An Analytical and Experimental Comparison of CSP Extensions and Tools

Ling Shi[1], Yang Liu[2], Jun Sun[3], Jin Song Dong[1], and Gustavo Carvalho[4]

[1] SoC, National Univ. of Singapore, {shiling,dongjs}@comp.nus.edu.sg
[2] Temasek Lab, National Univ. of Singapore, tslliuya@nus.edu.sg
[3] ISTD, Singapore Univ. of Technology and Design, sunjun@sutd.edu.sg
[4] Centro de Informática, UFPE, Brazil, ghpc@cin.ufpe.br

Abstract. Communicating Sequential Processes (CSP) has been widely applied to modeling and analyzing concurrent systems. There have been considerable efforts on enhancing CSP by taking data and other system aspects into account. For instance, CSP$_M$ combines CSP with a functional programming language whereas CSP# integrates high-level CSP-like process operators with low-level procedure code. Little work has been done to systematically compare these CSP extensions, which may have subtle and substantial differences. In this paper, we compare CSP$_M$ and CSP# not only on their syntax, but also operational semantics as well as their supporting tools such as FDR, ProB, and PAT. We conduct extensive experiments to compare the performance of these tools in different settings. Our comparison can be used to guide users to choose the appropriate CSP extension and verification tool based on the system characteristics.

1 Introduction

Communicating Sequential Processes (CSP) [3], a prominent member of the process algebra family, has been designed to formally model *concurrent systems*. It represents system behavior in terms of processes constituted by a rich set of compositional operators. CSP also provides algebraic laws such that equivalence of process expressions can be rigorously established. It has been applied to a variety of safety-critical systems [25].

With the increasing size and complexity of concurrent systems, it becomes clear that CSP has its limitations in modeling systems with non-trivial data structures (e.g., *array*) or functional aspects. To solve this problem, many considerable efforts on enhancing CSP have been made. The most noticeable is CSP$_M$ [15], a machine-readable dialect of CSP, combining CSP with a *functional* programming language. Recently, CSP# (Communicating Sequential Programs) [22] has been proposed to integrate high-level CSP-like process operators with low-level program constructs such as *assignments* and *while* loops. Although these languages support CSP-like modeling notations and can deal with similar types of concurrent systems, there are subtle and substantial differences between them. For example, concurrency is captured differently; CSP$_M$ only supports synchronous channel communications, while CSP# supports both synchronous/asynchronous channels and shared variables. In addition, those differences can lead to different verification capabilities empowered by their respective analysis tools, i.e., FDR (Failures Divergence Refinement) [10] and ProB [6] for CSP$_M$, and

PAT (Process Analysis Toolkit) [23] for CSP#. Little work has been conducted to provide a comprehensive investigation of these CSP extensions so as to help users choose appropriate languages/tools for various systems from the perspectives of modeling and verification needs.

In this work, we systematically compare CSP$_M$ and CSP# in terms of three aspects, i.e., language syntax, operational semantics, and reasoning power of their supporting tools. Firstly, we show the syntactic differences, followed by comparing the operational semantics. We also discuss the transformation between CSP$_M$ and CSP# models. Secondly, we characterize various reasoning techniques and verifiable properties of FDR, ProB and PAT, respectively. Next, we explore the strengths and limits of the languages and tools by modeling and verifying nine systems, each of which is designed to show particular features of the languages or the tools. Lastly, we investigate reasons behind the experiment results; particularly, the semantic differences between CSP$_M$ and CSP# lead to different state spaces and optimizations in model checking.

We believe that the comparison is useful for the following reasons. Firstly, our comparison may guide users to select an appropriate modeling language. The decision depends on system features (e.g., shared variables, etc.) and properties to prove (e.g., compositional refinement checking, etc.). Secondly, our analysis of languages that are designed for concurrent systems in terms of simplicity and expressiveness (e.g., communications via channels or shared memory) can act as a reference in designing new programming languages of concurrent systems. Thirdly, the translation discussed in the paper can help users to change their models between CSP$_M$ and CSP#, and hence to utilize different reasoning power of their respective reasoning tools. Lastly, our experiments with FDR, ProB, and PAT provide qualitative analysis of tool capability/efficiency.

2 CSP$_M$ vs. CSP#: Syntax

CSP$_M$ enriches CSP with an expression language that is based on *functional* foundations. It mainly uses event synchronization to specify concurrent systems, and supports operators like linked parallel $P[c <-> c']Q$ in which two different channels c and c' from processes P and Q respectively run synchronously. CSP# not only inherits event synchronization and compositional process constructs from CSP, but also supports additional features like asynchronous channel communication, imperative programs, etc. In this section, we elaborate the differences between these two languages in terms of their syntax. Table 1 shows common CSP, CSP$_M$ and CSP# process definitions, where P (and Q) is a process with an optional list of parameters; a is an event name; A and A' are sets of event names and channel expressions; b is a Boolean expression; c and c' are channel names; e is an expression; x and x' are variables; and V is a set of accepted values. We illustrate the detailed differences from two perspectives.

Data Perspective CSP$_M$ supports functional paradigm, where process parameters can take in processes, functions, and channels. This is not available in CSP# which adopts imperative paradigm, although this limitation may be resolved partially through 'clever' modeling. For instance, a CSP$_M$ concrete process $System = P(Sys1, Sys2)$ associated with an abstract process $P(P1, P2) = a \rightarrow P1 \; [] \; b \rightarrow P2$ can be translated to a

CSP	**CSP$_M$**	**CSP#**	**Description**
$STOP$	$STOP$	$Stop$	deadlock
$SKIP$	$SKIP$	$Skip$	termination
$CHAOS$	$CHAOS(A)$	-	chaotic process
$a \rightarrow P$	$a \rightarrow P$	$a \rightarrow P$	event prefixing
$c!e \rightarrow P$ $c?x \rightarrow P$	$c?x?x' : V!e \rightarrow P$	$c!e \rightarrow P$ $c?[b]x \rightarrow P$	channel communication
$P \,\Box\, Q$	$P \;[]\; Q$	$P \;[*]\; Q$	external choice
$P \sqcap Q$	$P \;\lvert\sim\rvert\; Q$	$P <> Q$	internal choice
$P;\ Q$	$P;\ Q$	$P;\ Q$	sequential composition
$P \setminus A$	$P \setminus A$	$P \setminus A$	hiding
$x := e$	-	$x := e$	assignment
$P \lhd b \rhd Q$	$if\ b\ then\ P\ else\ Q$	$if\ b\ then\ P\ else\ Q$	conditional choice
$P \parallel Q$	$P[\mid A \mid]Q$ $P[A \parallel A']Q$ $P[c <-> c']Q$	$P \parallel Q$	parallel composition
$P \mid\mid\mid Q$	$P \mid\mid\mid Q$	$P \mid\mid\mid Q$	interleaving
$P \triangle Q$	$P/\backslash Q$	$P\ interrupt\ Q$	interrupt

Table 1. Similar Syntax among CSP, CSP$_M$ and CSP#

CSP# concrete process $System = a \rightarrow Sys1\ [*]\ b \rightarrow Sys2$, here $Sys1$ and $Sys2$ are processes. However, it may not be possible to specify abstract process behavior (e.g., process P in this example) in CSP#, whose parameters are processes.

CSP$_M$ enables rich data expressions such as sequences, sets, Boolean, tuples, and lambda calculus. It also allows users to define data types using the reserved word "datatype". CSP# directly supports integers, Boolean, array of integers or Boolean. In addition, it supports user-defined data types and corresponding operations using imperative languages like C#[1], C, or Java. Functions can be declared in CSP$_M$ following the functional paradigm, while in CSP#, they are encoded as processes or defined as static C# methods (which can be invoked via method *call* in CSP# models).

A channel in CSP$_M$ is declared with an explicit type. Values communicated through a channel must be in their type range; otherwise, an error is reported at run time by FDR and ProB. Moreover, CSP$_M$ is dynamically typed in FDR, namely, there is no way to declare the types of functions and variables (process parameters), while ProB can type check the CSP$_M$ models in a dynamic or (optional) static way [7]. In contrast, CSP# is weak typed (a.k.a. loose typing) and therefore no type information is required when declaring a variable or channel. Channels are declared with its name and buffer size. If the buffer size is 0, then it is declared as a synchronous channel, otherwise it is an asynchronous channel. The process parameters and channel input variables can take in values with different types at different time. As long as there is no type mismatch (e.g., using an integer as a guard condition), the execution can proceed; otherwise, invalid type casting exception is raised at run time.

Process Perspective One big difference is that CSP# directly supports shared variables. Unlike CSP$_M$ which excludes assignments of shared variables [10], CSP# treats assignments as an important modeling feature. In CSP#, an event can be associated with an

[1] C# is the best supported language in PAT and used as the representative language in this paper.

imperative program, which is executed *atomically* together with the occurrence of the event. For instance, an event associated with a program (referred to as a data operation) is written as $a\{prog\} \rightarrow P$ where $prog$ is the program and a is an event name. We remark that a shared variable can be modeled as a process parallel to the one that uses the variable (see [3] and [17]). Recently, shared variable analyzer (SVA) [17], a front-end of FDR, has been developed to convert programs (like C programs) with shared variables into CSP$_M$ models, in which shared variables are modeled as *variable processes*; reading from/writing to those shared variables are carried out over channels. We illustrate the modeling of shared variables in Section 3.

Asynchronous channels, as a popular and practical type of communication mechanism for networked systems, are directly supported in CSP#. Given an asynchronous channel ac with a positive buffer size, $ac!e \rightarrow P$ evaluates expression e with the current variable valuation, puts the value into the tail of the respective buffer for ac and then behaves as P. In contrast, $ac?x \rightarrow P$ (and $ac?[b]x \rightarrow P$) gets the top element from the respective buffer, assigns it to variable x and then behaves as P (the latter further constrains the received data to satisfy the Boolean condition b). Buffers store messages in a first-in-first-out (FIFO) order. Notice that asynchronous channels in CSP# are similar to those supported in Promela [4]. Although asynchronous channels are not directly supported in CSP$_M$, they can be modeled as buffer processes by event synchronization, which will be shown in Section 3.

In CSP$_M$, users are required to indicate synchronized events in three kinds of parallel compositions, which are, *sharing* ($P[|\ A\ |]Q$), *alphabetized parallel* ($P[A \parallel A']Q$), and *linked parallel* ($P[c \leftrightarrow c']Q$). On the other hand, CSP# supports only alphabetized parallel composition and frees users from specifying explicit alphabets of processes in parallel; a sophisticated procedure [22] calculates automatically a *default* alphabet of a process which is the set of events that constitute the process expression. Nevertheless, this procedure may not work when an event name consists of global variables or process parameters which change through *recursive calls*; in such a case, users need to specify the alphabet of a process. Notice that in order to avoid data race, data operations are not a part of the alphabet and therefore are never synchronized.

In CSP#, an event can have the name tau to represent the invisible event τ in event prefixing or data operations, e.g., $tau \rightarrow Stop$ or $tau\{prog\} \rightarrow Stop$. With the support of tau event, users can avoid using hiding operator to explicitly hide some visible events by naming them tau. *External* and *internal* choices are supported in both languages. Moreover, CSP# allows *general choice* $P[]Q$ in which the choice is resolved by any event. This operator is more like the CCS + operator, which can be resolved by a τ event performed by either process. Nonetheless, the general choice operator can be simulated in CSP$_M$ [14].

Besides the common conditional choice, CSP# copes with two additional types of conditional choices to facilitate modeling: *atomic* conditional choice $ifa\ b\ \{P\}\ else\ \{Q\}$ and *blocking* conditional choice $ifb\ b\ \{P\}$. With the former, the checking of condition b is to be conducted *atomically* with the occurrence of the first event in P or Q. The latter is blocked when b is unsatisfied.

Both CSP$_M$ and CSP# define Boolean guard $b\&P$ and $[b]P$ respectively; process waits until condition b becomes true and then behaves as P. Replicated process operators, such as replicated external/internal choices, replicated parallel and interleaving, are also supported in both languages. Chaotic process ($\mathit{CHAOS}(A)$), event renaming ($P[[c \leftarrow c']]$), and untimed timeout ($P[> Q$) defined in CSP$_M$ are not directly handled in CSP#. We discuss how to model these features using CSP# operators in Section 3.

So far we have shown the syntactic differences between CSP$_M$ and CSP#. Both CSP$_M$ and CSP# support dedicated syntax which is unavailable in the other. Some special syntax operators in one can be indirectly achieved in the other. For instance, the *CHAOS* process in CSP$_M$ can be defined in CSP# using choices and event prefixing (discussed in the next section). Nonetheless, it is not always trivial to support some dedicated syntax operators such as shared variables in CSP$_M$ and channel communications in CSP# (which can involve multiple processes).

3 CSP$_M$ vs. CSP#: Operational Semantics

Operational semantics describes the sequences of computational steps that a model can take. We illustrate the operational semantics of CSP$_M$ and CSP# in the form of labeled transition systems (LTS). An LTS is 3-tuple $\mathcal{L} = (S, \mathit{init}, \rightarrow)$ where S is a set of system configurations; $\mathit{init} \in S$ is an initial system configuration and $\rightarrow$: $S \times \Sigma \cup \{\checkmark, \tau\} \times S$ is a labeled transition relation. Note that $\Sigma \cup \{\checkmark, \tau\}$ is the event space where Σ is the set of *visible* events, $\checkmark$ denotes a successful termination, and τ is an *invisible* event.

A system configuration S in CSP$_M$ is a *pair* of processes and environment where the latter maps variable identifiers to values such as data, processes, or a distinguished *error* configuration. In CSP#, S is composed of two components (V, P) where V maps variable names (or channel names) to values (or sequences of items in buffers), and P is a process expression. The operational semantics of a process construct is depicted by associated firing rule(s). CSP$_M$ and CSP# share very similar firing rules for some process constructs like *interrupt* [15,19,22]. We elaborate subtle differences in the operational semantics of six process constructs; the complete description of *all* different process constructs can be found in our technical report [21]. Note that CSP$_M$ process constructs like P in the firing rules below include the environment, same as [15].

SKIP Process SKIP means termination; namely, $\checkmark$ takes place followed by doing nothing, as captured by Stop in CSP#, whereas this is denoted by a special process term Ω in CSP$_M$. For simplicity, we use prefix M to refer to CSP$_M$ firing rules (e.g., M_skip), and # for CSP# (e.g., $\#\_skip$) in the following.

$$\frac{}{\mathit{SKIP} \xrightarrow{\checkmark} \Omega}\ [\ M\_skip\ ] \qquad\qquad \frac{}{(V, \mathit{Skip}) \xrightarrow{\checkmark} (V, \mathit{Stop})}\ [\ \#\_skip\ ]$$

Notice that in both CSP$_M$ and CSP#, $\checkmark$ may only be the last event of a trace. The semantic difference shown above thus will not result in different verification results in FDR, ProB and PAT[2]. Nonetheless, it should be noticed that this difference leads to a different semantics for parallel composition as we show later.

[2] except deadlock-freeness checking; namely, a process is deadlock free *iff* it satisfies the deadlock-freeness assertion in FDR and ProB, whereas it has to satisfy both deadlock-freeness and nontermination assertions in PAT.

CHAOS Process *CHAOS* in CSP$_M$ denotes the most non-deterministic process.

$$\frac{}{\mathit{CHAOS}(A) \xrightarrow{\tau} STOP}\ [\ M\_c1\ ] \qquad\qquad \frac{a \in A}{\mathit{CHAOS}(A) \xrightarrow{a} \mathit{CHAOS}(A)}\ [\ M\_c2\ ]$$

$\mathit{CHAOS}(A)$ is not directly supported by CSP# because of two main reasons. First, users have to specify all the events in set A to model *CHAOS*, whereas CSP# is designed to free users from specifying events associated with processes (if possible). Second, *CHAOS* is more useful in the failures/divergence checking, whereas CSP# models focus more on states/LTL checking. $\mathit{CHAOS}(A)$ can be manually captured in CSP# by constructing an equivalent process including all events. For example, let set A contains events a and b, one way to model $\mathit{CHAOS}(A)$ process in CSP# can be as follows.

$$\mathit{CHAOS\_A} = tau \rightarrow Stop\ []\ a \rightarrow \mathit{CHAOS\_A}\ []\ b \rightarrow \mathit{CHAOS\_A}$$

Channel communication Channel communications are crucial in concurrent systems and they are classified into two types: *synchronous* and *asynchronous*. CSP$_M$ directly supports the former, whereas CSP# supports both. Both languages have their own operational semantics to interpret channel communications, which is elaborated below. The transformation of channel communication between CSP$_M$ and CSP# is discussed later.

A general format to express a channel communication is $cf \rightarrow P$, where c is a channel name, f a sequence of communication fields, and P a process with the scope of the prefix. A communication field can be an output (by $!e$ where e is an expression), an unconstrained input (by $?x$ where x is a variable), or a constrained input (by $?x : V$ in CSP$_M$ where V is a value range, and by $?[b]x$ in CSP# where b is a Boolean condition).

In CSP$_M$, channels are synchronous and communications are achieved by means of event synchronization. Specifically, assume the type of data communicated over channel c is T, $c!e \rightarrow P$ outputs a communication $c.v$ where v is the value of e and $v \in T$, and $c?x \rightarrow P$ accepts an input of the form $\{c.v \mid v \in T\}$; $c?x : V \rightarrow P$ imposes an additional constraint for $c.v$, namely, $v \in V$. As a channel can be associated with a sequence of communication fields in CSP$_M$, multi-part communications involving multiple data transfers can occur within a single action. For instance, $c?x : V!e \rightarrow P$ engages communications of the form $\{c.v'.v \mid v'.v \in T \wedge v' \in V\}$ where v is a value of e. The firing rule of the CSP$_M$ channel communication is presented below, where function $comms(cf)$ returns the set of communications described by cf and function $subs(a, cf, P)$ returns a process whose identifier in process P bounded by cf is substituted by event a.

$$\frac{a \in comms(cf)}{cf \rightarrow P \xrightarrow{a} subs(a, cf, P)}\ [\ M\_com\ ]$$

In CSP#, a channel is defined as a buffer which stores messages in a first-in-first-out (FIFO) order. Channels are synchronous when their buffer sizes are zero, in which case communications are realized by the hand shaking mechanism. Channels are asynchronous when their buffer sizes are bigger than zero, and their communications are achieved by the message passing mechanism. Sending and receiving multiple messages at one time are supported in both synchronous and asynchronous communications. In addition, the data type of the messages are *untyped* in CSP#. We show below the firing rules of CSP# for channel communications.

– A synchronous communication occurs when both processes $c!e \rightarrow P$ and $c?x \rightarrow P$ (or $c?[b]x \rightarrow P$) can be executed *simultaneously* and the messages passed match (and condition b is true); event $c.v$ is transferred where v is the value of e with the *latest* valuation $eva(V, e)$. In the following firing rule which is associated with parallel composition (the case for interleaving is similar), process $Q[eva(V, e)/x]$ replaces x with the new value v.

$$\frac{(V, c!e \rightarrow P) \overset{c!eva(V,e)}{\rightarrow} (V, P),\ (V, c?[b]x \rightarrow Q) \overset{c?[b]x}{\rightarrow} (V, Q),\quad (V \wedge x = eva(V, e)) \Rightarrow b}{(V, c!e \rightarrow P \parallel c?[b]x \rightarrow Q) \overset{c.eva(V,e)}{\rightarrow} (V, P \parallel Q[eva(V, e)/x])}\ [\ \#\_par1\ ]$$

– An output process $ac!e \rightarrow P$, whereac is an asynchronous channel, is enabled if the associated buffer is not full. The process first evaluates e and then pushes the value into the tail of respective buffer for ac (denoted by function $app(V, ac!e)$), followed by the execution of P.

$$\frac{ac \text{ is not full in } V}{(V, ac!e \rightarrow P) \overset{ac!eva(V,e)}{\rightarrow} (app(V, ac!e), P)}\ [\ \#\_out\ ]$$

– A constrained input process $ac?[b]x \rightarrow P$ is enabled if the associated buffer size is not empty and b is valid with the latest valuation (denoted by function $top(ac)$). The process pops (denoted by function $pop(V, ac?x)$) and assigns the top element from the buffer to x, followed by the execution of P. Note that the checking of b is unnecessary for an unconstrained input process.

$$\frac{ac \text{ is not empty in } V\ \wedge (V \wedge x = top(ac)) \Rightarrow b}{(V, ac?[b]x \rightarrow P) \overset{ac?top(ac)}{\rightarrow} (pop(V, ac?x), P[top(ac)/x])}\ [\ \#\_in\ ]$$

Example 1. We exemplify below how CSP# captures CSP$_M$ multi-part synchronous channels and how CSP# asynchronous channels are represented in CSP$_M$. The event-like channel communication in CSP$_M$ can be modeled as alphabetized event-based synchronization in CSP#. We capture the channel communication by expanding the channel values according the type values. Specifically, an output process $c!e \rightarrow P$ is translated to a process $c.e \rightarrow P$ in CSP#, and an input process is transformed into a CSP# model which enumerates *all* possible communications using the general choices ([]) to combine relevant event prefixing processes. Taking the following CSP$_M$ model of a vending machine (VM) as an example,

1. $datatype\ Drink = Sprite \mid Coke \mid Tea \mid Coffee$
2. $channel\ offer : Drink$
3. $VM = offer?x : diff(Drink, \{Coffee\}) \rightarrow VM$

where process VM can perform any communication in the form $\{offer.x \mid x \in diff(Drink, \{Coffee\}) \wedge x \in Drink\}$; function $diff(Drink, \{Coffee\})$ restricts that a vending machine can offer any drink except coffee. This VM can be captured by the following CSP# process where all possible communications are explicitly specified.

$$VM = offer.Sprite \rightarrow VM\ []\ offer.Coke \rightarrow VM\ []\ offer.Tea \rightarrow VM$$

An asynchronous channel in CSP# can be modeled as a CSP$_M$ process which represents the FIFO buffer by sending/receiving messages to/from other processes. We provide such a CSP$_M$ process below, where a sequence is defined in process $Buffer$ to store the message in the FIFO order and rcv and snd are channels.

$$\begin{array}{ll}
1. & Buffer(c, \langle\rangle, N) = rcv?c?x \rightarrow Buffer(c, \langle x\rangle, N) \\
2. & Buffer(c, s \frown \langle a\rangle, N) = \#s < N-1 \& rcv?c?x \rightarrow Buffer(c, \langle x\rangle \frown s \frown \langle a\rangle, N) \\
3. & \qquad\qquad\qquad\qquad [] \; snd!c!a \rightarrow Buffer(c, s, N)
\end{array}$$

In the above $Buffer$ process, line 1 describes the situation where the buffer is empty, namely, only receiving messages from other process is allowed. Lines 2 and 3 depict message receiving and sending when the buffer is not full. This $Buffer$ process can be used to run in parallel with other process, say P, to perform asynchronous channel communication; for instance, a communication over an asynchronous channel ac with buffer size 2 can be modeled as $P[snd \leftrightarrow rcv, rcv \leftrightarrow snd]Buffer(ac, \langle\rangle, 2)$. We remark that asynchronous channel can be regarded as a special kind of shared variable, which is discussed in the next section; the way that asynchronous channels are modeled in CSP$_M$ is similar to handle shared variables in CSP$_M$ later.

Shared variables Shared variables are important in modeling shared resources. Variables in Hoare's CSP processes are local and disjoint. We elaborate below how shared variables are supported by CSP# directly and CSP$_M$ indirectly.

CSP# uses shared variables to model data states and operations in a procedural style. The operations are modeled as terminating sequential programs in the form $a\{prog\} \rightarrow P$, where programs $prog$ can contain local variables[3], if-then-else statements, while loops, the invocation of external libraries written in C#/Java (through the *reflection* techniques). The execution of the programs is atomic together with the occurrence of associated events. In the following firing rules, function $upd(V, prog)$ returns a modified valuation function according to the particular semantics of the program; in $prog$, both shared and local variables can be used and updated.

$$\frac{}{(V, a\{prog\}^4 \rightarrow P) \xrightarrow{a} (upd(V, prog), P)} \; [\ \#\_dataOp\ ]$$

Shared variables can be modeled in CSP$_M$ indirectly as discussed in [17]. To be specific, a shared variable is represented by a *variable process* which is executed concurrently with other *user processes* which invoke the variable. Variable processes are modeled as read/write operations, and hence user processes can read from/write to the shared variables by CSP$_M$ synchronous communication. For example, the following processes $Var(v, val)$ and $Var\_A(j, v, val)$ execute together as a variable process to denote a shared variable v, where val is the value of v and j denotes a unique id of a user process which invokes v. The constraint that only one process is allowed to read/write v is specified in Var_A which is triggered by event $start\_at?j!v$ from Var.

$$\begin{array}{ll}
1. & Var(v,\ val) \ = \ read?i!v!val \ \rightarrow \ Var(v,\ val) \\
2. & \quad [] \; write?i!v?x \ \rightarrow \ Var(v,\ x) \; [] \; start\_at?j!v \ \rightarrow \ Var\_A(j,\ v,\ val) \\
3. & Var\_A(j,\ v,\ val) \ = \ read.j!v!val \ \rightarrow \ Var\_A(j,\ v,\ val) \\
4. & \quad [] \; write.j!v?x \ \rightarrow \ Var\_A(j,\ v,\ x) \; [] \; end\_at?j!v \ \rightarrow \ Var(v,\ val)
\end{array}$$

[3] The scope of local variables is within $prog$, and they are not stored in valuation function V.

[4] Event a can also be an invisible event, denoted as tau, then the transition event becomes τ.

Example 2. The following CSP# model and CSP$_M$ model represent the same system which sums three process parameters, where the processes are selected non-deterministically from three processes. In the CSP# model below, sum and $count$ are shared variables with initial value 0, and their updates are executed atomically with the occurrence of event add in process $P(i)$.

1. $var\ count\ =\ 0;\quad var\ sum\ =\ 0;$
2. $P(i)\ =\ [count\ <\ 3]add\{sum = sum + i;\ \ count = count + 1;\ \}\ \rightarrow\ P(i);$
3. $System()\ =\ |||\ \ i : \{1..3\}@P(i);$

In the CSP$_M$ model, the shared variables sum and $count$ are modeled as variable processes $Var(sum, 0)$ and $Var(count, 0)$. In addition, process $P(i)$ is defined (lines 2 to 4) by a sequence of variable access events (e.g., events $start\_at!i!count$ and $end\_at!i!count$ for $count$).

```
1.  $datatype\ VarDt\ =\ count\ |\ sum\quad T = \{1..3\}\ \ Range\ =\ \{0..10\}$
2.  $P(i)\ =\ start\_at!i!count\ \rightarrow\ read!i?count?x\ \rightarrow\ x < 3\ \&\ add$
3.        $\rightarrow\ start\_at!i!sum\ \rightarrow\ read!i?sum?y\ \rightarrow\ write!i!sum!(y + i)$
4.        $\rightarrow\ write!i!count!(x + 1)\ \rightarrow\ end\_at!i!sum\ \rightarrow\ end\_at!i!count\ \rightarrow\ P(i)$
5.  $Processes()\ =\ |||\ \ i : \{1..3\}@P(i)$
6.  $Variables()\ =\ Var(count,\ 0)\ |||\ Var(sum,\ 0)$
7.  $SharedEvent\ =\ \{read.t.v.val,\ write.t.v.val,\ start\_at.t.v,\ end\_at.t.v\ |$
8.                       $t \leftarrow T,\ v \leftarrow VarDt,\ val \leftarrow Range\}$
9.  $System()\ =\ Variables()\ [|\ SharedEvent\ |]\ Processes()$
```

As shown above, CSP# allows users to specify shared variables and their operations in a way similar to imperative programming languages, which allows users to see variable states at each simulation step. In contrast, CSP$_M$ supports shared variables by the means of auxiliary processes and events; the additional operations may result in more system states during model checking, as shown later in our experiments.

Parallel composition The firing rules of parallel composition $P\ \|\ Q$ in CSP$_M$ and CSP# are similar except the way of handling the ✓ event. Both languages require *distributed termination*: process $P\ \|\ Q$ terminates if both P and Q terminate. This requirement is satisfied in CSP# by the following firing rule.

$$\frac{(V, P) \xrightarrow{\checkmark} (V, P'), (V, Q) \xrightarrow{\checkmark} (V, Q')}{(V, P \parallel Q) \xrightarrow{\checkmark} (V, Stop)}\ [\ \#\_par2\ ]$$

In addition, CSP$_M$ allows the termination of a paralleled process to be independent of its associated process. Firing rules $[M\_par1]$ below describes that the termination of P involves an invisible event τ and P becomes Ω; operator $\underset{X}{\|}$ is a general form of three kinds of parallel operators in CSP$_M$.

$$\frac{P \xrightarrow{\checkmark} P'}{P \underset{X}{\|} Q \xrightarrow{\tau} \Omega \underset{X}{\|} Q}\ [\ M\_par1\ ] \qquad\qquad \frac{}{\Omega \underset{X}{\|} \Omega \xrightarrow{\checkmark} \Omega}\ [\ M\_par2\ ]$$

The firing rule for Q is similar to $[M\_par1]$. When both processes become Ω, the parallel process terminates under the firing rule $[M\_par2]$. Notice that the verification results especially on non-terminating checking of parallel composition in CSP$_M$ and CSP# are the same although the former needs two more steps. Parallel processes involving synchronous channels in CSP# have been discussed early in Section 3 (by the firing rule $[\#\_par1]$). Parallel processes involving asynchronous channels execute independently and their firing rules can be found in our technical report [21].

Renaming CSP$_M$ supports *renaming* which renames a visible event when an associated process is running, shown in the rule $[M\_r3]$. In theory, event renaming $P[[R]]$ can be represented in CSP# by a process Q which is *almost* the same as P except the visible event from relation R being replaced. However, modeling the renaming process manually in CSP# may not be easy when the renaming relation is complicated, and it may lead to larger (LOC) specifications.

$$\frac{P \xrightarrow{\tau} P'}{P[[R]] \xrightarrow{\tau} P'[[R]]}\,[\,M\_r1\,] \qquad \frac{P \xrightarrow{\checkmark} P'}{P[[R]] \xrightarrow{\checkmark} \Omega}\,[\,M\_r2\,] \qquad \frac{P \xrightarrow{a} P',\, a\; R\; b,\, a, b \in \Sigma}{P[[R]] \xrightarrow{b} P'[[R]]}\,[\,M\_r3\,]$$

Discussion We have identified differences between CSP$_M$ and CSP# in terms of their operational semantics, and also discussed some possible translations between these two languages, especially their channel communications. Through the analysis, we can draw some general guidelines of their modeling features: CSP$_M$'s adoption of functional paradigm and support of more primitives such as *CHAOS* and *renaming* provide an approach to specify concurrent systems like this, starting with an abstract model first, then refining it to more concrete one. CSP# supports more primitives for modeling different forms of communication (e.g., message passing), and it is feasible to specify concrete system behaviors which require hand shaking, message passing and shared resources. In term of expressiveness, it can be shown that CSP$_M$ and CSP# are equivalent as both CSP$_M$ and CSP# process can be transformed into a normal form, which involves event-prefixing, internal choice and recursion only [15].

4 Verification Tool Support

CSP$_M$ is supported by FDR which is designed primarily for refinement checking in terms of trace, failures, divergences, refusals and revivals. ProB was initially designed as an animator and model checker for B method [1], and recently it supports CSP$_M$ with improvements on static type checking and associative tuples [7]; ProB integrates type checking, animation and model checking together. CSP# is supported by PAT which is an extensible framework for system modeling, simulation and verification. PAT implements a number of model checking techniques catering for different properties such as LTL properties and refinement checking. In the following, Section 4.1 illustrates the verification capabilities of FDR, ProB (for CSP$_M$) and PAT (only its CSP module), including properties supported and their model checking techniques; Section 4.2 investigates the efficiency of the three tools.

4.1 Verification

FDR, ProB and PAT support the analysis of many common properties such as deadlock, livelock, determinism, and refinement checking which includes trace, failure and failures/divergences refinement. In addition, FDR supports two additional refinement models: the refusal testing model and the revivals model [10]. PAT supports additional properties like *reachability analysis*, i.e., if a system can reach a bad state (e.g., array overflow).

Model checking LTL properties is common in practice. Although it is not directly supported in FDR, the relationship between refinement checking and LTL model checking has been studied (e.g., [16,11]). Particularly, Leuschel et al. [8] applied an emptiness test in a refinement between an unexpected specification and a process; the process is a synchronization of the implementation and a CSP process for an LTL formula. This approach has to deal with the high complexity of synchronization in FDR, and the process to construct CSP processes from LTL formulas is arduous. Lowe [9] used a refusal testing model to conduct the refusal refinement between a CSP process which denotes an LTL formula and its implementation; those supported LTL formulas exclude operators eventually ($\diamond$), until ($\mathcal{U}$), and negation. In contrast, ProB and PAT support various LTL formulas and analysis directly. Moreover, these formulas can constrain both states and events, and be analyzed under five types of fairness assumptions [23] in PAT.

FDR, ProB, and PAT all provide basic model checking techniques such as breadth first search and (bounded) depth first search. In addition, PAT implements the antichain approach in which the complete subset construction and computing the complete state space of the product are avoided for checking refinement. Further, PAT applies Loop/SCC searching algorithm for LTL verification under fairness assumptions. To cope with the problem of state space explosion during verification, FDR and PAT develop their own reduction techniques. To be specific, FDR proposes a hierarchical compression approach consisting of six methods to process an LTS representing a CSP$_M$ model [10,15,17]: enumerations, strongly node-labeled bisimulation, τ-loop elimination, diamond elimination, normalization, and factoring by semantic equivalence. On the other hand, PAT deploys three techniques. First, using the atomic sequence construct (denoted by $atomic\{P\}$), where a sequence of statements in a process executes as one *super-step* without any inference, to realize simple partial order reduction (POR). Second, applying POR dedicated to refinement checking to not only τ transitions but also visible events (in some case which is not supported in FDR [23]). Last but not least, providing process counter abstraction for parameterized systems under fairness against LTL formulas [24]. We remark that the implementation of FDR's hierarchical compression methods for CSP# in PAT is nontrivial due to shared variables supported in CSP#. For instance, a τ event in CSP# may update shared variables and therefore the event cannot not be pruned for compression.

4.2 Experiment

In this section, we evaluate the efficiency of FDR, ProB and PAT by verifying nine benchmark systems. The experiments with FDR and ProB are performed on an Intel$^{®}$

Model	N	Property	FDR		ProB		PAT	
			State	Time(s)	State	Time(s)	State	Time(s)
R/W	6	P [T= S	8	0.024	61365	125.94	9	0.04
R/W	200	P [T= S	202	1.434	-	-	203	0.11
R/W	500	P [T= S	502	19.651	-	-	503	0.057
R/W	1000	P [T= S	1002	156.162	-	-	1003	0.108
DP	6	P [F= S	1	0.06	14510	82.42	1762	0.174
DP	8	P [F= S	1	0.071	-	-	22362	2.995
DP	12	P [F= S	1	0.104	-	-	-	-
MCS	20	P [FD= S	40	0.043	-	-	60	0.114
MCS	50	P [FD= S	100	0.086	-	-	150	0.143
MCS	100	P [FD= S	200	0.246	-	-	300	0.53

Table 2. Experiment results on refinement checking

CPU E6550 (2.33 GHz) PC with 4GB memory running on 32-bit Linux. PAT is experimented with the same PC but on a 32-bit Windows.

We conduct five sets of experiments[5]. The first set investigates the performance of refinement checking, by verifying the same model and assertion with different reduction techniques. The results are shown in Table 2, where N is the number of processes. Column $State$ shows the number of visited states, and column $Time(s)$ records running time of the verification in seconds. Value "-" in a cell denotes that the experiment is aborted due to either memory overflow or execution time exceeding two hours. For readers/writers (R/W) models, although FDR applies some dedicated compression techniques, PAT has better performance. For dining philosopher (DP) models, FDR performs extremely well because of the strategy discussed in [18]. However, other experiments show that this strategy may not be as efficient for other models. For Milner's cyclic scheduler (MCS), PAT is comparable to FDR in terms of the number of states per second. FDR processes the LTS by applying its compression methods, whereas PAT applies a simple reduction method, i.e., using the keyword *atomic* to give higher priority to local events which are not synchronized, not updating any variable and not mentioned in the property.

The second set compares the performance of three model checkers on solving puzzles, inspired by work in [12]. The CSP$_M$ and CSP# models for these puzzles make the best use of their modeling power: CSP# specifies the puzzles using shared variables, which are solved by PAT through reachability analysis, whereas CSP$_M$ models the puzzles using multi-part event synchronization, which are solved by FDR and ProB through trace refinement. In addition, FDR simulates a bounded DFS algorithm by searching the divergence of a new system, in order to find a smaller counterexample. The new system P', like a watchdog, can only perform up to N events of the target implementation process P, and then performs an infinite number of events [**?**]. This approach can be used provided that the target process P is loop-free. Table 3 shows the performance results, where column $FDR\text{-}Div$ records the results of states and time using this algorithm; value $N.A.$ means there is no model with divergence checking to solve the puzzle. From Table 3, we can observe that the divergence checking approach can be used in the solitaire and chess knight tour models. However, this approach can-

[5] All models are available at `www.comp.nus.edu.sg/~pat/compare`.

Model	N	FDR		FDR-Div		ProB		PAT	
		State	Time(s)	State	Time(s)	State	Time(s)	State	Time(s)
Solitaire	26	4048216	46.303	1	0.169	-	-	11950	5.356
Solitaire	29	28249254	387.737	1	0.217	-	-	104395	54.681
Solitaire	32	-	-	1	5.318	-	-	10955	5.301
Solitaire	35	-	-	1	377.297	-	-	443230	279.454
Knight	5	508450	3.522	1	0.037	-	-	4256	0.29
Knight	6	-	-	1	15.399	-	-	129269	9.143
Knight	7	-	-	1	94.713	-	-	77238	6.754
Hanoi	6	729	0.052	N.A.	N.A.	1667	57.84	5775	0.416
Hanoi	7	2187	0.086	N.A.	N.A.	4969	196.5	92680	6.837
Hanoi	8	6561	0.181	N.A.	N.A.	14853	660.59	150918	11.524

Table 3. Experiment results on solving puzzles

not always significantly improve the performance, because it depends on the searching order. Moreover, it is costly to check if a system is loop-free or not, which is the premise for applying this approach. PAT solves the two puzzles in a reasonable time, and it is faster in the knight example than FDR and FDR-Div. For the hanoi puzzle, FDR has a better performance because the compression techniques it uses can effectively reduce the state space.

The third set explores the performance of FDR and PAT on verifying two models which involve shared variables. The first example is a concurrent stack which allows multiple readers to access the shared variable at the same time, but only one writer to update the value; readers cannot access the shared variable in the latter case. The modeling of shared variables in CSP$_M$ follows the approach discussed in Section 3. Results of this example in Table 4 show that PAT performs better than FDR for checking trace refinement (P[T=S), and this is because PAT uses DFS with anti-chain algorithm in the trace refinement. This algorithm is effective when the specification is non-deterministic. Here, N is the number of processes and $ConcurrentStack * 2$ in the *Model* column means that the stack size is 2. The second example is the Peterson algorithm. We obtain the CSP$_M$ model from the shared variable analyzer (SVA) [17]. To be fair, the CSP# model is specified at the same level of granularity as the CSP$_M$ model. The results show that PAT performs better. This is because local events associated as atomic statements in CSP# reduce the states significantly, whereas CSP$_M$ model defines additional events to represent reading/writing operations of shared variables. Although these additional events can be hidden as internal events to apply existing compression techniques in FDR, the effect is minor because the type range of reading/writing channels and operations over different variables can easily lead to state space explosion.

The forth set explores the performance on verifying LTL properties. We adopt the approach proposed by Lowe [9] to construct a CSP$_M$ process for the LTL formula and use FDR to perform the refusal refinement checking. As this approach cannot deal with operator *eventually* ($\Diamond$), we ignore the checking of property $\Box\Diamond eat.0$ in FDR. Table 5 indicates that PAT performs better than FDR and ProB. Notice that property $\Box\Diamond eat.0$ can be verified to be true using PAT under the strong or global fairness assumption.

Last but not least, we conduct a case study on translating CSP$_M$ model to CSP# through a real world problem, namely, the battery monitor component of the elevator

Model	**N**	**Property**	**FDR**		**PAT**	
			State	**Time(s)**	**State**	**Time(s)**
Concurrent Stack*2	3	P [T= S	453456	3.833	10860	1.023
Concurrent Stack*2	4	P [T= S	-	-	189920	75.915
Concurrent Stack*2	5	P [T= S	-	-	693828	293.382
Peterson	3	mutual exclusion	1011	1.192	3257	0.105
Peterson	4	mutual exclusion	105493	20.067	104686	3.776
Peterson	5	mutual exclusion	14810779	387.645	5722863	294.005

Table 4. Experiment results on shared variables

Model	**N**	**Property**	**Result**	**FDR**		**ProB**		**PAT**	
				State	**Time(s)**	**State**	**Time(s)**	**State**	**Time(s)**
RW	6	$\Box !error$	true	8	0.023	122722	104.8	15	0.059
RW	200	$\Box !error$	true	202	1.455	-	-	403	0.086
RW	500	$\Box !error$	true	502	19.901	-	-	1003	0.071
RW	1000	$\Box !error$	true	1002	154.33	-	-	2003	0.148
DP	6	$\Box\Diamond eat.0$	false	N.A.	N.A.	2420	1.11	166	0.019
DP	8	$\Box\Diamond eat.0$	false	N.A.	N.A.	13312	1.75	256	0.024
DP	12	$\Box\Diamond eat.0$	false	N.A.	N.A.	-	-	460	0.049

Table 5. Experiment results on LTL checking

control system described in [5]: the CSP$_M$ specification is translated from the battery monitor Simulink diagram, and the CSP# model is translated from the CSP$_M$ model. Then we analyze the Simulink diagram, using the Simulink simulator, to determine which output the battery monitor should produce for every given input. Thus, to assess if both models describe the same behaviour, we compose each one with parallel observers. We noticed that both CSP$_M$ and CSP# models provide the same output value for all relevant scenarios. Although both models were reviewed by CSP$_M$ and CSP# specialists, a formal proof of equivalence will be provided in our future work. Besides this analysis, we also checked basic properties like deadlock freeness and divergence freeness. The comparison was performed as a controlled experiment and we ran each assertion 30 times. By applying common statistics testing methods (particularly, Shapiro Wilk and Mann-Whitney U [20]) to experiment data, we can state that the difference between PAT and FDR performance is large. From Table 6, we can observe that the visited states in FDR and PAT are the same, and performance of these two tools is similar using the same verification algorithm. Note that the time for deadlock-freeness property in PAT consists of time for deadlock and non-terminating checking. We have not included ProB in this comparison as it is unable to recognize the CSP$_M$ syntax for this example.

Discussion We have explored the supporting tools of CSP$_M$ and CSP#, namely, FDR, ProB and PAT, by comparing their model checking techniques and analyzing their verification capabilities through nine benchmark systems. Our exploration leads to the following four general and practical rules for choosing these tools. First, FDR can be the best candidate when powerful built-in compression techniques are applicable in refinement checking. Second, PAT is a better choice to verify properties of models which involve shared variables. Third, to verify LTL properties, we can use ProB for CSP$_M$ models or FDR for some model where LTL formula can be verified by refusal checking,

Property	Result	FDR		PAT	
		State	Time(s)	State	Time(s)
Deadlock-free	True	2700	0.286	2700	0.748
Livelock-free	True	2700	0.296	2700	3.723

Table 6. Experiment results on battery monitor

and PAT for CSP# model. Lastly, PAT may be a better option to handle models where atomic reductions are applicable (e.g., readers/writers and Peterson algorithm).

5 Conclusion

In this work, we presented a comprehensive comparison of CSP$_M$ and CSP#, and their supporting tools FDR, ProB and PAT. We explored their modeling features from the view of their syntax and operational semantics. We also investigated the reasoning power of CSP$_M$ and CSP# in terms of the capability and efficiency of their supporting tools. Our work can guide users to select and assess appropriate modeling languages and reasoning tools for specifying and verifying concurrent systems. 1) CSP$_M$ may be more suitable to model systems with abstract behavior, and systems which involve multi-part event synchronization. On the other hand, CSP# could be a better candidate to handle systems which implement hand shaking or message passing communication mechanisms, and systems which need shared variables. 2) To perform the refinement checking, the decision relies on the reduction techniques which are more applicable (compression methods in FDR, atomic reduction in PAT) to the models. To verify LTL properties, we can use ProB for CSP$_M$ models or FDR for some model (discussed in Section 4), and PAT for CSP# models. Lastly, PAT may be a better option to verify systems with shared variables.

As for related work, Carvalho et al. have made an initial step to explore the differences between CSP$_M$ and CSP# [2]. They compare the two languages from the data and behavioral aspects. Our work here substantially extends their step by considering an in-depth and a wider range of comparisons; for instance, we investigate their intrinsic differences from the operational semantics aspect. Roscoe has briefly described tools which can animate, analyze, and verify CSP models[6]; these tools include FDR, ProB, PAT, ARC [13] and so on. He introduces these tools with strengths and limits from a high level. Our work can be considered as a concrete guideline for these tools, in particular, FDR, ProB for CSP$_M$, and PAT for CSP#, with intensive experiments.

The comparison of FDR, ProB and PAT so far has been focusing on the classical model checking techniques. In the future, we plan to extend the comparison to other techniques such as SAT-based FDR and BDD-based PAT. Proofs of the semantic equivalence of the translations and implementations of the translators are also our goals.

Acknowledgment The authors would like to thank Bill Roscoe for the review and suggestions on benchmarks for FDR, Michael Leuschel for the help on using ProB, and Augusto Sampaio, Alexandre Mota and Tarciana Dias for the valuable comments.

[6] The description is at `http://www.cs.ox.ac.uk/ucs/CSPtools.html`.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, USA, 1996.
2. G. H. P. Carvalho, T. Dias, A. Mota, and A. Sampaio. Analytical comparison of refinement checkers. In *SBMF*, pages 61–66, 2011.
3. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
4. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.
5. J. Jesus, A. Mota, A. Sampaio, and L. Grijo. Architectural verification of control systems using CSP. In *ICFEM*, pages 323–339, 2011.
6. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME*, pages 855–874, 2003.
7. M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. pages 278–297, 2008.
8. M. Leuschel, T. Massart, and A. Currie. How to make FDR Spin LTL model checking of CSP by refinement. In *FME*, pages 99–118, 2001.
9. G. Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Form. Asp. Comput.*, 20(3):277–294, May 2008.
10. F. S. E. Ltd. *Failures-Divergence Refinement - FDR2 User Manual (version 2.91)*.
11. T. Murray. On the limits of refinement-testing for model-checking CSP. *Form. Asp. Comput.*, pages 1–38, 2011.
12. H. Palikareva, J. Ouaknine, and A. W. Roscoe. Faster FDR counterexample generation using SAT-solving. *ECEASST*, 23, 2009.
13. A. N. Parashkevov and J. Yantchev. ARC - a tool for efficient refinement and equivalence checking for CSP. In *ICA3PP*, pages 68–75, 1996.
14. A. Roscoe. CSP is Expressive Enough for Pi. 2010.
15. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.
16. A. W. Roscoe. On the expressive power of CSP refinement. *Form. Asp. Comput.*, 17:93–112, August 2005.
17. A. W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., 2010.
18. A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS*, pages 133–152, 1995.
19. B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, 1998.
20. S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 3(52), 1965.
21. L. Shi. An Analytical and Experimental Comparison of CSP Extensions and Tools. Technical report, NUS, 2012. `http://www.comp.nus.edu.sg/~pat/compare`.
22. J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *TASE*, pages 127–135, 2009.
23. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 702–708, 2009.
24. J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In *FM*, pages 123–139, 2009.
25. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.