Mechanical Abstraction of CSP_Z Processes

Alexandre Mota*, Paulo Borba, and Augusto Sampaio

Universidade Federal de Pernambuco
Centro de Informática - CIn
P.O.Box 7851 - Cidade Universitária
50740-540 Recife - PE Brazil
{acm,phmb,acas}@cin.ufpe.br

Abstract. We propose a mechanised strategy to turn an infinite CSP_Z process (formed of CSP and Z constructs) into one suitable for model checking. This strategy integrates two theories which allow us to consider the infiniteness of CSP_Z as two separate problems: data independence for handling the behavioural aspect and abstract interpretation for handling the data structure aspect. A distinguishing feature of our approach to abstract interpretation is the generation of the abstract domains based on a symbolic execution of the process.

1 Introduction

Impressive efforts have been carried out to compact various classes of transition systems while still preserving most properties; currently, even a simple model checker can easily analyse millions of states. However, many systems cannot be analysed either because they are infinite state or are too large. This is normally induced by the use of (infinite or too large) data types on communications and process parameters. Indeed, various techniques have been proposed and are still being carefully studied in order to handle certain classes of such systems: local analysis [15,17], data independence [23,26], symmetry elimination and partial order reduction [10], test automation [28], abstract interpretation [22], integration of model checkers with theorem provers [29,30], etc. Unfortunately, the most powerful techniques still need a non-guided user support for a complete and adequate usage. This support concerns the elaboration of some kind of abstraction such that model checking can be applied successfully; in the current literature—to the best of our knowledge—there is no technique nor strategy to generate such abstractions from the system description itself.

The goal of this paper is to propose a strategy for analysing infinite CSP_Z processes in which user intervention is only needed to aid theorem proving. Therefore, even though the strategy is not fully automatic in general it can be mechanised via model checking and theorem proving integration, which seems to be a promising research direction in formal verification [30]. In particular, this strategy is a combination of data independence and abstract interpretation

* This work was financially supported by CNPq grant 143720/98-8.

in a slightly different manner than approaches available in the literature. More specifically, our approach is based on Lazić's work [26] to model checking data independent CSP processes and Wehrheim's work [13,14] to data abstracting CSP_{OZ} [8] (a combination of CSP and Object-Z), although we concentrate on CSP_Z [7,8] (an integration of CSP and Z). The reason to use Wehrheim's approach instead of, for example, the ones proposed in [24,9], is that her approach already uses a CSP algebraic style which is very convenient for using FDR [12]. Wehrheim's work can be seen as a CSP view of other approaches [24,9]. Lazić's work is used on the CSP part to fix a flaw in the work of Wehrheim. This is the reason why we consider the CSP part of a CSP_Z process data independent, while the Z part takes into account the data dependencies.

We show that a data dependent infinite state CSP_Z process can be transformed into a finite CSP_Z process by using generated subtypes on its channel declarations, state, input and output variables, and by rewriting some expressions (the postconditions of the schemas) in order to perform model checking. We present an algorithm for our strategy in which decidable aspects are transferred to the user by means of using theorem provers to answer the algorithm's questions. Currently, the strategy supports model checking of some classical properties, in general, and other properties on special situations. The classical properties readily available are: deadlock and livelock (See [4] for more details).

The major advantage of our approach is that, unlike related work in the literature, we calculate the data abstraction from the process description itself. Notably, the most promising works on this research area assume some kind of data abstraction determined by the user [24,9,29,16]. The success of our strategy is directly related to how much the expansion of the Z part yields infinite regular behaviours [1]. We deal with infinity in the following way: if the composition of schemas originate a *stable behaviour* from a certain state to infinite then we can obtain optimal abstraction. The abstraction is obtained by replacing infinite states for a representative state according to the *stable property*.

Although, in principle, we could employ our strategy to other specification languages, the choice for CSP_Z was based on the way a process is modelled and interpreted. Every CSP_Z process is seen as two independent and complementary parts: a behavioural one—described in CSP—and a data structure based—modelled in Z. The behavioural part is naturally data independent while data dependent aspects are confined to the data structures part. Finally, the data structures part has a very simple form which enables the mechanisation of the strategy to be relatively straightforward.

This paper is organised as follows. The following section presents an overview of CSP_Z through an example; its semantics is informally described to ease the understanding of our data abstraction approach. Section 3 introduces the notion of data independence for CSP processes. The theory of abstract interpretation is briefly described in Section 4. The main contribution of this paper is described in Section 5 where some examples are used for illustrating our approach to abstracting CSP_Z processes, before we present our algorithm for data abstraction. Finally, we present our conclusions including topics for further research.

2 Overview of CSP$_Z$

This section introduces the language CSP$_Z$ [7,8]; a process of the On-Board Computer (OBC) of a Brazilian artificial microsatellite (SACI-1) [11,3] is used for that purpose. The Watch-Dog Timer, or simply *WDT*, is responsible for waiting periodic reset signals that come from another OBC process, the Fault-Tolerant Router (*FTR*). If such a reset signal does not come, the *WDT* sends a recovery signal to the *FTR* asynchronously, to normalise the situation. This procedure is repeated three times; if, after that, the *FTR* still does not respond, then the *WDT* considers the *FTR* faulty, finishing its operation successfully.

CSP$_Z$ is based on the version of CSP presented by Roscoe [4] instead of the original version of Hoare [6]. A CSP$_Z$ specification is enclosed into a spec and end_spec scope with its name following these keywords. The *interface* is the first part of a CSP$_Z$ specification and there it is declared the external channels (keyword chan) and the local (or hidden) ones (keyword lchan). Each list of communicating channels has an associated type: a schema type, or $[v_1 : T_1; \ldots v_n : T_n \mid P]$ where $v_1, \ldots, v_n$ are lists of variables, $T_1, \ldots, T_n$ their respective types, and P is a predicate over $v_1, \ldots, v_n$. Untyped channels are simply events by CSP tradition. Types could be built-in or user-defined types; in the latter case, they might be declared outside the spec and end_spec scope, as illustrated by the following *given-set* and used to build the type of the channel *clockWDT*.

$[CLK]$

The *WDT* interface includes a communicating channel *clockWDT* (it can send or receive *CLK* data via variable *clk*), two external events *reset* and *recover*, and four local events *timeOut*, *noTimeOut*, *failFTR*, and *offWDT*.

spec *WDT*
  chan *clockWDT*: $[clk : CLK]$
  chan *reset*, *recover*
  lchan *timeOut*, *noTimeOut*, *failFTR*, *offWDT*

The concurrent behaviour of a CSP$_Z$ specification is introduced by the keyword main, where other equations can be added to obtain a more structured description: a hierarchy of processes. The equation main describes the *WDT* behaviour in terms of a parallel composition of two other processes, *Signal* and *Verify*, which synchronise in the event *offWDT*. The process *Signal* waits for consecutive *reset* signals (coming from the *FTR* process) or synchronises with *Verify* (through the event *offWDT*) when the *FTR* goes down. The process *Verify* waits for a clock period, then checks whether a *reset* signal arrived at the right period or not via the choice operator ($\Box$). If a *timeOut* occurs then the *WDT* tries to send, at most for three times, a recovery signal to the *FTR*. If the *FTR* is not ready to synchronise in this event, after the third attempt, then *Verify* assumes that the *FTR* is faulty (enabling *failFTR*) and then synchronises with *Signal* (at *offWDT*), in which case both terminate (behaving like SKIP).

From the viewpoint of the SACI-1 project, the WDT is turned off because it cannot restart (recover) the FTR anymore.

$$\mathsf{main} = Signal \underset{\{offWDT\}}{||} Verify$$
$$Signal = (reset \rightarrow Signal \;\Box\; offWDT \rightarrow \mathit{SKIP})$$
$$Verify = (clockWDT?clk \rightarrow (noTimeOut \rightarrow Verify \;\Box\; timeOut \rightarrow (recover \rightarrow Verify \;\Box\; failFTR \rightarrow offWDT \rightarrow \mathit{SKIP})))$$

The Z part complements the main equation by means of a state space and operations defining the state change upon occurrence of each CSP event. The system state (*State*) has simply a declarative part recording the number of cycles the WDT tries to recover the FTR, and the last clock received. The initialisation schema (*Init*) asserts that the number of cycles begins at zero; prime ($'$) variables characterises the resulting state. The number of cycles belongs to the constant set $LENGTH$ (used in the declarative part of the state space).

$$LENGTH == 0 \mathrel{..} 3 \qquad State \mathrel{\widehat=} [cycles : LENGTH;\ time : CLK]$$
$$Init \mathrel{\widehat=} [State' \mid cycles' = 0]$$

To fix a time out period we introduce the constant $WDTtOut$ of type CLK. To check whether the current time is a time out, we use the constant relation $WDTP$ which expresses when one element of CLK is a multiple of another.

$$WDTtOut : CLK$$
$$WDTP : CLK \leftrightarrow CLK$$

The following operations are defined as standard Z schemas (with a declaration part and a predicate which constrains the values of the declared variables) whose names originate from the channel names, prefixing the keyword com_. Informally, the meaning of a CSP_Z specification is that, when a CSP event c occurs the respective Z operation com_c is executed, possibly changing the data structures. When a given channel has no associated schema, this means that no change of state occurs. For events with an associated non-empty schema type, the Z schema must have input and/or output variables with corresponding names in order to exchange communicated values between the CSP and the Z parts. Hence, the input variable $clk?$ (in the schema com_$clockWDT$ below) receives values communicated via the *clockWDT* channel. For schemas where prime variables are omitted, we assume that no modification occurs in the corresponding component; for instance, in the schema com_*reset* below it is implicit that the time component is not modified ($time' = time$).

$$\mathsf{com\_}reset \mathrel{\widehat=} [\Delta State \mid cycles' = 0]$$
$$\mathsf{com\_}clockWDT \mathrel{\widehat=} [\Delta State;\ clk? : CLK \mid time' = clk?]$$

The precondition of the schema com_$noTimeOut$ specifies that the current time is not a multiple of the time out constant (the time out has not yet occurred) by $\neg\ WDTP(time, WDTtOut)$. Its complement is captured by com_$timeOut$.

$\mathsf{com\_}noTimeOut \mathrel{\widehat=} [\Xi State \mid \neg\ WDTP(time, WDTtOut)]$
$\mathsf{com\_}timeOut \mathrel{\widehat=} [\Xi State \mid WDTP(time, WDTtOut)]$

As already explained, the recovery procedure is attempted for 3 times, after which the WDT assumes that the FTR is faulty. This forces the occurrence of $failFTR$ and then turns off the WDT process.

$\mathsf{com\_}recover \mathrel{\widehat=} [\Delta State \mid cycles < 3 \land cycles' = cycles + 1]$
$\mathsf{com\_}failFTR \mathrel{\widehat=} [\Xi State \mid cycles = 3]$

end_spec WDT

2.1 Semantics and Refinement

A CSP_Z process is defined as a combination of a CSP and a Z part. Its semantics is given in terms of the semantic models of CSP, that is, traces, failures, and failures-divergences [7,8]. Thus, the Z part has a non-standard semantics (see the standard semantics of Z [20]) given by the standard semantics of CSP.

These semantic models yield different views of a process. The traces model ($\mathcal{T}$) is the simplest; it allows one to observe the possible behaviours of a process. The failures model ($\mathcal{F}$) is more complex: possible (traces) and non-possible (refusals) behaviours can be appreciated. The strongest model is the failures-divergences ($\mathcal{FD}$) model, which also considers divergent behaviours.

Following the CSP tradition, a specification is better than another in terms of the semantic models when it satisfies a (parameterised) refinement relation $\sqsubseteq_M$, where M is one of the three possible models. For example, let P and Q be CSP processes. We say that Q is better than P (in the semantical model M) iff

$$P \sqsubseteq_M Q$$

which means
$\mathcal{T}(Q) \subseteq \mathcal{T}(P)$, for the traces model,
$\mathcal{T}(Q) \subseteq \mathcal{T}(P) \land \mathcal{F}(Q) \subseteq \mathcal{F}(P)$, for the failures model, and
$\mathcal{F}(Q) \subseteq \mathcal{F}(P) \land \mathcal{D}(Q) \subseteq \mathcal{D}(P)$, for the failures-divergences model.

2.2 A Normal Form for CSP_Z Processes

In [2,3] we show how an arbitrary CSP_Z process can be transformed in a pure CSP process, for the purpose of model checking using FDR. In this approach, a CSP_Z specification is defined as the parallel composition of two CSP processes: the CSP part and the Z one. In the remaining sections we assume that this transformation has already been carried out. Let P be a CSP_Z process with $Interface = \{a_1, \ldots, a_n\}$. The normal form of P, as a pure CSP process, looks like $P_{NF} = main \underset{\{a_1, \ldots, a_n\}}{\parallel} Z^{State}$, where

$$Z^{State} = \begin{array}{l} \text{pre } com\_a_1 \ \& \ a_1 \rightarrow Z^{com\_a_1(State)} \\ \Box \ \text{pre } com\_a_2 \ \& \ a_2 \rightarrow Z^{com\_a_2(State)} \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \Box \\ \text{pre } com\_a_n \ \& \ a_n \rightarrow Z^{com\_a_n(State)} \end{array}$$

It is worth observing that schemas are transformed into functions[1]. This kind of normal form[2] is turned out to be very useful for our abstraction strategy as further discussed in the remainder of this paper.

3 Data Independence

Informally, a data independent system P [23,26] (with respect to a data type X) is a system where no operations involving values of X can occur; it can only input such values, store them, and compare them for equality. In that case, the behaviour of P is preserved by replacing any concrete data type (with equality) for X (X is a parameter of P). This is precisely defined by Lazić in [26] as:

Definition 1 (Data independence) *P is data independent in a type X iff:*

1. *Constants do not appear in P, only variables appear, and*
2. *If operations are used then they must be polymorphic, or*
3. *If comparisons are done then only equality tests can be used, or*
4. *If used, complex functions and predicates must originate from 2 and 3, or*
5. *If replicated operators are used then only nondeterministic choices over X may appear in P.* $\diamondsuit$

The combination of the items in Definition 1 yields different classes of data independent systems. In this section we consider the most simplest class to represent the CSP part of a CSP_Z process. This is done in order to leave the Z part free from (possible) influences originated by the CSP part.

The work of Lazić deals with the refinement relation between two data independent processes by means of the cardinality of their data independent types. The cardinality originates from the items in Definition 1 present in the processes bodies. That is, suppose $P \sqsubseteq_M Q$ has to be checked, for some model M, such that P and Q are infinite state and data independent. Further, consider X the unique data independent type influencing $P \sqsubseteq_M Q$. Then, Lazić guarantees this refinement provided $\#X \geq N$, for some natural N, and according to Definition 1.

These results form the basis to analyse the CSP part of a CSP_Z specification. Definition 2 states the kind of data independence we are focusing.

Definition 2 (Trivially Data Independent) *A trivially data independent CSP process is a data independent process which has no equality tests, no polymorphic operations, and satisfies $\#X \geq 1$ for all data independent type X.* $\diamondsuit$

[1] This is presented formally in Section 5.

[2] Indeed, this normal form is a simplified version of the original one (See Mota and Sampaio [3] for further information).

Definition 3 is used to guarantee that the CSP_Z specification we are analysing has the simplest data independent process description for its CSP part.

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

As long as the Z part of a CSP_Z specification is normally data dependent, the previous theory cannot be applied to handle it. Hence, a more powerful theory has to be introduced to deal with the Z part. Now, we briefly present the theory of abstract interpretation for treating data dependent questions.

4 Abstract Interpretation

Abstract interpretation is an attractive theory based on the notion of galois connections (or closure operators), and was originally conceived for compiler design [21,22]. Its role is to interpret a program in an abstract domain using abstract operations. Therefore, its main benefit is to obtain useful information about a concrete system by means of its abstract version.

For model checking [10], this approach is used to avoid state explosion by replacing infinite data types by finite ones; in view of this, model checking can be extended to analyse infinite state systems. The drawbacks of this approach are related to how to determine the abstract domains and operations, and the possible loss of precision coming from the choice of abstract domains.

Definition 4 (Galois connection) *Let $\langle A, \sqsubseteq_A \rangle$ and $\langle C, \sqsubseteq_C \rangle$ be lattices. Further, let $\alpha : C \to A$ (the abstraction map or left adjunction) and $\gamma : A \to C$ (the concretisation map or right adjunction) be monotonic functions such that*

- $\forall\, a : A \bullet \alpha \circ \gamma(a) \sqsubseteq_A a$
- $\forall\, c : C \bullet c \sqsubseteq_C \gamma \circ \alpha(c)$ *(whereas $c =_C \gamma \circ \alpha(c)$ for a galois insertion)*

then $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$ represents a galois connection. $\diamondsuit$

Note that, in the terminology of abstract interpretation, the order $\sqsubseteq$ is defined such that $x \sqsubseteq y$ means x is more precise than y. Hence, $\alpha \circ \gamma(a) \sqsubseteq_A a$ means $\alpha \circ \gamma(a)$ is the best approximation for a and $c \sqsubseteq_C \gamma \circ \alpha(c)$ means the application of $\gamma \circ \alpha$ adds no information to c. The lattice $\langle A, \sqsubseteq_A \rangle$ represents the lattice of properties of the system having $\langle C, \sqsubseteq_C \rangle$ as the usual semantic domain.

In the tradition of abstract interpretation, one has to establish adjunctions such that they form a galois connection (or insertion) and, for all concrete operators, propose abstract versions for them. Moreover, this proposal might be done such that the operators (concrete and abstract) be compatible in some sense; this compatibility originates the notions of soundness (safety) and completeness (optimality) [24,25]. For example, let $f : C \to D$ be a concrete operation defined over the concrete domains C and D. Let an abstract interpretation be specified by the following galois connections $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$ and $\langle D, \sqsubseteq_D \rangle \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \langle B, \sqsubseteq_B \rangle$. In addition, let $f^\star : A \to B$ be the corresponding abstract semantic operation for f. Then, $f^\star$ is sound for f if $\alpha' \circ f \sqsubseteq f^\star \circ \alpha$. Completeness is meant as the natural strengthening of the notion of soundness, requiring its reverse relation to hold. Hence, $f^\star$ is complete for f iff $\alpha' \circ f = f^\star \circ \alpha$.

Now, we present how an abstract interpretation can be defined in terms of CSP_Z elements as well as its integration with the notion of data independence.

5 CSP_Z Data Abstraction

In this section we present what means performing an *ad hoc* CSP_Z data abstraction, in terms of the theory of abstract interpretation.

Let P be a CSP_Z specification and *Interface* be its set of channel names. Abstract a CSP_Z specification P means to find an abstract interpretation for the data domains of channels and state of P, that is, define new domains and new operations for P. Thus, let D be the data domains of state variables and M_c the data domain of channel c ($c \in \mathit{Interface}$) to be abstracted. By convention, messages are split into input (M_c^{in}) and output (M_c^{out}) messages. Recall from Section 2.2 that *com_* operations are transformed into functions with signature.

$\langle\!| \mathit{com\_c} |\!\rangle : D \times M_c^{in} \to \mathbb{P}(D \times M_c^{out})$

We build abstract *com_* functions in terms of abstract data domains and abstract versions of primitive operations. Thus, let D^A and M_c^A be abstract data domains of variables and channels, and h and r_c be abstraction maps. Recall from Definition 4 that h and r_c are our left adjunctions while the concretisations are simply identity maps, that is, we are employing the concept of galois insertion.

$h : D \to D^A$

$r_c : M_c \to M_c^A$

The communication abstractions (r_c) are only defined over communicating channels; events are not abstracted.

An abstract interpretation $\{\!| \cdot |\!\}$ is defined over abstract domains. Thus, the signature of the abstract versions become.

$\{\!| \mathit{com\_c} |\!\} : D^A \times M_c^{in,A} \to \mathbb{P}(D^A \times M_c^{out,A})$

It is worth noting that $\{\!| \mathit{com\_c} |\!\}$ is compositional in the sense that, for example, let s, s_1, s_2 be state variables of type sequence then a predicate $s' = s_1 \frown s_2$ (in a *com_* function) is abstracted to $s'^A = s_1^A \tilde{\frown} s_2^A$.

To deal with abstract powerset of data domains we present the most natural extension of the previous abstract functions. Therefore, the functions h and R_c are extended naturally to the powerset of D as follows

$H : \mathbb{P}\, D \to \mathbb{P}\, D^A = \lambda\, \mathbb{D} : \mathbb{P}\, D \bullet \{d^A : D^A \mid d \in \mathbb{D} \wedge d^A = h(d)\}$

$R_c : \mathbb{P}\, M_c \to \mathbb{P}\, M_c^A = \lambda\, \mathbb{M} : \mathbb{P}\, M_c \bullet \{m^A : M_c^A \mid m \in \mathbb{M} \wedge m^A = r_c(m)\}$

Recall from Section 4 that abstract domains and operations might be found such that the new interpretation be optimal abstraction of the original system. In the following we present what that means for CSP_Z.

Definition 5 (Optimal abstraction) *An abstract interpretation* $\{\!| \cdot |\!\}$ *is optimal according to abstractions* h *and* r_c *iff*

$$\forall\, d : D;\ m : M \bullet \{\!|\ com\_c\ |\!\}(h(d), r_c(m)) = (H \times R_c(\langle\!|\ com\_c(d, m)\ |\!\rangle)) \qquad \diamondsuit$$

By convention, the process P^A denotes the abstract version of the process P via abstract interpretation $\{\!|\cdot|\!\}$. The abstract version is built by replacing the channel types for abstract versions (images of r_c) and all *com_* functions (that is, replacing inner operations, such as $+$, $\frown$, $\leq$, etc.) for their abstract versions.

Definition 5 can be seen as a combination between Z data refinement and interface abstraction. Due to the interface abstraction, a renaming must be used to link this result with the theory of CSP process refinement. Therefore, a renaming R based on the abstract communication functions is defined.

Definition 6 (Interface Abstraction) *Let $r_c : M_c \to M_c^A$ be communication abstractions for all channels ($c \in Interface$). Then, the interface abstraction is given by $R = \bigcup_{c \in Interface}\{(m, m^A) : r_c \bullet (c.m, c.m^A)\}$* $\diamondsuit$

The following lemma relates the original and abstract versions of a CSP_Z process. It is a corrected extension[3] of a theorem proposed by Wehrheim [13,14].

Lemma 1 *Let P be a partially data independent CSP_Z specification and P^A its abstract version defined by optimal abstract interpretation $\{\!|\cdot|\!\}$ with interface abstraction given by the renaming R. Then $P[\![R]\!] =_{\mathcal{FD}} P^A$.* $\diamondsuit$

It is worth noting that, in general, Lemma 1 concern only renamed versions of the CSP_Z original processes. Thus, only those properties preserved via renaming might be checked. Wehrheim [13,14] still tries to avoid this limitation via algebraic manipulation but the problem of infinity occurs again. This is exactly why we are primarily concerned with deadlock and livelock analysis.

Example 1 (An Ad Hoc Data Abstraction) *Consider the CSP_Z process*

spec P

$\quad$ *chan* $a, b : \mathbb{N}$

$\quad$ *main* $= a?x \to b?y \to$ *main*

$\quad State \mathrel{\widehat=} [c : \mathbb{N}]$

$\quad com\_a \mathrel{\widehat=} [\Delta State;\ x? : \mathbb{N} \mid c' = x?]$

$\quad com\_b \mathrel{\widehat=} [\Delta State;\ y? : \mathbb{N} \mid c * y? > 0 \land c' = y?]$

end_spec

Now, let $N^A = \{pos, nonPos\}$ be an abstract domain with abstraction maps

$$r_a = r_b = h = \{n : \mathbb{N} \mid n > 0 \bullet n \mapsto pos\} \cup \{n : \mathbb{N} \mid n \leq 0 \bullet n \mapsto nonPos\}$$

The renaming and abstract operator versions are defined as

$$R = \{(e^C, e^A) : r_a \bullet (a.e^C, a.e^A)\} \cup \{(e^C, e^A) : r_a \bullet (b.e^C, b.e^A)\}$$

$$s_1 \mathbin{\tilde{*}} s_2 = \begin{cases} pos, & s_1 = s_2 = pos \\ nonPos, & otherwise \end{cases} \qquad s_1 \mathbin{\tilde{>}} s_2 = \begin{cases} true, & s_1 = pos \land s_2 = nonPos \\ false, & otherwise \end{cases}$$

Then, applying r_a, r_b to the channels, h to state variable c and constants, and using the abstract operators, we get

[3] Please refer to Mota [1] for the proof of Lemma 1

$\mathsf{spec}\ P^A$

$\mathsf{chan}\ a, b : N^A$

$\mathsf{main} = a?x \rightarrow b?y \rightarrow \mathsf{main}$

$State^A \mathrel{\widehat=} [c : N^A]$

$com\_a^A \mathrel{\widehat=} [\Delta State^A;\ x? : N^A \mid c' = x?]$

$com\_b^A \mathrel{\widehat=} [\Delta State^A;\ y? : N^A \mid c\tilde{*}y?\tilde{>}nonPos \wedge c' = y?]$

$\mathsf{end\_spec}$ ◇

From Lemma 1 we have $P^A =_{\mathcal{FD}} P[\![R]\!]$. Note that the operator $\tilde{>}$ is optimal due to the predicate $x > 0$ be more restrict than $x > y$. By using the latter, N^A would be refined to $\{pos, zero, neg\}$ to achieve optimality (See [25] for details).

5.1 Guidelines for CSP$_Z$ Data Abstraction

This section introduces the guidelines for CSP$_Z$ data abstraction. Recall from Section 2.2 that the normal form of a CSP$_Z$ specification has a very simple structure for the Z part. This structure is exactly what eases the search for a data abstraction as described in the following examples.

Initially we present an example taken from Wehrheim's work [13,14], where the data abstraction was proposed by the user. We demonstrate that following our informal strategy we are able to calculate such a data abstraction.

Example 2 ***(Infinite Clock)*** *Let P_{Clock} be an infinite CSP$_Z$ process given by*

$\mathsf{spec}\ P_{Clock}$

$\mathsf{chan}\ tick, tack$

$\mathsf{main} = \Box e : \{tick, tack\} \bullet e \rightarrow \mathsf{main}$

$State \mathrel{\widehat=} [n : \mathbb{N}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_tick \mathrel{\widehat=} [\Delta State \mid n \bmod 2 = 0 \wedge n' = n + 1]$

$com\_tack \mathrel{\widehat=} [\Delta State \mid n \bmod 2 = 1 \wedge n' = n + 1]$

$\mathsf{end\_spec}$

Set the abstraction data domain to be equal to the concrete one. Set abstraction function h to be the identity map. Recall from Section 5 that the abstractions r_{tick} and r_{tack} are not defined since there is no communication. Therefore we already know that we do not need a renaming (interface abstraction). Our first step is very simple: expand (symbolically) the Z part[4] until the set of enabled preconditions in the current state has already occurred in an earlier state. This step yields the LTS of Figure 1. Note that the precondition $\mathrm{pre}\, com\_tick$, $n \mod 2 = 0$ (n is even) is valid in $n = 0$ and $n = 2$. At this point we perform our second step: try to prove that this repetition is permanent. Let conj be a conjunction of preconditions and comp be a sequential composition as follows

$conj \mathrel{\widehat=} \mathrm{pre}\, com\_tick \wedge \neg\ \mathrm{pre}\, com\_tack$

$comp \mathrel{\widehat=} com\_tick \fatsemi com\_tack$

[4] This is relatively simple due to the normal form presented in Section 2.2.

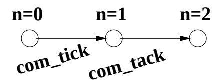

Fig. 1. LTS of the Z part of P_{Clock}

Then the general predicate to be proven is

$$\forall\, State;\ State' \mid conj \bullet comp \Rightarrow conj'$$

This predicate (we call it by stability predicate*) can be proven by theorem provers like Z-Eves [19] or ACL2 [18], for example.*

Our third step checks the proof status of the stability predicate. If it is valid then the abstraction function h is modified. Further, this validity assures an equivalence relation—under the conjunction of preconditions (the property)—between the states before and after the sequential composition of com_ *operations, including the operations inside the schema composition. That is, as long as* com_tick $⨾$ com_tack *is stable then the next possible schema operation must be* com_tick*, and after this the next must be* com_tack*, and so on. Thus, from the above predicate we build the equivalence relation*

$$E_{tick} = \{n : \mathbb{N} \mid n \bmod 2 = 0 \bullet n \mapsto n + 2\}^*$$
$$E_{tack} = \{n : \mathbb{N} \mid n \bmod 2 = 1 \bullet n \mapsto n + 2\}^*$$

and, for each partition, we take one element to build the abstraction function. It is worth noting that $n \bmod 2 = 0$ *is the essence (simplification) of the property* pre com_tick $\wedge$ $\neg$ pre com_tack *as well as* $n \bmod 2 = 1$ *is the essence of* $\neg$ pre com_tick $\wedge$ pre com_tack*.*

$$h(n) = \begin{cases} 0, & 0\ E_{tick}\ n \\ 1, & 1\ E_{tack}\ n \end{cases}$$

That is, the abstraction function is induced by the equivalence relation built. After that, we discard this execution path and try to explore another one, repeating the previous steps. Since our example does not have any other paths to explore, we start the final step which builds the abstract domains and abstract operators. For us, the abstract domain is $A = \{0, 1\}$ *(the image of h), and the abstract version of the successor operator is the application of the abstraction* ($\alpha = h$) *and concretisation* ($\gamma = i_A$) *functions as follows*

$$\{\!|\ \lambda\, x : \mathbb{N} \bullet x + 1\ |\!\} = \alpha \circ (\lambda\, x : \mathbb{N} \bullet x + 1) \circ \gamma = \lambda\, x^A : A \bullet h(x^A + 1)$$

That is, the abstraction is built by replacing the concrete domains, applying the abstraction function h to the constants, and the concrete operators are abstracted by an application of the abstraction function to the result. It is worth noting that our strategy is done in such a way that we do not have to abstract the preconditions. The reason for this is that our abstract domains are always the subsets of the original types determined from the lattice of the preconditions (repetition of the set of preconditions enabled).

Note that this abstraction is optimal by construction. The absence of communication abstractions (renaming) yields an equivalence under Z data refinement and process refinement, that is, $P_{Clock} \equiv_{\mathcal{FD}} P^A_{Clock}$ *(see Lemma 1).* $\diamondsuit$

It is worth noting that the stability predicate originates from the lattice of the preconditions of the Z part: all preconditions disabled lead to deadlock whereas all preconditions enabled lead to full nondeterminism. This lattice is known as the lattice of properties in the terminology of abstract interpretation [22]. If, during the symbolic execution of the Z part, we achieve a point (trace) such that after it the set of preconditions (a given property in the lattice of preconditions) is always the same, the domain used until that point can be seen as a representative for the future values because they all have the same property.

Example 3 ***(A Precise Loop)*** *Let P be a CSP_Z process given as*
spec P
 chan a, b
 main $= a \rightarrow$ *main* $\square\, b \rightarrow$ *main*

$State \mathrel{\widehat=} [c : \mathbb{N}]$ $Init \mathrel{\widehat=} [State' \mid c' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid c \leq 5 \wedge c' = c + 1]$ $com\_b \mathrel{\widehat=} [\Delta State \mid c \geq 5 \wedge c' = c + 1]$

end_spec

Start by setting the abstract domain as $\mathbb{N}$ and h to be the identity. After that, we explore the LTS (of the Z part) in a lazy fashion, observing whether the set of valid preconditions repeats. To ease the explanation, observe Figure 2. This figure shows that we need 6 expansions, and respectively 5 stability predicates with status false, in order to get a stable path. Let conj be the property being repeated and comp the sequential composition where this is happening

$$conj \mathrel{\widehat=} \mathrm{pre}\ com\_b \wedge \neg\ \mathrm{pre}\ com\_a$$
$$comp \mathrel{\widehat=} com\_b$$

and the general predicate to be proven is

$$\forall\, State;\ State' \mid conj \bullet comp \Rightarrow conj'$$

From this predicate we achieve the following unique equivalence relation, since the sequential composition is built by only one schema operation.

$$E = \{c : \mathbb{N} \mid c > 5 \bullet c \mapsto c + 1\}^*$$

where $c > 5$ is the reduced form for $\neg\ \mathrm{pre}\ com\_a \wedge \mathrm{pre}\ com\_b$. The abstraction function is built in terms of the least elements of each partition. Then

$$h(c) = \begin{cases} 6, & 6\ E\ c \\ c, & otherwise \end{cases}$$

which determines the abstraction $A = 0 \mathrel{..} 6$ and $\{\!|\ n + 1\ |\!\} = \lambda\, n^A : A \bullet h(n^A + 1)$. $\diamondsuit$

5.2 Algorithm

In this section we present the algorithm for CSP_Z data abstraction. It is described in a functional style using pattern matching. The main part of the algorithm concentrates on the function findAbstraction. The other functions are

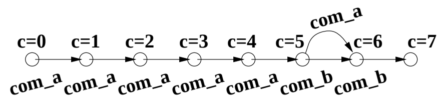

Fig. 2. LTS of the Z part of P

defined modularly as well and called by findAbstraction (See [1] for further details).

From Examples 2 and 3, we can note that the current state, trace, and property must be known. Recall from Section 2 that we can represent all this information using channel names. That is, via channel names we can build a (symbolic) trace (a sequence of channel names), the current state (a sequential schema composition where the schemas are built by prefixing the keyword com_ in front of the channel names) and a property (as a set of channels). For example, suppose a CSP$_Z$ process with interface $\{a, b, c\}$ (without values for ease). Let $\langle a, b, b, c\rangle$ be a trace of this process. The corresponding state is given by

$$Init \fatsemi \mathsf{com\_}a \fatsemi \mathsf{com\_}b \fatsemi \mathsf{com\_}b \fatsemi \mathsf{com\_}c$$

and, finally, a property could be characterised by the set $\{a, b\}$, which means

$$\mathrm{pre}\,\mathsf{com\_}a \land \mathrm{pre}\,\mathsf{com\_}b \land \neg\ \mathrm{pre}\,\mathsf{com\_}c$$

that is, if a channel does not belong to the property set, then we take the negation of the precondition of its corresponding com_ schema.

We introduce some short names, frequently used by the functions.

$PCh == \mathbb{P}\ ChanName$ $\qquad$ $Label == ChanName$
$AcceptanceSet == PCh$ $\qquad$ $Property == PCh$

As traces are sequences of events and the next alternatives as well as the current property can vary with the trace, we define the following structure.

$$Path == \mathrm{seq}(AcceptanceSet \times Label \times Property)$$

Our first function is findAbstraction; the kernel of our guided data abstraction.

The base case corresponds to the empty structure; that is, no further progress is possible, return the identity map, according to the data domains of the Z part (assume D as the type related to the schema $State$). This identity will be overridden recursively by the abstractions found.

When the $Path$ structure is not empty, the Z part is expanded, according to the elements in $accS_{curr}$. The first branch corresponds to $accS_{curr} = \varnothing$. If this is the case, then the current tuple of the $Path$ structure is discarded and a previous one takes place, recursively (findAbstraction t).

If the current acceptance set is not empty, the function findAbstraction tries a next transition, based on the event chosen to be engaged ($lt_{next} \in accS_{curr}$). A next transition can assume two differing forms:

1. t_{next}: an abstraction can be performed or the next acceptance set is empty;
2. $t_{further}$: no abstraction can be performed.

If $accS_{next} \neq \varnothing$ is verified, then we check if the property repeats[5]. If it repeats, then it calls checkStability. If an abstraction is possible, we calculate a new *Path* structure—t_{new}—by calling newExploration. Otherwise, a further expansion occurs (findAbstraction $t_{further}$).

```
findAbstraction :: Path → (D → D^A)
findAbstraction ⟨⟩ = i_D
findAbstraction ⟨(accS_curr, lt_curr, prop_curr)⟩ ⁀ t=
 if accS_curr = ∅ then findAbstraction t
 else
  let
   lt_next ∈ accS_curr
   t_next = ⟨(accS_curr \ {lt_next}, lt_next, prop_curr)⟩ ⁀ t
   accS_next =validOpers t_next Interface
   t_further = ⟨(accS_next, τ, accS_next)⟩ ⁀ t_next
  •
   if accS_next ≠ ∅ then
    if ∃ s : ran t_next • π3(s) = accS_next then /* Property repeats */
     let
      user =checkStability t_next accS_next
      t_new =newExploration t_next accS_next
     •
      case user of
        optimal : findAbstraction t_new ⊕ optimalAbs t_next accS_next
        none    : findAbstraction t_further
    else findAbstraction t_further
   else findAbstraction t_next
```

Recall from Section 2 that the Z part constrains the CSP part through the preconditions. That is, for a channel c, if the precondition of com_c is valid, then c is ready to engage with the CSP part. Otherwise, c is refused in the Z part and consequently in the CSP part too, because they cannot synchronise. The function validOpers has this purpose. It takes a path and an acceptance set as input and returns the set of channels (subset of the interface) which has the precondition valid for the current state (built using buildComp).

```
validOpers :: Path → AcceptanceSet → AcceptanceSet
validOpers t ∅ = ∅
validOpers t accS_curr =
 let
  e ∈ accS_curr
 •
  (if [[(buildComp t ∅) ⇒ (pre com_e)′]]^P = [[false]]^P then ∅
  else {e}) ∪ (validOpers t accS_curr \ {e})
```

[5] Note that the predicate $\exists\, s;\ \mathrm{ran}\, t_{next} \bullet \pi_3(s) = accS_{next}$ uses the function π_3. The function π is simply a projection function, that is, $\pi_3(a, b, c) = c$.

The term $[\![p]\!]^{\mathcal{P}}$ means the semantic interpretation of the predicate p. Hence, generally, the clause $[\![(\mathsf{buildComp}\ t\ \varnothing) \Rightarrow (\mathrm{pre}\ \mathsf{com\_}e)']\!]^{\mathcal{P}}$ needs some theorem proving support. But when all variables have an associated value, it is possible to get the same result by direct application of the current state to the preconditions. Recall from Section 2 that, for a given trace, we have a corresponding Z schema composition. For example, suppose that the trace $\langle a, b, c\rangle$ has occurred, then the state of the system is given by $Init \fatsemi \mathsf{com\_}a \fatsemi \mathsf{com\_}b \fatsemi \mathsf{com\_}c$. The function buildComp, presented in what follows, has this purpose.

```
buildComp :: Path → Property → SchemaExpr
buildComp ⟨⟩ prop  =  Init
buildComp ⟨(accS_curr, e, prop_curr)⟩ ⁀ t  prop  =
  if prop = prop_curr then com_e else (buildComp t prop) ⨾ com_e
```

Recall from Section 5.1 that we define a property to be a conjunction of preconditions. The functions validGuards and invalidGuards, together, build properties.

The function newExploration searches for an unexplored trace. It takes a path structure and a property as input. Associated to the *Path* structure only, we have two possibilities: Either it is empty and we return an empty sequence, or it is not empty and the resulting *Path* structure depends on the given property. The first two branches deal with the current property being equal to the given property. That is, we have found the element of the *Path* structure which is keeping the information concerning the previous repeated property. Here, two cases are checked: either the current tuple must be discarded ($alts_{curr} = \langle\rangle$), or this tuple still has a possible alternative to be considered ($alts_{curr} \neq \langle\rangle$). The last point simply discards the current tuple and considers the rest recursively.

```
newExploration :: Path → Property → Path
newExploration ⟨⟩ prop  =  ⟨⟩
newExploration ⟨(accS_curr, e, prop_curr)⟩ ⁀ t  prop  =
  if prop = prop_curr ∧ accS_curr = ∅ then t
  else
    if prop = prop_curr ∧ accS_curr ≠ ∅ then ⟨(accS_curr, e, prop_curr)⟩ ⁀ t
    else newExploration t prop
```

The function checkStability deserves special attention. Its purpose is to transfer the undecidability problem, related to the check for stability, to the user, via application of theorem proving. In this sense, we are integrating model checking with theorem proving; a research direction stated by Pnueli [30]. This function returns a user decision. Obviously, a user for us means some external interaction: a human being, a theorem prover, etc. That is, we can have a predicate which can be proven fully automatic by a theorem prover without a human being intervention. Hence, our strategy can be fully automatic as long as the predicates considered belong to a class of a decidable logic [23,27,18,28,5].

Therefore, before presenting the function checkStability, we introduce the user response using a free-type definition. It can be *optimal*—the abstraction is a total surjective function—or *none*—we must further expand this path.

$$USER ::= optimal \quad \text{– The abstraction is optimal}$$
$$\quad\quad\;\; | \; none \quad \text{– We cannot abstract this trace}$$

The function checkStability checks the validity of the predicate $\forall State; \; State' \mid conj \bullet comp \Rightarrow conj'$, where *conj* captures the stable property (conjunction of valid and invalid preconditions) and *comp* is a sequential schema composition.

```
checkStability :: Path → Property → USER
checkStability t prop =
  let
    conj = validGuards prop ∧ invalidGuards (Interface \ prop)
    comp = buildComp t prop
    stable = ∀ State; State' | conj • comp ⇒ conj'
  • if [[stable]]^P = [[true]]^P then optimal else none
```

The function validGuards yields the conjunction of the valid preconditions.

```
validGuards :: AcceptanceSet ⇸ ZPred
validGuards ∅ = true
validGuards accS_curr =
  let e ∈ accS_curr • pre com_e ∧ (validGuards accS_curr \ {e})
```

Complementarily, the function invalidGuards generates the conjunction of the invalid preconditions; those with a negation ($\neg$) in front of each precondition.

```
invalidGuards :: AcceptanceSet ⇸ ZPred
invalidGuards ∅ = true
invalidGuards accS_curr =
  let e ∈ accS_curr • ¬ pre com_e ∧ (invalidGuards accS_curr \ {e})
```

If the function checkStability results *optimal*, then we have to produce the expected data abstraction; that is, a (total and surjective) map between a small (finite) set and an infinite one. For instance, consider Example 2. In this example, the trace $\langle tick^0, tack^1, tick^2, tack^3, \ldots \rangle$ is abstracted by $\langle tick^0, tack^1 \rangle^k$ using

$$h(n) = \begin{cases} 0, 0 \; E_{tick} \; n \\ 1, 1 \; E_{tack} \; n \end{cases}$$

where

$E_{tick} = \{n : \mathbb{N} \mid n \bmod 2 = 0 \bullet n \mapsto n + 2\}^*$

$E_{tack} = \{n : \mathbb{N} \mid n \bmod 2 = 1 \bullet n \mapsto n + 2\}^*$

Prior to present the function which generates abstraction, we consider some auxiliary functions. First, buildTrace, identical to buildComp, except the response.

```
buildTrace :: Path → Property → seq ChanName
buildTrace ⟨⟩ prop = ⟨⟩
buildTrace ⟨(accS_curr, e, prop_curr)⟩ ⁀ t prop =
  if prop = prop_curr then ⟨e⟩ else (buildTrace t prop) ⁀ ⟨e⟩
```

The function cShiftT makes a cyclic shift in a trace; it always shifts the elements to the left. For instance, the call cShiftT $\langle a, b, b, c\rangle$ returns $\langle b, c, a, b\rangle$.

$$\begin{array}{|l|}\hline \mathsf{cShiftT}\ ::\ \mathrm{seq}\, ChanName \to \mathbb{N} \to \mathrm{seq}\, ChanName \\ \mathsf{cShiftT}\ \ s\ \ 0\ \ =\ \ s \\ \mathsf{cShiftT}\ \ \langle e\rangle \frown s\ \ n\ \ = (\mathsf{cShiftT}\ \ s\ \ (n-1)) \frown \langle e\rangle \\ \hline \end{array}$$

Finally, we have buildSeqC; it returns a sequential schema composition from a trace. Or, buildSeqC$\langle a, b, b, c\rangle$ returns $\mathsf{com\_}a \fatsemi \mathsf{com\_}b \fatsemi \mathsf{com\_}b \fatsemi \mathsf{com\_}c$.

$$\begin{array}{|l|}\hline \mathsf{buildSeqC}\ ::\ \mathrm{seq}\, ChanName \nrightarrow SchemaExpr \\ \mathsf{buildSeqC}\ \ \langle e\rangle\ \ =\ \ \mathsf{com\_}e \\ \mathsf{buildSeqC}\ \ \langle e\rangle \frown s\ \ = \mathsf{com\_}e \fatsemi (\mathsf{buildSeqC}\ \ s) \\ \hline \end{array}$$

The function optimalAbs generates the abstraction; a mapping between one fixed value, according to the equivalence relations of the periodic property, and their infinite equivalents. Note that the least value refers to the first element of the stable trace. The rest is obtained by sequential composition, since the future sequential compositions repeat indefinitely. And, differently from the Examples 2 and 3, this function builds the abstraction using the Z notation.

$$\begin{array}{|l|}\hline \mathsf{optimalAbs}\ ::\ Path \to Property \to (D \to D^A) \\ \mathsf{optimalAbs}\ \ t\ \ prop\ \ = \\ \quad \mathtt{let} \\ \qquad stable = \mathsf{buildTrace}\ \ t\ \ prop \\ \qquad 1 \le j \le \#stable \\ \qquad EqRel(j) = \{[\![\mathsf{buildSeqC}(\mathsf{cShiftT}\ \ stable\ \ (j-1))]\!]^e\} \\ \qquad abs_j \in EqRel(j) \\ \quad \bullet \\ \qquad \bigcup_{i=1}^{\#stable} \{s : EqRel(i) \bullet s \mapsto abs_i\} \\ \hline \end{array}$$

It is worth noting that, unlike the Examples 2 and 3, optimalAbs builds the equivalence relations ($EqRel$) implicitly. The difference is that while in the examples we deal with values directly, in this definition we are working with bindings (association between names and values) provided by the Z language [20].

In what follows, we present the proof of correctness for our algorithm. But first, a previous result is given (See [1] for a detailed proof).

Lemma 2 ***(Overhidden Preserved Abstraction)*** *Let P be a CSP_Z process, t and t' be Path structures, and prop be a property of P. If the overhidden*

$$\mathit{findAbstraction}\ t \oplus \mathit{optimalAbs}\ t'\ \ prop$$

could be applied and terminates successfully then it yields optimal abstraction. ♢

Now, the main result of this section can take place.

Theorem 1 ***(Optimal Abstraction)*** *Let P be a CSP_Z process. If the function findAbstraction, applied to the Z part of P, terminates then it yields optimal data abstraction.*

Proof. *The proof follows by induction on the size of the Path structure.*

- *Base case* ($\langle\rangle$): *trivial.*
- *Induction case* ($\langle(accS_{curr}, lt_{curr}, prop_{curr})\rangle \frown t$): *by case analysis where* $accS_{curr}$, lt_{next}, t_{next}, $t_{further}$, $accS_{next}$, *and* t_{new} *are given as in the algorithm*
  1. $accS_{curr} = \varnothing$: *via induction hypothesis on* $\mathsf{findAbstraction}\ t$.
  2. $accS_{curr} \neq \varnothing \wedge accS_{next} \neq \varnothing$: *it depends on the analysis of case 4.*
  3. $accS_{curr} \neq \varnothing \wedge accS_{next} = \varnothing$: *in this case, the call* $\mathsf{findAbstraction}\ t_{next}$ *occurs. As* $t_{next} \neq \langle\rangle$, *the induction case is considered again. The unique open situation is when the future calls belong to the present situation. Therefore, after m calls we get* $accS_{curr} = \varnothing$ *which yields optimal data abstraction by 1.*
  4. $accS_{curr} \neq \varnothing \wedge accS_{next} \neq \varnothing \wedge \exists\, s : ran\ t_{next} \bullet \pi_3(s) = accS_{next}$: *we have*

     $$\mathsf{findAbstraction}\ t_{new} \oplus \mathsf{optimalAbs}\ t_{next}\ accS_{next}$$

     which, by hypothesis and Lemma 2, yields optimal data abstraction.
  5. $accS_{curr} \neq \varnothing \wedge accS_{next} \neq \varnothing \wedge \neg\ \exists\, s : ran\ t_{next} \bullet \pi_3(s) = accS_{next}$: *as long as the call* $\mathsf{findAbstraction}\ t_{further}$ *terminates, by hypothesis, then optimal data abstraction is guaranteed by the previous situations.* ♢

Currently, we have a Haskell prototype[6] for the algorithm. It was integrated to the theorem prover Z-Eves [19]. The function checkStability generates the predicates—to be proven by Z-Eves—and the user controls every step, guiding the approach. Indeed, Examples 2 and 3 were built using the prototype. The Z part is introduced via a functional characterisation of the *com_* operations [2,3]. Further, when the postcondition of some *com_* operation is nondeterministic or is based on communication two approaches can used to compute the next state: one is based on testing [28] and another on theorem proving [16]. We have employed the testing approach on the WDT because it is less expensive and its nondeterminism is simple. The WDT was submitted to the prototype and we have confirmed our hypothesis stated in [2,3] which assumes that the WDT only needs two clock elements in its CLK given type: one for enabling the schema *com_noTimeOut* and another for *com_timeOut*. The WDT abstraction is optimal with interface abstraction (Please refer to [1] for further details).

Acknowledgements. We would like to thank Ana Cavalcanti, David Deharb, David Naumann, Jim Woodcock, and He Hifeng for comments on earlier drafts of this paper. We also thank Ranko Lazić and Heike Wehrheim for sending us drafts of their papers.

6 Conclusion

Our original goal was about model checking CSP_Z [2,3]. This effort has presented another difficulty: how to model check infinite state systems since they emerge naturally in CSP_Z specifications. The works of Lazić [26] and Wehrheim [13,14] has been adopted as a basis for this work due to their complementary contributions to our aim. However, both had some kind of limitation: Lazić's work allows only to check data independent refinements, whereas Wehrheim's work leaves the task of proposing abstract domains and operations (the most difficulty part of a data abstraction) to the user. In this sense, we believe that the results reported here contribute in the following way to the works: to our earlier work [2,3] by

[6] It is located at http://www.cin.ufpe.br/~acm/stable.hs

enabling model checking of infinite CSP_Z processes; to Lazić's work by capturing data dependencies in the Z part of a CSP_Z specification; and to Wehrheim's work by mechanising her non-guided data abstraction technique.

Another result was to find a flaw in some results of Wehrheim's work, based on Lazić's work (See [1] for further details). It is related to the CSP part of a CSP_Z process; Wehrheim's work does not discriminate what CSP elements the CSP part can use. Thus, if equality tests are allowed the CSP part can have stronger dependencies than those of the Z part. This was fixed on Lemma 1 by considering the CSP part to be trivially data independent.

In the direction of mechanisation, our approach is similar to the works of Stahl [16] and Shankar [29]. The main difference is that while they use boolean abstraction (replace predicates and expressions for boolean variables), we use subtype abstraction (replace types for subtypes and abstract operations for operations closed under the subtypes); this choice is crucial to make our work free from user intervention and can yield optimal abstraction, but it offers some limitations if the state variables are strongly coupled; on the other hand, Stahl and Shankar work with weakly coupled variables due to the boolean abstraction strategy, however they need an initial user support and focus on safe abstractions. The normal form of a CSP_Z specification [2,3] has also played an important role in this part of our work by allowing the Z part to be more easily analysed. Both their approach and ours need theorem proving support and follow the research direction of tool and theory integration [30]. Therefore, our work is also an inexperienced research in the direction of data abstraction mechanisation.

For future research we intend to investigate compositional results for optimal abstractions, analyse further properties beyond deadlock and livelock, classify processes according to the predicates they use, and incorporate the abstraction algorithm into our mechanised model checking strategy in order to handle infinite CSP_Z processes with minimum user assistance.

References

1. A.Mota. *Model checking CSP_Z: Techniques to Overcome State Explosion.* PhD thesis, Universidade Federal de Pernambuco, 2002.
2. A.Mota and A.Sampaio. Model-Checking CSP-Z. In *Proceedings of the European Joint Conference on Theory and Practice of Software*, volume 1382 of *LNCS*, pages 205–220. Springer-Verlag, 1998.
3. A.Mota and A.Sampaio. Model-Checking CSP-Z: Strategy, Tool Support and Industrial Application. *Science of Computer Programming*, 40:59–96, 2001.
4. A.W.Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.
5. B.Boigelot, S.Rassart, and P.Wolper. On the Expressiveness of Real and Integer Arithmetic Automata (Extended Abstract). *LNCS*, 1443:01–52, 1999.
6. C.A.R.Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
7. C.Fischer. Combining CSP and Z. Technical report, Univ. Oldenburg, 1996.
8. C.Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.
9. C.Loiseaux, S.Graf, J.Sifakis, A.Bouajjani, and S.Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. In *Formal Methods in System Design*, volume 6, pages 11–44. Kluwer Academic Publishers, Boston, 1995.

10. E.M.Clarke, O.Grumberg, and D.A.Peled. *Model Checking*. The MIT Press, 1999.
11. J.A.C.F.Neri et al. SACI-1: A cost-effective microssatellite bus for multiple mission payloads. Technical report, Instituto Nacional de Pesquisas Espaciais - INPE, 1995.
12. M.Goldsmith et al. *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd, August 2001.
13. H.Wehrheim. Data Abstraction for CSP-OZ. In J.Woodcock and J.Wing, editors, *FM'99 World Congress on Formal Methods*. LNCS 1709, Springer, 1999.
14. H.Wehrheim. Data Abstraction Techniques in the Validation of CSP-OZ Sp. In *Formal Aspects of Computing*, volume 12, pages 147–164, 2000.
15. K.Laster and O.Grumberg. Modular model checking of software. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1382 in LNCS, pages 20–35, 1998.
16. K.Stahl, K.Baukus, Y.Lakhneich, and M.Steffen. Divide, Abstract, and Model Check. *SPIN*, pages 57–76, 1999.
17. M.Huhn, P.Niebert, and F.Wallner. Verification based on local states. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1382 in LNCS, pages 36–51, 1998.
18. M.Kaufmann and J.Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. on Software Engineering*, 23(4):203–213, 1997.
19. M.Saaltink. The Z-Eves System. In *ZUM'97: The Z Formal Specification Notation*. LNCS 1212, Springer, 1997.
20. M.Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, 2nd edition, 1992.
21. P.Cousot and R.Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symp. on Principles of Programming Languages (POPL'79)*, pages 269–282. ACM Press, New York, 1979.
22. P.Cousot and R.Cousot. Abstract interpretation frameworks. *J. Logic. and Comp.*, 2(4):511–547, 1992.
23. P.Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.
24. R.Cleaveland and J.Riely. Testing-based abstractions for value-passing systems. In J. Parrow B. Jonsson, editor, *CONCUR'94*, volume 836, pages 417–432. Springer-Verlag Berlin, 1994.
25. R.Giacobazzi and F.Ranzato. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.
26. R.Lazić. *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*. PhD thesis, Oxford University, 1999.
27. S.A.Cook and D.G.Mitchell. Satisfiability Problem: Theory and Applications. In *Discrete Mathematics and Theoretical Computer Science*. AMS, 1997.
28. S.Liu. Verifying Consistency and Validity of Formal Specifications by Testing. In *FM'99 - Formal Methods*, pages 896–914. LNCS 1708, 1999.
29. S.Owre, S.Rajan, J.M.Rushby, N.Shankar, and M.K.Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV'96*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
30. Y.Kesten, A.Klein, A.Pnueli, and G.Raanan. A Perfecto Verification: Combining Model Checking with Deductive Analysis to Verify Real-Life Software. In J.M.Wing, J.Woodcock and J.Davies, editor, *FM'99-Formal Methods*, volume 1 of *LNCS 1708*, pages 173–194. Springer-Verlag, 1999.