Refining Industrial Scale Systems in Circus

Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, England

Abstract. *Circus* is a new notation that may be used to specify both data and behaviour aspects of a system, and has an associated refinement calculus. Although a few case studies are already available in the literature, the industrial fire control system presented in this paper is, as far as we know, the largest case study on the *Circus* refinement strategy. We describe the refinement and present some new laws that were needed. Our case study makes extensive use of mutual recursion; a simplified notation for specifying such systems and proving their refinements is proposed here.

1 Introduction

Circus (Concurrent Integrated Refinement CalculUS) [1, 2] characterises systems as processes that combine constructs that describe data and control behaviour. The Z notation [3, 4] is used to define most of the data aspects, and CSP [5] and Dijkstra's guarded-command language are used to define behaviour. The semantics of *Circus* is based on unifying theories of programming [6], a framework that unifies the science of programming across many different computational paradigms. *Circus*, unlike other combinations of data and behavioural aspects, such as CCS-Z [7, 8], CSP-Z [9], and CSP-OZ [10], supports refinement in a calculational style similar to that presented in [11].

A refinement strategy for *Circus* is presented in [2], with the complete development of a reactive buffer into a distributed implementation as an example. Refinement notions and many refinement laws are also presented. In the current paper, we provide a more significant case study on the *Circus* refinement calculus: a safety-critical fire protection system. As far as we know, it is the largest case study on the *Circus* refinement calculus.

Throughout the development of our case study there were some problems; we present the solutions for some of them in this paper. First, the set of laws presented in [2] was not sufficient; we propose new refinement laws. For instance, we require some laws for inserting and distributing assumptions, and a new process refinement law. In total, more than fifty new laws have been identified during the development of our case study.

In [2], the refinement of mutual recursive actions is not considered; our case study, however, includes mutually recursive definitions. We present here a notation used to prove refinement of such systems; this results in more concise and modular proofs. The necessary theorems that justify the notation have been proved in [12].

The main objective of this paper is to illustrate an application of the refinement strategy in an existing industrial application [13]. We believe that, with the results in this paper, we provide empirical evidence of the power of expression of *Circus* and, principally, that the strategy presented in [2] is applicable to large industrial systems.

In Section 2, we present an introduction to refinement in *Circus*: we describe *Circus* and the refinement notions for processes and their constituent actions. Section 3 presents our case study. Finally, we present our conclusions and discuss future work in Section 4.

2 Refinement in *Circus*

In what follows, we summarise the *Circus* notation and its refinement technique. More details can be found in [1, 2], and an example is presented in Section 3.

2.1 *Circus*

Circus programs are sequences of paragraphs: channel declarations, channel set definitions, Z paragraphs, or process definitions. A system is defined as a process that encapsulates some state and communicates through channels.

A channel declaration declares its name and type; if the channel is used purely for synchronisation, then no type is needed. The generic channel declaration **channel** $[T]\ c : T$ declares a family of channels c. In this declaration, $[T]$ is a parameter used to determine the type of the values that are communicated through channel c. We may introduce sets of channels in a **chanset** paragraph.

Processes may be defined explicitly or in terms of other processes (compound processes). An explicit process definition is delimited by the keywords **begin** and **end**: it is formed by a state definition, a sequence of paragraphs, and a nameless action, which defines its behaviour. In [2], we have introduced the keyword **state** before the state declaration in order to make it clear which schema represents a process state.

Compound processes are defined using the CSP operators of sequence, external (`occam ALT`) and internal choice, parallelism and interleaving, or their corresponding iterated operators, event hiding, or indexed operators, which are particular to *Circus* specifications. The parallelism follows the alphabetised approach adopted by [5], instead of that adopted by [14].

An action can be a schema, a guarded command, an invocation of another action, or a combination of these constructs using CSP operators. Three primitive actions are available: *Skip*, *Stop*, and *Chaos*. The prefixing operator is standard, but a guard construction may be associated with it. For instance, given a Z predicate p, if the condition p is $true$, the action $p \mathbin{\&} c?x \rightarrow A$ inputs a value through channel c and assigns it to the variable x, and then behaves like A, which has the variable x in scope. If, however, the condition p is *false*, the same action blocks. Such enabling conditions like p may be associated with any action.

The CSP operators of sequence, external and internal choice, parallelism, interleaving, their corresponding iterated operators, and hiding may also be used to compose actions. Communications and recursive definitions are also available.

To avoid conflicts in the access to the variables in scope, parallelism and interleaving of actions declare a synchronisation channel set and two sets that partition all the variables. In the parallelism $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the actions A_1 and A_2 synchronise on the channels in set cs, unlike occam, where we cannot determine the synchronisation channel set; both A_1 and A_2 have access to the initial values of all variables in both ns_1 and ns_2. However, A_1 and A_2 may modify only the values of the variables in ns_1 and ns_2, respectively. The changes made by A_1 in variables in ns_1 are not seen by A_2, and *vice-versa*.

Finally, an action may also be a variable block. Further operators are available in *Circus* [1]; only those that are used in this paper are described here.

2.2 Refinement Strategy

A refinement strategy for *Circus* is presented [2]. It is based on laws of simulation, a technique used to prove data refinement in Z, and action and process refinement; some of them are presented in Appendix A. We present further simulation and refinement laws in Appendix B.

The strategy aims at refining an abstract centralised specification to a distributed *Circus* program, which involves only executable constructs. The strategy consists of possibly many iterations involving simulation, actions, and process refinement; in each iteration a process is split as presented in Figure 1. In this figure, each process is represented as a box. For instance, before the simulation, we have a process with an internal state Sa, and actions $ActA1, \cdots, ActAk$; its behaviour is determined by the main action $ActA$. First, elements of the concrete system state are included using simulation; next, the state space and actions are partitioned in such a way that each partition, represented in the figure by internal boxes, groups some state components and the actions which access these components; and, finally, all these partitions become individual processes, which are combined in the same way as their main actions were in the previous process.

The semantics of *Circus* is defined using Hoare and He's unifying theories of programming. In [2], we have a definition for action refinement; process refinement amounts to refinement of the main action, with the state components taken as local variables. Backwards and forwards simulation are also defined and proved sound in [2]. Here, we do not use the definitions in [2], but simulation and refinement laws.

3 Case Study

Our case study consists of a fire control system that covers two separate areas. Each area is divided into two zones; two different zones cannot be covered by

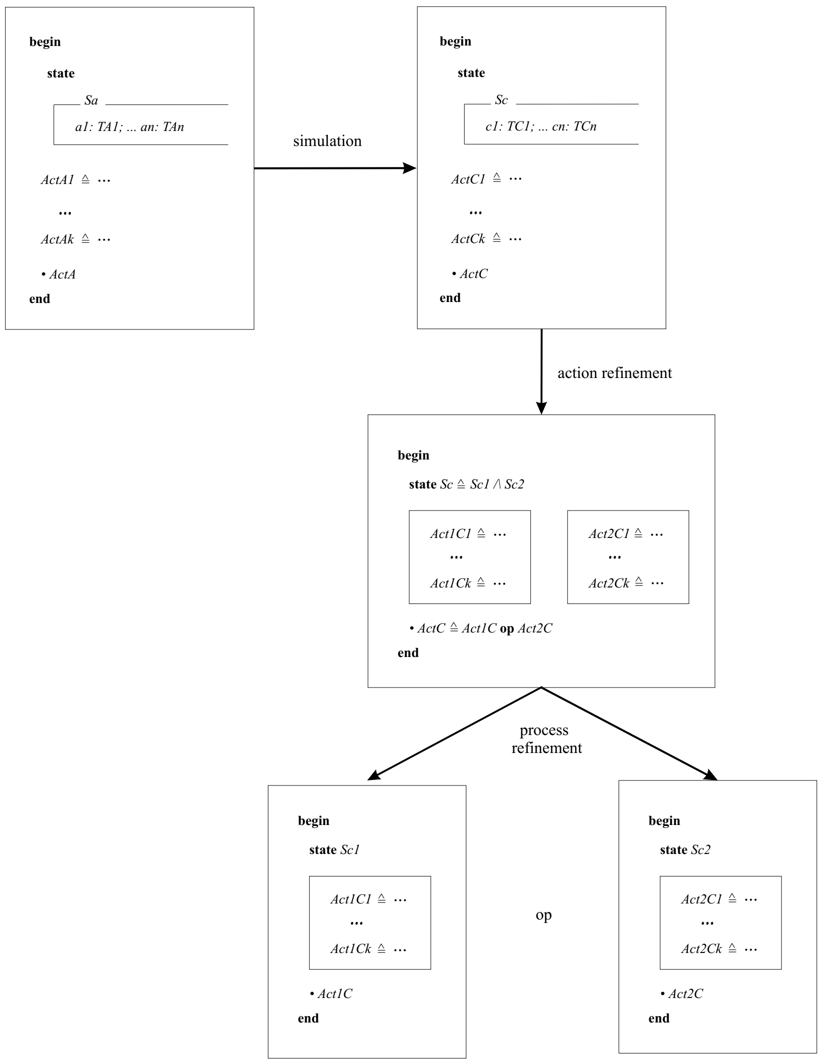

Fig. 1. An iteration of the refinement strategy

two different areas. Two extra zones are used for detection only. Fire detection happens in a zone, and, in consequence, a gas discharge may occur in the area that contains that zone.

The system includes a display panel composed of lamps that indicates whether the system is on or off, whether there are system faults, or a fire has been detected, whether the alarm has been silenced or not, the need to replace the actuators of the system, and gas discharges.

The system can be in one of three modes: manual, automatic, or disabled. In manual mode, an alarm sounds when a fire is detected, and the corresponding detection lamp is lit on the display. The alarm can be silenced, and, when the reset button is pressed, the system returns to normal. In manual mode, gas discharge is manually initiated.

System State	**Abst. FC Action**	**Conc. FC Action**	**Conc. Area Action**
$fireSysStart_s$	*AbstractFireSysStart*	*FireSysStart*	*StartArea*
$fireSys_s$	*AbstractFireSys*	*FireSys*	*AreaCycle*
$manual_s$	*AbstractManual*	*Manual*	*ManualArea*
$auto_s$	*AbstractAuto*	*Auto*	*AutoArea*
$reset_s$	*AbstractReset*	*Reset*	*ResetArea*
$countdown_s$	*AbstractCountdown*	*Countdown*	*WaitingDischarge*
$discharge_s$	*AbstractDischarge*	*Discharge*	*WaitingDischarge*
$fireSysD_s$	*AbstractFireSysD*	*FireSysD*	*AreaD*
$disabled_s$	*AbstractDisabled*	*Disabled*	*DisabledArea*

Table 1. The System States and Corresponding Actions

In automatic mode, a fire detection is also followed by the alarm being sounded; however, if a fire is detected in the second zone of the same area, the second stage alarm is sounded, and a countdown starts. When the countdown finishes, the gas is discharged and the circuit fault lamp is illuminated in the display; the system mode is switched to disabled.

In disabled mode, the system can only have the actuators replaced, identify relevant faults within the system, and be reset. The system is back to its normal mode after the actuators are replaced and the reset button is pressed.

The system may be in one of the states presented in Table 1. Initially, the system is on $fireSysStart_s$ state. After being switched on, its state is changed to $fireSys_s$; in this state, a fire detection yields to the state being changed to $manual_s$ or $auto_s$ depending on the system mode. In the state $reset_s$ the system is waiting to be reset; in $countdown_s$, it is waiting for the clock to finish the countdown. During gas discharge, the system is on the $discharge_s$ state, after which, the state is changed to $fireSysD_s$. Finally, if a fire is detected on $fireSysD_s$, the system state is changed to $disabled_s$.

Some further requirements should also be satisfied: the system must be started with a *switch* event, and, afterwards, the system *on* lamp should be illuminated; the system mode can be switched between manual and automatic mode provided no detection happens. Also, when the system is reset, all fire detection lamps must be switched off; if a gas discharge occurred, the actuators need to be replaced, and the system mode is switched to automatic. Following a fire detection, the corresponding lamp must be lit. After a gas discharge, no subsequent discharge may happen before the actuators are replaced.

The external channels of the fire control system are presented in Figure 2. Fire detection is indicated through channel *det*, which inputs the zone where it

channel $switch, silence, reset$
channel $actuatorsR, ckOn, ckOff$
channel $det : ZoneId$
channel $switchM : SwitchMode$
channel $extDisc : \mathbb{P}\ AreaId$
channel $fault : FaultId$
channel $alarm : AlarmStage$
channel $[T]lamp : T \times OnOff$
channel $buzzer : OnOff$
channel $sysSt : SystemState$

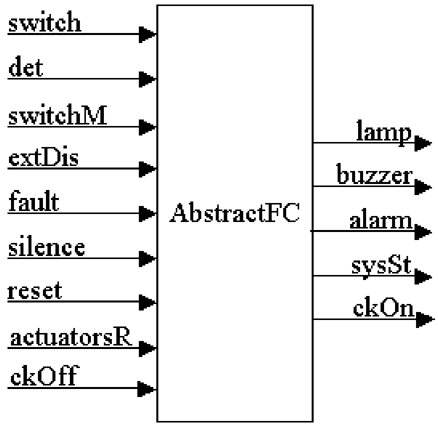

Fig. 2. System External Channels

happened. The system mode can be manually switched using channel *switch*. In manual mode, when the conditions that lead to a gas discharge are met, gas can be manually discharged using the channel *extDisc*. Faults are reported to the system through the channel *fault*. The channel *alarm* can be used to sound the alarm, which can be silenced through *silence*. Channel *reset* resets the system. The channel *actuatorsR* indicates that the actuators have been replaced. The system indicates that a lamp must be switched using the generic channel *lamp*; it provides the type of lamp and the new lamp mode. The buzzer is controlled using channel *buzzer*. After each state change, the system reports its current state using channel *sysSt*. The fire control system may request a clock to execute the countdown using channel *ckOn*; the clock indicates that the countdown is finished using channel *ckOff*.

The display is composed of the lamps and the buzzer. The lamps can be of three different types; however, the three types of lamps are instances of the same generic process *GenericLamp*, which has a component $status : OnOff$. Initially, all the lamps are switched *off*; they can be switched *on* using an appropriate instance of channel *lamp*.

3.1 Abstract Fire Control System

The basic types used within the system are presented in Figure 3. The areas

$$
\begin{aligned}
&\mathit{AreaId} ::= 0 \mid 1\\
&\mathit{ZoneId} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5\\
&\mathit{Mode} ::= \mathit{automatic} \mid \mathit{manual} \mid \mathit{disabled}\\
&\mathit{SwitchMode} == \mathit{Mode} \setminus \{\mathit{disabled}\}\\
&\mathit{OnOff} ::= \mathit{on} \mid \mathit{off}\\
&\mathit{AlarmStage} ::= \mathit{alarmOff} \mid \mathit{firstStage} \mid \mathit{secondStage}\\
&\mathit{LampId} ::= \mathit{zoneFaultL} \mid \mathit{earthFaultL} \mid \mathit{sounderLineFaultL} \mid \mathit{powerFaultL} \mid \mathit{sysOnL}\\
&\qquad\qquad \mid \mathit{remoteSignalL} \mid \mathit{actuatorLineFaultL} \mid \mathit{circuitFaultL} \mid \mathit{alarmSilencedL}\\
&\mathit{FaultId} ::= \mathit{ZoneF} \mid \mathit{earthF} \mid \mathit{sounderLineF} \mid \mathit{powerF} \mid \mathit{remoteSignal}\\
&\qquad\qquad \mid \mathit{actuatorLineF}\\
&\mathit{SystemState} ::= \mathit{fireSysStart}_s \mid \mathit{fireSys}_s \mid \mathit{fireSysD}_s \mid \mathit{auto}_s\\
&\qquad\qquad\qquad \mid \mathit{countdown}_s \mid \mathit{discharge}_s \mid \mathit{reset}_s \mid \mathit{manual}_s \mid \mathit{disabled}_s
\end{aligned}
$$

Fig. 3. System Types

and zones are identified by the types *AreaId* and *ZoneId*; the system modes are represented by the type *Mode*; the type *SwitchMode*, is a subset of type *Mode*. All the lamps and the buzzer of the display can be either on or off, which are represented by the type *OnOff*. The alarm states are represented by the type *AlarmStage*. The type *LampId* contains identifiers for all the lamps in the system's display. Faults are represented by the type *FaultId*. Finally, the system can be in one of the states of the type *SystemState*.

Process *AbstractFC* formalises the requirements previously described. Throughout this paper we omit some formal definitions for the sake of conciseness; they can be found in [12]. The abstract state is defined by the Z schema named *AbstractFCSt* presented below. Z schemas can either be represented as boxes, as *AbstractFCSt*, or in a horizontal notation as we shall see later in this paper. *AbstractFCSt* is composed of five components, which are declared in the declaration part of the schema: *mode* indicates the mode in which the fire control is running; *controlZns* is a total function that maps the areas to a set that contains their controlled zones; *actZns* maps the areas to the zones in which a fire detection has occurred; *discharge* indicates in which areas a gas discharged happened; finally, *active* contains the active areas identifications.

process $AbstractFC \mathrel{\widehat=}$ **begin**
state

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;AbstractFCSt\;}} \\
mode : Mode \\
controlZns, actZns : AreaId \rightarrow \mathbb{P}\, ZoneId \\
discharge, active : \mathbb{P}\, AreaId \\
\hline
\forall a : AreaId \bullet \\
\quad (mode = manual) \Rightarrow a \in active \Leftrightarrow \#actZns\ a \geq 1 \\
\quad \wedge\ (mode = automatic) \Rightarrow a \in active \Leftrightarrow \#actZns\ a \geq 2 \\
\quad \wedge\ actZns\ a \subseteq controlZns\ a \wedge controlZns\ a = getZones\ a \\
\hline
\end{array}
$$

The state invariant is declared in the predicate part of the schema; it determines that, if the system is running in *manual* mode (predicate $mode = manual$), an area is *active* if, and only if, some zone controlled by it is active. On the other hand, if the mode is *automatic*, an area is active if, and only if, there is more than one active zone controlled by it. Finally, for each area, its controlled zones are defined by the function *getZones*, whose definition we omit.

Initially, the system is in *automatic* mode, there is no active zone, and no discharge occurred in any area. The state invariant guarantees that there is no active area.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;InitAbstractFC\;}} \\
AbstractFCSt' \\
\hline
mode' = automatic \wedge discharge' = \emptyset \\
actZns' = \{ a : AreaId \bullet a \mapsto \emptyset \} \\
\hline
\end{array}
$$

Undashed variables represent the variable values before the execution of an operation; on the other hand, dashed variables represent the variable values after the execution of an operation. The decoration of a schema

$$Schema \mathrel{\widehat=} [\, x_1 : T_1 \ldots\ x_n : T_n \mid p \,]$$

is defined as the decoration of all the components of the schema, and the modification of the predicate part of the schema to reflect the new names of these components. For instance, we have that

$$Schema' \mathrel{\widehat=} [\, x_1' : T_1 \ldots\ x_n' : T_n \mid p\,[x_1'/x_1, \ldots\ , x_n'/x_n]\,].$$

Finally, the inclusion of the schema $AbstractFCSt'$ in the declaration part of *InitAbstractFC*, merges the declarations of both schemas, and conjoins their predicates.

Three operations are used to switch the system mode; they leave the other components unchanged. The first operation receives the new mode as argument.

For any schema *State* that describes the state of a system, $\Delta\ State$ is a schema that includes both *Schema* and *Schema′*. Furthermore, the name of input components must end with a query (?) and the name of output components must end with a shriek (!).

$$
\begin{array}{|l}
\hline
\textit{SwitchAbstractFCMode} \\
\Delta AbstractFCSt;\ nm? : Mode \\
\hline
mode' = nm? \land actZns' = actZns \land discharge' = discharge \\
\hline
\end{array}
$$

SwitchAbstractFC2Auto and *SwitchAbstractFC2Dis* do not receive arguments; they switch the mode to *automatic* and *disabled*, respectively.

The schema *AbstractActivateZone* receives a zone *nz*? and changes *actZns* by including *nz*? in the set of active zones of the area that controls it; *active* may also be changed to maintain the state invariant. All other state components are left unchanged.

$$
\begin{array}{|l}
\hline
\textit{AbstractActivateZone} \\
\Delta AbstractFCSt;\ nz? : ZoneId \\
\hline
mode' = mode \land discharge' = discharge \\
actZns' = actZns\ \oplus \{a : AreaId \mid nz? \in controlZns\ a \bullet \\
\qquad\qquad\qquad\qquad a \mapsto actZns\ a \cup \{nz?\}\} \\
\hline
\end{array}
$$

The schema *AbstractAutomaticDischarge* activates the discharge in the active areas, only *discharge* is changed. Finally, *AbstractManualDischarge* receives the areas in which the user wants to discharge the gas, but discharges only in those that are *active*.

All the other actions are defined using CSP operators. Basically, we have one action for each possible state within the system as described in Table 1.

The action *AbstractFireSysStart* starts by communicating the current system state. Then, it waits for the system to be switched on through channel *switch*, switches on the lamp *SysOnL*, initialises the system state and, finally, behaves like action *AbstractFireSys*.

$$
\begin{array}{l}
AbstractFireSysStart \mathrel{\widehat=} sysSt!fireSysStart_s \rightarrow switch \rightarrow \\
\qquad lamp[LampId].sysOnL!on \rightarrow InitAbstractFC;\ AbstractFireSys
\end{array}
$$

In action *AbstractFireSys*, after communicating the system state, the mode can be manually switched between *automatic* and *manual*. Furthermore, if any detection occurs, the zone in which the detection occurred is activated, the corresponding lamp is lit, the alarm sounds in *firstStage*, and then, the system behaves like *AbstractManual* or *AbstractAuto*, depending on the current system mode. If the actuators are replaced, the *circFaultL* is switched off, the system is set to *automatic* mode, and waits to be *reset*. Finally, if any *fault* is identified, the corresponding *lamp* is lit, and the buzzer is switched *on*.

$$
\begin{array}{l}
AbstractFireSys \mathrel{\widehat=} \\
\quad sysSt!fireSys_s \rightarrow \\
\qquad switchM?nm \rightarrow SwitchAbstractFCMode;\ AbstractFireSys \\
\qquad \Box\ det?nz \rightarrow AbstractActivateZone;\ lamp[ZoneId].nz!on \rightarrow \\
\qquad\quad alarm!firstStage \rightarrow \\
\qquad\qquad (mode = manual)\ \&\ AbstractManual \\
\qquad\qquad \Box\ (mode = automatic)\ \&\ AbstractAuto \\
\qquad \Box\ actuatorsR \rightarrow lamp[LampId].circFaultL!off \rightarrow \\
\qquad\quad SwitchAbstractFC2Auto;\ AbstractReset \\
\qquad \Box\ fault?faultId \rightarrow lamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\qquad\quad buzzer!on \rightarrow AbstractFireSys
\end{array}
$$

The function *getLampId* maps fault identifications to their corresponding lamp in the display.

Throughout this paper, we illustrate the refinement of the fire control system using these two actions only. For this reason, we omit the definitions of the remaining actions.

The main action of process *AbstractFireSys* is defined below.

- *AbstractFireSysStart* **end**

In the next section, we refine *AbstractFC* to a concrete distributed system.

3.2 Refinement

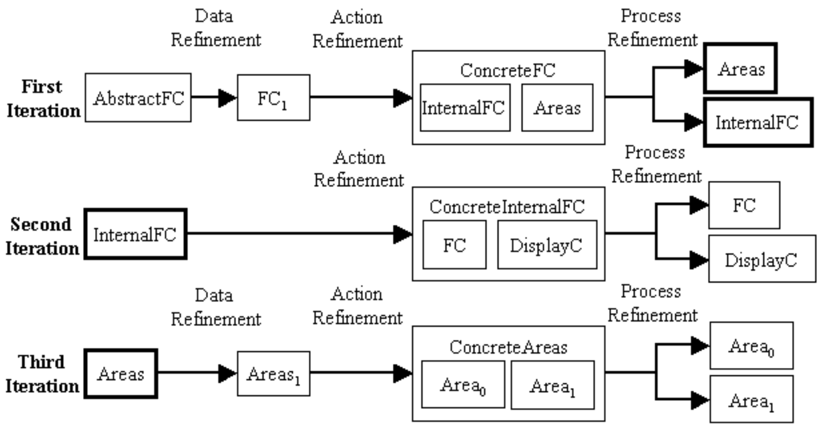

Fig. 4. Refinement Strategy for the Fire Control System

The motivation for the fire control system refinement is the distribution of the areas, in order to increase efficienct. Section 3.2 presents the target of our refinement, the concrete fire control system. In the following sections, we present the refinement steps summarised graphically in Figure 4.

In the first iteration, we split *AbstractFC* into two process *Areas* and *InternalFC*. The first models the areas of the system, and is split into two interleaved *Area* processes in interleaving in the last iteration. The second is the core of the system, which is split into a display controller *DisplayC* and the system controller *FC* in the second iteration.

Concrete Fire Control System The concrete fire control system has three components: the controller, the display, and the detection system. They communicate through the channels below.

channel $display, manDis : \mathbb{P}\, AreaId$
channel $switched, autoDis, anyDis, noDis, countdown, counting$
channel $gasDischarged, gasNotDischarged : AreaId$

The controller indicates discharges to the display through *display*. The display acknowledges this communication through channel *switched*. The controller request gas discharges to the detection process through *manDis* and *autoDis*. The detection process may reply to these requests indicating if the gas has been discharged (*anyDis*) or not (*noDis*); it may request a *countdown*, if it is *automatic* mode and the conditions for a gas discharge are met. The controller indicates that it started counting through *counting*. In Figure 5, we summarise the internal communications of the concrete fire control system.

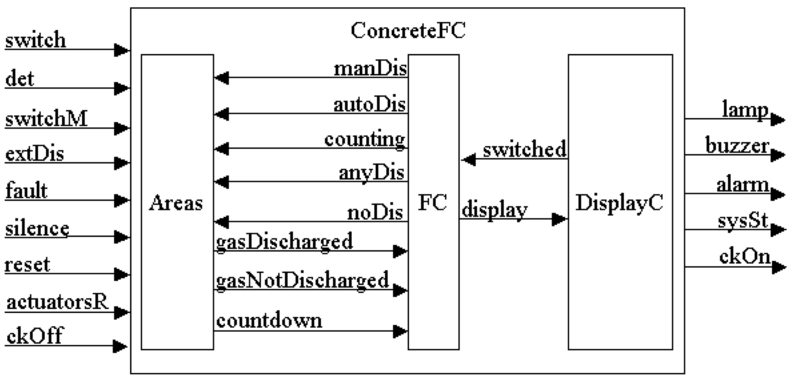

Fig. 5. Concrete Fire Control

Controller The process *FC* is similar to the abstract specification. However, all the state components and events related to the detection areas and to the display are removed. For conciseness, some schemas, as the system state presented below, are presented in their horizontal form $name \mathrel{\widehat=} [\, declaration \mid predicate \,]$.

$$
\begin{array}{l}
\textbf{process } FC \mathrel{\widehat=} \textbf{begin state } FCSt \mathrel{\widehat=} [\, mode_1 : Mode \,] \\
\quad InitFC \mathrel{\widehat=} [\, FCSt' \mid mode_1 = automatic \,]
\end{array}
$$

The state of the concrete fire control is composed of only one component, $mode_1$, which indicates the mode in which the system is running. This mode is initialised to *automatic*.

Three operations can be used to switch the system mode. The first one receives the new mode as argument.

$$SwitchFCMode \mathrel{\widehat=} [\, \Delta FCSt;\ nm? : Mode \mid mode_1 = nm? \,]$$

The second and third operations do not receive any argument; they simply switch the system mode to *automatic* or *disabled*.

The fire control system is responsible for communicating the current system state. After being switched on, the fire control initialises its state and behaves like action *FireSys*. Where a lamp was switched *on* in the abstract specification, an acknowledgment event *switched* is received from the the display controller.

$$FireSysStart \mathrel{\widehat=} sysSt!fireSysStart_s \rightarrow switch \rightarrow switched \rightarrow InitFC;\ FireSys$$

Similar to the abstract system, all the other actions corresponds to a possible state within the system as described in Table 1.

In action *FireSys*, after communicating the system state, the mode can be switched. Furthermore, if any detection occurs, the controller waits for a *switched* signal, sets the alarm to *firstStage*, and behaves like *Manual* or *Auto*, depending on the current system mode. Since the areas are the processes which have the area-zone information, following a *det* communication, the zone activation is not part of the controller behaviour. If the actuators are replaced, the system is set to *automatic* mode, and waits to be *reset*. Finally, all the faults are ignored by this process, except that it waits for a *switched* signal from the display.

$$
\begin{array}{l}
FireSys \mathrel{\widehat=}\ sysSt!fireSys_s \rightarrow \\
\qquad\qquad switchM?nm \rightarrow SwitchFCMode;\ FireSys \\
\qquad\qquad \Box\ det?nz \rightarrow switched \rightarrow alarm!firstStage \rightarrow \\
\qquad\qquad\qquad (mode_1 = manual)\ \&\ Manual \\
\qquad\qquad\qquad \Box\ (mode_1 = automatic)\ \&\ Auto \\
\qquad\qquad \Box\ actuatorsR \rightarrow switched \rightarrow SwitchFC2Auto;\ Reset \\
\qquad\qquad \Box\ fault?faultId \rightarrow switched \rightarrow FireSys
\end{array}
$$

$$\bullet\ FireSysStart\ \textbf{end}$$

As for the abstract system, we omit the definition of the remaining actions. The main action of process *FC* is *FireSysStart* presented above.

Display Controller This process models the display controller requests for the lamps to be switched *on* or *off* after the occurrence of the relevant events. It waits for the system to be *switch*ed *on*, switches the lamp *sysOnL on*, and indicates this to *FC* through *switched*. A gas discharge is indicated by *FC* to this process through *display*. If the system is *reset*, the display switches *off* the buzzer and all the lamps, except the lamps *circFaultL* and *sysOnL*.

Areas The process *Area* is parametrised by the area identifier.

$$\textbf{process } Area \mathrel{\widehat=} (id : AreaId \bullet \textbf{begin}$$

The state of an area is composed of the mode in which it is running, its controlled zones, the active zones in which a fire detection occurred, a boolean *discharge* that records whether a gas discharge has occurred or not, and a boolean *active* that records whether the area is willing to discharge gas or not.

state *AreaState*

$$\begin{array}{|l}
mode : Mode \\
controlZns, actZns : \mathbb{P}\, ZoneId \\
discharge, active : Bool \\
\hline
controlZns = getZones\ id \land actZns \subseteq controlZns \\
(mode = automatic) \Rightarrow active = true \Leftrightarrow \#actZns \geq 2 \\
(mode = manual) \Rightarrow active = true \Leftrightarrow \#actZns \geq 1
\end{array}$$

The invariant establishes that the component *actZns* is a subset of the controlled zones of this area, which is defined by *getZones*. Besides, if running in *automatic* mode, an area is active if, and only if, all controlled zone are *active*. On the other hand, if running in *manual* mode, an area is *active* if, and only if, any controlled zone is active.

Each area is initialised as follows: there is no active zone; no discharge occurred; and it is in *automatic* mode. The state invariant guarantees that it is not *active*.

InitArea

$$\begin{array}{|l}
AreaState' \\
\hline
actZns' = \emptyset \land discharge' = false \land mode' = automatic
\end{array}$$

The schema *SwitchAreaMode* receives the new mode and sets the area mode. Schemas *SwitchArea2Auto* and *SwitchArea2Dis* set the are *mode* to *automatic* and *disabled*. All other state components are left unchanged. A zone can be activated using the operation *ActivateZone*. If the given zone is controlled by the area, it is included in the *actZns*.

Initially, an area synchronises in the *switch* event, initialises its state, and starts its cycle.

$$StartArea \mathrel{\widehat=} switch \to InitArea;\ AreaCycle$$

During its cycle, if the *actuatorsR* event occurs, the mode is switched to *automatic* and the area waits to be *reset*. If the system mode is switched, so is the area mode. Finally, any detection may activate a zone, if it is controlled by this area; after this, the area behaves like either *AutoArea* or *ManualArea*, depending on its current mode.

$$
\begin{array}{ll}
AreaCycle \mathrel{\widehat=} & actuatorsR \rightarrow SwitchArea2Auto; ResetArea \\
& \Box\ switchM?nm \rightarrow SwitchAreaMode;\ AreaCycle \\
& \Box\ det?nz \rightarrow ActivateZone; \\
& \qquad\qquad\qquad\qquad (mode = automatic)\ \&\ AutoArea \\
& \qquad\qquad\qquad\qquad \Box\ (mode = manual)\ \&\ ManualArea \\
\bullet\ StartArea\ \mathbf{end})
\end{array}
$$

The main action of the process *Area* is the action *StartArea*.

The process *ConcreteAreas* represents all the areas within the system. Basically, it is a parallel composition of all areas. They synchronise on the channel set Σ_{areas}.

$$
\begin{array}{l}
\mathbf{chanset}\ \Sigma_{areas} == \{\!|\ switch, reset, switchM, det, silence, actuatorsR, \\
\qquad\qquad\qquad autoDis, manDis, anyDis, noDis, counting\ |\!\} \\
\mathbf{process}\ ConcreteAreas \mathrel{\widehat=}\ \parallel id : AreaId\ [\![\ \Sigma_{areas}\ ]\!] \bullet Area(id)
\end{array}
$$

The internal system is defined as the parallel composition of the fire control *FC* and the display controller *DisplayC*. All the communications between them are hidden.

$$
\begin{array}{l}
\mathbf{chanset}\ DisplaySync == \{\!|\ display, switched\ |\!\} \\
\mathbf{chanset}\ \Sigma_1 == \{\!|\ switch, reset, det, display, silence, actuatorsR, fault\ |\!\} \\
\mathbf{process}\ ConcreteInternalFC \mathrel{\widehat=} FC\ [\![\ \Sigma_1\ ]\!]\ DisplayC \setminus DisplaySync
\end{array}
$$

The concrete fire control is the parallel combination of *ConcreteInternalFC* and *Areas*. Internal communications are again hidden.

$$
\begin{array}{l}
\mathbf{chanset}\ GSync == \{\!|\ manDis, autoDis, countdown, counting, \\
\qquad\qquad\qquad gasDischarged, gasNotDischarged, anyDis, noDis\ |\!\} \\
\mathbf{chanset}\ \Sigma_2 == \\
\qquad \{\!|\ switch, reset, det, switchM, silence, actuatorsR\ |\!\} \cup GSync \\
\mathbf{process}\ ConcreteFC \mathrel{\widehat=} (ConcreteInternalFC\ [\![\ \Sigma_2\ ]\!]\ Areas) \setminus GSync
\end{array}
$$

In the following sections, we prove that *AbstractFC* is refined by *ConcreteFC*, or rather, $AbstractFC \sqsubseteq ConcreteFC$.

First Iteration: splitting the *AbstractFC* **into** *InternalFC* **and** *Areas*

Data refinement In this step we make a data refinement in order to introduce a state component that is used by the areas. The new $mode_A$ component indicates the mode in which the areas are running. The process *AbstractFC* is refined to the process FC_1 presented below.

$$\mathbf{process}\ FC_1 \mathrel{\widehat=} \mathbf{begin}$$
$$\mathbf{state}$$

$$\begin{array}{l}
FCSt_1 \\
\hline
mode_1, mode_A : Mode \\
controlZns_1, actZns_1 : AreaId \rightarrow \mathbb{P}\, ZoneId \\
discharge_1, active_1 : \mathbb{P}\, AreaId \\
\hline
\forall\, a : AreaId \bullet \\
(mode_1 = automatic) \Rightarrow a \in active_1 \Leftrightarrow \#actZns_1\ a \geq 2 \\
\land\ (mode_1 = manual) \Rightarrow a \in active_1 \Leftrightarrow \#actZns_1\ a \geq 1 \\
\land\ actZns_1\ a \subseteq controlZns_1\ a \land controlZns_1\ a = getZones\ a
\end{array}$$

The state $FCSt_1$ is the same as that of *AbstractFC*, except that it includes an extra component $mode_A$. In order to prove that the FC_1 is a refinement of the *AbstractFC*, we have to prove that there exists a forwards simulation between the main actions of FC_1 and *AbstractFC*. The retrieve relation *RetrFC* relates each component in the *AbstractFCSt* to one in $FCSt_1$.

$$\begin{array}{l}
RetrFC \\
\hline
AbstractFCSt;\ FCSt_1 \\
\hline
mode_1 = mode \land mode_A = mode \land controlZns_1 = controlZns \\
actZns_1 = actZns \land discharge_1 = discharge \land active_1 = active
\end{array}$$

The laws of *Circus* establish that simulation distributes through the structure of an action. The laws used here are in Appendices A and B; we refine each schema using Law A1. In the concrete initialisation, the new state component $mode_A$ is initialised in *automatic* mode.

$$\begin{array}{l}
InitFC_1 \\
\hline
FCSt_1' \\
\hline
mode_1' = automatic \land mode_A' = automatic \land discharge_1' = \emptyset \\
actZns_1' = \{a : AreaId \bullet a \mapsto \emptyset\}
\end{array}$$

The following lemma states that this is actually a simulation of the abstract initialisation. The symbol $\preceq$ represents the simulation relation.

Lemma 1. $InitAbstractFC \preceq InitFC_1$

Proof. The application of Law A1 raises two proof obligations. The first one concerns the preconditions of both schemas.

$$\forall \, AbstractFCSt; \; FCSt_1 \bullet RetrFC \land \text{pre } InitAbstractFC \Rightarrow \text{pre } InitFC_1$$

It is easily proved because the preconditions of both schemas are *true*. The second proof obligation concerns the postcondition of both operations.

$$\begin{array}{l}\forall \, AbstractFCSt; \; FCSt_1; \; FCSt_1' \bullet RetrFC \land \text{pre } InitAbstractFC \land InitFC_1 \Rightarrow \\ \quad \exists \, AbstractFCSt' \bullet RetrFC' \land InitAbstractFC\end{array}$$

This proof obligation can also be easily discarded using the one-point rule. When this rule is applied, we remove the universal quantifier, and then, we are left with an implication in which the consequent is present in the antecedent. □

There is no special rule to handle initialisation operations. This is because the behaviour of a process is defined by its main action; there is no implicit initialisation. An initialisation schema is just a simplified way of specifying an operation like any other.

All other schema expressions are refined in pretty much the same way. Their definitions are very similar to the corresponding abstract operations except that the value assigned to $mode_1$ is also assigned to the new state component $mode_A$.

For the remaining actions, we rely on distribution of simulation. The new actions have the same structure as the original ones, but use the new schemas. By way of illustration, we present the action $FireSysStart_1$ that simulates the action $AbstractFireSysStart$.

$$\begin{array}{l}FireSysStart_1 \mathrel{\widehat=} sysSt!fireSysStart_s \rightarrow switch \rightarrow \\ \qquad\qquad\qquad lamp[LampId].sysOnL!on \rightarrow InitFC_1; \; FireSys_1\end{array}$$

To establish the simulation, we need Laws A2 and A3. Since all the output and input values, and guards are not changed, only their second proviso must be proved. They follow from Lemma 1 and $FireSys \preceq FireSys_1$.

$FireSysStart_1$ is the main action of FC_1, and we have just proved that it simulates the main action of $AbstractFC$.

$\bullet \; FireSysStart_1$ **end**

This concludes this data refinement step.

Action Refinement In this step we change FC_1 so that its state is composed of two partitions: one that models the internal system and another that models the areas. We also change the actions so that the state partitions are handled separately.

process $ConcreteFC \mathrel{\widehat=}$ **begin**

The internal system state is composed only by its mode.

$$InternalFCSt \mathrel{\widehat=} [\, mode_1 : Mode \,]$$

The remaining components are declared as components of the areas partition of the state.

$$
\begin{array}{|l}
\hline
\mathit{AreasSt} \\
\mathit{mode}_A : \mathit{Mode} \\
\mathit{controlZns}_1, \mathit{actZns}_1 : \mathit{AreaId} \rightarrow \mathbb{P}\,\mathit{ZoneId} \\
\mathit{discharge}_1, \mathit{active}_1 : \mathbb{P}\,\mathit{AreaId} \\
\hline
\forall\, a : \mathit{AreaId} \bullet \\
\quad (\mathit{mode}_A = \mathit{automatic}) \Rightarrow a \in \mathit{active}_1 \Leftrightarrow \#\mathit{actZns}_1\, a \geq 2 \\
\quad \wedge\, (\mathit{mode}_A = \mathit{manual}) \Rightarrow a \in \mathit{active}_1 \Leftrightarrow \#\mathit{actZns}_1\, a \geq 1 \\
\quad \wedge\, \mathit{actZns}_1\, a \subseteq \mathit{controlZns}_1\, a \wedge \mathit{controlZns}_1\, a = \mathit{getZones}\, a \\
\hline
\end{array}
$$

The state of $FCSt_1$ is declared as the conjunction of the two previously defined schemas.

$$\textbf{state}\ \mathit{FCSt}_1 \mathrel{\widehat=} \mathit{InternalFCSt} \wedge \mathit{AreasSt}$$

The first group of paragraphs access only $mode_1$. It is initialised to *automatic*.

$$\mathit{InitInternalFC} \mathrel{\widehat=} [\, \mathit{InternalFCSt}';\ \mathit{AreasSt}' \mid \mathit{mode}_1' = \mathit{automatic}\,]$$

Another convention is used in the definitions that follow: for any schema Sch, ΞSch represents the schema that includes both Sch and Sch' and leaves the components values unchanged. The notation θSch denotes the bindings of components from Sch.

$$
\begin{array}{|l}
\hline
\Xi\mathit{Schema} \\
\mathit{Sch} \\
\mathit{Sch}' \\
\hline
\theta\mathit{Sch} = \theta\mathit{Sch}' \\
\hline
\end{array}
$$

The schema *SwitchInternalFCMode* receives the new mode as argument, and switches the *InternalFC* mode.

$$
\begin{array}{l}
\mathit{SwitchInternalFCMode} \mathrel{\widehat=} \\
\quad [\, \Delta\mathit{InternalFCSt};\ \Xi\mathit{AreasSt};\ \mathit{nm}? : \mathit{Mode} \mid \mathit{mode}_1' = \mathit{nm}?\,]
\end{array}
$$

Similarly, $SwitchInternalFC2Auto$ and $SwitchInternalFC2Dis$ set the *InternalFC* mode to *automatic* and *disabled*, respectively.

The behaviour of this internal system is very similar to that of the abstract one (Table 1); however, after being switched on, it initialises only $mode_1$ and behaves like action $FireSys_2$. All the operations related to the areas are no longer controlled by the internal system actions, but by the areas actions. For instance, consider the action $FireSysStart_2$ below.

$$
\begin{array}{l}
\mathit{FireSysStart}_2 \mathrel{\widehat=} \mathit{sysSt}!\mathit{fireSysStart}_s \rightarrow \mathit{switch} \rightarrow \\
\qquad\qquad \mathit{lamp}[\mathit{LampId}].\mathit{sysOnL}!\mathit{on} \rightarrow \mathit{InitInternalFC};\ \mathit{FireSys}_2
\end{array}
$$

When a synchronisation on *switchM* happens, only the *InternalFC* mode is switched by action $FireSys_2$. Furthermore, since the information about the areas are no longer part of this partition, following a *det* communication, this action does not activate the area in which the detection occurred. If the actuators are replaced, this action switches the corresponding lamp *on*, switches only $mode_1$ to *automatic*, and waits to be *reset*. The behaviour, if any *fault* happens, is not changed.

$$
\begin{array}{ll}
FireSys_2 \mathrel{\widehat=} & sysSt!fireSys_s \rightarrow \\
& \quad switchM?nm \rightarrow SwitchInternalFCMode;\ FireSys_2 \\
& \quad \Box\ det?nz \rightarrow lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
& \qquad (mode_1 = manual)\ \&\ Manual_2 \\
& \qquad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
& \quad \Box\ actuatorsR \rightarrow lamp[LampId].circFaultL!off \rightarrow \\
& \qquad SwitchInternalFC2Auto;\ Reset_2 \\
& \quad \Box\ fault?faultId \rightarrow lamp[LampId].(getLampId\ faultId)!on \rightarrow \\
& \qquad buzzer!on \rightarrow FireSys_2
\end{array}
$$

The second group of paragraphs is concerned with the areas. They are initialised in *automatic* mode; furthermore, there are no active zones, no *discharge* has occurred, and no area is *active*.

$$
\begin{array}{|l}
\hline
\textit{InitAreas} \\
AreasSt';\ InternalFCSt' \\
\hline
mode'_A = automatic \land discharge'_1 = \emptyset \\
actZns'_1 = \{a : AreaId \bullet a \mapsto \emptyset\} \\
\hline
\end{array}
$$

The areas mode can be switched to a given mode with schema *SwitchAreasMode*. The areas mode can also be switched to *automatic* or *disabled* mode with the schema operations *SwitchAreas2Auto* and *SwitchAreas2Dis*, respectively.

$$
\begin{array}{|l}
\hline
\textit{SwitchAreasMode} \\
\Delta AreasSt;\ \Xi InternalFCSt;\ nm? : Mode \\
\hline
mode'_A = nm? \land actZns'_1 = actZns_1 \land discharge'_1 = discharge_1 \\
\hline
\end{array}
$$

The schema *ActivateZoneAS* includes a given zone $nz?$ in the set of active zones of the area that controls $nz?$.

$$
\begin{array}{|l}
\hline
\textit{ActivateZoneAS} \\
\Delta AreasSt;\ \Xi InternalFCSt;\ nz? : ZoneId \\
\hline
mode'_A = mode_A \land discharge'_1 = discharge_1 \\
actZns'_1 = actZns_1\ \oplus \{a : AreaId \mid nz? \in controlZns_1\ a \bullet \\
\qquad\qquad\qquad\qquad a \mapsto actZns_1\ a \cup \{nz?\}\} \\
\hline
\end{array}
$$

Initially, the areas synchronise on *switch*, initialise the state, and start their cycle.

$$
StartAreas \mathrel{\widehat=} switch \rightarrow InitAreas;\ AreasCycle
$$

In *AreasCycle*, the actuators can be replaced, setting the mode to *automatic*, and the areas wait to be *reset*. If the system mode is switched, so is the areas mode. Any detection in a zone nz leads to the activation of nz; the behaviour afterwards depends on the *Areas* mode.

$$
\begin{array}{rl}
AreasCycle \mathrel{\widehat=} & actuatorsR \rightarrow SwitchAreas2Auto; ResetAreas \\
& \Box\; switchM?nm \rightarrow SwitchAreasMode; AreasCycle \\
& \Box\; det?nz \rightarrow ActivateZoneAS; \\
& \qquad\qquad\qquad (mode_A = automatic)\ \&\ AutoAreas \\
& \qquad\qquad\qquad \Box\; (mode_A = manual)\ \&\ ManualAreas
\end{array}
$$

As for the paragraphs of the internal system, the areas have an action corresponding to each action in the abstract system (Table 1); the remaining actions are omitted here.

The main action of *ConcreteFC* is the parallel composition of the actions $FireSysStart_2$ and *StartAreas*. These actions actually represent the initial actions of each partition within the process. They synchronise on the channel set Σ_2. All the synchronisation events between the internal system and the areas are hidden in the main action.

$$\bullet\ (FireSysStart_2 \llbracket \alpha(InternalFCSt) \mid \Sigma_2 \mid \alpha(AreasSt) \rrbracket\ StartAreas) \setminus GSync$$
end

Action $FireSysStart_2$ may modify only the components of *InternalFCSt*, and *StartAreas* may modify only the components of *AreasSt*.

Despite the fact that this is a significant refinement step, it involves no change of data representation. In order to prove that this is a valid refinement, we must prove that the main action of process *ConcreteFC* refines the main action of process FC_1; however, they are defined using mutual recursion, and for this reason, we use the result below in the proof. The symbol $\sqsubseteq_{\mathcal{A}}$ represents the action refinement relation.

Theorem 1 (Refinement on Mutual Recursive Actions). *For a given vector of actions S_S defined in the form $S_S \mathrel{\widehat=} [N_0, \ldots, N_n]$, where*

$$N_i \mathrel{\widehat=} F_i(N_0, \ldots, N_n)$$

we have that:

$$S_S \sqsubseteq_{\mathcal{A}} [Y_0, \ldots, Y_n] \Leftarrow \begin{pmatrix} F_0[Y_0, \ldots, Y_n/N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_0, \\ \ldots, \\ F_n[Y_0, \ldots, Y_n/N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_n \end{pmatrix}$$

In order to prove that a vector of actions S_S as defined above is refined by a vector of actions $[Y_0, \ldots, Y_n]$, it is enough to show that, for each action N_i in S_S, we can prove that its definition F_i, if we replace $N_0, \ldots, N_n$ with $Y_0, \ldots, Y_n$ in F_i, is refined by Y_i. This result is proved in [12].

We want to prove that

$$\mathit{FireSysStart}_1 \sqsubseteq_{\mathcal{A}} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GSync}$$

where $\parallel$ stands for $[\![\alpha(\mathit{InternalFCSt}) \mid \Sigma_2 \mid \alpha(\mathit{AreasSt})]\!]$. As $\mathit{FireSysStart}_1$ is defined using mutual recursion, we use the Theorem 1, with S_S as the vector including all actions involved in the definition of $\mathit{FireSysStart}_1$,

$$S_S = [\mathit{FireSysStart}_1, \mathit{FireSys}_1, \ldots]$$

to prove this refinement. The vector $[Y_0, \ldots, Y_n]$ includes

$$(\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GSync}$$

and all the refinements of each action in S_S as a parallel composition of the same form: with the same partition, the same synchronisation set, and the same hiding.

To prove this refinement, however, using Theorem 1, we need a modified S_S, in which some actions are preceded by an assumption. We introduce these assumptions using Law B8.

$$\begin{array}{l} [\mathit{FireSysStart}_1, \mathit{FireSys}_1, \ldots] \\ \sqsubseteq_{\mathcal{A}} [B8] \\ [\mathit{FireSysStart}_1, \{\mathit{mode}_1 = \mathit{mode}_A\};\ \mathit{FireSys}_1, \ldots] \end{array}$$

Although long, the proof obligation raised by this law application is trivial; we omit it here, for the sake of conciseness. Using Theorem 1 we get the following result.

$$\begin{array}{l} \begin{bmatrix} \mathit{FireSysStart}_1, \\ \{\mathit{mode}_1 = \mathit{mode}_A\};\ \mathit{FireSys}_1, \ldots \end{bmatrix} \\ \sqsubseteq_{\mathcal{A}} \begin{bmatrix} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GSync}, \\ (\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GSync}, \ldots \end{bmatrix} \\ \Leftarrow \\ \left(\begin{array}{ll} \mathit{FireSysStart}_1[\mathit{subst}] \sqsubseteq_{\mathcal{A}} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GSync}, & (1) \\ \mathit{FireSys}_1[\mathit{subst}] \sqsubseteq_{\mathcal{A}} (\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GSync}, \ldots & (2) \end{array} \right) \end{array}$$

Here, *subst* corresponds to the following substitution.

$$\mathit{subst} = \begin{pmatrix} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GSync}, \\ (\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GSync}, \ldots \end{pmatrix} / \begin{pmatrix} \mathit{FireSysStart}_1, \\ \mathit{FireSys}_1, \ldots \end{pmatrix}$$

Below, $A_1 \sqsubseteq_{\mathcal{A}} [\mathit{law}_1, \ldots, \mathit{law}_n]\{\mathit{op}_1\} \ldots \{\mathit{op}_n\}\ A_2$ denotes that A_1 may be refined to A_2 using laws $\mathit{law}_1, \ldots, \mathit{law}_n$, if $\mathit{op}_1, \ldots, \mathit{op}_n$ holds. Lemmas 2 and 3 prove refinements (1) and (2), respectively.

Lemma 2. (1) $FireSysStart_1[subst] \sqsubseteq_{\mathcal{A}} (FireSysStart_2 \parallel StartAreas) \setminus GSync$

Proof. We start the refinement using the definitions of $FireSysStart_1$ and substitution.

$$\begin{array}{l} FireSysStart_1[subst] \\ = [\text{Definition of } FireSysStart_1, \text{Definition of Substitution}] \\ sysSt!fireSysStart_s \to switch \to lamp[LampId].sysOnL!on \to \\ \quad\quad InitFC_1;\ (FireSys_2 \parallel AreasCycle) \setminus GSync \end{array}$$

First, we may expand the hiding since the channels $lamp$, $switch$, and $sysSt$ are not in $GSync$.

$$\begin{array}{l} = [A15]\ \{\{lamp, switch, sysSt\} \cap GSync = \emptyset\} \\ \left(\begin{array}{l} sysSt!fireSysStart_s \to switch \to lamp[LampId].sysOnL!on \to \\ \quad\quad InitFC_1;\ (FireSys_2 \parallel AreasCycle) \end{array}\right) \setminus GSync \end{array}$$

The schema $InitFC_1$ can be written as the sequential composition of two other schemas as follows. In [2], a refinement law is provided to introduce a schema sequence; however, in our case, we have a initialisation schema that has no reference to the initial state. For this reason, we use a new law that is similar to the one in [2]. Some trivial proof obligations are omitted.

$$\begin{array}{l} = [B3] \left(\begin{array}{l} sysSt!fireSysStart_s \to switch \to lamp[LampId].sysOnL!on \to \\ \quad\quad InitInternalFC;\ InitAreas;\ (FireSys_2 \parallel AreasCycle) \end{array}\right) \\ \setminus GSync \end{array}$$

Each one of the new inserted schema operations writes in a different partition of the parallelism that follows them. For this reason, we may distribute them over the parallelism. Again, two new laws are used: the first moves a (guarded) schema expression to one side of the parallelism; commutativity of parallelism is also provided as a new law.

$$\begin{array}{l} = [B13, B14] \\ \left(\begin{array}{l} sysSt!fireSysStart_s \to switch \to lamp[LampId].sysOnL!on \to \\ \quad\quad ((InitInternalFC;\ FireSys_2) \parallel (InitAreas;\ AreasCycle)) \end{array}\right) \setminus GSync \end{array}$$

Next, we move the $lamp$ event to the internal system side of the parallelism. This step is valid because all the initial channels of $AreasCycle$ are in Σ_2, and $lamp$ is not.

$$\begin{array}{l} = [A11]\ \{initials(AreasCycle) \subseteq \Sigma_2\}\ \{lamp \notin \Sigma_2\} \\ \left(\begin{array}{l} sysSt!fireSysStart_s \to switch \to \\ \quad\quad \left(\left(\begin{array}{l} lamp[LampId].sysOnL!on \to \\ \quad\quad InitInternalFC;\ FireSys_2 \end{array}\right) \parallel (InitAreas;\ AreasCycle)\right) \end{array}\right) \\ \setminus GSync \end{array}$$

Now, *switch* may be distributed over the parallelism because it is in Σ_2.

$$= [A14]\ \{switch \in \Sigma_2\}$$
$$\left(\begin{array}{l} sysSt!fireSysStart_s \rightarrow \\ \qquad \left(\begin{array}{l} \left(\begin{array}{l} switch \rightarrow \\ \qquad lamp[LampId].sysOnL!on \rightarrow \\ \qquad\qquad InitInternalFC;\ FireSys_2 \end{array}\right) \\ \parallel \\ \left(\begin{array}{l} switch \rightarrow InitAreas; \\ \qquad\qquad AreasCycle \end{array}\right) \end{array}\right) \end{array}\right)$$
$$\setminus\ GSync$$

Since it is not in Σ_2, *sysSt* may be moved to the internal system side of the parallelism.

$$= [B1, A11]\ \{sysSt \notin \Sigma_2\}$$
$$\left(\left(\begin{array}{l} sysSt!fireSysStart_s \rightarrow switch \rightarrow \\ \qquad lamp[LampId].sysOnL!on \rightarrow \\ \qquad\qquad InitInternalFC;\ FireSys_2 \end{array}\right) \parallel \left(\begin{array}{l} switch \rightarrow InitAreas; \\ \qquad\qquad AreasCycle \end{array}\right)\right)$$
$$\setminus\ GSync$$

Finally, using the definitions of $FireSysStart_2$ and $StartAreas$ we conclude this proof.

$$= [\text{Definition of } FireSysStart_2 \ and\ StartAreas]$$
$$(FireSysStart_2 \parallel StartAreas) \setminus GSync \qquad \square$$

The next lemma we present is the refinement of the action $FireSys_1$.

Lemma 3.

$$(2)$$
$$\{mode_1 = mode_A\};\ FireSys_1[subst]$$
$$\sqsubseteq_{\mathcal{A}}$$
$$(FireSys_2 \parallel AreasCycle) \setminus GSync$$

Proof. We start the proof using the definitions of $FireSys_1$ and substitution.

$$\{mode_1 = mode_A\};\ FireSys_1[subst]$$
$$= [\text{Definition of } FireSys_1, \text{Definition of Substitution}]$$
$$\begin{array}{l} \{mode_1 = mode_A\}; \\ sysSt!fireSys_s \rightarrow \\ \quad switchM?nm \rightarrow SwitchFCMode_1;\ (FireSys_2 \parallel AreasCycle) \setminus GSync \\ \quad \square\ det?nz \rightarrow ActivateZone_1;\ lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\ \qquad (mode_1 = manual)\ \&\ (Manual_2 \parallel ManualAreas) \setminus GSync \\ \qquad \square\ (mode_1 = automatic)\ \&\ (Auto_2 \parallel AutoAreas) \setminus GSync \\ \quad \square\ actuatorsR \rightarrow lamp[LampId].circFaultL!off \rightarrow \\ \qquad SwitchFC2Auto_1;\ (Reset_2 \parallel ResetAreas) \setminus GSync \\ \quad \square\ fault?faultId \rightarrow lamp[LampId].(getLampId\ faultId)!on \rightarrow \\ \qquad buzzer!on \rightarrow (FireSys_2 \parallel AreasCycle) \setminus GSync \end{array}$$

Next, we expand the hiding to the whole action. This is valid because all the events involved in the expansion are not in the hidden set of channels.

$$
\begin{array}{l}
= [A15] \\
\{ GSync \cap \{ sysSt, switchM, det, lamp, alarm, fault, buzzer, reset \} = \emptyset \} \\
\left(
\begin{array}{l}
\{ mode_1 = mode_A \}; \\
sysSt!fireSys_s \rightarrow \\
\quad switchM?nm \rightarrow SwitchFCMode_1; (FireSys_2 \parallel AreasCycle) \quad (3) \\
\quad \Box\ det?nz \rightarrow ActivateZone_1; \hfill (4) \\
\qquad lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\qquad\quad (mode_1 = manual)\ \&\ (Manual_2 \parallel ManualAreas) \\
\qquad\quad \Box\ (mode_1 = automatic)\ \&\ (Auto_2 \parallel AutoAreas) \\
\quad \Box\ actuatorsR \rightarrow lamp[LampId].circFaultL!off \rightarrow \hfill (5) \\
\qquad SwitchFC2Auto_1; (Reset_2 \parallel ResetAreas) \\
\quad \Box\ fault?faultId \rightarrow lamp[LampId].(getLampId\ faultId)!on \rightarrow (6) \\
\qquad buzzer!on \rightarrow (FireSys_2 \parallel AreasCycle)
\end{array}
\right) \\
\setminus\ GSync
\end{array}
$$

Next, we aim at the refinement of each branch to a parallelism in order to be able to apply the exchange Law A12. First, we refine (3) as follows: the schema $SwitchFCMode_1$ can be written as the sequential composition of $SwitchInternalFCMode$ and $SwitchAreasMode$.

$$
\begin{array}{l}
(3) = [A17]\ switchM?nm \rightarrow SwitchInternalFCMode; SwitchAreasMode; \\
\qquad\qquad\quad (FireSys_2 \parallel AreasCycle)
\end{array}
$$

Both schemas can be moved to different sides of the parallelism.

$$
\begin{array}{l}
= [B14, B13] \\
switchM?nm \rightarrow \\
\quad ((SwitchInternalFCMode; FireSys_2) \parallel (SwitchAreasMode; AreasCycle))
\end{array}
$$

Finally, as $switchM$ is in Σ_2, we may distribute this event over the parallelism. Here, a new law (distribution of input channels over parallelism) is used.

$$
\begin{array}{l}
= [B2]\ \{ switchM \in \Sigma_2 \} \\
\left(
\begin{array}{l}
switchM?nm \rightarrow \\
\quad SwitchInternalFCMode; FireSys_2
\end{array}
\right) \\
\parallel \\
\left(
\begin{array}{l}
switchM?nm \rightarrow \\
\quad SwitchAreasMode; AreasCycle
\end{array}
\right)
\end{array}
$$

For (4), we first use the assumption laws in order to move the assumption into the action.

$$
\begin{array}{l}
(4) \sqsubseteq_{\mathcal{A}} [B9, A7, A10, A16, B10, B12] \\
\quad det?nz \rightarrow ActivateZone_1; lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\qquad \{ mode_1 = mode_A \}; (mode_1 = manual)\ \&\ (Manual_2 \parallel ManualAreas) \\
\qquad \Box\ \{ mode_1 = mode_A \}; (mode_1 = automatic)\ \&\ (Auto_2 \parallel AutoAreas)
\end{array}
$$

Next, we use the assumption to change the guards.

$$
\begin{array}{l}
= [A8] \\
det?nz \rightarrow ActivateZone_1;\ lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\quad \{mode_1 = mode_A\}; \\
\qquad (mode_1 = manual \wedge mode_A = manual)\ \& \\
\qquad\quad (Manual_2 \parallel ManualAreas) \\
\quad \Box\ \{mode_1 = mode_A\}; \\
\qquad (mode_1 = automatic \wedge mode_A = automatic)\ \& \\
\qquad\quad (Auto_2 \parallel AutoAreas)
\end{array}
$$

The assumptions can then be absorbed by the guards.

$$
\begin{array}{l}
= [A4, A5, A10, A16] \\
det?nz \rightarrow ActivateZone_1;\ lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\quad (mode_1 = mode_A \wedge mode_1 = manual \wedge mode_A = manual)\ \& \\
\qquad (Manual_2 \parallel ManualAreas) \\
\quad \Box\ (mode_1 = mode_A \wedge mode_1 = automatic \wedge mode_A = automatic)\ \& \\
\qquad (Auto_2 \parallel AutoAreas)
\end{array}
$$

Now, using a new law, we distribute the guards over the parallelism, slightly changing them.

$$
\begin{array}{l}
= [B5] \\
det?nz \rightarrow ActivateZone_1;\ lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\quad \left(\begin{array}{c} \left(\begin{array}{l} mode_1 = mode_A \wedge \\ mode_1 = manual \end{array} \right) \& \\ Manual_2 \end{array} \right) \parallel \left(\begin{array}{c} \left(\begin{array}{l} mode_1 = mode_A \wedge \\ mode_A = manual \end{array} \right) \& \\ ManualAreas \end{array} \right) \\
\quad \Box \left(\begin{array}{c} \left(\begin{array}{l} mode_1 = mode_A \wedge \\ mode_1 = automatic \end{array} \right) \& \\ Auto_2 \end{array} \right) \parallel \left(\begin{array}{c} \left(\begin{array}{l} mode_1 = mode_A \wedge \\ mode_A = automatic \end{array} \right) \& \\ AutoAreas \end{array} \right)
\end{array}
$$

Now, since the guards invalidate each other, we may apply an exchange law. Furthermore, we simplify the guards.

$$
\begin{array}{l}
= [A12, A6] \\
det?nz \rightarrow ActivateZone_1;\ lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\quad \left(\begin{array}{l} (mode_1 = manual)\ \&\ Manual_2 \\ \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\
\quad \parallel \\
\quad \left(\begin{array}{l} (mode_A = manual)\ \&\ ManualAreas \\ \Box\ (mode_A = automatic)\ \&\ AutoAreas \end{array} \right)
\end{array}
$$

Next, we move the outputs channels to the left-hand side of the parallelism. This follows from the fact that the initial channels of both *ManualAreas* and

AutoAreas are in Σ_2, and *alarm* and *lamp* are not.

$$
\begin{array}{l}
= [B1, A11] \\
\{ initials(ManualAreas) \cup initials(AutoAreas) \subseteq \Sigma_2 \} \\
\{ \Sigma_2 \cap \{ alarm, lamp \} = \emptyset \} \\
det?nz \rightarrow ActivateZone_1; \\
\quad \left(\begin{array}{l}
lamp[ZoneId].nz!on \rightarrow \\
\quad alarm!firstStage \rightarrow \\
\quad\quad (mode_1 = manual) \,\&\, \\
\quad\quad\quad Manual_2 \\
\quad\quad \Box\, (mode_1 = automatic) \,\&\, \\
\quad\quad\quad Auto_2
\end{array} \right) \\
\quad \parallel \\
\quad \left(\begin{array}{l}
(mode_A = manual) \,\&\, \\
\quad ManualAreas \\
\Box\, (mode_A = automatic) \,\&\, \\
\quad AutoAreas
\end{array} \right)
\end{array}
$$

The schema $ActivateZone_1$ can easily be transformed to *ActivateZoneAS* using the schema calculus. The resulting schema can also be distributed over the parallelism. Finally, channel *det* can be distributed over the parallelism, since it is in Σ_2.

$$
\begin{array}{l}
= [\text{Schema Calculus}, B14, B13, B2] \; \{ det \in \Sigma_2 \} \\
\left(\begin{array}{l}
det?nz \rightarrow lamp[ZoneId].nz!on \rightarrow \\
\quad alarm!firstStage \rightarrow \\
\quad\quad (mode_1 = manual) \,\&\, Manual_2 \\
\quad\quad \Box\, (mode_1 = automatic) \,\&\, Auto_2
\end{array} \right) \\
\parallel \\
\left(\begin{array}{l}
det?nz \rightarrow ActivateZoneAS; \\
(mode_A = manual) \,\&\, \\
\quad ManualAreas \\
\Box\, (mode_A = automatic) \,\&\, \\
\quad AutoAreas
\end{array} \right)
\end{array}
$$

Using similar strategies, we refine (5) and (6) to the following external choice.

$$
\begin{array}{l}
(5,6) = [\ldots] \\
\quad \left(\begin{array}{l}
actuatorsR \rightarrow \\
\quad lamp[LampId].circFaultL!off \rightarrow \\
\quad\quad SwitchInternalFC2Auto;\; Reset_2
\end{array} \right) \\
\quad \parallel \\
\quad \left(\begin{array}{l}
actuatorsR \rightarrow \\
\quad SwitchAreas2Auto; \\
\quad\quad ResetAreas
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
fault?faultId \rightarrow lamp[LampId].(getLampId\, faultId)!on \rightarrow \\
\quad buzzer!on \rightarrow FireSys_2
\end{array} \right) \\
\quad\quad \parallel AreasCycle
\end{array}
$$

We are left with the external choice of parallel actions. Since the initial channels of the first three parallel actions are in the set Σ_2, we may apply the exchange law as follows.

$$
\begin{array}{l}
= [A12] \\
\mathit{sysSt}!\mathit{fireSys}_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{switchM}?nm \rightarrow \mathit{SwitchInternalFCMode};\ \mathit{FireSys}_2 \\
\Box\ \mathit{det}?nz \rightarrow \mathit{lamp}[\mathit{ZoneId}].nz!on \rightarrow \mathit{alarm}!\mathit{firstStage} \rightarrow \\
\qquad (\mathit{mode}_1 = \mathit{manual})\ \&\ \mathit{Manual}_2 \\
\qquad \Box\ (\mathit{mode}_1 = \mathit{automatic})\ \&\ \mathit{Auto}_2 \\
\Box\ \mathit{actuatorsR} \rightarrow \mathit{lamp}[\mathit{LampId}].\mathit{circFaultL}!\mathit{off} \rightarrow \\
\qquad \mathit{SwitchInternalFC2Auto};\ \mathit{Reset}_2
\end{array}\right) \\
\| \\
\left(\begin{array}{l}
\mathit{switchM}?nm \rightarrow \mathit{SwitchAreasMode};\ \mathit{AreasCycle} \\
\Box\ \mathit{det}?nz \rightarrow \mathit{ActivateZoneAS}; \\
\qquad (\mathit{mode}_A = \mathit{manual})\ \&\ \mathit{ManualAreas} \\
\qquad \Box\ (\mathit{mode}_A = \mathit{automatic})\ \&\ \mathit{AutoAreas} \\
\Box\ \mathit{actuatorsR} \rightarrow \mathit{SwitchAreas2Auto};\ \mathit{ResetAreas}
\end{array}\right)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\mathit{fault}?\mathit{faultId} \rightarrow \mathit{lamp}[\mathit{LampId}].(\mathit{getLampId}\ \mathit{faultId})!on \rightarrow \\
\qquad \mathit{buzzer}!on \rightarrow \mathit{FireSys}_2
\end{array}\right) \\
\quad \| \\
\quad \mathit{AreasCycle}
\end{array}\right)
\end{array}
$$

With small rearrangements, we have that the right-hand side of the first parallelism corresponds to the definition of the action *AreasCycle*. So, we have that both branches of the external choice have this action as the right-hand side of the parallelism. Since all the initials of *AreasCycle* are in Σ_2, we may apply the distribution of parallelism over external choice.

$$
\begin{array}{l}
= [A13]\ \{\mathit{initials}(\mathit{AreasCycle}) \subseteq \Sigma_2\} \\
\mathit{sysSt}!\mathit{fireSys}_s \rightarrow \\
\quad \left(\begin{array}{l}
\mathit{switchM}?nm \rightarrow \mathit{SwitchInternalFCMode};\ \mathit{FireSys}_2 \\
\Box\ \mathit{det}?nz \rightarrow \mathit{lamp}[\mathit{ZoneId}].nz!on \rightarrow \mathit{alarm}!\mathit{firstStage} \rightarrow \\
\qquad (\mathit{mode}_1 = \mathit{manual})\ \&\ \mathit{Manual}_2 \\
\qquad \Box\ (\mathit{mode}_1 = \mathit{automatic})\ \&\ \mathit{Auto}_2 \\
\Box\ \mathit{actuatorsR} \rightarrow \mathit{lamp}[\mathit{LampId}].\mathit{circFaultL}!\mathit{off} \rightarrow \\
\qquad \mathit{SwitchInternalFC2Auto};\ \mathit{Reset}_2 \\
\Box\ \mathit{fault}?\mathit{faultId} \rightarrow \mathit{lamp}[\mathit{LampId}].(\mathit{getLampId}\ \mathit{faultId})!on \rightarrow \\
\qquad \mathit{buzzer}!on \rightarrow \mathit{FireSys}_2
\end{array}\right) \\
\quad \| \\
\quad \mathit{AreasCycle}
\end{array}
$$

Finally, we can distribute *sysSt* and use the definition of $\mathit{FireSys}_2$ to conclude our proof. Again, this is valid because all the initials of *AreasCycle* are in Σ_2, and *sysSt* is not.

$$
\begin{array}{l}
= [B1, A11]\,\{initials(AreasCycle) \subseteq \Sigma_2\}\,\{\Sigma_2 \cap \{sysSt\} = \emptyset\} \\
\left(\begin{array}{l}
sysSt!fireSys_s \rightarrow \\
\quad switchM?nm \rightarrow SwitchInternalFCMode;\ FireSys_2 \\
\quad \Box\ det?nz \rightarrow lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\qquad (mode_1 = manual)\ \&\ Manual_2 \\
\qquad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\quad \Box\ actuatorsR \rightarrow lamp[LampId].circFaultL!off \rightarrow \\
\qquad SwitchInternalFC2Auto;\ Reset_2 \\
\quad \Box\ fault?faultId \rightarrow lamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\qquad buzzer!on \rightarrow FireSys_2
\end{array}\right) \\
\parallel \\
AreasCycle \\
= [\text{Definition of } FireSys_2] \\
(FireSys_2 \parallel AreasCycle) \setminus GSync \qquad \Box
\end{array}
$$

Using these lemmas, and those related to the remaining actions, which are omitted here, we prove that FC_1 is refined by *ConcreteFC*.

Process Refinement We partitioned the state of the process FC_1 into *InternalFCSt* and *AreasSt*. Each partition has its own set of paragraphs, which are disjoint, since, no action in one changes a state component in the other. Furthermore, the main action of the refined process is defined in terms of these two partitions. Therefore, we may apply Law A18 in order to split process *ConcreteFC* into two independent processes as follows.

$$\textbf{process}\ ConcreteFC \mathrel{\widehat=} (InternalFC\ [\![\, \Sigma_2\,]\!]\ Areas) \setminus GSync$$

The *ConcreteFC* is redefined as the parallel composition of *InternalFC* and *Areas*. Their definitions can be deduced from the definition of *ConcreteFC*.

Second Iteration: splitting *InternalFC* into two controllers In this iteration, we split *InternalFC* into two separated partitions: the first one corresponds to the *FC* controller, and the other the *DisplayC*ontroler (see Figure 4).

Action Refinement We rewrite the actions so that the *FC* paragraphs no longer deal with the display events, which are dealt by *DisplayC*. The fire control state is left unchanged.

$$
\begin{array}{l}
\textbf{process}\ ConcreteInternalFC \mathrel{\widehat=} \textbf{begin} \\
\quad FCSt \mathrel{\widehat=} [\, mode_1 : Mode\,]
\end{array}
$$

Furthermore, the display controller has no state at all. The new state is defined as follows.

$$\textbf{state}\ InternalFCSt_1 \mathrel{\widehat=} FCSt$$

The operations over the *InternalFCSt* are slightly changed: they are renamed and affect the *FCSt*, which is the same as the *InternalFCSt*. Their definitions, and those of all actions over *FCSt* have the same definition and description as those of *FC*. The display paragraphs are those of *DisplayC*, which can be found in Section 3.2.

The main action of the *ConcreteInternalFC* is as follows.

$$\bullet\ (\mathit{FireSysStart} \llbracket \alpha(\mathit{FCSt}) \mid \Sigma_2 \mid \alpha(\mathit{DisplayCState}) \rrbracket\ \mathit{StartDisplay}) \\ \setminus \mathit{DisplaySync}$$

end

We have the parallelism of action *FireSysStart* and *StartDisplay*, with the channels used exclusively for their communication hidden. Again, since $\mathit{FireSysStart}_2$, *FireSysStart*, and *StartDisplay* are defined using mutual recursion, we use Theorem 1 to prove that the process *InternalFC* is refined by *ConcreteInternalFC*.

Process Refinement Each partition in *ConcreteInternalFC* has its own set of paragraphs, which are disjoint. Furthermore, we define the main action of the refined process in terms of these two partitions. Applying Law A18, we get the following result.

$$\textbf{process}\ \mathit{ConcreteInternalFC} \mathrel{\widehat=} (\mathit{FC} \llbracket \Sigma_1 \rrbracket\ \mathit{DisplayC}) \setminus \mathit{DisplaySync}$$

The processes *FC* and the *DisplayC* were already described in the specification of the concrete system in Section 3.2.

Third Iteration: splitting the *Areas* into individual *Area*s This last iteration aims at splitting *Areas* in individual processes *Area* for each area.

Data Refinement First, we must apply a data refinement to the original process *Areas*.

$$\textbf{process}\ \mathit{Areas}_1 \mathrel{\widehat=} \textbf{begin}$$

We introduce a local state *AreaState* of an individual *Area*. Its definition is very similar to that of the concrete system, but includes an identifier $id : AreaId$. The global state *AreasSt* is rewritten with a total function from *AreaId* to local states. The invariant is slightly changed to handle the new data structure.

state

$$\begin{array}{|l} \hline \mathit{AreasSt}_1 \\ \mathit{areas} : \mathit{AreaId} \rightarrow \mathit{AreaState} \\ \hline \forall a : \mathit{AreaId} \bullet (\mathit{areas}\ a).\mathit{id} = a \\ \quad \wedge ((\mathit{areas}\ a).\mathit{mode} = \mathit{automatic}) \Rightarrow \\ \qquad (\mathit{areas}\ a).\mathit{active} = \mathit{true} \Leftrightarrow \#(\mathit{areas}\ a).\mathit{actZns} \geq 2 \\ \quad \wedge ((\mathit{areas}\ a).\mathit{mode} = \mathit{manual}) \Rightarrow \\ \qquad (\mathit{areas}\ a).\mathit{active} = \mathit{true} \Leftrightarrow \#(\mathit{areas}\ a).\mathit{actZns} \geq 1 \\ \quad \wedge (\mathit{areas}\ a).\mathit{actZns} \subseteq (\mathit{areas}\ a).\mathit{controlZns} \\ \quad \wedge (\mathit{areas}\ a).\mathit{controlZns} = \mathit{getZones}\ a \\ \hline \end{array}$$

The retrieve relation is very simple and is defined below.

$$
\begin{array}{l}
\textit{RetrieveAreas} \\
\hline
\textit{AreasSt};\ \textit{AreasSt}_1 \\
\hline
\forall\, a : \textit{AreaId} \bullet (\textit{areas}\ a).\textit{mode} = \textit{mode}_A \\
\qquad \wedge\ (\textit{areas}\ a).\textit{controlZns} = \textit{controlZns}_1\ a \\
\qquad \wedge\ (\textit{areas}\ a).\textit{actZns} = \textit{actZns}_1\ a \\
\qquad \wedge\ (\textit{areas}\ a).\textit{discharge} = \textit{true} \Leftrightarrow a \in \textit{discharge}_1 \\
\qquad \wedge\ (\textit{areas}\ a).\textit{active} = \textit{true} \Leftrightarrow a \in \textit{active}_1 \\
\hline
\end{array}
$$

The mode in each of the local areas is that of *Areas*; the controlled and active zones of an area is defined as the corresponding image in the global state; a discharge has occurred in an area, if it is in $\textit{discharge}_1$; and finally, the area is active if it is in $\textit{active}_1$.

We introduce the paragraphs related to the local state *AreaState*. Basically, we have a corresponding local action for each global action. They are identical to those presented within the process *Area* in the concrete system, and are omitted at this point for conciseness.

Next, we redefine each of the global operations. Basically, all global operations have an effect in each of the individual local states. For instance, *InitAreas* is refined below.

$$
\begin{array}{l}
\textit{InitAreas}_1 \\
\hline
\textit{AreasSt}_1' \\
\hline
\forall\, a : \textit{AreaId} \bullet (\textit{areas}'\ a).\textit{actZns} = \emptyset \\
\qquad \wedge\ (\textit{areas}'\ a).\textit{discharge} = \textit{false} \\
\qquad \wedge\ (\textit{areas}'\ a).\textit{mode} = \textit{automatic} \\
\hline
\end{array}
$$

The proof of the simulations are simple, but long. As before, for the main action, we rely on the fact that forwards simulation distributes through action constructors. The new actions have the same structure as the original ones, but use new schema actions.

$$
\begin{array}{l}
\textit{StartAreas}_1 \mathrel{\widehat=} \textit{switch} \rightarrow \textit{InitAreas}_1;\ \textit{AreasCycle}_1 \\
\textit{AreasCycle}_1 \mathrel{\widehat=}\ \textit{actuatorsR} \rightarrow \textit{SwitchAreas2Auto}_1; \textit{ResetAreas}_1 \\
\qquad \Box\ \textit{switchM?nm} \rightarrow \textit{SwitchAreasMode}_1;\ \textit{AreasCycle}_1 \\
\qquad \Box\ \textit{det?nz} \rightarrow \textit{ActivateZoneAS}_1; \\
\qquad\qquad (\forall\, a : \textit{AreaId} \bullet (\textit{areas}\ a).\textit{mode} = \textit{automatic})\ \& \\
\qquad\qquad\qquad \textit{AutoAreas}_1 \\
\qquad\qquad \Box\ (\forall\, a : \textit{AreaId} \bullet (\textit{areas}\ a).\textit{mode} = \textit{manual})\ \& \\
\qquad\qquad\qquad \textit{ManualAreas}_1
\end{array}
$$

Since all the output and input values are not changed, in the application of Law A2 we only rely on distribution. On the other hand, all the guards are changed. Both provisos raised by Law A3 need to be proved. For instance, to prove the refinement of $\textit{AreasCycle}_1$ we need the following lemma.

Lemma 4. *For any Mode m,*

$$\forall\, AreasSt;\ AreasSt_1 \bullet RetrieveAreas \Rightarrow \\ mode_A = M \Leftrightarrow \forall\, a : AreaId \bullet (areas\ a).mode = M$$

Proof. The proof of this lemma follows from predicate calculus, using the retrieve relation *RetrieveAreas* to relate $mode_A$ with each individual area's *mode*. □

The main action of the areas, $Areas_1$, is the simulation of the original action.

$\bullet\ StartAreas_1$ **end**

This concludes this data refinement step.

Action Refinement In order to apply a process refinement that splits the *Areas* process into individual areas, we redefine each of the paragraphs within the processes areas as a promotion of the corresponding original one.

The local paragraphs and the global state remain unchanged. However, a promotion schema is introduced; it relates the local state to the global one.

Promotion

$$\Delta AreasSt_1;\ \Delta AreaState;\ id? : AreaId$$

$$\theta AreaState = areas\ id? \wedge areas' = areas \oplus \{id? \mapsto \theta AreaState'\}$$

The global operations are refined to a definition in terms of the corresponding local operations. For instance, the initialisation is refined as follows.

$$InitAreas_1 \mathrel{\widehat=} \forall\, id? : AreaId \bullet InitArea \wedge Promotion$$

This can be proved using the action refinement laws presented in [12]. The redefinition of the remaining operations are trivially similar and omitted here.

The function $\mathbf{promote_2}$ promotes a given *Circus* action. The promotion of schemas is as in Z, and the promotion of *Skip*, *Stop*, *Chaos*, and channels do not change them.

$$\mathbf{promote_2}(c.e \rightarrow A) \mathrel{\widehat=} c.\mathbf{promote_2}(e) \rightarrow \mathbf{promote_2}(A)$$

References to the local components have to become references to the corresponding component in the global state; all other references remain unchanged. An implicit parameter is a function f that maps indexes to instances of the local state. Another implicit parameter is the index i that identifies an instance of the local state in the global state.

$\mathbf{promote_2}(x) \mathrel{\widehat=} (f\ i).x$ provided x is a component of $L.st$

$\mathbf{promote_2}(x) \mathrel{\widehat=} x$ provided x is not a component of $L.st$

This function is very similar to the function **promote** presented in [2]; however, it does not promote channels as the original one does.

Each action is defined as an iterated parallelism of the promotion of the corresponding local operation, but substituting the area *id* by the indexing variable i. Each branch of the parallelism may change its corresponding local state $areas\ i$; the remaining branches j, such that $j \neq i$, may change the remaining local states $areas\ j$. For instance, the actions $StartAreas_1$ and $AreasCycle_1$ can be rewritten as follows.

$$\begin{array}{rl} StartAreas_2 \mathrel{\widehat=} & \| \, i : AreaId \, [\![\, \theta\,(areas\ i) \mid \Sigma_{areas} \mid \bigcup_{j:AreaId \mid j \neq i} \theta\,(areas\ j) \,]\!] \bullet \\ & (\mathbf{promote_2}\ StartArea)\,[id, id? := i, i] \end{array}$$

The remaining actions are rewritten in a very similar way. Finally, we replace the main action.

$\bullet\ StartAreas_2$ **end**

Since $StartAreas_1$ and $StartAreas_2$ use mutual recursion, we use Theorem 1 again.

Process Refinement This last process split needs a new process refinement law. Law 31 presented below applies to processes containing a local and a global state *LState* and *GState*, local paragraphs that do not affect the global state, a promotion schema, and global paragraphs expressed in terms of the promotion of local paragraphs to the global state using iterated parallelism. The operation $L.pps \uparrow GState$ conjoins each schema expression in the paragraphs $L.pps$ with $\Xi GState$; this means that they do not change the components of *GState*. The results of this application are two processes: a local process L parametrised by an identifier *id* and a global process G defined as an iterated parallelism of local processes.

Law 31

process $G \mathrel{\widehat=}$ **begin**
$LState \mathrel{\widehat=} [\, id : Range;\ comps \mid pred_l \,]$
state $GState \mathrel{\widehat=}$
$[\, f : Range \rightarrow LState \mid \forall j : \mathrm{dom}\, f \bullet (f\ j).id = j \land pred_g \,]$
$L.schema_j \uparrow GState$
$L.action_k \uparrow GState$
$L.act \uparrow GState$

Promotion

$\Delta LState;\ \Delta GState;\ id? : Range$

$\theta LState = f\ \ id? \land f' = f \oplus \{id? \mapsto \theta LState'\}$

$$
\begin{array}{l}
\quad G.schema_j \mathrel{\widehat=} \forall\, id? : Range \bullet L.schema_j \wedge Promotion \\
\quad G.action_k \mathrel{\widehat=} \ \parallel i : Range \llbracket\, \theta\,(f\ i) \mid cs \mid \bigcup_{j:Range \mid j \neq i} \theta\,(f\ j) \rrbracket \bullet \\
\qquad\qquad\qquad (\mathbf{promote_2}\ L.action_k)\,[id, id? := i, i] \\
\quad G.act \mathrel{\widehat=} \ \parallel i : Range \llbracket\, \theta\,(f\ i) \mid cs \mid \bigcup_{j:Range \mid j \neq i} \theta\,(f\ j) \rrbracket \bullet \\
\qquad\qquad (\mathbf{promote_2}\ L.act)\,[id, id? := i, i] \\
\quad \bullet\ G.act\ \mathbf{end} \\
= \mathbf{process}\ L \mathrel{\widehat=} \ (id : Range \bullet \mathbf{begin\ state}\ LState \mathrel{\widehat=} [\, comps \mid pred_l\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad L.schema_j\ \ L.action_k\ \ \bullet L.act \\
\qquad\qquad\qquad\qquad \mathbf{end}) \\
\quad \mathbf{process}\ G \mathrel{\widehat=} \ \parallel id : Range \llbracket cs \rrbracket \bullet L(id)
\end{array}
$$

We can apply this law to $Areas_1$ in order to express the *Areas* process as the following parallelism of individual *Area* processes.

$$
\mathbf{process}\ ConcreteAreas \mathrel{\widehat=} \ \parallel id : AreaId \llbracket\, \Sigma_{areas} \rrbracket \bullet Area(id)
$$

The *Area* definition corresponds to that in the concrete system.

4 Conclusions

In this work, we present a development of a case study on the *Circus* refinement calculus. Using the refinement strategy presented in [2], we derive a distributed fire protection system from an abstract centralised specification. The result of the refinement presented here does not involve only executable constructs; additional simple schema refinements using [15] were omitted here. Our case study has motivated the proposal of new refinement laws; some of them can be found in Appendix B. There are more than fifty new laws, including process refinement laws. Their definitions can be found in [12]. Furthermore, some laws presented in [2] were found to be incorrect and corrected here. For instance, Law B15 did not have any proviso in its original version in [2].

Refinement has been studied for combinations of Object-Z and CSP [16]; however, as far as we know, nothing has been proposed in a calculational style like ours. In [17], Olderog presents a stepwise refinement for action systems, in which most refinement steps involve sequential refinements; the decomposition of atomic actions introduces parallelism. The main difference of action systems formalism and *Circus* is that, using CSP operators, *Circus* has a much richer control flow than the flat structure of action systems, where auxiliary variables simulating program counters guarantee the proper sequencing of actions.

The development of programs is supported by a design calculus for occam-like [18] communicating programs in [19]; semantics of programs and specifications are presented in a uniform predicative style, which is close to that used in the unifying theories of programming. This work is another source of inspiration for *Circus* refinement laws.

In this paper, we show that, using *Circus*, we were able to specify elegantly both behavioural and data aspects of an industrial scale application. The refinement strategy presented in [2] was also proved to be applicable to large systems. In our case study, the development consists of three iterations: the first one splits the system into a system controller and the sensors. In the second iteration, the control is subdivided into two different controllers: one for the system and one for the display. Finally, the third iteration splits the sensors into individual processes, one for each area.

All the laws presented in [2] and [12] are currently being proved using the theorem prover ProofPower-Z. These proofs make the basis for a tool that supports our refinement strategy and the application of a considerable subset of the existing refinement laws of *Circus*. By providing this tool, we intend to transform the *Circus* refinement calculus into a largely used development method in industry.

A Existing Refinement Laws

Simulation Laws

Law A1 $ASExp \preceq CSExp$
provided

- $\forall P_1.st;\ P_2.st;\ L \bullet R \land \text{pre } ASExp \Rightarrow \text{pre } CSExp$
- $\forall P_1.st;\ P_2.st;\ P_2.st';\ L \bullet$
  $R \land \text{pre } ASExp \land CSExp \Rightarrow (\exists P_1.st';\ L' \bullet R' \land ASExp)$

Law A2 $c!ae \rightarrow A_1 \preceq c!ce \rightarrow A_2$
provided $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow ae = ce\ and\ A_1 \preceq A_2.$

Law A3 $ag\ \&\ A_1 \preceq cg\ \&\ A_2$
provided $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow (ag \Leftrightarrow cg)\ and\ A_1 \preceq A_2.$

Action Refinement Laws

Law A4 $\{\, g \,\};\ A = \{\, g \,\};\ g\ \&\ A$

Law A5 $g_1\ \&\ (g_2\ \&\ A) = (g_1 \land g_2)\ \&\ A$

Law A6 $g_2\ \&\ A \sqsubseteq_{\mathcal{A}} g_3\ \&\ A$ **provided** $g_2 \Rightarrow g_3$

Law A7 $\{p\};\ (A_1 \Box A_2) = (\{p\};\ A_1) \Box (\{p\};\ A_2)$

Law A8 $\{\, g_1 \,\};\ (g_2\ \&\ A) = \{\, g_1 \,\};\ (g_3\ \&\ A)$ **provided** $g_1 \Rightarrow (g_2 \Leftrightarrow g_3)$

In the following law we refer to a predicate ass'. In general, for any predicate p, the predicate p' is formed by dashing all its free undecorated variables. We consider an arbitrary schema that specifies an action in *Circus*: it acts on a state St and, optionally, has input variables $i?$ of type T_i, and output variables $o!$ of type T_o.

Law A9

$$[\Delta St;\ i? : T_i;\ o! : T_o \mid p \wedge ass'] \\ = \\ [\Delta St;\ i? : T_i;\ o! : T_o \mid p \wedge ass'];\ \{ass\}$$

Law A10 $\{p\} \sqsubseteq_{\mathcal{A}} \mathit{Skip}$

Law A11 $(A_1;\ A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3 = A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)$
provided

- $\mathit{initials}(A_3) \subseteq cs$;
- $cs \cap \mathit{usedC}(A_1) = \emptyset$;
- $\mathit{wrtV}(A_1) \cap \mathit{usedV}(A_3) = \emptyset$

Law A12 $(A_1 \llbracket cs \rrbracket A_2) \mathbin{\Box} (B_1 \llbracket cs \rrbracket B_2) = (A_1 \mathbin{\Box} B_1) \llbracket cs \rrbracket (A_2 \mathbin{\Box} B_2)$
provided $A_1 \llbracket cs \rrbracket B_2 = A_2 \llbracket cs \rrbracket B_1 = \mathit{Stop}$

Law A13 $A_1 \llbracket cs \rrbracket (A_2 \mathbin{\Box} A_3) = (A_1 \llbracket cs \rrbracket A_2) \mathbin{\Box} (A_1 \llbracket cs \rrbracket A_3)$
provided *initials*$(A_1) \subseteq cs$ *and* A_1 *is deterministic*

Law A14 $c \rightarrow (A_1 \llbracket cs \rrbracket A_2) = (c \rightarrow A_1) \llbracket ns_1 \mid cs \cup \{\!| c |\!\} \mid ns_2 \rrbracket (c \rightarrow A_2)$
syntactic restriction $c \notin \mathit{usedC}(A_1) \cup \mathit{usedC}(A_2)$ *or* $c \in cs$

Law A15 $F(A \setminus cs) = F(A) \setminus cs$ **provided** $cs \cap \mathit{usedC}(F(\_)) = \emptyset$

Law A16 $\mathit{Skip};\ A = A = A;\ \mathit{Skip}$

Law A17

$$[\Delta S_1;\ \Delta S_2;\ i? : T \mid \mathit{preS}_1 \wedge \mathit{preS}_2 \wedge \mathit{CS}_1 \wedge \mathit{CS}_2] \\ = \\ [\Delta S_1;\ \Xi S_2;\ i? : T \mid \mathit{preS}_1 \wedge \mathit{CS}_1];\ [\Xi S_1;\ \Delta S_2;\ i? : T \mid \mathit{preS}_2 \wedge \mathit{CS}_2]$$

syntactic restrictions

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$
- $FV(\mathit{preS}_1) \subseteq \alpha(S_1) \cup \{i?\}$ *and* $FV(\mathit{preS}_2) \subseteq \alpha(S_2) \cup \{i?\}$
- $DFV(\mathit{CS}_1) \subseteq \alpha(S_1')$ *and* $DFV(\mathit{CS}_2) \subseteq \alpha(S_2')$
- $UDFV(\mathit{CS}_2) \cap DFV(\mathit{CS}_1) = \emptyset$.

Process Refinement Laws

Law A18 *Let qd and rd stand for the declarations of the processes Q and R, determined by Q.st, Q.pps, and Q.act, and R.st, R.pps, and R.act, respectively, and pd stand for the process declaration above. Then*

$$pd = (qd\ rd\ \textbf{process}\ P \mathrel{\widehat=} F(Q, R))$$

provided *Q.pps and R.pps are disjoint with respect to R.st and Q.st.*

B New Refinement Laws.

Action Refinement Laws.

Law B1 $c \rightarrow A = (c \rightarrow \mathit{Skip}); A$

Law B2 $c?x \rightarrow (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (c?x \rightarrow A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c?x \rightarrow A_2)$
provided $c \notin usedC(A_1) \cup usedC(A_2)$ *or* $c \in cs$

Law B3

$$[\, S_1'; \; S_2' \mid preS_1 \wedge preS_2 \wedge CS_1 \wedge CS_2 \,]$$
$$=$$
$$[\, S_1' \mid preS_1 \wedge CS_1 \,]; \; [\, S_2' \mid preS_2 \wedge CS_2 \,]$$

provided

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$
- $FV(preS_1) \subseteq \alpha(S_1)$ *and* $FV(preS_2) \subseteq \alpha(S_2)$
- $DFV(CS_1) \subseteq \alpha(S_1')$ *and* $DFV(CS_2) \subseteq \alpha(S_2')$
- $UDFV(CS_2) \cap DFV(CS_1) = \emptyset$

Law B4 $\Box_i \, g_i \;\&\; (A_i \llbracket \, ns_1 \mid cs \mid ns_2 \, \rrbracket \, A) = (\Box_i \, g_i \;\&\; A_i) \llbracket \, ns_1 \mid cs \mid ns_2 \, \rrbracket \, A$
provided $initials(A) \subseteq cs$

Law B5 $(g_1 \wedge g_2) \;\&\; (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (g_1 \;\&\; A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \;\&\; A_2)$
provided $g_1 \Leftrightarrow g_2$ *or* $initials(A_1) \cup initials(A_2) \subseteq cs$

In the following law we refer to a predicate $assump'$.

Law B6 $[State' \mid p \wedge assump'] = [State' \mid p \wedge assump']; \; \{assump\}$

Law B7 $\{g_1\} \sqsubseteq_{\mathcal{A}} \{g_2\}$ **provided** $g_1 \Rightarrow g_2$

Law B8 $\mu \, P \bullet V(P) \sqsubseteq_{\mathcal{A}} \mu \, P \bullet V(P)[\{g\}; F_i(P)/F_i(P)]$
provided $\{g\}; \; (F(P) \; before \; X_i) \sqsubseteq_{\mathcal{A}} (F(P) \; before \; X_i); \; \{g\}$ *for all* $F(P)$ *in* $V(P)$
where

$P = X_1, \ldots, X_n$
$V(P) = F_1(X_1, \ldots, X_n), \ldots, F_n(X_1, \ldots, X_n)$
$V(P)[exp/F_i(P)]$ *express the substitution of the* i*-th element of the vector* $V(P)$ *by the expression* exp

Law B9 $\{g\}; \; c!x \rightarrow A = c!x \rightarrow \{g\}; \; A$

Law B10 $\{g\}; \; c?x \rightarrow A = c?x \rightarrow \{g\}; \; A$ **provided** $x \notin FV(g)$

Law B11 $\{g\}; c \rightarrow A = c \rightarrow \{g\}; A$

Law B12 $\{g\}; [d \mid p] = [d \mid p]; \{g\}$ **provided** $g \wedge p \Rightarrow g'$

Law B13

$$(\Box_i\, g_i \,\&\, SExp_i); (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$$
$$\sqsubseteq_{\mathcal{A}}$$
$$((\Box_i\, g_i \,\&\, SExp_i); A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$$

provided

- $\bigcup_i wrtV(SExp_i) \subseteq ns_1 \cup ns_1'$
- $\bigcup_i wrtV(SExp_i) \cap usedV(A_2) = \emptyset$

Law B14 $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_2 \llbracket ns_2 \mid cs \mid ns_1 \rrbracket A_1$

Law B15 $A \llbracket cs \rrbracket Stop = Stop \llbracket cs \rrbracket A = Stop$ **provided** $initials(A) \subseteq cs$

References

1. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *circus*. In L Eriksson and PA Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *LNCS*, pages 451–470. Springer-Verlag, unknown 2002.
2. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
3. J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof.* Prentice-Hall, 1996.
4. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.
5. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
6. C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.
7. A. J. Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-perspective Systems.* PhD thesis, University of Teeside, School of Computing and Mathematics, 1996.
8. K. Taguchi and K. Araki. The state-based ccs semantics for concurrent z specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283 – 292. IEEE, 1997.
9. A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. In D. Gollmann, editor, *ESORICS 94*, volume 1214 of *LNCS*, pages 33 – 54. Springer-Verlag, 1994.
10. C. Fischer. Csp-oz: a combination of object-z and csp. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423 – 438. Chapman & Hall, 1997.
11. Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1994.
12. M. V. M. Oliveira. The development of a fire control system in *circus*. Technical report, University of York, Department of Computer Science, University of York, York, UK, May 2004. At http://www.cs.york.ac.uk/˜marcel/circus/fcs.pdf.
13. Data sheet mpe.130. At http://www.cs.york.ac.uk/˜marcel/circus/mpe130.html.

14. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
15. A. L. C. Cavalcanti and J. C. P. Woodcock. Zrc - a refinement calculus for z. *Formal Aspects of Computing*, 10(3):267 – 289, 1999.
16. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.
17. R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2-3):133 – 180, 1990.
18. G. Jones and M. Goldsmith. *Programming in occam 2*. Prentice-Hall, 1988.
19. E. R. Olderog. Towards a design calculus for communicating programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR'91: Proc. of the 2nd International Conference on Concurrency Theory*, pages 61–77. Springer, Berlin, Heidelberg, 1991.