Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

From Verified Model to Executable Program – the PAT Approach

Huiquan Zhu · Jing Sun · Jin Song Dong · Shang-Wei Lin

Received: date / Accepted: date

Abstract CSP# is a formal modeling language that emphases on the design of communication in concurrent systems. PAT framework provides a model checking environment for the simulation and verification of CSP# models. Although the desired properties can be formally verified at the design level, it is not always straightforward to ensure the correctness of the system's implementation confirms to the behaviors of the formal design model. To avoid human error and enhance productivity, it would be beneficial to have a tool support to automatically generate the executable programs from their corresponding formal models. In this paper, we propose such a solution for translating verified CSP# models into C# programs in the PAT framework. We encoded the CSP# operators in a C# library – "PAT.Runtime", where the event synchronization is based on the "Monitor" class in C#. The precondition and choice layers are built on top of the CSP event synchronization to support language specific features. We further developed a code generation tool to automatically transform CSP# models into multi-threaded C# programs. We proved that the generated C# program and original CSP# model are equivalent on the trace semantics. This equivalence guarantees that the verified properties of the CSP# models are preserved in the generated C# programs. Further-

H. Zhu
Department of Computer Science, National University of Singapore, Singapore
E-mail: huiquanz@comp.nus.edu.sg

J. Sun
Department of Computer Science, University of Auckland, New Zealand
E-mail: jing.sun@auckland.ac.nz

J.S. Dong
Department of Computer Science, National University of Singapore, Singapore
E-mail: dcsdjs@nus.edu.sg

S.W. Lin
Temasek Laboratories, National University of Singapore, Singapore
E-mail: tsllsw@nus.edu.sg

more, based on the existing implementation of choice operator, we improved the synchronization mechanism by pruning the unnecessary communications among the choice operators. The experiment results showed that the improved mechanism notably outperforms the standard JCSP library.

Keywords Modeling Checking · CSP# · Multi-threaded Programming · C#

1 Introduction

A concurrent software system contains multiple computational processes running in parallel. Each process performs a number of operations sequentially and they communicate among each other to collaborate on complex tasks. At the design phase, it is useful to apply formal specification and verification techniques to enhance the correctness. Building formal model of the concurrent system helps to avoid design faults in early stage, as these faults are much more difficult to discover or fix after the system has been implemented [38]. The requirements on the system's concurrency can be represented as *properties* of the models, which describe the kinds of communication sequences that are allowed by the system [28]. These properties can be further formally verified with the help of the supporting tools, such as model checking [20, 39] and theorem proving [18].

Communicating Sequential Processes (CSP) [13] is one of the most popular formal languages in specifying concurrent systems. CSP# [29] extends CSP with programming features such as shared variables, event-attached code segments, 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. Process Analysis Toolkit (PAT) [24, 32] is a model checking framework, which can simulate, verify and analyze the concurrent properties that are specified and to be satisfied on the CSP# models.

After the communication patterns have been verified in PAT, the CSP# model will be implemented in the programming language used in the target platform. The other parts of the system, which do not involve in the communication, will be added between the communications, or they will replace certain events that represent non-communication functionalities. On the other hand, the verified properties of the model shall be preserved in the implemented program. We need to ensure under what situations the other parts of the system do not violent properties of the system as a whole. Additionally, the data flow of the CSP# model shall be maintained in the implemented program. Better tools and environment supports are needed to facilitate the usage of CSP# from designing phase to implementation phase.

At present, the developers have to manually implement the CSP# model on the target platform. JCSP [33] uses the built-in Java concurrent features, such as *monitor*, to implement CSP operators in Java. In [35] the authors

proved that the operators of JCSP are equivalent to the ones in classic CSP. Similar to JCSP, CSP.NET [19] implements CSP operators in .NET framework. CSP++ [10] is a framework that generates C++ source code from CSP models. It translates the validated CSP_M model to C++ program as the concurrent control layer. The functionality code and the control layer are weaved into the final program by the CSP++ framework. In [23], Lin et. al. introduced a method to transform CSP# model to state machine models then software code is generated from the translated state machines. However, the approach cannot represent CSP# communication, as CSP# involves both message passing communication and shared memory communication. There is no intuitive way to implement CSP# models in an object-oriented languages such as C#.

In this paper, we propose a solution to integrate CSP# into the development process, from the design to implementation phas. We applied CSP# in multi-threaded program design and generated its implementation in C# language. Firstly, we define the traces equivalence encoding between the CSP# model and its representation in C# program. Based on this equivalence, CSP# operators are implemented in the "PAT.Runtime" C# library. We further developed a code generation tool to transform CSP# models into multi-threaded C# programs, which makes use of the "PAT.Runtime" library for communications among threads. We proved that the generated C# program and original CSP# model are equivalent based on the trace semantics. This equivalence guarantees that the verified properties of the CSP# models are preserved in the generated C# programs.

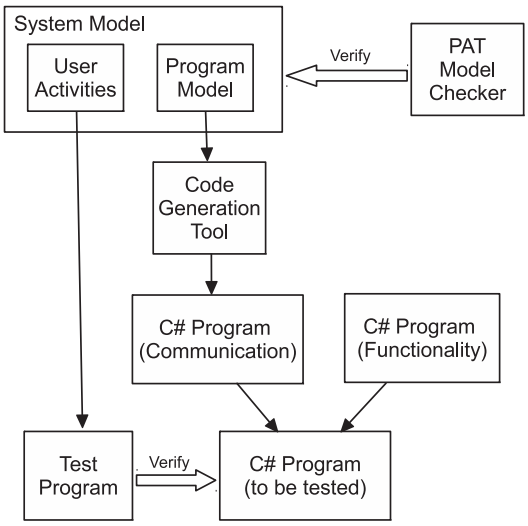

Fig. 1 Overall Approach to Translating CSP# Models into C# Programs

As shown in Figure 1, a typical system is modeled in two parts: the program model represent all the communication that happen inside the software

system; the external environment, the user activities, and their interaction with the software system are modeled as the "user activities". PAT verify the whole system to ensure it fulfils the requirements related to concurrency. After the verification, the code generation tool takes the formal model to generate C# program that controls the communication of the system. From the user validation perspective, we should also be able to generate the test program to further test the whole system.

Developers can use CSP# to model the communication aspects of the concurrent system and verify them in the PAT framework. Our code generation tool that is built in PAT framework can generate the C# program that has the same communication behaviors as the CSP# model. On the other hand, developers can implement other functionalities of the system in C#, where these C# codes can be imported into the CSP# model before the code generation phase. Alternatively, they could be added to the generated C# program as non-communication code, if they do not interfere with the inter-thread communication of the target program. Furthermore, based on the existing implementation of choice operator, we improved the synchronization mechanism by pruning the unnecessary communications among the choice operators. The experiment results showed that the improved mechanism notably outperforms the standard JCSP library.

The reset of the paper is organized as follows. Section 2 presents the background knowledge related to the approach. In section 3, we present the encoding of the CSP# notation into C# programs and its automated tool support for code generation. Section 4 presents the proof of correctness based on the trace equivalence on the proposed translation. In section 5, two case studies were developed for demonstrating usage and effectiveness of the approach. Section 6 presents the improvements on the code generation and its performance evaluation. Section 7 discusses the related work. Finally, section 8 concludes the paper and outlines the future work.

2 Background

2.1 CSP# and PAT

A CSP model is composed of a set of sequential processes communicating via events and synchronized channels. CSP# [29, 30] extends CSP to allow high-level modeling operators mixed with low-level sequential programs. It shares the principle ideas as TCOZ [25] that integrates the state specifications of the components with the interact operations between themselves. In addition, CSP# supports the communication via shared variables and asynchronous channels. A *process* in CSP# is defined as follows:

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

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

Let ✓ denote the special event of successful termination and αP denote the alphabet set of process P. Here αP contains all the events in P excluding ✓. The *Stop* communicates nothing and $Skip = ✓ \rightarrow Stop$. The *event prefix* $e \rightarrow P$ performs event e and then performs as P. Likewise, the *data operation* $e\{prog\} \rightarrow P$ first executes the C# code of *prog* then performs as P. The *channel output* $ch!x \rightarrow P$ evaluates the expression x, if the channel ch is not full, it sends the evaluated x to channel ch and behaves as P. Similarly, the *channel input* $ch?x \rightarrow P$ evaluates x and reads the evaluated x from channel ch then performs as P afterwards. The *guarded process* $[b]P$ is blocked until expression b becomes *true* and performs as P. It requires the expression b being *true* and P's first event being engaged together happen atomically. The *conditional choice* $if(b)\{P\}\ else\{Q\}$ 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 then behaves as Q. General choice $P[]Q$ can perform as P or Q. If P performs an event first, $P[]Q$ will behave as P afterwards, otherwise it will behave 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 shared variables and channels too. In the *interleaving composition* $P \,|||\, Q$, P and Q run independently and only communicate through shared variables and channels. The *interrupt* $P \triangle Q$ behaves as P until the first event of Q is engaged, then the process behaves as Q afterwards.

Shared variables in CSP# can be read by conditional expressions and they can be read and written by event-attached code segments. 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 combinations of shared memory and message passing communications. Besides the general *conditional choice*, CSP# has an *atomic conditional choice* operator 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 $ifb(b)\{P\}$ blocks the process until b becomes *true*, but it does not require the first event of P to be engaged atomically. It is the complement of the *guarded process*.

A *trace* of a process is a finite sequence of the events' names that the process has already engaged. The *traces* of a process is the set of all possible traces that the process can perform, denoted as $traces(P)$. Process Analysis Toolkit (PAT) [24, 31] is a generic model checking framework that supports the modeling, simulating and verification on the concurrent, real-time and

probabilistic systems. PAT also supports full set of Linear Temporal Logic (LTL) [1] properties being verified on CSP# model. A LTL formula F on CSP# is defined as:

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

Here e is an event or a channel input/output, *prop* is a proposition defined on shared variables. $\Box F$ 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.

2.2 Use Monitor to Implement CSP Operators

Monitor [12, 21] is a fundamental mechanism to synchronize between threads. It can provide *mutual exclusion*, *waiting* and *signaling* operations between threads. JCSP [33, 35] is a Java implementation of CSP operators. The authors used the monitor objects in Java to implement the *event* and *channel* communications. To verify their implementation, the authors model the *monitor* as CSP process. Two CSP models are built: the processes in one model use CSP channel to communicate and the ones in the other model use the monitor's CSP process to communicate. The equivalence of these two models was verified by a model checker. Similar proofs were applied to the *event*, *alternative* and *parallel* operators in JCSP.

3 Encoding CSP# Semantics in C# Language

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" [13], 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 $traces(G) = traces(M)$. 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 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.

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. Here we choose the *trace* semantics of CSP# as it is well defined and observable. 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*.

3.1 Thread Communication on CSP# Operator Level

With the equivalence defined on *traces*, this subsection discusses the implementation of the CSP# operators in the C# programming language, where the process level equivalence and the alphabet management are described in the next section.

3.1.1 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# pro-

grams 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 2, 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 $\{st.e, ed.e\}$ 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.

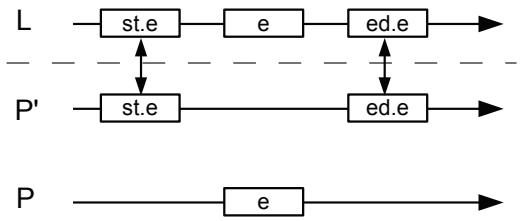

Fig. 2 Event Engagement Equivalence

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.

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 $O_e.m$.

3.1.2 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 $P = [b](e_p \rightarrow \mathit{Skip})$ and a data operation process $Q = e_q\{b = \mathit{true}\} \rightarrow \mathit{Skip}$. Suppose the boolean variable b is *false* at 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 3. 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.

```
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 :  ...
```

Fig. 3 Wait and Notify Example

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.

3.1.3 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 4.

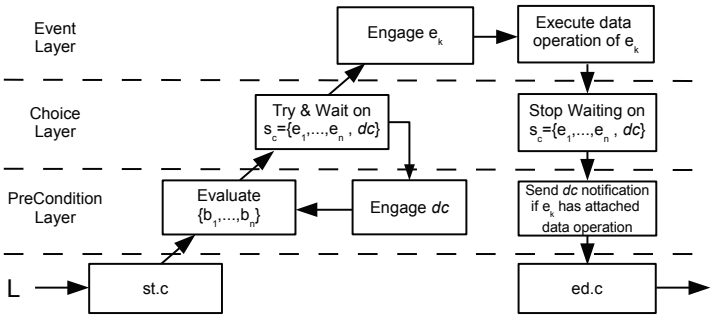

Fig. 4 General Choice Structure

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 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.

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

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

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

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 1.

	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 1 CSP# Operators Represented with "G" Operators

3.2 Process Level Implementation and Alphabet Management

3.2.1 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.

3.2.2 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 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 5 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.

```
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(){...}
}
```

Fig. 5 A Process Class Example

3.2.3 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.

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 2 Generated C# Code for Simple Operators

For the *Stop, Skip, event* and *channel* operators, their corresponding C# statements in the "run()" method are relative simple. Table 2 lists their ini-

tialization code in the "setParas()" method 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 $P = (e1 \rightarrow Q)[](e2 \rightarrow R)$, the following pseudocode is in the "run()" method of process P.

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 1.

```
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;
}
```

3.2.4 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 subsection, 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.

$$P() = atomic\{e1 \rightarrow e2 \rightarrow e3 \rightarrow Skip\};$$
$$Q() = e4 \rightarrow e5 \rightarrow e2 \rightarrow Skip;$$
$$R() = e6 \rightarrow e5 \rightarrow e7 \rightarrow Skip;$$
$$Sys() = P() \,||\, Q() \,||\, R();$$

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. The $\triangledown$ symbol 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.

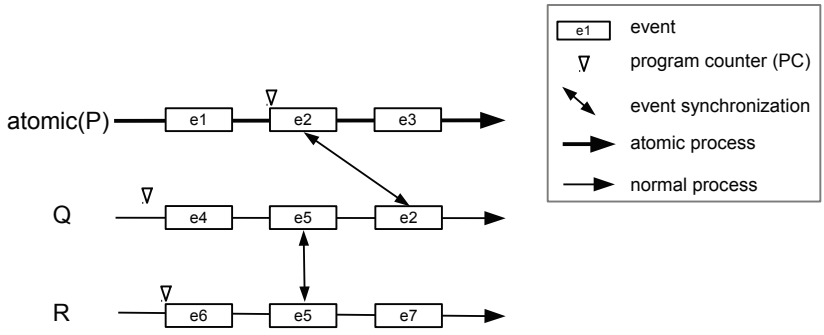

Fig. 6 Atomic Process Example

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.

3.2.5 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 previously, 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*. 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 [31]. 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 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.

4 Correctness Proofs of the Translation

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.

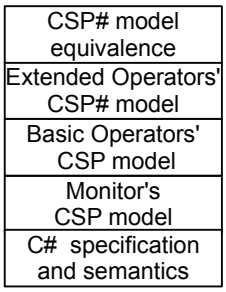

Fig. 7 the Equivalence Hierarchy

The proof is discussed on different functionality layers as shown on Figure 7. 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.

4.1 CSP Operators Level Equivalence

For the implementation of CSP operator in program, The *monitor* [12, 14, 22] 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 [33,35] 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 better support terminating programs. In [35], 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. 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 $Monitor(c)$; the read and write operations on the variables are represented as the channel events on $Hold(c)$ and $Empty(c)$.

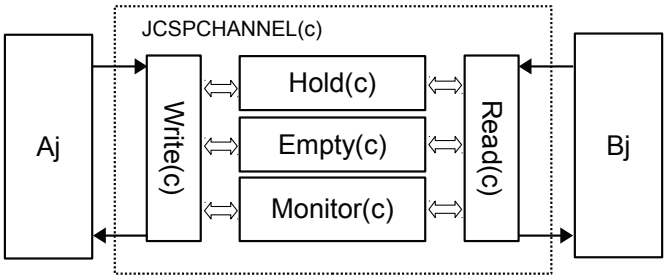

Fig. 8 JCSP Channel Communication

Similar case is on process $Write(c)$. The structure of the $JCSPCHANNEL$ model is shown in Figure 8. The Aj synchronizes with the $Write(c)$ on the beginning and ending of $Write(c)$ to send the message, Bj synchronizes with $Read(c)$ 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* [35].

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.

4.2 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.

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

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.

$$
\begin{array}{lll}
GMul() & = & dcstart \rightarrow dcend \rightarrow GMul() \ []\\
& & \quad evstart \rightarrow evend \rightarrow GMul();
\end{array}
$$

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.

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

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

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

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.

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

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*.

$$
\begin{aligned}
&\#assert\; M0m()\; refines\; M1m();\\
&\#assert\; M1m()\; refines\; M0m();
\end{aligned}
$$

4.3 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 1 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.

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 (except the remainder operator "%"). 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.

Theorem 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.

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

$\varepsilon = Stop$: 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 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 $M_\varepsilon \approx C_\eta$.

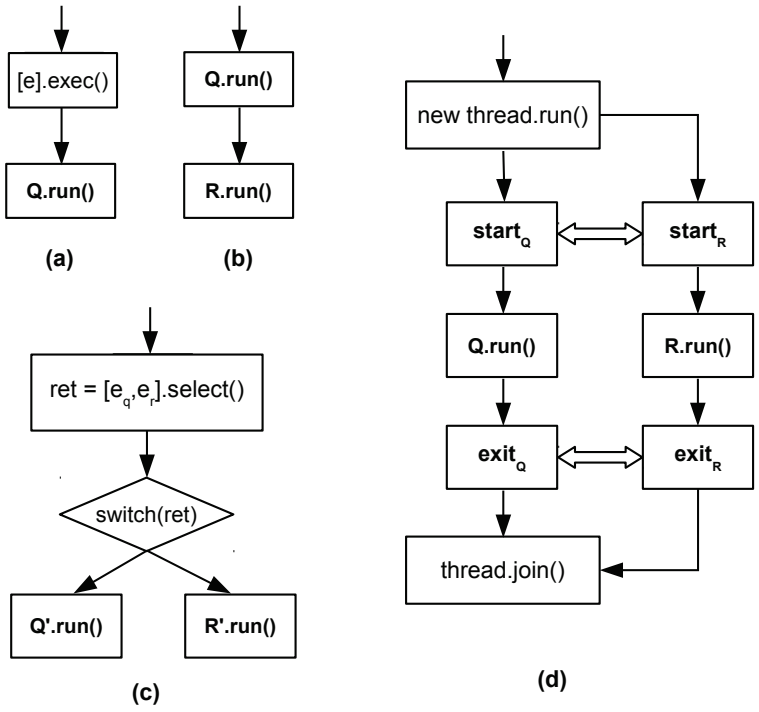

Fig. 9 Structure of the Generated "run()" Mehtods

Similar results apply to $\varepsilon = e\{prog\} \rightarrow Q$, $\varepsilon = ch!x \rightarrow Q$ and $\varepsilon = ch?x \rightarrow Q$. 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 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 ✓, 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 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 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 $\varepsilon = Q \;|||\; R$ 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 $F = e \mid \Box F \mid \Diamond F \mid X\, F \mid F_1\, U\, F_2 \mid F_1\, R\, F_2$.

5 Case Studies

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. 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.

5.1 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 as illustrated in Figure 10, 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 the 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 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

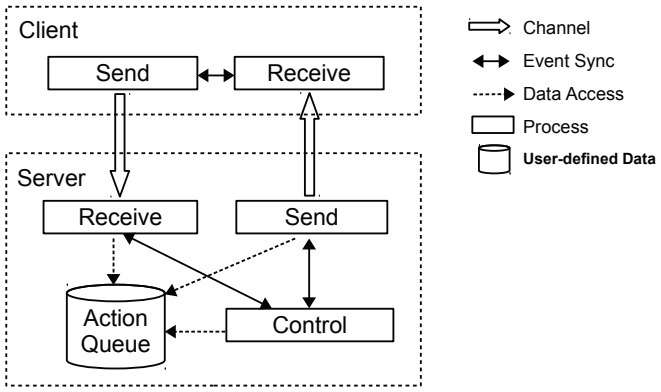

Fig. 10 Turn-based Game Server-client Hierarchy

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 $TbServerReceiveRd$.

$$TbServerReceiveRd(i) = ctos[i]?i \rightarrow enqueue.i\{num = num + 1\} \rightarrow edHalfRound \rightarrow TbServerReceiveRd(i);$$

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 : *new Parallel*(*new CSProcess*[]
{*server*, *client*1, .., *clientn*})).*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 using the CSP# operators in "PAT.Runtime" library. After initializing the events and channel objects, the threads can communicate via these objects the same way as those in the original CSP# model.

5.2 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 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 11.

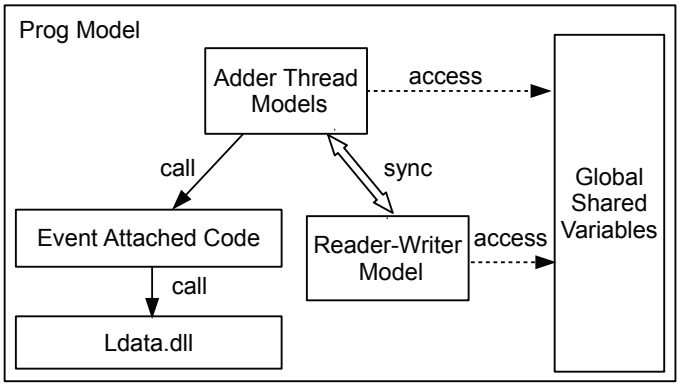

Fig. 11 Concurrent Accumulator Model Overview

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.

5.2.1 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.

$$
\begin{aligned}
&ReaderHead(i) = [noOfReading < M \&\& !writing]startR.i \\
&\quad \rightarrow startreadop.i\{noOfReading = noOfReading + 1;\ \} \\
&\quad \rightarrow startRout.i \rightarrow Skip; \\
&ReaderTail(i) = [noOfReading > 0]endR.i \\
&\quad \rightarrow stopreadop.i\{noOfReading = noOfReading - 1;\ \} \\
&\quad \rightarrow endRout.i \rightarrow Skip; \\
&Reader(i) = ReaderHead(i);\ ReaderTail(i);\ Reader(i);
\end{aligned}
$$

$$
\begin{aligned}
&Writer(i) = [noOfReading == 0 \&\& !writing]startW.i \\
&\quad \rightarrow startwriteop.i\{writing = true;\ \} \rightarrow startWout.i \\
&\quad \rightarrow endW.i \rightarrow stopwriteop.i\{writing = false;\ \} \\
&\quad \rightarrow endWout.i \rightarrow Writer(i);
\end{aligned}
$$

$$
\begin{aligned}
&Controller() = (startR.0 \rightarrow startRout.0 \rightarrow Controller()) \\
&\quad [](endR.0 \rightarrow endRout.0 \rightarrow Controller()) \\
&\quad [](startW.0 \rightarrow startWout.0 \rightarrow Controller()) \\
&\quad [](endW.0 \rightarrow endWout.0 \rightarrow Controller()) \\
&...
\end{aligned}
$$

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)*".

$$
\begin{aligned}
&RWReaders() = Reader(0) \,||\, Reader(1) \,||\, Reader(2); \\
&RWWriters() = Writer(0) \,||\, Writer(1) \,||\, Writer(2); \\
&ReadersWriters() = RWReaders() \,||\, RWWriters() \,|| \\
&\qquad\qquad Controller();
\end{aligned}
$$

$$
\begin{aligned}
&\#defineexclusive!(writing == true \&\& noOfReading > 0); \\
&\#assertReadersWriters() \,|= [] exclusive; \\
&\#definesomeonereadingnoOfReading > 0; \\
&\#assertReadersWriters() \,|= [] <> someonereading;
\end{aligned}
$$

$\#definesomeonewritingwriting == true;$
$\#assertReadersWriters() \models [] <> someonewriting;$

5.2.2 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.

$startR.i \rightarrow (read\ operation) \rightarrow endR.i$
$startW.i \rightarrow (write\ operation) \rightarrow endW.i$

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.

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

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

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

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 $\{d_0, d_1, .., d_{n-1}\}$.

5.2.3 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 creates 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 generated project is ready to be built in Visual Studio for execution. The "ReadersWriters" part is non-terminating, so the program does not exit. The "Adder" part performs the calculation, prints out the results 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 Performance Improvement and Evaluation

In this section, 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.

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 [33], CSP.NET [19] and CTJ [26] 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.

6.1 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 [35]. The choice operator 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 12. For a thread to engage a choice operator, it may go into the 9 activities that are marked as "S1" to "S9" in Figure 12.

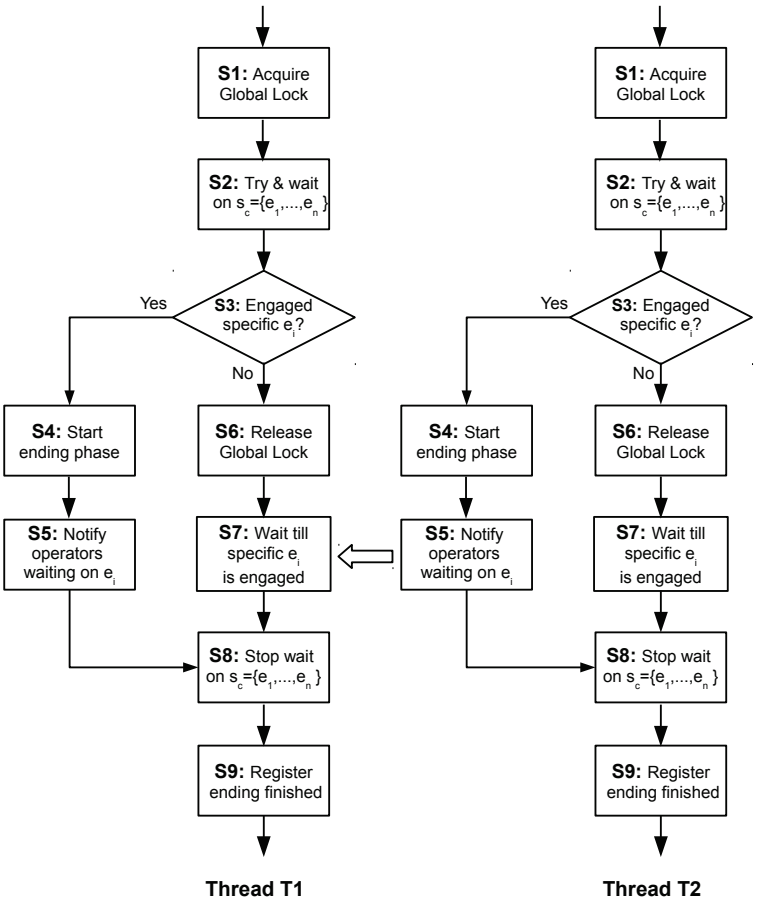

Fig. 12 CSP Choice Operator Communication

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 12, 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".

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 nth 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 (nth) 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.

6.2 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.

6.2.1 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 12, 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[1], the choice operator follows the route "$\langle S1, S2, S3, S4, S5, S8, S9\rangle$" 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.

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

6.2.2 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 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 "$\langle S1, S2, S3, S4\rangle$" and "$\langle S1, S2, S3, S6\rangle$" 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 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 13. Compared to Figure 12, 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

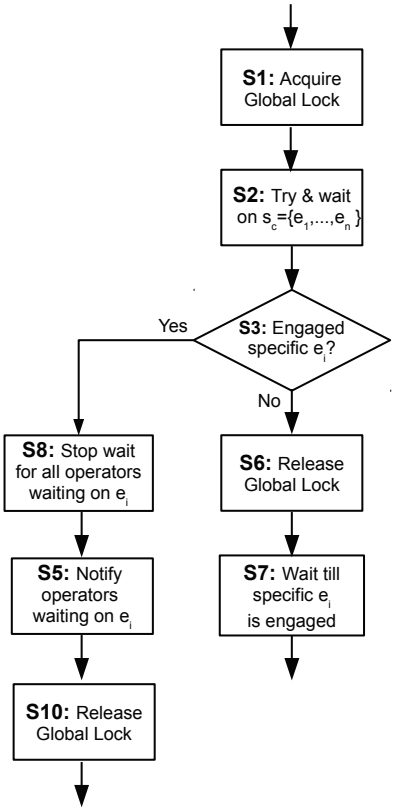

Fig. 13 Improved Choice Operator Communication

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.

6.2.3 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 3.1.3, 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

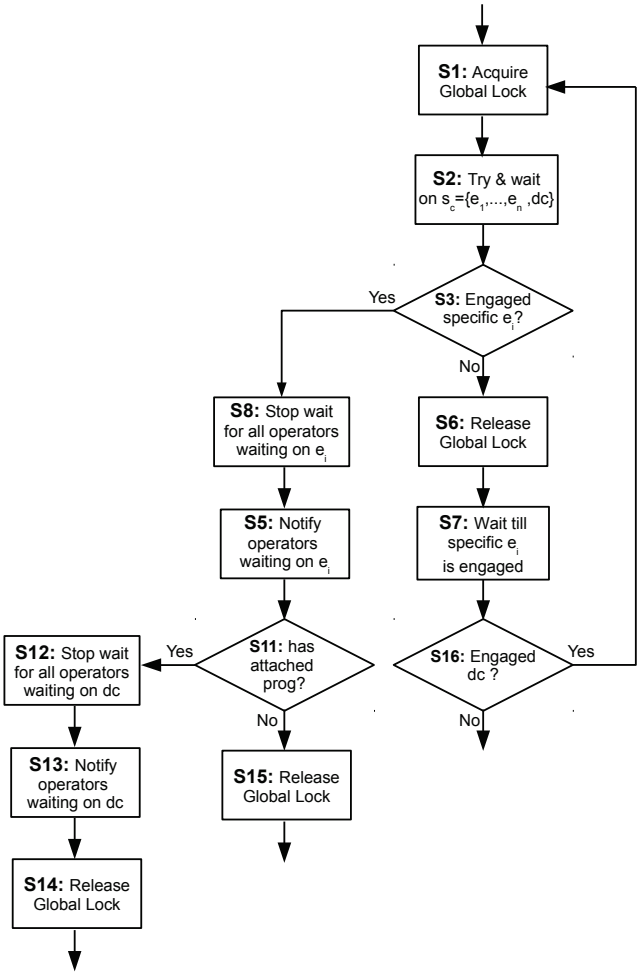

Fig. 14 Improved CSP# Choice Operator Communication

improved cooperation mechanism of the "general choice" operator is shown in Figure 14. 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.

6.3 Experiment and Performance

We implemented the improved version of the CSP# operators in "PAT.Runtime" library. To compare the performance, we used the common *dining philosopher* model as the benchmark example. Performance of the CSP operators from the JCSP, the original "PAT.Runtime" and the improved "PAT.Runtime" libraries are compared. 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 6.1. The operators in the improved "PAT.Runtime" library use our improved mechanism in Section 6.2.2.

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 3.

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 3 Performance Comparison between JCSP, CSP# Operator and Improved Operator

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 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.

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 4 Performance Comparison between Hand-Coded, CSP# Operator and Improved Operator

Table 4 compares the performances among the hand-coded program, the program using the "PAT.Runtime" and the one using improved "PAT.Runtime". All three programs implement the multiple threads in synchronizing 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 could observe that the program using the improved "PAT.Runtime" library saves much running time compared to the one using the 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 the original "PAT.Runtime" library, the running time of the program using improved library is much closer to the hand-coded one.

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.

7 Related Work

JCSP [33, 34] 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 [35] 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 [11,26] provides the CSP operators in multi-threaded Java programs. It replaces the OS scheduler to provide more flexibility. JCSProB [36, 37] apply JCSP's idea to provide the operators for the B+CSP model. JACK [7] is another CSP framework for Java program. C++CSP [3–5] provides CSP operators in C++ language. CSP.NET [19] 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 [15] provides *named channels*, *parallel* and *choice* operators for process communication. PyCSP [2] brings CSP to Python via external library. The Go language [27] also uses CSP style channels for synchronization.

In [16, 17] 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++ [8–10] 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 [11], Hilderink proposed a graphical notation of CSP. Based on this notation, gCSP [6] provides a graphical tool to design model in CSP diagram. The design model in gCSP can also generate code in Occam and C languages. However, there is no formal guarantee between the generated C program and its CSP diagrams. Our approach integrates the code generation with the PAT model checking framework by providing formal semantics of the CSP# language into the runtime library of the model checker. A code generation tool was built on top of PAT to automatically transform verified CSP# models into executable multi-threaded C# programs. This approach further extends the already-popular PAT model checking framework with code generation facilities.

8 Conclusion

In this paper, we tackled the problem of automatic generation of executable programs from verified formal models. We applied our approach to the CSP# language in the PAT framework into corresponding C# programs. We first investigated the differences between the CSP# specifications verified by the model checker and the C# program running in the target platform. With these concerns in mind, we chose to emphasize on the *trace* semantic to define the equivalence between CSP# model and the C# program. Based on the trace equivalence, we designed the "PAT.Runtime" library to provide the CSP# operators in C# programs. The basic event synchronization in the library makes use of the *monitor* class in C#. On top of the event synchronization, we added 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#.

By introducing the alphabet as well as shared variable management to the "PAT.Runtime" library, our code generation tool in PAT automatically 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 verified properties on traces of the original CSP# model.

Two case studies are performed to demonstrate the use of the "PAT.Runtime" library 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. In addition, based on the existing implementation of choice operator, we further improved the synchronization mechanism to remove the unnecessary communications among these choice operators. The experiment results show the improved mechanism notably outperforms the standard JCSP library.

With the "PAT.Runtime" library and the code generation tool, CSP# can be easily 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, under the current implementation, the developers need to manually ensure the non-communication codes do not interfere with 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.

Acknowledgements This work was partially supported by the TRF project "Research and Development in the Formal Verification of System Design and Implementation".

References

1. Baier, C., Katoen, J.: Principles of Model Checking. The MIT Press (2008)
2. Bjørndalen, J.M., Vinter, B., Anshus, O.J.: PyCSP - Communicating Sequential Processes for Python. In: A.A. McEwan, S.A. Schneider, W. Ifill, P.H. Welch (eds.) The 30th Communicating Process Architectures Conference, CPA 2007, pp. 229–248 (2007)
3. Brown, N.: C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. Communicating Process Architectures 2007: WoTUG-30 pp. 183–205 (2007)
4. Brown, N., Welch, P.: An introduction to the Kent C++ CSP Library. Communicating Process Architectures **2003**, 139–156 (2003)
5. Brown, N.C.: C++ CSP networked. Communicating Process Architectures **2004**, 185–200 (2004)
6. East, I., Martin, J., Welch, P., Duce, D., Green, M.: gCSP: a graphical tool for designing CSP systems. Communicating Process Architectures 2004 **27**, 233 (2004)
7. Freitas, L.: JACK: A process algebra implementation in Java. Ph.D. thesis, Centro de Informatica, Universidade Federal de Pernambuco (2002)
8. Gardner, W.: CSP++: An object-oriented application framework for software synthesis from CSP specifications. Ph.D. thesis, Politecnico di Milano, Italy (2000)
9. Gardner, W.: 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, pp. 237–245. IEEE (2003)
10. Gardner, W.: CSP++: How Faithful to CSPm. Proc. Communicating Process Architectures 2005 (WoTUG-27) pp. 129–146 (2005)

11. Hilderink, G., Bakkers, A., Broenink, J.: 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, pp. 400–407. IEEE (2000)
12. Hoare, C.: Monitors: An Operating System Structuring Concept. Communications of the ACM **17**(10), 549–557 (1974)
13. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International Series in Computer Science. Prentice/Hall International (1985)
14. Howard, J.H.: Proving monitors. Commun. ACM **19**(5), 273–279 (1976). DOI 10.1145/360051.360079. URL `http://doi.acm.org/10.1145/360051.360079`
15. Jones, G.: Programming in Occam. Prentice-Hall International London (1986)
16. Kleine, M.: Using CSP for Software Verification. In: Proceedings of Formal Methods 2009 Doctoral Symposium, Eindhoven University of Technology, pp. 8–13 (2009)
17. Kleine, M.: CSP as a Coordination Language. In: Coordination Models and Languages, pp. 65–79. Springer (2011)
18. Lee, S.J., Dobbie, G., Sun, J., Groves, L.: Theorem prover approach to semistructured data design. Formal Methods in System Design **37**(1), 1–60 (2010). DOI 10.1007/s10703-010-0099-4. URL `http://dx.doi.org/10.1007/s10703-010-0099-4`
19. Lehmberg, A., Olsen, M.: An introduction to CSP.NET. Communicating Process Architectures **2006**, 13–30 (2006)
20. Li, Y., Dong, J.S., Sun, J., Liu, Y., Sun, J.: Model checking approach to automated planning. Formal Methods in System Design **44**(2), 176–202 (2014). DOI 10.1007/s10703-013-0197-1. URL `http://dx.doi.org/10.1007/s10703-013-0197-1`
21. Liang, H., Dong, J.S., Sun, J.: Evolution and Runtime Monitoring of Software Systems. In: SEKE '07: Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering, pp. 343–348. Knowledge Systems Institute Graduate School, Skokie, Illinois, USA (2007)
22. Liang, H., Dong, J.S., Sun, J., Duke, R., Seviora, R.E.: Formal Specification-based Online Monitoring. In: ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems, pp. 152–160. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/ICECCS.2006.1690364
23. Lin, S.W., Liu, Y., Hsiung, P.A., Sun, J., Dong, J.S.: Automatic generation of provably correct embedded systems. In: Formal Methods and Software Engineering, pp. 214–229. Springer (2012)
24. Liu, Y., Sun, J., Dong, J.S.: PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In: ISSRE, pp. 190–199 (2011)
25. Mahony, B., Dong, J.S.: Blending Object-Z and Timed CSP: an introduction to TCOZ. In: Proceedings of the 20th international conference on Software engineering(ICSE'98), pp. 95–104. IEEE Computer Society (1998)
26. Schaller, N., Hilderink, G., Welch, P.: Using Java for Parallel Computing: JCSP versus CTJ, a Comparison. Communicating Process Architectures pp. 205–226 (2000)
27. Summerfield, M.: Programming in Go: Creating Applications for the 21st Century. Addison-Wesley Professional (2012)
28. Sun, J., Dong, J.S., Jarzabek, S., Wang, H.: Computer-Aided Dispatch System Family Architecture and Verification: An Integrated Formal Approach. IEE Proceedings - Software **153**(3), 102–112 (2006). URL `http://ieeexplore.ieee.org/iel5/5658/34486/01645517.pdf?isnumber=34486&arnumber=1645517`
29. Sun, J., Liu, Y., Dong, J.S., Chen, C.: 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), pp. 127–135 (2009)
30. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., Étienne André: Modeling and verifying hierarchical real-time systems using stateful timed csp. ACM Transactions on Software Engineering and Methodology **22**(1), 3:1–3:29 (2013). DOI 10.1145/2430536.2430537. URL `http://doi.acm.org/10.1145/2430536.2430537`
31. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09), *Lecture Notes in Computer Science*, vol. 5643, pp. 709–714. Springer (2009)

32. Sun, J., Liu, Y., Dong, J.S., Sun, J.: Compositional encoding for bounded model checking. Frontiers of Computer Science in China **2**(4), 368–379 (2008). DOI 10.1007/s11704-008-0035-6. URL http://dx.doi.org/10.1007/s11704-008-0035-6
33. Welch, P., Brown, N., Moores, J., Chalmers, K., Sputh, B.: Integrating and extending JCSP. Communicating Process Architectures 2007 **65**, 349–370 (2007)
34. Welch, P., Martin, J.: A CSP Model for Java Threads (and Vice-Versa)
35. Welch, P., Martin, J.: Formal analysis of concurrent java systems. Communicating Process Architectures **58**, 275–301 (2000)
36. Yang, L., Poppleton, M.: JCSProB: Implementing Integrated Formal Specifications in Concurrent Java. Communicating Process Architectures **65**, 67–88 (2007)
37. Yang, L., Poppleton, M.: Java implementation platform for the integrated state-and event-based specification in PROB. Concurrency and Computation: Practice and Experience **22**(8), 1007–1022 (2010)
38. Yuan, L., Dong, J.S., Sun, J., Basit, H.A.: Generic Fault Tolerant Software Architecture Reasoning and Customization. IEEE Transactions on Reliability **55**(3), 421–435 (2006). URL http://ieeexplore.ieee.org/iel5/24/35614/01688078.pdf?isnumber=35614&arnumber=1688078
39. Zhang, J., Liu, Y., Sun, J., Dong, J.S., Sun, J.: Model checking software architecture design. In: High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on, pp. 193–200 (2012). DOI 10.1109/HASE.2012.12