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 \texttt{channel } [T] \texttt{c : T} declares a family of channels \texttt{c}. In this declaration, \texttt{[T]} is a parameter used to determine the type of the values that are communicated through channel \texttt{c}. We may introduce sets of channels in a \texttt{chanset} paragraph.

Processes may be defined explicitly or in terms of other processes (compound processes). An explicit process definition is delimited by the keywords \texttt{begin} and \texttt{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 \texttt{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 (\texttt{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: \texttt{Skip}, \texttt{Stop}, and \texttt{Chaos}. The prefixing operator is standard, but a guard construction may be associated with it. For instance, given a Z predicate \texttt{p}, if the condition \texttt{p} is \texttt{true}, the action \texttt{p & c?x → A} inputs a value through channel \texttt{c} and assigns it to the variable \texttt{x}, and then behaves like \texttt{A}, which has
the variable \( x \) in scope. If, however, the condition \( p \) is \( \text{false} \), 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 \( A_1 \parallel [ns_1 \mid cs \mid ns_2] \parallel A_2 \), the actions \( A_1 \) and \( A_2 \) synchronise on the channels in set \( cs \), unlike \texttt{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 \textit{Circus} [1]; only those that are used in this paper are described here.

### 2.2 Refinement Strategy

A refinement strategy for \textit{Circus} is presented [2]. It is based on laws of simulation, a technique used to prove data refinement in \textit{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 \textit{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 \( S_0 \), and actions \( ActA_1, \ldots, ActA_k \); 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 \textit{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.

<table>
<thead>
<tr>
<th>System State</th>
<th>Abst. FC Action</th>
<th>Conc. FC Action</th>
<th>Conc. Area Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>fireSysStart</td>
<td>AbstractFireSysStart</td>
<td>FireSysStart</td>
<td>StartArea</td>
</tr>
<tr>
<td>fireSys</td>
<td>AbstractFireSys</td>
<td>FireSys</td>
<td>AreaCycle</td>
</tr>
<tr>
<td>auto</td>
<td>AbstractAuto</td>
<td>Auto</td>
<td>AutoArea</td>
</tr>
<tr>
<td>reset</td>
<td>AbstractReset</td>
<td>Reset</td>
<td>ResetArea</td>
</tr>
<tr>
<td>countdown</td>
<td>AbstractCountdown</td>
<td>Countdown</td>
<td>WaitingDischarge</td>
</tr>
<tr>
<td>discharge</td>
<td>AbstractDischarge</td>
<td>Discharge</td>
<td>WaitingDischarge</td>
</tr>
<tr>
<td>fireSysD</td>
<td>AbstractFireSysD</td>
<td>FireSysD</td>
<td>AreaD</td>
</tr>
<tr>
<td>disabled</td>
<td>AbstractDisabled</td>
<td>Disabled</td>
<td>DisabledArea</td>
</tr>
</tbody>
</table>

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 state. After being switched on, its state is changed to fireSys; in this state, a fire detection yields to the state being changed to manual or auto, depending on the system mode. In the state reset, the system is waiting to be reset; in countdown, it is waiting for the clock to finish the countdown. During gas discharge, the system is on the discharge state, after which, the state is changed to fireSysD. Finally, if a fire is detected on fireSysD, the system state is changed to disabled.

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 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.

![Fig. 2. System External Channels](image-url)
3.1 Abstract Fire Control System

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

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

Fig. 3. System Types

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

Process \text{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 \text{AbstractFCSt} presented below. Z schemas can either be represented as boxes, as \text{AbstractFCSt}, or in a horizontal notation as we shall see later in this paper. \text{AbstractFCSt} is composed of five components, which are declared in the declaration part of the schema: \text{mode} indicates the mode in which the fire control is running; \text{controlZns} is a total function that maps the areas to a set that contains their controlled zones; \text{actZns} maps the areas to the zones in which a fire detection has occurred; \text{discharge} indicates in which areas a gas discharged
happened; finally, active contains the active areas identifications.

\[
\text{process AbstractFC} \equiv \text{begin} \\
\text{state} \\
\quad \text{AbstractFCSt} \\
\quad \text{mode} : \text{Mode} \\
\quad \text{controlZns, actZns} : \text{AreaId} \to \mathbb{P} \text{ZoneId} \\
\quad \text{discharge, active} : \mathbb{P} \text{AreaId} \\
\quad \forall a : \text{AreaId} \cdot \\
\quad \hspace{1em} (\text{mode} = \text{manual}) \implies a \in \text{active} \iff \#\text{actZns} a \geq 1 \\
\quad \hspace{1em} \land (\text{mode} = \text{automatic}) \implies a \in \text{active} \iff \#\text{actZns} a \geq 2 \\
\quad \hspace{1em} \land \text{actZns} a \subseteq \text{controlZns} a \land \text{controlZns} a = \text{getZones} a
\]

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.

\[
\text{InitAbstractFC} \\
\text{AbstractFCSt'} \\
\quad \text{mode'} = \text{automatic} \land \text{discharge'} = \emptyset \\
\quad \text{actZns'} = \{ a : \text{AreaId} \cdot a \mapsto \emptyset \}
\]

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

\[
\text{Schema} \equiv [x_1 : T_1 \ldots x_n : T_n | p]
\]

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

\[
\text{Schema'} \equiv [x'_1 : T_1 \ldots x'_n : T_n | p[x'_1/x_1, \ldots, x'_n/x_n]].
\]

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 \( \textit{State} \) that describes the state of a system, \( \Delta \textit{State} \) is a schema that includes both \( \textit{Schema} \) and \( \textit{Schema}' \). Furthermore, the name of input components must end with a query (?) and the name of output components must end with a shriek (!).

\[
\begin{align*}
\text{SwitchAbstractFCMode} \\
\Delta\text{AbstractFCSt}; \ nm? : \text{Mode} \\
\text{mode}' = nm? \land \text{actZns}' = \text{actZns} \land \text{discharge}' = \text{discharge}
\end{align*}
\]

\( \text{SwitchAbstractFC2Auto} \) and \( \text{SwitchAbstractFC2Dis} \) do not receive arguments; they switch the mode to \textit{automatic} and \textit{disabled}, respectively.

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

\[
\begin{align*}
\text{AbstractActivateZone} \\
\Delta\text{AbstractFCSt}; \ nz? : \text{ZoneId} \\
\text{mode}' = \text{mode} \land \text{discharge}' = \text{discharge} \\
\text{actZns}' = \text{actZns} \oplus \{ a : \text{AreaId} \mid nz? \in \text{controlZns} a \bullet a \mapsto \text{actZns} a \cup \{nz?\}\}
\end{align*}
\]

The schema \( \text{AbstractAutomaticDischarge} \) activates the discharge in the active areas, only \( \text{discharge} \) is changed. Finally, \( \text{AbstractManualDischarge} \) receives the areas in which the user wants to discharge the gas, but discharges only in those that are \textit{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 \( \text{AbstractFireSysStart} \) starts by communicating the current system state. Then, it waits for the system to be switched on through channel \( \text{switch} \), switches on the lamp \( \text{SysOnL} \), initialises the system state and, finally, behaves like action \( \text{AbstractFireSys} \).

\[
\begin{align*}
\text{AbstractFireSysStart} \equiv \text{sysSt!'fireSysStart} \rightarrow \text{switch} \rightarrow \text{lamp}[\text{LampId}].\text{sysOnL!on} \rightarrow \text{InitAbstractFC}; \text{AbstractFireSys}
\end{align*}
\]

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

AbstractFireSys \equiv
sysSt\text{FireSys}, \rightarrow
\text{switchM?nm \rightarrow SwitchAbstractFCMode; AbstractFireSys}
\text{det?nz \rightarrow AbstractActivateZone; lamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow}
\text{(mode = manual) \& AbstractManual}
\text{actuatorsR \rightarrow lamp[LampId], circFaultL!off \rightarrow}
\text{SwitchAbstractFC2Auto; AbstractReset}
\text{fault?faultId \rightarrow lamp[LampId], (getLampId faultId)!on \rightarrow}
\text{buzzer!on \rightarrow AbstractFireSys}

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

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

3.2 Refinement

![Refinement Diagram](image)

**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 efficiency. 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 : Π 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.

![Concrete Fire Control System Diagram](image)

**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 $name \equiv [ \text{declaration} \mid \text{predicate}]$.

**process FC** $\equiv$ **begin state** $FCSt \equiv [ mode_1 : Mode ]$

$InitFC \equiv [ FCSt' \mid mode_1 = \text{automatic} ]$

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.

$SwitchFCMode \equiv [ \Delta FCSt; \text{nm}? : Mode \mid mode_1 = \text{nm}? ]$

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.

$FireSysStart \equiv sysSt!fireSysStart_s \rightarrow switch \rightarrow switched \rightarrow InitFC; FireSys$

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.

$FireSys \equiv sysSt!fireSys_s \rightarrow$

$\square det?nz \rightarrow switched \rightarrow alarm!firstStage \rightarrow$

$(mode_1 = \text{manual}) \& \text{Manual}$

$\square (mode_1 = \text{automatic}) \& \text{Auto}$

$\square actuatorsR \rightarrow switched \rightarrow SwitchFC2Auto; \text{Reset}$

$\square fault?faultId \rightarrow switched \rightarrow FireSys$

• $FireSysStart$ 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 switched 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.

```
process Area ≝ (id : AreaId • begin

state

<table>
<thead>
<tr>
<th>AreaState</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode : Mode</td>
</tr>
<tr>
<td>controlZns, actZns : P ZoneId</td>
</tr>
<tr>
<td>discharge, active : Bool</td>
</tr>
</tbody>
</table>

controlZns = getZones id ∧ actZns ⊆ controlZns
(mode = automatic) ⇒ active = true ⇔ #actZns ≥ 2
(mode = manual) ⇒ active = true ⇔ #actZns ≥ 1
```

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

AreaState'

actZns' = Ø ∧ discharge' = false ∧ mode' = automatic
```

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 ≝ switch → InitArea; AreaCycle
```
During its cycle, if the \textit{actuatorsR} event occurs, the mode is switched to \textit{automatic} and the area waits to be \textit{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 \textit{AutoArea} or \textit{ManualArea}, depending on its current mode.

\[
\text{AreaCycle} \equiv \text{actuatorsR} \rightarrow \text{SwitchArea2Auto}; \text{ResetArea} \\
\quad \text{\quad switchM?nm} \rightarrow \text{SwitchAreaMode}; \text{AreaCycle} \\
\quad \text{\quad det?nz} \rightarrow \text{ActivateZone}; \\
\quad \quad (\text{mode} = \text{automatic}) \& \text{AutoArea} \\
\quad \quad \text{(mode} = \text{manual}) \& \text{ManualArea}
\]

\textbf{• StartArea end)

The main action of the process \textit{Area} is the action \textit{StartArea}.

The process \textit{ConcreteAreas} represents all the areas within the system. Basically, it is a parallel composition of all areas. They synchronise on the channel set \(\Sigma_{\text{areas}}\).

\[
\text{chanset } \Sigma_{\text{areas}} \equiv \{ \text{\textit{switch, reset, switchM, det, silence, actuatorsR,}} \\
\quad \text{\quad autoDis, manDis, anyDis, noDis, counting} \}
\]

\[
\text{process } \text{ConcreteAreas} \equiv \ || \text{id} : \text{AreaId} \| \Sigma_{\text{areas}} \| \text{Area(id)}
\]

The internal system is defined as the parallel composition of the fire control FC and the display controller \(\text{DisplayC}\). All the communications between them are hidden.

\[
\text{chanset } \Sigma_1 \equiv \{ \text{\textit{switch, reset, det, display, silence, actuatorsR, fault} \}}
\]

\[
\text{process } \text{ConcreteInternalFC} \equiv \text{FC} \| \Sigma_1 \| \text{DisplayC} \setminus \text{DisplaySync}
\]

The concrete fire control is the parallel combination of \textit{ConcreteInternalFC} and \textit{Areas}. Internal communications are again hidden.

\[
\text{chanset } \Sigma_2 \equiv \{ \text{\textit{manDis, autoDis, countdown, counting,}} \\
\quad \text{\quad gasDischarged, gasNotDischarged, anyDis, noDis} \}
\]

\[
\text{process } \text{ConcreteFC} \equiv (\text{ConcreteInternalFC} \| \Sigma_2 \| \text{Areas}) \setminus \text{GSync}
\]

In the following sections, we prove that \textit{AbstractFC} is refined by \textit{ConcreteFC}, or rather, \textit{AbstractFC} \(\subseteq\) \textit{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 $\text{mode}_A$ component indicates the mode in which the areas are running. The process AbstractFC is refined to the process $FC_1$ presented below.

$$\text{process } FC_1 \equiv \text{begin}$$

$$\text{state}$$

$$\begin{align*}
FCSt_1 &= \text{mode}_1, \text{mode}_A : \text{Mode} \\
&\text{controlZns}_1, \text{actZns}_1 : \text{AreaId} \rightarrow \wp \text{ZoneId} \\
&\text{discharge}_1, \text{active}_1 : \wp \text{AreaId} \\
\forall a : \text{AreaId} \bullet \\
&\text{(mode}_1 = \text{automatic}) \Rightarrow a \in \text{active}_1 \Leftrightarrow \#\text{actZns}_1 \ a \geq 2 \\
&\wedge \text{(mode}_1 = \text{manual}) \Rightarrow a \in \text{active}_1 \Leftrightarrow \#\text{actZns}_1 \ a \geq 1 \\
&\wedge \text{actZns}_1 \ a \subseteq \text{controlZns}_1 \ a \wedge \text{controlZns}_1 \ a = \text{getZones} \ a
\end{align*}$$

The state $FCSt_1$ is the same as that of AbstractFC, except that it includes an extra component $\text{mode}_A$. 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 $\text{RetrFC}$ relates each component in the AbstractFCSt to one in $FCSt_1$.

$$\begin{align*}
\text{RetrFC} \\
\text{AbstractFCSt}; \ FCSt_1 \\
\text{mode}_1 = \text{mode} \wedge \text{mode}_A = \text{mode} \wedge \text{controlZns}_1 = \text{controlZns} \\
\text{actZns}_1 = \text{actZns} \wedge \text{discharge}_1 = \text{discharge} \wedge \text{active}_1 = \text{active}
\end{align*}$$

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 $\text{mode}_A$ is initialised in automatic mode.

$$\begin{align*}
\text{InitFC}_1 \\
\text{FCSt}_1' \\
\text{mode}_1' = \text{automatic} \wedge \text{mode}_A' = \text{automatic} \wedge \text{discharge}' = \emptyset \\
\text{actZns}_1' = \{ a : \text{AreaId} \bullet a \mapsto \emptyset \}
\end{align*}$$

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

**Lemma 1.** $\text{InitAbstractFC} \preceq \text{InitFC}_1$
Proof. The application of Law A1 raises two proof obligations. The first one concerns the preconditions of both schemas.

\[ \forall \text{AbstractFCSt}; FCSt_1 \bullet \text{RetrFC} \land \text{pre InitAbstractFC} \Rightarrow \text{pre InitFC}_1 \]

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

\[ \forall \text{AbstractFCSt}; FCSt_1; FCSt'_1 \bullet \text{RetrFC} \land \text{pre InitAbstractFC} \land \text{InitFC}_1 \Rightarrow \exists \text{AbstractFCSt}' \bullet \text{RetrFC'} \land \text{InitAbstractFC} \]

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 \( \text{mode}_1 \) is also assigned to the new state component \( \text{mode}_A \).

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 \( \text{FireSysStart}_1 \) that simulates the action \( \text{AbstractFireSysStart} \).

\[ \text{FireSysStart}_1 \equiv \text{sysSt!} \text{fireSysStart} \rightarrow \text{switch} \rightarrow \text{lamp[LampId]} \text{sysOn!} \text{on} \rightarrow \text{InitFC}_1; \text{FireSys}_1 \]

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 \( \text{FireSys} \preceq \text{FireSys}_1 \).

\( \text{FireSysStart}_1 \) is the main action of \( FC_1 \), and we have just proved that it simulates the main action of \( AbstractFC \).

- \( \text{FireSysStart}_1 \) 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 \( \text{ConcreteFC} \equiv \text{begin} \)

The internal system state is composed only by its mode.

\[ \text{InternalFCSt} \equiv [\text{mode}_1 : \text{Mode}] \]
The remaining components are declared as components of the areas partition of the state.

AreasSt

modeA : Mode
controlZns1, actZns1 : AreaId → ℙ ZoneId
discharge1, active1 : ℙ AreaId

∀ a : AreaId •
  (modeA = automatic) ⇒ a ∈ active1 ⇔ #actZns1 a ≥ 2
  ∧ (modeA = manual) ⇒ a ∈ active1 ⇔ #actZns1 a ≥ 1
  ∧ actZns1 a ⊆ controlZns1 a ∧ controlZns1 a = getZones a

The state of FCSt1 is declared as the conjunction of the two previously defined schemas.

state FCSt1 ≡ InternalFCSt ∧ AreasSt

The first group of paragraphs access only mode1. It is initialised to automatic.

InitInternalFC ≡ [ InternalFCSt; AreasSt' | mode'1 = automatic ]

Another convention is used in the definitions that follow: for any schema Sch, Ξ Sch 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.

Ξ Schema
  Sch
  Sch' = θSch'

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

SwitchInternalFCMode ≡
  [ ΔInternalFCSt; ΞAreasSt; nm : Mode | mode'1 = nm? ]

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 mode1 and behaves like action FireSys2. 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 FireSysStart2 below.

FireSysStart2 ≡ sysSt!FireSysStarts → switch →
  lamp[LampId],sysOnL!on → InitInternalFC; FireSys2

When a synchronisation on switchM happens, only the InternalFC mode is
switched by action FireSys2. 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 mode1 to automatic, and waits to be reset. The behaviour, if any fault happens, is not changed.

\[
\text{FireSys}_2 \equiv \text{sysSt}_2!\text{fireSys}_s \rightarrow \\
\text{switchM'?nm} \rightarrow \text{SwitchInternalFCMode}; \text{FireSys}_2 \\
\quad \Box \text{det?nz} \rightarrow \text{lamp}[\text{ZoneId}], nz!\text{on} \rightarrow \text{alarm!firstStage} \rightarrow \\
\quad \quad (\text{mode}_1 = \text{manual}) \& \text{Manual}_2 \\
\quad \Box (\text{mode}_1 = \text{automatic}) \& \text{Auto}_2 \\
\quad \Box \text{actuatorsR} \rightarrow \text{lamp}[\text{LampId}], \text{circFaultL'off} \rightarrow \\
\quad \quad \text{SwitchInternalFC2Auto}; \text{Reset}_2 \\
\quad \Box \text{fault?faultId} \rightarrow \text{lamp}[\text{LampId}], (\text{getLampId faultId})!\text{on} \rightarrow \\
\quad \quad \text{buzzer!on} \rightarrow \text{FireSys}_2
\]

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.

\[
\text{InitAreas} \\
\text{AreasSt'}; \text{InternalFCSt'} \\
\quad \text{mode}_A' = \text{automatic} \land \text{discharge}_1' = \emptyset \\
\quad \text{actZns}_1' = \{ a : \text{AreaId} \cdot a \mapsto \emptyset \}
\]

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.

\[
\text{SwitchAreasMode} \\
\Delta \text{AreasSt}; \exists \text{InternalFCSt}; \text{nm?} : \text{Mode} \\
\quad \text{mode}_A' = \text{nm?} \land \text{actZns}_1' = \text{actZns}_1 \land \text{discharge}_1' = \text{discharge}_1
\]

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

\[
\text{ActivateZoneAS} \\
\Delta \text{AreasSt}; \exists \text{InternalFCSt}; \text{nz?} : \text{ZoneId} \\
\quad \text{mode}_A' = \text{mode}_A \land \text{discharge}_1' = \text{discharge}_1 \\
\quad \text{actZns}_1' = \text{actZns}_1 \oplus \{ a : \text{AreaId} \mid \text{nz?} \in \text{controlZns}_1 \cdot a \cdot a \mapsto \text{actZns}_1 \cup \{ \text{nz}? \} \}
\]

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

\[
\text{StartAreas} \equiv \text{switch} \rightarrow \text{InitAreas}; \text{AreasCycle}
\]
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.

\[
\text{AreasCycle} \triangleq \text{actuators}R \rightarrow \text{SwitchAreas2Auto}; \text{ResetAreas} \\
\quad \square \text{switchM} \rightarrow \text{SwitchAreasMode}; \text{AreasCycle} \\
\quad \square \text{det} \rightarrow \text{ActivateZoneAS} \\
\quad (\text{mode}_A = \text{automatic}) \& \text{AutoAreas} \\
\quad \square (\text{mode}_A = \text{manual}) \& \text{ManualAreas}
\]

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* and *StartAreas*. These actions actually represent the initial actions of each partition within the process. They synchronise on the channel set \( \Sigma_2 \). All the synchronisation events between the internal system and the areas are hidden in the main action.

\[
\bullet (\text{FireSysStart} \mid \text{\alpha(InternalFCSt)} \mid \Sigma_2 \mid \text{\alpha(AreasSt)} \mid \text{StartAreas}) \setminus \text{GSync}
\]

Action *FireSysStart* 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 \( \sqsubseteq_A \) represents the action refinement relation.

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

\[
N_i \triangleq F_i(N_0, \ldots, N_n)
\]

we have that:

\[
S_S \sqsubseteq_A [Y_0, \ldots, Y_n] \Leftrightarrow \left( F_0[Y_0, \ldots, Y_n/N_0, \ldots, N_n] \sqsubseteq_A Y_0, \ldots, F_n[Y_0, \ldots, Y_n/N_0, \ldots, N_n] \sqsubseteq_A Y_n \right)
\]

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

\[
FireSysStart_1 \sqsubseteq_A (FireSysStart_2 \parallel StartAreas) \setminus GSync
\]

where \(\parallel\) stands for \([\alpha(InternalFCSt) \mid \Sigma_2 \mid \alpha(AreasSt)]\). As \(FireSysStart_1\) is defined using mutual recursion, we use the Theorem 1, with \(S_S\) as the vector including all actions involved in the definition of \(FireSysStart_1\),

\[
S_S = [FireSysStart_1, FireSys_1, \ldots]
\]
to prove this refinement. The vector \([Y_0, \ldots, Y_n]\) includes

\[
(FireSysStart_2 \parallel StartAreas) \setminus GSync
\]

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.

\[
[FireSysStart_1, FireSys_1, \ldots] \sqsubseteq_A [B8]
[FireSysStart_1, \{mode_1 = mode_A\}; FireSys_1, \ldots]
\]

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.

\[
\begin{align*}
&\left[\begin{array}{c}
FireSysStart_1, \\
\{mode_1 = mode_A\}; FireSys_1, \ldots
\end{array}\right] \\
\sqsubseteq_A &\left[\begin{array}{c}
(FireSysStart_2 \parallel StartAreas) \setminus GSync, \\
(FireSys_2 \parallel AreasCycle) \setminus GSync, \ldots
\end{array}\right] \\
\Leftarrow &\left(\begin{array}{c}
FireSysStart_1[subst] \sqsubseteq_A (FireSysStart_2 \parallel StartAreas) \setminus GSync, \\
(FireSys_1[subst] \sqsubseteq_A (FireSys_2 \parallel AreasCycle) \setminus GSync, \ldots
\end{array}\right)
\end{align*}
\]

Here, \(subst\) corresponds to the following substitution.

\[
subst = \left(\begin{array}{c}
(FireSysStart_2 \parallel StartAreas) \setminus GSync, \\
(FireSys_2 \parallel AreasCycle) \setminus GSync, \ldots
\end{array}\right) / \left(\begin{array}{c}
FireSysStart_1, \\
FireSys_1, \ldots
\end{array}\right)
\]

Below, \(A_1 \sqsubseteq_A [law_1, \ldots, law_n \mid op_1 \ldots op_n]\) \(A_2\) denotes that \(A_1\) may be refined to \(A_2\) using laws \(law_1, \ldots, law_n\), if \(op_1, \ldots, op_n\) holds. Lemmas 2 and 3 prove refinements (1) and (2), respectively.
Lemma 2. (1) FireSysStart_1 [subst] ⊑_A (FireSysStart_2 || StartAreas) \ GSync

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

\[
\begin{aligned}
\text{FireSysStart}_1 [\text{subst}] &= [\text{Definition of FireSysStart}_1, \text{Definition of Substitution}] \\
\text{sysSt}!\text{fireSysStart}_s \rightarrow \text{switch} \rightarrow \text{lamp}[\text{LampId}].\text{sysOnL}!\rightarrow \\
&\quad \text{InitFC}_1; (\text{FireSys}_2 || \text{AreasCycle}) \setminus \text{GSync}
\end{aligned}
\]

First, we may expand the hiding since the channels lamp, switch, and sysSt are not in \(\text{GSync}\).

\[
\begin{aligned}
= \begin{Bmatrix}
A_{15} \{\{\text{lamp, switch, sysSt}\} \cap \text{GSync} = \emptyset\} \\
\quad (\text{sysSt}!\text{fireSysStart}_s \rightarrow \text{switch} \rightarrow \text{lamp}[\text{LampId}].\text{sysOnL}!\rightarrow \\
&\quad \text{InitFC}_1; (\text{FireSys}_2 || \text{AreasCycle})) \end{Bmatrix} \setminus \text{GSync}
\end{aligned}
\]

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.

\[
\begin{aligned}
= \begin{Bmatrix}
B_3 \\
&\quad (\text{sysSt}!\text{fireSysStart}_s \rightarrow \text{switch} \rightarrow \text{lamp}[\text{LampId}].\text{sysOnL}!\rightarrow \\
&\quad \text{InitInternalFC}; \text{InitAreas}; (\text{FireSys}_2 || \text{AreasCycle})) \end{Bmatrix} \setminus \text{GSync}
\end{aligned}
\]

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.

\[
\begin{aligned}
= \begin{Bmatrix}
B_{13}, B_{14} \\
&\quad (\text{sysSt}!\text{fireSysStart}_s \rightarrow \text{switch} \rightarrow \text{lamp}[\text{LampId}].\text{sysOnL}!\rightarrow \\
&\quad (\text{InitInternalFC}; \text{FireSys}_2 || (\text{InitAreas}; \text{AreasCycle}))) \end{Bmatrix} \setminus \text{GSync}
\end{aligned}
\]

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 \(\Sigma_2\), and lamp is not.

\[
\begin{aligned}
= \begin{Bmatrix}
A_{11i} \{\text{initials}(\text{AreasCycle}) \subseteq \Sigma_2\} \{\text{lamp} \notin \Sigma_2\} \\
\quad (\text{sysSt}!\text{fireSysStart}_s \rightarrow \text{switch} \rightarrow \\
&\quad (\text{lamp}[\text{LampId}].\text{sysOnL}!\rightarrow \\
&\quad \text{InitInternalFC}; \text{FireSys}_2 || (\text{InitAreas}; \text{AreasCycle}))) \end{Bmatrix} \setminus \text{GSync}
\end{aligned}
\]
Now, $switch$ may be distributed over the parallelism because it is in $\Sigma_2$.

$$= [A14] \{ switch \in \Sigma_2 \}
\begin{align*}
sysSt! & fireSysStart_s \rightarrow \\
& \left( \\
& \begin{array}{l}
switch \rightarrow \\
lamp[LampId].\text{sysOnL}!on \rightarrow \\
\text{InitInternalFC}; \text{FireSys}_2 \\
\end{array}
\right)
\end{align*}
\] GSync

Since it is not in $\Sigma_2$, $sysSt$ may be moved to the internal system side of the parallelism.

$$= [B1, A11] \{ sysSt \notin \Sigma_2 \}
\begin{align*}
& sysSt! fireSysStart_s \rightarrow switch \rightarrow \\
& \left( \\
& \begin{array}{l}
lamp[LampId].\text{sysOnL}!on \rightarrow \\
\text{InitInternalFC}; \text{FireSys}_2 \\
\end{array}
\right) \parallel \left( \\
& \begin{array}{l}
switch \rightarrow \text{InitAreas}; \\
AreasCycle \\
\end{array}
\right)
\end{align*}
\] GSync

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

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

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

**Lemma 3.**

$$\text{(2)} \{ mode_1 = mode_A \}; FireSys_1[subst] \\implies\ [A14] \{ switch \in \Sigma_2 \}
\begin{align*}
& \begin{array}{l}
\square_A \\
(FireSys_2 \parallel AreasCycle) \setminus GSync
\end{array}
\end{align*}
\]

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

$$\text{(2)} \{ mode_1 = mode_A \}; FireSys_1[subst] \\implies\ [A14] \{ switch \in \Sigma_2 \}
\begin{align*}
& \begin{array}{l}
\square_A \\
(FireSys_2 \parallel AreasCycle) \setminus GSync
\end{array}
\end{align*}
\]

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.

\[
\{ \text{GSync} \cap \{ \text{sysSt, switchM, det, lamp, alarm, fault, buzzer, reset} \} = \emptyset \}
\]

\[
\begin{align*}
\{ \text{mode}_1 = \text{mode}_A \}; \\
\text{sysSt} \land \text{fireSys}_2 \rightarrow \\
\text{switchM} \land \text{nm} \rightarrow \text{SwitchFCMode}_1; (\text{FireSys}_2 \parallel \text{AreasCycle}) \\
\square \text{det} \land \text{nz} \rightarrow \text{ActivateZone}_1; \\
\Box \text{lamp}[\text{ZoneId}].\text{nz} \land \text{on} \rightarrow \text{alarm} \land \text{firstStage} \rightarrow \\
\{ \text{mode}_1 = \text{manual} \} \land (\text{Manual}_2 \parallel \text{ManualAreas}) \\
\Box \text{fault} \land \text{faultId} \rightarrow \text{lamp}[\text{LampId}].\text{nz} \land \text{on} \rightarrow (\text{fireSys}_2 \parallel \text{AreasCycle}) \\
\text{GSync}
\end{align*}
\]

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 \(\text{SwitchFCMode}_1\) can be written as the sequential composition of \(\text{SwitchInternalFCMode}\) and \(\text{SwitchAreasMode}\).

\[
(3) = [A17] \text{switchM} \land \text{nm} \rightarrow \text{SwitchInternalFCMode}; \text{SwitchAreasMode}; \\
(FireSys_2 \parallel \text{AreasCycle})
\]

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

\[
= [B14, B13] \\
\text{switchM} \land \text{nm} \rightarrow \\
((\text{SwitchInternalFCMode}; \text{FireSys}_2) \parallel (\text{SwitchAreasMode}; \text{AreasCycle}))
\]

Finally, as \(\text{switchM}\) is in \(\Sigma_2\), we may distribute this event over the parallelism. Here, a new law (distribution of input channels over parallelism) is used.

\[
= [B2] \{ \text{switchM} \in \Sigma_2 \} \\
(\text{switchM} \land \text{nm} \rightarrow \\
\text{SwitchInternalFCMode}; \text{FireSys}_2) \\
\parallel \\
(\text{switchM} \land \text{nm} \rightarrow \\
\text{SwitchAreasMode}; \text{AreasCycle})
\]

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

\[
(4) \subseteq_A [B9, A7, A10, A16, B10, B12] \\
\text{det} \land \text{nz} \rightarrow \text{ActivateZone}_1; \text{lamp}[\text{ZoneId}].\text{nz} \land \text{on} \rightarrow \text{alarm} \land \text{firstStage} \rightarrow \\
\{ \text{mode}_1 = \text{mode}_A \}; (\text{mode}_1 = \text{manual}) \land (\text{Manual}_2 \parallel \text{ManualAreas}) \\
\square \{ \text{mode}_1 = \text{mode}_A \}; (\text{mode}_1 = \text{automatic}) \land (\text{Auto}_2 \parallel \text{AutoAreas})
Next, we use the assumption to change the guards.

$$= [A8]$$
$$\text{det}\!\?\text{nz} \rightarrow \text{ActivateZone}_1; \text{lamp}[\text{ZoneId}], \text{nz!on} \rightarrow \text{alarm!firstStage} \rightarrow$$
$$\{\text{mode}_1 = \text{mode}_A\};$$
$$\quad (\text{mode}_1 = \text{manual} \land \text{mode}_A = \text{manual}) \land$$
$$\quad (\text{Manual}_2 \parallel \text{ManualAreas})$$
$$\square (\text{mode}_1 = \text{mode}_A);$$
$$\quad (\text{mode}_1 = \text{automatic} \land \text{mode}_A = \text{automatic}) \land$$
$$\quad (\text{Auto}_2 \parallel \text{AutoAreas})$$

The assumptions can then be absorbed by the guards.

$$= [A4, A5, A10, A16]$$
$$\text{det}\!\?\text{nz} \rightarrow \text{ActivateZone}_1; \text{lamp}[\text{ZoneId}], \text{nz!on} \rightarrow \text{alarm!firstStage} \rightarrow$$
$$\quad (\text{mode}_1 = \text{mode}_A \land \text{mode}_1 = \text{manual} \land \text{mode}_A = \text{manual}) \land$$
$$\quad (\text{Manual}_2 \parallel \text{ManualAreas})$$
$$\square (\text{mode}_1 = \text{mode}_A \land \text{mode}_1 = \text{automatic} \land \text{mode}_A = \text{automatic}) \land$$
$$\quad (\text{Auto}_2 \parallel \text{AutoAreas})$$

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

$$= [B5]$$
$$\text{det}\!\?\text{nz} \rightarrow \text{ActivateZone}_1; \text{lamp}[\text{ZoneId}], \text{nz!on} \rightarrow \text{alarm!firstStage} \rightarrow$$
$$\quad \left( \begin{array}{c}
\text{mode}_1 = \text{mode}_A \\
\text{mode}_1 = \text{manual}
\end{array} \right) \land$$
$$\quad \left( \begin{array}{c}
\text{mode}_1 = \text{mode}_A \\
\text{mode}_1 = \text{automatic}
\end{array} \right) \land$$
$$\quad \left( \begin{array}{c}
\text{Manual}_2 \\
\text{Auto}_2
\end{array} \right)$$
$$\parallel$$
$$\quad \left( \begin{array}{c}
\text{mode}_1 = \text{mode}_A \\
\text{mode}_1 = \text{manual}
\end{array} \right) \land$$
$$\quad \left( \begin{array}{c}
\text{mode}_1 = \text{mode}_A \\
\text{mode}_1 = \text{automatic}
\end{array} \right) \land$$
$$\quad \left( \begin{array}{c}
\text{ManualAreas} \\
\text{AutoAreas}
\end{array} \right)$$

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

$$= [A12, A6]$$
$$\text{det}\!\?\text{nz} \rightarrow \text{ActivateZone}_1; \text{lamp}[\text{ZoneId}], \text{nz!on} \rightarrow \text{alarm!firstStage} \rightarrow$$
$$\quad \left( \begin{array}{c}
\text{mode}_1 = \text{manual} \land \text{Manual}_2
\end{array} \right)$$
$$\lor$$
$$\quad \left( \begin{array}{c}
\text{mode}_A = \text{manual} \land \text{ManualAreas}
\end{array} \right)$$
$$\lor$$
$$\quad \left( \begin{array}{c}
\text{mode}_A = \text{automatic} \land \text{AutoAreas}
\end{array} \right)$$

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 \text{ManualAreas} and
AutoAreas are in $\Sigma_2$, and alarm and lamp are not.

$$= [B1, A11]\{\text{initials}(\text{ManualAreas}) \cup \text{initials}(\text{AutoAreas}) \subseteq \Sigma_2\}$$

$$\{\Sigma_2 \cap \{\text{alarm, lamp}\} = \emptyset\}$$

$\text{det}\?\text{nz} \rightarrow \text{ActivateZone}_1;$

$$\begin{align*}
\text{lamp}[\text{ZoneId}], \text{nz!}\text{on} \rightarrow \\
\text{alarm!}\text{firstStage} \rightarrow \\
(\text{mode}_1 = \text{manual}) &; \\
\text{Manual}_2 \\
\Box (\text{mode}_1 = \text{automatic}) &; \\
\text{Auto}_2
\end{align*}$$

$$\parallel$$

$$\begin{align*}
(\text{mode}_A = \text{manual}) &; \\
\text{ManualAreas} \\
\Box (\text{mode}_A = \text{automatic}) &; \\
\text{AutoAreas}
\end{align*}$$

The schema $\text{ActivateZone}_1$ can easily be transformed to $\text{ActivateZoneAS}$ using the schema calculus. The resulting schema can also be distributed over the parallelism. Finally, channel $\text{det}$ can be distributed over the parallelism, since it is in $\Sigma_2$.

$$= \{\text{Schema Calculus, B14, B13, B2}\} \{\text{det} \in \Sigma_2\}$$

$\text{det}\?\text{nz} \rightarrow \text{lamp}[\text{ZoneId}], \text{nz!}\text{on} \rightarrow \\
\text{alarm!}\text{firstStage} \rightarrow \\
(\text{mode}_1 = \text{manual}) &; \\
\text{Manual}_2 \\
\Box (\text{mode}_1 = \text{automatic}) &; \\
\text{Auto}_2
\parallel$$

$$\begin{align*}
\text{det}\?\text{nz} \rightarrow \text{ActivateZoneAS}; \\
(\text{mode}_A = \text{manual}) &; \\
\text{ManualAreas} \\
\Box (\text{mode}_A = \text{automatic}) &; \\
\text{AutoAreas}
\end{align*}$$

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

$$(5, 6) = [\ldots]$$(5, 6) = [\ldots]$$

$$\begin{align*}
\text{actuatorsR} \rightarrow \\
\text{lamp}[\text{LampId}], \text{circFaultL!}\text{off} \rightarrow \\
\text{SwitchInternalFC2Auto}; \text{Reset}_2
\parallel$$

$$\begin{align*}
\text{actuatorsR} \rightarrow \\
\text{SwitchAreas2Auto}; \\
\text{ResetAreas}
\parallel$$

$$\begin{align*}
\text{fault?}\text{faultId} \rightarrow \text{lamp}[\text{LampId}], (\text{getLampId}\text{faultId})!\text{on} \rightarrow \\
\text{buzzer!}\text{on} \rightarrow \text{FireSys}_2
\parallel \text{AreasCycle}
\end{align*}$$
We are left with the external choice of parallel actions. Since the initial channels of the first three parallel actions are in the set \( \Sigma_2 \), we may apply the exchange law as follows.

\[
= \mathbf{[A12]}
\]

\[
sysSt! fireSys_2 \rightarrow
\begin{align*}
\begin{aligned}
&switchM?nm \rightarrow SwitchInternalFCMode; FireSys_2 \\
&\quad (\ \square \ det?nz \rightarrow lamp[ZoneId].nz!on \rightarrow \text{alarm}!\text{firstStage} \rightarrow \\
&\quad \quad \quad \quad \quad \text{(mode}_1 = \text{manual}) & \text{& Manual}_2 \\
&\quad \quad \quad \quad \quad \text{(mode}_1 = \text{automatic}) & \text{& Auto}_2 \\
&\quad \quad \quad \quad \quad \square \text{actuatorsR} \rightarrow \text{lamp}[\text{LampId}], \text{circFaultL}!\text{off} \rightarrow \\
&\quad \quad \quad \quad \quad \quad \text{SwitchInternalFC2Auto} & \text{; Reset}_2 \\
\end{aligned}
\end{align*}
\]

\[
\bigg| \\
\begin{aligned}
&\begin{aligned}
&\quad switchM?nm \rightarrow SwitchAreasMode; AreasCycle \\
&\quad \quad \quad \quad \quad \square \det?nz \rightarrow \text{ActivateZoneAS} \\
&\quad \quad \quad \quad \quad \quad \text{(mode}_A = \text{manual}) & \text{& ManualAreas} \\
&\quad \quad \quad \quad \quad \quad \text{(mode}_A = \text{automatic}) & \text{& AutoAreas} \\
&\quad \quad \quad \quad \quad \square \text{actuatorsR} \rightarrow SwitchAreas2Auto; \text{ResetAreas} \\
&\quad \quad \quad \quad \quad \quad \text{fault}?faultId \rightarrow \text{lamp}[\text{LampId}], \text{(getLampId faultId)!on} \rightarrow \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{buzzer}!\text{on} \rightarrow FireSys_2 \\
\end{aligned}
\end{aligned}
\bigg|
\]

\[
\text{AreasCycle}
\]

With small rearrangements, we have that the right-hand side of the first parallelism corresponds to the definition of the action \( \text{AreasCycle} \). 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 \( \text{AreasCycle} \) are in \( \Sigma_2 \), we may apply the distribution of parallelism over external choice.

\[
= \mathbf{[A13]} \{ \text{initials}(\text{AreasCycle}) \subseteq \Sigma_2 \}
\]

\[
sysSt! fireSys_2 \rightarrow
\begin{align*}
\begin{aligned}
&switchM?nm \rightarrow SwitchInternalFCMode; FireSys_2 \\
&\quad (\ \square \ det?nz \rightarrow \text{lamp}[ZoneId].nz!on \rightarrow \text{alarm}!\text{firstStage} \rightarrow \\
&\quad \quad \quad \quad \quad \text{(mode}_1 = \text{manual}) & \text{& Manual}_2 \\
&\quad \quad \quad \quad \quad \text{(mode}_1 = \text{automatic}) & \text{& Auto}_2 \\
&\quad \quad \quad \quad \quad \square \text{actuatorsR} \rightarrow \text{lamp}[\text{LampId}], \text{circFaultL}!\text{off} \rightarrow \\
&\quad \quad \quad \quad \quad \quad \text{SwitchInternalFC2Auto} & \text{; Reset}_2 \\
&\quad \quad \quad \quad \quad \quad \text{fault}?faultId \rightarrow \text{lamp}[\text{LampId}], \text{(getLampId faultId)!on} \rightarrow \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{buzzer}!\text{on} \rightarrow FireSys_2 \\
\end{aligned}
\end{align*}
\]

\[
\bigg| \\
\begin{aligned}
&\text{AreasCycle}
\end{aligned}
\bigg|
\]

Finally, we can distribute \( sysSt \) and use the definition of \( FireSys_2 \) to conclude our proof. Again, this is valid because all the initials of \( \text{AreasCycle} \) are in \( \Sigma_2 \),
and \( \text{sysSt} \) is not.

\[
\begin{align*}
&= [B1, A11] \{ \text{initials}(\text{AreasCycle}) \subseteq \Sigma_2 \} \{ \Sigma_2 \cap \{ \text{sysSt} \} = \emptyset \} \\
&\quad \text{switchM?nm} \rightarrow \text{SwitchInternalFCMode; FireSys}_2 \\
&\quad \Box \text{det?nz} \rightarrow \text{lamp}[\text{ZoneId}].\text{nz!on} \rightarrow \text{alarm!firstStage} \rightarrow \text{firstStage} \\
&\quad \text{actuatorsR} \rightarrow \text{lamp}[\text{LampId}].\text{circFaultL!off} \rightarrow \text{SwitchInternalFC2Auto; Reset}_2 \\
&\quad \Box \text{fault?faultId} \rightarrow \text{lamp}[\text{LampId}].(\text{getLampId faultId})!\text{on} \rightarrow \text{FireSys}_2 \\
&\quad \| \text{AreasCycle} \\
&= [\text{Definition of FireSys}_2] \\
&\quad (\text{FireSys}_2 \| \text{AreasCycle}) \setminus \text{GSync} \quad \Box
\end{align*}
\]

Using these lemmas, and those related to the remaining actions, which are omitted here, we prove that \( FC_1 \) is refined by \( \text{ConcreteFC} \).

Process Refinement We partitioned the state of the process \( FC_1 \) into \( \text{InternalFCSt} \) and \( \text{AreasSt} \). 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 \( \text{ConcreteFC} \) into two independent processes as follows.

\[
\text{process} \quad \text{ConcreteFC} \equiv (\text{InternalFC} \| \Sigma_2 \| \text{Areas}) \setminus \text{GSync}
\]

The \( \text{ConcreteFC} \) is redefined as the parallel composition of \( \text{InternalFC} \) and \( \text{Areas} \). Their definitions can be deduced from the definition of \( \text{ConcreteFC} \).

Second Iteration: splitting \( \text{InternalFC} \) into two controllers In this iteration, we split \( \text{InternalFC} \) into two separated partitions: the first one corresponds to the \( FC \) controller, and the other the \( \text{DisplayController} \) (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 \( \text{DisplayC} \). The fire control state is left unchanged.

\[
\text{process} \quad \text{ConcreteInternalFC} \equiv \text{begin} \\
\quad \text{FCSt} \equiv [\text{mode}_1 : \text{Mode}] \\
\]

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

\[
\text{state} \quad \text{InternalFCSt}_1 \equiv \text{FCSt}
\]

The operations over the \( \text{InternalFCSt} \) 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.

\[ \text{• (FireSysStart} \parallel \alpha(\text{FCSt}) \mid \Sigma_2 \mid \alpha(\text{DisplayCState}) \parallel \text{StartDisplay}) \backslash \text{DisplaySync} \]

We have the parallelism of action FireSysStart and StartDisplay, with the channels used exclusively for their communication hidden. Again, since FireSysStart₂, 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.

**process** ConcreteInternalFC \(\equiv\) (FC \(\parallel\) \(\Sigma_1\) \(\parallel\) DisplayC) \(\backslash\) 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 Areas** 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.

**process** Areas \(\equiv\) 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 \(id : \text{AreaId}\). 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**

\[
\begin{array}{c}
\forall a : \text{AreaId} \bullet (\text{areas} a).id = a \\
\wedge ((\text{areas} a).mode = \text{automatic}) \Rightarrow \\
(\text{areas} a).active = \text{true} \Leftrightarrow \#(\text{areas} a).actZns \geq 2 \\
\wedge ((\text{areas} a).mode = \text{manual}) \Rightarrow \\
(\text{areas} a).active = \text{true} \Leftrightarrow \#(\text{areas} a).actZns \geq 1 \\
\wedge (\text{areas} a).actZns \subseteq (\text{areas} a).controlZns \\
\wedge (\text{areas} a).controlZns = \text{getZones} a
\end{array}
\]
The retrieve relation is very simple and is defined below.

\[
\text{RetrieveAreas} \quad \text{AreasSt; AreasSt}_1
\]

\[
\forall \; a \; \text{: AreaId} \bullet (\text{areas } a).\text{mode} = \text{mode}_A
\]

\[
\land (\text{areas } a).\text{controlZns} = \text{controlZns}_1 a
\]

\[
\land (\text{areas } a).\text{actZns} = \text{actZns}_1 a
\]

\[
\land (\text{areas } a).\text{discharge} = \text{true} \Leftrightarrow a \in \text{discharge}_1
\]

\[
\land (\text{areas } a).\text{active} = \text{true} \Leftrightarrow a \in \text{active}_1
\]

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 discharge\(_1\); and finally, the area is active if it is in 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.

\[
\text{InitAreas}_1
\quad \text{AreasSt'}_1
\]

\[
\forall \; a \; \text{: AreaId} \bullet (\text{areas' } a).\text{actZns} = \emptyset
\]

\[
\land (\text{areas' } a).\text{discharge} = \text{false}
\]

\[
\land (\text{areas' } a).\text{mode} = \text{automatic}
\]

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.

\[
\text{StartAreas}_1 \equiv \; \text{switch} \rightarrow \; \text{InitAreas}_1; \; \text{AreasCycle}_1
\]

\[
\text{AreasCycle}_1 \equiv \; \text{actuatorsR} \rightarrow \; \text{SwitchAreas2Auto}_1; \text{ResetAreas}_1
\]

\[
\square \; \text{switchM?nm} \rightarrow \; \text{SwitchAreasMode}_1; \; \text{AreasCycle}_1
\]

\[
\square \; \text{det?nz} \rightarrow \; \text{ActivateZoneAS}_1;
\]

\[
(\forall \; a \; \text{: AreaId} \bullet (\text{areas } a).\text{mode} = \text{automatic}) \&
\]

\[
\text{AutoAreas}_1
\]

\[
\square \; (\forall \; a \; \text{: AreaId} \bullet (\text{areas } a).\text{mode} = \text{manual}) \&
\]

\[
\text{ManualAreas}_1
\]

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 AreasCycle\(_1\) we need the following lemma.
Lemma 4. For any Mode $m$,

\[
\forall \text{AreasSt}; \text{AreasSt}_1 \bullet \text{RetrieveAreas} \Rightarrow \text{mode}_A = M \leftrightarrow \forall a : \text{AreaId} \bullet (\text{areas } a). \text{mode} = M
\]

Proof. The proof of this lemma follows from predicate calculus, using the retrieve relation $\text{RetrieveAreas}$ to relate $\text{mode}_A$ with each individual area’s mode. □

The main action of the areas, $\text{Areas}_1$, is the simulation of the original action.

• $\text{StartAreas}_1$ end

This concludes this data refinement step.

Action Refinement In order to apply a process refinement that splits the $\text{Areas}$ 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.

\[
\begin{align*}
\Delta \text{AreasSt}; \Delta \text{AreaState}; id? : \text{AreaId} \\
\theta \text{AreaState} = \text{areas } id? \land \text{areas}' = \text{areas} \oplus \{id? \mapsto \theta \text{AreaState}'\}
\end{align*}
\]

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

\[
\text{InitAreas}_1 \triangleq \forall id? : \text{AreaId} \bullet \text{InitArea} \land \text{Promotion}
\]

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 $\text{promote}_2$ promotes a given $\text{Circus}$ action. The promotion of schemas is as in Z, and the promotion of $\text{Skip}$, $\text{Stop}$, $\text{Chaos}$, and channels do not change them.

\[
\text{promote}_2(c.e \rightarrow A) \triangleq c.\text{promote}_2(e) \rightarrow \text{promote}_2(A)
\]

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.

\[
\begin{align*}
\text{promote}_2(x) &= (f i).x & \text{provided } x \text{ is a component of } L.st \\
\text{promote}_2(x) &= x & \text{provided } x \text{ is not a component of } L.st
\end{align*}
\]

This function is very similar to the function $\text{promote}$ 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.

\[ \text{StartAreas}_2 \equiv \parallel i : \text{AreaId} \parallel (\theta(\text{areas } i)) \mid \Sigma_{\text{areas}} \mid \bigcup_{j : \text{AreaId}_{j \neq i}} \theta(\text{areas } j) \parallel \bullet \text{promote}_2 \text{StartArea}[\text{id, id} := i, i] \]

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

• \text{StartAreas}_2 \text{ end}

Since \text{StartAreas}_1 and \text{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 \text{LState} and \text{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 \text{L.pps} \uparrow \text{GState} conjoins each schema expression in the paragraphs \text{L.pps} with \Xi \text{GState}; this means that they do not change the components of \text{GState}. The results of this application are two processes: a local process \text{L} parametrised by an identifier \text{id} and a global process \text{G} defined as an iterated parallelism of local processes.

Law 31

\[
\text{process } G \triangleq \text{begin}
L\text{State } \triangleq [\text{id} : \text{Range}; \text{comps} | \text{pred}_1] \]
\text{state } G\text{State } \triangleq 
\[ [f : \text{Range} \rightarrow \text{LState} | \forall j : \text{dom } f \bullet (f j).\text{id} = j \land \text{pred}_j] \]
\text{L.schema}_{\text{k}} \uparrow \text{GState}
\text{L.act}_{\text{k}} \uparrow \text{GState}
\]

<table>
<thead>
<tr>
<th>Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Delta L\text{State}; \Delta G\text{State}; \text{id}? : \text{Range}</td>
</tr>
</tbody>
</table>

\[ \theta L\text{State} = f \text{id}? \land f' = f \oplus \{\text{id}? \mapsto \theta L\text{State}'\} \]
\[
G.\text{schema}_j \triangleq \forall id : \text{Range} \bullet L.\text{schema}_j \land \text{Promotion} \\
G.\text{action}_k \triangleq \| i : \text{Range} || \theta(f i) | cs | \bigcup_{j : \text{Range} \neq i} \theta(f j) \| \bullet \\
\text{(promote}_2 L.\text{action}_k)[id, id? := i, i] \\
G.\text{act} \triangleq \| i : \text{Range} || \theta(f i) | cs | \bigcup_{j : \text{Range} \neq i} \theta(f j) \| \bullet \\
\text{(promote}_2 L.\text{act})[id, id? := i, i] \\
\quad \bullet G.\text{act} \text{ end} \\
= \text{process } L \triangleq (id : \text{Range} \bullet \text{begin state } L.\text{State} \triangleq \| \text{comps} | \text{pred}_i \| \bullet L.\text{schema}_j \bullet L.\text{action}_k \bullet L.\text{act} \text{ end}) \\
\text{process } G \triangleq \| id : \text{Range} || cs \| L(id)
\]

We can apply this law to \(\text{Areas}_1\) in order to express the \(\text{Areas}\) process as the following parallelism of individual \(\text{Area}\) processes.

\[
\text{process } \text{ConcreteArea} \triangleq \| id : \text{AreaId} \| \sum \text{areas} \| \bullet \text{Area}(id)
\]

The \(\text{Area}\) definition corresponds to that in the concrete system.

4 Conclusions

In this work, we present a development of a case study on the \textit{Circus} refinement calculus. Using the refinement strategy presented in [2], we derive a distributed fire protection system from an abstract centralized 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 \textit{Circus} is that, using CSP operators, \textit{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 \textit{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 \textit{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 \( \text{ASExp} \preceq \text{CSExp} \)
provided

\[ \forall P_1, st_1; P_2, st_2; L \bullet R \land \text{pre ASExp} \Rightarrow \text{pre CSExp} \]
\[ \forall P_1, st_1; P_2, st_2'; L \bullet R \land \text{pre ASExp} \land \text{CSExp} \Rightarrow (\exists P_1, st_1'; L' \bullet R' \land \text{ASExp}) \]

Law A2 \( \text{c!ae} \rightarrow A_1 \preceq \text{c!ce} \rightarrow A_2 \)
provided \( \forall P_1, st_1; P_2, st_2; L \bullet R \Rightarrow \text{ae} = \text{ce} \) and \( A_1 \preceq A_2 \).

Law A3 \( \text{ag} \land A_1 \preceq \text{cg} \land A_2 \)
provided \( \forall P_1, st_1; P_2, st_2; L \bullet R \Rightarrow (\text{ag} \Leftrightarrow \text{cg}) \) and \( A_1 \preceq A_2 \).

Action Refinement Laws

Law A4 \( \{ g \}; A = \{ g \}; g \land A \)

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

Law A6 \( g_2 \land A \sqcap A \land A \) provided \( g_2 \Rightarrow g_3 \)

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

Law A8 \( \{ g_1 \}; (g_2 \land A) = \{ g_1 \}; (g_3 \land A) \) provided \( g_1 \Rightarrow (g_2 \Leftrightarrow g_3) \)

In the following law we refer to a predicate \( \text{ass}' \). 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

\[
\begin{align*}
[\Delta St; \ i? : T_1; \ o! : T_o & \ | \ p \land ass'] \\
= & \ [\Delta St; \ i? : T_1; \ o! : T_o & \ | \ p \land ass']; \ \{ass\}
\end{align*}
\]

Law A10 \{p\} ⊑ A Skip

Law A11 \((A_1; \ A_2) \parallel ns_1 \mid cs \mid ns_2 A_3 = A_1; \ (A_2 \parallel ns_1 \mid cs \mid ns_2) A_3\)

provided

- initials(A1) ⊆ cs;
- cs ∩ usedC(A1) = ∅;
- \(\text{wrtV}(A_1) \cap \text{usedV}(A_3) = ∅\)

Law A12 \((A_1 \parallel cs A_2) \Box (B_1 \parallel cs B_2) = (A_1 \Box B_1) \parallel cs (A_2 \Box B_2)\)

provided \(A_1 \parallel cs B_2 = A_2 \parallel cs B_1 = \text{Stop}\)

Law A13 \((A_1 \parallel cs A_2) \Box (A_1 \parallel cs A_2) = (A_1 \parallel cs A_2) \Box (A_1 \parallel cs A_3)\)

provided initials(A1) ⊆ cs and A1 is deterministic

Law A14 \(c → (A_1 \parallel cs A_2) = (c → A_1) \parallel (ns_1 \mid cs ∪ \{c\} \mid ns_2) (c → A_2)\)

syntactic restriction \(c ∉ \text{usedC}(A_1) ∪ \text{usedC}(A_2)\) or \(c ∈ cs\)

Law A15 \(F(A \setminus cs) = F(A) \setminus cs\) provided \(cs ∩ \text{usedC}(F(\_)) = ∅\)

Law A16 Skip; \(A = A = A; \ \text{Skip}\)

Law A17

\[
\begin{align*}
[\Delta S_1; \ \Delta S_2; \ i? : T & \mid \text{preS}_1 \land \text{preS}_2 \land \text{CS}_1 \land \text{CS}_2] \\
= & \ [\Delta S_1; \ \Xi S_2; \ i? : T \mid \text{preS}_1 \land \text{CS}_1]; \ [\Xi S_1; \ \Delta S_2; \ i? : T \mid \text{preS}_2 \land \text{CS}_2]
\end{align*}
\]

syntactic restrictions

- \(α(S_1) \cap α(S_2) = ∅\)
- \(\text{FV}(\text{preS}_1) ⊆ α(S_1) \cup \{i?\}\) and \(\text{FV}(\text{preS}_2) ⊆ α(S_2) \cup \{i?\}\)
- \(\text{DFV}(\text{CS}_1) ⊆ α(S_1)\) and \(\text{DFV}(\text{CS}_2) ⊆ α(S_2)\)
- \(\text{UDFV}(\text{CS}_2) \cap \text{DFV}(\text{CS}_1) = ∅\).

Process Refinement Laws

Law A18 Let pd 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

\(pd = (pd \ rd \ \text{process} \ P \equiv F(Q, R))\)

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 \( c \rightarrow A = (c \rightarrow \text{Skip}); A \)

Law B2 \( c?x \rightarrow (A_1[n_1 | cs | n_2]A_2) = (c?x \rightarrow A_1)[n_1 | cs | n_2](c?x \rightarrow A_2) \) provided \( c \notin \text{usedC}(A_1) \cup \text{usedC}(A_2) \) or \( c \in cs \)

Law B3

\[
[S'_1; S'_2 | \text{pre}S_1 \land \text{pre}S_2 \land CS_1 \land CS_2] = [S'_1 | \text{pre}S_1 \land CS_1]; [S'_2 | \text{pre}S_2 \land CS_2]
\]

provided

- \( \alpha(S_1) \cap \alpha(S_2) = \emptyset \)
- \( \text{FV}(\text{pre}S_1) \subseteq \alpha(S_1) \) and \( \text{FV}(\text{pre}S_2) \subseteq \alpha(S_2) \)
- \( \text{DFV}(CS_1) \subseteq \alpha(S'_1) \) and \( \text{DFV}(CS_2) \subseteq \alpha(S'_2) \)
- \( \text{UDFV}(CS_2) \cap \text{DFV}(CS_1) = \emptyset \)

Law B4 \( \Box^i g_i \land (A_1[n_1 | cs | n_2]A) = (\Box^i g_i \land A_1)[n_1 | cs | n_2]A \) provided \( \text{initials}(A) \subseteq cs \)

Law B5 \( (g_1 \land g_2) \land (A_1[n_1 | cs | n_2]A_2) = (g_1 \land A_1)[n_1 | cs | n_2]((g_2 \land A_2) \) provided \( g_i \leftrightarrow g_2 \text{ or } \text{initials}(A_1) \cup \text{initials}(A_2) \subseteq cs \)

In the following law we refer to a predicate \( \text{assump'} \).

Law B6 \( \text{State'} | p \land \text{assump'} = \text{State'} | p \land \text{assump'}; \{\text{assump}\} \)

Law B7 \( \{g_1\} \sqsubseteq_A \{g_2\} \) provided \( g_1 \Rightarrow g_2 \)

Law B8 \( \mu P \bullet V(P) \sqsubseteq_A \mu P \bullet V(P)[\{g\}; F_i(P)/F_i(P)] \) provided \( \{g\}; (F(P) \text{ before } X_1) \sqsubseteq_A (F(P) \text{ before } X_1); \{g\} \) for all \( F(P) \) in \( V(P) \)
where

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

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 \( x \notin \text{FV}(g) \)
Law B11 \( \{g\}; c \rightarrow A = c \rightarrow \{g\}; A \)

Law B12 \( \{g\}; [d \mid p] = [d \mid p]; \{g\} \text{ provided } g \land p \Rightarrow g' \)

Law B13

\[
(\square_i g_i \land SExp_i) \sqsubseteq_A (A_1 \parallel ns_1 \mid cs \mid ns_2 \parallel A_2)
\]

provided

\[
\begin{align*}
\bigcup_i \text{ wrtV}(SExp_i) & \subseteq ns_1 \cup ns_1' \\
\bigcap_i \text{ wrtV}(SExp_i) \cap \text{ usedV}(A_2) & = \emptyset
\end{align*}
\]

Law B14

\[
A_1 \parallel [ns_1 \mid cs \mid ns_2 \parallel A_2 = A_2 \parallel [ns_2 \mid cs \mid ns_1 \parallel A_1
\]

Law B15

\[
A \parallel [cs \parallel Stop = Stop \parallel [cs \parallel A = Stop \text{ provided } \text{ initials}(A) \subseteq cs
\]

References