Efficient Analysis of Infinite CSP_Z Specifications

Adalberto Farias, Alexandre Mota, and Augusto Sampaio

Universidade Federal de Pernambuco, Centro de Informática,
Av. Prof. Luis Freire, s/n, Cidade Universitária, Recife,
Pernambuco, Brasil. CEP 50740-540

Abstract. Model Checking is an automatic technique becoming widely used by software industry. However, in some cases, it cannot be applied due to the large size of the systems, especially the infinite ones; this limitation is known as the state explosion problem. Many techniques have been developed in order to avoid such a problem. This work proposes an algorithm, based on a data abstraction approach, which allows Model Checking to verify infinite CSP_Z processes. The algorithm is an optimisation of a previous work, where sequence of events performed by the systems are also analysed.

1 Introduction

Program verification stands for checking whether a given program satisfies a given property, written in a logical language [10]. Several methods and tools are used to accomplish this task; theorem proving and model checking deserve special attention among them. The former is based on deduction and requires user interaction, whereas the latter is automatic because it consists in a search procedure on finite structures. Furthermore, it does not require user support.

Although model checking is completely automatic, it is unable to check infinite or very large systems – a limitation known as the state explosion problem. As an attempt to overcome this problem, various methods and techniques were proposed to transform an infinite system into a finite one, while still preserving most of its properties.

In this work we are interested in *data abstraction*, a technique that consists of using abstract domains and operations instead of the original ones, while still preserving almost all properties of the original system. The language used is CSP_Z [7,8], a combination of the process algebra CSP [16] and the model-based language Z [19]. The model checker Failures Divergences Refinement (FDR) [9] and the theorem prover Z-Eves [18] are also used for proving some properties asked by the algorithm.

A CSP_Z specification has two distinct parts: CSP and Z. The data abstraction we are interested in was proposed by Mota [15] and consists of investigating only the Z part. This can generate more situations to be analysed than those really necessary. Here, we present an improvement which also considers the influence of the CSP part over the Z one (the CSP part can be seen as a master process where the Z part is the slave). Thus, we can improve substantially the efficiency of the algorithm described by Mota [15]. Furthermore, we formally characterise the influence of the CSP part over the Z one.

This work is organised as follows. Section 2 gives a general idea about CSP_Z, including an overview of its semantics. Section 3 presents the technique *data abstraction* and gives a brief explanation about the theory of *data independence*. The two following sections show the algorithm and the improvement proposed by this work. Finally, Section 6 presents our conclusions and future work related to CSP_Z data abstraction.

2 The CSP_Z Language

Reusing theories and tools has been a very effective trend for specifying and analysing complex systems. Consolidated notations and tools can be combined in order to produce more powerful frameworks. A combined language is useful due to the reuse of different expressivenesses. Each component language is more or less adequate for modelling a specific aspect of a system. In concurrent systems modelling, behavioural aspects are better modelled by process algebras like CSP or CCS [13], while data structure aspects are better described by model-based notations like Z or VDM [3].

The CSP_Z language is a conservative extension of CSP and Z, in the sense of preserving almost all expressivenesses of both languages. The idea is using CSP to describe behaviour (process interaction) and Z to describe data structures in a complementary way. We will show CSP_Z through an example.

2.1 CSP Behaviour

Operationally, the CSP behaviour can be defined by a *Labelled Transition System* (LTS), which is a set of nodes and, for each event a in some set, a relation $\xrightarrow{a}$ between nodes. It is a directed graph with a label on each edge representing one event that happens when we take the action which the edge represents. For each operator, there exists a corresponding graph that is easier to understand. Here, we show LTSs for some CSP operators. For a complete description, see [16].

The prefix operator is used for building processes in the following way:

$$P = a \rightarrow b \rightarrow c \rightarrow STOP$$

P is a process which accepts a, performs it and then behaves like $b \rightarrow c \rightarrow STOP$. After that, it performs b and behaves like $c \rightarrow STOP$, and so on. Until $STOP$ (standing for a canonical deadlock) is reached, P performs the following sequence of events: $\langle a, b, c \rangle$. Figure 1 shows its progress.

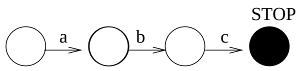

Fig. 1. Prefix operator LTS

External and internal choices can be exemplified, respectively, by:

$P = a \to STOP \; \Box \; b \to c \to STOP$
$P = a \to STOP \sqcap b \to c \to STOP$

In external choice, the process P accepts a or b, depending on the environment synchronisation, whereas in internal choice the process decides whether to perform a *or* b, without considering the environment. Figure 2 shows their LTSs, where τ stands for an internal action.

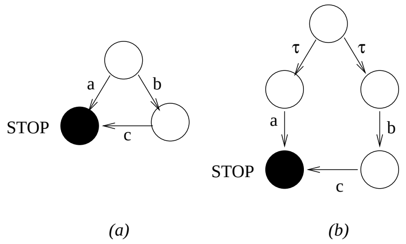

Fig. 2. (a) External choice and (b) Internal choice

Every process has a set of performed events, which is called *alphabet*. The process showed by Fig. 1 has the alphabet $\{a, b, c\}$.

As an LTS describe the behaviour of a process, we can observe that in each step the process is ready to perform (accept) some event. After an event is performed, it accepts another one, and so on. These acceptances can be captured by *initials* as follows:

$initials(STOP) = initials(SKIP) = \{\}$
$initials(a \to P) = \{a\}$
$initials(P \;\Box\; Q) = initials(P \sqcap Q) = initials(P) \cup initials(Q)$

On the other hand, after a sequence of events (trace) has been performed, the process changes its behaviour. This is denoted by P/s, which corresponds to a new LTS resulting from P after performing the trace s. For example:

$(a \to b \to c \to STOP) \,/\, \langle a \rangle = (b \to c \to STOP)$
$(a \to b \to STOP \;\Box\; c \to STOP) \,/\, \langle a \rangle =$
$\quad (a \to b \to STOP \sqcap c \to STOP) \,/\, \langle a \rangle = (b \to STOP)$
$(a \to b \to STOP \;\Box\; c \to STOP) \,/\, \langle c \rangle =$
$\quad (a \to b \to STOP \sqcap c \to STOP) \,/\, \langle c \rangle = STOP$

2.2 Example

The example we use was presented in [20] and represents a simple clock which performs two events: *tick* and *tack*, which are represented by channels:

$$chan\ tick,\ tack$$

The CSP part is represented by a process named *main* containing a behavioural description:

$$main = tick \rightarrow tack \rightarrow main$$

The Z part deals with state definition, system initialisation and operations. Z schemas have their names composed by *com_* prefix and a *channel* name. Each schema execution synchronises with the corresponding event from the CSP part. For example, the schema *com_tick* synchronises with *tick* and *com_tack* with *tack*. The Z part is presented below:

$$\begin{array}{|l}\multicolumn{1}{l}{State} \\ \hline n : \mathbb{N} \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{Init} \\ \hline State' \\ \hline n' = 0 \\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{com\_tick} \\ \hline \Delta State \\ \hline n' = n + 1 \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{com\_tack} \\ \hline \Delta State \\ \hline n' = n + 1 \\ \hline\end{array}$$

The *Init* schema is executed first in order to initialise the system. After that, all enabled schemas (pre condition is true) are ready to execute. Although many schemas can be enabled and the corresponding events are accepted by the CSP part, only one will be executed and then the Z part offers all enabled schemas again.

If either the CSP part has not acceptances or the Z part has not enabled schemas then the system deadlocks.

Grouping the CSP and Z parts and putting them between *spec* and *end_spec* keywords, we have the complete CSP_Z specification:

spec Clock

chan tick, tack

$main = tick \rightarrow tack \rightarrow main$

$$\begin{array}{|l}\multicolumn{1}{l}{State} \\ \hline n : \mathbb{N} \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{Init} \\ \hline State' \\ \hline n' = 0 \\ \hline\end{array}$$

$$\begin{array}{|l}\multicolumn{1}{l}{com\_tick} \\ \hline \Delta State \\ \hline n' = n + 1 \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{com\_tack} \\ \hline \Delta State \\ \hline n' = n + 1 \\ \hline\end{array}$$

end_spec Clock

Figure 3 shows a CSP_Z LTS. From one state to another, only one event happens and the corresponding schema is also executed. Thus, filled arrows

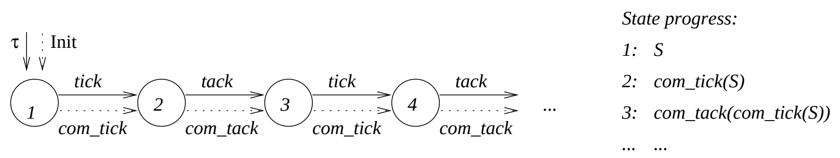

Fig. 3. CSP_Z LTS

represent CSP transitions while schema executions are represented by dotted arrows.

At initialisation, the CSP part performs, nondeterministically, a τ-transition and the Z part executes the *Init* schema. After that, the following behaviour depends on both parts. Thus, a CSP_Z specification can be represented as a process built from both parts:

$$CSP_Z = P_{CSP} \underset{I}{||} P_Z$$

where I is the synchronisation interface and

$$P_{CSP} = main, \text{ and}$$

$$P_Z = Z(State) = \begin{pmatrix} pre\ com\_tick\ \&\ tick \rightarrow Z(com\_tick(State)) \\ \Box\ pre\ com\_tack\ \&\ tack \rightarrow Z(com\_tack(State)) \end{pmatrix}$$

In words: the CSP part of the specification is the *main* process itself, whereas the Z part is a recursive process which offers all events (provided their preconditions hold) and then recurses with the corresponding state change. The statement "*condition* & *Process*" is another way for representing conditional choice in CSP. It is similar to "*if condition then Process else STOP*".

2.3 CSP_Z Model Checking

Model checking means verifying automatically the satisfaction relation $M \models p$, which states that the model M satisfies the property p (described as a logical formula) [10]. Roscoe [17] observed that this could be made via refinement-checking for CSP: $(M \models p) \Leftrightarrow (S_p \sqsubseteq S_M)$, where S_P and S_M are CSP specifications and S_p is built (or predefined in cases such as deadlock-freedom) as abstract as possible exhibiting the desired property p.

In CSP there are three models under which processes can be analysed: *traces*($\mathcal{T}$), *failures* ($\mathcal{F}$) and *failures-divergences* ($\mathcal{FD}$). This works is interested in the simplest one: *traces*. In this model, a process is analysed according to its performed events. Thus, the process $a \rightarrow SKIP$ is equivalent to $a \rightarrow SKIP \sqcap STOP$, because they have the same behaviour in terms of performed events. The set of traces: $\{\langle\rangle, \langle a\rangle\}$ can be performed by both. However, it is not difficult to see the second process can decide if accepts a or deadlocks. Under this viewpoint (*failures* model), they are different.

In order to reuse the support tool FDR, Mota and Sampaio defined a strategy [14] for converting CSP_Z specifications into CSP_M [9], the ASCII version of CSP accepted by FDR. The tool presented in [4,5] minimizes the efforts of such a translation.

Although FDR can check CSP_Z after translation, there is a problem with infinite specifications: it is impossible to represent them by using transition diagrams (the state explosion problem). As consequence, model checking cannot be applied. Fortunately, many techniques like *data abstraction* are becoming common because they try to overcome state explosion. The next section gives a brief explanation about it.

3 Data Abstraction

Data abstraction is a technique useful to transform an infinite system in a finite one, while still preserving almost all its properties. Its origins are from *abstract interpretation* works [1,2]. To use this technique in CSP_Z specifications, some conditions of *data independence* have to be guaranteed. This section gives a general idea about data abstraction and its usefulness to model checking.

3.1 Abstract Interpretation

Abstract interpretation is a mathematical framework to describe computations in another universe of abstract objects, so that the results of abstract execution give some information on the actual computations [1,2]. Thus, properties can be verified on the abstract representation instead of the concrete one. Furthermore, it can be viewed as a transformation ($f : C \rightarrow A$) that maps elements of a domain (concrete) to another (abstract) preserving almost all its properties.

3.2 Data Independence

The data abstraction technique can change data domains, affecting all mathematical structures that depend on it (especially the CSP part). Thus, some conditions have to be guaranteed before abstracting data types.

According to [12], a concurrent system is data independent with respect to a data type X if and only if it cannot perform any operations involving values of type X, but it can only input such values, store them, and perform equality tests between them. The following definition states data independence:

Definition 1. *P is a data independent process with respect to a type X if and only if:*

1. *Constants do not appear in P, only variables appear, and*
2. *Only polymorphic operations are used, and*
3. *Only equality tests are used, and*
4. *More complex functions and predicates must be defined in terms of 2 and 3, and*
5. *No replicated constructs (such as indexed parallel compositions) over X may appears in P, other than replicated nondeterministic choices over X.*

The work of Lazić is focused on establishing theorems in order to prove that a system *Spec* is refined by *Impl*, where both of them are data independent. If a process P is data independent with respect to X and X is infinite, then there is a minimum subset of X which is sufficient to apply mechanical verification over P. Lazić shows how to find this subset, called *threshold*, and its minimum cardinality. Indeed, if a system is data independent, the manipulated value does not matter in sense of verification. There is no need to represent the system by considering all values of type X, but only a constrained sufficient subset of it.

In addition to Lazić's work, Mota defined another type of data independence by adding more constraints:

Definition 2. *(Trivially Data Independence) A trivially data independent CSP process P is a data independent process which has no equality tests, nor polymorphic operations. Then,* $\#X \geq 1$ *is a sufficient threshold for all data type X independent in P, in order that all properties of P are preserved.*

The definition above is a more constrained version of Def. 1 such that any subset of X whose cardinality ($\#$) is greater than or equals to 1 is sufficient for verifying a process which is trivially data independent.

Definition 3. *(Partially Data Independence) A* CSP_Z *specification is partially data independent if its CSP part is trivially data independent.*

The reason for the definitions above is to guarantee that the data refinement on the Z part can be made without affecting the CSP part.

3.3 CSP_Z Data Abstraction

The idea of using data abstraction in CSP_Z is trying to find some abstract interpretation only for the Z part (P_Z) in order to produce a new (finite) specification that refines the original one, i.e. finding a new process P'_Z such that $P_Z \sqsubseteq_{\mathcal{T}} P'_Z$, where $\sqsubseteq_{\mathcal{T}}$ is the refinement relation under the *traces* model.

This relation affects a CSP_Z specification as a whole in sense of producing refinement (see [8] for details about the proof). In other words, a refined version of a CSP_Z specification is produced by refining its Z part and preserving its CSP one:

$$P_Z \sqsubseteq_{\mathcal{T}} P'_Z \Rightarrow P_{CSP_Z} \sqsubseteq_{\mathcal{T}} P'_{CSP_Z}$$

where $P_{CSP_Z} = P_{CSP} \underset{I}{\|} P_Z$ and $P'_{CSP_Z} = P_{CSP} \underset{I}{\|} P'_Z$.

In addition, we have to guarantee the specification is partially data independent and the abstraction for P_Z does not affect the interface I. The first requirement can be guaranteed by investigating the CSP part, whereas the second one does not happen because we do not have communicated values in this work.

Abstracting the CSP Part. According to the definitions given in this section, we have to guarantee that the CSP part is trivially data independent. The first one can be made by a simple syntactical analysis over the CSP part in sense of applying Def. 1. If the data independence property is observed then both the CSP part and interface are preserved (the interface is not affected by data domain changes).

Abstracting the Z Part. Abstracting the Z part consists of finding new domain(s) to the state components such that operations have their execution preserved. This implies each schema will be modified to a new version, but the semantics of the Z part will not be affected by the change when executing them on abstract domain.

First of all, each state variable v_i has a domain D_i determining the range of its values. As the state is a tuple of variables (i.e. $(v_1, ..., v_n)$), the state domain can be represented as the cartesian product of all variables domains $(D_1 \times D_2 \times ... \times D_n)$ that we simply treat by D. In the same way, abstract domains are represented by D_i^A for each state component and by $D^A(D_1^A \times D_2^A \times ... \times D_n^A)$ for the state as a whole.

The *abstraction* function is one that maps concrete types to abstract ones and can be defined for each state component:

$$h_i : D_i \to D_i^A.$$

This function can be extended do the state: $h : D \to D^A$ such that

$$h(d_1, d_2, ..., d_n) = (h_1(d_1), h_2(d_2), ..., h_n(d_n))$$

The function h maps each value from concrete state value to an abstract one, however this task can be not so simple as it seems. Not only the data type domain has to be changed. Operations over the substituted data type have to change in order to reflect the abstraction. Thus, an abstract version of each operation has to be defined. We represent the abstract state by $\mathcal{S}$ and operations by $com\_op_i^A$. The new Z part can be built in the following way:

$$P_Z^A \equiv Z(\mathcal{S}) = \begin{pmatrix} \quad pre\ com\_op_1\ \&\ op_1 \to Z(com\_op_1^A(\mathcal{S})) \\ \Box\ pre\ com\_op_2\ \&\ op_2 \to Z(com\_op_2^A(\mathcal{S})) \\ \ldots \\ \Box\ pre\ com\_op_n\ \&\ op_n \to Z(com\_op_n^A(\mathcal{S})) \end{pmatrix}$$

Now, based on Fischer's work [8] and if the CSP part is trivially data independent, then the following refinement holds:

$$P_{CSP} \underset{I}{\parallel} P_Z \sqsubseteq_{\mathcal{T}} P_{CSP} \underset{I}{\parallel} P_Z^A$$

4 Algorithm

This section presents an algorithm proposed by Mota [15] in a functional style based on pattern matching. Here, the algorithm is presented in a different language that combines imperative and first-order logic statements, because we are interested in a future implementation using Java [11]. The algorithm tries to find out an abstraction for an infinite CSP_Z process by only considering the Z part, what can have an exponential expansion. Thus, we propose an improvement to the algorithm that considers the CSP part as well. This approach – central focus of this work – reduces substantially the state explosion.

First of all, we present the structures of the algorithm and then we show the algorithm as a whole.

4.1 Data Structures

The general idea is expanding each state of the system in order to explore all possibilities. The new branches are created based on enabled schemas. Furthermore, when a new state is created by executing a specific schema, the corresponding event is performed. Figure 4 shows one step of this expansion and Fig. 5 shows the possibilities of executing two schemas. The operation $t\hat{\ }s$ represents sequence concatenation.

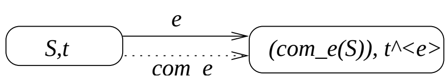

Fig. 4. Schema execution

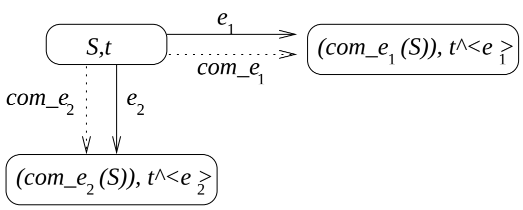

Fig. 5. Two possibilities

In order to regard the synchronisation, the transition diagram nodes present a different structure from that showed in Sect. 2.1. Now, a node is composed of a state (current) and the trace performed up to it:

$$\mathit{Node} = (\mathit{State},\ \mathit{Trace})$$

During the execution, all nodes have to be kept in some structure in order to know which nodes were expanded:

$$\mathit{ExpandedNodes} = \text{set of nodes}$$

As many nodes have to be expanded, *CurrentNode* denotes the current node to be expanded:

$$\mathit{CurrentNode} = (\mathit{CurrentState}, \mathit{CurrentTrace}) = \text{node to be expanded}$$

If there is only a single node to be expanded then its children will be generated first[1]. However, there can be more than one. Thus, the expansion should be made in stages where all children of the current stage will be generated first and processed only on the next stage. In order to consider this, *CurrentStage* and *NextStage* are defined as follow:

$$\begin{aligned} \mathit{CurrentStage} &= \text{set of nodes to be expanded at current stage} \\ \mathit{NextStage} &= \text{set of nodes to be expanded after current stage} \end{aligned}$$

4.2 Description

Once the necessary data structures have been described, we present the algorithm that deals only with the Z part of a CSP_Z specification. Each step is first explained and then presented in an algorithmic language.

According to the CSP_Z operational semantics [7,8], the *Init* schema creates the initial state and the CSP part performs a τ-transition. For us, the `Init` operation creates the initial node, containing as *CurrentState* the initial state and as *CurrentTrace* the empty trace. After that, *CurrentNode* is added to *ExpandedNodes* and *CurrentStage*:

```
Init
ExpandedNodes := { CurrentNode }
CurrentStage := { CurrentNode }
```

Once the state has been initialised, the progress depends only on the Z schemas: in each stage, the following nodes are determined by executing all enabled schemas in that stage. The new nodes are created on current stage and processed only on the next stage. After all current nodes have been processed, all created children will be also (and so on). This is regarded by the following loop:

```
while(CurrentStage has more nodes)
   Process all nodes in CurrentStage
   CurrentState := NextStage
   NextStage := {}
while-end
```

The action of processing all nodes in current stage consists of expanding all children of *CurrentNode* for all nodes in *CurrentStage*. As the algorithm tries

[1] This algorithm behaves as Breadth First Search.

to avoid infinite expansions, it has to check some condition before expanding a branch: if the branch puts the system in an infinite stable behaviour then there is no need to create that branch (because it does not add new behaviour). Otherwise, the next node (*NextState*, *NextTrace*) has to be generated. Furthermore, if the new node has just been expanded, it will not be added to *ExpandedNodes* and *NextStage*:

```
∀node ∈ CurrentStage
  CurrentNode := node
  ∀com_e enabled
    if not checkStable(CurrentTrace, e)
      NewState := com_e(CurrentState)
      NewTrace := CurrentTrace^⟨e⟩
      NewNode := (NewState, NewTrace)
      if NewNode ∉ ExpandedNodes
        ExpandedNodes := ExpandedNodes ∪ {NewNode}
        NextStage := NextStage ∪ {NewNode}
      fi
    fi
  ∀-end
∀-end
```

The function `checkStable(t,e)` checks if the event `e` had ever been performed along the trace `t`. If so, then this repetition is verified (by a theorem prover) in sense of determining its infinity. For example, looking at Fig. 6, we can see that, although new nodes are generated, the system behaviour is stable because the sequence of traces $\langle tick, tack \rangle$ repeats forever.

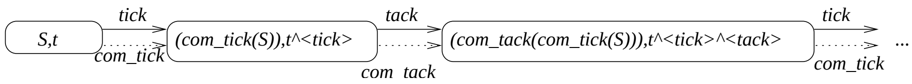

Fig. 6. Infinite stable behaviour

The repetition could be avoided if, before repeating a sequence of traces $\langle tick, tack \rangle$, a stability check was performed. It is worth noting that the *stability* property involves all schemas (enabled and disabled) and can be constructed by a simple logic expression (AND of pre conditions). Thus, for the process represented by Fig. 6, the predicate to be checked is:

$$\forall\, State, State' \mid (pre\ com\_tick \wedge pre\ com\_tack) \bullet (com\_tick \mathbin{⨾} com\_tack) \Rightarrow (pre\ com\_tick \wedge pre\ com\_tack)'$$

Informally, if the system is in a state where *com_tick* and *com_tack* are enabled (i.e. *precom_tick* $\wedge$ *precom_tack* is true), such that executing the composition of *com_tick* and *com_tack* (*com_tick* ⨾ *com_tack*) results in a state where *com_tick* and *com_tack* are enabled again, then the system has a stable behaviour.

Now we present the algorithm as a whole:

```
Init
ExpandedNodes := { CurrentNode }
CurrentStage := { CurrentNode }
while(CurrentStage has more nodes)
  ∀node ∈ CurrentStage
    CurrentNode := node
    ∀com_e enabled
      if not checkStable(CurrentTrace, e)
        NewState := com_e(CurrentState)
        NewTrace := CurrentTrace^⟨e⟩
        NewNode := (NewState, NewTrace)
        if NewNode ∉ ExpandedNodes
          ExpandedNodes := ExpandedNodes ∪ {NewNode}
          NextStage := NextStage ∪ {NewNode}
        fi
      fi
    ∀-end
  ∀-end
  CurrentState := NextStage
  NextStage := {}
while-end
```

If the process is infinite but has a stable behaviour, then the algorithm finds out the point where it can avoid expansions. However, if the system is never stable then the algorithm never stops. Some concerns related to termination are carefully discussed in [15].

A very important problem with this approach is the exponential expansion. Looking only at the Z part, the branches can grow according to the number of enabled schemas (n). In the first stage, the system has only one node (generated by `Init`). In the second stage, the number of nodes is n (n new children). In the third stage it is n^2 and so on.

5 Improvement: Considering the CSP Part

In this section we present an improvement consisting of expanding only the enabled schemas accepted by the CSP part. Thus, each node has to keep information about the current state, performed trace (as defined before) and the current LTS which contains the events that can be performed when expanding the current node:

$$Node = (State, LTS, Trace)$$

As shown in Sect. 2.1, the acceptances of the CSP part can be captured by *initials*. Now, the algorithm has to expand the nodes whose enabled schemas have their corresponding events in *initials(LTS)* (line 8):

```
1   Init
2   ExpandedNodes := {CurrentNode}
3   CurrentStage := {CurrentNode}
4   while(CurrentStage has more nodes)
5     ∀node ∈ CurrentStage
6       CurrentNode := node
7       ∀com_e enabled
8         if e ∈ initials(CurrentLTS)
9           if not checkStable(CurrentTrace, e)
10             NewState := com_e(CurrentState)
11             NewLTS := CurrentLTS / e
12             NewTrace := CurrentTrace^⟨e⟩
13             NewNode := (NewState, NewLTS, NewTrace)
14             if NewNode ∉ ExpandedNodes
15               ExpandedNodes := ExpandedNodes ∪ {NewNode}
16               NextStage := NextStage ∪ {NewNode}
17             fi
18           fi
19         fi
20       ∀-end
21     ∀-end
22     CurrentState := NextStage
23     NextStage := {}
24   while-end
```

This new version leads the execution having a CSP-driven behaviour: both CSP and Z parts progress together avoiding, therefore, unnecessary expansions.

The function `checkStable(t,e)` is slightly different from its earlier version: the stability property has to be verified over the CSP part as well. The strategy used here is refinement.

This is rather straightforward and can be made by the following steps:

1. From the current trace, extract the part where a stable behaviour seems to happen: $\langle \mathit{tick}, \mathit{tack} \rangle$.
2. Build a CSP process SP which performs the trace and has a stable behaviour: $SP = \mathit{tick} \rightarrow \mathit{tack} \rightarrow SP$.
3. Build a CSP process P by forcing the CSP part to behave exactly like SP: $P = \mathit{main} \underset{A}{||} SP$, where A is the alphabet of *main*. This form is important because we are interested only in a specific behaviour of *main*, so we have to force it to have such a specific behaviour forever.

4. Try to prove that $P = SP$: $SP \sqsubseteq_{\mathcal{T}} P \ \wedge \ P \sqsubseteq_{\mathcal{T}} SP$. According to *traces* model, $SP \sqsubseteq_{\mathcal{T}} P \Leftrightarrow traces(P) \subseteq traces(SP)$. Verifying $SP \sqsubseteq_{\mathcal{T}} P$ guarantees P does not perform traces differently from SP, however it is not sufficient to guarantee that P is infinite stable (it can stop or terminate). On the other hand, if $P \sqsubseteq_{\mathcal{T}} SP$ is also valid, then P is infinite. As an example, consider the same Z part from Sect. 2.2 and the CSP part as the following process:

   $main = tick \rightarrow tack \rightarrow tick \rightarrow tack \rightarrow SKIP$

   Along the execution, the trace $\langle tick, tack \rangle$ is performed and then the function `checkStable(<tick,tack>,tick)` checks if $\langle tick, tack \rangle$ is an infinite repetition. This task is identical to the one described in Sect. 4.2 where a theorem prover performs the Z part. Now, the idea is using FDR to check the the CSP stability property. P and SP are built in the following way:

   $SP = tick \rightarrow tack \rightarrow SP$, and
   $P = main \underset{\{tick, tack\}}{||} SP$

   Clearly, we can see that $traces(P) \subseteq traces(SP)$ and therefore $SP \sqsubseteq_{\mathcal{T}} P$. However $P \sqsubseteq_{\mathcal{T}} SP$ does not hold because P terminates. Thus, the CSP part is not infinite stable.

5.1 Comparison

Although the two versions of the algorithm are apparently similar, the efficiency can be very distinct depending on the CSP part. Figure 7 shows the executions of the two versions of the algorithm: (a) considering only the Z part, and (b) considering the CSP part as well. It also includes the sequence of events performed before entering in an infinite stable behaviour. Dashed arrows denote a stable point (there is no need to continue expanding), whereas dotted ones denote the events refused by CSP part.

If the system performs a lot of events before reaching a stable behaviour then the expansion can be a very expensive task (without considering the CSP part). However, if the synchronisation between the CSP and Z parts is also considered, a substantial reduction of the growing will be observed. Figure 7(b) shows the reduction of state explosion due to the CSP part acceptances.

6 Conclusions

The use of model checking in software verification is an important activity to guarantee the final product correctness. In some cases, the technique cannot be applied, especially on infinite systems. For such systems it is impossible to represent them by using transition diagrams (the state explosion problem). Therefore, alternative techniques are essential to analyse such systems [15,17].

The present work dealt with *data abstraction* (Sect. 3) which makes use of *abstract interpretation* [1,2], a powerful framework for converting infinite systems into finite ones, while still preserving most of its properties. We also showed the

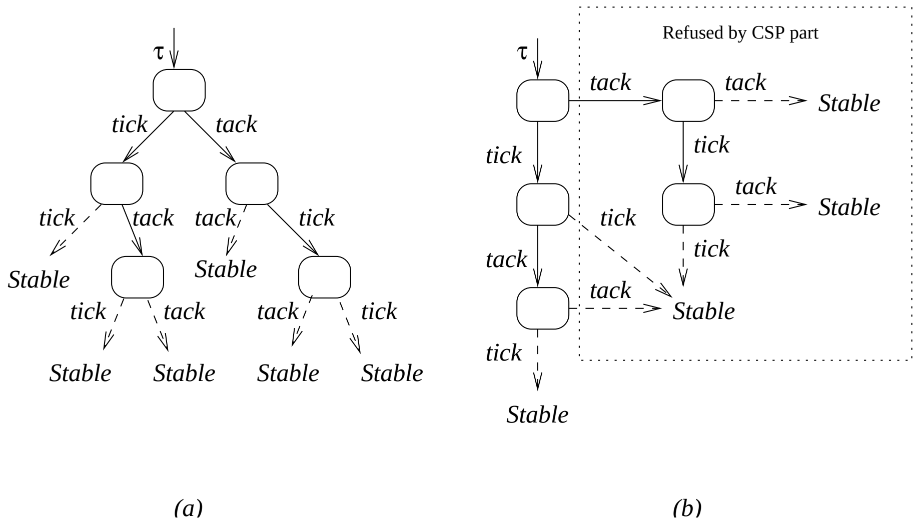

Fig. 7. (a) P_Z expansion and (b) P_{CSP_Z} expansion

CSP_Z language through an example and an algorithm [15] which applies the technique of data abstraction.

Although the algorithm overcomes the state explosion, it takes into account only the Z part and, therefore, can have an exponential expansion. In Sect. 5 we proposed an improvement for this algorithm by considering the two specification parts: Z and CSP; that is more faithful to CSP_Z operational semantics [7,8]. We also showed that this improvement causes a potential reduction of state expansion where the calculations can become linear in complexity (see Fig. 7).

Another contribution concerns support tools: integrated tools can be developed in order to implement the algorithm and perform data abstraction by reusing existing tools (FDR and Z-Eves).

As future work we intend to add data abstraction features to the translator described in [4,5] in order to build a support environment for validating CSP_Z specifications [6]. Another important approach is considering values in communications.

In terms of abstract interpretation, we also intend to add features to the algorithm: at the end, the algorithm can build an abstraction function. This makes the algorithm more complete because it does not only determine whether the process has a stable behaviour, but also gives the abstraction function.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles

of Programming Languages, pages 238 - 252, Los Angeles, California, 1977. ACM Press, New York, NY, USA
2. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. J. Logic. and Comp., **2(4)** (1992) 511 - 547
3. Dawes, J.: The VDM-SL Reference Guide. Pitman, London, UK, 1991
4. Farias, A., Mota, A., Sampaio, A.: Um Conversor da Notação CSP_Z para CSP_M. Revista Eletrônica de Iniciação Científica, Edição Agosto 2001. Available at: *http://www.sbc.org.br/reic/edicoes/edicao1*
5. Farias, A., Mota, A., Sampaio, A.: De CSP_Z para CSP_M: Uma Ferramenta Transformacional Java. Workshop em Métodos Formais (WMF2001) (2001) 1 - 10
6. Farias, A.: An Environment for Infinite CSP_Z Processes Verification. Master thesis (to appear)
7. Fischer, C.: Combining CSP and Z. Technical Report, University of Oldenburg, 1996
8. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000
9. Formal Systems (Europe): FDR User Manual version 2.28, 1997
10. Grumberg, O., Clarke, E. M., Peled, D. A.: Model Checking. The MIT Press, Cambrige, MA, 1999
11. Horstman, C., Cornell, G. Core Java 2. Sun Microsystems press, vols. I and II, 2000
12. Lazić, R.: A semantic study of data independence with applications to the mechanical verification of concurrent systems. PhD thesis. Oxford University Computing Laboratory, 1999
13. Milner, R.: Calculus of Communicating Systems. Lecture Notes in Computer Science, vol 92, Springer, Berlin, 1980
14. Mota, A., Sampaio, A.: Model-Checking CSP_Z: strategy, tool support and industrial application. Science of Computer Programming, **40** (2001) 59 - 96
15. Mota, A.: Model Checking CSP_Z: Techniques to Overcome State Explosion. PhD thesis. Federal University of Pernambuco, Brazil, 2001
16. Roscoe, A. W.: The Theory and Practice of Concurrency. Prentice Hall, 1998
17. Roscoe, A. W.: Model checking CSP. In: A.W. Roscoe (Ed), A Classical Mind: Essays in Honour of C.A.R. Hoare, Prentice Hall, Englewood Cliffs, NJ, 1994
18. Saaltink, M.: Z-Eves System. In ZUM'97: The Formal Specification Notation, volume 1212, LNCS, Springer, 1997
19. Spivey, M.: The Z Notation: A Reference Manual. 2^{nd} Edition, Prentice Hall International, Englewood Cliffs, NJ, 1992
20. Wehrheim, H.: Data Abstraction for CSP-OZ. FM'99 World Congress on Formal Methods, Lecture Notes in Computer Science, vol. 1709, Springer, Berlin, 1999