{OBJ\_Get\_WAITING}(\mathit{oid}) + 1);\ \} \rightarrow$
    $\mathit{ch\_notify}?(\mathit{oid}) \rightarrow$
    $\mathit{OBJ\_Lock}(\mathit{tid}, \mathit{oid}));$
  $\}$
```

Here the field "WAITING" of object o is used to keep track of the number of the threads that are waiting on the object o. After unlocking the monitor and increasing the field "WAITING", the process "OBJ_Wait" blocks itself on the channel read operation " ch_notify?(oid)". The thread that *notifies* on object o will use the channel write operation " ch_notify!(oid)" to wake up the thread that is waiting on o.

Accordingly, the *notify* and *notifyall* operation are translated to processes "OBJ_Pulse(tid, oid)" and "OBJ_PulseAll(tid, oid)" as follows.

```
$OBJ\_Pulse(tid, oid) = atomic\{$
    $If(memory.OBJ\_Get\_WAITING(oid) == 0)\{Skip\}$
    $else\{$
      $ch\_notify!(oid) \rightarrow$
      $\tau\{memory.OBJ\_Set\_WAITING(oid, memory.OBJ\_Get\_WAITING(oid) - 1);\ \} \rightarrow Skip;$
    $\}$
  $\}$
$OBJ\_PulseAll(tid, oid) = atomic\{$
    $If(memory.OBJ\_Get\_WAITING(oid) == 0)\{Skip\}$
    $else\{$
      $ch\_notify!(oid) \rightarrow$
      $\tau\{memory.OBJ\_Set\_WAITING(oid, memory.OBJ\_Get\_WAITING(oid) - 1);\ \} \rightarrow$
      $OBJ\_PulseAll(tid, oid);$
    $\}$
  $\}$
```

The processes first test whether there are some threads waiting on o. If no thread is waiting, the processes do nothing and exit. Otherwise, the "OBJ_Pulse()" performs the channel write operation " ch_notify!(oid)" for one time. The "OBJ_PulseAll()" performs the channel write operation " ch_notify!(oid)" for "WAITING" times, which will wake up every threads that are waiting on o.

In the translated CSP# model, when a process performs a subprocess p_m that corresponds to method m, p_m registers the slots for its local variables in the "memory" object. The variable slots can be referenced in the model by the thread ID and the variable index, which is assigned to each local variable. The "memory" object also holds a variable *ret* for each thread to store the return value of the called method. The "return" statement in C# is translated to the assignment on *ret*.

Table 4.2 gives a summary of the translation for different C# statements. We use ϕ to represent the C# statements and φ to represent the translated CSP# process of ϕ. Some C# statements, such as "for", produce additional sub-processes in CSP#. The third column of the table lists these additional sub-processes. For simplicity in the table, we do not list the full process definitions for the complicated sub-processes, such as "OBJ_Wait()" and "OBJ_Pulse()".

	CSP# model	Model after transformation
Stop	Stop	Stop
Skip	Skip	Skip
Prefix	e->Q	G(e){Q}
Data Op	e{prog}->Q	G(e{prog}){Q}
Channel Out	ch!x->Q	G(ch!x){Q}
Channel In	ch?x->Q	G(ch?x){Q}
Guarded	[b](e->Q)	G([b]e){Q}
General If	if(b){Q}else{R}	G([b]tau | [!b]tau){Q | R}
Atomic If	ifa(b){e1->Q}else{e2->R}	G([b]e1 | [b2]e2) {Q | R}
Blocking If	ifb(b){Q}	G([b]tau){Q}
General Choice	(e1->Q)[](e2->R)	G(e1 | e2){Q | R}
Parallel	(e1->Q) || (e2->R)	G(e1 | e2) {(Q || (e2->R)) | ((e1->Q) || R)}
Interleave	(e1->Q) ||| (e2->R)	G(e1 | e2) {(Q ||| (e2->R)) | ((e1->Q) ||| R)}

Table 4.2: Translation for Specific C# Statements

4.5 Case Study

In this section, we use two examples to demonstrate the translation from C# program to CSP# model. The first example is based on the *dining philosophers* that is introduced in Chapter 2. This example shows how the inter-thread communications in C# are represented in CSP# using shared memory communications. The other example is based on a *leader election* algorithm in a ring network. With customized translation for the "Random" objects in C#, the probability related properties can be verified on the translated model of the algorithm.

4.5.1 Dining Philosophers Example

The dining philosophers problem is usually used to illustrate the synchronization problems that different processes or threads are competing for resources. The forks [6] are taken as the resources and the philosophers are the processes that try to use the resources. A multi-threaded C# version of dining philosophers problem is shown in Figure 4.4 (before the end of this chapter).

[6]The *forks* here shall not be taken as the "fork()" in the programming languages, such as C language.

In the C# program, the “Fork” is the shared object that can be “lock” by the philosopher threads. Each “Philosopher” runs its own thread to sequentially “lock” the left-hand fork and the right-hand fork. When a “Philosopher” thread holds the two forks’ locks, the philosopher thread can have dinner. In the program, the thread output a line saying “Philosopher x is eating” (here the “x” is the index number of the philosopher thread). The program’s static “Main()” method first creates *N* fork objects, then starts *N* philosopher threads to try locking these fork objects.

From this C# source file, the translator generates a user-defined class library “DiningPhilCls.cs” and the CSP# model file “DiningPhil.csp”. All of the “Fork”, “Philosopher” and “DiningPhil” class in C# source have their corresponding classes in the user-defined class library.

For the simplest class “Fork”, although it does not have any fields or methods in original C# source, it still needs to represent the monitor attached to it. So its corresponding class, also named “Fork”, has two fields to model the monitor attached to every objects in C# program. To be exported in the state of the program, it also provides a property “ID” and a method “GetClone()”. The classes “ForkCls” and “Memory” use this property and method to exprot the program state. The translated “Fork” class is shown in Figure 4.5.

The “ForkCls” class manages the list of the “Fork” objects in the program. When the translated model is about to create an instance of “Fork”, the “ForkCls” creates the “Fork” object, inserts it in the list, and returns the object ID. The “ForkCls” class is shown in Figure 4.6.

Similarly, class “Philosopher” has the two classes, “Philosopher” and “PhilosopherCls”, generated in the user-defined library. They help to maintain the “Philosopher” objects and to export their data in the program state. The “Philosopher” class in C# has several fields. The “left” and “right” are references to “Fork” objects. In CSP# model, the objects are referenced on the object ID. So these object reference fields are translated to integer type in “Philosopher”. The translated C# code is listed in Appendix A.2. The “run()” method of “Philosopher” is translated to a CSP# process “Philosopher_run(pid, obj)” in the model. The “pid” is the process ID that the model assigns to each process when it is started in parallel with other processes. The parameter “obj” is the object ID for the “Philosopher”. The translated process is listed as follows.

```
Philosopher_run(pid, obj) =
        (Fork_Lock(pid, memory.Philosopher_Get_left(obj));
        (Fork_Lock(pid, memory.Philosopher_Get_right(obj));
        (Fork_Unlock(pid, memory.Philosopher_Get_right(obj));
        (Fork_Unlock(pid, memory.Philosopher_Get_left(obj))
        ))));
```

The statement that prints to the screen is not in the translated CSP# model, as it has no effect when the model is being verified. The "lock" statements are translated as the paired lock and unlock subprocesses.

In the C# program, the "run()" method of "Philosopher" is used to start the new thread. In the CSP# model, these new threads run as processes in parallel with the whole model. A process "CreateNewThread" takes care of all the new parallel processes creation. When the model wants to start a new process that run the "run()" method of "Philosopher", the process "CreateNewThread" starts the "Philosopher_run" process, as shown below.

```
CreateNewThread() = create_thread?ti.pid.obj
    -> if(-1 == ti) {Skip} else{NewThread(ti, pid, obj)};
NewThread(ti, pid, obj) = case {
        (ti == 1) : Philosopher_run(pid, obj)
        default: Skip
        } || CreateNewThread();
```

The static "Main()" method of class "DiningPhil" is translated to process "DiningPhil_Main(pid)". It uses two loop statements to create the "Fork" objects and "Philosopher" objects and starts the philosopher threads. The process "DiningPhil_Main(pid)" creates one subprocess for each loop statement. The detail of it can be found in Appendix A.1. The whole C# program is modeled as.

```
System() = DiningPhil\_Main(0) || CreateNewThread();
```

Here the "System" include two programs running as processes in parallel. The "DiningPhil_Main" represent the program of the original C# program and the “CreateNewThread” simulates the behavior of the operating system on creating new threads.

The properties can be defined on this program process “System()” and be verified by PAT model checker. The deadlock-freeness property will be defined on it as “#assert System() deadlockfree;”. With slightly modification, such as adding some tracing events or variables, the translated CSP# model can be verified on more interesting properties including the ones in LTL formulas.

4.5.2 Leader Election Algorithm Example

Leader election is a common task in distributed system. Given a set of connected nodes, the leader election algorithm chooses a unique node among them. In this section, we consider the C# program simulating the leader election algorithm in a synchronous ring network.

The ring network contains N nodes connected as ring. The messages are sent in clockwise in the ring. Each node can only receive the message from the node that precedes it, and it only sends message to the node that follows it. There is a global clock to synchronize the communication in the ring network. At each round, a node receives the message sent to it in the previous round, and it can send the message to the next node in the ring.

The algorithm starts when each node chooses a random number from 0 to K as its ID. Each round the nodes send their ID to the next node in ring. When the IDs are unique for each node, the algorithm stops and the node with the maximum ID is the leader. If not, the nodes choose their new random IDs and repeat the process. There are more details for the algorithm in [55].

In the C# program there are two classes “Counter” and “ProcSyn”. When the program is running, the “Counter” object simulates the global clock in the ring network. N “ProcSyn” objects are created and linked in the ring order. They represent the nodes in the network. The “Counter” object controls the nodes to send and receive messages. When the leader is elected, it put the nodes in the “Done” state and stops the program. The C# source code of the leader election algorithm is included in Appendix A.3.

Based on our translation approach, the C# program “LeaderSyn.cs” is translated to the user-defined library “LeaderSynCls.cs” and a CSP# model file “LeaderSyn.csp”. In the original C# program, each “ProcSyn” object chooses its new state based on its ID and the message it get on the current round. The translations of these statements are similar as the dining philosopher example in the previous section. The most difference is that the nodes call the “Random” object to choose an ID. The “Random” is in the “mscorlib.dll” from the .NET framework and the source code of it is not accessible. Here we introduce the customized translation for the“Random” class. Currently, these customized translations are manually defined by users.

Based on the description of the “Random” class in .NET framework, calling “Next(1, K)” on a “Random” object returns a number greater than or equal to “1” and less than “K”. The customized translation for this method in CSP# is defined as follows:

```
RdmNext(pid) =
      (tau{ memory.SetRet(pid, 1); } -> Skip)
      [] (tau{ memory.SetRet(pid,  2); } -> Skip)
      ...
      [] (tau{ memory.SetRet(pid,  K - 1); } -> Skip);
```

This customized translation adds the nondeterminism to the translated CSP# model. With this random behavior of the program being modeled, we can use the PAT to verify whether the program may deadlock, or whether the program can always eventually select a unique leader in the ring network.

```
#assert Prog() deadlockfree;
#define success (goal == 1);
#assert Prog() reaches success;
#assert Prog() |= []<> success;
```

To verify the probabilistic properties on this leader election example, we manually substitute the above customized translation for “Next(1, K)” to the probabilistic choice process as follows.

```
RdmNext(pid) = pcase{
    1 : (tau{ memory.SetRet(pid, 1); } -> Skip)
    1 : (tau{ memory.SetRet(pid, 2); } -> Skip)
    ...
    1 : (tau{ memory.SetRet(pid, K - 1); } -> Skip)
};
```

With this modification, the translated CSP# model becomes a PCSP# model. We can verify the model reaches the "success" state with the probability range. Furthermore, the PCSP# model can be used to estimate the expected number of the communication rounds before the leader has been elected. A "rew" event is added to each round of communication. The model define 1 weight on this "rew" event. We can verify the model will reach the "success" state and calculate the estimated reward when it reaches the state. The assertions for these properties are listed as follows.

```
#assert Prog() reaches success with prob;
#reward rew 1;
#assert Prog() reaches success with reward;
```

For the 3-node case, the verification takes about 7 second and give the result that the estimate reward is about "4.0635".

4.6 Summary

CSP# bases on classic CSP and adds various language features for intuitive system modeling. In CSP#, the interaction between processes and the operation on the shared variables are integrated. These features shorten the gap between the modeling language and the programing language. Making uses of these helps on applying model checking on the software programs.

In our translation approach to verify C# program, CSP# provides the flexibilities on the boundary between program and model. The user-defined class library contains the fields of the classes in C# program. The inheritance and polymorphism are naturally preserved in the class library. The methods of the classes are translated to CSP# processes. They focus on the control flow and the synchronization between threads.

For some complex statements, such as “wait” and “notify” on the monitor, the translations are based on the equivalence on their behavior. The internal data representations for these statements are different between the C# program and the translated CSP# model, but their behaviors have the same effect from the translated model perspective. Using different translation on these statements also influence the state space of the translated CSP# models.

The translated CSP# model allows us to traverse the original C# program state space on the atomicity of the translation. The requirement can be verified as properties defined on the classes and objects of the original program. With customized translations on specific statements or objects, the properties including probability and rewards can be verified on the modified models.

In the implementation of PAT framework [103, 71], the states are internally represented as *string*. Our translation based approach needs to provide the interfaces that PAT framework requires. Integers and booleans can be directly used in CSP# and PAT. Other data need to be represented as user-defined data types and be exported as strings in program states. The translation is also limited on the source code level. The atomicity is at least on the statement level, unless using customized translation. The errors on the lower level, such as the IL code level, may not be detected in the translated CSP# model.

```
public class Fork
{ }
public class Philosopher {
        int name;
        Fork left;
        Fork right;
        public Philosopher (Fork le, Fork ri, int na) {
                left = le;
                right = ri;
                name = na;
                new Thread(new ThreadStart(run)).Start();
        }
        public void run () {
                // think!
                lock (left) {
                        lock (right) {
                Console.WriteLine ("Philosoper {0} eating.", name);
                        }
                }
        } //end run()
}
public class DiningPhil {
        public const int N = 6;
        static public void Main () {
        Fork[] forks = new Fork[N];
        for (int i = 0; i < N; i++)
            forks[i] = new Fork ();
        for (int i = 0; i < N; i++)
            new Philosopher (forks[i], forks[(i + 1) % N], i);
        }
}
```

Figure 4.4: Dining Philosophers Problem in C#

```
public class Fork
{
    public int LOCK = -1;
    public int WAITING;
    public string ID {
        get {
            StringBuilder sb = new StringBuilder("[");
            sb.Append(LOCK.ToString() + ',');
            sb.Append(WAITING.ToString() + ',');
            return sb.ToString().TrimEnd(',') + "]";
        }
    }
    public Fork GetClone() {
        Fork f = new Fork();
        f.LOCK = this.LOCK;
        f.WAITING = this.WAITING;
        return f;
    }
}
```

Figure 4.5: Translated “Fork” class in User-defined Library

```
public class ForkCls
{
    List<Fork> list = new List<Fork>();
    public int CreateObj() {
        list.Add(new Fork());
        return PcMacro.create\_object(PcConst.Fork, list.Count - 1);
    }
    public Fork GetObj(int obj){
        return list[PcMacro.get\_index(obj)];
    }
    public string ID {
        get {
            StringBuilder sb = new StringBuilder("[");
            foreach (Fork el in list)
                sb.Append('[' + el.ID + "],");
            return sb.ToString().TrimEnd(',') + "]";
        }
    }
    public ForkCls GetClone() {
        ForkCls fc = new ForkCls();
        foreach(Fork s in this.list)
            fc.list.Add(s.GetClone() as Fork);
        return fc;
    }
}
```

Figure 4.6: Translated “ForkCls” class in User-defined Library

Chapter 5

VM-Based Verification

5.1 Overview

The approach discussed in the previous chapter is based on the translation from the C# source codes to CSP# models. It is good enough to find out the logic errors in the C# programs. However, the minimum atomicity of the CSP# model is on C# statement level. At the level of .NET virtual machine, one C# statement is generally compiled to one or more *Intermediate Language* (IL) codes. Verifying the C# programs on the IL code level helps on detecting errors at lower level. On the other hand, the program's source code is not always available but we still want to verify the concurrent properties on that program. The translation-based approach usually cannot handle these situations. A possible solution is to check the properties when the program is executed by the virtual machine. This can provide the IL code level verification on the C# programs.

Debuggers are available in most development environments. It is used to examine the target program at runtime. Usually it can stop the target program at specific breakpoints and access the program's data in memory. With necessary modification, the virtual machine and its debugger can execute the target program to traverse all its possible states and validate the properties on each state. Based on these analyses, we propose an approach to verify C# programs based on virtual machine.

We take the multi-threaded C# program at runtime as a LTS model. The snapshot of the program's data in memory will be the state of the program. The execution of one or more IL instructions of this program changes the state of the LTS. These executions compose the transition relation of the model. The properties of the model can be defined on the states or on the *traces* of the LTS.

When the multi-threaded C# program is running, the operating system chooses which thread to be scheduled next. One execution of the program will only traverses one possible trace of the LTS. Furthermore, some errors only happen in some rare cases. To check the properties defined on all the traces of the system, we need a mechanism to visit all possible scheduling of the multi-threaded program at runtime. If some trace violates the properties, this trace can be re-executed by the OS so the developers can investigate the problems.

5.2 Taking Multi-Threaded C# Program as LTS

We focus on verifying single process multi-threaded C# programs. For such a program at runtime, the state of the program contains the following three parts:

Static Data

The static data contains the static fields for each class in the program.

Dynamic Data

The dynamic data contains all the objects created in the heap. The object's data includes all its non-static fields.

Stack Data

A thread has its program context and its stack frame that provides the storage of the return value, parameters and local variables for method calls. All these data are stored as the stack data for each thread.

Suppose a program P has n threads $\{T_1, T_2, ..., T_n\}$, we defined the program state as $s_P = (C, D)$. Here $C = \{c_1, .., c_n\}$ is the program context for each thread. The program data $D = \{D_s, D_d, D_t\}$ is composed with the program's static data D_s, dynamic data D_d, and stack frame data D_t.

When one thread t_i in the program executes one IL code, the program context of this thread changes from c_i to c_i'. The program contexts of all the other threads will remain unchanged, as these threads are not be scheduled to run. Running the IL code also changes the program data from D to D'. We use the thread ID t_i and the address of the IL code a_j as the label, so the transition can be represented as $s \xrightarrow{(t_i,a_j)} s'$. Here we also have $s = (C, D)$, $s' = (C', D')$, $C = \{c_1, .., c_i, .., c_n\}$ and $C' = \{c_1, .., c_i', .., c_n\}$.

A sequence of program state $< s_0, s_1, ..., s_{n-1} >$ is the program execution trace to s_{n-1}, given the followings:

1. s_0 is the initial state of the progam;
2. $s_i \in S$ with $0 \leqslant i < n-1$;
3. there is a runnable transition (t_j, a_k) from state s_i to s_{i+1} for $0 \leqslant i < n-2$;

With the above definition, the C# program at runtime is represented as a LTS system. The state space of the LTS represents the possible behavior of the C# program, given the atomicity of the transition is small enough. The safety properties can be defined as the proposition on the all the state S. The LTL properties in Section 2.1 are based on all possible trace of the program. The *Deadlock* of the program is when there is at least one thread which has not reach its end, but the program state s_i does not have a runnable outgoing transition from s_i.

5.3 Atomicity Control

The above LTS definition is based on the IL code level. The program's state space on the IL level is extreme large. Appropriate abstraction on the IL-based transition significantly reduces the time and space for checking properties on the model. Most programs use the packages provided by the programming languages or third-party libraries. The interfaces of these packages are usually well defined. If a method call to these packages does not directly change the value of the variables in the properties, and the called method is thread-safe, we can abstract the method call as a single transition in the LTS. This is similar to abstract the system at higher level. To allow flexible configuration, we allow the users to decide the atomicity of the program, which influence how the program's state space to be traversed by the checking algorithms.

For flexibility, we suggest three typical atomicity configurations for a multi-threaded program. The *IL level* atomicity takes every IL execution as a transition and labels it with its IL address. The *Source Code level* atomicity models the execution of one line of source code to be a transition. The label of this transition is the line number of the source code, which can be read from the debug symbol file. The *User-Defined level* atomicity allows the users group one or more IL codes inside the same block to be a transition. Providing this flexibility to the user can make the customized virtual machine more useful. When there are blocks of code that are not related to synchronization between threads, the user can mark them as a single atomic transition. Different kinds of abstractions can be implement with user-defined atomicity. They can significantly reduce the state space when the C# program is taken as a LTS. Currently, our tool supports atomicity on IL level and he source code level.

Besides the atomicity configurations, the *package filter* contains the packages' namespaces which are considered as well tested. The code that calls the method in these namespaces will be model as a single transition.

5.4 Traverse the State Space

Based on the configured atomicity, we use the Depth-First Search (DFS) to traverse the program's reachable state space. The search is on the compiled C# program running on the target environment, so there is no false alarm on this VM-based property checking. The modified virtual machine runs the program as usually. When it reaches the end of a transition, it checks the properties and stores the current program state, then continues traversing based on DFS. The outline of the traverse algorithm is listed as follows.

1. Choose an unvisited transition started from current state, run to the end of this transition.
2. At the end of the transition, check whether the current program state violates the property. If yes, stop and print out the trace from begin to current state.
3. Store the current data snapshot and the program context.
4. If the current state has been visited, backtrack to a state which has unvisited transition and continue. If the traversing backtracks to the bottom of the stack and it has no unvisited transition, the search finishes and declares the property is satisfied.
5. If the current state is unvisited, push the state to the stack and query the possible transitions started from it, then go to step 1.

Our tool currently can verify the deadlock-freeness and safety properties on the C# programs. The safety properties are based on the program state. The verifications on the safety properties only need read access to the program state. To verify the deadlock-freeness property, the algorithm checks the outgoing transition on every state. If a non-terminal state has no transition started from it, deadlock occurs on this state. If all non-terminal states have outgoing transition, the program is deadlock-free. As verifying these properties does not change the program state, when the program resumes, its behaviors as it was suspended by the debugger.

The backtracking will restore the program state on both the program context C and the program data D. The program context C will be set for each thread in the program. The program data D are restored for the whole program, as it is shared by all the threads.

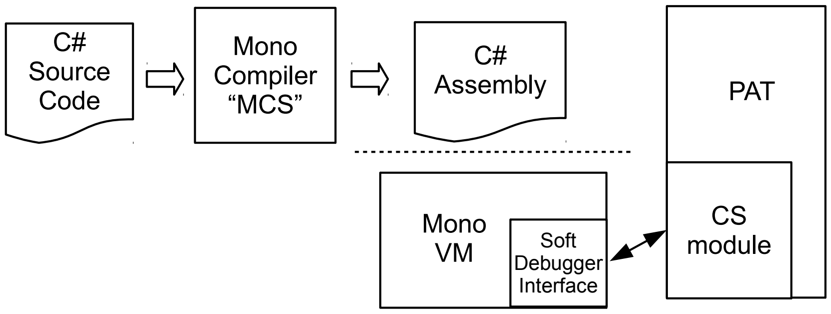

Figure 5.1: Process of VM-based Checking C# program

5.5 Implementation

Our tool targets multi-threaded C# programs running on .NET framework in Windows platform. After the program was compiled, the modified Mono virtual machine (mono-runtime) executes the program's assembly with our customized "Model Checking" mode. At the same time, the CS module in PAT acts as the backend control to communicate with mono-runtime via the soft debugger interface.

As the Figure 5.1 shows, the original C# source file are compiled by the Mono project's C# compiler "mcs". An extra class is built into the runnable assembly, to store the control information used by our model checking. When the C# assembly is built, it runs as normal C# program. The properties to be checked are represented as the attributes of the *Main* method of the program. When the program is run on the modified mono-runtime virtual machine, switching on the "–modelchecking" option will enable "Model Checking" mode. In this mode, the mono-runtime will try to traverse all the possible traces of the program and verify the properties at the end of each transition. The modified mono-runtime communicates with the PAT's CS module via the "Soft Debugger Wire Protocol", which is similar to "Java Debug Wire Protocol" [69].

Since the original soft debugger of Mono does not provide thread scheduling, additional internal events are added to mono-runtime in order to control the state of each individual thread. At each time, the chosen thread is woken up to execute one transition. The other threads are set to wait on the internal events. Which thread to be woken up is chosen by PAT, based on the current program state. Each time the soft debugger finished one transition, it calls a preset method to check the property on the program state. When a thread changes its status, especially from “running” to “waiting”, the mono-runtime sends a special event to debugger thread. As the multi-threaded C# program runs one transition from one thread each time, if the transition is blocked then there will not be any runnable thread in the system. Getting this information the debugger thread will inform PAT and let PAT decide whether it shall wake up another runnable thread or backtrack to the previous state. Compared to the translation-based approach, which need to know explicitly what statement may block the thread, the VM-based verification approach monitors each thread’s status and needs not to recognize the transitions that communicate between threads.

5.6 Case Study

We use the classic dining philosopher problem with two philosophers to demonstrate our VM-based verification tool. The C# source code of this example is shown in Figure 5.2. The line numbers in the figure are the same as the ones in the C# source code file.

The objects “l1” and “l2” represent the two forks on the table. The C# program has two threads running concurrently. Each of them represents a philosopher trying to get the forks. One philosopher tries to get “l1” and “l2” sequentially, the other tries to get them in reverse order. The deadlock happens when each philosopher has acquired one fork and keeps trying to get the other fork.

Users can edit the C# source code and check the syntax in the built-in editor of PAT framework. When the user finishes editing and chooses to verify the program, first the source code is compiled to .NET assembly. The “deadlock-freeness” is put in the properties by default. The safety property can be inputted as the “MCSharp” attribute for the “Main()” method of the C# program. The safety property shall be a valid boolean expression on the program data. When compiling the program, the PAT extracts the safety property and shows it with the “deadlock-freeness” property in the verification window.

```
class PossibleDeadlock{
  public object l1 = new object();
  public object l2 = new object();
  public void Get12() {
    lock (l1) {
      lock (l2) {
        Deadlock.i++;
        Console.WriteLine("Got locks l1&l2");
        Deadlock.i--;
  }}}
}
class Deadlock {
  static public int i = 0;
  [MCSharp("Deadlock.i == 2")]
  public static void Main (string[] args) {
    PossibleDeadlock dl = new PossibleDeadlock();
    Thread sub = new Thread(new ThreadStart(dl.Get12)); sub.Start();
    lock(dl.l2) {
      lock(dl.l1) {
        Deadlock.i++;
        Console.WriteLine("Got locks l2&l1");
        Deadlock.i--;
    }}
  }
}
```

Figure 5.2: Deadlock Example

For the dining philosopher example, the verification on the “deadlock-freeness” gives the “Not Valid” as result. The PAT tool also provides the trace from the program’s initial state to the deadlock state as the counter example. Here the deadlock trace is represented as “ < Init -> t-0-Main-s-20 -> t-0-Main-s-22 -> t-2-Get12-s-9 -> t-0-Main-s-23 -> t-2-Get12-s-10 -> t-0-Main-s-24> ”.

The symbol “->” links the program states together to form the trace. For each state, the number after “t-” is the thread ID. The string after the thread ID is the current method name. In the example they may be “Main”, “Get12” or “Get21”. The number after the “s-” is the line number in the source code file.

Based on the trace we can track back how the deadlock happens. After the main thread “t-0” started the sub-thread “t-2”, it got fork “l2” by event “t-0-Main-s-23”, and the thread “t-2” got fork “l1” by the event “t-2-Get12-s-10”. The main thread “t-0” went on to try acquiring fork “l1” on “t-0-Main-s-24” and was blocked there forever. When the VM-based verification tool tried to schedule the sub-thread, it found all the threads in the program are blocked and at least one thread has not reach its end, the tool declared the “deadlock” was found and printed out the trace.

Checking the safety property of “Deadlock.i == 2” in this example returns “NOT valid”. We can see that the “Deadlock.i” cannot reach 2 unless both two threads acquire the two forks. Obviously, this does not happen in the C# program no matter what scheduling sequence.

We modified the source code in the example slightly, swapping the line 11 and 12, and line 24 and 25 as well. The changed source code is shown as follows.

```
            ......
10 :    lock (l1) {
11 :       Deadlock.i++;
12 :       lock (l2) {
            ......
23 :    lock(dl.l2) {
24 :       Deadlock.i++;
25 :       lock(dl.l1) {
            ......
```

Recheck the safety property “Deadlock.i == 2”, the tool confirms the property can be reached with the trace “ < -> Init -> t-0-Main-s-20 -> t-0-Main-s-22 -> t-2-Get12-s-9 -> t-2-Get12-s-10 -> t-0-Main-s-23 -> t-2-Get12-s-12 -> t-0-Main-s-24>”.

5.7 Summary

The CS module in PAT and the modified Mono virtual machine work together to verify properties on C# programs. The “model checking” mode is added in the Mono virtual machine. When a C# program is executed in this mode, the Mono virtual machine takes control of the thread scheduling and divides the program by transitions at configurable atomicity. The virtual machine takes the program state as snapshot at the end of each transition. PAT guides the Mono virtual machine to execute the C# program forwards and backwards to traversing the state space of the program.

Our approach does not change either the programs’ assemblies or their behaviors on virtual machine. The customized scheduling and the state extraction on C# program are implemented via the debugger interface and embedded code on threading package of the Mono virtual machine. The user can configure the atomicity of the traversing and filter specific domains in the verification, making this approach more adaptable to different modeling and verification tasks. Currently it has limitations that backtrack across multiple methods and the program states are constrained on the information that the debugger can access.

Compared to the translation-based approach in the previous chapter, the VM-based approach has both advantages and disadvantages. It does not require the source code of the C# programs. Different from the translation-based approach which uses variables to logically represent the synchronization, the VM-based approach captures the synchronization on thread states and program counters in C# programs. As it works on lower level, it is easier to support other third-party concurrent libraries. However, VM-based approach generates more states in the LTS and this influences the verification performance. The translation-based approach supports all the properties that can be verified on CSP# models. After the translation, the user may add extra events in the model, or add rewards on specific events. These features make the translation-based approach can fit in a variety of applications. Currently, the VM-based approach only supports deadlock-freeness and safety properties. In practice, it is good complement to testing and debugging at implementation stage.

Using the debugger interface and the internal data of the virtual machine to verify the program is on the boundary of testing and model checking. Similar approaches can be applied to other programming languages that use intermediate representation and virtual machine. VM-based verification approaches usually suffer from the state space explosion more severely. The storage of the program state also costs large amount of memory. More efficient abstractions are needed on the transition atomicity control and program state representation.

Chapter 6

CSP# Model to C# Program

6.1 Overview

As mentioned in the Introduction chapter, CSP has become a popular formalism to model and verify concurrent systems. In recent years, more and more methods and tools have been developed for CSP to be used in the design phase [37, 93, 113, 34]. Some approaches also integrated CSP with other modeling tools such as Z notation and B method [74, 77, 57, 98]. To facilitate implementing CSP models, new programming languages are introduced with built-in support of the CSP operators, such as Occam, Ada and Go etc. [91, 56, 82]. Other approaches provide the CSP operators as third-party libraries [108, 95, 106]. With the further improvement on these methods and tools, CSP becomes trendy in designing and implementing concurrent system in modern programming languages.

CSP# has supports for the program-friendly features such as shared variables and event-attached programs. It allows the shared memory communication and message passing communication inside the same model simultaneously. These language features provide flexibility on modeling the complex distributed systems with CSP#. It is desirable for the developers to have a tool supporting CSP# from design to implementation.

In this chapter, we propose an approach to use CSP# as a designing tool for concurrent system and provide the tool to help implementing CSP# models as multi-threaded C# programs. In the approach, the CSP# model describes the communications between the processes in the system. The other functional operations, which do not relate to concurrent control, are performed between these communications. In implementation phase, the communications and other functional codes are linked together to composite the system in C#. This constructive process needs to ensure the implemented system preserve the properties that have been validated in CSP# models. We structurally prove that the generated C# program is equivalent on the *trace* semantics to the original CSP# model.

6.2 CSP# Semantics in C# Program

Our overall goal is to transform a CSP# model to a multi-threaded C# program that has the same concurrent behavior as the original CSP# model. The communications between CSP# processes are represented as communications between threads. All the processes, events and channels in the CSP# model will have their corresponding classes in the generated C# programs. In this section, we discuss and define the equivalence relation on the behavior between the CSP# model and the C# program.

Let us start the discussion from event equivalence. Each event in the CSP# model shall have a corresponding representation in the C# program. They can be a source code statement, a block of statement, a method call etc. As the event is considered “instantaneous or an atomic action without duration” [46], we would make the event corresponding code as simple as possible. Suppose each event corresponds to one statement, the concurrent behavior of the C# program can be represented by the possible sequences that the program executes these statements. When a CSP# process performs an event, it needs to synchronize the other processes that have the events with the same event name. Therefore, in the statement corresponding to the event in CSP#, the inter-thread synchronization shall be conducted internally.

To make the generated C# program concise and readable, we choose to use a C# method call in one thread to represent a CSP# event synchronization on one process. Based on this, we analyze the differences between the model checker running the CSP# model and the C# program running on the operating system.

When we use a model checker to validate the CSP# model, the model checker can access all the information of the processes. The model checker takes control of the execution of all processes. Based on the current state of each process, the model checker knows what events are enabled and it chooses to perform an event in the enabled event set. When the CSP# model performs the chosen event, all the processes that have this event in their alphabets perform the same event and go to their next states. The model checker then re-computes the enabled event set and makes its next move.

In the multi-threaded C# program, there is no central control to manage the current state for the threads. Each thread only knows its alphabet and its current state. They have to choose one in the enabled event set to execute. The operating system's scheduler decides which thread is executed next. Potential conflicts may occur when different threads have chosen different events to engage.

In the CSP# model, it is assumed that after one event finished, the next enabled event can be performed immediately. On the contrary, after the C# program has finished executing an event method call, when it reaches the next event method not only depends on the program itself, but also depends on when the operating system schedules this thread to execute. Therefore, in the C# program, there is always an interval between the end of the previous executed event and the engagement of the next event. This is similar to how CSP deals with the time-consuming operations. The duration of a time-consuming operation is represented as the two sequential events: the *start* and the *finish* of the operation.

We define the equivalence of model and program on *trace* of the model and the *trace* of the program's execution. The visible events include the event engagements and the channel operations in the CSP# model. In the C# program, there are specific critical sections corresponding to the visible events in CSP# model. Each of these critical sections has one entrance and one exit. The finish of a critical section means the engagement of the corresponding event in CSP# model. For an execution of a C# program, the sequence of these critical sections be executed is defined as the *trace* of this execution. These critical sections are encapsulated as methods of the classes that are defined in the concurrent library "PAT.Runtime".

The transformed C# program shall also require multiple threads to engage a synchronized event consecutively. Suppose threads $\{p_1, p_2, \ldots, p_n\}$ are to engage event e. Here we denote the engagement of thread p_i on event e as $p_i.e$. In the trace of the C# program, when event e is engaged, all the n threads shall engage e. The engagements of all these threads shall occur consecutively, in any possible permutation of $\{p_1.e, p_2.e, \ldots, p_n.e\}$. In the traces of the C# program we group these consecutive event engagements $\{p_1.e, p_2.e, \ldots, p_n.e\}$ from different threads and use a single event e to substitute them. After the substitution, the possible traces of the C# program are the same as one possible trace in the *traces* of the original CSP# model.

Based on the traces definition on C# program, the equivalence of the CSP# model and its generated C# program is defined on their traces. A C# program G is *traces equivalent* to its CSP# model M if $\mathit{traces}(G) = \mathit{traces}(M)$.

We divide the source code in generated C# program into three kinds. The first kind is the message passing communications between threads. They include the event synchronizations and channel communications in CSP#. We use *CSP# synchronization code* to refer this kind of C# code.

The second kind is the *data operation codes* that include the C# code that access the shared variables or the channel buffers. These data are shared globally in the program and they evoke the shared memory communication in CSP#.

The first two kinds of C# code come from CSP# model and they control the communications between threads. We refer them as *communication code* in the C# program. To ensure the atomicity of these communications between threads, they are organized into individual critical sections. Each of these critical sections corresponds to one CSP# communication.

The last kind of C# code does not come from CSP# model and they shall not influence the communication code. We use *non-communication code* to refer them. The non-communication code shall satisfy the following three conditions: (1) They do not access the shared variables or channel buffers in CSP#; (2) They do not modify the control flow related to the communication code; (3) They need to finish in finite time. The non-communicating code can be inserted in the intervals between two critical sections of CSP# communication. As the non-communication code only causes delay between two communications, the *traces* of the program are not influenced by the inserted non-communication code.

Here we choose the *trace* semantics of CSP# as it is well defined and observable. [1] The CSP# model lies in the *communication codes* and it only manages the event traces and the variables that may influence the event traces of the program. For the functionalities that do not go across processes, the programmers can implement them by adding more data and operations in the *non-communication codes*.

6.3 Thread Communication on CSP# Operator Level

With the equivalence defined on *traces*, this section discusses the implementation of the CSP# operators in the C# program. The process level equivalence and the alphabet management will be described in the next section.

6.3.1 Synchronization Using the "PAT.Runtime" Library

In our approach, all the communications across threads are conducted by the operator classes defined in library "PAT.Runtime". The generated C# programs interact with the object instances of these CSP# operator classes. We use a simple event engagement to illustrate the relation between the generated C# program and the objects of the CSP# operators.

As shown in Figure 6.1, the behaviors of process $P = e \rightarrow P$ are simulated by the processes P' and L. Process L actually conducts the synchronization on event e as process P does. Before and after the e's synchronization, process L synchronizes with process P' on events $st.e$ and $ed.e$. Here

[1]The proposition in the properties of CSP# may not preserve in the C# program. Appendix D presents a counter example of these cases.

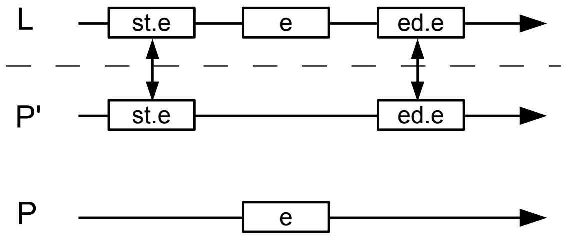

Figure 6.1: Event Engagement Equivalence

$\{st.e, ed.e\}$ only occur locally in the alphabets of processes P' and L. After hiding these two local events, it is easy to verify that process $(P' \parallel L)\backslash\{st.e, ed.e\}$ is equivalent to the original process P.

We take the process L as a method call to the object of event e in the library. The process P' will be a thread in the generated C# program. The interaction between P' and L are actually that the thread P' calls the event's method L. The $st.e$ represents the C# instruction to call the event method and $ed.e$ represents the method's return instruction. When the method returns, the e has been engaged and process P' can execute the code after the event e. All the behaviors of the origin process P engaging e are now happening in this method call.

As discussed in the previous section, we encapsulate the process L as a method m of the CSP# operator object O_e. The event synchronization on $st.e$ represents the start of the method call to m and $ed.e$ represents the return of the method call. With this operator object O_e managed in the generated C# program, the event synchronization behavior of P is represented as the program calls the method $O_e.m$.

6.3.2 Shared Memory Communication

The shared memory communications in CSP# happen on the conditional operators and the data operations. These communications start from a process P that is performing data operation *prog* to the other processes that are waiting on conditional expressions. We use a simple example to explain these communications. Two processes are in the model: a guarded process $P = [b](e_p \rightarrow Skip)$ and a data operation process $Q = e_q\{b = true\} \rightarrow Skip$. Suppose the boolean variable b is *false* at

```
1 : public class Comm
2 : {
3 :   boolean b = false;
4 :   public void RunWait() {
5 :     lock(this){
6 :       while(!b){
7 :         Monitor.Wait(this);
8 :       }
9 :     //now b is true
10 :    }
11 :  }
12 :  public void RunPulse() {
13 :    lock(this){
14 :      b = true;
15 :      Monitor.PulseAll(this); //notify b changed
16 :    }
17 :  }
18 :  ...
```

Figure 6.2: Wait and Notify Example

start, process *P* will be blocked at the guarded condition *b*. After process *Q* has engaged event e_q and executed the attached program "$b = true$", process *P* can engaged e_p as the guarded condition *b* is satisfied. The communication starts from process *Q* when it finishes executing "$b = true$" to process *P* when it is waiting on condition *b*.

The above communications in CSP# are similar to the *wait* and *notify* in multi-threaded C# programs. One typical example in C# is shown in Figure 6.2. Suppose thread *A* is running the "RunWait()" and thread *B* is running the "RunPulse()". The initial value of *b* is *false*. At first, thread *A* is blocked on "Wait()" at line 7 before *b* becomes *true*. When thread *B* gets the lock and changes the value of *b*, it uses "PulseAll()" at line 15 to notify the threads that are waiting on the same "Comm" object. After getting the notification, thread *A* resumes and gets out of the loop from line 6 to 8. At this point, *b* is guaranteed to be *true* until *A* releases the lock at line 10.

Comparing the above C# program and the CSP# communication example, they have the same behaviors based on the boolean variable *b*. We can use the wait and notify mechanism in C#

to implement the communications from the CSP# data operations to the conditional expressions. However, the atomic conditional choice (e.g. $P = ifa(b)\{e_1 \rightarrow Q\}else\{e_2 \rightarrow R\}$) requires the evaluation of the branch's conditional expression (e.g. b or $!b$) and the engagement of the branch's first event (e.g. e_1 or e_2) happen atomically. Therefore, the message passing communication and the shared memory communication cannot be detached into two steps. Waiting on the events to become enabled and waiting the shared variables to be changed shall be represented in the same way in communication between threads. In the C# program, a special event "dc" represents the notification that the shared variables have been changed. This event has higher priority than any other events and channel operations so that no other events can happen before the re-evaluations of the conditional expressions. When a data operation *prog* finishes, it engages this "dc" event. All the processes that are waiting on conditional expressions will engage "dc" event immediately and re-evaluate the conditional expressions. With this "dc" event, a process can wait on message passing and shared memory communications in one operator. It also enables the support for the atomic operators with conditional expressions.

6.3.3 General Choice Operator in C#

To combine the message passing and shared memory communications, a *general choice* operator is used in our concurrent library "PAT.Runtime". It generalizes the event engagements (including channel communication), choice operator and conditional operators in CSP#. The general choice includes a set of alternative events with optional precondition and attached data operation on each event. Based on the functionalities, the general choice operator is divided into three layers as shown in Figure 6.3.

Each layer has two operations, one occurring before the event engagement and the other occurring after it. The *precondition layer* is the lowest layer. It evaluates the preconditions $\{b_1, \ldots, b_n\}$ for each branch before events engagement. If the precondition is *true*, the branch's first events will be put into the event set s_c for next layer. After the event engagement and other layers' operations, the second operation in this layer sends "dc" notification if the engaged event has attached program *prog*. In the middle is the *choice layer*. It allows trying and waiting on an event set s_c. If there are

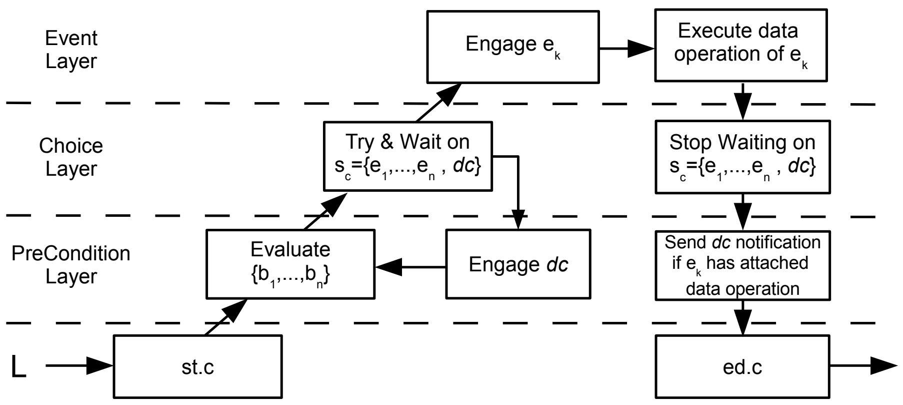
Event Layer
Choice Layer
PreCondition Layer
L
st.c
Evaluate {b1,...,bn}
Try & Wait on sc={e1,...,en , dc}
Engage ek
Engage dc
Execute data operation of ek
Stop Waiting on sc={e1,...,en , dc}
Send dc notification if ek has attached data operation
ed.c

Figure 6.3: General Choice Structure

preconditions on the branches, the data change event "dc" is also included in s_c. After the event engagement, at this layer the general choice operator removes itself from the waiting list of each event in s_c. The *event layer* is the uppermost layer. It engages the first enabled event e_k in s_c. The non-determinism and fairness mechanism are based on the OS scheduler and the sequence to try the events in s_c. If presented, the data operations *prog* attached on event e_k is executed right after the event engagement. If the general choice operator engaged the "dc" event, it return to the start of precondition layer, to re-evaluate the preconditions $\{b_1, \ldots, b_n\}$.

The general choice operator is the fundamental synchronization unit in our "PAT.Runtime". All the synchronizations in CSP# can be represented using a general choice and a general conditional choice operator. For simplicity, we define a *general operator* "G" to discuss the representations. This operator can be represented as the CSP# model below.

$$\begin{aligned}&G([b_1]e_1\{prog_1\} \mid [b_2]e_2\{prog_2\})\{Q \mid R\}\\ =\ &([b_1]e_1\{prog_1\} \rightarrow Q)[]([b_2]e_2\{prog_2\} \rightarrow R)\end{aligned}$$

With the "G" operator, the *event prefix* process $P = e \rightarrow Q$ can be represented as $P = G(e)\{Q\}$. The *atomic conditional choice* $P = ifa(b)\{e_1 \rightarrow Q\}else\{e_2 \rightarrow R\}$ is represented as $P = G([b]e_1 \mid [!b]e_2)\{Q \mid R\}$.

When the *parallel* process $P = (e_1 \rightarrow Q \mid\mid e_2 \rightarrow R)$ is used as one of the branches of the

	CSP# model	Model after transformation
Stop	Stop	Stop
Skip	Skip	Skip
Prefix	e->Q	G(e){Q}
Data Op	e{prog}->Q	G(e{prog}){Q}
Channel Out	ch!x->Q	G(ch!x){Q}
Channel In	ch?x->Q	G(ch?x){Q}
Guarded	[b](e->Q)	G([b]e){Q}
General If	if(b){Q}else{R}	G([b]tau | [!b]tau){Q | R}
Atomic If	ifa(b){e1->Q}else{e2->R}	G([b]e1 | [!b]e2) {Q | R}
Blocking If	ifb(b){Q}	G([b]tau){Q}
General Choice	(e1->Q)[](e2->R)	G(e1 | e2){Q | R}
Parallel	(e1->Q) || (e2->R)	G(e1 | e2) {(Q || (e2->R)) | ((e1->Q) || R)}
Interleave	(e1->Q) ||| (e2->R)	G(e1 | e2) {(Q ||| (e2->R)) | ((e1->Q) ||| R)}

Table 6.1: CSP# Operators Represented with “G” Operators

choice operator, the choice operator may choose one event in the possible first event sets of the paralleled subprocesses. After the choice operator performs the chosen event, the whole parallel process is started. This can also be easily represented with “G” operator as

$$P = G(e_1 \mid e_2)\{(Q \mid\mid (e_2 \rightarrow R)) \mid ((e_1 \rightarrow Q) \mid\mid R)\}$$

The representation of *interleave* process is similar to the *parallel* process. The other representation of the CSP# operators using “G” operators are listed in Table 6.1.

6.4 Process Level Implementation and Alphabet Management

6.4.1 Alphabet Management for the Processes

Processes are the basic units for composition in CSP# models. The process expression explicitly defines the behavior of the process. It also implicitly defines the alphabet and first visible event set for the process. In the C# program, process classes have to provide corresponding interfaces as in the model. For a simple process $P = e \rightarrow Q$, the alphabet αP contains e and αQ. The process Q

needs to provide its alphabet αQ to process P. For the choice process $P = Q[]R$, P uses the first visible event sets of two subprocesses Q and R to decide which branch to perform. Therefore, in the C# program, each process class provides two methods: one represents its alphabet set and the other represents its first visible event set.

We use κP to denote the first visible event set of process P and $\rho(e)$ to denote the number of threads that e is synchronized on. With the alphabet interface defined for process objects, the alphabet management in C# program includes the following four scenarios.

- When a process calculates its alphabet, it adds all the events in process expression (e.g. e in $P = e \rightarrow Q$) and the alphabet of all its subprocesses (e.g. αQ in $P = e \rightarrow Q$) to its alphabet.
- When a process calculates its first visible event set, it adds the first visible event set for each of its branches. For a process defined as $P = (e \rightarrow Q)[]R$, the κP contains e and κR.
- For a *parallel* process $P = Q \mid\mid R$, if an event e is in the alphabets of both Q and R (i.e. $e \in \alpha Q$ and $e \in \alpha R$), it will be synchronized by one more threads. Therefore, when P starts, $\rho(e)$ is increased by 1 and after both Q and R finish $\rho(e)$ is decreased by 1.
- For a *interleave* process $P = Q \mid\mid\mid R$, if an event e is in the alphabets of both Q and R, an extra event e' is used to represent e in process R. Event e' does not synchronize with e, but the other processes which have e in their alphabets now alternatively synchronize to e or e'. When P starts, process Q synchronizes e as usual and the e in process R is substituted by e'. For the other processes in the model, if they have e in their alphabets, $e[]e'$ is used to substitute the e in their alphabets. After both Q and R finish, the event e' is removed from all the processes that have e' in their alphabets.

6.4.2 Interface of the Process Class

To provide the alphabet management discussed above, we use an abstract class "PatProc" in "PAT.Runtime" library to manage the process interface. All the process classes need to inherit the "PatProc" class

```
public class P : PatProc
{
  ...
  static public HashSet⟨string⟩ Alphabet(...){...}
  public ChoiOptSet FirstOpts(...){...}
  constructor of process P
  public void setParas(...){...}
  public void init(){...}
  public void run(){...}
}
```

Figure 6.4: A Process Class Example

and implement its abstract methods. There are several events and data operations containers defined in the fields of "PatProc". The process classes use these containers to store the local events objects.

As shown in Figure 6.4 shows, a process class will implement the five abstract methods in "PatProc". The "Alphabet()" will provide the alphabet of process *P* given the parameters. The "FirstOpts()" method returns a set that contains all possible first events with their preconditions. The "setParas()" and "init()" methods are in charge of setting up the parameters and initialize the subprocesses.

6.4.3 Transforming the Process Expressions

The "run()" method in the C# process class is directly transformed from the process expression in CSP# model and it is structurally similar to the original process expression. The operators and the alphabet for the process have been properly managed in the process' initialization methods, i.e. "setParas()" and "init()". In the "run()" method, the statements that perform the CSP# operators are organized similar to the process expression. In the following, we discuss the transformations of different operators.

For the *Stop, Skip, event* and *channel* operators, their corresponding C# statements in the "run()" method are relative simple. Table 6.2 lists their initialization code in the "setParas()" method

Operator	Initialization	Execute
Skip	evchs[“Skip”] = new PSkip();	evchs[“Skip”].exec();
Stop	evchs[“Stop”] = new PStop();	evchs[“Stop”].exec();
$\rightarrow e.i \rightarrow$	evchs[“e.i”] = new PEvent(“e”,paras[“i”],..);	evchs[“e.i”].exec();
$\rightarrow ch!i \rightarrow$	evchs[“ch!i”] = new PChannelOutput(“ch”,paras[“i”],..);	evchs[“ch!i”].exec();
$\rightarrow ch?i \rightarrow$	evchs[“ch?i”] = new PChannelInput(“ch”,paras[“i”],..);	evchs[“ch?i”].exec();

Table 6.2: Generated C# Code for Simple Operators

in the column “Initialization”. The column “execution” are the statements which will be put in the “run()” method of the process class.

For the *data operation* operator $e\{prog\}$, in CSP#, the e no longer synchronizes with other events even if they have the same name. Only the *prog* will be put in the “run()” method. Before and after the *prog*, the thread need to acquire and release the global data lock to prevent data race on shared variables. The statements in *prog* are valid C# program so they do not need much transformation. The only difference is about how to access the variables and process parameters. In CSP# these variables are globally accessible, but in C# program we use a “Glo” class to store the shared variables. The process parameters are the fields of the process class. Appropriate prefixes are added to the variables in *prog* before they are put in “run()” method.

“PAT.Runtime” provides a special class “TSeq” to sequentially executes the “run()” methods of the processes in its internal stack. To perform a subprocess in C# program, we only need to create the subprocess instance and put it in the “TSeq” object that is attached to current thread. When the current “run()” method returns, the “TSeq” object automatically executes the “run()” of the subprocess. For a sequence process $P = Q_1;\ Q_2;\ \ldots;\ Q_n$, in the “run()” of P we add the subprocesses in reverse sequence to current thread’s “TSeq” object. The last added process will be at top of the internal stack of“TSeq”. Therefore, “TSeq” can execute these n subprocesses in the correct sequence.

Both the parallel and interleave operators start multiple threads to execute their subprocesses respectively. As discussed in the previous section, the alphabet sets shall be expanded before executing these subprocesses. After these subprocesses finished, the alphabet sets will be contracted.

With appropriate expansion and contraction on the alphabet sets, the parallel and interleave operators use the “run()” method of “PatParallel” class to start the subprocesses simultaneously. This “run()” method returns only after all the subprocesses have finished their own “run()”.

The *choice* operator is the only operator for which the structure in the C# program is slightly different from its corresponding process expression in CSP#. It uses the general choice operator “PChoice” to gather the first possible events and their preconditions for all the branches. After this initialization, calling the “select()” of the “PChoice” object will start the operator. After the method returns, the returned value indicates which event it has performed. Base on the returned index, a *switch..case* statement takes the program to the chosen branch. For an example $P = (e1 \to Q)[](e2 \to R)$, the following pseudocode is in the “run()” method of process P .

```
$Event[]\ ev\ =\ new\ Event[]\{e1,\ e2\};$
$PChoice\ pc\ =\ new\ PChoice(ev);$
$int\ sel\ =\ pc.select();$
$switch(sel)\ \{$
  $case\ 0:$
    // go to branch Q

    $break;$
  $case\ 1:$
    // go to branch R

    $break;$
$\}$
```

The conditional operators, including *case, guarded, IF, IFA* and *IFB*, share the same code structure as *choice* in the “run()” method. The difference is that for each branch, the conditional operators will insert the appropriate conditional expression as the branch’s precondition. For the *case, IF* and *IFB*, extra τ events are inserted at the beginning of each branch. The structures of the C# code of these operators follow the “general operator” representations for CSP# operators in Table 6.1.

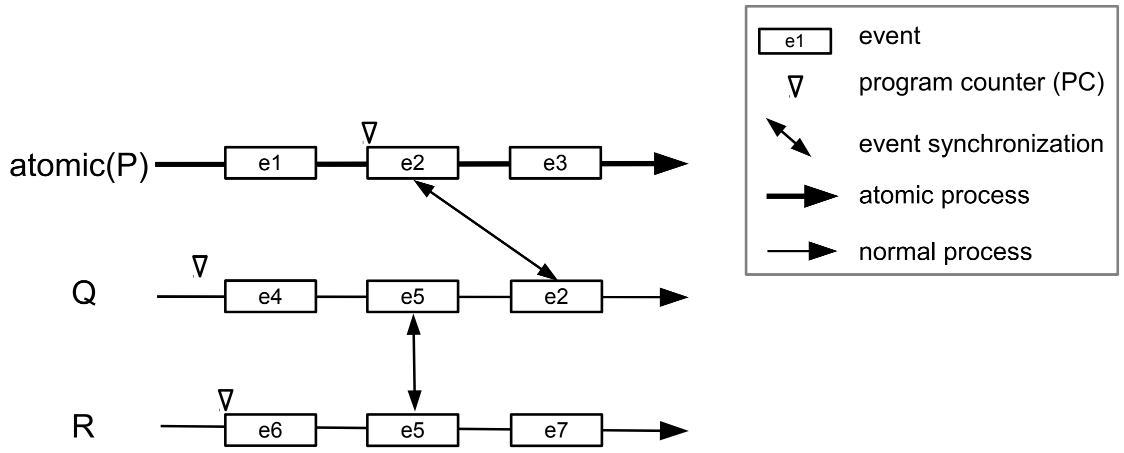

Figure 6.5: Atomic Process Example

6.4.4 Discussion on *Atomic* and *Interrupt* Operators

The *atomic* and *interrupt* operators are not supported in our tool by design. In theory, they can be implemented as the other CSP# operators. In practice, they may bring down the concurrent performance of the program. In this section, we use an example to discuss how the performance is influenced.

The *atomic* operator is denoted as *atomic*$\{P\}$. It assigns the higher priority to the process inside the *atomic* operator. When *atomic*$\{P\}$ is performed as one process in the model, if P is about to engage an enabled event, it will be engaged before any other events from the non-atomic processes in the model. When *atomic*$\{P\}$ is blocked on some event e, other processes are allowed to execute. But once event e becomes enabled, process *atomic*$\{P\}$ regains its higher priority and continue to execute until it finishes or is blocked again.

When there is one or more *atomic* processes are blocked in the model, the corresponding C# program may need extra communication between the threads corresponding to the non-atomic processes. Let us consider the model as follows.

$$
\begin{aligned}
P() &= atomic\{e1 \rightarrow e2 \rightarrow e3 \rightarrow Skip\};\\
Q() &= e4 \rightarrow e5 \rightarrow e2 \rightarrow Skip;\\
R() &= e6 \rightarrow e5 \rightarrow e7 \rightarrow Skip;\\
Sys() &= P() \mid\mid Q() \mid\mid R();
\end{aligned}
$$

The process P is an *atomic* so the model *Sys* will first engage event $e1$ and P will be blocked on $e2$. At this point the processes Q and R are allowed to execute. The generated C# program in this situation is shown in Figure 6.5. The $\triangledown$ in the figure indicates the current PC position of the thread. When thread R reaches the event $e6$ and thread Q is at the interval before event $e4$, R cannot engaged $e6$ although it is enabled. The reason is that in the C# program, thread R does not know whether Q will enable the *atomic* thread P, when Q finishes running the program in the interval. In this case, the system cannot allow any non-atomic threads to engage an event until all the threads finished their intervals. After one or more non-atomic threads engaged an event ei, the other threads still cannot engaged the enabled events. They have to wait until the threads which engaged event ei to finish their intervals again. This adds additional communication between the non-atomic threads although they originally do not have to communicate.

The *interrupt* operator $P \ \triangle \ Q$ behaves as P normally. Once the first event of process Q is engaged, P is interrupted and the process behaves as Q afterwards. Supposing the first event of Q happens when P is in one of its intervals, i.e. running the non-communicating code between two event engagements, there are two possible behaviors on P to stop itself. The first possible behavior is that P do not stop immediately and it keeps on running until it is about to engage the next event. The second possible behavior is that process Q actively stop P right after Q engage its first event. Both cases produce the same trace as the CSP# model. However, the second possible behavior will add the communications between threads happen the critical section and the non-communicating code. Currently, we retain the implement of *interrupt* operator to avoid the ambiguity.

6.4.5 The State Space of Generated C# Programs

CSP# can model both terminating and non-terminating multi-threaded programs. For terminating programs, one execution traverses one route in its state space. This route ends with either a success finish state or an error state. There are two kinds of non-terminating programs. Programs in the first kind will constantly visit their initial states. Programs of the other kind do not visit the initial state anymore from some point in their executions.

When the model checking algorithm verifies a CSP# model, it traverse all possible traces that the model can produce under certain fairness constraint. For the generated program, its executions are different to what model checking algorithm does. For example, the programs do not need to backtrack to a previous state, nor do it store the visited states. As stated in Section 6.2, our code-generation approach is based on the *trace* semantics. Given a model M and its generated program G, $traces(G)$ is equivalent to $traces(M)$. This means the generated program G can also traverse the state space as its model M does. For a terminating program, repeatedly executing the program for infinite times will traverse the state space of its corresponding model. For a non-terminating program that constantly visit its initial state, one execution of the program traverses its model's state space. For a non-terminating program that does not constantly visit initial state, we also need to repeatedly execute it for infinite times to ensure it traverse the whole state space of the model.

Additionally, to ensure the repeated executions traverse the state space of the model, the program shall behave with the fairness constraint as in the model checking algorithm. Our implementation of the CSP# operator supports the *weak fairness* as discussed in Section 2.1. When the operator starts the "try and wait" operation on an event set s_c, the operator tries to chooses an enabled event e_i in s_c. If none of the events is enabled in s_c, the operator waits on all the events in s_c. If multiple events are enabled in s_c, one of them is chosen non-deterministically. Weak fairness guarantee if an event is enabled after some point in the execution, it will be engaged infinitely often [103]. To fulfill this constraints, we need to ensure no continuous enabled event in s_c is ignored forever.

The implementation of general choice operator keeps track of the index of last engaged event e_i in s_c. On the next time this operator is executed, this e_i is given the lowest priority and the event e_{i+1}

is given the highest priority. Supposed the operator G has n events in s_c, if an event e is continuously enabled, the operator will eventually choose this event before its nth iterations. This is because at most after n iteration on the operator, the event e will be set to the highest priority in s_c. As for at most every n iteration, the operator G will engage event e once. To keep the event e enabled continuously, the model will have constantly engage the operator G if the number of processes (i.e. $\rho(e)$) that has e in their alphabet does not change. If $\rho(e)$ decreases, another operator G' that still has e in its s_c' will guarantee e be engaged before at most n' iterations on G'. When $\rho(e)$ increases, if there is a upper bound on $\rho(e)$, the last operator G'' to "try and wait" on e in its s_c'' guarantee e be engaged before at most n'' iterations. If there is no upper bound on $\rho(e)$, the model is an infinite model, which falls outside the scope of this thesis.

We have discussed the fairness constraints on the model and the generated program, however, in practices there are user activities in the non-communication code. For the (repeated) execution(s) of the generated program to traverse the state space as the original model, these user activities shall not violate the fairness assumption on the model.

6.5 The Proof of Correctness

In this section, we prove that the generated C# program performs the same possible *traces* set as the original CSP# model does. This trace equivalence guarantees that the validated properties of the model are preserved in the generated C# program.

The proof is discussed on different functionality layers as shown on Figure 6.6. The basic event synchronization is implemented on C# "Monitor" class. For the shared memory and message passing communication, we use CSP# to build the model of the general choice operator and validate this model generates the same trace as the model of original CSP# operator. On the process and model level, we prove that the "run()" methods in C# program have the same structures as process expressions and executing the program produces a valid instance in the *traces* of the original CSP# model.

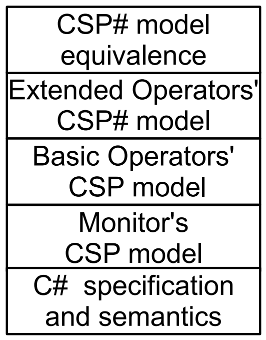
CSP# model equivalence
Extended Operators' CSP# model
Basic Operators' CSP model
Monitor's CSP model
C# specification and semantics

Figure 6.6: the Equivalence Hierarchy

6.5.1 CSP Operators Level Equivalence

For the implementation of CSP operator in program, The *monitor* [47, 54] is a fundamental mechanism to synchronize between threads in the programming languages which adopt the shared memory communication. Languages like Java and C# provide the built-in support for monitor. Monitors in these languages usually at least provide *mutual exclusion* on specific objects and the *waiting* and *signaling* between threads.

Common operations on a monitor object include *enter*, *leave*, *wait*, *notify* and *notifyall*. The "enter(obj) " operation allows a thread to "enter" the monitor "obj" if no other thread has already entered, otherwise it blocks the thread until other thread "leave" the monitor and the operating system's scheduler chooses this thread to run. If a thread has entered the monitor "obj", the "leave(obj) " allow the thread to leave the monitor and other threads can enter the monitor "obj" thereafter. The *wait*, *notify* and *notifyall* operations require the thread has already entered the monitor "obj". The "wait(obj)" operation leaves "obj" and blocks the thread until some other thread calls "notify(obj)" and the operating system chooses this thread to be unblocked, or some other thread calls "notifyall(obj) " which unblocks all the threads that are waiting on "obj".

JCSP [108, 110] is a Java implementation of classic CSP operators. The JCSP library includes the *event*, *channel*, *choice* and *parallel* operators. For the convenience of development, JCSP also added some additional features on event and channel, such as the *bucket* and *poison* structures, to

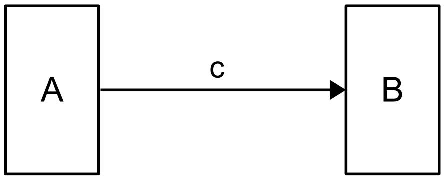

Figure 6.7: Original Channel Communication

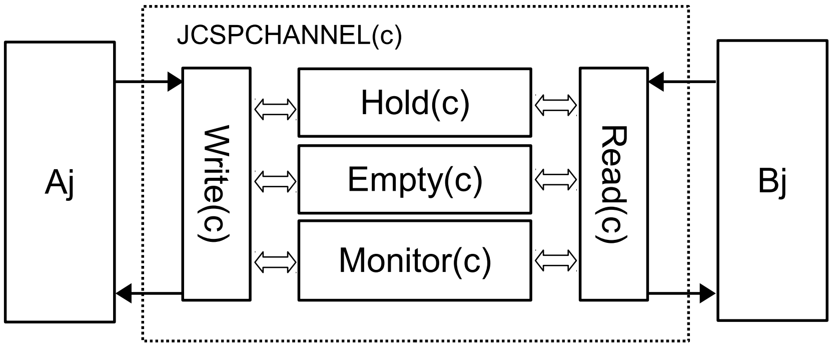

Figure 6.8: JCSP Channel Communication

better support terminating programs. In [110], Welch et al. used CSP to build the models of the Java implementation of the operators. These CSP models are checked in the FDR tool to ensure they are equivalent to the ones defined in CSP.

Welch et al. modeled the monitor's communication on 5 CSP channels: *claim*, *release*, *wait*, *notify* and *notifyall*. The model of the monitor is composed of two active processes: One ensures only one thread can entered the monitor at any time. The other maintains the list of the threads that are waiting on this monitor

To verify the implemented JCSP channel equivalence, Welch et al. used a CSP model containing process A send a message via channel c to process B (as in Figure 6.7). The JCSP implementation model is processes Aj and Bj paralleling with JCSP's channel model $JCSPCHANNEL(c)$.

The $JCSPCHANNEL(c)$ is composed with a monitor's process ($Monitor(c)$), two variable processes ($Hold(c)$ and $Empty(c)$), process $Read(c)$ and $Write(c)$. The $Read(c)$ is the abstracted from the method “read()” of class “One2OneChannel” in JCSP. In the method “Read()”, the statements to operate on the monitor will be represented as the channel event on $\mathit{Monitor}(c)$; the read and write operations on the variables are represented as the channel events on $\mathit{Hold}(c)$ and $\mathit{Empty}(c)$.

Similar case is on process $\mathit{Write}(c)$. The structure of the *JCSPCHANNEL* model is shown in Figure 6.8.

The *Aj* synchronizes with the $\mathit{Write}(c)$ on the beginning and ending of $\mathit{Write}(c)$ to send the message, *Bj* synchronizes with $\mathit{Read}(c)$ at its beginning and ending. The synchronizations on the beginning and ending represent a thread start calling the Java method “read()” and when “read()” finishes, it returns to the program context which calls the method.

With the above models all in CSP, Welch et al.used FDR to check and confirm the equivalence of these two models. Besides the *channel*, Welch et al. also verified the equivalence of other JCSP operators including *event*, *choice* and *parallel* [110].

As the message passing communications in CSP# are equivalent to the ones in CSP, they are implemented on C# “Monitor” class in the similar way as in Welch’s approach. The *trace* equivalence of JCSP applies to the message passing operators in CSP#. Next, we discuss the equivalence of CSP# specific operators, which include both message passing and shared memory communications.

6.5.2 CSP# Models of the Extended operators

We use *atomic conditional choice* as a typical CSP# extended operator to prove the correctness. The other CSP# specific operators can be proved in similar way.

First we build the CSP# model of “PChoice” working with data-operation events. This model ensures that the evaluation of the condition expressions will be mutually excluded from the data-operation and it can be notified when shared variables have been changed. Based on the “PChoice” model, after filled with the condition expressions and the branches of an *IFA* operator, we got the process “G1” as follows.

```
$DataChg() \;=\; (dc \rightarrow G1());$
$G1() \;=\; (evstart \rightarrow$
    $if(b \;==\; TRUE)\;\{$
        $evend \rightarrow (DataChg()\;[]\;CBranch(0))$
    $\}\; else\; if\; (\;!(b \;==\; TRUE)\;)\;\{$
        $evend \rightarrow (DataChg()\;[]\;CBranch(1))$
    $\}\; else\; \{$
        $evend \rightarrow DataChg()$
    $\}$
  $)\;[]\;DataChg();$
```

The “G1” do not accept *dc* between events “evstart” and “evend”. These two events model that the precondition evaluation needs to acquire the global data exclusive lock. The data operations and precondition evaluations are mutual excluded to each other. This is modeled by the process “GMul” as follows.

$$GMul() \;=\; dcstart \rightarrow dcend \rightarrow GMul()\;[]\;evstart \rightarrow evend \rightarrow GMul();$$

Each data operation synchronizes on “dcstart” before accessing shared variables and synchronizes on “dcend” after the operation ends. At the end of the data operation, it sends out the *dc* notification. If the process “G1” has not visited the “CBranch” branches, it synchronize the *dc* and restart the precondition evaluation. We use a process “Alt” to model there are always some processes trying to set the variable *b* to *true* and some others trying to set it to *false*. An “OutSys” process simulates at any time there may be some other event happening.

```
$AT() \;=\; dcstart \rightarrow atomic\{dt\{b = TRUE\} \rightarrow dc \rightarrow Skip\};\;(dcend \rightarrow Skip);$
$AF() \;=\; dcstart \rightarrow atomic\{df\{b = FALSE\} \rightarrow dc \rightarrow Skip\};\;(dcend \rightarrow Skip);$
$Alt() \;=\; (AT()\;[]\;AF());\;Alt();$
$OutSys() \;=\; os \rightarrow OutSys();$
```

With above four parts of model, the CSP# model of the C# implementation of *IFA*, denoted as $M1m$, is presented in following.

$M1() = Alt() \;||\; GMul() \;||\; G1() \;||\; OutSys();$
$M1m\_r() = start\{b = FALSE\} \rightarrow M1();$
$M1m() = M1m\_r()\backslash\{evstart, evend, dcstart, dcend\};$

The origin CSP# *IFA* model, “G0”, is straightforward. Only an extra branch is added to allow the *dc* event to happen. With the same processes “Alt”, “GMul” and “OutSys”, the process “M0m” is modeled as follows.

$G0() = (ifa(b == TRUE)\ \{CBranch(0)\}$
  $else\ \{CBranch(1)\}$
  $)\ []\ (dc \rightarrow G0());$
$M0() = Alt() \;||\; GMul() \;||\; G0() \;||\; OutSys();$
$M0m\_r() = start\{b = FALSE\} \rightarrow M0();$
$M0m() = M0m\_r()\backslash\{evstart, evend, dcstart, dcend\};$

Both “M0m” and “M1m” hide the events “evstart”, “evend”, “dcstart” and “dcend”. These events are not in the trace of the *IFA* operator and they are only used to avoid data race. Using the refinement checking in PAT tool, we get the desired result that “M0m” and “M1m” are equivalent on their *traces*.

$\#assert\ M0m()\ refines\ M1m();$
$\#assert\ M1m()\ refines\ M0m();$

6.5.3 The Model Level Equivalence

As we have proved that the operators in “PAT.Runtime” library generate the equivalent visible trace as their corresponding CSP# operators. At the process and model level, we will prove the generated C# program at runtime is a *bi-simulation* of the *Labelled Transition System* of the CSP# model.

Definition 6 *Given two LTS $L_0 = (S_0, \Sigma, \longrightarrow_0, s_0)$ and $L_1 = (S_1, \Sigma, \longrightarrow_1, s_1)$, p and p' are two states from S_0 and S_1. We say that p and p' are bi-simulation of each other (denoted as $p \approx p'$) if and only if:*

- *For all $e \in \Sigma$ if $p \xrightarrow{e}_0 q$, then there exists $p' \in S_1$ such that $p' \xrightarrow{e}_1 q'$ and $q \approx q'$*
- *For all $e \in \Sigma$ if $p' \xrightarrow{e}_1 q'$, then there exists $p \in S_0$ such that $p \xrightarrow{e}_0 q$ and $q \approx q'$*

Two LTS are bi-simulation $L_0 \approx L_1$ if and only if $s_0 \approx s_1$.

In the LTS of CSP# model, a state is represented as (P,V,C) and a transition is represented as $(P,V,C) \xrightarrow{e} (P',V',C')$. Here the valuation V contains all the globally shared variables in CSP# model. In the generated C# program, these variables are put in the static fields of the "Glo" class. The valuation C contains all the cached channel data on the model's current state. In the generated C# program, the cached channel data are stored in a first-in-first-out queue. When the C# program starts, it initializes the values of these variables and channels. As long as the operation on these variables and channels are equivalent, the valuation of V and C are equivalent for the CSP# model and its generated C# program.

As defined in Section 6.2, the non-communication codes are not allowed to access the shared variables and channel buffers. Therefore, only the message passing communications and data operation codes may change the values of V and C. In the generated C# programs, these two kinds of codes come from our code generation tool. And in CSP#, the embedded event-attached programs are in a subset of C# language. In the supported C# statements, they share the same operation semantics. [2] In this way, if the process expression P in the CSP# model and the generated C# program are equivalent, and the operator level guarantee the atomicity of the message passing communications and data operation codes, the state (P,V,C) will be equivalent. In the rest of this section, we discuss the equivalence on process expression P.

In the generated C# program, the labeled transition $\xrightarrow{e}$ is one thread running one or more statements but at most one of these statement is event synchronization or channel read/write operation. The success transition $\xrightarrow{\checkmark}$ is a successful termination of a process or subprocess.

Given a process expression ε, the LTS with ε is denoted as M_ε. We use η to denote the generated C# program from ε. The LTS of the generated C# program is denoted as C_η.

[2]Except the remainder operator "%".

Theorem 6.5.1 *The LTS of the η is bi-simulation of the LTS of origin process ε. i.e. $M_{\varepsilon} \approx C_{\eta}$.*

Proof: As the operator level equivalence is validated in CSP# models, the proof focuses on the generated C# program have the same possible transitions as in the CSP# model. We make a structural induction on the CSP# process definitions. Currently, the code generation tool supports the following CSP# operators in the process definition.

$$\begin{aligned} P = \; & Stop \mid Skip \mid e \rightarrow P \mid e\{prog\} \rightarrow P \\ & \mid ch!x \rightarrow P \mid ch?x \rightarrow P \mid [b]P \\ & \mid if(b)\{P\}else\{Q\} \mid P;\; Q \mid P[]Q \\ & \mid P \parallel Q \mid P \mid\mid\mid Q \end{aligned}$$

$\varepsilon = Stop$: CSP# defines *Stop* to have no transition out. The implementation of *Stop* in "PAT.Runtime" is to block the thread forever. It will not perform any transition, so $M_{Stop} \approx C_{Stop}$

$\varepsilon = Skip$: In CSP# model, $Skip = \checkmark \rightarrow Stop$. The implementation of *Skip* in "PAT.Runtime" is to exit the "Skip.run()" method. Nothing will be performed after the exit. Obviously, this means first successfully exit the *Skip* and have no transition after that, i.e. $C_{Skip} = \checkmark \rightarrow C_{Stop}$. As $M_{Stop} \approx C_{Stop}$, so we have $M_{Skip} \approx C_{Skip}$.

$\varepsilon = e \rightarrow Q$: In LTS of CSP# model there is only one transition $\xrightarrow{e}$from M_{ε} to M_{Q}. The generated C# program η is shown in Figure 6.9(a). The "run()" method will first run to the method call to engage event operator e. Executing the "exec()" method of the operator will either block the thread till the event becomes enabled, or engaged e if it turns to enabled by this call. As the non-communicating code cannot change the control flow of the "run()" method to skip this method call, we get the generated C# program η always executes the engagement of e before it goes to call the "Q.run()". Because "exec()" of e is the corresponding critical section of e in the CSP# model, assuming "Q.run()" is bi-simulate to M_{Q} (by the induction base), we get the $M_{\varepsilon} \approx C_{\eta}$.

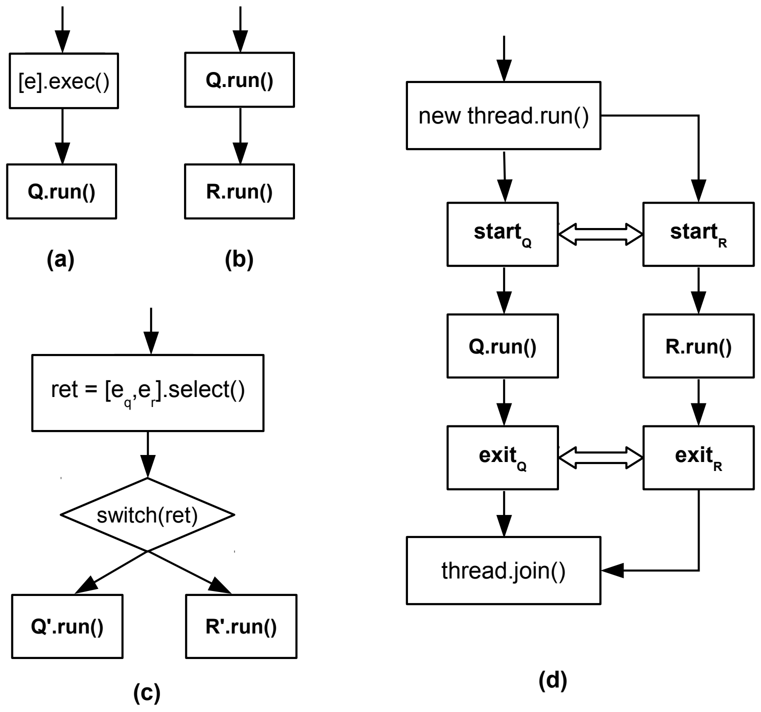

Figure 6.9: Structure of the Generated "run()" Mehtods

Similar results apply to $\varepsilon = e\{prog\} \to Q$, $\varepsilon = ch!x \to Q$ and $\varepsilon = ch?x \to Q$. Each of them only has one transition from M_ε to M_Q and this transition has corresponding single branch C# statements from C_η to $C_Q \approx M_Q$.

$\varepsilon = Q; R$: The CSP# model behaves like Q until Q's successful termination and behaves as R afterwards. So in M_ε, there is one transition $\xrightarrow{\checkmark}$ from the M_Q to M_R. The generated C# program η is shown in Figure 6.9(b). η first executes "Q.run()" and after "Q.run()" successfully returns, it executes"R.run()". The successful return of the "Q.run()" is simulated to $\checkmark$, thus $C_\eta \approx M_\varepsilon$.

$\varepsilon = Q[]R$: Suppose Q and R each has only one possible first visible event, denoted as e_q and e_r respectively. According to the CSP# operational semantics, two transitions start from $Q[]R$: $(Q[]R, V, C) \xrightarrow{e_q} (Q', V, C)$ and $(Q[]R, V, C) \xrightarrow{e_r} (R', V, C)$. Here Q' and R' are the Q and R processes with their first events being skipped. The generated C# program for $Q[]R$ is a two-step program, shown in Figure 6.9(c). The first part is running "select()" on the choice operator which contains the first event of Q and R. Here e_q and e_r are the first events of Q and R respectively. The choice operator may engage e_q and return 0 or engage e_r and return 1, depending on the environment. The second part is based on this return value, switching to "Q'.run()" if it returns 0, or to "R'.run()" if it returns 1. The "Q'.run()" is starting the "run()" method of Q but skip the first event of it, so it is bi-simulated to the Q' in the original CSP# model. Same relation holds for the "R'.run()" and the R'. Now we have two transitions from C_η: one engages e_q and goes to $C_{Q'} \approx M_{Q'}$, the other engages e_r and goes to $C_{R'} \approx M_{R'}$. So $C_\eta \approx M_\varepsilon$. When Q and R contain more than one first event, the transition number will be the total number of the first events of Q and R. The bi-simulation relation holds for the C_η and M_ε.

Similar results apply to $\varepsilon = [b]Q$, $\varepsilon = if\ \ b\ \ \{Q\}else\{R\}$ and other extended conditional operators. They are different on the branch number and transition number, but all of them are sharing the same structure in their "run()" method.

$\varepsilon = Q \parallel R$: According to the CSP# operational semantics, M_ε has three sets of transitions $\{e_q, e_r, e_{qr}\}$. Here e_q is the first events of Q and $e_q \in \alpha Q, e_q \notin \alpha R$; e_r is the first events of R and $e_r \in \alpha R, e_r \notin \alpha Q$; e_{qr} is the common first events of Q and R, $e_{qr} \in \alpha Q \cap \alpha R$. The C# program for $Q \parallel R$ is shown in Figure 6.9(d). It first creates new threads and manage the alphabets for Q and R. The events $E = \{e \mid e \in \alpha Q \cap \alpha R\}$ are expanded and each event in E will be synchronized by one more process. After the alphabet management, the threads of Q and R synchronize on the invisible "start" event, then execute their own "run()" methods. Based on the operator level equivalence, if $e \in \alpha Q \cap \alpha R$ and it is the first event of both Q and R, e is enabled. For the cases that $e \in \alpha Q$ and $e \notin \alpha R$, or $e \in \alpha R$ and $e \notin \alpha Q$, event e is also enabled. As for each transition in M_ε there are corresponding transition in C_η and vice versa, we have $M_\varepsilon \approx C_\eta$.

Similar results apply to $\varepsilon = Q \mid\mid\mid R$ as the only difference is on how to expand the alphabets in "DistributeEvent()". For the interleave process, the "DistributeEvent()" will created an alternative event e' for each event $e \in \alpha Q \cap \alpha R$. The other processes in the model will synchronize to $e[]e'$ if they originally synchronize to e. Subprocess Q will synchronize on original event e and R will synchronize the alternative one e'.

Above we have proved that for each case in the supported process definition, the generated C# program at runtime is bi-simulated to the original LTS of the CSP# model. ■

With this equivalence proved above, the properties on CSP# model will preserve on the generated C# program. The properties include deadlock-freeness, reachability on the event, and LTL properties. For the CSP# model it also allows the formula F defined on the proposition on the global variables. Currently the generated C# program does not preservation on these formulas. The preserved LTL formula F is defined as $F = e \mid \Box F \mid \Diamond F \mid X F \mid F_1 \, U \, F_2 \mid F_1 \, R \, F_2$.

6.6 Case Study

We demonstrate the C# code generation form CSP# model with two case studies. In the first example, the *Turn-Based Game*, we demonstrate directly using the CSP# operators of "PAT.Runtime.dll" in C# program. The detail about how to use the operators is discussed in Appendix E. In the second example, the *Concurrent Accumulator*, we first design the concurrent system model in CSP# then combine the model with user-defined data structures to generate the C# program.

6.6.1 Turn-Based Game

The concurrency library "PAT.Runtime" provides the CSP# operators as C# classes to communicate between threads. From a CSP# model, the programmer can use these operators to implement the model with ease. In this section, we demonstrate the usage of "PAT.Runtime" using a turn-based game program as example. In a turn-based game there are n players connecting to a game server. The server starts the game after all n players joined. At the beginning of each turn, each player submits his action to the server. After all players have submitted their actions, the server sends all the players' actions to every player in the second half of that turn.

For the responsiveness, there is one client serving one player. A client has a "send" thread that sends the action to server on each turn. It also has a "receive" thread that only waits to receive other

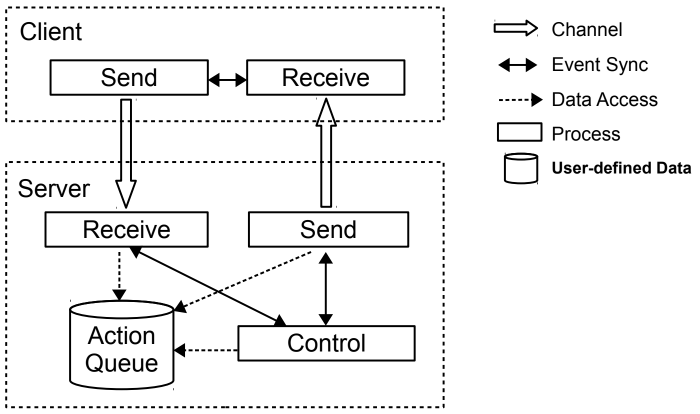

Figure 6.10: Turn-based Game Server-client Hierarchy

players' actions from server. The server has one "send" thread and one "receive" thread for each client. The server side "receive" thread receives the action from the client-side "send" thread while the server-side "send" thread sends the actions to the client-side "receive" thread. At server-side, a "control" thread checks whether all the players have submitted their actions. After all the players have submitted, this thread will inform all the "send" threads to reveal the actions.

Each pair of "send" and "receive" threads use *channel* to transmit the messages. Inside the server or the client, the threads use the event to synchronize the beginning and the ending of each turn. A thread needs to use the *data operation* to access the action queue. The "control" thread use the *guard* operator to track the number of received action. For example, the server-side "receive" thread is designed as the following process *TbServerReceiveRd*.

$$\begin{array}{l} \mathit{TbServerReceiveRd}(i) \ = \\ \quad \mathit{ctos}[i]?i \rightarrow \mathit{enqueue}.i\{\mathit{num} \ = \ \mathit{num} \ + \ 1\} \\ \quad \ \rightarrow \mathit{edHalfRound} \rightarrow \mathit{TbServerReceiveRd}(i); \end{array}$$

The C# program implements the communication of this process in the "run()" method of class

"TbServerReceiveRd" as follows.

```
1 :  string str = (string)chReceive.read();
2 :  GloBase.DataOpBegin();
3 :  server.RoundQueue.Enqueue(str);
4 :  GloBase.DataOpEnd();
5 :  edCanReceive.sync();
```

The "chReceive" in line 1 is the channel object which links to the client. After reading from channel, at line 2 to 4, the program uses the data operation to put the action to queue "RoundQueue". At line 5, the thread synchronizes on the object "edCanReceive". This event object is synchronized by the "control" thread and all the "receive" thread at server side.

The major part of the "run()" method of the "control" thread is as follows.

```
6 :  int res = choices.select();
7 :  GloBase.DataOpBegin();
8 :  output the actions of the current round
9 :  GloBase.DataOpEnd();
10 :  edHalfRound.sync();
11 :  edRoundEnd.sync();
```

The line 6 represents the waiting on all the player actions to be submitted. When the "select()" returns, the "RoundQueue" has already contained all the actions. After the data operation that processes the actions of the current round (line 7 to 9), the "control" thread synchronizes on "edHalfRound" (line 10) to inform the server-side "send" threads to send all the actions to each players. The "edRoundEnd" event (line 11) represents all the operations for this round have finished. It is synchronized by the "control" threads and all the "send" thread at server-side.

After the implementation of client and server threads, they are organized in the program's "Main()" method. The program first creates the event and channel objects, initializes them with the capacity based on the number of client. The server and client objects are created and linked via these event and channel objects. At last the "Main()" method use a *Parallel* object to start them as follows.

$12: \quad \textit{new Parallel}(\textit{new CSProcess}[]\{\textit{server}, \textit{client}1, .., \textit{clientn}\})).\textit{run}();$

Compile and run the C# program, the client and server communicate correctly as desired. In this case study, we can see that with a designed model in CSP#, it is convenient to implement the model with the CSP# operators in "PAT.Runtime". After initializing the event and channel objects, the threads can communicate via these objects the same way as in CSP# model.

6.6.2 Concurrent Accumulator Development

In the previous turn-based game example, the event objects are manually created and linked between multiple threads and objects. The developers have to ensure the control flow do not skip the CSP# operators related statement in the program. Our code generation tool helps to ease these tedious works with automatic alphabet management. The generated C# program has a clean structure that is similar to the origin CSP# model. In this second case study, we demonstrate the development of a *multi-threaded accumulator* to calculate the summation of array concurrently.

The array has n integer elements $\{d_0, d_1, .., d_{n-1}\}$. The elements $\{d_0, .., d_{n-2}\}$ contain the numbers need to be added to d_{n-1}. The result of the summation will be stored in d_{n-1}. The program will start m threads to sequentially read the array and add the result to d_{n-1}. After all m threads finished, $d_{n-1} = m\sum_{i=0}^{n-2} d_i$. Supposing each of these m threads has a read cache limit k_i, when it finishes reading k_i elements, it adds the summation of the k_i elements to the shared array's $n-1$ elements and starts a new round on the next element until it has added all the $n-2$ elements.

The program is divided into three parts, each focuses on one aspect of the program. To avoid data race on the shared array, we use CSP# to model a reader-writer lock to protect the shared array. To access the data in the array and output result on screen, a user-defined class "Ldata" is implemented as a dynamic class library "ldata.dll". The CSP# model can import this library and use the "Ldata" object in event-attached programs.

The m threads are modeled as m "adder" processes and their subprocesses in CSP#. They

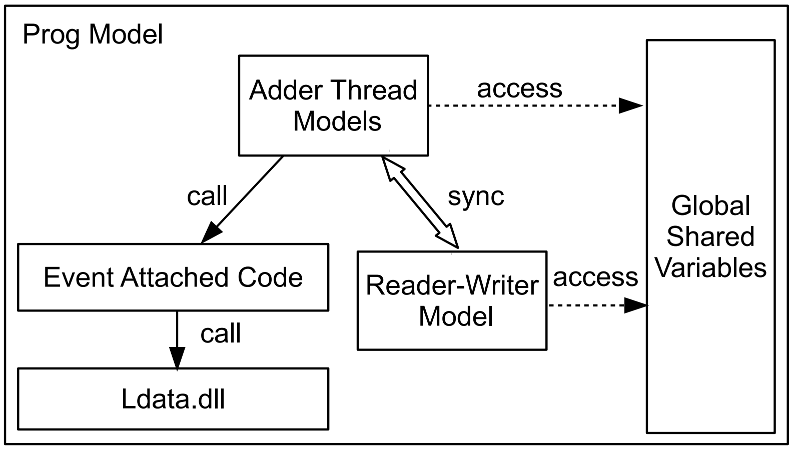

Figure 6.11: Concurrent Accumulator Model Overview

synchronizes with the reader-writer lock model and access the “ldata” objects via the data-operation events. The variables in the condition expression, which control the flow of the model, are stored as the global shared variables. The startup process “prog” initializes the “ldata” object and the control variables, then start *m* “adder” processes and the reader-writer lock process in parallel. After the *m* “adder” processes terminate, the model calls the “print()” method of “ldata” to output the result on screen. The structure of the design is show in figure 6.11.

The development starts on implementing the “Ldata” class without concerning the concurrency, then uses the reader-writer model to add concurrent protection to the shared array, the program-wide functionalities are provided by the “Adder” model. The “Ldata” implementation is quite intuitive so we start from discussing the development of the reader-writer lock model and the “Adder” model.

Reader-writer Lock Model

In the reader-writer lock model, we use an integer “noOfReading” to track how many threads are in reading status and a boolean variable “writing” to track whether the writing lock is already acquired by some thread. Whether a thread can enter the lock is protected by a *Guarded* condition. The prerequisite for entering reader lock is “*noOfReading >M && !writing*” and the one for entering writer lock is “*noOfReading == 0 && !writing*”. The “Controller” process ensures once a thread has been chosen to enter the lock, it atomically tests the prerequisites and set control variables without being interrupted by other threads.

$$
\begin{array}{l}
\mathit{ReaderHead}(i) = [\mathit{noOfReading} < M \;\&\&\; !\mathit{writing}]\mathit{startR.i} \\
\quad \rightarrow \mathit{startreadop.i}\{\mathit{noOfReading} = \mathit{noOfReading} + 1;\ \} \\
\quad \rightarrow \mathit{startRout.i} \rightarrow \mathit{Skip}; \\
\mathit{ReaderTail}(i) = [\mathit{noOfReading} > 0]\mathit{endR.i} \\
\quad \rightarrow \mathit{stopreadop.i}\{\mathit{noOfReading} = \mathit{noOfReading} - 1;\ \} \\
\quad \rightarrow \mathit{endRout.i} \rightarrow \mathit{Skip}; \\
\mathit{Reader}(i) = \mathit{ReaderHead}(i);\ \mathit{ReaderTail}(i);\ \mathit{Reader}(i); \\
\\
\mathit{Writer}(i) = [\mathit{noOfReading} == 0 \;\&\&\; !\mathit{writing}]\mathit{startW.i} \\
\quad \rightarrow \mathit{startwriteop.i}\{\mathit{writing} = \mathit{true};\ \} \rightarrow \mathit{startWout.i} \\
\quad \rightarrow \mathit{endW.i} \rightarrow \mathit{stopwriteop.i}\{\mathit{writing} = \mathit{false};\ \} \\
\quad \rightarrow \mathit{endWout.i} \rightarrow \mathit{Writer}(i); \\
\\
\mathit{Controller}() = (\mathit{startR}.0 \rightarrow \mathit{startRout}.0 \rightarrow \mathit{Controller}()) \\
\quad [\,]\ (\mathit{endR}.0 \rightarrow \mathit{endRout}.0 \rightarrow \mathit{Controller}()) \\
\quad [\,]\ (\mathit{startW}.0 \rightarrow \mathit{startWout}.0 \rightarrow \mathit{Controller}()) \\
\quad [\,]\ (\mathit{endW}.0 \rightarrow \mathit{endWout}.0 \rightarrow \mathit{Controller}()) \\
\ldots
\end{array}
$$

The process “ReadersWriters” is the parallel of processes “Readers”, “Writers”, and “Controller”. They form an autonomous component in the system. The requirement related to the reader-writer lock can be verified on process “ReadersWriters”. For example, the requirement “when the data is being written, no thread shall hold the reader lock. ” is represented as “*!(writing == true && noOfReading >0)*”.

$$
\begin{array}{l}
\mathit{RWReaders}() = \mathit{Reader}(0) \;||\; \mathit{Reader}(1) \;||\; \mathit{Reader}(2); \\
\mathit{RWWriters}() = \mathit{Writer}(0) \;||\; \mathit{Writer}(1) \;||\; \mathit{Writer}(2); \\
\\
\mathit{ReadersWriters}() = \mathit{RWReaders}() \;||\; \mathit{RWWriters}() \;||\; \mathit{Controller}(); \\
\\
\#\mathit{define\ exclusive}\ !(\mathit{writing} == \mathit{true} \;\&\&\; \mathit{noOfReading} > 0); \\
\#\mathit{assert\ ReadersWriters}() \models [\,]\ \mathit{exclusive}; \\
\#\mathit{define\ someonereading\ noOfReading} > 0; \\
\#\mathit{assert\ ReadersWriters}() \models [\,] <> \mathit{someonereading}; \\
\#\mathit{define\ someonewriting\ writing} == \mathit{true}; \\
\#\mathit{assert\ ReadersWriters}() \models [\,] <> \mathit{someonewriting};
\end{array}
$$

The Adder Process

The high-level functionalities are added to the program by the “Adder” process. Each “Adder” process represents one of the *m* threads. They use the “Ldata” object to read and write on the shared array.

As the design, when a thread wants to read the array, it shall synchronize with the “ReadersWriters” process on *startR.i* and *endR.i* events before and after the read operation respectively. Similarly, before and after the write operation, the thread’s model shall synchronize the *startW.i* and *endW.i* events as shown following.

$$\begin{array}{l} startR.i \ \rightarrow \ (read\ operation) \ \rightarrow \ endR.i \\ startW.i \ \rightarrow \ (write\ operation) \ \rightarrow \ endW.i \end{array}$$

The “Adder” process uses global shared variables to control the flow of the threads and to store the summation of already read elements. They include the *cur*, *cnt*, *gap* and *accu*. The *cur* indicates which element the thread is reading; *cnt* counts how many elements have been read at this round; The *gap* is the capacity that the thread can read at one round; The *accu* stores the summation of the previous read data from the shared array.

To decide whether the “Adder” process shall terminate, the *IF* operator tests the condition “*cur[i] <len*”, where *len* equals to $n - 1$ defined above.

$$
\begin{array}{l}
GAdder(i) \;=\; if(cur[i] \;<\; len \;\&\&\; cnt[i] \;<\; gap[i])\; \{ \\
\quad startR.i \\
\quad\; \rightarrow s1\{accu[i] \;=\; accu[i] \;+\; ldata.get(cur[i])\} \\
\quad\; \rightarrow s2\{cur[i] \;=\; cur[i] \;+\; 1;\;\; cnt[i] \;=\; cnt[i] \;+\; 1\} \\
\quad\; \rightarrow endR.i \;\rightarrow\; GAdder(i) \\
\};
\end{array}
$$

$$
\begin{array}{l}
GWrite(i) \;=\; startW.i \\
\quad\; \rightarrow s3\{accu[i] \;=\; accu[i] \;+\; ldata.get(len);\; \} \\
\quad\; \rightarrow s4\{ldata.set(len,\; accu[i]);\; \} \\
\quad\; \rightarrow s5\{accu[i] \;=\; 0;\;\; cnt[i] \;=\; 0;\; \} \\
\quad\; \rightarrow endW.i \;\rightarrow Skip;
\end{array}
$$

$$
\begin{array}{l}
LAdder(i) \;=\; if(cur[i] \;<\; len)\; \{GAdder(i);\; GWrite(i);\; LAdder(i)\}; \\
Adder(i) \;=\; addstart\{cur[i] \;=\; 0;\; cnt[i] \;=\; 0;\; \} \;\rightarrow\; LAdder(i);
\end{array}
$$

Using predefined input data, the safety properties can be checked on the "Adder" process. For example, the "*exclusive*" property can be checked on a specific array $\{d_0, d_1, .., d_{n-1}\}$.

The complete CSP# model of the concurrent accumulator is listed in Appendix C.

Generating the Program

Above we have implemented the user-defined data structure "Ldata"and designed the "ReadersWriters" process and the "Adder" process. Lastly, a "Prog" process is added to do the initialization of the data, and start the "ReadersWriters" process and the "Adder" process in parallel.

Choose the "Prog" as the start-up process, the code generation tool generates the C# project from the CSP# model. There is one class for each process definition and one extra class "Glo" is generated to hold the global shared variables. The project references to the dynamic libraries "Ldata.dll", "PAT.Common.dll" and "PAT.Runtime.dll". The generate project is ready to build in Visual Studio. Executing the program will print the result on screen. The "ReadersWriters" part is non-terminating, so the program does not exit. The "Adder" part will do the calculate job, print out the result and exit.

Examine the source code of each process class, the "run()" methods have the same structures as the process definition in CSP# model. The event name can be read on the event synchronization statement. For example, the statement

```
evchs["endRout.i"].exec();
```

represents the event "endRout.i". The event name of data-operation is displayed in the comment right after the entering of data-operation.

```
// event stopreadop.i
Glo.DataOpBegin();
Glo.noOfReading = (Glo.noOfReading - 1);
Glo.DataOpEnd();
```

Debugging the program is convenient as the statements in the "run()" methods can trace back to the operators in the process expression. Other methods of the process classes are used to manipulate the alphabets and process parameters. These methods are usually executed before or after the process' running.

The initialization of the program can be substituted to read the real data in practice. C# code can be added in the "run()" methods of the processes. The inserted code shall not have side effect on the shared variables of the CSP# model. For example, we can insert the C# code to print a message if the summation in one round is greater than 10 as follows.

```
public void run()
{
   ...
   // event s1
   ...
   Glo.DataOpEnd();
   if(Glo.accu[parai] > 10) {Console.WriteLine("accu > 10");}
   // event s2
   Glo.DataOpBegin();
   ...
}
```

6.7 Summary

To expand the use of CSP# in development, we discussed the differences between the CSP# model checked by model checker and the C# program running in the target platform. With these concerns in mind, we chose the *trace* semantic to define the equivalence between CSP# model and the C# program. Based on this equivalence, we designed the "PAT.Runtime" library to provide the CSP# operators in C# programs. The basic event synchronization in this library is based on the *monitor* class in C#. On the event synchronization, we add the choice layer and precondition layer to implement the general choice operator of CSP#. The shared memory and message passing communications are combined in the *general choice operator* to ensure the C# programs have the same atomicity as the CSP# models. With the CSP# operators from "PAT.Runtime" library, the developers can implement the CSP# model in a similar structure in C#.

With the alphabet and shared variable management being added to the "PAT.Runtime" library, our code generation tool in PAT generates C# programs from verified CSP# models. Executing the generated C# program produces the same possible *traces* set as the original CSP# model does. From the operator to the model level, we proved the trace equivalence of the CSP# model and the C# program. The generated C# program preserves the validated properties on traces of the original CSP# model.

Two case studies are performed to demonstrate the usage of the "PAT.Runtime" and the code generation tool. In the turn-based game example, the CSP# operators and the alphabets are manually managed by developers. In the concurrent accumulator example, the C# project is automatically generated from the original CSP# model. The process classes and their alphabets are managed automatically.

With the "PAT.Runtime" library and the code generation tool, CSP# can be easier to be used in the concurrent software development, from the design to the implementation phases. The representations of the requirement specification, design and implementation are consistent. Our approach help improve the efficiency and reliability of software development with formal CSP# modeling.

For the limitation, currently the developers need to ensure the non-communication codes do not interfere the flow of the communication codes. A better cooperation between our tool and other development tools is preferable. Fit the current semantics equivalence with the whole development process shall further improve the consistency and efficiency of the concurrent software development.

Chapter 7

Improvement on the Implementations of CSP# Operators

In last chapter, we proposed an approach to use CSP# from design to implementation phase in development process. CSP# has demonstrated its advantages in modeling the system behaviors and our tool generates the C# program with the designed behaviors. In this chapter, we discuss the performance of the implementations of the CSP# operators in programming languages. Here the CSP# operators are used to represent inter-thread communications based on *monitor*. To improve the performance of the multi-thread programs that implement the CSP# models, we modify the mechanism of event synchronization for CSP# operators.

7.1 Overview

In the programming languages like Java and C#, the *monitor* provides "mutual exclusion" and "waiting and signaling" between threads. It is a concise and convenient synchronization tool in shared memory concurrent system. For the popular CSP libraries, such as JCSP [108], CSP.NET [67] and CTJ [95] etc., they use monitor to implement the message passing communication in Java or C#.

More specific, the CSP processes are represented as threads in the program. When a process is blocked by an event in the model, the corresponding threads will be waiting on a monitor object. When the event becomes enabled, the thread will be notified via that monitor. However, the monitor objects in program and the events in model are not one-to-one correspondence. A *choice* process can nondeterministically wait on multiple events. In the program, the choice operator corresponds to the monitor that the thread is waiting on. The first visible events on each branch of the choice operator compose the condition variable for this monitor to be notified. Suppose a CSP# choice operator is represented as follows.

$$\begin{aligned}
P = & (e_1 \rightarrow Q_1()\ ) \\
& [] \ (e_2 \rightarrow Q_2()\ ) \\
& \ldots \\
& [] \ (e_n \rightarrow Q_n()\ );
\end{aligned}$$

The process P is waiting on a monitor object and the event set $\{e_1, e_2, \ldots, e_n\}$ is the condition variable of this monitor object.

As the threads in the program are running concurrently, two sets of threads can engage two events at the same time. This violates the event atomicity in CSP. To prevent this violation, a global lock is added to ensure that an event engagement can only happen after the previous event has finished its engagement. Here the engagement includes the maintenance of the related monitors and condition variables, which are done by different threads.

Optimizing the cooperation among the global lock, the monitors and the condition variables shall improve the performance of the inter-thread communication via CSP operators. In this chapter we investigate the synchronization of the CSP# operators in the "PAT.Runtime" library. With rearrange the cooperation between locks and monitors, the duration of the event synchronization is decreased and the communication to signal "the end of the event" is removed. With the optimization, the running time of the programs using "PAT.Runtime" are decreased about 40%. It helps the concurrency library "PAT.Runtime" be more practical for software development with CSP# as the designing tool.

7.2 Current Synchronization Mechanism

Let us first focus on the current implementation of the CSP event synchronization. This implementation is proposed by Welch et al in [110]. The choice operator[1] is the basic unit for synchronization. It contains the event set as its internal variables. A choice operator object has a "choice monitor" that can be waited and notified. When it starts engagement, it actively acquires or releases the global lock. The major activities of the choice engagement are shown in Figure 7.1. For a thread to engage a choice operator, it may go into the 9 activities that are marked as "S1" to "S9" in Figure 7.1.

The "S1", "S4", "S6" and "S9" are related to the global lock. The global lock has 3 states: *available*, *enabling* and *releasing*. When the global lock is in "available" state, any choice operator can get the lock and change it to the "enabling" state. After the choice finishes trying all the events in its event set, if a specific event is chosen to be engage, the choice operator can change the global lock to "releasing" state; if none event is enabled, the choice operator leaves the lock and the state of it changes back to "available".

After an event is chosen to engage, the global lock enters the "releasing" state and the program starts an "ending phase" for the threads related to this event. In this "ending phase", the active thread that starts the event engagement will notify every thread that are waiting on this event. In Figure 7.1, thread "T2" successfully starts the event engagement and notifies "T1" to resume its choice engagement. Each thread that is resumed to finish the engagement needs to conduct two steps, i.e. "S8" and "S9". On step "S8" the choice operator removes itself on the events' waiting list and on step "S9" it registers on the global lock, informing that it has finished the maintenance. When all these threads register the finish of the maintenance, the global lock set itself back to "available" state. This is actually conducted by the last thread that registers to the global lock.

The event objects work as condition variables in the choice operators. It has internal management on how many threads this event has to synchronize. A choice can have two operations on the event object: "enable" and "disable".

[1]In [110], it is called the "ALTing" construct. Here we use the choice operator as it is used in the rest of this thesis.

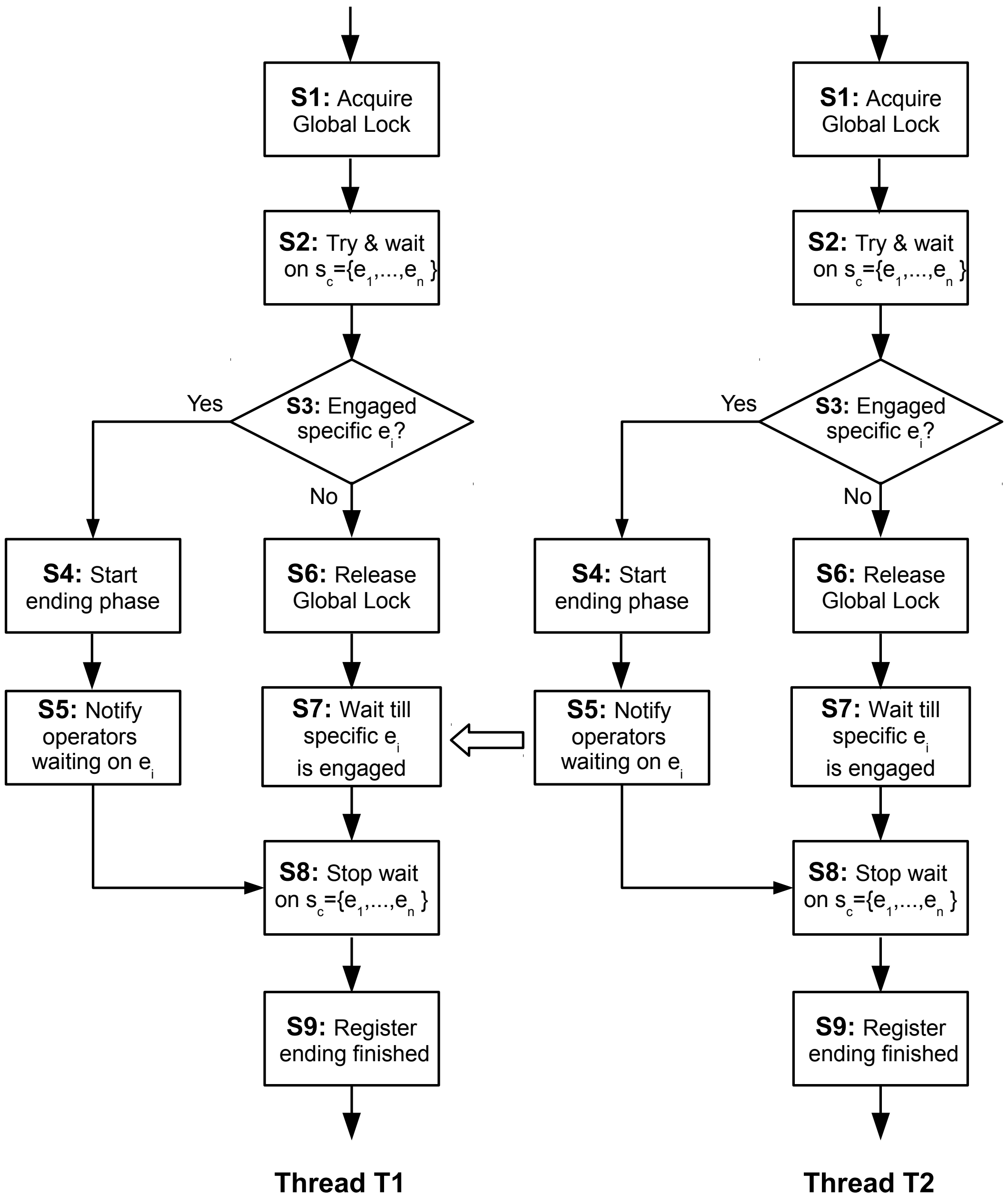
S1: Acquire Global Lock
S2: Try & wait on $s_c=\{e_1,...,e_n\}$
S3: Engaged specific e_i?
Yes
No
S4: Start ending phase
S5: Notify operators waiting on e_i
S6: Release Global Lock
S7: Wait till specific e_i is engaged
S8: Stop wait on $s_c=\{e_1,...,e_n\}$
S9: Register ending finished
Thread T1
Thread T2

Figure 7.1: CSP Choice Operator Communication

Suppose event *e* is synchronizing *n* threads. Let us discuss the event's behavior when a choice tries to "enable" event *e*, if this is not the *n*th thread to synchronize *e*, the event object put the monitor object of the choice in the waiting list of *e*, and return *false* to inform the choice operator that it needs to wait. If this is the last (*n*th) thread to synchronize e, the event will be enabled by this thread. After notifying the previous $n-1$ threads to resume, the event return *true* so this thread knows it can start the “ending phase” and do the maintenance on remove itself from the waiting list of the other events objects.

The event’s “disable” operation is the only way that the choice operator remove itself from the event’s waiting list. When the choice calls this operation on each event in its event set, only the chosen event that is being engaged returns *true*, all the other events return *false*. Based on this return value, the choice operator knows which event has been engaged.

7.3 Improving the Cooperation among events, choices and global lock

We have discussed the synchronization mechanism of CSP operators in last section. To improve the performance of this mechanism, we first analyze the communication in it. Based on the analysis, we propose a simplified cooperation mechanism among events, choices and global lock. This cooperation mechanism is also adapted to support the CSP# operators in “PAT.Runtime” library.

7.3.1 Analysis the Functionalities in CSP Operator Synchronization

For the cooperation among the events, choices and global lock, we try to list the functionalities of them and to see whether they need inter-thread communications.

The event object has a waiting list containing the monitors of choice operators that are waiting on this event. This waiting list may be accessed by multiple threads concurrently. Therefore, a “mutex” lock is attached to each event object to protect the waiting list. The “enable” and “disable” operations on the event object need to acquire this “mutex” lock. Usually they do not send notification to other threads. Only when the choice operator acts as the last thread to “enable” the event, it actively notifies other threads that are in the waiting list of this event.

The global lock ensures no two events can occur simultaneously. When one event is engaging, it also ensure all the choice operators involved in this event synchronization will finish their maintenance in the “ending phase”. Different from the mutual exclusion on event, which happens between different threads when they want to access the same event, the mutual exclusion enforced by the global lock happens between any two threads in the program.

In most cases, the choice operator uses the global lock and the event objects to communicate. As shown in the Figure 7.1, if it is not the active thread to start the event engagement, it has to acquire global lock at “S1” and release it at “S6”. The activities “S2” and “S3” is protected by the global lock. If the choice is the last one to synchronize on the event object[2], the choice operator follows the route “<S1, S2, S3, S4, S5, S8, S9>” and this whole route of activities are protected by the global lock. On step “S5” it notifies all other choices that are waiting on the same event e_i. As at this moment it has already changed the state of the global lock to “releasing” on step “S4”, all the other choices are also protected by the global lock. In other word, when the threads of the other choices resumes, they equivalently hold the global lock until the lock exit the “ending phase”.

The choice operator only gives out the global lock between activity “S6” and “S7”. When the choice operator gets the global lock on “S7”, the global lock is already in the “releasing” state. Therefore, two kinds of communication happen on the choice operators. The first case, the choice notifies the other choices that are waiting on the same event when it successfully starts an event engagement. This communication happens is not global as it only evolves the operators that are waiting on the same event. The second case, the choice registers itself to the global lock in the “ending phase”. This *register* operation blocks any other choice operators that want to access the global lock.

7.3.2 Improved Synchronization Mechanism

For a multi-threaded program that uses the CSP operators, the communications are represented as that when a thread of the program tries to engage a CSP operator, it waits on other threads, or it

[2]i.e. the thread running the choice operator is the nth thread to synchronize on the event object that require n threads to synchronize.

actively resumes other threads. The waiting can occur on two levels. The “global” level waiting occurs between any two threads that want to access the engagements of operators. The “local” level waiting occurs between any two threads that want to access the same event.

The “global” level waiting occurs between operators’ engagements. For convenience, we define two routes of steps “<S1, S2, S3, S4>” and “<S1, S2, S3, S6>” as “trying phase”. Not like the “ending phase” that includes the operations from multiple threads, only one operator can be in “trying phase” at any time. When one operator tries to enter “trying phase”, it may need to wait till another thread to exit its “trying phase”, or wait till multiple threads to finish their “ending phases”. If the “trying phase” and the “ending phase” can run concurrently, the waiting between threads will be decreased considerably. Here the “ending phase” is to ensure the operators to remove themselves from the waiting lists and the waiting lists are stored in different event objects. We will investigate the “local” level waiting and the internal data of event objects to see whether they can make the “ending phase” more efficient.

Before an event object is engaged, the operators can put itself in the waiting list of the event, or remove itself from it. These operations are at “local” level and they have already been protected by the “mutex” lock of the event object. When an operator successfully starts the engagement of an event that needs to synchronize n operator, the program will be in the “ending phase” on this event and this block any other operators that want to enter “trying phase”. This event object will first notify the other $n-1$ operators and the “ending phase” does not finished until all these $n-1$ operators have done their maintenances. The maintenances include the operations on this engaged events and the other unengaged events. That is the main reason that the “ending phase” needs the protection of the global lock.

To minimize the waiting between the “ending phase” and other operators that are not evolved in this event engagement, all the operators’ maintenances are shifted from the individual operators to the operator which is the active one to start the engagement. After this change, the maintenances previously in the “ending phase” have already done, thus the “ending phase” can be removed.

To cooperate with this change, the maintenance of the operator is merged with the step that the

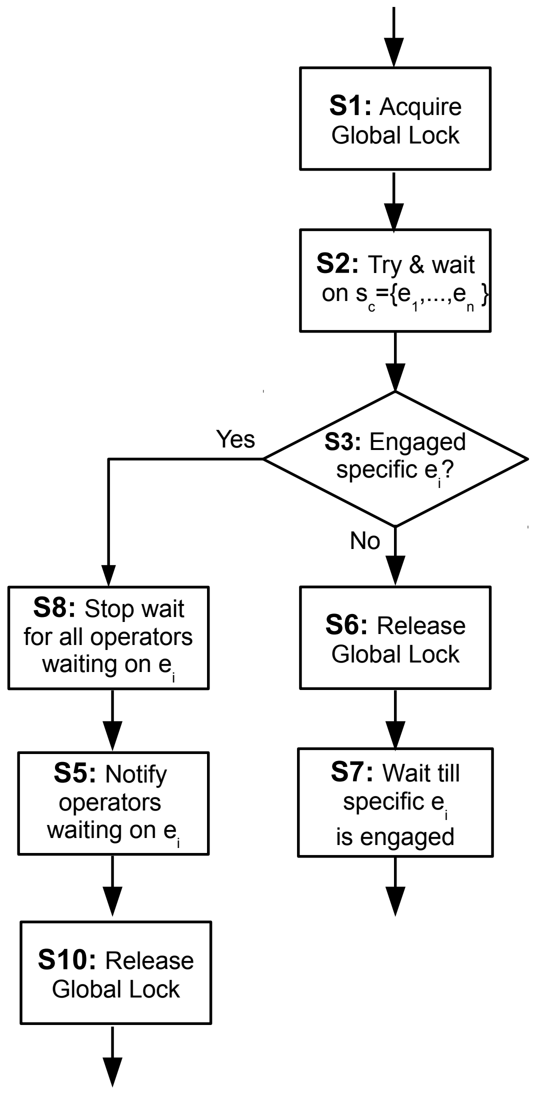
S1: Acquire Global Lock
S2: Try & wait on s_c={e_1,...,e_n}
S3: Engaged specific e_i?
Yes
No
S8: Stop wait for all operators waiting on e_i
S6: Release Global Lock
S5: Notify operators waiting on e_i
S7: Wait till specific e_i is engaged
S10: Release Global Lock

Figure 7.2: Improved Choice Operator Communication

operator is notified. The global lock does not have the "releasing" state any more. The maintenances in previous "ending phase" will be protected in the "enabling" state of the global lock.

The improved mechanism of the operator activities is shown in Figure 7.2. Compared to Figure 7.1, step "S9" is no longer necessary and has been removed. Step "S8" is not on the route if the choice operator is the passive one to be notified by another operator. The step is moved to the route when this choice operator starts the event engagement and becomes the active operator to notify other operators that synchronize the same event. The step "S8" now precedes the step "S5" that sends the notification. It is under the protection of the global lock. As the step "S5" happens after "S8", when the other operators are notified and resumed, the maintenances are already done. They no longer need to acquire the global lock again.

7.3.3 Adapting the Improved Mechanism to the CSP# Operators

An improved cooperation between event objects, choice operators and global lock has been introduced on CSP operator implementation. CSP# bases on classic CSP and provides the shared memory communication in the model. In Section 6.3.3, we have described the solution to combine the shared memory communication can message passing communication in the "general choice" operator. We apply the improved cooperation mechanism CSP# operators to work with the shared memory communications.

Compared the choice operator in classic CSP, the "general choice" operator in CSP# has two extra routes related to the "data-change" event *dc*. The improved cooperation mechanism of the "general choice" operator is shown in Figure 7.3. When the operator has notified the other operators that synchronize to e_i, it checks whether e_i has event-attached program. If it has, the operator does the same maintenances and notifications for the "data-change" event *dc*. The two sets of maintenances and notifications, for the event e_i and *dc*, are safe under the same protection of the global lock.

On the passive route, when the operator is notified by other operator, it checks whether it has engaged a regular event e_i or the "data-change" event *dc*. If it engaged *dc*, the operator needs to go back to beginning, acquiring the global lock before it proceeds.

Other changes on the CSP# operators are on the *buffer* management. The general choice operator needs to have a local buffer if it contains *channel* operations in its event set. When the OS schedules the choice operator that has engaged channel operation, the channel reads the value in the local buffer instead of the channel buffer. As the local buffer interacts with the channel buffer in the protection of the global lock, the behavior of the channel is still consistent with the CSP# semantics.

Unit: millisecond

Loop	2-philosopher			3-philosopher			4-philosopher		
	JCSP	CSP#	Improved	JCSP	CSP#	Improved	JCSP	CSP#	Improved
1,000	136	188	85	212	256	118	275	650	265
10,000	1207	1041	665	1988	2021	977	2650	6637	2347
100,000	12144	9818	5814	19566	20842	9985	26734	62339	20654

Table 7.1: Performance Comparison between JCSP, CSP# Operator and Improved Operator

7.4 Experiment and Performance

We implemented the improved version of the CSP# operators in "PAT.Runtime" library. Using the *dining philosopher* model introduced in Section 2.2, we compare the performance of the CSP operators from the JCSP, the original "PAT.Runtime" and the improved "PAT.Runtime" libraries [3]. The experiments are performed on a PC running Windows 8 Pro 64 bit edition. It has Intel i7-2670QM CPU and 8 GB RAM. The JCSP library version is "jcsp-1.1-rc4". The Java programs are running on JVM of version "1.7.0.21". The C# programs are running on .NET Framework 3.5. The JCSP library and the original "PAT.Runtime" library are using the synchronization mechanism described in Section 7.2. The operators in the improved "PAT.Runtime" library use our improved mechanism in Section 7.3.2.

For the forks and philosophers in the example, each of them is running as a thread in the programs. They synchronize on the "get" and "put" event objects. These event objects are initialized in the "Main()" methods and distributed to the fork and philosopher threads. The "Main()" method starts all the fork and philosopher threads and waits the ends of all these threads. The running time is the duration (in milliseconds) from these threads' starts to the ends of them, as shown in Table 7.1.

The program using JCSP has comparable running time as the one using original "PAT.Runtime" library. When there are fewer threads, the program using of original "PAT.Runtime" runs faster than the one using JCSP. But when the number of threads increases, the program using JCSP has better

[3]All three versions are hand-coded. They only use the CSP or CSP# operators to communicate between threads.

Unit: millisecond

Loop	Two-Thread			Three-Thread			Four-Thread		
	Coded	CSP#	Improved	Coded	CSP#	Improved	Coded	CSP#	Improved
1,000	29	60	36	18	39	24	30	93	67
10,000	138	251	160	161	355	207	351	929	787
100,000	1299	2475	1586	1522	3520	1972	3417	9480	7993

Table 7.2: Performance Comparison between Hand-Coded, CSP# Operator and Improved Operator

performance. This may related to the differences between thread scheduling mechanism in Java virtual machine and .NET Framework. The program using improved “PAT.Runtime” library has the best performance among the three programs. On the 2-philosopher case, it saves about 44% running time as the one using original “PAT.Runtime” library. When the number of threads goes up, the saved time on the communication between threads for the improved mechanism also increases. On the 4-philosopher case, the program using improved library save about 64% running time on average. Compared to the program using JCSP library, the one using improved “PAT.Runtime” shows better performance even when the number of threads is increased. On average, it saves about 45%, 48% and 13% of running time to the one using JCSP for the 2, 3 and 4-philosopher cases.

Table 7.2 compares the performances between the hand-coded program, the program using “PAT.Runtime” and the one using improved “PAT.Runtime”. All three programs implement that multiple threads trying to synchronize with each other on a single event. Only when all the threads wait on this event can they finish the synchronization and go to next loop. On each loop these programs synchronize on the event once. The hand-coded program uses the “monitor” to communicate between threads. The CSP# and the improved CSP# programs use the CSP# event to synchronize.

From the table we can see that the program using improved “PAT.Runtime” library saves much compared to the one using original “PAT.Runtime” library. The percentage of the saved time does not increase but drop in this case. The reason is that the choice operators in this experiment only contains one event, while the choice operators of the “fork” in previous example have more events in the event set. The hand-coded program still has the best performance. Compared to the one using original “PAT.Runtime” library, the running time of the program using improved library is much closer to the hand-coded one.

7.5 Discussion and Summary

For the popular programming languages such as Java and C#, the inter-thread communications are based on shared memory communication. We analyzed the classic solution that implements CSP message passing communication on the shared memory communication in these languages. Based on the synchronization mechanism in this solution, we proposed our improved CSP operators' synchronization mechanism.

In the improved synchronization, the data maintenances on related operators are merged to one operation and it is carried out by the operator that activates the event engagement. With related modifications on the operations of the events and global lock, the original "ending phase" of the engagement is removed to avoid unnecessary mutual exclusions. This improved mechanism is adapted to support the CSP# operator and is implemented in the "PAT.Runtime" library. The experiment results show that the performance of the improved mechanism is much better than the original mechanism.

The improved mechanism uses the active operator to access the data of other operators. This requires the operators' data be shared by the whole program. As the alphabets of the CSP and CSP# models are global already, they can also be safely shared in C# programs. The improved mechanism only needs the operators have one extra variable to caching the data being communicated. Hence, this can be regarded as space-time tradeoff. Similar technique may be extended to apply on the multi-process and network situations.

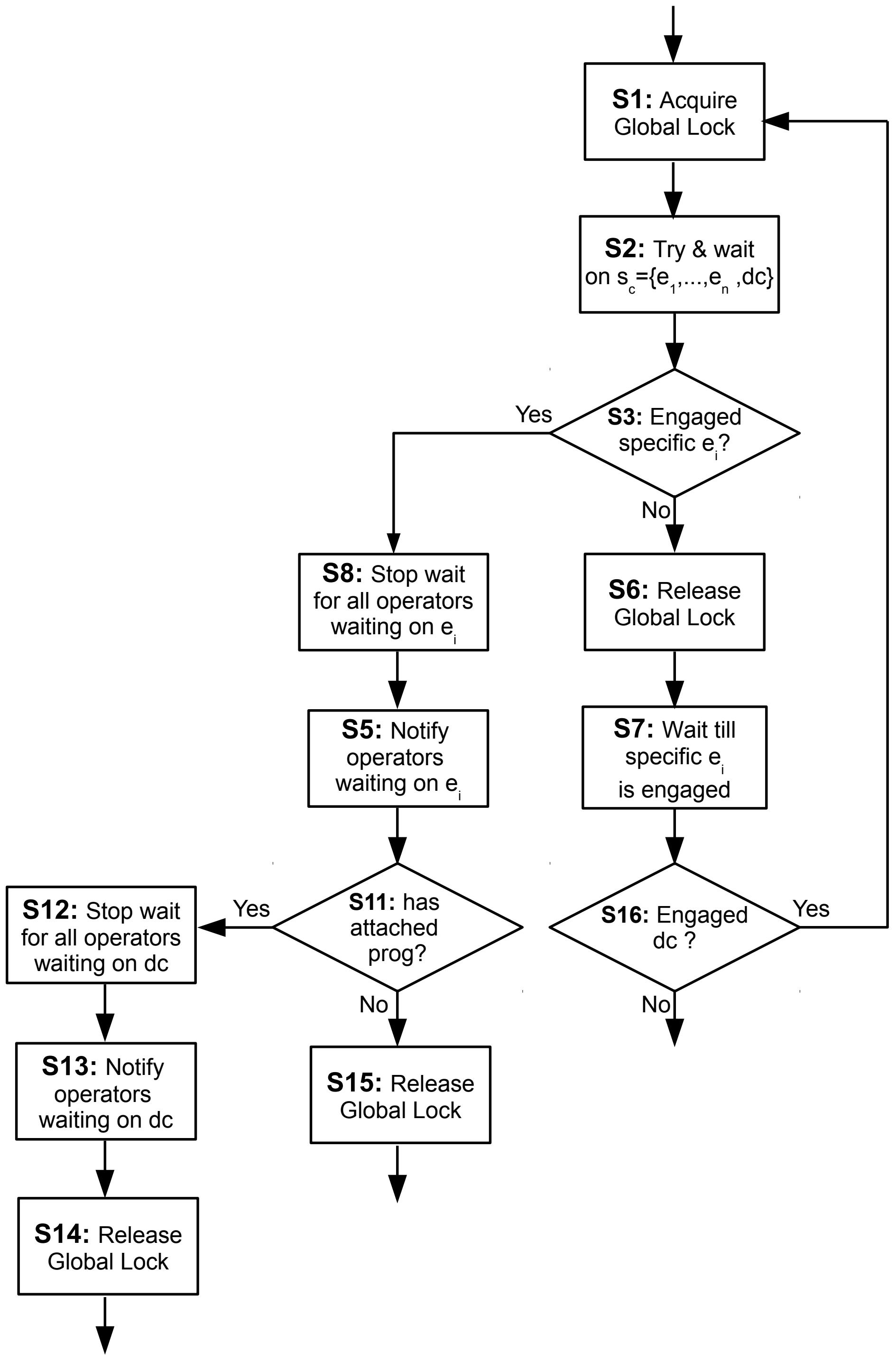
S1: Acquire Global Lock
S2: Try & wait on s_c={e_1,...,e_n ,dc}
S3: Engaged specific e_i?
Yes
No
S8: Stop wait for all operators waiting on e_i
S6: Release Global Lock
S5: Notify operators waiting on e_i
S7: Wait till specific e_i is engaged
S12: Stop wait for all operators waiting on dc
Yes
S11: has attached prog?
S16: Engaged dc ?
Yes
No
No
S13: Notify operators waiting on dc
S15: Release Global Lock
S14: Release Global Lock

Figure 7.3: Improved CSP# Choice Operator Communication

Chapter 8

Conclusion

In this chapter, we first compare our approaches with other related works, then give a summary of our contributions, and finally discuss the possible future extensions based on our existing works.

8.1 Comparison with Related Works

Compared to the related works, our approaches are based on a more feature-rich modeling language CSP#. The translation-based approach tries to make the best of the CSP# features to simplify the translated model. The event-attached program in CSP# provides a flexible container to represent the statements in the C# program, as long as they do not perform synchronization or flow control. With the control of these event-attached programs, our approach allows customized atomicity defined the translated CSP# models. AS CSP# model supports C# program attached to events in the model, the class hierarchy in C# program can remain in the model. Benefit from this, our approach makes minimum changes on the source code.

Our translation-based approach uses the channel communication more often than the Java PathFinder V1. This avoids adding more local or temporal variables that may increase the state size. The flexible boundary between programs and processes allows the abstraction be performed on our translated models, which is not easy to do in JPF. In our approach, the process need to register the local variables on the "Memory" object. As Promela can have local variables defined inside process, JPF can handle the local variables more convenient.

Unlike Bogor that translate the program to the intermediate language BIR [88], our translation based approach does not use intermediate representation between the program and model. In translated CSP# models, the users or other tools can easily recognize the boundary between inter-thread communication and the event-attached program in C#. Our translation tries to extract the concurrent aspect of the programs explicitly and to make them manageable in CSP#. On the other hand, the intermediate language BIR in Bogor provides finer atomicity that is comparable to our VM-based verification approach.

Our VM-based verification approach is tightly integrated in the Mono virtual machine but it does not change the mechanism to execute the IL code in virtual machine. The program running on the customized Mono virtual machine execute the IL code as usual, all the operations related to the VM-based verification is performed after the IL code's execution. The Java PathFinder version 2 adopts the instrument approach that uses one or more Java statements in source code to represent one Java bytecode in the compiled Java programs. These Java source code form an extra layer on the virtual machine. Similar mechanism on IL code level is used in MoonWalker for the C# programs. Compared to the instrument approaches, our tool runs faster on each IL code, or on bigger atomicity levels, such as method level. However, the instrument approaches can capture the changes on the program states more efficiently than our tool. When the program is verified on bigger atomicity levels, our tool is more efficient. When verified on smaller atomicity levels, the instrument approaches have advantages. Another difference is that our tool is running at lower level that can gather the information of the programs and even the native libraries of C# language, which are not supported in Java Pathfinder or MoonWalker.

Compared to CSP, CSP# supports more language features. The CSP# channel supports asynchronous communication and matching on the sent data. The conditional choice operator has the atomic and blocking variants for the shared memory communication. These feature- rich behaviors are encapsulated in the general choice operator for our concurrency class library "PAT.Runtime".

The JCSP and CTJ libraries are based on classic CSP and they do not provide these shared memory communication related features. JCSP has also added the poison concept on the channels to chain terminate the components in the program. At present “PAT.Runtime” library only provides the operators that are defined in CSP#.

In addition, our concurrency library provides process level alphabet management. Even without the code generation tool, the developers can use these management facilities to organize threads in C# program as the processes in CSP# model. When the PAT, the code generation tool and “PAT.Runtime” library work together, the developers can use CSP# from the design to implement phase. Our tool shares the similar goal with CSP++ framework that also provides the automatic code generation. Both approaches use the modeling language to describe the concurrent aspect and implement the functionalities in the user-defined data. However, the user-defined data are imported in CSP# model from the design phase in our approach. Therefore, the properties that are verified on the CSP# model can access both the concurrent and the functionality aspects of the program.

The first implementation of “PAT.Runtime” library uses the same synchronization mechanism on *monitor* as the JCSP library, but our operators additionally support the shared memory communication. In the experiments, the performance of our approach is slightly behind JCSP when the number of thread increases. In the “PAT.Runtime” that adopts the improved synchronization mechanism, although our operators support more concurrent behavior, it uses less communications to achieve the synchronization between threads. The performance of the improved “PAT.Runtime” library is notably ahead of the JCSP.

8.2 Summary of Current Works

Our research focuses on better integrating CSP# with development process, including design, verification and implementation phases. Based on PAT and CSP#, we applied translation-based and VM-based verification approaches for verifying the properties on concurrent C# programs. On the other direction, we provided a C# concurrency library and a tool in PAT to provide automatic generation of C# programs from CSP# models.

The translation-based approach takes advantages on the program friendly features of CSP#. It uses the user-defined data in CSP# model to preserve most the object-orient characteristics. For a C# program, it generates a class library and a CSP# model. The class library keeps all the fields of the classes in original C# program and it is imported in the generated CSP# model. In this way, the CSP# model can create and access the objects in the C# program as the shared variables in the model. The methods of the classes are translated to the processes in the CSP# model. By adjusting the data and model boundary and by controlling the atomicity on the translation, the translated CSP# model can be at different abstraction levels of the original C# program.

The requirements on the C# program are represented as properties on the model. PAT exhaustively traverses the translated CSP# model and verifies the properties on the state space of the model. With customized translation on specific methods or statement, PAT can verify the properties related to probability and expected rewards on the translated model.

In our VM-based verification approach, the execution of a multi-threaded C# program is represented as a LTS system. The state of the LTS contains the program contexts of the threads and the static, dynamic and stack data of the program. One transition is a certain thread in the program executing one or more IL codes and all the other threads remaining at their previous contexts.

The modified Mono virtual machine has a "modelchecking" mode. When the C# program runs in this mode, the virtual machine breaks the program based on configured atomicity, extracts the program state at the end of each transition. With these information extracted, PAT communicates with the Mono virtual machine to control the traverse of the program state space at runtime. The transition can be set to different atomicity levels, such as *IL level* or *Source Code level*. Specific namespaces in the C# program can be filtered out to enhance the efficiency of the verification.

The VM-based verification approach supports deadlock-freeness and safety properties. It uses DFS to traverse the state space of the C# program. Once the state that satisfies the property is found, PAT prints out the transition trace from the start of the program to this state as counterexample.

We choose the *trace* semantics to define the equivalence between CSP# model and the implemented C# program. Based on this equivalence, we developed a concurrency library "PAT.Runtime" to provide the CSP# operators on .NET framework. The shared memory communication and the message passing communication are combined and encapsulated in the "general choice" operator. After importing the "PAT.Runtime" library, the C# program can use the CSP# operator objects to synchronize multiple threads in the same way as in CSP# model.

An automatic code generation tool was developed in PAT framework. From a verified CSP# model, the tool generates a C# program that has the same behavior as the origin CSP# model. The generated C# program contains the communication code that generated from the CSP# model. And it allows non-communication code to inserted between the communication code. Developers can implemented the functionalities in the user-defined data structures as non-communication code, then design the concurrent aspect in CSP# model, and at last use our tool to generate the executable C# program from the user-defined data structures and CSP# model. We structurally proved the equivalence between original CSP# model and its generated C# program, from the operators to the process and model level. Based on the equivalence on trace semantics, the LTL properties validated on the original CSP# model preserve on the generated C# program.

To improve the performance of the "PAT.Runtime" library, we analyze the synchronization mechanism when the CSP operators are implemented on the *monitors*. Based on the existing solution, we merge the alphabet maintenance and the notification for the operator that starts the event engagement. With the related tuning on the communication, the "ending phase" for the event engagement is removed. The improved synchronization is adapted to the CSP# operators that contain both share memory and message passing communications. The experiment results show good performance gains for the improved synchronization mechanism.

The key contributions of this thesis are recapitulated and listed below.

- Our approaches expand the usage of the feature-rich language CSP# on both the design and implement phases. With the tools performing transformations between CSP# and C#, the concurrent properties can be directly verified on the C# programs.
- The first approach defines the translation from the C# program source code to CSP# model. The concurrent aspect of the C# program is explicitly represented in the CSP# model. De-

pending on configuration, the other aspects can be put in the user-defined library or in the model.

- On the Intermediate Language code level, our VM-based verification tool uses the debug interface and the synchronization event in the virtual machine to traverse the C# program state space and verify the deadlock-freeness and safety properties.
- The trace semantics of CSP# is defined on C# program. Based on this trace semantics, we further define the equivalence between CSP# models and C# programs. With this equivalence, the verified properties on traces are preserved in the C# program with the same traces.
- Based on the trace equivalence, the CSP# operators are implemented in C# library "PAT.Runtime". They can be used in C# programs to manage synchronizations between threads. Experiment shows that our operators have better performance than the related CSP library.
- For the transformation from CSP# to C#, we develop the code generation tool in PAT framework. We structurally prove that the generated program is equivalent to original model on the trace semantics.

8.3 Future Works

The CSP# language is evolving to better support the event-attached program and to provide easier manipulation on shared variables. The fields of the objects can be directly accessed in the event-attached programs. The object reference can be used as parameters in process definitions. Local variables are also supported inside the event-attached programs. These new features make the object references are used as the integer and boolean types in CSP#. The translation based approach can redesign the object lists in the "Memory" class which is used to export the program state. The polymorphism can be more naturally represented in the model. With the evolution of CSP#, the program can be further integrated in the model without much predefined or customized statement translations. Obviously, the side effects of the programs shall be constrained, especially on the expressions in the CSP# models.

The modeling languages like CSP# have become friendlier to the useful features in programming languages. On the other hand, the programming languages become extensible to allow the analysis and verification tools, such as the CodeContract for C# [35, 25] and JML for Java [66, 21]. The VM-based verification approach shall more tightly integrate with the programming languages' virtual machine. It is convenient to use extension of the language, such as the annotation in C#, to embed the verification related information in the debug versions of the programs. When the programs are running in "modelchecking" mode these information can help to decide the atomicity, extract the program state and apply abstraction of the program.

Currently, both the "PAT.Runtime" library and the C# code generation tool are based on multi-threaded program running on single machine. One possible extension is to adapt the library to support multi-process in network environment. The global environment and the monitor-based communication will have to be redesigned. In the network, a special process can be set up to manage the alphabets for events and channels, as the "Glo" class in multi-threaded programs.

In this thesis, we use C# as the typical Object-Oriented programming language. Other similar languages, such as Java, can be applied the same approaches, with slight modifications and adaptions. PAT and CSP# have already provided the interfaces for the other programming languages, to implement user-defined data in the models. Our approaches emphasize on the defined semantics equivalence between the models and the programs. The approaches also maneuver the boundary of the user-defined data and the CSP# processes to gain flexible and concise transformations. These methods and techniques are transferrable to other scenarios.

Bibliography

[1] G. Agha. An overview of actor languages. In *OOPWORK: Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming: Yorktown Heights, New York, United States*, volume 9, pages 58–67, 1986.

[2] P. Amey and B. Dobbing. High integrity ravenscar. *Reliable Software TechnologiesąłAda-Europe 2003*, pages 637–637, 2003.

[3] E. C. M. Association et al. Standard ECMA-334: C# Language Specification, 2005.

[4] J. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2):131–146, 2005.

[5] C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[6] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, and S. Qadeer. Preemption Sealing for Efficient Concurrency Testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 420–434, 2010.

[7] T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Technical Report 2000-14, Microsoft Research, 2000.

[8] T. Ball and S. Rajamani. The SLAM toolkit. In *Computer aided verification*, pages 260–264. Springer, 2001.

[9] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. *ACM SIGPLAN Notices*, 37(1):1–3, 2002.

[10] L. Baresi, V. Rafe, A. Rahmani, and P. Spoletini. An efficient solution for model checking graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 213(1):3–21, 2008.

[11] J. Barnes. *High integrity software: the SPARK approach to safety and security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] G. Barrett. Model checking in practice: The t9000 virtual channel processor. *Software Engineering, IEEE Transactions on*, 21(2):69–78, 1995.

[13] R. Bellman. A Markovian Decision Process. Technical report, DTIC Document, 1957.

[14] M. Ben-Ari. *Principles of the Spin model checker*. Springer-Verlag New York Inc, 2008.

[15] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):505–525, 2007.

[16] N. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. *Communicating Process Architectures 2007: WoTUG-30*, pages 183–205, 2007.

[17] N. Brown and P. Welch. An introduction to the Kent C++ CSP Library. *Communicating Process Architectures*, 2003:139–156, 2003.

[18] N. C. Brown. C++ CSP networked. *Communicating Process Architectures*, 2004:185–200, 2004.

[19] N. H. A. D. Brugh, V. Y. Nguyen, and T. C. Ruys. MoonWalker: Verification of .NET Programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, ETAPS 2009*.

[20] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1066–1071. IEEE, 2011.

[21] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERPąŕ02), Las Vegas, Nevada, USA*, pages 322–328, 2002.

[22] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[23] L. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[24] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, H. Zheng, et al. Bandera: Extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[25] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[26] F. de Assis, N. Maruyama, and F. Takase. MODEL CHECKING A ROV REACTIVE CONTROL ARCHITECTURE. *ABCM Symposium Series in Mechatronics*, 4:683–692, 2010.

[27] C. Demartini, R. Iosif, and R. Sisto. Modeling and validation of Java multithreading applications using SPIN. In *Proceedings of the 4th SPIN Workshop*, 1998.

[28] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.

[29] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. *Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, 1999.

[30] D. Detlefs, G. Nelson, and J. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[31] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *ACM SIGAda Ada Letters*, volume 18, pages 1–6. ACM, 1998.

[32] M. Dwyer, J. Hatcliff, et al. Bogor: an extensible and highly-modular software model checking framework. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 267–276. ACM, 2003.

[33] M. Dwyer, J. Hatcliff, and M. Hoosier. Building your own software model checker using the bogor extensible model checking framework. In *Computer Aided Verification*, pages 227–238. Springer, 2005.

[34] I. East, J. Martin, P. Welch, D. Duce, and M. Green. gCSP: a graphical tool for designing CSP systems. *Communicating Process Architectures 2004*, 27:233, 2004.

[35] M. Fahndrich, M. Barnett, D. Leijen, and F. Logozzo. Integrating a set of contract checking tools into visual studio. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 43–48. IEEE, 2012.

[36] L. Freitas. *JACK: A process algebra implementation in Java*. PhD thesis, Centro de Informatica, Universidade Federal de Pernambuco, 2002.

[37] W. Gardner. *CSP++: An object-oriented application framework for software synthesis from CSP specifications*. PhD thesis, Politecnico di Milano, Italy, 2000.

[38] W. Gardner. Bridging csp and c++ with selective formalism and executable specifications. In *Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*, pages 237–245. IEEE, 2003.

[39] W. Gardner. CSP++: How Faithful to CSPm. *Proc. Communicating Process Architectures 2005 (WoTUG-27)*, pages 129–146, 2005.

[40] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 12–21. ACM, 2002.

[41] K. Havelund. Java PathFinder, a translator from Java to Promela. *Lecture notes in computer science*, pages 152–152, 1999.

[42] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366 – 381, 2000.

[43] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. *Model Checking Software*, pages 624–624, 2003.

[44] G. Hilderink, A. Bakkers, and J. Broenink. A distributed Real-Time Java system based on CSP. In *Object-Oriented Real-Time Distributed Computing, 2000.(ISORC 2000) Proceedings. Third IEEE International Symposium on*, pages 400–407. IEEE, 2000.

[45] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer, 2006.

[46] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science.

[47] C. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[48] C. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[49] G. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[50] G. Holzmann. Logic verification of ANSI-C code with SPIN. *SPIN Model Checking and Software Verification*, pages 131–147, 2000.

[51] G. Holzmann. The SPIN Model Checker: Primer and Reference Manual. *Reading: Addison-Wesley*, 2003.

[52] G. Holzmann and M. H Smith. Software model checking: extracting verification models from source code? *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

[53] G. J. Holzmann. Design and Validation of Protocols. *Tutorial Computer Networks and ISDN Systems*, 25:981–1017, 1990.

[54] J. H. Howard. Proving monitors. *Commun. ACM*, 19(5):273–279, May 1976.

[55] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[56] G. Jones. *Programming in Occam*. Prentice-Hall International London, 1986.

[57] G. Kassel and G. Smith. Model checking Object-Z classes: Some experiments with FDR. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 445–452. IEEE, 2001.

[58] C. Keller, D. Saha, S. Basu, and S. Smolka. FocusCheck: A tool for model checking and debugging sequential C programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 563–569, 2005.

[59] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[60] M. Kleine. Using CSP for Software Verification. In *Proceedings of Formal Methods 2009 Doctoral Symposium, Eindhoven University of Technology*, pages 8–13, 2009.

[61] M. Kleine. CSP as a Coordination Language. In *Coordination Models and Languages*, pages 65–79. Springer, 2011.

[62] M. Kleine, B. Bartels, T. Göthel, S. Helke, and D. Prenzel. LLVM2CSP: extracting csp models from concurrent programs. *NASA Formal Methods*, pages 500–505, 2011.

[63] M. Kleine and S. Helke. Low-level code verification based on CSP models. *Formal Methods: Foundations and Applications*, pages 266–281, 2009.

[64] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101–150, 2002.

[65] J. Lawrence. Practical application of CSP and FDR to software design. *Communicating Sequential Processes. The First 25 Years*, pages 717–721, 2005.

[66] G. T. Leavens and Y. Cheon. Design by Contract with JML. *Draft, available from jmlspecs. org*, 2006.

[67] A. Lehmberg and M. Olsen. An introduction to CSP.NET. *Communicating Process Architectures*, 2006:13–30, 2006.

[68] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. *Model Checking Software*, pages 80–102, 2001.

[69] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[70] Y. Liu, J. Sun, and J. S. Dong. Scalable multi-core model checking fairness enhanced systems. In K. Breitman and A. Cavalcanti, editors, *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 426–445. Springer, 2009.

[71] Y. Liu, J. Sun, and J. S. Dong. PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In *ISSRE*, pages 190–199, 2011.

[72] B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Proceedings of the 20th international conference on Software engineering(ICSE'98)*, pages 95–104. IEEE Computer Society, 1998.

[73] B. Mahony and J. S. Dong. Timed communicating object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[74] B. Mahony and J. S. Dong. Deep semantic links of tcsp and object-z: Tcoz approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

[75] A. I. McInnes. Using csp to model and analyze tinyos applications. In *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, pages 79–88. IEEE, 2009.

[76] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. *Model Checking Software*, pages 902–902, 2005.

[77] A. Mota and A. Sampaio. Model-checking csp-z: strategy, tool support and industrial application. *Science of Computer Programming*, 40(1):59–96, 2001.

[78] M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *ACM SIGPLAN Notices*, volume 43, pages 362–371. ACM, 2008.

[79] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. *Redmond: Microsoft Research Technical Report, MSRTR-2007-149*, 2007.

[80] W. B. Nelson. *Accelerated Testing: Statistical Models, Test Plans, and Data Analysis*, volume 344. Wiley-Interscience, 2009.

[81] V. Y. Nguyen and T. C. Ruys. Memoised garbage collection for software model checking. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, pages 201–214. Springer-Verlag, 2009.

[82] K. W. Nielsen. Task coupling and cohesion in ada. *ACM SIGAda Ada Letters*, 6(4):44–52, 1986.

[83] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Making Java PathFinder Cooperate with Behavior Protocol Checker. Technical report, Citeseer, 2006.

[84] C. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pages 83–90. IEEE, 2009.

[85] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[86] C. PĺčsĺČreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26. ACM, 2008.

[87] V. Rafe and A. Rahmani. Towards automated software model checking using graph transformation systems and Bogor. *Journal of Zhejiang University-Science A*, 10(8):1093–1105, 2009.

[88] M. Robby, M. Dwyer, and J. Hatcliff. Bogor: A flexible framework for creating software model checkers. 2006.

[89] A. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Computer Security Foundations Workshop, 1995. Proceedings., Eighth IEEE*, pages 98–107. IEEE, 1995.

[90] A. Roscoe et al. Model-checking CSP. *A Classical Mind, Essays in Honour of CAR Hoare. Prentice-Hall*, pages 353–378, 1994.

[91] A. Roscoe and C. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988.

[92] A. Roscoe, C. Hoare, and R. Bird. *The theory and practice of concurrency*, volume 169. Prentice Hall Engelwood Cliffs, NJ, 1997.

[93] A. Saifhashemi and P. A. Beerel. SystemVerilogCSP: Modeling Digital Asynchronous Circuits Using SystemVerilog Interfaces. *CPA-2011: WoTUG-33*, pages 287–302, 2011.

[94] B. Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, University of Oxford, 1998.

[95] N. Schaller, G. Hilderink, and P. Welch. Using Java for Parallel Computing: JCSP versus CTJ, a Comparison. *Communicating Process Architectures*, pages 205–226, 2000.

[96] B. Schlich and S. Kowalewski. Model Checking C Source Code for Embedded Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(3):187 – 202, 2009.

[97] S. Schneider. Verifying authentication protocols in CSP. *Software Engineering, IEEE Transactions on*, 24(9):741–758, 1998.

[98] S. Schneider and H. Treharne. Verifying controlled components. In *Integrated Formal Methods*, pages 87–107. Springer, 2004.

[99] S. Song. An Efficient Method of Probabilistic Model Checking. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 24–25. IEEE, 2010.

[100] M. Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley Professional, 2012.

[101] J. Sun, Y. Liu, J. Dong, and H. Wang. Specifying and verifying event-based fairness enhanced systems. *Formal Methods and Software Engineering*, pages 5–24, 2008.

[102] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating Specification and Programs for System Modeling and Verification. In *Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, pages 127–135, 2009.

[103] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[104] J. Sun, S. Song, and Y. Liu. Model Checking Hierarchical Probabilistic Systems. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2010.

[105] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In *CONCURą́r99 Concurrency Theory*, pages 46–65. Springer, 1999.

[106] B. Vinter, J. Bjørndalen, and O. Anshus. PyCSP-Communicating Sequential Processes for Python. 2007.

[107] W. Visser, C. PĺčsĺČreanu, and S. Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[108] P. Welch, N. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and extending JCSP. *Communicating Process Architectures 2007*, 65:349–370, 2007.

[109] P. Welch and J. Martin. A CSP Model for Java Threads (and Vice-Versa).

[110] P. Welch and J. Martin. Formal analysis of concurrent java systems. *Communicating Process Architectures*, 58:275–301, 2000.

[111] L. Yang and M. Poppleton. JCSProB: Implementing Integrated Formal Specifications in Concurrent Java. *Communicating Process Architectures*, 65:67–88, 2007.

[112] L. Yang and M. Poppleton. Java implementation platform for the integrated state-and event-based specification in PROB. *Concurrency and Computation: Practice and Experience*, 22(8):1007–1022, 2010.

[113] H. Yolkcnnk, G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. CSP Design Model and Tool Support. In *Communicating Process Architectures 2000: WoTUG-23: Proceedings of the 23rd World Occam and Transputer User Group Technical Meeting: 10-13 September 2000, Canterbury, United Kingdom*, volume 30, page 33. Ios PressInc, 2000.

[114] M. Zheng, D. Sanán, J. Sun, Y. Liu, J. S. Dong, and Y. Gu. State space reduction for sensor networks using two-level partial order reduction. In *Verification, Model Checking, and Abstract Interpretation*, pages 515–535. Springer, 2013.

[115] M. Zheng, J. Sun, Y. Liu, J. S. Dong, and Y. Gu. Towards a model checker for nesc and wireless sensor networks. In *Proceedings of the 13th international conference on Formal methods and software engineering*, pages 372–387. Springer-Verlag, 2011.

Appendix A

Translation-based Approach Examples

A.1 The translated CSP# model of Dining Philosophers example

The translated CSP# model in Section 4.5 is as follows:

```
#import "DiningPhilCls − r";

var < Memory > memory;
var cpid;
channel create_thread 0;

CreateNewThread() = create_thread?ti.pid.obj →
   if(−1 == ti){Skip} else {NewThread(ti, pid, obj)};

NewThread(ti, pid, obj) =
   case{
   (ti == 1) : Philosopher_run(pid, obj)
   default : Skip
   } || CreateNewThread();

Fork_Lock(pid, obj) =
   [−1 == memory.Fork_Get_LOCK(obj)]tau{memory.Fork_Set_LOCK(obj, pid); } → Skip;
```

```
Fork_Unlock(pid, obj) =
   assert(memory.Fork_Get_LOCK(obj) == pid);
   ((tau{memory.Fork_Set_LOCK(obj, −1); } → Skip)
   );
Philosopher_CreateObj(pid, le, ri, na) =
   ((tau{memory.reg(pid, 2); } → Skip);
   ((tau{memory.SetTP(pid, 0, memory.Philosopher_CreateObj(le, ri, na)); } → Skip);
   ((tau{memory.SetTP(pid, 1, memory.GetNextPid()); } → Skip);
   ((create_thread!1.(memory.GetTP(pid, 1)).(memory.GetTP(pid, 0)) → Skip);
   ((tau{memory.unreg(pid); } → Skip)
   )))));
Philosopher_run(pid, obj) =
   (Fork_Lock(pid, memory.Philosopher_Get_left(obj));
   (Fork_Lock(pid, memory.Philosopher_Get_right(obj));
   (Fork_Unlock(pid, memory.Philosopher_Get_right(obj));
   (Fork_Unlock(pid, memory.Philosopher_Get_left(obj))
   ))));

DiningPhil_Main(pid) =
   ((tau{memory.DiningPhil_Set_i(0); } → Skip);
   (DiningPhil_Main_For(pid);
   ((tau{memory.DiningPhil_Set_i(0); } → Skip);
   (DiningPhil_Main_For_1(pid);
   ((create_thread!(−1).0.0 → Skip)
   )))));

DiningPhil_Main_For(pid) =
   ((tau{memory.DiningPhil_Set_forks(memory.DiningPhil_Get_i(),
      memory.Fork_CreateObj()); } → Skip);
   ((tau{memory.DiningPhil_Set_i(memory.DiningPhil_Get_i() + 1); } → Skip);
   ((if(memory.DiningPhil_Get_i() < memory.DiningPhil_Get_N())
      {DiningPhil_Main_For(pid)}else{Skip})
   )));
```

```
DiningPhil_Main_For_1(pid) =
   (Philosopher_CreateObj(pid, memory.DiningPhil_Get_forks(memory.DiningPhil_Get_i()),
      memory.DiningPhil_Get_forks((memory.DiningPhil_Get_i() + 1)
      % memory.DiningPhil_Get_N()), memory.DiningPhil_Get_i());
   ((tau{memory.DiningPhil_Set_i(memory.DiningPhil_Get_i() + 1); } → Skip);
   ((if(memory.DiningPhil_Get_i() < memory.DiningPhil_Get_N())
      {DiningPhil_Main_For_1(pid)} else {Skip})
   )));
System() = DiningPhil_Main(0) || CreateNewThread();
#assert System() deadlockfree;
```

A.2 The translated “Philosopher” and “PhilosopherCls”

After the translation, the classes “Philosopher” and “PhilosopherCls” classes in the class library contain the data of the “Philosopher” class in the original C# program.

```
public class Philosopher
{
  public int LOCK = −1;
  public int WAITING;
  public int name;
  public int left;
  public int right;
  public string ID {
    get {
      StringBuilder sb = new StringBuilder("[");
      sb.Append(LOCK.ToString() + ',');
      sb.Append(WAITING.ToString() + ',');
      sb.Append(name.ToString() + ',');
      sb.Append(left.ToString() + ',');
      sb.Append(right.ToString() + ',');
      return sb.ToString().TrimEnd(',') + "]";
    }
  }
  public Philosopher GetClone() {
    Philosopher p = new Philosopher();
    p.LOCK = this.LOCK;
    p.WAITING = this.WAITING;
    p.name = this.name;
    p.left = this.left;
    p.right = this.right;
    return p;
  }
}
```

```
public class PhilosopherCls
{
  List < Philosopher >  list = newList < Philosopher > ();
  public int CreateObj(int left, int right, int name){
  Philosopher obj = new Philosopher();
    obj.left = left;
    obj.right = right;
    obj.name = name;
    list.Add(obj);
    return PcMacro.create_object(PcConst.Philosopher, list.Count − 1);
  }
  public Philosopher GetObj(int obj){
    return list[PcMacro.get_index(obj)];
  }
  public string ID {
    get {
      StringBuilder sb = new StringBuilder("[");
      foreach(Philosopher el in list)
        sb.Append('[' + el.ID + "],");
      return sb.ToString().TrimEnd(',') + "]";
    }
  }
  public PhilosopherCls GetClone() {
    PhilosopherCls pc = new PhilosopherCls();
    foreach(Philosopher s in this.list)
      pc.list.Add(s.GetClone());
    return pc;
  }
}
```

A.3 The C# Program of Leader Election Algorithm

```
public class AsyRingLeaderPri {
  public static void Main() {
    ProcessorPri.N = 4;
    ProcessorPri[] pro = new ProcessorPri[4];
    for(int i = 0; i < 4; i++) {
      pro[i] = new ProcessorPri();
    }
    for(int i = 0; i < 4 - 1; i++) {
      pro[i].next = pro[i+1];
    }
    pro[3].next = pro[0];
    for(int i = 0; i < 4; i++) {
      new Thread(new ThreadStart(pro[i].run)).Start();
    }
  }
}
public class ProcessorPri {
  public static Random rdm = new Random();
  public static object prolock = new object();
  public static int N;
  public ProcessorPri next;
  public int c1; // counter
  public int s1; // state
  public int p1; // preference
  public int receive1; // variable for received
  public int sent1; // variable for sent
  public ProcessorPri() {
    c1 = 0;
    s1 = 0;
    p1 = 0;
    receive1 = 0;
    sent1 = 0;
  }
  public void run() {
    while(s1 != 4) {
      SelfSend();
      Thread.Sleep(0);
    }
    if(s1 == 4) {
      Console.WriteLine("Done");
    } else {
      Console.WriteLine("not reach here ");
    }
  }
```

```
public void Repick()
{ p1 = ProcessorPri.rdm.Next(2);  }
public void SelfSend() {
  lock(prolock) {
  switch(s1) {
  case 0 :
    Repick();
    s1 = 1;
    break;
  case 1 :
    if(0 == sent1) {
      // [p12]
      if(0 == SendPreference())
        sent1 = 1;
    }
    break;
  case 2 :
    if(0 == sent1) {
      // [p12]
      if(0 == SendPreference()) {
        sent1 = 1;
        p1 = 0;
      }
    } else if(1 == sent1) {
      if(1 == receive1) {
        // [c12]
        if(0 == SendCounter())
          sent1 = 2;
      } else if (2 == receive1) {
        // [c12]
        if(0 == SendCounter()) {
          s1 = 0;
          p1 = 0;
          c1 = 0;
          sent1 = 0;
          receive1 = 0;
        }
      }
    }
    break;
    case 3 :
      if( (receive1 > 0) && (sent1 == 0) ) {
        // [p12]
        if(0 == SendPreference()) {
          sent1 = 1;
          p1 = 0;
        }
      } else if((receive1 == 2) && (sent1 == 1)) {
        // [c12]
        if(0 == SendCounter()) {
          s1 = 3;
          p1 = 0;
          c1 = 0;
          sent1 = 0;
          receive1 = 0;
        }
      }
      break;
    default :
      break;
    }//switch
    }//lock
}
public int SendPreference()
{
    int ret = −1;
    switch(next.s1) {
    case 1 :
      if(0 == next.receive1) {
        if ( !( (next.p1 == 0) && (p1 == 1) ) ) {
          next.s1 = 2;
          next.receive1 = 1;
          ret = 0;
        } else if ( (next.p1 == 0) && (p1 == 1) ) {
          next.s1 = 3;
          next.receive1 = 1;
          ret = 0;
        }
      }
      break;
```

```
  case 3 :
    if(next.receive1 == 0) {
      next.p1 = p1;
      next.receive1 = 1;
      ret = 0;
    }
    break;
  default :
    break;
  }//switch
  return ret;
}
public int SendCounter()
{
  int ret = −1;
  switch(next.s1) {
  case 2 :
    if(1 == next.receive1) {
      if(next.sent1 < 2) {
        next.receive1 = 2;
        ret = 0;
      } else if (next.sent1 == 2) {
        if(c1 == N − 1) {
          next.s1 = 4;
          next.p1 = 0;
          next.c1 = 0;
          next.sent1 = 0;
          next.receive1 = 0;
          ret = 0;
        } else if (c1 < N − 1) {
          next.s1 = 0;
          next.p1 = 0;
          next.c1 = 0;
          next.sent1 = 0;
          next.receive1 = 0;
          ret = 0;
        }
      }
    }
    break;
    case 3 :
      if ((next.receive1 == 1) && (c1 < N − 1)) {
        next.c1 = c1 + 1;
        next.receive1 = 2;
        ret = 0;
      }
      break;
    default :
      break;
    }
    return ret;
  }
}
```

Appendix B

The Turn-based Game Example

The turn-based game is designed in CSP#, then it is manually implemented in C# with “PAT.Runtime” library. The CSP# model is as follows:

$\#define\ M\ 3;$
$channel\ stoc[M]\ 0;$
$channel\ ctos[M]\ 0;$
$var\ num\ =\ 0;$
$TbClientSend(i)\ =\ ctos[i]!i \rightarrow edNextRound \rightarrow TbClientSend(i);$
$TbClientReceive(i)\ =\ stoc[i]?i \rightarrow edNextRound \rightarrow TbClientReceive(i);$
$TbClient(i)\ =\ TbClientSend(i)\ ||\ TbClientReceive(i);$
$TbAllClient()\ =\ ||\ x : \{0..M-1\}@TbClient(x);$
$TbServerSendRd(i)\ =$
$\quad edHalfRound \rightarrow stoc[i]!i \rightarrow edRoundEnd \rightarrow TbServerSendRd(i);$
$TbServerReceiveRd(i)\ =$
$\quad ctos[i]?i \rightarrow enqueue.i\{num\ =\ num\ +\ 1\}$
$\quad \rightarrow edHalfRound \rightarrow TbServerReceiveRd(i);$
$TbServerRd(i)\ =\ TbServerSendRd(i)\ ||\ TbServerReceiveRd(i);$
$TbServerRoundGuard()\ =\ [num >= M]\ ($
$\qquad tau \rightarrow dequeue\{num\ =\ 0\} \rightarrow edHalfRound$
$\qquad \rightarrow edRoundEnd \rightarrow TbServerRoundGuard()$
$\quad);$
$TbServer()\ =\ TbServerRoundGuard()\ ||\ (||\ x : \{0..M-1\}@TbServerRd(x));$
$System()\ =\ TbAllClient()\ ||\ TbServer();$

After the model was verified, we manually implemented the model in MS Visual Studio 2010. The C# classes are implemented following their model in CSP#.

For the client side, the class "TbClientSend" implements the process *TbClientSend*. The "run()" method of class "TbClientSend" is as follows.

```
chSend.write(client.Action);
edNextRound.sync();
```

Here the "chSend" is the channel *ctos* for the client to send its action to the server.

Similarly, the "run()" method of class "TbClientReceive" is as follows.

```
chReceive.read();
edNextRound.sync();
```

The "chReceive" repesents the channel *stoc* for the server to send other players' actions to each player.

The server side has the counterparts of the client's "send" and "receive" threads. For these server side threads, the communications on the channels are similar to the ones on client side. The server has the action queue to store the received players' actions. In the model, only the length of this queue is in the global shared variables. In the program, the threads need to get exclusive lock before writing on the queue.

As in Section 6.6.1 we have list the "run()" methods of the "TbServerReceiveRd" and "TbServerRoundGuard", only the "run()" of "TbServerSendRd" is listed as follows.

```
edReveal.sync();
chSend.write(server.RoundResult);
edNextRound.sync();
```

Here the "edReveal" is the *edHalfRound* event in the CSP# model, the "edNextRound" is the *edRoundEnd* event. The server's "RoundResult" will not change until the next round start. So here the access to "RoundResult"need not to grant extra lock.

Appendix C

The Concurrent Accumulator Example

The following is the CSP# model used for the Concurrent Accumulator example in the Section 6.6.2.

```
#define M 2;
var writing = false;
var noOfReading = 0;
var⟨Ldata⟩ ldata;
var len = 0;
var cur[3] = [0, 0, 0];
var gap[3] = [1, 2, 4];
var accu[3] = [0, 0, 0];
var cnt[3] = [0, 0, 0];
ReaderHead(i) = [noOfReading < M && !writing]startR.i
   → startreadop.i{noOfReading = noOfReading + 1; }
   → startRout.i → Skip;
ReaderTail(i) = [noOfReading > 0]endR.i
   → stopreadop.i{noOfReading = noOfReading − 1; }
   → endRout.i → Skip;
Reader(i) = ReaderHead(i); ReaderTail(i); Reader(i);

Writer(i) = [noOfReading == 0 && !writing]startW.i
   → startwriteop.i{writing = true; } → startWout.i
   → endW.i → stopwriteop.i{writing = false}
   → endWout.i → Writer(i);
```

```
Controller() = (startR.0 → startRout.0 → Controller())
  [] (endR.0 → endRout.0 → Controller())
  [] (startW.0 → startWout.0 → Controller())
  [] (endW.0 → endWout.0 → Controller())
...
RWReaders() = Reader(0) || Reader(1) || Reader(2);
RWWriters() = Writer(0) || Writer(1) || Writer(2);

ReadersWriters() = RWReaders() || RWWriters() || Controller();

GAdder(i) = if(cur[i] < len && cnt[i] < gap[i]) {
  startR.i
   → s1{accu[i] = accu[i] + ldata.get(cur[i])}
   → s2{cur[i] = cur[i] + 1; cnt[i] = cnt[i] + 1}
   → endR.i → GAdder(i)
};

GWrite(i) = startW.i
   → s3{accu[i] = accu[i] + ldata.get(len); }
   → s4{ldata.set(len, accu[i]); }
   → s5{accu[i] = 0; cnt[i] = 0; }
   → endW.i → Skip;

LAdder(i) = if(cur[i] < len) GAdder(i); GWrite(i); LAdder(i);
Adder(i) = addstart{cur[i] = 0; cnt[i] = 0; } → LAdder(i);
PalAdd() = Adder(0) || Adder(1) || Adder(2);
OutPrint() = pend{ldata.print()} → Skip;
ThreeAdd() = PalAdd(); OutPrint();
RealProg() = ThreeAdd() || ReadersWriters();
Prog() = pstart{
 ldata.init3();
 len = ldata.Len() − 1
} → RealProg();
#define exclusive !(writing == true && noOfReading > 0);
#assert ReadersWriters() |= [] exclusive;
#define someonereading noOfReading > 0;
#assert ReadersWriters() |= [] <> someonereading;
#define someonewriting writing == true;
#assert ReadersWriters() |= [] <> someonewriting;
```

In the above the "M" is the maximum number of reader that can grant the "reader" lock. The "ldata" is an instance of the user-defined data structure. Other variable definitions refer to Section 6.6.2. All the properties have been verified by PAT tool.

Appendix D

Different Atomicity on CSP# Model and Generated C# Program

In Chapter 6, the generated C# program preserves the properties on the trace semantics of CSP# model. However, the properties related to propositions, defined on the shared variables, may not preserve in the program. The reason is that CSP# takes the event-attached programs as atomic operations. In the C# program, these event-attached programs cannot be considered as atomic. Let us consider the following CSP# model.

$$\begin{array}{l} \mathit{Var}\ a\ =\ 0; \\ P()\ =\ e1\{a\ =\ a\ +\ 1;\ \ a\ =\ a\ -\ 1\} \rightarrow e2 \rightarrow P(); \\ \#\mathit{define}\ \mathit{err}\ (a\ ==\ 1); \\ \#\mathit{assert}\ P()\ \mathit{reaches}\ \mathit{err}; \end{array}$$

Verifying the model in PAT, the model “P” never reach the state “err”, which means the variable “a” is equal to “1”. When we generate a C# program from this model, obviously we cannot say the program never reach “(a == 1)”. After executing the first statement (i.e. “a = a + 1;”) in the attached program of event “e1”, the program does reach the state that satisfies “(a == 1)”.

Appendix E

Using PAT.Runtime in C# Program

"PAT.Runtime" provides the communication between C# threads. It has simpler interface and its semantics is more concise. The developers can use it in place of the C# threading package. In this section, we will discuss the CSP# operators provided by "PAT.Runtime" and how to use them in C# program to control concurrency.

"PAT.Runtime" defines an interface "CSProcess" to represent the CSP# *process*. The operators in definition of CSP# *process* will be put in the "run()" method of the CSPProcess. A CSPProcess object "pP" can be run as a thread or as part of a thread by running "pP.run()". It can also be in a composition of other CSPProcess. The "run()" method taks no parameters and does not have return value.

The two primitive CSP# processes, *Stop* and *Skip*, are built into the PAT.Runtime library. They are used as the normal processes. For example, the statement to perform *Skip* is

$new\ Skip().run();$

A CSP# *event* is represented as a protected C# class *EventBase* in the PAT.Runtime library. When the program is about to engage an event, it does not directly access the EventBase object. Instead it gets an *EventDock* object of this event. These docks are created before the thread starts and they synchronize each other via an EventBase.

For an event e to be synchronized by n threads, it shall create n EventDock objects, one for each thread. These n EventDock objects are created by calling

$$EventDock[]\ eds\ =\ EventDock.create(n);$$

The program shall manually assign the n EventDock objects to the n threads.

In these n threads' "run()" methods, the following statement will engage the event, where ed is one of its EventDock objects.

$$ed.sync();$$

If m new threads start later and they are also synchronizing on event e, we use one of the existing EventDock of e (such as $eds[0]$) to create m new EventDock objects as follows:

$$EventDock[]\ new\_eds\ =\ eds[0].expand(m);$$

After the expansion, the event e is synchronized by $n+m$ threads. When these m threads are about to terminate, the event e needs to contract on these m EventDock objects as follows:

$$eds[0].contract(new\_eds);$$

After being contracted for m, the event e is synchronized by n threads as before.

The event expansion and contraction are commonly used before and after the *parallel* operator.

The CSP# *channel* is implemented in PAT.Runtime library with *One2OneChannel, One2AnyChannel, Any2OneChannel* and *Any2AnyChannel.* They vary on how many *write* ends and *read* ends they can provide. So the *One2AnyChannel* means it has one *write* end and multiple *read* ends. The most general *Any2AnyChannel* can be created as follows.

$$Any2AnyChannel\ ch\ =\ Channel.any2any();$$

The process can get the channel's *write* (or *read*) end to write (or read) objects to (or from) the channel.

$$\begin{array}{l} \mathit{ChannelInput}\ \mathit{chin}1\ =\ \mathit{ch.chin}(); \\ \mathit{ChannelOutput}\ \mathit{chout}1\ =\ \mathit{ch.chout}(); \end{array}$$

Reading and writing on the channel ends is as follows.

$$\begin{array}{l} \mathit{chin}1.\mathit{read}(); \\ \mathit{chout}1.\mathit{write}(\mathit{obj}); \end{array}$$

The *choice* operator [] is implemented as the *Choice* class. The first events of the possible branches are used as the parameters to create a *Choice* object. After that, calling the "select()" method starts the engagement on the *choice* operator. When the "select()" returns, the chosen event has already finished its synchronization. The return value indicates which branch the *choice* has chosen. Based on it, the program shall go to run the rest of statements in the chosen branch.

If none of the events in the choice is enabled, the "select()" method blocks the thread. When one or more events in the *choice* become enabled, the thread is notified. The "select()" will choose one of the enabled events and return the index of it.

For a choice between *ed0* and *ed1*, the C# source code structure is given as follows:

$$\begin{array}{l} \mathit{Choice}\ \mathit{choi}\ =\ \mathit{new}\ \mathit{Choice}(\mathit{new}\ \mathit{Opt}[]\{\mathit{ed}0,\ \mathit{ed}1\}); \\ \mathit{switch}(\mathit{choi.select}())\{ \\ \mathit{case}\ 0: \\ \quad \mathit{go\ to\ ed}0'\mathit{s\ branch} \\ \mathit{case}\ 1: \\ \quad \mathit{go\ to\ ed}1'\mathit{s\ branch} \\ \} \end{array}$$

The "Opt" is an abstract class here. The *EventDock, Skip* and *Stop* inherit "Opt" so they all can be used as an "Opt" object.

The CSP# operator *parallel* is implemented as *Parallel* in the PAT.Runtime library. It is created by providing the subprocesses array as parameters. The *Parallel* also implements "CSPProcess" interface. Suggest the subprocesses are $\{p1, , \ldots, pn\}$, the following statements create a Parallel process and run it immediately.

$new\ parallel(\{p1,\ ..,pn\}).run();$

The Parallel starts the subprocesses simultaneously, each subprocess at its own thread. The "run()" method of Parallel returns only after all the subprocesses have terminated.

Noted that the synchronized events are not automatic managed here. Developers shall expand the EventDock objects based on the alphabets of the subprocesses. After the Parallel finished, the expanded EventDock shall be contracted, otherwise the program will be blocked on the expanded EventDock causing deadlocks.

The global shared variables need protection in the C# program. The read and write operations on these variables are considered as *Data Operation* in CSP# model. The "GloBase" class provides "DataOpBegin()" and "DataOpEnd()" to be called before and after the access to global shared variables. They can be used as follows.

```
GloBase.DataOpBegin();
// the statements accessing global shared variables
GloBase.DataOpEnd();
```

It is recommended to creat a class "Glo" to manage all the events, channels and variables. It will make the CSP# model of the C# program more readable. The "Glo" shall inherit the "GloBase" class as it provides a lot of handy methods to manage the alphabet. However, the developers can also choose to manage the alphabets in other ways.

The condition expressions in CSP# are represented as C# delegate in the "PAT.Runtime" as follows:

```
public delegate bool EvaGuardExpr();
```

The delegates of “EvaGuardExpr” can access the global shared variables and the data protection will be provided when the delegates are used in “GChoice” objects.

For the “EvaGuardExpr” to be used in a “GChoice” object, they shall be put in a “LstExpr” object. The “LstExpr” class manages a list of the “EvaGuardExpr” for each branch. If a certain branch does not have condition expression attached, we insert an empty list of “EvaGuardExpr” at the branch’s index.

A “GChoice” object is created with a “LstExpr” object and an “Opt” array. The “GChoice” provides the atomic evaluation on the condition expressions and the engagement of the first event in corresponding branch. If a certain branch does not want to be atomically engaged on its first events, it needs to insert an EventDock object before its first events. The inserted EventDock object shall not synchronize to any other thread so it is always enabled.

With the “LstExpr” and “Opt” array be correctly configured, the “GChoice ” object is used as the “Choice”. Calling its “select()” will start the operator and the return value indicates which branch is chosen.

The below source code shows the implementation of an “IFA” operator using “GChoice”. The model is $\mathit{ifa}\ (a > 0)\{e_0 \rightarrow Q_0\}\mathit{else}\{e_1 \rightarrow Q_1\}$ We assume the EventDock objects “ed0, ed1” represent e_0, e_1, “q0, q1” represent Q_0, Q_1

```
class Global {
   static public int a;
}

bool expr0() {return (Global.a > 0); }
bool expr1() {return !(Global.a > 0); }

LstExpr lst = new LstExpr();
List⟨EvaGuardExpr⟩ g0 = new List⟨EvaGuardExpr⟩();
List⟨EvaGuardExpr⟩ g1 = new List⟨EvaGuardExpr⟩();
g0.Add(expr0);  g1.Add(expr1);
lst.Add(g0);  lst.Add(g0);

GChoice gchoi = new GChoice(lst, new Opt[]{ed0, ed1});
switch(choi.select()){
case 0 :
   q0.run();   break;
case 1 :
   q1.run();   break;
}
```

For the above example, if we change the last part to the following code, adding the always enabled before e_0, e_1, the model becomes $if\ (a > 0)\{e_0 \rightarrow Q_0\}else\{e_1 \rightarrow Q_1\}$.

```
EventDock tau0 = EventDock.create();
EventDock tau1 = EventDock.create();
GChoice gchoi = new GChoice(lst, new Opt[]{tau0, tau1});
switch(gchoi.select()){
case 0 :
   ed0.syn();
   q0.run();
case 1 :
   ed1.syn();
   q1.run();
}
```