Analytical Comparison of Refinement Checkers

Gustavo H. P. Carvalho, Tarciana Dias,
Alexandre Mota, and Augusto Sampaio

Informatics Center, Federal University of Pernambuco,
Pernambuco, 50740-560, Brazil
{ghpc,acm,acas}@cin.ufpe.br
tarciana.dias@gmail.com

Abstract. *The process algebra CSP is one of the most famous concurrent specification languages and it is based on a refinement theory, which served to create the standard refinement checker FDR2. In recent years PAT is being claimed as a promising alternative. In this work we perform a comparison between these tools based on their behavioral and data aspects. Due to the absence of a complete operational semantics definition for CSP# (PAT's input language), behavioral analysis is based on the comparison of the corresponding state machines. We realized that CSP# specifications have to be adjusted to preserve the semantics of parallelism and determinism already provided by CSP_M. This results in bigger state machines for CSP# specifications. Furthermore, CSP# is based on an imperative paradigm whereas CSP_M uses a more expressive functional paradigm. As a conclusion we show that it is necessary to perform many adaptations to have an equivalent CSP# specification to a given CSP one.*

Keywords: Process algebra, model checker, CSP, FDR2, PAT

1 Introduction

Traditionally, FDR2 [2] has been used as a standard model checker for the process algebra CSP, although in recent years Process Analysis Toolkit (PAT) [1] is being claimed as a promising alternative. PAT is a self-contained framework for composing and simulating concurrent, real-time and probabilistic systems, where each such an aspect is represented by selecting the corresponding PAT module. The PAT language for specifying concurrent processes is also known as CSP#.

Assuming CSP_M as the reference machine-readable language for CSP, we perform a systematic comparison between CSP_M and CSP# by showing how to capture behavioral and data constructs of CSP_M in CSP#. Thus, for each CSP_M construct, we first provide the expected CSP# equivalent description. Then we check whether equivalence really holds. Finally we suggest alternatives when differences are identified. All analysis described here were done considering the PAT 3.3.0 version (build 16884) and the FDR2 2.9.1 academic version.

We assume here a certain knowledge on the process algebra CSP although we provide some explanation along the text. This work is structured as follows:

Section 2 compares the support of types and data structures provided by CSP$_M$ and CSP#. Section 3 carries out a behavioral comparison of CSP$_M$ and CSP#. Section 4 compares this work with similar initiatives. Finally, Section 5 provides our final remarks and future work.

2 CSP$_M$ and CSP# Analysis: Data Perspective

CSP$_M$ supports a functional paradigm whereas CSP# is based on an imperative one. The limitation imposed by the imperative paradigm adopted by CSP# is the prohibition of passing processes, functions and channels as arguments of processes. Therefore, the definition of more generic processes (processes with other processes and channels as parameters) is not possible in CSP#. This can create larger specifications due to replication of code.

CSP$_M$ allows user defined data types via the reserved word "datatype". CSP# needs C# integration based on restricted C# classes. It is worth noting that CSP# allows this C# integration but the CSP# constructions can only deal with integers, booleans and integer arrays. Therefore, for instance, one cannot communicate a user data defined instance.

CSP$_M$ supports booleans, integers, tuples, sets, and sequences directly. From these, CSP# only supports booleans and integers directly although it supports arrays that are not directly available in CSP$_M$. Other structures may be declared as C# classes and then used in CSP# code. Manipulations of these structures are allowed: (i) directly in CSP# code via method calls that return an integer, boolean or integer array or (ii) in special CSP# events known as data operations.

3 CSP$_M$ and CSP# Analysis: Behavioral Perspective

In this section we compares the behavioral aspects of CSP$_M$ and CSP#.

3.1 Communication Model

CSP# supports communication through message passing (much like CSP$_M$) as well as shared memory. Thus reading an integer, boolean or integer array shared value is directly supported in CSP#. However, to (i) perform a writing operation or (ii) call a method that does not return an integer, boolean or integer array value, a special structure called data operations must be used. Data operations are encapsulated inside an event. These operations are executed sequentially and atomically. Code 1.1 illustrates this functionality: the assignments performed inside the *update* event are atomically performed. An important remark is: CSP# does not synchronize on events that perform data operations. Hence many possibilities arises, creating a non-deterministic behavior (R process is not deterministic in Code 1.1). This is a CSP# design decision to avoid race conditions on operations.

Code 1.1. CSP# - Data Operations with Common Event Names

```
var x; P()=update{x=2}->P(); Q()=update{x=3}->Q(); R()=P()||Q();
```

3.2 Channel Definition

CSP# channels do not have an explicit type like in CSP$_M$. They are declared by simply considering its name and a buffer_size. If buffer_size is equal to 0, we have a synchronous channel, otherwise it is asynchronous. Furthermore, it is worth observing that CSP# does not assume an environment, like in CSP. That is, CSP# assumes a closed world like PROMELA [3] what is not compositional. For instance, observe Code 1.2. The CSP# processes are not deadlock free. This is a consequence of the open versus closed world assumption: P wants to read but there is no corresponding process (in the case of CSP, the environment provides such a communication) and Q wants to write but there is no corresponding process to read. PAT does not also allow the user to specify communication restrictions based on set pertinence properties, like CSP$_M$ does.

Code 1.2. CSP# - Channel Input and Output

```
channel in 0; channel out 0;
    P() = in?value -> out!value -> P(); R() = out!2 -> R();
```

3.3 Choices

CSP# offers more conditional choice operators than CSP$_M$. Despite this benefit, new different behaviors emerge. For example, the operational semantics of the CSP# if-else construction is different when compared to the CSP$_M$ one. In CSP#, evaluating a condition means performing an event. This does not happen in CSP$_M$. To obtain the same CSP$_M$ behavior, CSP# offers the ifa construction where the condition evaluation does not add a transition to the LTS.

The CSP# internal choice semantics seems to be equivalent to the CSP one. Concerning external choice, CSP# supports two versions: [] (named general choice) and [∗] (named external choice). The CSP$_M$ external choice ([]) corresponds to the [∗] operator of CSP# only. The CSP# general choice operator ([]) has another behavior: the process that performs the first event, even if it is invisible, takes control of the execution flow.

3.4 Event Renaming

A CSP# missing critical feature is process renaming. At least, the official PAT documentation, as well as its grammar do not mention any support concerning this feature (an important CSP operator). The alternative is to replicate the processes and manually rename the desired events. This also limits the definition of more generic processes and leads to a bigger code than the equivalent CSP$_M$.

3.5 Parallel Composition

CSP# does not support multiway rendezvous for channel communication like CSP does. For instance, consider an example with one sender and two receivers. In CSP# the synchronization happens with a pair of processes. There is no control over which synchronization happens. Actually, it is not clear if it is a CSP# design decision or if it is a bug as there is no mention to this fact in the official documentation. To overcome this CSP# limitation we propose an alternative consisted in using asynchronous communications with explicit control events to obtain a similar CSP multiway rendezvous behavior (see Code 1.3): (i) declare asynchronous channels where its buffer size is equal to the number of processes (N) interested in reading the communicated value [line 1]; (ii) the process P, which sends the data, performs N writes to the buffer atomically [line 4]; (iii) while a process P is sending data through a channel, no other process can read or write anything in different channels [the atomic construction in line 4]; (iv) after P finishes its written action, only the processes interested in reading the written values can be engaged [line 3]; (v) P waits all reads to be performed prior to send new data [lines 5, 7–11]; (vi) the engaged processes must read the value in a deterministic order [line 13–14]; (vii) after reading a value, the engaged process needs to wait the others engaged processes to read [line 15]. Despite this solution produces a LTS with more states and transitions, now the specification is equivalent to the multiway rendezvous behavior provided by CSP_M. The specification is now also deterministic and between the sender and receivers communication no other events can occur due to the locks. Thus, we also guarantee the atomicity of multiway-rendezvous.

Code 1.3. CSP# - Parallel Composition

```
 1 channel c 2; var lock=false; var channel_read_counter=0;
 2 Sender() =
 3 ifa ( lock==false ) {
 4  atomic{lock_channel{lock=true}->c!1->c!1->Skip }
 5  ; Sender_Wait_Read()
 6 } else { wait_sender_turn -> Sender() };
 7 Sender_Wait_Read() =
 8 ifa ( channel_read_counter==2 ) {
 9  free_channel_lock{channel_read_counter=0;lock=false}
10  -> Sender()
11 } else { wait_channel_read -> Sender_Wait_Read() };
12 Receiver_0() =
13 ifa ( channel_read_counter==0 ) {
14  atomic{ c?1 -> update_read_0{channel_read_counter++}
15  -> receiver_received -> Skip }
16  ; Receiver_0()
17 } else { wait_receiver_0 -> Receiver_0() };
18 Receiver_1() =
19 ifa ( channel_read_counter==1 ) { ...
20 System() = Sender || (Receiver_0() || Receiver_1());
```

3.6 Assertions

PAT offers the same assertions offered by FDR2: deadlock freedom, determinism and livelock freedom. However, there is one important difference: in FDR2 the deadlock freedom test looks for a Ω state whereas in PAT it looks for states without transitions that were not achieved by a "terminate" event. In other words: a process which terminates with a Skip, and consequently a "terminate" event, is considered deadlock free. To detect this situation, PAT offers another assertion called non-termination. Therefore, the FDR2 deadlock analysis is verified in PAT by the deadlock free and non-termination test. Besides these differences, PAT also supports assertions of Linear Temporal Logic formulas and reachability conditions what is not supported by FDR2.

3.7 CSP Refinement

As well as CSP$_M$, CSP# offers three semantic models for refinement checking: traces, failures and failures divergence. We selected 32 CSP laws described in [5] and verified instances of them in CSP# using refinement checking. Our tests showed that apparently the CSP# refinement checking algorithms work properly. However, we found out one CSP law that is not directly true in CSP#: $(P[|X|]STOP) = STOP, iff X \subseteq \alpha(P)$.

According to the CSP# semantics for the parallel operator: PAT automatically computes the alphabets of both processes and they are free to perform exclusive events of one of them but must synchronize on common events. Therefore, what happens is that, by default, CSP# tries to automatically find out the synchronization alphabet. Then for process P it infers the $\{a\}$ alphabet and for *Stop* the empty alphabet . The intersection of these two alphabets is empty, thus the synchronization alphabet is empty. Therefore, in this case, the parallel synchronization becomes an interleaving of $(P|||STOP) = P$.

To achieve the same CSP behavior, it is necessary to explicitly specify the synchronization alphabet. To do this in CSP#, it is necessary to specify an alphabet for each process using the *alphabet* clause and then the synchronization alphabet is automatically computed as the intersection of these alphabets. Then the initial law refusal relies on the fact that CSP# does not allow an explicit definition of the Stop alphabet. To overcome this limitation, a new process must be declared with a behavior equivalent to Stop. Thus, this process can have its alphabet explicitly declared. See Code 1.4 that behaves like the CSP semantics defined in [5]. Therefore, the CSP# feature of automatically determination of process alphabets can lead to misunderstandings.

Code 1.4. CSP# - Parallel Composition Law

```
#alphabet P {a}; #alphabet P_STOP {a};
P()=a->P(); P_STOP()=Stop;
P1()=(P()||P_STOP()); P2()=Stop;
#assert P1() refines<FD> P2(); #assert P2() refines<FD> P1(); //both true
```

4 Related Work

Some academic works portrayed the comparison between model checkers [4] [6]. However, despite these mentioned PAT and FDR2 analysis, these works focus purely on a quantitative analysis and do not provide a more detailed comparison of the CSP# semantics. Therefore, our approach intended to fill this gap, focusing on analyzing the CSP# and CSP$_M$ semantics and uncovering significant differences between their behavior, as well as the refinement checkers PAT and FDR2.

5 Conclusion

The Process Analysis Toolkit (PAT) provides a tool with better usability to manipulate CSP-like specifications than the FDR2 tool. Besides its good usability, the PAT's CSP module offers some particular interesting features like: manipulation of complex data structures defined in terms of C# and usage of LTL to validate specifications and extensibility of its model checking algorithms. These three aspects are not provided by FDR2. Despite these benefits, we noticed that the claims made in the official PAT documentation is not true: it is necessary to perform many adaptations to have an equivalent CSP# specification to a given CSP one. Unfortunately, due to the absence of a complete operational semantics definition for CSP#, we were unable to formally compare both notations. Therefore, we compared them using an empirical nevertheless systematised perspective, considering each language construct, the semantic models, the refinement checking algorithms, and even the interface environment provided by the tools. As future work we see the following main relevant studies: (i) provide an operational semantics for CSP# and compare it to that of CSP$_M$ and (ii) study a possible (semi)automatic translation between CSP$_M$ and CSP#.

Acknowledgement

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[1]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, by CNPq grant 482462/2009-4 and by the Brazilian Space Agency (UNIESPAÇO 2009).

References

1. PAT: User Manual and Tutorial, version 3.3.0. (2011)
2. Goldsmith, M.: FDR: User Manual and Tutorial, version 2.77. Formal Systems (Europe) Ltd (2001)
3. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
4. Liu, Y., Chen, W., Liu, Y., Sun, J.: Model Checking Linearizability via Refinement. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Lecture Notes in Computer Science, vol. 5850, pp. 321–337. Springer Berlin / Heidelberg (2009)
5. Roscoe, A.: The Theory and Practice of Concurrency. Prentice-Hall (1998)
6. Sun, J., Liu, Y., Dong, J.S.: Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. Leveraging Applications of Formal Methods Verification and Validation pp. 307–322 (2009)

[1] http://www.ines.org.br/