Model-Checking Processes with States: An Industrial Case Study

Alexandre Mota and *Augusto Sampaio*
Federal University of Pernambuco
P. O. BOX 7851 Cidade Universitária
50740-540 Recife - PE Brazil
{acm,acas}@di.ufpe.br

Abstract

In this paper we present a formal specification of part of the SACI-1 microsatellite on-board computer whose development is led by the Brazilian Space Research Institute (INPE). The specification is written in CSP-Z, a specification language that integrates CSP (which allows one to focus on the concurrent aspects of the application) and Z (for modeling the relevant data structures). We also describe a strategy for model-checking processes with states (developed by the authors) and its implementation using the FDR model-checker. Finally, using this tool, we carry out an automatic proof that the SACI-1 specification is deadlock-free.

Keywords: Model-Checking, Formal Methods, Industrial Case Study, Verification, Tools, Concurrent and Model-Based Specifications.

1 Introduction

On 4 June 1996, the maiden flight of the Ariane 5 [14] launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project immediately started to investigate the failure which has resulted in the discovery of an error in the conversion of numerical data types.

Accidents of this nature are frequent due to the loss of control over innumerable aspects to be considered in the design of such complex systems. In view of this, it is important to elaborate a precise documentation that serves as a reference for building a reliable system, through a formal investigation of its main critical parts. Formal methods helps one to produce documents having these attributes. Two main requirements must be observed when a specification is to be developed. First, one should use appropriate languages to specify the system: the language should include sufficient features to capture the several aspects of the relevant application. Second, these languages should have a reasonable practical acceptance and tool support.

The specification formalism used in this paper is CSP-Z [7, 8]. The choice of CSP-Z is mainly due to the elegant way in which it combines CSP and Z, and to the wide acceptance of both CSP [11, 16] and Z [19] by the computer science and software engineering communities. CSP-Z is a conservative extension of both CSP and Z in the sense that the syntactical and semantical aspects of CSP is fully preserved while Z operations have a slightly different interpretation. Currently, there are a lot of language integration proposals. Some examples are LOTOS [3], Temporal Logic and CSP [15], LOTOS and Z [6], among others.

In [2], Mota and Sampaio present a strategy for model-checking CSP-Z specifications emphasizing deadlock analysis of complex networks of processes employing a partition technique. Some ideas were discussed on how to implement this strategy using the FDR model-checker [9]. Here we further develop this strategy by presenting conversion patterns for all CSP-Z operators into the FDR notation. In particular, we show how sequential composition of CSP-Z specifications can be treated, which has not been considered in [2]. Furthermore, we apply the extended strategy to a realistic case study: a subset of the On-Board Computer (OBC) of the SACI-1 [5] microsatellite, whose development is led by the Brazilian Space Research Institute (INPE). The whole specification can be found in [1].

The rest of this paper is organised as follows. Section 2 introduces the CSP-Z language through an example, and briefly describes its syntax and semantics. In Section 3 we describe a strategy for model-checking CSP-Z. A guideline of how to translate a CSP-Z specification into FDR is given in Section 4. Section 5 illustrates this approach through the specification and analysis of the OBC of the SACI-1. Finally, we consider what are the benefits of using an integrated language and the practical advantages and limitations of using FDR in this setting. We assume some familiarity with the languages CSP and Z.

2 An Overview of CSP-Z

The language CSP-Z [7, 8] is a conservative extension of both CSP and Z in the sense that the syntactical and semantical aspects of CSP is fully preserved while Z operations have a slightly different interpretation. In order to give an overview of CSP-Z we present part of the specification of our case study, fully described in Section 5. In [8] the integration of CSP with an object oriented extension of Z is presented. Here we consider the plain Z notation.

2.1 *A simple Example*

The Watch-Dog Timer or simply WDT is a process of the SACI-1 microsatellite responsible for waiting a reset signal that comes (periodically) from another SACI-1 process, the Fault-Tolerant Router (FTR). If this reset signal does not come, the WDT sends a recovery signal to the FTR in order to initiate a recovery process to normalise the situation. This procedure occurs three times and, if after that, the FTR does not respond, than the WDT considers the FTR faulty.

A CSP-Z specification is encapsulated into a **spec** and **end_spec** scope, where the name of the specification follows these keywords. The interface is the first part of a CSP-Z specification and is used to declare the external channels (keyword **channel**) and the local (or hidden) ones (keyword **local_channel**). Each list of channels has an associated Z schema type, where the empty schema type ([]) denotes a list of events, i.e., channels which do not communicate values. Channel types must be declared outside the **spec** and **end_spec** scope as illustrated by the given-set definition of **CLK** presented below.

[CLK]

The concurrent behaviour of the system is introduced by the keyword main, where other equations can be added to obtain a more structured CSP specification.

spec *WDT*
  channel *clockWDT*: [clk : CLK]
  channel *reset, recover, failFTR*: []
  local_channel *timeOut, noTimeOut*: []

The equation introduced below with the keyword main describes a totally independent behaviour between the processes *Signal* and *Verify* using the CSP interleaving operator (|||). *Signal* is simply characterised by waiting for consecutive reset signals, i.e., waiting for a *reset* and then (→) behaving like *Signal* again (i.e., waiting for another signal). *Verify* waits for a clock, then checks whether a reset signal arrived at the right period or not via the choice operator (□). If a *timeOut* occurs then the WDT tries to send a recovery signal to the FDR. If the FTR is not ready to synchronise in this event then the WDT assumes that the FTR is faulty and then finishes its execution (behaving like skip).

```
main=Signal ||| Verify
Signal=(reset→Signal)
Verify=(clockWDT?c→(noTimeOut→Verify
                    □ timeOut→(recover→Verify
                               □ failFTR→skip)))
```

After introducing the behaviour of the WDT, the data structures used are declared. In order to fix a timeout and to know if the clock achieved this maximum we introduce two constants, WDTtOut and WDTP. The system state (State) has simply a declarative part where is recorded the number of cycles that the WDT tries to recover the FTR and the value of the last clock received. The initialisation schema (Init) asserts that the number of cycles is initially zero.

| WDTtOut : CLK
| WDTP : CLK ↔ CLK

State ≙ [cycles : LENGTH; time : CLK]
Init ≙ [State′ | cycles′ = 0]

The following schemas are standard Z schemas (with a declaration part and a predicate which constrains the values of the declared variables) except that their names are originated 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. Further, when there is no schema name associated with a given channel, this means that no change of state occurs. An observation is that every external communication has a type, then when no type is explicit CSP-Z assumes the type signal, where the desired behaviour is merely that of a synchronisation and not a value passing. For events with an associated non-empty schema type, the Z schema must have input or output variables with corresponding names in order to exchange communicated values between the CSP and the Z parts. Hence, the input variable clk? receives values communicated through the *clockWDT* channel. For schemas where prime (') variables are omitted, we assume that no modifications occur, i.e., in the schema com_reset below it is implicit that the time component is not modified (time' = time).

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

When a Z schema has a precondition differing from true then it imposes a restriction on the occurrence of a CSP event. It is like a CSP guard, i.e., if the precondition is true then the event is allowed to occur normally, otherwise it is refused and the process behaves like the canonical deadlock process (stop).

Note that the precondition of the schema com_noTimeOut is given by the simple predicate ¬ WDTP(time, WDTtOut), meaning that the timeout has not yet occurred, whereas the precondition of com_timeOut specifies the occurrence of timeout.

$$\mathsf{com\_noTimeOut} \mathrel{\widehat{=}} [\Xi\mathsf{State} \mid \neg\ \mathsf{WDTP}(\mathsf{time}, \mathsf{WDTtOut})]$$
$$\mathsf{com\_timeOut} \mathrel{\widehat{=}} [\Delta\mathsf{State} \mid \mathsf{WDTP}(\mathsf{time}, \mathsf{WDTtOut}) \land \mathsf{cycles}' = \mathsf{cycles} + 1]$$

As already explained, the recovery process is attempted for 3 times, after which the WDT assumes that the FTR is faulty.

$$\mathsf{com\_recover} \mathrel{\widehat{=}} [\Xi\mathsf{State} \mid \mathsf{cycles} < 3]$$
$$\mathsf{com\_failFTR} \mathrel{\widehat{=}} [\Xi\mathsf{State} \mid \mathsf{cycles} = 3]$$

end_spec *WDT*

2.2 *Brief Explanation of the Semantics of CSP-Z*

The CSP semantical model assumed as standard is the Failures-Divergence model [4]. This means that a specification can be characterised by a set of pairs ($\mathcal{F}$, $\mathcal{D}$) where $\mathcal{F}$ is the failures set and $\mathcal{D}$ is the set of divergences. The failures of a process P is a set of pairs (s, X), of traces (observed events) and refusals, such that after P performs the trace s it cannot engage in any event of the refusal set X. The divergences of a process P are sets of traces such that after P performs any trace of this set it engages in an infinite loop of hidden events. The language CSP-Z is a semantical integration of CSP and Z in that it is given a Failures-Divergence meaning to Z [7, 8]. This interpretation is required in order to allow Z components to be combined using the CSP operators like interleaving (|||) and parallelism (||).

As explained above, a CSP-Z specification is a parallel combination of the CSP and the Z parts via the channel names, such that on the occurrence of a channel c the corresponding Z schema com_c is activated. As the semantics of CSP-Z is also based on the Failures-Divergence model, we should explain what happens when a given event c occurs successfully, when it is refused and when it leads to divergence. These situations are considered below.

Suppose that c is a CSP untyped channel with corresponding schema com_c. If the event c occurs, the guard of the event and the precondition of the schema com_c are satisfied, this characterises a successful execution step. In this case, the state space is subjected to the predicate part of com_c and the CSP part also evolves (where the event c is added to the trace of the process). Now, suppose that the channel c is a typed channel. If c?x is performed and the value v assigned to x cannot be treated by the input part of com_c, due to a type incompatibility, then c is refused. Similarly, if com_c exhibits a value v from one of its output variables which cannot be communicated through c!v then c is also refused. Finally, suppose that c is not refused by the Z part, according to the above explanation, then if the value communicated falsifies the precondition of com_c then the whole process diverges. A more formal presentation of the semantics is given in [8].

Formalising the above explanation, we can state precisely a refusal or a divergence introduced by the Z part. Let c be a channel and tr be a trace then

- com_c is defined as a standard Z schema operation describing the state effect on the occurrence of c
- enable_c $\widehat{=} \exists$ State′; In?; Out! • com_c is a constraint between the values communicated from the CSP part to the Z part and vice-versa
- pre com_c $\widehat{=} \exists$ State′; Out! • com_c
- com_$\langle\rangle$ $\widehat{=}$ Init, for an empty trace
- com_$(\langle c \rangle ^\frown tr)$ $\widehat{=}$ com_c $\fatsemi$ com_tr, where $^\frown$ is the concatenation operator and $\fatsemi$ is the Z schema composition operator

Thus, a refusal can occur if ¬enable_c and a divergence if enable_c ∧ ¬ pre com_c.

3 Model-Checking CSP-Z

Instead of creating a new technique and a tool for model-checking CSP-Z specifications, we used one of the most important principles of Software Engineering: *reuse*. Since CSP-Z is a language whose semantics (see Section 2.2) is based on the Failures-Divergences model of CSP it seems wise attempting to extend the existing model-checking technology for CSP in the FDR system [9]. In order to achieve that, we need to answer the following questions:

1. How to describe a state-space in CSP?
2. How to execute Z operations according to the CSP behaviour?
3. How to limit the CSP behaviour based on the Z operations and state values?

3.1 *Introducing a State-Space*

The unique way to consider CSP processes with states is to parametrise all CSP processes. For example, Roscoe [16] models an infinite buffer as follows:

$$\begin{aligned} B^{\infty}_{\langle\rangle} &= \mathsf{left?x : T} \rightarrow B^{\infty}_{\langle x\rangle} \\ B^{\infty}_{s^\frown\langle y\rangle} &= (\mathsf{left?x : T} \rightarrow B^{\infty}_{\langle x\rangle^\frown s^\frown\langle y\rangle}) \;\Box\; (\mathsf{right!y} \rightarrow B^{\infty}_{s}) \end{aligned}$$

Note that each process carries some expression which represents its "state-space". This description also reveals another interesting feature, required by CSP-Z processes: each CSP-Z process has its own state-space.

3.2 *Synchronising CSP events with Z operations*

In Section 2, we show that for each CSP event e there is a corresponding Z operation com_e, even if no com_e is present in the process scope, such that when e occurs, *com_e* is executed (simultaneously). To demonstrate how to accomplish this, we transform the previous example to obtain the following:

$$\begin{aligned} B^{\infty}_{\langle\rangle} &= \mathsf{left?x : T} \rightarrow B^{\infty}_{\mathsf{cat}(\langle x\rangle,\langle\rangle)} \\ B^{\infty}_{\mathsf{cat}(s,\langle y\rangle)} &= (\mathsf{left?x : T} \rightarrow B^{\infty}_{\mathsf{cat}(\langle x\rangle,\mathsf{cat}(s,\langle y\rangle))}) \;\Box\; (\mathsf{right!y} \rightarrow B^{\infty}_{s}) \end{aligned}$$

where $\mathsf{cat}(s_1, s_2) = s_1{}^\frown s_2$ (sequence concatenation). cat is a state-based operation which, in CSP-Z, wolud be described as a Z schema. In pure CSP, the evaluation of cat is not simultaneous to the occurrence of left. However, this is a question concerning time considerations because, by the Failures-Divergences point of view, one cannot distinguish between this and its simultaneous version. Therefore, we can adopt this strategy when converting CSP-Z operations into CSP annotations which describe the state of a process.

3.3 *CSP Behaviour as a Function of the State-Space*

The last consideration concerns how to restrict CSP behaviour based on the state-space of the process. This is explained with an example of a finite buffer, also found in [16].

$$\begin{aligned} B^{N}_{\langle\rangle} &= \mathsf{left?x : T} \rightarrow B^{N}_{\mathsf{cat}(\langle x\rangle,\langle\rangle)} \\ B^{N}_{\mathsf{cat}(s,\langle y\rangle)} &= ((\mathsf{left?x : T} \rightarrow B^{N}_{\mathsf{cat}(\langle x\rangle,\mathsf{cat}(s,\langle y\rangle))}) \nless \#s < N-1 \ngtr \mathsf{stop}) \\ &\quad \Box(\mathsf{right!y} \rightarrow B^{N}_{s}) \end{aligned}$$

The above process represents an N-place buffer, where #s is the size of the sequence s and $P \nless b \ngtr Q$ is the process that behaves like P if b is true and as Q, otherwise. From this process one can conclude that it refuses all left events when its "state-space" fills its N cells ($\#s = N$). Conditionals based on the current value of components of the state space will serve as a standard pattern to implement the preconditions of Z operations described as schemas.

3.4 *Special Considerations on Sequential Composition*

The meaning of the CSP-Z process P ; Q is that the final state of P will be the initial state of Q, whenever P terminates successfully. It differs completely from the other CSP-Z operators because it must transfer the state from one process to another. We propose a way to deal with this operator based only on its algebraic laws [16, 18].

This problem could be solved easily if we could compute P ; Q with state as

$$(\mathsf{P} \mathbin{;} \mathsf{Q})(\mathsf{S_{Initial}}) \mathrel{\widehat{=}} \mathsf{Let}\ \mathsf{S_{Final}} = \mathsf{P}(\mathsf{S_{Initial}})\ \mathsf{in}\ \mathsf{Q}(\mathsf{S_{Final}})$$

That is, if a CSP process had a value of return, this could be passed to the following process in the sequence. As this is not possible, an alternative strategy to model sequential composition is to apply exhaustively the following rule

$$(?x : A \rightarrow P) \mathbin{;} Q = ?x : A \rightarrow (P \mathbin{;} Q)$$

such that we bring the skip's —inside the definition of P— closer to the composed process Q and then apply the rule skip ; Q = Q, subsequently.

As a final example we illustrate this strategy: Suppose we have a limited buffer that when it achieves its maximum capacity it discharges its contents to a subsequent process Q which treats this data in a convenient, but not relevant, way. Thus, our buffer is given by

$$\begin{array}{lcl} B^N_{\langle\rangle} & = & left?x : T \rightarrow B^N_{cat(\langle x\rangle,\langle\rangle)} \\ B^N_{cat(s,\langle y\rangle)} & = & ((left?x : T \rightarrow B^N_{cat(\langle x\rangle,cat(s,\langle y\rangle))}) \lhd \#s < N-1 \rhd skip) \\ & & \Box(right!y \rightarrow B^N_s) \end{array}$$

and $(B^N_{\langle\rangle} \mathbin{;} Q)(S)$ is rewritten as

$$\begin{array}{lcl} B^N_{\langle\rangle} & = & left?x : T \rightarrow B^N_{cat(\langle x\rangle,\langle\rangle)} \\ B^N_{cat(s,\langle y\rangle)} & = & ((left?x : T \rightarrow B^N_{cat(\langle x\rangle,cat(s,\langle y\rangle))}) \lhd \#s < N-1 \rhd Q_{cat(s,\langle y\rangle)}) \\ & & \Box(right!y \rightarrow B^N_s) \end{array}$$

4 Translating CSP-Z into FDR

In this section we present systematic guide-lines to convert a CSP-Z specification into the FDR notation, based on the considerations of the previous section. For analysing CSP-Z using FDR we have to define the following elements: State (the system state space), Init (the initialisation schema), com_c (schema associated with channel c), pre com_c (precondition of the schema com_c) and the communication of values between the CSP and the Z parts of the specification. In general, Z operations are relations between initial and final states, as well as input and output values. However, for simplicity we assume in the following that these relations are functional.

• State: The state space is represented in FDR as a parameter (see Section 3). When a schema com_c updates the state space the final state produced must be taken as the initial state for the next "execution step".
• Init: As FDR cannot represent a state space as a global element then the Init schema is represented as a process without parameters such that it initialises the data structures used by the main equation. So, $\mathsf{Init} = \mathsf{main}(\mathsf{S_{Initial}})\backslash\mathsf{local\_channels}$, where $\mathsf{S_{Initial}}$ is a tuple defining an initial value for each state component. The hiding in the main equation is necessary because FDR does not provide a means for modularisation in the sense of scope found in CSP-Z specifications.
• com_c: A Z schema can be translated into FDR as a function. The arguments to this function are the (current) state and the values of the input variables; the function result

is formed by the final state defined by the schema and the values of the output variables. This function does not embody the precondition part of the schema, only the effect.

• pre com_c: A precondition is also encoded as an FDR function of type $\mathsf{State} \times \mathsf{Input} \twoheadrightarrow \mathbb{B}$; it evaluates to true in the states and input values which satisfy the precondition of the com_c schema, and to false otherwise.

• **Communications**: Values communicated in the CSP part of the FDR script must be passed to the Z part, and vice-versa. All conversion patterns below have the form of a CSP guarded command. For an input, the condition of the guard is a prefix choice of a suitable value for the input parameter. The expression $\mathsf{a?x} : \{\mathsf{a.x} \bullet \mathsf{x} : \mathsf{T}, \mathsf{pre\_com\_a}(\mathsf{S},\mathsf{x})\}$ is a set comprehension which generates the set of elements a.x where x ranges over T and satifies the predicate $\mathsf{pre\_com\_a}(\mathsf{S},\mathsf{x})$. For an output we simply pass the result of the Z part to the CSP part.

The following conversion patterns implement the above strategy and ease the encoding of a CSP-Z specification into FDR:

CSP-Z	FDR CSP-Z	Meaning
$P{=}a{\rightarrow}P$	`P(S)=pre_com_a(S) & (let S'=com_a(S) within a -> P(S'))`	Simple prefix
$P{=}a?x{\rightarrow}P$	`P(S)=a?x:{x | x <- T, pre_com_a(S,x)} -> (let S'=com_a(S,x) within P(S'))`	Input
$P{=}a!e{\rightarrow}P$	`P(S)=pre_com_a(S) & (let (S',e)=com_a(S) within a!e -> P(S'))`	Output
$P \setminus a$	`P(S) \ a`	Hiding
$P \Box Q$	`P(S) [] Q(S')`	External choice
$P \sqcap Q$	`P(S) |~| Q(S')`	Internal choice
$P \mid\mid\mid Q$	`P(S) ||| Q(S')`	Interleaving
$P \underset{a}{\|} Q$	`P(S) [| a |] Q(S')`	Alphatised parallel
$P \mathbin{;} Q$	`P(S) [Q(S_final of P)/SKIP]`	Sequential composition
stop	`STOP`	Stop process
skip	`SKIP`	Successfull termination
chaos(Σ)	`CHAOS(S, Σ)`	Chaos process

The translation of channel declarations, constants and free types is a straightforward syntactical conversion, as presented in [1]. Note that the conversion pattern for sequential composition was discussed in Section 3.

5 Case Study: The SACI-1 Microsatellite

In this section we present the CSP-Z specification of two processes which combined in parallel with that introduced in Section 2 results in a final specification that represents the simplified behaviour of the SACI-1 OBC. We also show how to translate the specification into our FDR representation and then we carry out a deadlock analysis using FDR.

The SACI-1 OBC is a fault-tolerant distributed processing system which combines software and hardware components [5]. Its critical parts are: its Watch-Dog Timer (WDT) and its Fault-Tolerant Router (FTR). Due to its fault-tolerant aspects, the SACI-1 was designed with redundant components. It has three WDT's, three FTR's, etc. However, for illustrative purposes we consider here a simplification of the real configuration, removing indices and presenting its behaviour. This simplification originates a slightly different behaviour when compared to the original specification and it is considered in Section 5.2.

5.1 *The CSP-Z Specification of the SACI-1 Main Components*

Fault-Tolerant Router. The FTR is responsible for some tasks and for periodically sending a reset signal to the WDT. In order to model the FTR as close as possible to its original conception we consider that it can stop temporarily or permanently. In a temporary stop, the FTR can be reanimated through a recover signal. However, in a permanent one the FTR cannot be restarted.

```
spec FTR
    channel clockFTR:[clk : CLK]
    channel reset, recover:[]
    local_channel resetWDT, task, taskDone, problem:[]
    main=(Normal ||| Problem)
    Normal=(clockFTR?c→(resetWDT→reset→Normal
                               □ task→((taskDone→Normal ; (problem→stop))))
    Problem=(recover→Problem)

  ┌ State ─────────────         │ WDTRstP : CLK
  │ time : CLK
  └────────────────────         com_task ≙ ¬ com_resetWDT

  ┌ com_resetWDT ──────────     ┌ com_clockFTR ─────
  │ ΞState                      │ ΔState
  ├────────────────────────     │ clk? : CLK
  │ WDTP(time, WDTRstP)         ├───────────────────
  └────────────────────────     │ time′ = clk?
                                └───────────────────
end_spec FTR
```

OBC Clock. As CSP-Z cannot capture precisely the temporal aspects of a system, we need some way to characterise the SACI-1 as a system dependent of time. We model a process which exhibits events, carrying clock values, that control the behaviour of the WDT and the FTR.

```
spec SCLOCK
    channel clockWDT, clockFTR : [clk : CLK]
    local_channel tic: []
    main=(clockWDT!c→skip ||| clockFTR!d→skip);(tic→main)
```

```
| noneClock : CLK
| IncC : CLK ⇸ CLK

┌ State ──────────
│ time : CLK
└─────────────────

┌ Init ───────────
│ State'
├─────────────────
│ time' = noneClock
└─────────────────

┌ com_clockWDT ───
│ ΞState
│ clk! : CLK
├─────────────────
│ clk! = time
└─────────────────

com_clockFTR ≙ com_clockWDT

┌ com_tic ────────
│ ΔState
├─────────────────
│ time' = IncC(time)
└─────────────────
```

end_spec *SCLOCK*

SACI-1. The simplified behaviour of the SACI-1 microsatellite is given by an alphabetised parallel composition ([|||]) of the previous three CSP-Z components. In this specification, the elements inside the brackets of the parallel operator are the synchronisation points.

spec *SACI-1*
  main=*(WDT* [| *{reset, recover}* |] *FTR)*
         [| *{clockWDT, clockFTR}* |] *SCLOCK*
end_spec *SACI-1*

5.2 *SACI-1 in FDR*

In this section we present the translation of the CSP-Z specification of the SACI-1 into FDR. Remember that every CSP-Z process has a main process (equation), a State schema, an Init schema and Z operations with names derived from the channel names. Hence, when translating into FDR we suffix these elements with the process name where can occur name clashing. For example, the initialisation schema (process) for *WDT* is named here Init_WDT, and so on.

The first process to be translated is the *SCLOCK* which is the most simple one, whose specification was presented in the beginning of this section. In this translation the CLK given-set is declared as a kind of free type (`datatype`) so that FDR can analyse the process. The *SCLOCK* process is the unique process of the case study that uses sequential composition. Note that in its translation the skip process does not appear neither the sequential composition operator. This was a consequence of applying the strategy discussed in Section 3.4.

```
datatype CLOCK = noneClock | clk1 | clk2 | clk3 | clk4 | clk5 | clk6
-- spec SCLOCK
channel clockWDT,clockFTR: CLOCK
channel tic

main_SCLOCK(S)=(let (S',c)=com_clockWDT_SCLOCK(S)
                within clockWDT!c -> (let S''=com_tic(S')
                                      within tic -> main_SCLOCK(S''))
                ||| let (S',c)=com_clockFTR_SCLOCK(S)
                    within clockFTR!c -> (let S''=com_tic(S')
                                        within tic -> main_SCLOCK(S'')))
-- Initialisation process
 Init_SCLOCK = main_SCLOCK(noneClock)\{tic}
-- schemas
com_clockWDT_SCLOCK(S) = (S, S) -- returns actual state and actual clock
com_clockFTR_SCLOCK(S) = com_clockWDT_SCLOCK(S)
com_tic(noneClock)     = clk1
com_tic(clk1)          = clk2
com_tic(clk2)          = clk3
com_tic(clk3)          = clk4
com_tic(clk4)          = clk5
com_tic(clk5)          = clk6
com_tic(clk6)          = noneClock
-- end_spec SCLOCK
```

The second process to be considered is the *WDT*, where the main difference from the above translation is the presence of functions which implement the pre-condition calculation of the corresponding Z schemas.

Recall that the WDTP function checks whether a time-out has occurred. The way we have implemented it in FDR is by checking if the current time is clk3 or clk6.

```
-- spec WDT
channel reset, recover, failFTR, timeOut, noTimeOut

main_WDT(S)   = Signal(S) ||| Verify(S)
Signal(S) = reset -> Signal(com_reset(S))
Verify(S) = (clockWDT?c -> ((let S'=com_clockWDT(S,c)
                          within pre_com_noTimeOut(S') &
                            noTimeOut -> Verify(S'))
                         [] (let S'=com_clockWDT(S,c)
                            within pre_com_timeOut(S') &
                              timeOut -> ((let S''=com_timeOut(S')
                                          within pre_com_recover(S'') &
                                            recover -> Verify(S''))
                                        [] (let S''=com_timeOut(S')
                                            within pre_com_failFTR(S'') &
                                              failtFTR -> SKIP)))))

WDTtOut = {clk3, clk6}
WDTP(time, timeout) = member(time, timeout)
-- Initialisation process
Init_WDT = main_WDT((0, noneClock))\{timeOut,noTimeOut,failFTR}
-- Preconditions
pre_com_noTimeOut((cycles,time)) = not WDTP(time, WDTtOut)
pre_com_timeOut((cycles,time)) = WDTP(time, WDTtOut)
pre_com_recover((cycles,time)) = cycles < 3
pre_com_failFTR((cycles,time)) = cycles == 3
-- Schemas
com_reset((cycles,time)) = (0, time)
com_clockWDT((cycles,time),clk) = (cycles, clk)
com_timeOut((cycles,time)) = (cycles + 1, time)
-- end_spec WDT
```

The last process converted is the *FTR*. Observe that the same implementation of timeout explained above is adopted here for enabling reset signals.

```
-- spec FTR
channel resetWDT
channel task, taskDone, problem -- They're local

main_FTR(S)    = Normal(S) ||| Problem(S)
Normal(S)  = clockFTR?x -> ((let S'=com_clockFTR_FTR(S,x)
                             within (pre_com_resetWDT_FTR(S') &
                                      (resetWDT -> reset -> Normal(S')))
                            [] (let S'=com_clockFTR_FTR(S,x)
                               within (pre_com_task(S') &
                                        (task -> ((taskDone -> Normal(S'))
                                                  [>(problem -> STOP)))))))

Problem(S) = (recover -> Problem(S))
-- Initialisation process
Init_FTR = main_FTR(noneClock)\{resetWDT, task, taskDone, problem}
-- WDTResetPeriod definition
WDTRstP = {clk3, clk6}
-- Preconditions
pre_com_resetWDT_FTR(S) = WDTP(S, WDTRstP)
pre_com_task(S) = not pre_com_resetWDT_FTR(S)
-- Schemas
com_clockFTR_FTR(S, c) = c
-- end_spec FTR
```

Finally we have the top level specification of the SACI-1 process, which is an alphabetised parallel composition of the three processes. Note that the processes used are the initialisation processes, since they correspond to the main processes carrying as arguments the initial values of the state space. Note also in the specification below the synchronisation points of the three processes.

```
-- spec SACI-1
System = (Init_WDT [|{|reset,recover|}|] Init_FTR)
         [|{|clockWDT, clockFTR|}|] Init_SCLOCK
-- end_spec SACI-1
```

6 Conclusion

We extended a model-checking strategy to analyse CSP-Z specifications using FDR introduced in [2] giving a more semantical treatment and considering the whole CSP-Z operators. In particular we have also explained the difficulty to model sequential composition and how to overcome it. To illustrate the practical applicability of applying the strategy we have developed a complex and realistic case study: the SACI-1 OBC [5].

The SACI-1 project as originally conceived lacked formal documentation. A first contribution to the industry was the formalisation of a subset of the SACI-1 [1]: its OBC system. The main aspect of the formalisation task was to develop a formal specification free from problems, hence our second contribution was to verify this requirement for a subset of our specification using the strategy presented here. The problems found during the formalisation and analysis were reported to the members of the SACI-1 project so that they could take the considerations into account in the implementation of the system.

An inductive approach to model-checking which deals with arbitrary topologies of CSP processes is presented in [12]. One research direction we intend to pursue is to extend our model-checking strategy to arbitrary topologies of CSP-Z processes. Because CSP-Z deals with a state space, we should also develop an induction principle to deal with infinite data structures.

Another topic for further research is the integration of tools to deal with CSP-Z specifications. In [1], it is shown how to use Z-EVES [17] to type-check the Z part of the SACI-1 specification and to refine some of its data structures. Furthermore, the ZANS animator [13] was also used in [1] to analyse the behaviour of the data structures in the Z part of the SACI-1 specification. Ideally, these tools should also be adapted to work for CSP-Z, as we did with FDR. The ultimate goal would be linking all these tools into a uniform development environment for CSP-Z.

A final remark is that although we have based our work on CSP-Z, the results could be easily adapted to other approaches to integrate CSP and Z, such as, for example [10].

7 Acknowledgements

We thank people from the Brazilian Space Research Institute (INPE), and in particular Alderico R. Paula Jr. for help in the understanding of the SACI-1. We also thank Clemens Fischer and Paulo Borba for discussions about CSP-Z and FDR, and for suggestions and criticisms which helped us to improve our approach to model-checking CSP-Z.

References

[1] A. Mota. Formalisation and Analysis of the SACI-1 Microsatellite in CSP-Z (in Portuguese). Master's thesis, Federal University of Pernambuco, 1997.

[2] A. Mota and A. Sampaio. Model-Checking CSP-Z. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 205–220. Springer-Verlag Berlin, 1998.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1987.

[4] S. D. Brookes and A. W. Roscoe. An improved failures model for communication processes. In *Lecture Notes on Computer Science*, volume 197, pages 281–305, 1985.

[5] A. R. de Paula Jr. Fault-Tolerance Aspects of the On-Board Computer of the First INPE Microsatellite for Scientific Applications. *VI Brazilian Symposium on Fault-Tolerant Computers*, August 1995.

[6] E. Boiten, H. Bowman, J. Derrick and M. Steen. Viewpoint Consistency in Z and LOTOS: A Case Study. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 644–664. Springer Verlag, 1997.

[7] C. Fischer. Combining CSP and Z. Technical report, University of Oldenburg, 1996.

[8] C. Fischer. CSP-OZ: A Combination of Object-Z and CSP. In *2nd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMODDS'97)*. Chapman Hall, 1997.

[9] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.01*, August 1996.

[10] G. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 62–81. Springer Verlag, 1997.

[11] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[12] J. N. Reed, D. M. Jackson, B. Deianov and G. M. Reed. Automated Formal Analysis of Networks: FDR Models of Arbitrary Topologies and Flow-Control Mechanisms. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 239–254. Springer-Verlag Berlin, 1998.

[13] X. Jia. *A Tutorial of ZANS - A Z Animation System*, 1995.

[14] J. L. Lions. Ariane 5: A Falha do Vôo 501. Technical report, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, 1996.

[15] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[16] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, 1998.

[17] M. Saaltink. The Z/EVES System. In *ZUM'97: The Z Formal Specification Notation*, pages 72–85. Lecture Notes in Computer Science, 1212, Springer, 1997.

[18] A. Sampaio. *An Algebraic Approach to Compiler Design.* World Scientific Publishing, 1997.

[19] M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 2nd edition, 1992.