Compositional Abstraction of CSP_Z Processes

Adalberto Farias, Alexandre Mota and Augusto Sampaio

Centre of Informatics
Federal University of Pernambuco
P.O. Box 7851, Cidade Universitária,
Zip 50732-970 - Recife - PE - BRAZIL
{acf | acm | acas }@cin.ufpe.br

Received 10 March 2008; accepted 24 April 2008

Abstract

Data abstraction is a powerful technique to overcome state explosion in model checking. For CSP_Z (a formal integration of the well-known specification languages CSP and Z), current approaches can mechanically abstract infinite domains (types) as long as they are not used in communications. This work presents a compositional and systematic approach to data abstract CSP_Z specifications even when communications are based on infinite domains. Therefore, we deal with a larger class of specifications than the previous techniques. Our approach requires that the domains (used in communications) being abstracted do not affect the behaviour of the system (data independence). This criteria is used to achieve an internal partitioning of the specification in such a way that complementary techniques for abstracting data types can be applied to the components of the partition. Afterwards, the partial results can be compositionally combined to abstract the entire specification. We propose an algorithm that implements the partitioning and show the application of the entire approach to a real case study.

Keywords: Formal Methods, Model Checking, Data Abstraction, CSP, Z, Compositionality.

1. Introduction

Integrated notations are powerful to provide separation of concerns when describing systems. The language CSP_Z [9], for example, integrates the process algebra CSP [18] and the model-based language Z [22] in such a way that behavioural and data aspects are modelled simultaneously but orthogonally; while control flow is described in CSP (the behavioural part), data aspects are modelled in Z (the data part). The syntax and semantics of the constituent languages are almost fully reused in CSP_Z, which also provides flexibility for applying techniques to perform compositional refinement and analysis. For example, process and data refinement techniques can be used relatively independently to achieve more concrete specifications. Concerning analysis, theorem proving [13] (Z proofs) or model checking [5] (CSP proofs) can be used for verifying properties in CSP_Z. The former method requires user intervention in general, whereas the latter is fully automatic, but unable to analyse systems with infinite state-spaces (the *state explosion* problem). To overcome such a limitation, several state-space compression techniques [5], like *data abstraction*, for instance, have been applied.

Abstracting a system means finding an *approximation* that preserves desirable properties. This simpler representation can be *safe* or *optimal* with respect to the original system [6]. Safe abstraction does not preserve all properties, whereas optimal abstraction represents the original system more faithfully. This work considers only optimal abstractions of CSP_Z specifications. The language yields infinite state-space systems very naturally because infinite domains are allowed as types of state and communication variables. In the first situation, the infiniteness is already handled by the data abstraction approach reported in [8, 16]. However, the infiniteness occurring in communications is still an open problem.

The approach presented in this work is based on a syntactic splitting to isolate the infiniteness problem. This originates two internal parts that can be data abstracted by specific techniques. Afterwards, the resulting abstract parts are combined to originate the abstraction for the entire system. Thus, our technique increases the class of

problems handled by data abstraction through the application of a decomposition and reusing existing techniques to analyse the originated parts separately. After that, we integrate the results and obtain a more complex abstraction. We have observed that this is simpler than analysing the entire specification.

We emphasize that our approach is applied to a CSP_Z process in isolation; we split the data (Z) part into a data dependent and a data independent part. We can also apply the approach to all CSP_Z processes (components) of a network of processes, provided the communication of the analysed component does not affect the behaviour of the other processes. In this sense, the state explosion of the entire network can be handled by applying our strategy to its components.

We also point out that our approach is related to refinement checking [18] rather than to classical model checking [5]. Thus, instead of proving a specific property in a given model, we aim at finding an optimal abstraction ($S^{\mathcal{A}}$) that preserves almost all properties of the original specification (S). The unique distinction between S and $S^{\mathcal{A}}$ occurs when communicated values are abstracted; however, if these values do not affect the behaviour of other processes, we can consider $S^{\mathcal{A}}$ instead of S (in isolation or in a network of processes) . Furthermore, as the equivalence between S and $S^{\mathcal{A}}$ is given in terms of the failures-divergences model of CSP [18], our approach allows the verification of safety and liveness properties as well as application specific properties.

The main contributions of this work are:

- a systematic strategy for partitioning Z specifications into a data independent and a data dependent components;
- the algorithmic implementation for the partitioning strategy;
- the reuse of existing techniques to overcome the state explosion problem;
- a compositional approach for abstracting infinite domains of state and communication variables in CSP_Z;
- application of the strategy to a realistic case study.

Although our strategy is developed for CSP_Z, it can be extended to other notations that associate events with state change, such as CSP_{OZ} [9] or CSP-B [19].

This work is organised as follows. Section 2 provides an overview of CSP_Z and presents part of the specification of a real system that is used as our case study; this system cannot be mechanically analysed by existing data abstraction approaches because it presents infinite communications. We propose an approach to deal with such a problem in Section 3. The approach is based on a syntactic splitting, which is described algorithmically in Section 4. Afterwards, we discuss related work in Section 5 and present our final remarks and future directions for this work in Section 6. The proofs of all lemmas and theorems presented in this work can be found in Appendix A.

2. Background on CSP_Z

The notation CSP_Z provides a convenient way for modelling concurrent systems with state information. Its semantics is defined in such a way that developers can reason about behavioural (CSP) and data (Z) aspects orthogonally. This section introduces several aspects of CSP_Z: syntax, semantics, model checking and data abstraction. We introduce CSP and Z separately.

2.1. The CSP Notation

The process algebra CSP [18] can be viewed as a notation for describing concurrent systems whose component processes interact with each other by communication, or as a collection of mathematical models that help one to reason about processes formally.

The most fundamental element in CSP is a communication event, which can be viewed as an atomic transaction (or a possible synchronisation point) between two or more processes; an event is also an abstract way of representing a real computation such as a method/function call, statement, input/output, internal action, an so on. An event occurs in a communicating *channel*, which can support data types. For example, if a channel a does not support types, it defines the event a; otherwise, it defines a family of events. The communication $a?x$ involves an input on channel a, whereas $a!y$ represents an output on the same channel. Inputs and output are generically denoted by $ch.v$, where ch is a channel and v is a value. Thus, if $v \in \mathbb{N}$, the communication $a.v$ corresponds to the infinite set of events $\{a.1, a.2, ...\}$.

On the other hand, processes are used to describe some behaviour; each process has an associated *alphabet*, which is the set of all events occurring in process's body. Thus, a process P has the alphabet αP.

The most basic processes in CSP are *STOP* and *SKIP*. The former represents a deadlock and do not communicate any observable event; the latter denotes successful termination after performing the special event $\checkmark$, which is also used to synchronise processes upon successful termination.

The construction of more complex processes is also possible by using operators. Table 1 provides a brief explanation of the main operators.

In CSP, process definitions are similar to equations, where the left-hand-side is the process name (possibly pa-

Table 1. Basic constructs for processes

Term	Explanation
$ev \rightarrow P$	$ev \rightarrow P$ is built by *prefixing* P with the event ev. This originates a process that communicates the event ev and then behaves like P.
$P \Box Q$	The process $P \Box Q$ is defined by an *external choice* of P or Q. This decision depends on the environment or on the other processes $P \Box Q$ interacts with.
$P \sqcap Q$	The process $P \sqcap Q$ is defined by an *internal choice* of P or Q. This decision is nondeterministically performed by the process itself.
$P[\![R]\!]$	The process $P[\![R]\!]$ is obtained by applying the event renaming R to the process P. For example, $(a \rightarrow STOP)[\![b/a]\!] = b \rightarrow STOP$.
$P \backslash S$	It represents a new process that is obtained by *hiding* in P the events of the set S. For example, $(a \rightarrow b \rightarrow SKIP) \backslash \{b\} = a \rightarrow SKIP$.
$P \underset{X}{\parallel} Q$	It denotes a process obtained by *synchronising* P and Q on all events from X (the *synchronisation interface*). If $\alpha P = \alpha Q$ and $\alpha P \subseteq X$, then P and Q are in full synchronisation.
$cond \,\&\, P$	The *guarded* process $cond \,\&\, P$ behaves like P only if $cond$ is valid; otherwise, it deadlocks. It behaves like "if $cond$ then P else $STOP$".
$\Box_{i:1..n} \bullet P_i$	The *indexed* external choice is equivalent to $P_1 \Box P_2 \Box \ldots \Box P_n$.

rameterised) and the right-hand-side is the process body. For example, the specification

$$channel\ tick, tack$$
$$Clock = tick \rightarrow Clock \Box tack \rightarrow Clock$$

describes the behaviour of a clock that infinitely offers *tick* or *tack* (defined by non-typed channels) using recursion. Its graphic representation is given in terms of a ***L**abelled **T**ransition **S**ystem* (Figure 1).

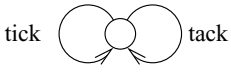

Figure 1. LTS representation of the process *Clock*.

The set $initials(P)$ denotes the set of acceptances (events) that can be performed by the process P in a specific context. For example, $initials(Clock) = \{tick, tack\}$ and $initials(a \rightarrow b \rightarrow SKIP) = \{a\}$.

The meaning of a CSP process is defined according to three models [18]: traces, failures or failures-divergences. The traces model ($\mathcal{T}$) is based on the observable behaviour, where a process is represented by a set of traces (sequences of events). For example, the processes *STOP* and *SKIP* are represented by $\{\langle\rangle\}$ and $\{\langle\rangle, \langle\checkmark\rangle\}$, respectively. The process $a \rightarrow b \rightarrow STOP$ is represented by $\{\langle\rangle\ \langle a\rangle, \langle a, b\rangle\}$; $\langle\rangle$ means no event has been performed yet, $\langle a\rangle$ denotes only a was performed, and $\langle a, b\rangle$ means the process performed a followed by b.

It is worth noting that the traces model captures what a process "can" do. Actually, processes can reject events, originating the notion of *refusals* (the events a process can reject in a context). The *failures* model ($\mathcal{F}$) captures this and represents a process as a set of failures; each one is defined as a pair (s, X) where s is a trace and X is a set of refusals after performing s. For example, the processes $a \rightarrow STOP \Box b \rightarrow STOP$ and $a \rightarrow STOP \sqcap b \rightarrow STOP$ are represented by the same set of traces ($\{\langle\rangle, \langle a\rangle, \langle b\rangle\}$). However, the second process can nondeterministically reject a before performing any event. The failures of these processes are respectively given by $\{(\langle\rangle, \varnothing), (\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$ and $\{(\langle\rangle, \varnothing), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$. Note that the first process is more predictable (deterministic) than the second in $\mathcal{F}$ because it has less failures. Intuitively, a process that offers the external choice ($\Box$) of certain events is better than (a refinement of) a process that "decides" internally ($\sqcap$) on which events to engage.

Besides traces and refusals, processes can perform internal actions that are not captured by $\mathcal{T}$ or $\mathcal{F}$. The failures-divergences model ($\mathcal{FD}$) gives meaning to processes based on their failures and *divergences*. A divergence is a trace (and all its extensions) for which a process is not deadlocked and does not show any observable behaviour (it infinitely performs internal actions). A divergent behaviour is similar to an infinite loop doing nothing.

According to [18], process refinement ($\sqsubseteq$) is defined in terms of set inclusion and the equivalence ($\equiv$) is defined in terms of refinement. That is,

- $P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow traces(Q) \subseteq traces(P)$
- $P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow traces(Q) \subseteq traces(P) \wedge$ $failures(Q) \subseteq failures(P)$
- $P \sqsubseteq_{\mathcal{FD}} Q \Leftrightarrow failures(Q) \subseteq failures(P) \wedge$ $divergences(Q) \subseteq divergences(P)$
- $P \equiv_M Q \Leftrightarrow P \sqsubseteq_M Q \wedge Q \sqsubseteq_M P$, for any CSP model M.

Concerning tool support, CSP specifications can be analysed by the refinement checker FDR [10]. The tool is able to prove properties of specifications by applying the above refinement definitions. To achieve that, the left-hand specification P must satisfy the desired property. Then an arbitrary specification Q also satisfies the same property if $P \sqsubseteq Q$. Because FDR calculates all traces, failures and divergences of processes, it is not able to deal with systems with an infinite state-space.

2.2. The Z Notation

The Z language [22] presents powerful structuring and abstraction mechanisms for describing data and sequential aspects. It is based on set theory and first-order logic, and provides two internal languages: the *mathematical* and the *schema* languages. The former is used to describe various aspects of a design: objects (abstract data types, functions, predicates, etc.), and the relationships between them; the latter is intended to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for reuse.

There are many ways of defining new types in Z. Table 2 shows the main type constructors.

Table 2. Type constructs of Z

Construct	Explanation
$[Id]$	It is a *given set* that introduces *Id* as a new type without specifying its values.
$N == Id$	It is an *abbreviation* that defines a type synonym. Thus, *N* is another name for the previously defined type *Id*.
$nat ::= zero \mid succ \langle\langle nat \rangle\rangle$	It is a *free type* that introduces the type *nat* (symbolic natural numbers) as either *zero* or the successor of a natural number. Thus, *nat* is the smallest set containing the following collection of distinct elements: *zero*, *succ zero*, *succ*(*succ zero*), *succ*(*succ*(*succ zero*)), and so on.

A Z schema is a construction where declarations and predicates are combined for defining new objects, with the general form,

```
┌─ Name ──────────
│ declaration
├──────────
│ predicate
└─────────────────
```

Schemas have a name and are suitable for modelling state, initialisation and operations. When modelling the state, the declarative part defines all state elements and the predicate establishes an invariant that must be preserved. When representing operations, the declarative part contains all manipulated variables (state, inputs and outputs) and the predicate establishes "what" a schema does (postcondition) as long as "certain" conditions (preconditions) are satisfied. When a precondition is not valid, the postcondition of a schema might generate an arbitrary state.

In this sense, schemas are relations from a before state and an input to an after state and an output. This allows one to manipulate them using operators over relations. Table 3 shows some operators and their semantics. R_1, R_2 and R are relations, whereas s is a set. The relevant operators are relational composition ($\fatsemi$), domain restriction ($\lhd$) and subtraction ($\lhd\!\!\!-$), and range restriction ($\rhd$) and subtraction ($\rhd\!\!\!-$).

Table 3. Relational operators of Z

Op	Semantics
$\fatsemi$	$R_1 \fatsemi R_2 = \{(x,y) \mid (x,z) \in R_1 \wedge (z,y) \in R_2\}$
$\lhd$	$s \lhd R = \{(x,y) \mid (x,y) \in R \wedge x \in s\}$
$\rhd$	$R \rhd s = \{(x,y) \mid (x,y) \in R \wedge y \in s\}$
$\lhd\!\!\!-$	$s \lhd\!\!\!- R = \{(x,y) \mid (x,y) \in R \wedge x \notin s\}$
$\rhd\!\!\!-$	$R \rhd\!\!\!- s = \{(x,y) \mid (x,y) \in R \wedge y \notin s\}$

A Z specification is represented as a triple containing a state, an initialisation and a set of operations; that is, $(State, Init, Ops)$. The following specification describes a simple clock, whose state contains a natural number as internal counter ($x : \mathbb{N}$). The initialisation assigns 0 to the next value of the state variable ($x' = 0$).

```
┌─ State ─────────
│ x : ℕ
└─────────────────

┌─ tick ──────────
│ ΔState
├──────────
│ x mod 2 == 0
│ x' = x + 1
└─────────────────

┌─ Init ──────────
│ State'
├──────────
│ x' = 0
└─────────────────

┌─ tack ──────────
│ ΔState
├──────────
│ x mod 2 == 1
│ x' = x + 1
└─────────────────
```

Regarding operations, the simple clock presents two schemas: *tick* and *tack*. Both of them can change the state ($\Delta State$) by incrementing the state variable ($x' = x + 1$). However, they are enabled for different range of values: while *tick* is enabled for even numbers ($x \bmod 2 = 0$), *tack* is enabled for odd ones ($x \bmod 2 = 1$).

Note that, while the above Z specification describes state change, the CSP description (represented by Figure 1) establishes a random execution of *tick*'s and *tacks*'s without considering any state information. In the next section we show how to integrate these complementary features and associate behaviour with state manipulation.

2.3. Integrating CSP and Z

The orthogonal and complementary characteristics of CSP and Z were the motivation for integrating them in a framework for describing behavioural and data aspects simultaneously. In this sense, CSP_Z reuses as much as possible the existing syntaxes and semantics to provide a more expressive language. We first present the syntactic integration of CSP and Z before giving an informal view of its operational semantics.

Figure 2 illustrates the general form of a CSP_Z specification. It may contain global data types and the process description that is composed by two parts: CSP and Z. The global types are defined before the keywords

spec/end_spec, which are used to limit the scope of the process; *ProcessName* is the name of the process.

```
Global data types

spec ProcessName
    CSP part - Interface
    CSP part - Behavioural description
    Z part
end_spec ProcessName
```

Figure 2. Structure of a CSP_Z specification

In the CSP part, the *Interface* contains channel declarations; they define all events the process can perform. The behavioural description contains process definitions; they are used to define the control flow of the entire process starting in a *main* process equation.

The data part is a Z specification that works conjointly with the CSP part. A CSP_Z specification is basically the union of a behavioural description with a data one. For example, the specification of Figure 3 describes the simple clock with control flow and state information together.

```
spec Clock
    chan tick, tack                          CSP part (Interface)
    main = (tick → main                      CSP part (Behaviour)
           □
           tack → main)
                                             Z part
    ┌ State ──────────
    │ x : ℕ
    └─────────────────
    ┌ com_tick ───────
    │ ΔState
    ├─────────────────
    │ x mod 2 == 0
    │ x' = x + 1
    └─────────────────
    ┌ Init ───────────
    │ State'
    ├─────────────────
    │ x' = 0
    └─────────────────
    ┌ com_tack ───────
    │ ΔState
    ├─────────────────
    │ x mod 2 == 1
    │ x' = x + 1
    └─────────────────
end_spec Clock
```

Figure 3. CSP_Z specification of an infinite clock

Note that the names of the Z operations were changed by adding the prefix *com_*. This associates a Z schema with a CSP channel in order to synchronise events with schema executions. Thus, the CSP part performs an event if, and only if, the Z part executes the associated operation. This allows the Z part to affect the behaviour of the CSP one (and vice-versa): invalid preconditions cause event refusal (the blocking view of CSP_Z [9]). Note that this is different of the pure Z semantics and originates an LTS affected by control flow and state information together (Figure 4). The *Init* schema yields the initial state while the CSP part performs an internal action (τ). Transitions are labelled with an event and the execution of the corresponding schema is implicit. The state (possibly) changes after each transition according to the associated operation schema.

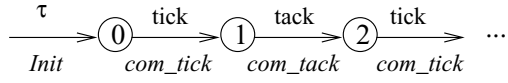

Figure 4. LTS of the CSP_Z process *Clock*

2.4. CSP_Z Model Checking

The simultaneous and synchronised execution of the behavioural and the data parts of a CSP_Z specification has been the key point for the development of a model checking strategy [17]: the CSP and the Z parts are translated into pure CSP processes (P_{CSP} and P_Z, respectively) that synchronise on all events from the *Interface*, as formalised by Equation 1.

$$P_{CSP_Z}(State) = P_{CSP} \underset{\alpha P_{CSP_Z}}{||} P_Z(State) \qquad (1)$$

The component processes have the same alphabet as P_{CSP_Z}; that is, $\alpha P_{CSP_Z} = \alpha P_{CSP} = \alpha P_Z$. Therefore, P_{CSP} and P_Z are in full synchronisation where $P_Z(State)$ is responsible only for state manipulation and has the normal form given by Definition 2.1.

Definition 2.1 *Let $P_Z(State)$ be the process representing the Z specification $(State, Init, Ops)$. The* ***normal form*** *of $P_Z(State)$ is given by*

$$P_Z(State) = \Box_{com_ev \in Ops} \bullet \\ \quad \text{pre } com_ev \ \& \ ev \rightarrow P_Z(com_ev(State)) \qquad \diamondsuit$$

The process $P_Z(State)$ is defined by a recursion whose body is an external choice of all operations ($\Box_{com_ev \in Ops}$). As long as a guard pre *com_ev* is valid, P_Z performs *ev* (in synchronisation with P_{CSP}) and recurses using an updated state ($com_ev(State)$).

Once a CSP_Z specification is represented as a process, its analysis can be carried out using any CSP model. The standard model $\mathcal{FD}$ is adopted in this work.

Although Equation 1 is a concise CSP representation of a CSP_Z specification, it cannot always be directly analysed by model checking because *State* may assume infinite values in P_Z or because αP_{CSP_Z} can be infinite. For instance, in the process *Clock* the state variable *x* is incremented at each transition indefinitely. This originates an infinite LTS representation (Figure 4). An alternative way to avoid such a problem is using *data abstraction*, which is able to determine a finite range of values for *x* such that the behaviour of the process is preserved.

2.5. CSP_Z Data Abstraction

Data abstraction [8, 16, 18, 21] is a powerful state-space compression technique, suitable for systems that manipulate data. It allows one to calculate simpler models that preserve desirable properties and are analysable via model checking.

The underlying theory of data abstraction is *abstract interpretation* [6]. It is a general framework for establishing correspondence between semantics. In this theory, elements of a concrete domain have an abstract meaning given by *abstraction relations*, such that values and operations over concrete domains are interpreted as values and operations over abstract ones. The theory allows one to find an approximation (*safe* or *optimal*) for a given concrete semantics (value, operation, etc.). They are simpler models that keep information about the actual (concrete) semantics. Safe abstractions preserve some properties, whereas optimal abstractions preserve all properties.

In CSP_Z, abstract interpretation has been used to determine the minimum values of the state variables that preserve the behaviour of the entire process. For example, recall the process *Clock* from Figure 3. There is an essential information that affects the occurrence of *tick* or *tack*: x is even or odd. Intuitively, the values 0 and 1 would be sufficient to preserve this observable behaviour. The approach proposed in [8, 16] assures this by using model checking and theorem proving; it expands the process and checks if a repeated trace is infinitely allowed by the CSP part via model checking, and by the Z part via theorem proving. For example, the trace $\langle tick, tack\rangle$ is allowed by the CSP part because it performs any sequence of *tick*'s and *tack*'s. In the Z part, the execution of the corresponding schema composition $comp \mathrel{\widehat{=}} com_tick \fatsemi com_tack$ enables the composition again (infinitely), as captured by the *stability* theorem (Equation 2).

$$\forall State;\ State' \mid \mathrm{pre}\, comp \wedge comp \bullet (\mathrm{pre}\, comp)' \qquad (2)$$

If Equation 2 is valid, the future states can be represented by the previous ones (an equivalence relation). For example, in the process *Clock*, the natural numbers are partitioned according to

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

where * means the reflexive closure operator of relations.

The equivalence classes E_{tick} and E_{tack} are used to define the abstraction function $h : \mathbb{N} \to \{0, 1\}$ as follows.

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

The function h is used to abstract the types of the variables, the constants and the post-conditions of schemas; concrete preconditions are reused by the abstract schemas. The application of h to the process *Clock* originates the abstract specification of Figure 5. Note that the abstract domain is given by the range of h; the original preconditions are preserved; and the abstract post-conditions are obtained by simply applying h to the expressions assigned to the state variable. Thus, $h(0)$ and $h(x+1)$ (where $x \in \{0, 1\}$) do not yield values outside $\{0, 1\}$. This means that the abstract domain is closed under initialisation and under the operation $+1$.

spec $Clock^A$
  chan *tick*, *tack* — CSP part (Interface)
  $main = (tick \to main$ — CSP part (Behaviour)
  $\Box$
  $tack \to main)$

Z part

$State^A$
$x : \{0, 1\}$

com_tick^A
$\Delta State^A$
$x \bmod 2 == 0$
$x' = h(x+1)$

$Init$
$(State^A)'$
$x' = h(0)$

com_tack^A
$\Delta State^A$
$x \bmod 2 == 1$
$x' = h(x+1)$

end_spec $Clock^A$

Figure 5. The abstract version of *Clock*

The LTS representation of $Clock^A$ is depicted in Figure 6, where the state explosion caused by the state variable x was overcome. If this same variable were involved in communications, the approach of [8, 16] could not be applied. In the next section we show an example that belongs to this class of problems.

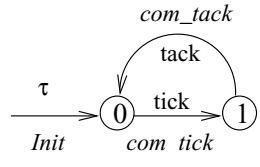

Figure 6. LTS of $Clock^A$.

2.6. A Real Example

In this section we present a process that describes a module of the on-board computer of a Brazilian artificial microsatellite [16]. Figure 7 shows the specification.

The process uses some global data types. The free-type *Message* is defined in terms of the element *nullMsg* (used for initialisation purposes) and the constructors *TC*

and *TM* (used for classifying messages as telecommands or telemetries, respectively). Both *TC* and *TM* messages have a parameter *Fields* that is defined as the cartesian product of *Data* and integers ($\mathbb{Z}$). *Data* is another free-type with three values (*nullData* denoting an empty message, *sendTM* standing for messages that should be sent to the Earth, and *extra* to represent other kinds of messages). The integer is used for recording the message ordering. Therefore, *Message* is an infinite type.

The process *Telemetry* is responsible for maintaining the most recent telemetry data (temperature, voltage, some process status, etc.) and sending them to the Earth.

In the CSP part, the channel *FTR_TM* is used for inputting the most recent telemetry message captured by the other processes, whereas the channel *sendEarth* is used to communicate the stored telemetry data to the Earth. The remaining channels do not support data communication.

The behaviour of *Telemetry* is established by its *main* equation: it first accepts an input value on *FTR_TM* (*FTR_TM*?*msg*) and sends data from the satellite to the Earth (*SEND*) or stores the new message (*STORE*). Sending data depends on whether an internal buffer is empty or not. If it is empty, the process performs *emptyTM* and behaves like *main*; otherwise, the process performs *moreTM*, sends a stored data to the Earth (*sendEarth*!*msg*) and behaves like *SEND* again.

Storing a new message also depends on the buffer status: if it is full, the process performs *storeTMFull*; otherwise it performs *storeTMNotFull*.

Concerning the data part, the state of *Telemetry* contains a variable (*currMsg*) that keeps a new message and a finite buffer of messages (represented by the sequence $STM : \mathrm{seq}\, Message$), whose size is limited by an invariant ($\#STM \leq 3$). The initialisation assigns *nullMsg* to *currMsg* and the empty sequence to the component *STM*.

The operations *com_emptyTM* and *com_moreTM* do not change the state; they use their preconditions to simply check whether *STM* is empty or not, respectively.

To make our decomposition strategy clear later on, we declare the components explicitly instead of using the Z conventions $\Delta State$ and $\Xi State$ as state change and preservation, respectively.

The remaining operations possibly yield state change. In *com_FTR_TM*, the state change is due to the input of a new message ($currMsg' = msg?$). When the internal buffer is full ($\#STM = 3$), *com_storeTMFull* discards the oldest message and stores the newest one in the last position ($STM' = tail\, STM \frown \langle currMsg \rangle$). Otherwise ($\#STM < 3$), *com_storeTMNotFull* simply appends the storage with *currMsg* ($STM' = STM \frown \langle currMsg \rangle$).

As long as the storage is not empty ($STM \neq \langle\rangle$), the operation *com_sendEarth* sends the oldest message to the Earth ($msg! = head\, STM$) and discards it from the storage ($STM' = tail\, STM$) .

The operations *com_FTR_TM* and *com_sendEarth* present an input and output, respectively. In the latter operation, the output assumes the value of a stored message. On the other hand, in *com_FTR_TM*, the input $msg? : Message$ is not specified and can be any value of type *Message*. This naturally originates state explosion that cannot be handled by any existing approach for CSP_Z, including [8, 16]. Fortunately, as the behaviour of *Telemetry* is not affected by the type *Message*, we can determine a minimum subset of it that is relevant to capture the behaviour of the system. Thus, data abstraction is still possible even when communications are based on infinite domains. In the next section we present an approach for handling such a class of problems.

3. Data Abstraction Based on Data Independence

The theory of data independence [14] is able to abstract infinite types based on syntactic properties (restrictions); it can be used in any context where these restrictions are related to the type being abstracted. We have observed that, for some systems, variables with infinite types can be isolated, even when they participate in communications. This has been the key point of our approach: using syntactic restrictions to achieve a separation of concerns and applying complementary techniques to abstract infinite types. In this sense, our approach uses decomposition and compositional reasoning in the context of data abstraction.

The decomposition uses the data independence criteria (reproduced in Definition 3.1) to originate an internal *partition* of the Z part. This allows one to isolate the infiniteness problem occurring in variables (state and communication), in such a way that data independence can be used to abstract their domains. The remaining variables are analysed according to our data abstraction approach.

Definition 3.1 *A system P is data independent with respect to a data type X if, and only if:*

(1) it must not contain constants, only input/output variables of type X;

(2) it may contain only equality tests and polymorphic operations involving type X;

(3) it may contain more complex operations, as long as they are defined in terms of equality tests and polymorphic operations;

(4) no replicated constructs (such as indexed parallelism) over the data type may appear, other than replicated nondeterministic choices. ♢

$Data ::= nullData \mid sendTM \mid extra$
$Fields == Data \times \mathbb{Z}$
$Message ::= nullMsg \mid TC\langle\langle Fields\rangle\rangle \mid TM\langle\langle Fields\rangle\rangle$

spec *Telemetry*

chan $FTR_TM, sendEarth : [msg : Message]$ (*CSP part - Interface*)
chan $emptyTM, moreTM, storeTMFull, storeTMNotFull$

$main = FTR_TM?msg \rightarrow (SEND \Box STORE)$ (*CSP part - behaviour*)

$SEND = emptyTM \rightarrow main$
$\Box$
$moreTM \rightarrow sendEarth!msg \rightarrow SEND$

$STORE = storeTMFull \rightarrow main$
$\Box$
$storeTMNotFull \rightarrow main$

(*Z part*)

State
$STM : \mathrm{seq}\, Message$
$currMsg : Message$
$\#STM \leq 3$

Init
$STM' : \mathrm{seq}\, Message$
$currMsg' : Message$
$STM' = \langle\,\rangle$
$currMsg' = nullMsg$

com_FTR_TM
$STM, STM' : \mathrm{seq}\, Message$
$currMsg, currMsg' : Message$
$msg? : Message$
$STM' = STM$
$currMsg' = msg?$

com_emptyTM
$STM, STM' : \mathrm{seq}\, Message$
$currMsg, currMsg' : Message$
$STM = \langle\,\rangle$
$STM' = STM$
$currMsg' = currMsg$

com_storeTMFull
$STM, STM' : \mathrm{seq}\, Message$
$currMsg, currMsg' : Message$
$\#STM = 3$
$STM \neq \langle\rangle$
$STM' = tail\, STM \frown \langle currMsg\rangle$
$currMsg' = currMsg$

com_moreTM
$STM, STM' : \mathrm{seq}\, Message$
$currMsg, currMsg' : Message$
$STM \neq \langle\,\rangle$
$STM' = STM$
$currMsg' = currMsg$

com_storeTMNotFull
$STM, STM' : \mathrm{seq}\, Message$
$currMsg, currMsg' : Message$
$\#STM < 3$
$STM' = STM \frown \langle currMsg\rangle$
$currMsg' = currMsg$

com_sendEarth
$STM, STM' : \mathrm{seq}\, Message$
$currMsg, currMsg' : Message$
$msg! : Message$
$STM \neq \langle\rangle$
$msg! = head\, STM$
$STM' = tail\, STM$
$currMsg' = currMsg$

end_spec *Telemetry*

Figure 7. The process *Telemetry*

The items of Definition 3.1 define degrees of independence of a process P with respect to a data type X. Based on these items, it is possible to calculate the minimum cardinality of X to preserve P's behaviour: the *threshold* of P with respect to X or $tld(P, X)$, for short. For instance, if P satisfies *(1)* then $tld(P, X) = 1$. Thus, the type X must have at east 1 element, or $\#X \geq 1$ is the unique constraint the type X must satisfy to preserve P's behaviour. This is the key idea we use to abstract the data type manipulated by a CSP_Z specification. For instance, in Figure 1 the Z part is data independent with respect to the state variable, whereas in Figure 7 the data independence property is re-

lated to the communication (input/output) variables.

Actually, data independence does not distinguish state from communication variables. Concerning the latter, there is preservation of behaviour, but communications (with concrete values) involving the process and its environment are lost when restricting the cardinality of the domain. Therefore, our approach is only applicable to a CSP_Z system, as long as its communicated values do not affect the other CSP_Z components it interacts with (so-called closed systems). Furthermore, the reduction of the communicated values (of type X) of a process is achieved by applying a renaming of events that reduces the data communicated by P to a set whose cardinality is greater than or equal to $tld(P,X)$. In this case, the renaming *preserves* P's behaviour (Lemma 3.1). For example, let P be a process that performs $ch.x$ ($x \in \mathbb{N}$) and $tld(P,\mathbb{N}) = 1$. Suppose that, by data independence analysis, we restrict $\mathbb{N}$ to the set $\{0\}$. Thus, $ch.x$ is replaced with $ch.0$ and P's behaviour is preserved because $\#\{0\} \geq tld(P,\mathbb{N})$.

Lemma 3.1 *Let P be a CSP process. Let c be a channel of P with type T_c and $R : A \rightarrow B$ a renaming function, such that $A = \{c.v \in \alpha P\}$ and $B = \{c.v' \in \Sigma\}$. If P is data independent with respect to T_c and $\# ran(\{c\} \lhd R) \geq tld(P,T_c)$, then applying R **preserves** P's behaviour.* $\diamondsuit$

Lemma 3.1 follows directly from data independence [14] and extends the idea of preservation of behaviour considering all channels of a process; Σ represents the set of all events a process can perform, and $\lhd$ is an extended version of the domain restriction operator of Z [22], used for filtering relations. The type of a channel ch is denoted by T_{ch}. As events have the form ch or $ch.v$, we use $\lhd$ to filter the events occurring in a channel. Thus, $\lhd : \mathbb{P}\,\Sigma \times (\Sigma \leftrightarrow \Sigma) \nrightarrow (\Sigma \leftrightarrow \Sigma)$ such that

$$A \lhd S = \{(x,y) \mid x \in A \wedge ((x,y) \in S \vee \exists\, v : T_x \bullet (x.v, y) \in S)\}$$

For example, let $S = \{(a.1,b),(a.2,c),(e,f),(c,f)\}$. Then, $\{a\} \lhd S = \{(a.1,b),(a.2,c)\}$ and $\{c\} \lhd S = \{(c,f)\}$.

Note that, if no values are communicated by channel c, R becomes the identity map and, hence, $P = P[\![R]\!]$. Moreover, when specific (or data dependent) operations of the type being abstracted are used, data independence is not applicable. However, we can still isolate the data independent aspects to apply a complementary technique to deal with the data dependent aspects separately. This is achieved by using Definition 3.1 to factor out the Z part, originating a partition of it.

Figure 8 illustrates the steps of the complete strategy. Step **1** splits the Z part of a CSP_Z process, originating two internal subparts: one data independent (DI) and another data dependent (DD). Then, Step **2** translates all structures (CSP part, DI and DD subparts) into CSP processes according to the strategy proposed in [17]. This originates a compound process that can still have an infinite state-space. To overcome this problem, Step **3** applies data independence to the parallelism of the CSP part and the DI component, and data abstraction to the DD component. The latter task gives an abstract process ($P_Z^{dd\mathcal{A}}$) that is combined with the data independent part ($P_{CSP} \underset{Interface^{\mathcal{A}}}{||} P_Z^{di}$) to produce the abstraction for the entire process, considering a new and finite synchronisation interface ($Interface^{\mathcal{A}}$).

To split the data part we introduce some definitions.

Definition 3.2 *A Z specification is **DI** if it is data independent with respect to the types of its variables (state and communication).* $\diamondsuit$

Note that a DI specification is classified according to Definition 3.1 and, therefore, can be data abstracted by data independence. The other category of specifications (Definition 3.3) is also used when values of the type being abstracted occur in data dependent operations.

Definition 3.3 *A Z specification is **DD** if it is not DI and has no infinite inputs (communicated to the environment) in data dependent operations.* $\diamondsuit$

Note that Definitions 3.2 and 3.3 are almost complementary. Actually, Definition 3.3 is complementary to Definition 3.2 with an extra restriction: infinite inputs are not allowed. This is necessary because our data abstraction strategy is able to deal only with finite data dependent communications. The essential advantage of using Definitions 3.2 and 3.3 is to provide a partition of a Z specification, where infinite communications are data independent. Thus, if a Z specification can be decomposed into two specifications such that Definitions 3.2 and 3.3 are satisfied by each resulting specification separately, a *simple bi-partition* is originated (Definition 3.4). We assume that a schema sch belongs to the specification $(State, Init, Ops)$ if it is the state, the initialisation or one of the operations ($sch \in (\{State\} \cup \{Init\} \cup Ops)$), and that two schemas are *disjoint* if they do not have variables in common.

Definition 3.4 *Let $Z_{spec} = (State, Init, Ops)$, $Z_{spec}^{di} = (State^{di}, Init^{di}, Ops^{di})$ and $Z_{sp}^{dd} = (State^{dd}, Init^{dd}, Ops^{dd})$ be Z specifications. If Z_{spec}^{di} is DI, Z_{spec}^{dd} is DD and for all schema $sch \in Z_{spec}$ there are two corresponding and disjoint schemas $sch^{di} \in Z_{spec}^{di}$ and $sch^{dd} \in Z_{spec}^{dd}$ such that, $sch = sch^{di} \wedge sch^{dd}$, then Z_{spec}^{di} and Z_{spec}^{dd} form a **simple bi-partition** of Z_{spec}.* $\diamondsuit$

Definition 3.4 involves the notions of DI specification, DD specification and disjointness of schemas. This allows one to reason about compositional behaviour of

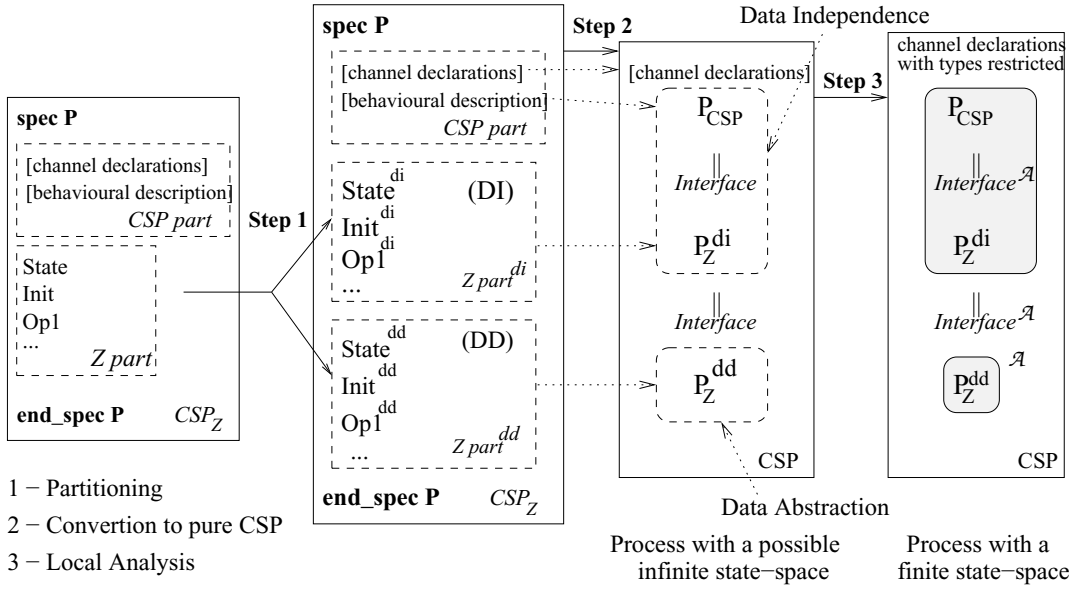

Figure 8. Overview of the strategy

Z specifications: executing an operation is similar to simultaneously executing (parallelism) its DI and DD components. We also use this idea when applying data abstraction: each component is analysed separately and the results are combined to yield a solution for the entire specification. Thus, before abstracting types, we convert each component into a process and capture the entire behaviour by the parallelism of such processes, considering all events from the *Interface*. This follows the same idea as that presented in [18], where parallelism captures conjunction of specifications. The correspondence between the original and the compound CSP representation of a Z specification is formalised by Theorem 3.1.

Theorem 3.1 *Let $Z_{spec} = (State, Init, Ops)$, $Z^{di}_{spec} = (State^{di}, Init^{di}, Ops^{di})$ and $Z^{dd}_{spec} = (State^{dd}, Init^{dd}, Ops^{dd})$ be Z specifications such that Z^{di}_{spec} and Z^{dd}_{spec} form a simple bi-partition of Z_{spec}. Let P_Z, P^{di}_Z and P^{dd}_Z be CSP processes capturing the behaviours of Z_{spec}, Z^{di}_{spec} and Z^{dd}_{spec}, respectively. Then,*

$$P_Z(State) = P^{di}_Z(State^{di}) \underset{\alpha P_Z}{\parallel} P^{dd}_Z(State^{dd}) \qquad \diamondsuit$$

Using Theorem 3.1 in Equation 1, and the associativity of the parallel operator [18], we obtain

$$P_{CSP_Z}(State) = (P_{CSP} \underset{\alpha P_Z}{\parallel} P^{di}_Z(State^{di})) \underset{\alpha P_Z}{\parallel} P^{dd}_Z(State^{dd}) \quad (3)$$

Note that Equation 3 separates a CSP_Z process into two component processes: one data independent ($P_{CSP} \underset{I}{\parallel} P^{di}_Z(State^{di})$) and another data dependent ($P^{dd}_Z(State)$). This allows the application of data independence to the first component and data abstraction to the second one [16]. The results of this separated analysis can be compositionally combined to yield the abstraction for the entire process. Of course, because communicated data are abstracted, we must consider a more restricted set of events performed by the abstract process (an abstract interface); it is calculated by applying a special renaming (*interface abstraction*) to the concrete events. As events are associated to channels, we use a renaming function (r_{ev}) for each typed channel *ev* to map events involving values from an infinite domain (the type of the channel) to values from a finite one (the abstract type); if the channel is non-typed, the corresponding renaming is the identity over its name. For example, suppose that $\mathbb{N}$ is the type of *ev* and $h_{ev} : \mathbb{N} \twoheadrightarrow \{0\}$, where $\twoheadrightarrow$ stands for total surjections, is given by $h_{ev}(x) = 0$. Then, $r_{ev} = \{x : \mathbb{N} \bullet ev.x \mapsto ev.h_{ev}(x)\}$.

The union of all renaming functions of a process originates a renaming for the entire interface, as captured by the following definition.

Definition 3.5 *Let P_Z be the process representation of a Z specification. Let chs be the channels of P_Z and r_{ev} a renaming function for the channel ev ($ev \in chs$). The* ***interface abstraction*** *of P_Z is given by*

$$R = \bigcup_{ev \in chs} r_{ev} \qquad \diamondsuit$$

The interface abstraction maps concrete events into abstract ones, by only restricting data (channel names are preserved). Its use in Equation 3 allows one to define the

abstract version of a CSP_Z process, expressed as a parallelism of a data independent and a data dependent processes considering $\operatorname{ran} R$ as the abstract interface. This is formalised in [16] and reproduced in Theorem 3.2.

Theorem 3.2 *Let P_{CSP} be a CSP_Z process with interface I. Let P_{di} and P_{dd} be CSP processes such that P_{dd} is data dependent, P_{di} is data independent, and $P = P_{di} \underset{I}{\parallel} P_{dd}$. Let $P_{dd}^{\mathcal{A}}$ be an optimal abstraction for P_{dd} with interface abstraction R. If R preserves P_{di}'s behaviour, then*

$$P_{CSP_Z}^{\mathcal{A}} = P_{di} \underset{\operatorname{ran} R}{\parallel} P_{dd}^{\mathcal{A}} \qquad \diamondsuit$$

According to Theorem 3.2 the abstract version of Equation 3 (obtained by Step **3** of Figure 8) is given by

$$P_{CSP_Z}^{\mathcal{A}}(State) = (P_{CSP} \underset{\operatorname{ran} R}{\parallel} P_Z^{di}(State^{di})) \underset{\operatorname{ran} R}{\parallel} P_Z^{dd\,\mathcal{A}}(State^{dd}) \quad (4)$$

It is worth pointing out that the partitioning strategy focuses on originating components whose schemas are disjoint (they do not have variables in common). Nevertheless, this is not true in general. In the next section we show how to deal with this for a specific (but significant) class of problems.

3.1. Extending the Partitioning Strategy

In the ideal scenario, the components originated by the partitioning strategy are disjoint and valid (all expressions in the predicate part of each schema refer to variables occurring in the declaration part). Nevertheless, this is not true in general. If there exist at least one schema whose components are non-disjoint, our data abstraction approach cannot be applied. On the other hand, if an expression of a component refers to a variable of the other (disjoint) component, we can still abstract domains, as long as the expression is data independent with respect to the type of the variable. Furthermore, we have to check, at the end, if the abstract domain respects the minimum cardinality required by the expression.

To validate two disjoint component schemas, we need to adjust them by adding a new declaration and applying a syntactic substitution. For example, consider the following state schemas $State^{di}$ and $State^{dd}$ (inv_y is a data dependent invariant over y), and the operation schema op, which contains a data dependent predicate p_y with respect to the type T_y.

$State^{di}$
$x : T_x$

$State^{dd}$
$y : T_y$
inv_y

op
$x : T_x$
$y : T_y$
$x' = y$
p_y

When partitioning op, the predicate p_y is placed into the data dependent component (op^{dd}) and the predicate $x' = y$ is placed into op^{di}.

op^{di}
$x : T_x$
$x' = y$

op^{dd}
$y : T_y$
p_y

Note that op^{di} is not a valid schema because y has not been declared. To fix this problem we introduce a new input variable ($in? : State^{dd}$) in the declaration part op^{di} and replace all occurrences of y in the predicate part with $in?.y$; this is achieved through the syntactic substitution $[in?.y/y]$. Dually, the schema op^{dd} also receives $State^{di}$ as an input parameter that is not used in the predicate part. This is similar to the idea adopted in [3], where new events between decomposed operations are created to solve dependencies and to maintain the original semantics (action refinement). In this work we just exchange the states between the component operations. Thus, op^{di} and op^{dd} become

op^{di}
$x : T_x$
$in? : State^{dd}$
$x' = in?.y$

op^{dd}
$y : T_y$
$in? : State^{di}$
p_y

We also point out that this technique is possible because $x' = y$ is a data independent expression with respect to T_y. Therefore, the values of y can be abstracted and must respect the minimum cardinality required by $x' = y$. On the other hand, if an expression of a component is data dependent with respect to the type of the variable placed in the other partition, this adjustment cannot be applied. This is pointed as a topic for future work in Section 6.

To make the CSP representation of the Z part uniform, we consider the extended normal form given in Definition 3.6. It is obtained by substituting P_Z with $P_{Z_{ext}}$ in Definition 2.1 and by adding a new event before offering all enabled events.

Definition 3.6 *Let $P_Z(State)$ be the process representing the Z specification $(State, Init, Ops)$. Let $(State^{di}, Init^{di}, Ops^{di})$ and $(State^{dd}, Init^{dd}, Ops^{dd})$ be Z specifications that form a simple bi-partition of $(State, Init, Ops)$. The* ***extended normal form*** *of $P_Z(State)$ is given by,*

$$P_{Z_{ext}}(State) = communicate.State^{di}.State^{dd} \rightarrow P_Z(State)[P_{Z_{ext}}/P_Z] \qquad \diamondsuit$$

The purpose of $communicate.State^{di}.State^{dd}$ is only to communicate the components of $State$. Its occurrence does not affect the compound form of a simple bi-partition, as stated by Theorem 3.3.

Theorem 3.3 *Let $Z_{spec} = (State, Init, Ops)$, $Z^{di}_{spec} = (State^{di}, Init^{di}, Ops^{di})$ and $Z^{dd}_{spec} = (State^{dd}, Init^{dd}, Ops^{dd})$ be Z specifications such that Z^{di}_{spec} and Z^{dd}_{spec} form a simple bi-partition of Z_{spec}. Let $P_{Z_{ext}}, P^{di}_{Z_{ext}}$ and $P^{dd}_{Z_{ext}}$ be CSP processes in the extended normal form capturing the behaviour of Z_{spec}, Z^{di}_{spec} and Z^{dd}_{spec}, respectively. Then,*

$$P_{Z_{ext}}(State) = P^{di}_{Z_{ext}}(State^{di}) \underset{\alpha P_{Z_{ext}}}{\|} P^{dd}_{Z_{ext}}(State^{dd}) \qquad \diamondsuit$$

In Theorem 3.3, $P^{di}_{Z_{ext}}(State^{di})$ and $P^{dd}_{Z_{ext}}(State^{dd})$ are respectively given by

$$\begin{array}{l} communicate!State^{di}?s^{dd} \rightarrow \Box_{ev \in chs} \bullet \\ \quad \text{pre } com_ev^{di} \ \& \ ev \rightarrow P^{di}_{Z_{ext}}(com_ev^{di}(State^{di}, s^{dd})) \end{array}$$

and

$$\begin{array}{l} communicate?s^{di}!State^{dd} \rightarrow \Box_{ev \in chs} \bullet \\ \quad \text{pre } com_ev^{dd} \ \& \ ev \rightarrow P^{dd}_{Z_{ext}}(com_op^{dd}(s^{di}, State^{dd})) \end{array}$$

The way $communicate.State^{di}.State^{dd}$ is used in $P^{di}_{Z_{ext}}$ allows the process to output its state ($!State^{di}$) and input the state of the data dependent component ($?s^{dd}$). Dually, it also allows $P^{dd}_{Z_{ext}}$ to input the state of the data independent component ($?s^{di}$) and output its state ($!State^{dd}$). Although an operation *com_ev* has two disjoint states as parameters, it yields a new state for its corresponding component. Furthermore, as the events occurring on channel *communicate* are limited to the scope of $P^{di}_{Z_{ext}} \underset{\alpha P_{Z_{ext}}}{\|} P^{dd}_{Z_{ext}}$, they can be hidden. Therefore, Theorem 3.1 is still valid when considering the extended normal form (no internal actions, failures or divergences are originated). That is, there is a correspondence between $P_{Z_{ext}} \backslash \{| communicate |\}$ and P_Z (Theorem 3.4).

Theorem 3.4 *Let $P_{Z_{ext}}$ and P_Z be CSP process representations for the Z specification $(State, Init, Ops)$ such that P_Z is in the normal form and $P_{Z_{ext}}$ is in the extended normal form. Then,*

$$P_{Z_{ext}} \backslash \{| \ communicate \ |\} = P_Z \qquad \diamondsuit$$

In the next section we show the application of the splitting strategy to the process *Telemetry*.

3.2. The Splitting of Telemetry

Considering the schema *State* of the process *Telemetry* (Figure 7), we analyse two declarations. As the invariant involves the variable *STM* in a data dependent operation, we place *currMsg* into $State^{di}$ and *STM* into $State^{dd}$.

$$\begin{array}{|l} \hline State^{di} \\ currMsg : Message \\ \hline \end{array} \qquad \begin{array}{|l} \hline State^{dd} \\ STM : \mathrm{seq}\, Message \\ \hline \#STM \leq 3 \\ \hline \end{array}$$

Regarding the initialisation, we expand $State'$ and observe that the predicate of the invariant involves STM'. Therefore, the initialisation splitting yields

$$\begin{array}{|l} \hline Init^{di} \\ currMsg' : Message \\ \hline currMsg' = nullMsg \\ \hline \end{array} \qquad \begin{array}{|l} \hline Init^{dd} \\ STM' : \mathrm{seq}\, Message \\ \hline STM' = \langle\, \rangle \\ \#STM' \leq 3 \\ \hline \end{array}$$

Concerning operations, *com_emptyTM* is split into two schemas ($com_emptyTM^{di}$ and $com_emptyTM^{dd}$) that do not change their respective states. The variable *in*? is not used by the $com_emptyTM^{di}$ because they are disjoint and do not access variables placed in $com_emptyTM^{dd}$ (and vice-versa).

$$\begin{array}{|l} \hline com_emptyTM^{di} \\ currMsg : Message \\ currMsg' : Message \\ in? : State^{dd} \\ \hline currMsg' = currMsg \\ \hline \end{array} \qquad \begin{array}{|l} \hline com_emptyTM^{dd} \\ STM : \mathrm{seq}\, Message \\ STM' : \mathrm{seq}\, Message \\ in? : State^{di} \\ \hline STM = \langle\rangle \\ STM' = STM \\ \#STM \leq 3 \\ \#STM' \leq 3 \\ \hline \end{array}$$

In *com_FTR_TM* the declarations $currMsg : Message$, $currMsg' : Message$ and $msg? : Message$ and the predicate $currMsg' = msg?$ are placed into $com_FTR_TM^{di}$, whereas the remaining declarations and predicates belong to $com_FTR_TM^{dd}$. This means that only $State^{di}$ is changed by the input *msg*?.

$$\begin{array}{|l} \hline com_FTR_TM^{di} \\ currMsg : Message \\ currMsg' : Message \\ msg? : Message \\ in? : State^{dd} \\ \hline currMsg' = msg? \\ \hline \end{array} \qquad \begin{array}{|l} \hline com_FTR_TM^{dd} \\ STM : \mathrm{seq}\, Message \\ STM' : \mathrm{seq}\, Message \\ in? : State^{di} \\ \hline STM' = STM \\ \#STM \leq 3 \\ \#STM' \leq 3 \\ \hline \end{array}$$

The splitting of the operation *com_sendEarth* yields

$$\begin{array}{|l} \hline com_sendEarth^{di} \\ currMsg : Message \\ currMsg' : Message \\ msg! : Message \\ in? : State^{dd} \\ \hline msg! = head\, STM \\ currMsg' = currMsg \\ \hline \end{array} \qquad \begin{array}{|l} \hline com_sendEarth^{dd} \\ STM : \mathrm{seq}\, Message \\ STM' : \mathrm{seq}\, Message \\ in? : State^{di} \\ \hline STM \neq \langle\rangle \\ \#STM \leq 3 \\ \#STM' \leq 3 \\ STM' = tail\, STM \\ \hline \end{array}$$

Note that $com_sendEarth^{di}$ and $com_sendEarth^{dd}$ are disjoint but the former is not valid because it uses a variable (STM) declared in the latter. To fix this, we replace all occurrences of STM in the predicate part with $in?.STM$. This is possible because $head\ STM$ is a polymorphic (data independent) operation. Thus,

$$\begin{array}{|l}
\hline
com_sendEarth^{di} \\
currMsg : Message \\
currMsg' : Message \\
msg! : Message \\
in? : State^{dd} \\
\hline
msg! = head\ in?.State^{dd} \\
currMsg' = currMsg \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
com_sendEarth^{dd} \\
STM : \mathrm{seq}\, Message \\
STM' : \mathrm{seq}\, Message \\
in? : State^{di} \\
\hline
STM \neq \langle\rangle \\
\#STM \leq 3 \\
\#STM' \leq 3 \\
STM' = tail\ STM \\
\hline
\end{array}$$

The splitting of $com_storeTMFull$ and $com_storeTMNotFull$ also needs this adjustment.

$$\begin{array}{|l}
\hline
com_storeTMFull^{di} \\
currMsg : Message \\
currMsg' : Message \\
in? : State^{dd} \\
\hline
currMsg' = currMsg \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
com_storeTMFull^{dd} \\
STM : \mathrm{seq}\, Message \\
STM' : \mathrm{seq}\, Message \\
in? : State^{di} \\
\hline
\#STM = 3 \\
STM \neq \langle\rangle \\
STM' = tail\ STM \frown \langle in?.currMsg \rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
com_storeTMNotFull^{di} \\
currMsg : Message \\
currMsg' : Message \\
in? : State^{dd} \\
\hline
currMsg' = currMsg \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
com_storeTMNotFull^{dd} \\
STM : \mathrm{seq}\, Message \\
STM' : \mathrm{seq}\, Message \\
in? : State^{di} \\
\hline
\#STM < 3 \\
STM' = STM \frown \langle in?.currMsg \rangle \\
\hline
\end{array}$$

After partitioning the process *Telemetry*, we translated the components to CSP, using the extended normal form (Definition 3.6) for the Z part. Then we represented the entire process according to Equation 3.

The analysis of $P_{CSP} \underset{I}{\parallel} P_Z^{di}(State^{di})$ by data independence revealed that $tld(P_{CSP} \underset{I}{\parallel} P_Z^{di}(State^{di}), Message)=1$. The application of data abstraction to $P_Z^{dd}(State^{dd})$ produced $Message^{\mathcal{A}} = \{nullMsg\}$ as abstract domain, the abstract process $P_Z^{dd\mathcal{A}}$ and the interface abstraction given by

$$\begin{aligned}
R = {} & \{emptyTM \mapsto emptyTM, moreTM \mapsto moreTM\} \cup \\
& \{storeTMFull \mapsto storeTMFull\} \cup \\
& \{storeTMNotFull \mapsto storeTMNotFull\} \cup \\
& \{x : Message \bullet sendEarth.x \mapsto sendEarth.h(x)\} \cup \\
& \{x : Message \bullet FTR_TM.x \mapsto FTR_TM.h(x)\}
\end{aligned}$$

where the abstraction function $h : Message \rightarrow Message^{\mathcal{A}}$ (also calculated by the approach) is given by

$$h(m) = nullMsg$$

Because $Message$ is communicated on channel FTR_TM and $\#(\{FTR_TM\} \lhd R\}) \geq 1$, R preserves the behaviour of $P_{CSP} \underset{I}{\parallel} P_Z^{di}(State^{di})$. The abstract version of *Telemetry* is then given by

$$P_{CSP_Z}^{\mathcal{A}}(State) = P_{CSP} \underset{\mathrm{ran}\, R}{\parallel} P_Z^{di}(State^{di}) \underset{\mathrm{ran}\, R}{\parallel} P_Z^{dd}(State^{dd})^{\mathcal{A}}$$

where the types of the channels FTR_TM and $sendEarth$ were changed from $Message$ to $Message^{\mathcal{A}}$.

The abstract version of the components of com_FTR_TM are given by

$$\begin{array}{|l}
\hline
com_FTR_TM^{di^A} \\
currMsg : Message^A \\
currMsg' : Message^A \\
msg? : Message^A \\
in? : State^{dd^A} \\
\hline
currMsg' = h(msg?) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
com_FTR_TM^{dd^A} \\
STM : \mathrm{seq}\, Message^A \\
STM' : \mathrm{seq}\, Message^A \\
in? : State^{di^A} \\
\hline
STM' = STM \\
\#STM \leq 3 \\
\#STM' \leq 3 \\
\hline
\end{array}$$

We have verified safety and liveness properties of *Telemetry* using FDR [10] (see Figure 9). The Telemetry partitioned and renamed is the process

$$\begin{aligned}
& TelemetryPartRen = \\
& \quad ((main \underset{Interface}{\parallel} P_Z^{di}(State^{di})) \underset{Interface}{\parallel} P_Z^{dd}(State^{dd}))[\![R]\!]
\end{aligned}$$

And the abstract Telemetry partitioned is given by

$$\begin{aligned}
& AbsTelemetryPart = \\
& \quad (main \underset{\mathrm{ran}\, R}{\parallel} P_Z^{di}(State^{di^A})) \underset{\mathrm{ran}\, R}{\parallel} P_Z^{dd\mathcal{A}}(State^{dd^A})
\end{aligned}$$

Both processes were deadlock-free, livelock-free and equivalent in $\mathcal{FD}$.

To provide a mechanisation of the partitioning strategy, we propose an algorithm that originates a simple bipartition of the Z part of a CSP_Z specification.

4. A Partitioning Algorithm

The partitioning strategy presented in the previous section is systematic and syntactic-based. In this section we describe it algorithmically.

We use some auxiliary functions. The most important function determines if a given expression is data independent with respect to a given type. The function $is_di : Exp \times T \rightarrow Boolean$ (Figure 10) implements the data independence classification according to Definition 3.1. Note that is_di could also be described in terms

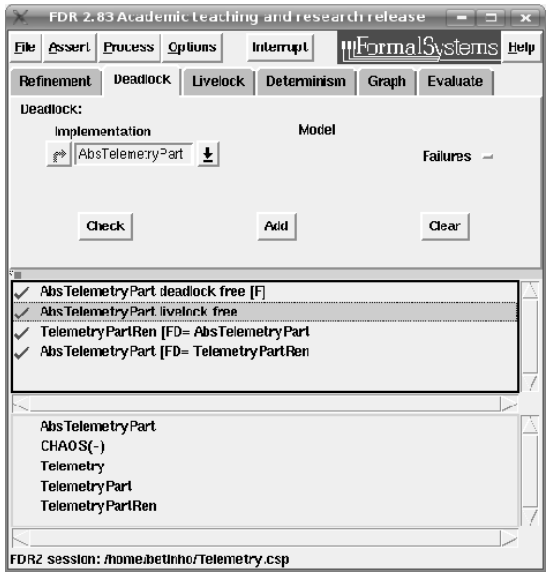

Figure 9. Analysis of Telemetry in FDR

of syntactic analysis of expressions, as proposed in the literature of compilers [1]. However, such a representation would require a detailed description of all kinds of expressions that are allowed (a subset of the Z grammar). In this work we use a natural language style for conciseness.

We assume that schemas are normalised [22]; that is, declarations have the form $v : T_v$ where v is a variable of type T_v, and all constraint information (including the invariant) appears in the predicate part, which must be in the conjunctive normal form. This is necessary because the declaration and the predicate parts are split, originating schemas whose conjunction must be equal to the original one (Definition 3.4). This is established by the semantics of the schema conjunction [22], where declarations are merged and predicates are combined by conjunction.

For a given normalised schema $sch \mathrel{\widehat{=}} [D \mid P]$, the functions $decl(sch)$ and $pred(sch)$ give the declaration (D) and the predicate (P) parts of sch, respectively, as sets. The elements of $decl(sch)$ have the form $v : T_v$ and each element of $pred(sch)$ is any kind of expression allowed in the predicate part. Thus, $pred(sch)$ contains all conjuncts (propositional components of a conjunction) of the predicate part. For example,

$$
\begin{array}{|l}
\hline
Op \\
x_1 : T_x \\
x_1' : T_x \\
x_2 : T_x \\
z : T_z \\
\hline
x_1 < x_2 \wedge x_1 > z \\
x_1' = x_1 \\
\hline
\end{array}
\qquad
\begin{aligned}
decl(Op) &= \{x_1 : T_x, x_1' : T_x, \\
&\qquad x_2 : T_x, z : T_z\} \\
pred(Op) &= \{x_1 < x_2, \\
&\qquad x_1 > z, \\
&\qquad x_1' = x_1\}
\end{aligned}
$$

New declarations and predicates are inserted into $decl(Op)$ and $pred(Op)$ by using set inclusion. For example, $decl(Op) \leftarrow decl(Op) \cup \{d\}$ means the declaration d is included into $decl(Op)$.

The function *include* (Figure 11) includes a schema sch into a Z specification $(State, Init, Ops)$. The state schema ($sch = State$) is placed into the first component, the initialisation ($sch = Init$) is placed into the second one, and operations are included into Ops. We use the functions fst, snd and trd to capture the first, the second and the third components of a Z specification, respectively. Thus, $fst(Z_{spec}) = State$, $snd(Z_{spec}) = Init$ and $trd(Z_{spec}) = Ops$.

The algorithm is presented in Figure 12 and starts by considering only the Z part of the given specification (line 1). Initially, the partitions Z^{di} and Z^{dd} (line 2) are empty (without state, initialisation and operations) and each original schema sch (line 3) is analysed subsequently. For each analysed schema, the corresponding data independent (sch^{di}) and data dependent (sch^{dd}) components are initialised as empty schemas (line 4). Then the declarations of the current schema are analysed (line 5). The type of the declared variable (T_v) is taken (line 6) and used to classify all expressions of the predicate part (line 7). We point out that there may be many expressions in the predicate part for a same declared variable. Because we analyse expressions separately, it may originate non-disjoint schemas (the same declaration is placed into different components). Thus, if the analysed expression is data independent with respect to T_v (line 8), both the declaration and the expression are placed into sch^{di} (lines 9 and 10). If the analysed expression is data dependent with respect to T_v, we check (line 13) if the schemas are disjoint (that is, if the declaration has not already been placed into sch^{di}). If so, the declaration and the expression are placed into the data dependent schema sch^{dd} (lines 14 and 15). Otherwise, sch^{di} and sch^{dd} are non-disjoint and the algorithm returns an error (line 18). Independently of sch^{di} and sch^{dd} being disjoint or not, the algorithm solves the dependence between variables and expressions placed into different schemas (lines 11 and 16) by calling the function *link* (Figure 13), which is defined in a pattern matching style and uses the function $vars : Exp \rightarrow \mathbb{P}\, VarName$ to obtain the set of all variables occurring in an expression. For example, $vars(x' = x + y? * 100) = \{x', x, y?\}$.

The function *link* receives a schema (sch^{di} or sch^{dd}), a declaration (d) and an expression (e). If the declared variable is referred by e and has not been declared in the schema, a new declaration ($in? : State^{dd}$ or $in? : State^{di}$) is inserted and the substitution $[in?.v/v]$ is applied to the predicate part.

We point out that the disjointness check performed by the algorithm can also be achieved by pre-processing the

$$is_di(e,T) = \begin{cases} true, & e \text{ does not present constants of type } T \text{ and} \\ & e \text{ can present only input/output variables of} \\ & \quad \text{type } T \text{ or} \\ & e \text{ is an equality test or a polymorphic operation} \\ & \quad \text{over type } T \text{ or} \\ & e \text{ is defined in terms of equality tests and} \\ & \quad \text{polymorphic operations over type } T \\ false, & otherwise \end{cases}$$

Figure 10. The function *is_di*

$include(sch, Z_{spec}) =$

$\quad$ *if* $sch = State$ *then*
$\quad\quad Z_{spec} \leftarrow (sch, snd(Z_{spec}), trd(Z_{spec}))$
$\quad$ *else if* $sch = Init$ *then*
$\quad\quad Z_{spec} \leftarrow (fst(Z_{spec}), sch, trd(Z_{spec}))$
$\quad$ *else*
$\quad\quad Z_{spec} \leftarrow (fst(Z_{spec}), snd(Z_{spec}), trd(Z_{spec}) \cup \{sch\})$

Figure 11. The function *include*

input: a CSP_Z specification *Spec*
output: a simple bi-partition (Z^{di} and Z^{dd}) of the Z part of *Spec* or an error if the Z part does not satisfy Definition 3.4

1. **let** $(State, Init, Ops)$ be the Z part of *Spec* **in**
2. $\quad$ **let** $Z^{di} = Z^{dd} = ([\,], [\,], \varnothing)$ **in**
3. $\quad\quad \forall\, sch \in (State, Init, Ops) \bullet$
4. $\quad\quad\quad$ **let** $sch^{di} = sch^{dd} = [\,]$ **in**
5. $\quad\quad\quad\quad \forall\, d \in decl(sch) \bullet$
6. $\quad\quad\quad\quad\quad$ **let** $v : T_v = d$ **in**
7. $\quad\quad\quad\quad\quad\quad \forall\, e \in pred(sch) \bullet$
8. $\quad\quad\quad\quad\quad\quad\quad$ *if* $is_di(e, T_v)$ *then*
9. $\quad\quad\quad\quad\quad\quad\quad\quad decl(sch^{di}) \leftarrow decl(sch^{di}) \cup \{d\}$
10. $\quad\quad\quad\quad\quad\quad\quad\quad pred(sch^{di}) \leftarrow pred(sch^{di}) \cup \{e\}$
11. $\quad\quad\quad\quad\quad\quad\quad\quad link(sch^{di}, d, e)$
12. $\quad\quad\quad\quad\quad\quad\quad$ *else*
13. $\quad\quad\quad\quad\quad\quad\quad\quad$ *if* $d \notin decl(sch^{di})$ *then*
14. $\quad\quad\quad\quad\quad\quad\quad\quad\quad decl(sch^{dd}) \leftarrow decl(sch^{dd}) \cup \{d\}$
15. $\quad\quad\quad\quad\quad\quad\quad\quad\quad pred(sch^{dd}) \leftarrow pred(sch^{dd}) \cup \{e\}$
16. $\quad\quad\quad\quad\quad\quad\quad\quad\quad link(sch^{dd}, d, e)$
17. $\quad\quad\quad\quad\quad\quad\quad\quad$ *else*
18. $\quad\quad\quad\quad\quad\quad\quad\quad\quad$ return error: schemas are not disjoint
19. $\quad\quad\quad\quad\quad\quad\quad$ *if-end*
20. $\quad\quad\quad\quad\quad\quad$ $\forall$-*end*
21. $\quad\quad\quad\quad\quad$ **let**-*end*
22. $\quad\quad\quad\quad$ $\forall$-*end*
23. $\quad\quad\quad$ **let**-*end*
24. $\quad\quad\quad include(sch^{di}, Z^{di})$
25. $\quad\quad\quad include(sch^{dd}, Z^{dd})$
26. $\quad\quad$ $\forall$-*end*
27. $\quad$ **let**-*end*
28. **let**-*end*
29. return (Z^{di}, Z^{dd})

Figure 12. Partitioning algorithm

$link(sch^{di}, d, e) =$
  let $v : T_v = d$ **in**
    $if\ v \in vars(e) \wedge d \notin decl(sch^{di})\ then$
      $decl(sch^{di}) \leftarrow decl(sch^{di}) \cup \{in? : State^{dd}\}$
      $pred(sch^{di}) \leftarrow pred(sch^{di})[in?.v/v]$
  let-*end*

$link(sch^{dd}, d, e) =$
  let $v : T_v = d$ **in**
    $if\ v \in vars(e) \wedge d \notin decl(sch^{dd})\ then$
      $decl(sch^{dd}) \leftarrow decl(sch^{dd}) \cup \{in? : State^{di}\}$
      $pred(sch^{dd}) \leftarrow pred(sch^{dd})[in?.v/v]$
  let-*end*

Figure 13. The function *link*

specification. However, this requires the same loops used in the partitioning. We perform the splitting and the disjointness check in the same processing.

After processing all schemas, the sch^{di} and the sch^{dd} instances are placed into the suitable partitions (lines 24 and 25), and the algorithm considers another schema. The final result is a pair of Z specifications that form a simple bi-partition of the Z part of a CSP_Z specification (line 29).

It is worth pointing out that our algorithm always terminates when analysing a CSP_Z specification. This is a direct consequence of the finiteness of Z specifications; they have a finite number of schemas, where each one also contains a finite number of declarations and predicates. Hence, the loops of lines 3, 5 and 7 have finite iterations. The other statements involve declarations, initialisations, set inclusion, set intersection, comparisons and the functions *is_di*, *include*, *link*, *vars*, *fst*, *snd* and *trd*; they do not introduce non-termination.

5. Related Work

When abstracting systems, property preservation can be total or partial. In property-guided approaches, the abstract model depends on the properties to be verified. An example is predicate abstraction [11], which has been used in automatic verification [2, 4, 12]. In our approach, by using a refinement theory, the abstract model does not depend on the properties to be verified. Thus, more properties (safety and liveness) can be verified. Nevertheless, this makes automation much more difficult to achieve.

Compositional analysis is the focus of several works. In [20], safety and liveness properties can be compositionally verified in a network of CSP-B processes, whose components contain a control and a data part. The approach is based on the analysis of the control part of each process and on the analysis of each CSP-B component separately; it does not address any abstraction on data domains. Thus, if a CSP-B component presents state explosion, the entire network (and the component itself) cannot be directly verified. In this sense, we use compositionality differently from [20]; while that work focuses on all processes of a network, we focus on a single component. However, our approach can also be used in a network of CSP_Z processes, as long as the abstracted values are irrelevant in communications between components.

The strategy proposed in [15] combines splitting, symmetry and data type reductions (a specific kind of abstract interpretation) to deal with verification of structures of infinite size. It has been used in hardware verification and requires that the user specify refinement relations between the implementation and the abstract model. In our approach, the abstract model is equivalent (modulo renaming) to the original one by construction [16] and user interaction to define refinement relations is unnecessary.

The approach presented in [7] handles infinite communications by using the notion of IO transformers: special operations defined over inputs/outputs that map infinite domains to finite ones. Their constructions are based on an abstraction function that must be given by the user. The approach does not allow relations between state and output variables (outputs must depend only on the inputs). Moreover, abstractions can be calculated in terms of *forward* or *backward* simulations [22]. In our work, we consider only abstractions based on *forward* simulation and allow relations between state and communication variables. We also do not require user assistance to give the abstraction function explicitly; it is mechanically determined.

6. Conclusions

This work further extends a previous data abstraction approach [8, 16] to deal with CSP_Z processes that present infinite and data independent communications. Using the data independence criteria, we apply an internal partitioning to the Z part of a CSP_Z specification (Definition 3.4). This originates two components—a data independent (DI) and a data dependent (DD). Then we convert the entire specification (the CSP part and the components of the bi-partition) to CSP and use data independence and data abstraction to analyse data independent and the data dependent parts of the resulting process. This yields abstraction functions that are used to calculate the abstract domains and the abstract versions of all operations (of the DI and the DD components). As long as the abstract domains satisfy the minimum cardinality of the parallel composition of the CSP and the DI component, the abstract process is valid and can be analysed in FDR. In this sense, data independence is used to factor out the Z part and to determine the minimal bounds on the relevant data types of its data

independent components. The data dependent one is data abstracted systematically.

We showed that, even when partitioned, the Z part can be represented as a process (Theorem 3.1). Moreover, after all sub-processes (P_{CSP} conjointly with P_Z^{di} and P_Z^{dd}) have been analysed, the results can be combined in a compositional way to build the abstraction for the entire process (Theorem 3.2). The resulting abstraction is equivalent (modulo renaming) to the original specification [16].

We have also identified a particular kind of dependence between the components of the partition originated by our partitioning strategy. To solve this dependence we extended the normal form of the process representation of a Z specification (Definition 3.6) in such a way that the partitioning is still valid (Theorem 3.3). To assure this, we have proved that both process representations (according to Definitions 2.1 and 3.6) are equivalent when making their interfaces equal (Theorem 3.4).

We have proposed an algorithm to implement the partitioning strategy. The algorithm receives an original CSP_Z specification and gives a simple bi-partition of its Z part; it decomposes schemas according to the data independence property of their internal expressions.

We point out that the splitting strategy is orthogonal to the technique used to analyse the data dependent partition. Thus, our approach allows the use of different techniques to abstract domains of the resulting partition. For example, in this work we used an existing mechanical data abstraction approach [8, 16] to analyse the data dependent partition because user intervention is required only to prove internal theorems automatically generated by that strategy. However, the technique presented in [7] could be used alternatively to increase the class of problems (data dependent and infinite communications), but it would require user intervention to be applied (calculation of the abstraction functions, calculation of the IO transformers, construction of the abstract schemas, etc.).

There might also be other kinds of dependencies between the components of a bi-partition that require a more elaborate analysis. For example, data dependence of expressions in one component with respect to variables placed into the other component has not been addressed by this work. We intend to investigate techniques that allow to deal with such a class of problems in the future. This improvement will certainly lead to a more general and elegant approach to abstract data types.

Concerning mechanisation, we intend to implement our syntactic-based splitting in the tool presented in [8]. Actually, this will require a module to apply the partitioning and a module to apply data independence. The data abstraction module is already implemented in [8].

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] T. Ball and et al. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods (IFM 2004)*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.

[3] A. Cavalcanti, A. Sampaio, and J. Woodcock. Unifying classes and processes. *Software and Systems Modeling*, 40(3):277–296, 2005.

[4] E. Clarke and et al. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, 2004.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic. and Comp.*, 2(4):511–547, 1992.

[7] J. Derrick and H. Wehrheim. On using data abstractions for model checking refinements. *Acta Informatica*, 44(1):41–71, 2007.

[8] A. Farias, A. Mota, and A. Sampaio. Efficient CSP_Z data abstraction. In Erke Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods (IFM 2004)*, volume 2999 of *LNCS*, pages 108 – 127. Springer, 2004.

[9] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.

[10] M. Goldsmith. *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd, August 2001.

[11] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. LNCS, 1997.

[12] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM Press, 2002.

[13] J. Larrecq and I. Mackie. *Proof Theory and Automated Deduction*, volume 6 of *Applied Logic Series*. Kluwer Academic Publishers, May 1997.

[14] R. Lazić. *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*. PhD thesis, OUCL, 1999.

[15] K. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods*, pages 219–234, 1999.

[16] A. Mota, P. Borba, and A. Sampaio. Mechanical abstraction of CSP-Z processes. In L.H. Eriksson and P. Lindsay, editors, *Formal Methods Europe (FME'2002)*, volume 2391 of *LNCS*, pages 163–183, 2002.

[17] A. Mota and A. Sampaio. Model-checking CSP-Z: Strategy, tool support and industrial application. *SCP*, 40:59–96, 2001.

[18] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[19] H. Treharne and S. Schneider. Using a process algebra to control B operations. Technical Report CSD-TR-99-01, University of London, 1999.

[20] H. Treharne, S. Schneider, and M. Bramble. Combining specification with composition. In *ZB2003*, volume 2651, pages 58–78. LNCS, 2003.

[21] H. Wehrheim. Data abstraction for CSP-OZ. In J.Woodcock and J.Wing, editors, *FM'99 World Congress on Formal Methods*, volume 1709. LNCS, Springer, 1999.

[22] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

A. Auxiliary Lemmas

This appendix provides auxiliary laws and lemmas that are necessary to prove Theorems 3.1, 3.3 and 3.4.

Laws A.1 to A.8 are reproduced from [18]. We have proposed and proved laws A.9 to A.11.

Law A.1 states how the generalised parallel operator works. In summary, there are three options of behaviour: both processes progress together, only one of them is nondeterministically chosen to progress, or only one of them is deterministically chosen to progress.

Law A.1 *Let* $P = ?x : A \rightarrow P'$ *and* $Q = ?x : B \rightarrow Q'$ *be CSP processes. Then*

$$\begin{aligned} P \underset{X}{\parallel} Q = ?x : C \rightarrow & (P' \underset{X}{\parallel} Q') \nless x \in X \ngtr \\ & (((P' \underset{X}{\parallel} Q) \sqcap (P \underset{X}{\parallel} Q')) \nless x \in A \cap B \ngtr \\ & ((P' \underset{X}{\parallel} Q) \nless x \in A \ngtr (P \underset{X}{\parallel} Q'))) \end{aligned}$$

where $C = (X \cap A \cap B) \cup (A \backslash X) \cup (B \backslash X)$. ♢

Law A.2 states the commutativity of parallelism and Law A.3 establishes deadlock as the result of synchronising any process with *STOP*.

Law A.2 $P \underset{X}{\parallel} Q = Q \underset{X}{\parallel} P$ ♢

Law A.3 $STOP \underset{X}{\parallel} P = STOP, \ (\alpha P \subseteq X)$ ♢

Law A.4 establishes the behaviour of an external choice. It offers the initial events of both left- and right-hand-side processes. If the processes have any initial acceptance in common and the environment is ready to engage on it, the external choice behaves nondeterministically ($\sqcap$). Otherwise, the external choice engages on some initial event and behaves accordingly.

Law A.4 $$\begin{aligned} (?x : A \rightarrow P) \Box (?x : B \rightarrow Q) = \\ ?x : A \cup B \rightarrow & ((P \sqcap Q) \\ & \nless x \in A \cap B \ngtr \\ & (P \nless x \in A \ngtr Q)) \end{aligned}$$ ♢

Law A.5 defines the process *STOP* as the unit element of the external choice operator.

Law A.5 $STOP \Box P = P$ ♢

Laws A.6 and A.7 establish the semantics of the conditional choice based on the trivial values of its condition.

Law A.6 $P \nless true \ngtr Q = P$ ♢

Law A.7 $P \nless false \ngtr Q = Q$ ♢

Law A.8 establishes the way the hide operator works for processes defined by prefixing.

Law A.8 $$(a \rightarrow P) \backslash X = \begin{cases} P \backslash X & \text{if } a \in X \\ a \rightarrow (P \backslash X) & \text{if } a \notin X \end{cases}$$ ♢

Law A.9 states the use of hiding in a guarded process.

Law A.9 $(c \ \& \ ev \rightarrow P) \backslash X = c \ \& \ ev \rightarrow (P \backslash X)$ *provided* $ev \notin X$ ♢

Proof. By case analysis.

- For $\neg c$: the hiding has no effect ($STOP \backslash X = STOP$).
- For c:
  $$\begin{aligned} & (ev \rightarrow P) \backslash X \\ = & \ ev \rightarrow (P \backslash X) && \textit{(by Law A.8)} \\ = & \ c \ \& \ ev \rightarrow (P \backslash X) && \textit{(because c)} \end{aligned}$$

■

Law A.10 states the idempotence of generalised parallelism, as also informally addressed in [18].

Law A.10 *Let P be a deterministic CSP process. Let X be a set of events such that $\alpha P \subseteq X$. Then,*

$$P \underset{X}{\parallel} P = P \qquad \diamondsuit$$

Proof. By case analysis where P is deterministic:

- $P = STOP$: trivial.
- $P = SKIP$: trivial.
- P is an arbitrary deterministic process. From [18], we can write P as $?x : initials(P) \rightarrow P'$. By induction, the law is valid for a context P' (hypothesis) and we must prove for the next context $?x : initials(P) \rightarrow P'$ (thesis):

$$\begin{aligned}
&(?x : initials(P) \rightarrow P') \underset{X}{\parallel} (?x : initials(P) \rightarrow P') \\
&= ?x : initials(P) \rightarrow (P' \underset{X}{\parallel} P') && \textit{(by Law A.1)} \\
&= ?x : initials(P) \rightarrow P' && \textit{(by hypothesis)} \\
&= P && \textit{(by definition of P)}
\end{aligned}$$

■

Prefixing a conditional choice is similar to prefixing each branch of the conditional choice (Law A.11).

Law A.11 $a \rightarrow (P \nless b \ngtr Q) = (a \rightarrow P) \nless b \ngtr (a \rightarrow Q)$ $\diamondsuit$

Proof. By case analysis.

- For b:

$$\begin{aligned}
&a \rightarrow (P \nless true \ngtr Q) \\
&= a \rightarrow P && \textit{(by Law A.6)} \\
&= (a \rightarrow P) \nless true \ngtr (a \rightarrow Q) && \textit{(by Law A.6)} \\
&= (a \rightarrow P) \nless b \ngtr (a \rightarrow Q) && \textit{(because b)}
\end{aligned}$$

- For $\neg\, b$:

$$\begin{aligned}
&a \rightarrow (P \nless false \ngtr Q) \\
&= a \rightarrow Q && \textit{(by Law A.7)} \\
&= (a \rightarrow P) \nless false \ngtr (a \rightarrow Q) && \textit{(by Law A.7)} \\
&= (a \rightarrow P) \nless b \ngtr (a \rightarrow Q) && \textit{(because } \neg\, b\textit{)}
\end{aligned}$$

■

In the following we present some useful lemmas. Lemma A.1 allows one to represent a guarded process by using parallelism of the same process with weaker guards.

Lemma A.1 *Let a and b be conditionals and ev be a CSP event. Let P, P_a and P_b be the processes given by*

$$\begin{aligned}
P &= (a \wedge b) \;\&\; ev \rightarrow P \\
P_a &= a \;\&\; ev \rightarrow P_a \\
P_b &= b \;\&\; ev \rightarrow P_b
\end{aligned}$$

Then, $P = P_a \underset{\alpha P}{\parallel} P_b$ $\diamondsuit$

Proof. By case analysis on the conditionals. We call $P_a \underset{\alpha P}{\parallel} P_b$ by Pab and use Law A.1 to show that $P_{ab} = ((a \wedge b) \;\&\; ev \rightarrow P)[P_{ab}/P]$.

- For $a \wedge b$:

$$\begin{aligned}
&P_{ab} \\
&= P_a \underset{\alpha P}{\parallel} P_b && \textit{(by definition of } P_{ab}\textit{)} \\
&= a \;\&\; ev \rightarrow P_a \underset{\alpha P}{\parallel} b \;\&\; ev \rightarrow P_b && \textit{(by definition of } P_a \textit{ and } P_b\textit{)} \\
&= ev \rightarrow P_a \underset{\alpha P}{\parallel} ev \rightarrow P_b && \textit{(because } a \wedge b\textit{)} \\
&= ev \rightarrow (P_a \underset{\alpha P}{\parallel} P_b) && \textit{(by Law A.1)} \\
&= ev \rightarrow P_{ab} && \textit{(by definition of } P_{ab}\textit{)} \\
&= (a \wedge b) \;\&\; ev \rightarrow P_{ab} && \textit{(because } a \wedge b\textit{)}
\end{aligned}$$

Thus, $(a \wedge b) \;\&\; ev \rightarrow P_{ab} = (a \wedge b) \;\&\; ev \rightarrow P[P_{ab}/P]$.

- For $a \wedge \neg b$:

$$\begin{aligned}
&P_{ab} \\
&= P_a \underset{\alpha P}{\parallel} P_b && \textit{(by definition of } P_{ab}\textit{)} \\
&= a \;\&\; ev \rightarrow P_a \underset{\alpha P}{\parallel} b \;\&\; ev \rightarrow P_b && \textit{(by definition of } P_a \textit{ and } P_b\textit{)} \\
&= ev \rightarrow P_a \underset{\alpha P}{\parallel} STOP && \textit{(because } a \wedge \neg\, b\textit{)} \\
&= STOP && \textit{(by Law A.3)} \\
&= (a \wedge b) \;\&\; ev \rightarrow P_{ab} && \textit{(because } a \wedge \neg\, b\textit{)}
\end{aligned}$$

Thus, $(a \wedge b) \;\&\; ev \rightarrow P_{ab} = (a \wedge b) \;\&\; ev \rightarrow P[P_{ab}/P]$.

- For $\neg a \wedge b$: it is similar to $a \wedge \neg b$.
- For $\neg a \wedge \neg\, b$:

$$\begin{aligned}
&P_{ab} \\
&= P_a \underset{\alpha P}{\parallel} P_b && \textit{(by definition of } P_{ab}\textit{)} \\
&= a \;\&\; ev \rightarrow P_a \underset{\alpha P}{\parallel} b \;\&\; ev \rightarrow P_b && \textit{(by definition of } P_a \textit{ and } P_b\textit{)} \\
&= STOP \underset{\alpha P}{\parallel} STOP && \textit{(because } \neg\, a \wedge \neg\, b\textit{)} \\
&= STOP && \textit{(by Law A.3)} \\
&= (a \wedge b) \;\&\; ev \rightarrow P_{ab} && \textit{(because } \neg\, a \wedge \neg\, b\textit{)}
\end{aligned}$$

Thus, $(a \wedge b) \;\&\; ev \rightarrow P_{ab} = (a \wedge b) \;\&\; ev \rightarrow P[P_{ab}/P]$.

■

The external choice of processes without common initial acceptances can be represented as a conditional. If the environment is ready to engage into an event offered by the external choice, only one of the component processes will progress. This is precisely stated by Lemma A.2.

Lemma A.2 *Let $P = ?a : initials(P) \rightarrow P'$ and $Q = ?c : initials(Q) \rightarrow Q'$ be deterministic CSP processes such that $initials(P) \cap initials(Q) = \varnothing$. Let x be an event from $initials(P) \cup initials(Q)$. Then,*

$$P \,\Box\, Q = P \nless x \in initials(P) \ngtr Q \qquad \diamondsuit$$

Proof. Direct consequence of Law A.4, where $A = initials(P)$ and $B = initials(Q)$. We consider $I = initials(P) \cup initials(Q)$.

$$
\begin{aligned}
& P \,\Box\, Q \\
=& ?x : I \rightarrow \quad ((P' \sqcap Q') \\
& \qquad \nless x \in initials(P) \cap initials(Q) \ngtr \\
& \qquad (P' \nless x \in initials(P) \ngtr Q')) && \textit{(by Law A.4)} \\
=& ?x : I \rightarrow (P' \nless x \in initials(P) \ngtr Q') \\
& && \textit{(because } initials(P) \cap initials(Q) = \varnothing\textit{)} \\
=& (?x : I \rightarrow P') \nless x \in initials(P) \ngtr (?x : I \rightarrow Q') \\
& && \textit{(by Law A.11)} \\
=& ?x_p : initials(P) \rightarrow P' \nless x \in initials(P) \ngtr ?x_q : initials(Q) \rightarrow Q' \\
& && \textit{(because } initials(P) \cap initials(Q) = \varnothing\textit{)} \\
=& P \nless x \in initials(P) \ngtr Q && \textit{(by definition of P and Q)}
\end{aligned}
$$

■

Lemma A.3 states the distribution of $\Box$ over $\underset{X}{\|}$.

Lemma A.3 *Let P, Q and R be deterministic CSP processes such that $\alpha Q = \alpha R$, $initials(Q) = initials(R)$ and $initials(P) \cap initials(Q) = initials(P) \cap initials(R) = \varnothing$. Then,*

$$P \,\Box\, (Q \underset{I}{\|} R) = (P \,\Box\, Q) \underset{\alpha P \,\cup\, \alpha Q}{\|} (P \,\Box\, R) \qquad \diamondsuit$$

Proof. From Law A.1 we calculate $initials(P) \cup initials(Q)$ as the initial acceptances of $(P \,\Box\, Q) \underset{\alpha P \,\cup\, \alpha Q}{\|} (P \,\Box\, R)$. As $initials(P) \cap initials(Q) = \varnothing$, we have two options to analyse:

- For $x \in initials(P)$: by Law A.1, Q and R become unavailable and $(P \,\Box\, Q) \underset{\alpha P \,\cup\, \alpha Q}{\|} (P \,\Box\, R)$ behaves like

  $$
  \begin{aligned}
  & P \underset{\alpha P \cup \alpha Q}{\|} P \\
  =\ & P && \textit{(by Law A.10)}
  \end{aligned}
  $$

- For $x \notin initials(P)$: by Law A.1, P becomes unavailable and $(P \,\Box\, Q) \underset{\alpha P \,\cup\, \alpha Q}{\|} (P \,\Box\, R)$ behaves like

  $$
  \begin{aligned}
  & Q \underset{\alpha P \cup \alpha Q}{\|} R \\
  =\ & Q \underset{\alpha Q}{\|} R && \textit{(because events outside } \alpha Q \textit{ are irrelevant)}
  \end{aligned}
  $$

From the above case analysis we conclude that

$$
\begin{aligned}
& P \nless x \in initials(P) \ngtr Q \underset{\alpha Q}{\|} R \\
=\ & P \,\Box\, (Q \underset{\alpha Q}{\|} R) && \textit{(by Lemma A.2)}
\end{aligned}
$$

■

When two external choices involving guarded processes are put into parallel, the guards are interchangeable. Lemma A.4 states this.

Lemma A.4 *Let P, Q, R_1 and R_2 be deterministic processes such that $\alpha P = \alpha Q$, $\alpha R_1 = \alpha R_2$, $\alpha P \cap \alpha R_1 = \varnothing$ and $initials(P) \cap initials(R_1) = initials(Q) \cap initials(R_2) = \varnothing$. Let c_1, c_2 be conditionals. Then,*

$$
\begin{aligned}
& (P \,\Box\, c_1 \,\&\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, c_2 \,\&\, R_2) \\
& \qquad = \\
& (P \,\Box\, c_2 \,\&\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, c_1 \,\&\, R_2)
\end{aligned}
\qquad \diamondsuit
$$

Proof. By case analysis on the conditionals. Moreover, because $\alpha P = \alpha Q$, $\alpha R_1 = \alpha R_2$ and $\alpha P \cap \alpha R_1 = \varnothing$, we have that $\alpha Q \cap \alpha R_1 = \alpha P \cap \alpha R_2 = \varnothing$.

- For $c_1 \wedge c_2$:

  $$
  \begin{aligned}
  & (P \,\Box\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, R_2) \\
  =\ & (P \,\Box\, c_2 \,\&\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, c_1 \,\&\, R_2) \\
  & && \textit{(because } c_1 \wedge c_2\textit{)}
  \end{aligned}
  $$

- For $\neg\, c_1 \wedge \neg\, c_2$:

  $$
  \begin{aligned}
  & (P \,\Box\, STOP) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, STOP) \\
  =\ & (P \,\Box\, c_2 \,\&\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, c_1 \,\&\, R_2) \\
  & && \textit{(because } \neg\, c_1 \wedge \neg\, c_2\textit{)}
  \end{aligned}
  $$

- For c_1 and $\neg\, c_2$:

  $$
  \begin{aligned}
  & (P \,\Box\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} (Q \,\Box\, STOP) \\
  =\ & (P \,\Box\, R_1) \underset{\alpha P \,\cup\, \alpha R_1}{\|} Q && \textit{(by Law A.5)} \\
  =\ & P \underset{\alpha P \,\cup\, \alpha R_1}{\|} Q \\
  & \quad \textit{(because } initials(P) \cap initials(R_1) = \alpha Q \cap \alpha R_1 = \varnothing\textit{)} \\
  =\ & P \underset{\alpha P \,\cup\, \alpha R_1}{\|} Q \,\Box\, R_2 \\
  & \quad \textit{(because } initials(Q) \cap initials(R_2) = \alpha P \cap \alpha R_2 = \varnothing\textit{)} \\
  =\ & P \,\Box\, STOP \underset{\alpha P \,\cup\, \alpha R_1}{\|} Q \,\Box\, R_2 && \textit{(by Law A.5)} \\
  =\ & P \,\Box\, c_2 \,\&\, R_1 \underset{\alpha P \,\cup\, \alpha R_1}{\|} Q \,\Box\, c_1 \,\&\, R_2 \\
  & && \textit{(because } c_1 \wedge \neg\, c_2\textit{)}
  \end{aligned}
  $$

- For $\neg\, c_1$ and c_2: similar to c_1 and $\neg\, c_2$.

■

Recall from Lemma A.1 that a guarded process can be written as a parallelism of the same process with weaker guards. Analogously, the external choice of guarded processes can also be expressed as a parallelism of external choices. In this sense, Lemma A.5 extends Lemma A.1.

Lemma A.5 *Let P, P_a and P_b be CSP processes given by*

$$
\begin{aligned}
P &= \Box_i \bullet (a_i \wedge b_i) \,\&\, ev_i \rightarrow P \\
P_a &= \Box_i \bullet a_i \,\&\, ev_i \rightarrow P_a \\
P_b &= \Box_i \bullet b_i \,\&\, ev_i \rightarrow P_b
\end{aligned}
$$

where a_i and b_i are conditionals and ev_i is an event. Then,

$$P = P_a \underset{\alpha P}{\|} P_b \qquad \diamondsuit$$

Proof. By induction on the index of the external choice.

Base Case: $i = 1$. Guaranteed by Lemma A.1.

Inductive Case. The lemma is valid for $i = n$ (hypothesis) and we prove for $i = n + 1$ (thesis). We rewrite the process P when $i = n + 1$ to use the hypothesis. Thus,

$$(\Box_{i=n} \bullet (a_i \wedge b_i) \,\&\, ev_i \rightarrow P) \,\Box\, (a_{n+1} \wedge b_{n+1}) \,\&\, ev_{n+1} \rightarrow P$$

$$= (\Box_{i=n} \bullet (a_i \wedge b_i) \,\&\, ev_i \rightarrow P) \,\Box\, (a_{n+1} \& ev_{n+1} \rightarrow P_a \underset{\alpha P}{\|} b_{n+1} \& ev_{n+1} \rightarrow P_b) \quad \textit{(by Lemma A.1)}$$

$$= ((\Box_{i=n} \bullet a_i \,\&\, ev_i \rightarrow P_a) \underset{\alpha P}{\|} (\Box_{i=n} \bullet b_i \,\&\, ev_i \rightarrow P_b)) \,\Box\, (a_{n+1} \& ev_{n+1} \rightarrow P_a \underset{\alpha P}{\|} b_{n+1} \& ev_{n+1} \rightarrow P_b) \quad \textit{(by hypothesis)}$$

$$\begin{aligned} = \quad & (\Box_{i=n} \bullet a_i \,\&\, ev_i \rightarrow P_a) \,\Box\, a_{n+1} \,\&\, ev_{n+1} \rightarrow P_a \\ \underset{\alpha P}{\|} \; & (\Box_{i=n} \bullet b_i \,\&\, ev_i \rightarrow P_b) \,\Box\, a_{n+1} \,\&\, ev_{n+1} \rightarrow P_a \\ \underset{\alpha P}{\|} \; & (\Box_{i=n} \bullet a_i \,\&\, ev_i \rightarrow P_a) \,\Box\, b_{n+1} \,\&\, ev_{n+1} \rightarrow P_b \\ \underset{\alpha P}{\|} \; & (\Box_{i=n} \bullet b_i \,\&\, ev_i \rightarrow P_b) \,\Box\, b_{n+1} \,\&\, ev_{n+1} \rightarrow P_b \end{aligned}$$

(by applying Lemma A.3 twice)

$$\begin{aligned} = \quad & (\Box_{i=n} \bullet a_i \,\&\, ev_i \rightarrow P_a) \,\Box\, a_{n+1} \,\&\, ev_{n+1} \rightarrow P_a \\ \underset{\alpha P}{\|} \; & (\Box_{i=n} \bullet b_i \,\&\, ev_i \rightarrow P_b) \,\Box\, b_{n+1} \,\&\, ev_{n+1} \rightarrow P_a \\ \underset{\alpha P}{\|} \; & (\Box_{i=n} \bullet a_i \,\&\, ev_i \rightarrow P_a) \,\Box\, a_{n+1} \,\&\, ev_{n+1} \rightarrow P_b \\ \underset{\alpha P}{\|} \; & (\Box_{i=n} \bullet b_i \,\&\, ev_i \rightarrow P_b) \,\Box\, b_{n+1} \,\&\, ev_{n+1} \rightarrow P_b \end{aligned}$$

(by Lemma A.4)

$$\begin{aligned} = \quad & (\Box_{i=n+1} \bullet a_i \,\&\, ev_i \rightarrow P_a) \\ \underset{\alpha P}{\|} \; & (\Box_{i=n+1} \bullet b_i \,\&\, ev_i \rightarrow P_b) \\ \underset{\alpha P}{\|} \; & (\Box_{i=n+1} \bullet b_i \,\&\, ev_i \rightarrow P_b) \\ \underset{\alpha P}{\|} \; & (\Box_{i=n+1} \bullet a_i \,\&\, ev_i \rightarrow P_a) \end{aligned} \quad \textit{(by grouping } \Box_i\textit{)}$$

$$\begin{aligned} = \quad & (\Box_{i=n+1} \bullet a_i \,\&\, ev_i \rightarrow P_a) \\ \underset{\alpha P}{\|} \; & (\Box_{i=n+1} \bullet a_i \,\&\, ev_i \rightarrow P_a) \\ \underset{\alpha P}{\|} \; & (\Box_{i=n+1} \bullet b_i \,\&\, ev_i \rightarrow P_b) \\ \underset{\alpha P}{\|} \; & (\Box_{i=n+1} \bullet b_i \,\&\, ev_i \rightarrow P_b) \end{aligned} \quad \textit{(by Law A.2)}$$

$$= (\Box_{i=n+1} \bullet a_i \,\&\, ev_i \rightarrow P_a) \underset{\alpha P}{\|} (\Box_{i=n+1} \bullet b_i \,\&\, ev_i \rightarrow P_b) \quad \textit{(by Law A.10)}$$

$$= P_a \underset{\alpha P}{\|} P_b \quad \textit{(by definition of } P_a \textit{ and } P_b\textit{)}$$

■

Lemma A.6 allows one to distribute hide over an indexed external choice, as long as the hidden events are not initially accepted by the options.

Lemma A.6 *Let P be a CSP process and X a set of events. Let c_i be a conditional and ev_i an event such that $ev_i \notin X$. Then,*

$$(\Box_i \bullet (c_i \,\&\, ev_i \rightarrow P))\backslash X = \Box_i \bullet (c_i \,\&\, ev_i \rightarrow P\backslash X) \quad \diamondsuit$$

Proof. By induction on the index of the external choice.

Base Case: $i = 1$. Guaranteed by Law A.8.

Inductive Case. The lemma is valid for $i = n$ (hypothesis) and we prove for $i = n+1$ (thesis). By rewriting $(\Box_{i=n+1} \bullet (c_i \,\&\, ev_i \rightarrow P))\backslash X$ to use the hypothesis, we have

$$((\Box_{i=n} \bullet c_i \,\&\, ev_i \rightarrow P) \,\Box\, c_{n+1} \,\&\, ev_{n+1} \rightarrow P)\backslash X$$

$$= (\Box_{i=n} \bullet c_i \,\&\, ev_i \rightarrow P)\backslash X \,\Box\, (c_{n+1} \,\&\, ev_{n+1} \rightarrow P)\backslash X \quad \textit{(because } ev_i \textit{ and } ev_{n+1} \notin X\textit{)}$$

$$= (\Box_{i=n} \bullet c_i \,\&\, ev_i \rightarrow P\backslash X) \,\Box\, (c_{n+1} \,\&\, ev_{n+1} \rightarrow P)\backslash X \quad \textit{(by hypothesis)}$$

$$= (\Box_{i=n} \bullet c_i \,\&\, ev_i \rightarrow P\backslash X) \,\Box\, (c_{n+1} \,\&\, ev_{n+1} \rightarrow P\backslash X) \quad \textit{(by Law A.9)}$$

$$= \Box_{i=n+1} \bullet c_i \,\&\, ev_i \rightarrow P\backslash X \quad \textit{(by grouping } \Box_i\textit{)}$$

■

Now we present the proof of Theorem 3.1. It is related to the process representation of a simple bi-partition. We point out that, because the component specifications of a simple bi-partition are disjoint, the preconditions of each schema and the state are also disjoint. As the theorem is related to behaviour, the state information is irrelevant in the proof and, therefore, can be omitted.

Theorem 3.1 *Let $Z_{spec} = (State, Init, Ops)$, $Z^{di}_{spec} = (State^{di}, Init^{di}, Ops^{di})$ and $Z^{dd}_{spec} = (State^{dd}, Init^{dd}, Ops^{dd})$ be Z specifications such that Z^{di}_{spec} and Z^{dd}_{spec} form a simple bi-partition of Z_{spec}. Let P_Z, P^{di}_Z and P^{dd}_Z be CSP processes capturing the behaviour of Z_{spec}, Z^{di}_{spec} and Z^{dd}_{spec}, respectively. Then,*

$$P_Z(State) = P^{di}_Z(State^{di}) \underset{\alpha P_Z}{\|} P^{dd}_Z(State^{dd}) \quad \diamondsuit$$

Proof. From Definition 3.4 we know that P_Z, P^{di}_Z and P^{dd}_Z have the same set of operations and the same alphabets. Thus,

$$\begin{aligned} & P_Z \\ = \; & \Box_{com_ev \in Ops} \bullet \text{pre } com_ev \,\&\, ev \rightarrow P_Z \quad \textit{(by Definition 2.1)} \\ = \; & \Box_{com_ev \in Ops} \bullet (\text{pre } com_ev^{di} \wedge \text{pre } com_ev^{dd}) \,\&\, ev \rightarrow P_Z \quad \textit{(by Definition 3.4)} \\ = \; & \Box_{com_ev \in Ops} \bullet (\text{pre } com_ev^{di} \,\&\, ev \rightarrow P^{di}_Z \underset{\alpha P_Z}{\|} \text{pre } com_ev^{dd} \,\&\, ev \rightarrow P^{dd}_Z) \quad \textit{(by Lemma A.1)} \\ = \; & (\Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{di} \,\&\, ev \rightarrow P^{di}_Z) \underset{\alpha P_Z}{\|} (\Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{dd} \,\&\, ev \rightarrow P^{dd}_Z) \quad \textit{(by Lemma A.5)} \\ = \; & P^{di}_Z \underset{\alpha P_Z}{\|} P^{dd}_Z \quad \textit{(by Definition 2.1)} \end{aligned}$$

■

In the following we show the proof of Theorem 3.3, which is related to the compound form of P_Z using the extended normal form (Definition 3.6).

Theorem 3.3. *Let $Z_{spec} = (State, Init, Ops)$, $Z^{di}_{spec} = (State^{di}, Init^{di}, Ops^{di})$ and $Z^{dd}_{spec} = (State^{dd}, Init^{dd}, Ops^{dd})$ be Z specifications such that Z^{di}_{spec} and Z^{dd}_{spec} form a simple bi-partition of Z_{spec}. Let $P_{Z_{ext}}, P^{di}_{Z_{ext}}$ and $P^{dd}_{Z_{ext}}$ be CSP processes in the extended normal form capturing the behaviour of Z_{spec}, Z^{di}_{spec} and Z^{dd}_{spec}, respectively. Then,*

$$P_{Z_{ext}}(State) = P^{di}_{Z_{ext}}(State^{di}) \underset{\alpha P_{Z_{ext}}}{\|} P^{dd}_{Z_{ext}}(State^{dd}) \quad \diamondsuit$$

Proof. It starts by considering $P_{Z_{ext}}(State)$.

$$\begin{aligned} & P_{Z_{ext}}(State) \\ = \; & communicate.State^{di}.State^{dd} \rightarrow P_Z(State)[P_{Z_{ext}}/P_Z] \quad \textit{(by Definition 3.6)} \\ = \; & communicate.State^{di}.State^{dd} \rightarrow (P^{di}_Z(State^{di}) \underset{\alpha P_Z}{\|} P^{dd}_Z(State^{dd})[P_{Z_{ext}}/P_Z] \quad \textit{(by Theorem 3.1)} \\ = \; & communicate.State^{di}.State^{dd} \rightarrow \end{aligned}$$

$$
\begin{array}{l}
\quad (\ \Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{di} \ \&\ ev \rightarrow P_Z^{di}(com_ev^{di}(State^{di})) \\
\qquad \underset{\alpha P_Z}{||} \\
\qquad \Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{dd} \ \&\ ev \rightarrow P_Z^{dd}(com_ev^{dd}(State^{dd})) \\
\quad)[P_{Z_{ext}}/P_Z] \qquad\qquad \textit{(by Definition 2.1)} \\
= communicate.State^{di}.State^{dd} \rightarrow \\
\quad (\ \Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{di} \ \&\ ev \rightarrow P_{Z_{ext}}^{di}(com_ev^{di}(State^{di})) \\
\qquad \underset{\alpha P_{Z_{ext}}}{||} \\
\qquad \Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{dd} \& ev \rightarrow P_{Z_{ext}}^{dd}(com_ev^{dd}(State^{dd}))) \\
\qquad\qquad \textit{(by replacing } P_Z \textit{ with } P_{Z_{ext}}\textit{)} \\
= communicate.State^{di}.State^{dd} \rightarrow \\
\qquad \Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{di} \ \&\ ev \rightarrow P_{Z_{ext}}^{di}(com_ev^{di}(State^{di})) \\
\quad \underset{\alpha P_{Z_{ext}}}{||} \\
\quad communicate.State^{di}.State^{dd} \rightarrow \\
\qquad \Box_{com_ev \in Ops} \bullet \text{pre } com_ev^{dd} \& ev \rightarrow P_{Z_{ext}}^{dd}(com_ev^{dd}(State^{dd})) \\
\qquad\qquad \textit{(by Law A.1)} \\
= P_{Z_{ext}}^{di}(State^{di}))\ \underset{\alpha P_{Z_{ext}}}{||}\ P_{Z_{ext}}^{dd}(State^{dd})) \qquad \textit{(by Definition 3.6)}
\end{array}
$$

■

In the following we present the proof of Theorem 3.4.

Theorem 3.4. *Let $P_{Z_{ext}}$ and P_Z be processes representations for the Z specification $(State, Init, Ops)$ such that P_Z in the normal form and $P_{Z_{ext}}$ is in the extended normal form. Then,*

$$P_{Z_{ext}} \backslash \{| \ communicate \ |\} = P_Z \qquad \diamond$$

Proof. From Definition 2.1 we know that $\alpha P_Z = \alpha P_Z^{di} = \alpha P_Z^{dd}$ and from Definition 3.6 we know that $\alpha P_{Z_{ext}} = \alpha P_{Z_{ext}}^{di} = \alpha P_{Z_{ext}}^{dd} = \alpha P_Z^{di} \cup \{| \ communicate \ |\}$. We represent $\{| \ communicate \ |\}$ by X and base the proof on syntactical equality, which means process equivalence [18].

$$
\begin{array}{l}
\quad P_{Z_{ext}}(s) \backslash X \\
= (communicate.s^{di}.s^{dd} \rightarrow P_Z(s)[P_{Z_{ext}}/P_Z]) \backslash X \\
\qquad\qquad \textit{(by Definition 3.6)} \\
= (communicate.s^{di}.s^{dd} \rightarrow \Box_{com_ev \in Ops} \bullet \text{pre } com_ev \ \& \\
\qquad ev \rightarrow P_Z(s'))[P_{Z_{ext}}/P_Z] \backslash X \qquad \textit{(by Definition 2.1)} \\
= (communicate.s^{di}.s^{dd} \rightarrow \Box_{com_ev \in Ops} \bullet \text{pre } com_ev \ \& \\
\qquad ev \rightarrow P_{Z_{ext}}(s')) \backslash X \qquad \textit{(by replacing } P_Z \textit{ with } P_{Z_{ext}}\textit{)} \\
= (\Box_{com_ev \in Ops} \bullet \text{pre } com_ev \ \&\ ev \rightarrow P_{Z_{ext}}(s')) \backslash X \\
\qquad\qquad \textit{(by Law A.8)} \\
= \Box_{com_ev \in Ops} \bullet \text{pre } com_ev \ \&\ ev \rightarrow P_{Z_{ext}}(s') \backslash X \\
\qquad\qquad \textit{(by Lemma A.6)}
\end{array}
$$

Note that $\Box_{com_ev \in Ops} \bullet \text{pre } com_ev \ \&\ ev \rightarrow P_{Z_{ext}}(s') \backslash X$ can also be given by $\Box_{ev \in chs} \bullet \text{pre } com_ev \ \&\ ev \rightarrow P_Z(s')[P_{Z_{ext}} \backslash X / P_Z]$. This simple syntactic substitution establishes the equality (and equivalence) between $P_{Z_{ext}} \backslash X$ and P_Z. ■