Computer Science at Kent

A Refinement Calculus for *Circus* - Mini-Thesis

Marcel Oliveira

Technical Report No. 8-04
April 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

Abstract

Most software developments do not use any of the existing theories and formalisms. This leads to a loss of precision and correctness on the resulting softwares. Two different approaches to formal techniques have been raised in the past decades: one focus on data aspects, and the other focus on the behavioural aspects of the system.

Some combined languages have already been proposed to bring these two schools together. However, as far as we know, none of them has a related refinement calculus. Using *Circus* as the specification language, we can describe both data and control behaviour.

The objective of this work is to formalise a refinement calculus for *Circus*. A refinement strategy for *Circus*, new refinement laws and their proofs are presented. The proofs are based on an extension of the existing *Circus* semantics, which is based on the unifying theory of programming. This extension, and its mechanisation, and the proof of the laws on PowerProof are also part of this work.

We intend to provide a tool that supports the *Circus* refinement calculus. Furthermore, as an extension of the existing refinement strategy for *Circus*, we present a translation strategy for *Circus* programs. This translation strategy can be used as a guideline in the translation of *Circus* programs to Java. Furthermore, the mechanisation of this translation is also feasible.

We present a case study, a safety-critical fire protection system, that, as far as we know, is the largest case study on the *Circus* refinement calculus. We present the refinement of its abstract centralised specification to a concrete distributed one. Finally, the translation of the concrete specification of the system to Java, using our translation strategy, is also presented.

Throughout this mini-thesis, some sections, and even chapters are not written. They have not been removed from the mini-thesis on purpose. Our intention is to give an idea of the scope and the structure of our final thesis, which is discussed in details in the final chapter of this document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we present the motivation for the use of formal methods, *Circus*, and its refinement calculus. Furthermore, the objectives and an overview of the whole dissertation are also discussed.

1.1 Motivation

Most software developments do not use the already existing theories and formalisms. This lack of formalism raises difficulties in developing a relatively low cost trustworthy software, where the time of development is controllable. Milner affirms that software development theories are as important as computing theories [18]. The experience with the informal techniques is the main reason for using formal methods in the development processes.

Throughout the past decades two schools have been developing formal techniques for precise, correct, and concise software development. However, they have taken different approaches: one of them has focused on data aspects of the system, while the other one has focused on the behavioural aspects of the system.

Languages like Z [34], VDM [15], Abstract State Machines [5], and B [2], use a model-based approach, where mathematical objects from set theory form the basis of the specification. Although possible in a rather difficult and implicit fashion, specification constructs to model behavioural aspects such as choice, sequence, parallelism, and others, are not explicitly provided by any of these languages.

On the other hand, CSP [13, 26] and CCS [19] provide constructs that can be used to describe the behaviour of the system. However, they do not support a concise and elegant way to describe the data aspects of the system.

Many attempts were made in order to bring these two schools together. In these attempts, both data and behaviour aspects of the system are dealt together. Combinations of Z with CCS [11, 31], Z with CSP [27], and Object-Z with CSP [9] are some of these attempts to combine both schools. As far as we know, however, none of them has a related refinement calculus. This lack of support for refinement in a calculational style as that presented in [20] has motivated the creation of *Circus* [33, 1].

Circus characterises systems as processes, which group constructs that describe data and control behaviour. The Z notation [30] is used to define most of the data aspects, and CSP, and guarded constructs are used to define behaviour. The semantics of *Circus* is based on the unifying theories of programming [14], a framework that unifies the programming science across many different computational paradigms.

In [1], a refinement strategy for *Circus*, as well as some refinement laws, was presented. However, the verification of the laws, the proposition of a comprehensive set of refinement laws, and further case studies were left as future work.

1.2 Objectives

The main objective of this work is to provide a refinement calculus for *Circus*. As discussed above, in [1] the authors have introduced a generic refinement strategy for *Circus*. Their case study, however, refines an abstract specification to a concrete

one. We intend to go further in our strategy and case studies: the refinement from an abstract specification to a Java code is the objective of our work.

First, refinement notions for *Circus* processes and their constituent actions must be presented. In order to verify the usefulness and soundness of the set of laws proposed in [1], a more significant case study on the refinement of *Circus* programs is taken into account. This case study is a safety-critical fire protection system, and, as far as we know, it is the largest case study on the *Circus* refinement strategy. The transformation of an abstract centralised specification of this system to the Java implementation of a distributed one is in the scope of this work. A significant set of laws that raises from this case study is documented.

The proof of the existing and the new laws is another objective of this work. We intend to use an extension of the semantics of *Circus* programs, presented in [33], as a basis. In [33], a *Circus* program is represented as a Z specification. This allows us to use ProofPower [3] to mechanise the model of *Circus*, and to prove all the laws of the refinement calculus of *Circus*. Furthermore, this provides a basis for proving any property of *Circus* programs. In the end of this work, we intend to present a final reference for *Circus* semantics.

A strategy for the translation of *Circus* programs to Java is another objective of our work. Such a systematic strategy can be used as a guideline for implementing *Circus* programs, and further, as a guideline for mechanising the translation of *Circus* programs to Java.

Finally, we intend to provide a prototype of a tool that supports the refinement calculus for *Circus* presented in this work.

1.3 Outline

In Chapter 2, we present an introduction to *Circus*. Both its syntax and its semantics are presented here. As the semantics of *Circus* programs is based on the unifying theory of programming, we also present an introduction to this framework before actually presenting the semantics of *Circus*.

Chapter 3 discusses the refinement notions for *Circus* processes and their constituent actions. The simulation technique and the refinement strategy presented in [1] are also discussed in this chapter. Finally, this chapter presents some of the new refinement laws proposed in this work.

In Chapter 4, we present the aspects involving the implementation of a prototype of a tool that supports the application of the refinement calculus of *Circus*.

Chapter 5 presents a safety-critical fire protection system as our case study on the refinement calculus of *Circus*. We present its abstract centralised specification, and then, its refinement to a distributed system.

In Chapter 6, a strategy for implementing *Circus* programs in JCSP [25, 24] is

presented. First, we present a brief introduction to JCSP, a Java library that can be used to support the implementation of CSP programs in Java. Then, we present the translation strategy itself. In order to illustrate the translation strategy, we present the translation of our case study.

Finally, an overview of the contributions of our work, related works, and topics for future work are presented in Chapter 7.

Chapter 2

Circus

In this chapter we introduce *Circus*, a concurrent language which is appropriate for refinement. It is based on imperative CSP [26], and adds specification facilities in the Z [34] style. This enables both state and communications aspects to be captured in the same specification, as in [29].

Circus is a language that is not only suitable for the specification of concurrent and reactive systems; it has also a theory of refinement associated to it. Its objective is to give a sound basis to the development of concurrent and distributed system in a calculational style like that of [20].

In Section 2.1 we briefly present the syntax of *Circus*, and in Section 2.2 we present some of the semantic definitions of *Circus*.

2.1 Syntax

2.1.1 *Circus* Programs

In the same way as Z specifications, *Circus* programs are formed by a sequence of paragraphs.

$$\mathsf{Program} \quad ::= \quad \mathsf{CircusPar}^*$$

Here, $\mathsf{CircusPar}^*$ denotes a possibly empty list of elements of the syntactic category CircusPar of *Circus* paragraphs.

Each of these paragraphs can either be a Z paragraph, here denoted by the syntactic category Par, definition of channels, a channel set definition, or a process definition.

$$\begin{array}{lcl} \mathsf{CircusPar} & ::= & \mathsf{Par} \mid \textbf{channel}\ \mathsf{CDecl} \mid \textbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \\ & \mid & \mathsf{ProcessDefinition} \end{array}$$

The syntactic category Par is that of Z paragraphs defined in [30]. The syntactic category N is that of valid Z identifiers.

In the following sections, the main constructs of *Circus* are illustrated using a simple example: a small variation of an example presented in [33]. We describe a process that, when requested, outputs the Fibonacci sequence (See Figure 2.1). The process may also be restarted, in which case, it output the Fibonacci sequence from the beginning again.

2.1.2 Channel Declarations

All the channels that are used within a process must be declared. The syntactic categories Exp and Schema-Exp are those of Z expressions and schema expressions defined in [30].

$$\begin{array}{lcl} \mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\ \mathsf{SimpleCDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Exp} \mid [\mathsf{N}^+]\mathsf{N}^+ : \mathsf{Exp} \mid \mathsf{Schema\text{-}Exp} \end{array}$$

In a channel declaration, we declare the name of the channel and the type of the values it can communicate. However, if the channel does not communicate any value, but it is used only as a synchronising event, its declaration contains only its name; no type is defined.

A channel declaration may declare more than one channel of the same type. In this case, instead of a single channel name, we have a comma-separated list of channel names.

Generic channel declarations introduce a family of channels. For instance, the declaration **channel** $[T]$ $c : T$ declares a family of channels c. For every actual

$$
\begin{array}{l}
\textbf{channel}\ out : \mathbb{N} \\
\textbf{channel}\ restart \\
\textbf{chanset}\ FibAlphabet == \{\!|\ out, restart\ |\!\} \\
\\
\textbf{process}\ Fib \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ FibState \mathrel{\widehat=} [x, y : \mathbb{N}] \\
\\
\quad InitFibState \mathrel{\widehat=} [FibState' \mid x' = y' = 1] \\
\quad InitFib \mathrel{\widehat=} out!1 \rightarrow out!1 \rightarrow InitFibState \\
\quad OutFibState \mathrel{\widehat=} [\Delta FibState;\ next! : \mathbb{N} \mid next' = y' = x + y \land x' = y] \\
\quad OutFib \mathrel{\widehat=} \mu\, X \bullet \textbf{var}\ next : \mathbb{N} \bullet OutFibState; \left(\begin{array}{l} out!next \rightarrow X \\ \Box\ restart \rightarrow Skip \end{array} \right) \\
\quad FibCycle \mathrel{\widehat=} \mu\, X \bullet InitFib; OutFib; X \\
\\
\quad \bullet\ FibCycle \\
\textbf{end}
\end{array}
$$

Figure 2.1: A Fibonacci Generator

type S, we have a channel $c[S]$ that communicates values of type S. Channels can also be declared using schemas that group channel declarations, but do not have a predicate part. This follows from the fact that the only restriction that may be imposed to channels is the type it communicates.

Our example process outputs natural numbers through the channel *out*. Furthermore, it may be restarted through channel *restart*.

$$
\begin{array}{l}
\textbf{channel}\ out : \mathbb{N} \\
\textbf{channel}\ restart
\end{array}
$$

The channel *restart* is used just for synchronisation. So, it does not have a type.

2.1.3 Channel Set Declarations

We may introduce sets of previously defined channels in a **chanset** paragraph. In this case, we declare the name of the set and a channel-set expression, which determines the channels that are members of this set. The empty set of channels $\{\!|\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and expressions formed by the Z set operators are the elements of the syntactic category CSExp.

In our example, we declare the alphabet of the process as a channel set as follows.

$$
\textbf{chanset}\ FibAlphabet == \{\!|\ out, restart\ |\!\}
$$

This is not really used in our simple example, but channel sets are important to make definitions more concise.

2.1.4 Process Declarations

The declaration of a process is composed by its name and by its specification. A process is specified as a (possibly)parametrised process, or as an indexed process.

$$\mathsf{ProcessDefinition} \quad ::= \quad \textbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{ParProc} \mid \textbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{IndexProc}$$

If a process is parametrised or indexed, we first have the declaration of its parameters. The syntactic category Decl is the same as in [30]. Afterwards, following a $\bullet$, in the case of parametrised processes, or a $\odot$, in the case of indexed processes, we have the declaration of the process body (an element of the syntactic category Proc). In both cases, the parameters may be used as local variables in the definition of the process. If the process is not parametrised, we have only the definition of its body.

$$\mathsf{ParProc} \quad ::= \quad \mathsf{Decl} \bullet \mathsf{Proc} \mid \mathsf{Proc}$$

$$\mathsf{IndexProc} \quad ::= \quad \mathsf{Decl} \odot \mathsf{Proc}$$

A process may be explicitly defined, or it may be defined in terms of other processes (composed processes).

$$\mathsf{Proc} \quad ::= \quad \mathsf{ExpProc} \mid \mathsf{CompProc}$$

An explicit process definition is delimited by the keywords **begin** and **end**; it is formed by a sequence of process paragraphs and a distinguished nameless main action, which defines the process behaviour, in the end. Furthermore, in *Circus* we use the Z notation to define the state of a process. It is described as an schema paragraph, after the keyword **state**.

$$\mathsf{ExpProc} \quad ::= \quad \textbf{begin}\ \mathsf{PPar}^*\ \textbf{state}\ \mathsf{Schema\text{-}Exp}\ \mathsf{PPar}^*\ \bullet\ \mathsf{Action}\ \textbf{end}$$

The syntactic category Schema-Exp is that of Z schema expressions.

Our example in Figure 2.1 is defined in this way.

$$\textbf{process}\ \mathit{Fib} \mathrel{\widehat=} \textbf{begin state}\ \mathit{FibState} \mathrel{\widehat=} [x, y : \mathbb{N}] \ldots \bullet\ \mathit{FibCycle}\ \textbf{end}$$

The schema *FibState* describes the internal state of the process *Fib*: it contains two natural numbers x and y; the latter records the last value output, and the former records the value output before the last. The behaviour of *Fib* is described by the unnamed action after a $\bullet$. The process *Fib* behaves like the recursive action *FibCycle* that we described latter in this section.

2.1.5 Compound Processes

In *Circus*, processes may be defined in terms of other previously defined processes using the process name, CSP operators, iterated CSP operators, or indexed operators, which are particular to *Circus* specifications.

$$
\begin{array}{rcl}
\mathsf{CompProc} & ::= & \mathsf{N} \mid \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \Box \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
& \mid & \mathsf{Proc} \llbracket \mathsf{CSExp} \rrbracket \mathsf{Proc} \mid \mathsf{Proc} \mid\mid\mid \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
& \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Proc} \mid \Box\, \mathsf{Decl} \bullet \mathsf{Proc} \mid \sqcap\, \mathsf{Decl} \bullet \mathsf{Proc} \\
& \mid & \Vert\ \mathsf{Decl} \llbracket \mathsf{CSExp} \rrbracket \bullet \mathsf{Proc} \mid \mid\mid\mid\ \mathsf{Decl} \bullet \mathsf{Proc} \\
& \mid & \mathsf{ParProc}(\mathsf{Exp}^+) \\
& \mid & \mathsf{IndexProc}\lfloor \mathsf{Exp}^+ \rfloor \mid \mathsf{Proc}[\mathsf{N}^+ := \mathsf{N}^+] \\
& \mid & \fatsemi\ \mathsf{Decl} \odot \mathsf{Proc} \mid \Box\, \mathsf{Decl} \odot \mathsf{Proc} \mid \sqcap\, \mathsf{Decl} \odot \mathsf{Proc} \\
& \mid & \Vert\ \mathsf{Decl} \llbracket \mathsf{CSExp} \rrbracket \odot \mathsf{Proc} \mid \mid\mid\mid\ \mathsf{Decl} \odot \mathsf{Proc}
\end{array}
$$

Processes P_1 and P_2 can be combined in sequence using the sequence operator: $P_1;P_2$. This process executes the process P_2 after the execution of P_1 terminates. The external choice $P_1 \Box P_2$ initially offers events of both processes. The performance of the first event resolves the choice in favour of the process that performs it. Differently from the external choice, the environment has no control over the internal choice $P_1 \sqcap P_2$, in which the process internally (non-deterministically) resolves the choice.

In the parallelism operator, *Circus* follows the alphabetised approach adopted by [26], instead of that adopted by [13]: when processes are put in parallel, the set of events on which they synchronise must be explicitly specified. Events that are not listed occur independently. For instance, the process $P_1 \llbracket cs \rrbracket P_2$ synchronise on the set of events cs. Processes can also be composed in interleaving. For instance, a process that outputs the Fibonacci sequence through the channel *out* twice can be defined as follows.

$$Fib Twice \mathrel{\widehat=} Fib \mid\mid\mid Fib$$

However, an event *restart* leads to a non-deterministic choice of which *Fib* process of the interleaving actually restarts: one of the processes restarts, and the other one does not.

The event hiding operator $P \setminus cs$ is used to encapsulate the events that are in the channel set cs. This removes these events from the interface of P, which become no longer visible to the environment.

As CSP, *Circus* provides iterated operators that can be used to generalise the binary operators of sequence, external and internal choice, parallelism, and interleaving. Furthermore, we may instantiate a parametrised process by providing values for each of its parameters. For instance, we may have either $P(v)$, where $P \mathrel{\widehat=} (x : \mathbb{N} \bullet Proc)$, or $(x : \mathbb{N} \bullet Proc)(v)$. Except from sequence, all the iterated

operators are commutative and associative. For this reason, there is no concern about the order of the elements in the type of the indexing variable. However, for the sequence operator, we require this type to be a sequence. As expected, the process $⨟ x : T \bullet P(x)$ is the sequential composition of processes $P(v)$, with v taken from T, which must be a sequence, in the order that they appear.

Circus introduces a new operator that can be used to define processes. The indexed process $i : T \odot P$ behaves exactly like P, but for each channel c of P, we have a freshly named channel c_i. These channels are implicitly declared by the indexed operator, and communicate pairs of values: the first element, the index, is a value i of type T, and the second element is the a value of the original type of the channel. An indexed process P can be instantiated using the instantiation operator: $P\lfloor e \rfloor$, which behaves just like P, however, the value of the expression e is used as the first element of the pairs communicated through all the channels.

For instance, we may define a process similar to that previously defined as *FibTwice*, in order to have the same interleaved output of two Fibonacci sequences, but with an identification of which process generated each output. The indexed process *IndexFib* presented below outputs through channel *out_i* and can be restarted through channel *restart_i*.

$$IndexFib \mathrel{\widehat=} i : \{1, 2\} \odot Fib$$

Now, we may instantiate the process *IndexFib*: the process $IndexFib\lfloor 1 \rfloor$ outputs pairs through channel *out_i* whose first elements are 1 and the second elements are the values of the Fibonacci sequence. It may be restarted by sending the value 1 through the channel *restart_i*. Similarly, we have the process $IndexFib\lfloor 2 \rfloor$. Finally, the process *FibTwiceId* presented below produces an arbitrary merge of two sequences of pairs: the first element of the pairs identifies the generator and the second is a Fibonacci number.

$$FibTwiceId \mathrel{\widehat=} IndexFib\lfloor 1 \rfloor \mid\mid\mid IndexFib\lfloor 2 \rfloor$$

The renaming operator $P[oldc := newc]$ replaces all the communications that are done through channels *oldc* by communications through channels *newc*, which are implicitly declared. Usually, indexing and renaming are used in conjunction as in the redefinition of the process *FibTwice* presented below.

$$FibTwice \mathrel{\widehat=} FibTwiceId[out\_i, restart\_i := outid, restartid]$$

We may also combine instantiations of a indexed process using the iterated indexed operators of sequence, external and internal choice, parallelism, or interleaving. By way of illustration, we redefine the process *FibTwiceId* as follows.

$$FibTwiceId \mathrel{\widehat=} ||| i : \{1, 2\} \odot Fib\lfloor i \rfloor$$

The same characteristics and restrictions that apply to the iterated operators, also apply to the iterated indexed operators. For instance, the process $\fatsemi x : T \odot P\lfloor x \rfloor$ is the sequential composition of processes $P\lfloor v \rfloor$, with v taken from T, which must also be a sequence, in the order that they appear.

2.1.6 Actions

When a process is explicitly defined, besides the definitions of the state and the main action, we have in its body Z paragraphs, definitions of (parametrised)actions, and variable sets definitions; they are used to specify the main action of the process.

$$\mathsf{PPar} \quad ::= \quad \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{ParAction} \mid \mathbf{nameset}\ \mathsf{N} == \mathsf{NSExp}$$

As for channel sets, the empty set {}, variable name enumerations enclosed in { and }, and expressions formed by the Z set operators are the elements of the syntactic category NSExp.

$$\begin{array}{lcl} \mathsf{NSExp} & ::= & \{\,\} \mid \{\mathsf{N}^+\} \mid \mathsf{N} \mid \mathsf{NSExp} \cup \mathsf{NSExp} \mid \mathsf{NSExp} \cap \mathsf{NSExp} \\ & \mid & \mathsf{NSExp} \setminus \mathsf{NSExp} \end{array}$$

As processes, an action may be parametrised, in which case we have the declaration of the parameters followed by a $\bullet$, and then, the body of the action.

$$\mathsf{ParAction} \quad ::= \quad \mathsf{Decl} \bullet \mathsf{Action} \mid \mathsf{Action}$$

An action can be a schema, a guarded command, an invocation to a previous defined action, or a combination of these constructs using CSP operators.

$$\mathsf{Action} \quad ::= \quad \mathsf{Schema\text{-}Exp} \mid \mathsf{CSPAction} \mid \mathsf{Command} \mid \mathsf{N}$$

Two of the process *Fib*'s paragraphs are schemas (see Figure 2.1): *InitFibState* initialises both the state components with the value 1. It is a schema that follows the standard style of *Z* of defining initialisation schemas. The schema *OutFibState* defines the value of the local variable *next*!, which is the next value to be output by process *Fib*. It records in *next* and y the next output value $x + y$, and records in x the value of the previous output value y.

In pure Z, dash and shriek decorations are used to refer to after-state and output variables, respectively. In *Circus*, however, they may be used interchangeably. For instance, we can use either $next'$ or *next*! to refer to the after-state value of the local variable *next*.

Three primitive actions are available in *Circus*: *Skip*, *Stop*, and *Chaos*. The action *Skip* does not communicate any value or changes the state: it terminates

immediately. The action *Stop* deadlocks, and the action *Chaos* diverges. The only guarantee in both cases is that the state invariant is maintained.

$$
\begin{array}{lcl}
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \mathbin{\&} \mathsf{Action} \\
& \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
& \mid & \mathsf{Action} \llbracket \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \rrbracket \mathsf{Action} \\
& \mid & \mathsf{Action} \llbracket \mathsf{NSExp} \mid \mathsf{NSExp} \rrbracket \mathsf{Action} \\
& \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \mu\, \mathsf{N} \bullet \mathsf{Action} \mid \mathsf{ParAction}(\mathsf{Exp}^{+}) \\
& \mid & \fatsemi\, \mathsf{Decl} \bullet \mathsf{Action} \mid \Box\, \mathsf{Decl} \bullet \mathsf{Action} \mid \sqcap\, \mathsf{Decl} \bullet \mathsf{Action} \\
& \mid & \Vert\, \mathsf{Decl} \llbracket \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \rrbracket \bullet \mathsf{Action} \\
& \mid & \Vert\Vert\, \mathsf{Decl} \llbracket \mathsf{NSExp} \mid \mathsf{NSExp} \rrbracket \bullet \mathsf{Action} \\
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParameter}^{*} \\
\mathsf{CParameter} & ::= & ?\,\mathsf{N} \mid ?\,\mathsf{N} : \mathsf{Pred} \mid !\,\mathsf{Exp} \mid .\,\mathsf{Exp}
\end{array}
$$

The syntactic category Pred is that of Z predicates defined in [30].

The prefixing operator is standard. However, a guard construction may be associated to 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 action *InitFib* in Figure 2.1 exemplifies the output prefixing operator. It outputs the value 1 twice, and then, it invokes the operation *InitFibState* to initialise the state of the process.

All free variables must be in the scope of an action. All the state components are in the scope of any action within the process. Input communications introduce new variables in scope, which may not be used as targets of assignments.

The CSP operators of sequence, external and internal choice, parallelism, interleaving, and hiding may also be used to compose actions. However, differently from the level of process, at the level actions, communications and recursive definitions are also available.

In our example, the action *FibCycle* is recursively defined. It first behaves like action *InitFib*, which, as already mentioned, outputs the value 1 twice and records this by initialising the state components. Then, *FibCycle* behaves like action *OutFib*, which outputs as many numbers of the Fibonacci sequence as required. Finally, the action *FibCycle* recurses.

At the level of actions, the parallel and the interleaving operators are slightly different from that of CSP. In order to avoid conflicts in the access to the variables in scope, parallelism and interleaving of actions must declare a synchronisation channel set and two sets that partition all the variables in scope: state components, and input and local variables. In the parallelism $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, for instance,

the actions A_1 and A_2 synchronise on the channels in set cs. Besides, both A_1 and A_2 have access to the initial values of all variables in ns_1 and ns_2. However, action A_1 may only modify the values of the variables in ns_1, and, similarly, action A_2 may only modify the values of the variables in ns_2.

Parametrised actions can be instantiated: for instance, we can have the action $A(x)$, if A is a previously defined single-parametrised action; we can also have an instantiation of the form $(x : \mathbb{N} \bullet A)(x)$.

As for processes, the iterated operators for sequence, external and internal choice, parallelism, and interleaving can also be used for actions in order to generalise the corresponding operators.

Actions may also be defined using Dijkstra's guarded commands [7].

$$\begin{array}{lcl} \mathsf{Command} & ::= & \mathsf{N}^+ : [\,\mathsf{Pred}, \mathsf{Pred}\,] \mid \mathsf{N}^+ := \mathsf{Exp}^+ \\ & \mid & \textbf{if}\ \mathsf{GActions}\ \textbf{fi} \mid \textbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \\ \mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \,\Box\, \mathsf{GActions} \end{array}$$

An action can be a (multiple) assignment, or a guarded alternation. For instance, the action *InitFib* of the process *Fib* can be written as follows.

$$InitFib \mathrel{\widehat=} out!1 \rightarrow out!1 \rightarrow x, y := 1, 1$$

Variable blocks can also be used in an action specification. Finally, in the interest of supporting a calculational approach to development, an action can also be written as a specification statement in the style of Morgan's refinement calculus [20].

The action *OutFib* in our example is recursively defined. Its body has a new local variable *next* in scope. Its value is calculated by the operation *OutFibState*, and then, an external choice is given to the environment, which can choose between the next value of the sequence (*out*), or the restart of the output sequence (*restart*). If the first option is chosen, the action *OutFib* outputs the next value and recurses; if the second option is chosen, *OutFib* skips, leading to a new cycle of the process *Fib* (*FibCycle*).

The complete syntax of *Circus* is summarised in Appendix A.

2.2 Semantics

The semantic model of *Circus* was first presented in [33]. Basically, a *Circus* program is represented as a Z specification, in which the model of a process is itself a Z specification, and the model of an action is a schema. The motivation for using Z for the *Circus* semantic model is the possibility of using tools as Z-EVES [17] and ProofPower [3] to analyse the model of *Circus*.

2.2.1 The Unifying Theories of Programming

The semantic model of *Circus* is based on the Hoare & He's Unifying Theories of Programming (UTP) [14]. The UTP is a framework in which the theory of relations is used as a unifying basis for the programming science across many different computational paradigms: procedural and declarative, sequential and parallel, closely-coupled and distributed, and hardware and software. All programs, designs, and specifications are interpreted as relations between an initial observation and a single subsequent observation, which may be either an intermediate or a final observation, of the behaviour of program execution.

Common ideas, such as sequential composition, conditional, nondeterminism, and parallelism are shared by different theories. For instance, sequential composition is a relational composition, conditionals are Boolean connectives, nondeterminism is disjunction, and parallelism is a restricted form of conjunction. Miracle is interpreted as an empty relation, abortion is interpreted as the universal relation, and correctness and refinement are interpreted as inclusions of relations. All the laws of relational calculus may be used for reasoning about correctness in all theories and in all languages.

Three aspects are used to differentiate different programming languages and design calculi: the alphabet, a set of names that characterise a range of external observations of a program behaviour; the signature, which provides syntax for denoting the objects of the theory; and the healthiness conditions, which select the objects of a sub-theory from those of a more expressive theory in which it is embedded.

The alphabet of a theory collects the names within the theory that identifies observation variables that are important to describe all relevant aspects of a program behaviour. The initial observations of each of these variables are undecorated, and subsequent observations are decorated with a dash. This allows a relation to be expressed as in Z by its characteristic predicate. Table 2.1 summarises the variables that are used in the semantics of *Circus*.

In *Circus*, some combinations of these variables have interesting semantic meaning. For instance, $okay' \land wait'$ represents a non-divergent process that is waiting for some interaction with the environment; if, however, we have $okay' \land \neg\ wait'$, the non-divergent process has terminated; finally, $\neg\ okay'$ represents a divergent process.

Besides these variables, UTP also presents some other variables that may be used to represent program control, real time clock, or resource availability. For each theory, we may select its appropriate relevant variables subset.

The signature of a theory is a set of operators and atomic components of this theory: it is the syntax of the language. The smaller the signature, the simpler the proof techniques to be applied for reasoning. Signatures may vary according to its

okay	This Boolean variable indicates if the system has been properly started in a stable state, in which case its value is *true*, or not; $okay'$ means subsequent stabilisation in an observable state.
tr	This variable, whose type is a sequence of events, records all the events in which a program has engaged.
wait	This Boolean variable distinguishes the intermediate observations of waiting states from final observations of termination. In a stable intermediate state, *wait* has *true* as its value; a *false* value for *wait* indicates that the program has reached a final state.
ref	This variable describes the responsiveness properties of the process; its type is a set of events. All the events that are refused by a process before the program has started are elements of *ref*, and refused events at a later moment are referred by ref'.
v	All program variables (state components, input and local variables, and parameters) are collectively denoted by v.

Table 2.1: *Circus* Alphabet

purpose. Specification languages are least restrictive and often includes quantifiers, and all relational calculus operators. Design languages successively remove non-implementable operators. The negation is the first one to be removed. Thus, all operators are monotonic, and recursion can safely be introduced as a fixed-point operator. Finally, programming languages present only implementable operators in their signature. They are commonly defined in terms of their observable effects using the more general specification language.

Healthiness conditions can be used to test a specification or design for feasibility, and reject it, if it makes implementation impossible in the programming language that we wish to use as target. Typically, each of the external variables have a healthiness condition associated to it.

The *Circus* semantic model satisfies the eight healthiness conditions that are stated for CSP processes. They are summarised in Table 2.2: three of them characterises reactive processes in general; two of them constrain these reactive processes to be CSP ones; finally, the last three constrain even more the CSP reactive processes.

Reactive Process	**R1**	The execution of a reactive process never undoes any event that has already been performed.
	R2	The behaviour of a reactive process is oblivious of what has gone before.
	R3	Intermediate stable states do not progress.
CSP Process	**CSP1**	No prediction can be made about a process that has not started.
	CSP2	A process may not be required to abort.
	CSP3	CSP processes do not depend on the initial value of the *ref* variable when *wait* is *false*. If, however, *wait* is *true*, it must behave as **R3** requires.
	CSP4	The value of the variable ref' has no relevance after termination.
	CSP5	The refusal set must be subset closed.

Table 2.2: *Circus* Healthiness Conditions

2.2.2 *Circus* Semantic Model

In this section we present some of the definitions of the *Circus* semantic model, which is described in detail in [33].

Channel Environment

A channel environment is defined in order to store information about all the channels in scope for a given process. This environment is basically changed by any channel declaration: for each channel declaration, this environments maps the name of the channel to its type. If, however, a channel declaration contains no type, in which case the channel is not used for communication, but only as a synchronising event, its type is recorded as *Sync*. Channel set declarations are considered to be expanded by replacing references to channel sets with the sets of channels it denotes.

Process Environment

A process environment is also used: this allows a process definition to refer to other processes previously defined. This environment is defined as a sequence of pairs: the first element is the process name, and the second element is the Z specification that corresponds to the process model.

The semantic function $[\![\_]\!]^{\mathcal{PD}}$ presented below gives the meaning of a process definition as a process environment that records just the single process it declares. Fur-

thermore, new channels introduced by the semantic are also recorded in a channel environment, which is also returned by this semantic function. The syntactic category ProcessDefinition represents a *Circus* process definition, and the types *ChanEnv* and *ProcEnv*, are those of the channels and processes environments, respectively.

$$[\![\_]\!]^{\mathcal{PD}} : \mathsf{ProcessDefinition} \pfun ChanEnv \pfun ProcEnv \pfun (ChanEnv \times ProcEnv)$$
$$[\![\mathbf{process}\ N \mathrel{\widehat=} P]\!]^{\mathcal{PD}}\ \gamma\ \rho =$$
$$\quad \mathbf{let}\ Ps == [\![P]\!]^{\mathcal{P}}\ \gamma\ \rho\ \mathbf{in}\ (first\ Ps, \langle (N, second\ Ps) \rangle)$$

The semantics Ps of the process P is given by function $[\![\_]\!]^{\mathcal{P}}$, which is defined later on: it is a pair containing a channel environment and a Z specification. The semantics of the process P definition is the pair formed by the returned channel environment and the process environment that associates N to the Z specification that corresponds to the semantics of P.

Programs

The semantics of a *Circus* program is given by the following function.

$$[\![\_]\!]^{\mathcal{PROG}} : \mathsf{Program} \pfun \mathsf{ZSpecification}$$

For a given well-formed *Circus* program, it returns a Z specification.

The state of a process is described by the schema *ProcessState* described below. The type *Bool* is a given type that represents boolean values, and the given type *Event* includes the possible communications of the program.

$$ProcessState \mathrel{\widehat=} [\, trace, tr : \mathrm{seq}\, Event;\ \ ref : \mathbb{P}\, Event;\ \ okay, wait : Bool \,]$$

Besides the variables of the unifying theory model, we also have a *trace* variable, which records the events that occured since last observation.

The schema *ProcessStateObs* constraints the changes to the process state.

$$ProcessStateObs \mathrel{\widehat=} [\Delta ProcessState \mid tr\ \mathsf{prefix}\ tr' \land trace' = tr' - tr\,]$$

A process observation is valid only if the trace is increased.

Paragraphs

The semantics of each process paragraph is defined by the following semantic function.

$$[\![\_]\!]^{\mathcal{CPAR}} : \mathsf{CircusPar} \pfun ChanEnv \pfun ProcEnv \pfun$$
$$\qquad (\mathsf{ZSpecification} \times ChanEnv \times ProcEnv)$$

The syntactic category CircusPar represents a *Circus* paragraph. Each paragraph may affect the final Z specification by introducing new paragraphs, or may extend the channel environment, or, finally, may extend the process environment.

$$
\begin{array}{l}
[\![pd]\!]^{\mathcal{CPAR}}\ \gamma\ \rho = \\
\quad \textbf{let}\ pds = [\![pd]\!]^{\mathcal{PD}}\ \gamma\ \rho\ \textbf{in} \\
\quad\quad (second\ ((second\ pds).1), \gamma\ \oplus first\ pds, \rho \oplus second\ pds)
\end{array}
$$

The translation of *pd* returns a pair: the first element of this pair is a channel environment, and the second element is a process environment. The process environment that it returns (*second pds*) has just one element (.1). This element is itself a pair, whose second element is the Z specification corresponding to the process. The channel and process environments are overwritten with the corresponding environments returned by the translation of *pd*.

Z paragraphs are added to the process specification as they are. They also do not affect the channel and the process environment. A slight change is applied in order to type as *Sync* any untyped state components, which are assumed to be synchronisation events declarations. *Sync* is a given set. Channel declarations give rise to a few paragraphs in *Z* and enrich the channel environment. Finally, a process definition determines a Z specification of its model in the UTP. The process environment and possibly the channel environment are also enriched.

For a list of paragraphs, the Z specification is formed by the paragraphs corresponding to the first *Circus* paragraph, followed by the paragraphs corresponding to the rest of the list, whose semantics is taken in enriched channel and process environments that records the declarations(s) in the first *Circus* paragraphs.

$$
\begin{array}{l}
[\![\_]\!]^{\mathcal{CPARL}} : \mathsf{CircusPar}^* \pfun ChanEnv \pfun ProcEnv \pfun \mathsf{ZSpecification} \\
[\![cp]\!]^{\mathcal{CPAR}}\ \gamma\ \rho = first\ ([\![cp]\!]^{\mathcal{CPAR}}\ \gamma\ \rho) \\
[\![cp\ \ cpl]\!]^{\mathcal{CPAR}}\ \gamma\ \rho = \\
\quad \textbf{let}\ cps = [\![cp]\!]^{\mathcal{CPAR}}\ \gamma\ \rho\ \textbf{in}\ cps.1\ ([\![cpl]\!]^{\mathcal{CPAR}}\ cps.2\ cps.3)
\end{array}
$$

The possible repetition of names across different process definitions is removed by prefixing each name with the name of the process in which it is declared.

Processes

The semantics of process declarations is given by function $[\![\_]\!]^{\mathcal{P}}$, which, given a process, a channel environment, and a process environment, returns a pair: the first element is a new channel environment, and the second element is the Z specification corresponding to the process.

Processes names do not change the channel environment; as we assume that well-formed process definitions do not refer to undeclared processes, its semantics

corresponds to the Z model of the process whose name was given to the semantic function.

$$[\![\_]\!]^{\mathcal{P}} : \mathsf{Proc} \pfun \mathit{ChanEnv} \pfun \mathit{ProcEnv} \pfun (\mathit{ChanEnv} \times \mathsf{ZSpecification})$$
$$[\![N]\!]^{\mathcal{P}}\ \gamma\ \rho = (\gamma, \mathit{modelOf}\ \ N\ \ \mathit{in}\ \ \rho)$$

The expression $\mathit{modelOf}\ N\ \mathit{in}\ \rho$ corresponds to the Z model of process N is the process environment ρ.

For explicit process declarations **begin** ppars_1 **state** $\mathit{USt}\ \mathit{ppars}_2\ \bullet\ A$ **end**, we have that the semantic function is defined as a Z specification containing a schema *ProcObs* describing the observations that may be made of the process, the existing Z paragraphs as they are, and for each action, a schema constraining the process observations. Schemas that define operations are not translated as Z paragraphs, but as actions.

The process *State* includes the state components defined by the user (USt), and the components of the schema *ProcessState*, which represent the variables of the Unifying Theory.

$$\mathit{State}(\mathit{USt}) \mathrel{\widehat=} \mathit{USt} \land \mathit{ProcessState}$$

A process observation corresponds to a state change.

$$\mathit{ProcObs}(\mathit{USt}) \mathrel{\widehat=} \Delta \mathit{USt} \land \mathit{ProcessStateObs}$$

Actions

For each action $N \mathrel{\widehat=} A$ declared within a process we have a new schema $N \mathrel{\widehat=} [\![A]\!]^{\mathcal{A}}\gamma\ \mathit{USt}$. Given an action, the current channel environment, and the name of the user state, the semantic function $[\![\_]\!]^{\mathcal{A}}$ returns a schema corresponding to the given action. The schema corresponding to the main action is given a fresh name. An action may behave in three different ways: it may behave in a normal way, diverge, or not terminate.

$$[\![\_]\!]^{\mathcal{A}} : \mathsf{Action} \pfun \mathit{ChanEnv} \pfun \mathsf{N} \pfun \mathsf{Schema} - \mathsf{Exp}$$
$$[\![A]\!]^{\mathcal{A}}\ \gamma\ \ \mathit{USt} = [\![A]\!]^{\mathcal{A}_{\mathcal{N}}}\ \gamma\ \ \mathit{USt} \lor \mathit{Diverge}(\mathit{USt}) \lor\ \mathit{Wait}(\mathit{USt})$$

Divergence is characterised by the fact that *okay* is false. In this case, the only guarantee we have is that the trace is extended, which is specified by *ProcObs*.

$$\mathit{Diverge}(\mathit{USt}) \mathrel{\widehat=} [\mathit{ProcObs}(\mathit{USt}) \mid \neg\ \mathit{okay}]$$

For $\mathit{Wait}(\mathit{USt})$, we have that there is no divergence. However, the previous action has not yet terminated. In this case, the user state cannot change.

$$\mathit{Wait}(\mathit{USt}) \mathrel{\widehat=} [\Xi \mathit{State}(\mathit{USt}) \mid \mathit{okay} \land \mathit{wait}]$$

Finally, in a normal behaviour, there is no divergence, and the previous action has terminated.

$$Normal(USt) \mathrel{\widehat=} [ProcObs(USt) \mid okay \land \neg\, wait]$$

Schema expressions may be activated in a state that satisfies its precondition, or not. If the precondition is satisfied, the trace is not modified, and the operation terminates; however, if the precondition is not satisfied, the operation diverges.

$$[\![SExp]\!]^{\mathcal{A}}\, \gamma\ USt = SExp \land (OpNormal(USt) \lor OpDiverge(USt, SExp))$$

These cases are specified by *OpNormal* and *OpDiverge*, respectively.

$$OpNormal(USt) \mathrel{\widehat=} [Normal(USt) \mid trace' = \langle\rangle \land okay' \land \neg\, wait']$$
$$OpDiverge(USt, SExp) \mathrel{\widehat=}$$
$$[Normal(USt);\ SExp \lor \neg\, SExp \mid \neg \text{ pre } SExp \land \neg\, okay']$$

Possible input and output variables are put in scope by $SExp \lor \neg\, SExp$.

The action *Skip* does not change the trace, does not diverge, and terminates.

$$[\![Skip]\!]^{\mathcal{A}}\, \gamma\ USt = [Normal(USt) \land \Xi USt \mid trace' = \langle\rangle \land okay' \land \neg\, wait']$$

The action *Stop* is similar, but does not terminate. This deadlock is characterised with a *true* value for $wait'$. The action *Chaos* diverges, which is represented by a *false* value for $okay'$.

The sequence operator is defined in terms of a function sequence on $ProcObs(USt)$ as follows.

$$[\![A;B]\!]^{\mathcal{A}_{\mathcal{N}}}\, \gamma\ USt$$
$$Normal(USt)$$
$$\theta ProcObs(USt) = \theta([\![A]\!]^{\mathcal{A}}\, \gamma\ USt)\ sequence\ ([\![B]\!]^{\mathcal{A}}\, \gamma\ USt)$$

The function *sequence* returns the process observation that characterises the sequential composition of two processes observations.

$$\_\ sequence\_ : ProcObs(USt) \times ProcObs(USt) \nrightarrow ProcObs(USt)$$
$$\forall\, a, b, c : ProcObs(USt) \mid$$
$$c = a\ sequence\ b \Leftrightarrow \left(\begin{array}{l} before\ c = before\ a \\ \land\ after\ a = before\ b \\ \land\ after\ b = after\ c \end{array} \right)$$

The well-definement of a sequential composition of two process observations depends on the equality between the final state of the first with the initial state of the second, which are projected, respectively, by the functions *after* and *before* presented below.

$$after : ProcObs(USt) \rightarrow State(USt)$$
$$\forall\, ProcObs(USt) \bullet before\ \theta ProcObs(USt) = \theta State(USt)'$$

$$before : ProcObs(USt) \rightarrow State(USt)$$
$$\forall\, ProcObs(USt) \bullet before\ \theta ProcObs(USt) = \theta State(USt)$$

In [33], the semantics for communications, external and internal choice, parallelism, interleaving, hiding, recursion on actions, and commands are presented. The formalisation of the healthiness conditions (Table 2.2) can also be found in the same document.

Process Expressions

Process expressions use CSP operators to combine existing processes. The semantics of an expression P op Q, where op is any binary CSP operator, except parallelism and interleaving, can be defined as follows.

$$\begin{aligned} P\ \mathsf{op}\ Q = \mathbf{begin}\\ &\mathbf{state}\ State \mathrel{\widehat=} P.State \land Q.State\\ &P.PPar \uparrow Q.State\\ &Q.PPar \uparrow P.State\\ &\bullet\ P.Act\ \mathsf{op}\ Q.Act\\ \mathbf{end}& \end{aligned}$$

The state of the processes P and Q are denoted by $P.State$ and $Q.State$, respectively. In a similar way, the notation $P.PPar$ and $Q.PPar$ is used to refer to the paragraphs in the definitions of processes P and Q. Their main actions are denoted as $P.Act$ and $Q.Act$. The schema expressions within $P.PPar$ and $Q.PPar$ need to be lifted to work on the extended state $State$. The operation $PPar \uparrow St$ simply conjoins each schema in $PPar$ with ΞSt: actions on P are not supposed to affect the state components that are inherited from Q, and vice-versa.

As discussed in Section 2.1.6, the parallel and the interleaving operators for actions are slightly different from those used for processes: two sets that partition all the variables in scope must also be declared. In order to give a definition for parallelism and interleaving, we use a new operator $PPar \uparrow_{PAR} St$, which lifts the paragraphs in $PPar$ to the extended state by conjoining the schemas with ΔSt. The definitions of $P \llbracket\, cs \,\rrbracket\ Q$ and $P \mid\mid\mid Q$ are similar to that given above for the other binary operators. However, only the state is defined in the same way: the other paragraphs are defined using the lifting operator $\uparrow_{PAR}$, and the main action must take into account the partition of the variables in scope. The paragraphs in $P.PPar$ are allowed to apply any change to the components of $Q.State$. These

changes, however, are ignored since the components of $Q.State$ are declared, in the partition ns_2 of action $Q.Act$; and similarly for the paragraphs in $Q.PPar$. For the parallel operator, we have the following definition.

$$
\begin{array}{l}
P \,[\![\, cs \,]\!]\, Q = \textbf{begin} \\
\qquad\qquad \textbf{state}\ State \mathrel{\widehat{=}} P.State \wedge Q.State \\
\qquad\qquad P.PPar \uparrow_{PAR} Q.State \\
\qquad\qquad Q.PPar \uparrow_{PAR} P.State \\
\qquad\qquad \bullet\ P.Act \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, Q.Act \\
\qquad\quad \textbf{end}
\end{array}
$$

The definition for interleaving is very similar.

The semantics of the hiding operator is even simpler: the process paragraphs of P are included as they are; only the main action is modified to include the hiding.

$$
\begin{array}{l}
P \setminus cs = \textbf{begin} \\
\qquad\qquad \textbf{state}\ P.State \\
\qquad\qquad P.PPar \\
\qquad\qquad \bullet\ P.Act \setminus cs \\
\qquad\quad \textbf{end}
\end{array}
$$

The semantics of the other process expressions can be found in [33].

Chapter 3

Refinement: Notions and Laws

In this chapter we discuss the refinement notions for *Circus* processes and their constituent actions. The simulation technique, a refinement strategy for the development of centralised specifications into distributed implementations, and some laws presented in [1] are also discussed. Furthermore, new refinement laws are presented in this chapter.

3.1 Refinement Notions and Strategy

The central notion in the unifying theories of programming (UTP) [14] is refinement, which is expressed as an implication: an implementation P satisfies a specification S if, and only if, $[P \Rightarrow S]$, where the square brackets denote the universal quantifier over the alphabet, as in [8], which must be the same for implementation and specification. In *Circus*, the basic notion of refinement is that of actions [6].

Definition 3.1 (Action Refinement) *For actions A_1 and A_2 on the same state space, the refinement $A_1 \sqsubseteq_{\mathcal{A}} A_2$ holds if, and only if, $[A_2 \Rightarrow A_1]$.* □

For processes, since we have that the state of a process is private, we have a slightly different definition. Basically, the main action of a process defines its behaviour. For this reason, process refinement is defined in terms of actions refinement of local blocks. In the following, $P_1.State$ and $P_1.Act$ denote the local state and the main action of process P_1; similarly for process P_2.

Definition 3.2 (Process Refinement) *$P_1 \sqsubseteq_{\mathcal{P}} P_2$ if, and only if,*

$$(\exists\, P_1.State;\ P_1.State' \bullet P_1.Act\,) \sqsubseteq_{\mathcal{A}} (\,\exists\, P_2.State;\ P_2.State' \bullet P_2.Act\,) \qquad \square$$

The actions $P_1.Act$ and $P_2.Act$ may act on different states space, and so may not be comparable. Actually, we compare the actions we obtain by hiding the state components of processes P_1 and P_2, as if they were declared in a local variable block, whose semantics is given by existential quantification. We are left with a state space containing only the UTP variables *okay*, *wait*, *tr*, and *ref*.

As discussed above, the state of a process is private. This allows processes components to be changed during a refinement. This can be achieved in much the same way as we can data refine variable blocks and modules in imperative programs [21]. Two well-known techniques of data refinement in those contexts are forwards and backwards simulation [12].

In [1], the standard simulation techniques used in Z were adopted to handle processes and actions [34]. A simulation is a relation between the states of two processes that satisfies a number of properties.

Definition 3.3 (Forwards Simulation) *A forwards simulation between actions A_1 and A_2 of processes P_1 and P_2, with local state L, is a relation R between $P_1.State$, $P_2.State$, and L satisfying*

- *(Feasibility)* $\forall\, P_2.State;\ L \bullet (\exists\, P_1.State \bullet R)$
- *(Correctness)* $\forall\, P_1.State;\ P_2.State;\ P_2.State';\ L \bullet R \wedge A_2 \Rightarrow (\exists\, P_1.State';\ L' \bullet R' \wedge A_1)$

In this case, we write $A_1 \preceq_{P_1,P_2,R,L} A_2$. *A forwards simulation between* P_1 *and* P_2 *is a forwards simulation between their main actions.* □

Notice that, differently from the usual definition of forwards simulation, in Definition 3.3 no applicability requirement concerning preconditions exists. This follows from the fact that actions are total. If an action is executed outside its preconditions, it diverges; however, its behaviour is not arbitrary, since the state invariant is implicitly maintained, and arbitrary new synchronisation and communications can be observed, but no past observations are affected. Furthermore, no specific conditions is imposed on the initialisation: any state initialisation must be explicitly included in the main action.

A theorem presented in [1], and proved in [28], ensures that, if we provide a forwards simulation between two processes P_1 and P_2, we can substitute P_1 by P_2 in a *Circus* program.

Theorem 3.1 (Forwards simulation is sound) *When a forwards simulation exists between two processes* P_1 *and* P_2, *we also have that* $P_1 \sqsubseteq_{\mathcal{P}} P_2$. □

The definition of backwards simulation is very similar to that of forwards simulation.

Definition 3.4 (Backwards Simulation) *A backwards simulation between actions* A_1 *and* A_2 *of processes* P_1 *and* P_2, *with local state* L, *is a relation* R *between* $P_1.State$, $P_2.State$, *and* L *satisfying*

- *(Feasibility)* $\forall P_2.State;\ L \bullet (\exists P_1.State \bullet R)$
- *(Correctness)* $\forall P_1.State';\ P_2.State;\ P_2.State';\ L' \bullet R' \land A_2 \Rightarrow (\exists P_1.State;\ L \bullet R \land A_1)$

A backwards simulation between P_1 *and* P_2 *is a backwards simulation between their main actions.* □

A refinement strategy for *Circus* has already been presented [1]. This strategy, although simple, can effectively serve as a tool to guide and transform an abstract (usually centralised) specification into a concrete (usually distributed) solution of the system. This strategy is based on laws of simulation, and action and process refinements, which are present in Appendix C. We, however, present further simulation and refinement laws in Appendix D.

Each iteration within the refinement strategy, which may include many iterations, includes three steps: simulation, action refinement, and process refinement. Figure 3.1, taken from [1], summarises one of these iterations. The first two steps are used to reorganise the internal structure of the process: we use simulation to

introduce the elements of the concrete system state, and then, the actions are refined in order to be partitioned in a way that each partition operates on different components of the modified state. This changes results in the splitting of the state space and the accompanying actions into two different partitions, in such a way that each partition groups some state components and the actions which access these components. After the second step, we have a structure in which each partition clearly has a independent state and behaviour. The third step of the strategy upgrades each of these partitions to individual processes: the resulting processes are combined in the same way as their main actions were in the previous process.

As discussed before, it may be the case that several refinement iterations are needed: processes resulting from one decomposition may be further decomposed. Basically, one iteration of the strategy is necessary for every process that needs decomposing, just like it is needed when the starting point is a centralised process specification.

The sequence of steps do not need to be strictly followed an iteration of the refinement strategy. For instance, many applications of a given step (i.e. data or action refinement) may be applied, if it is convenient for modularising the development; the starting process itself may already be ready for decomposition, in which case, only a process refinement is needed. Basically, an iteration of the strategy is characterised by one application of a process decomposition. The number of applications of simulation and action refinement depends on each particular development.

3.2 Laws of Simulation

In order to carry the data refinement in a stepwise way, some laws of simulation are provided. These laws provide support to prove that a relation R is indeed a forwards simulation. Besides, they can be used to justify proving simulations for schema actions, in much the same way as we do in Z. All the simulation laws can be found in Appendices C and D.

The primitive actions *Skip*, *Stop*, and *Chaos* are not affected by forwards simulations. Schema action simulations raise the same provisos as in the standard Z rule. The law C.1 presented below includes an applicability condition, which does not appear in the definition of forwards simulation, since it is concerned with the semantics of actions, which are total operations the state that include the UTP variables. A schema expression, on the other hand, is an operation over the process state, and it is not total.

Law C.1 (Schema Expressions).

$$ASExp \preceq CSExp$$

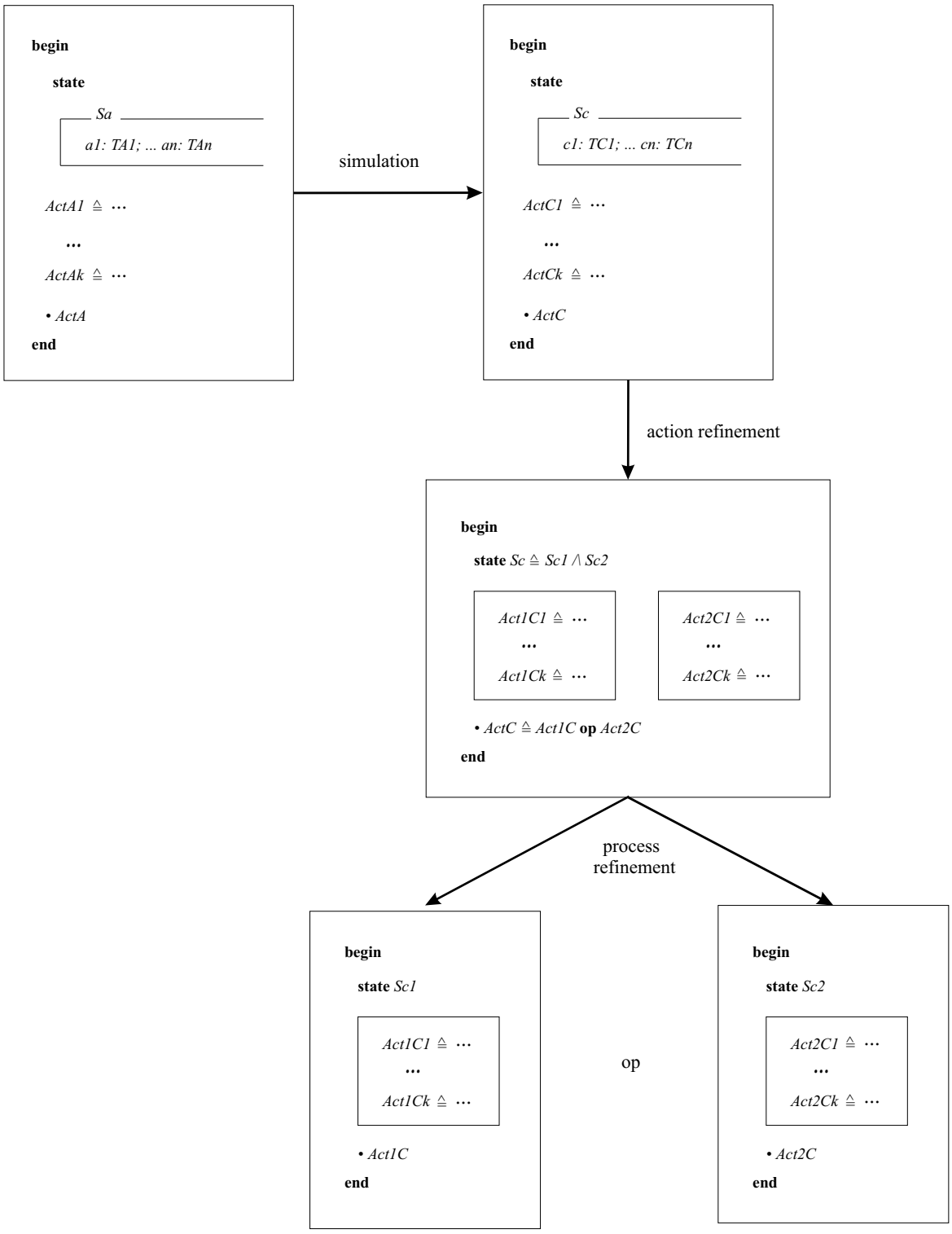

Figure 3.1: An iteration of the refinement strategy

provided

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

Forwards simulation distributes through the other constructs. In the following, we present some of the distributions laws. The first one is the rule for input prefix.

Law C.2 (Input prefix distribution).

$$c?x \rightarrow A_1 \preceq c?x \rightarrow A_2$$

provided $A_1 \preceq A_2$.
In the output prefixing, the abstract and the concrete expressions must be equal, with respect to the retrieve relation.

Law C.3 (Output prefix distribution).

$$c!ae \rightarrow A_1 \preceq c!ce \rightarrow A_2$$

provided

- $\forall\, P_1.st;\ P_2.st;\ L \bullet R \Rightarrow ae = ce$
- $A_1 \preceq A_2$ □

Parallel actions work on disjoint parts of the state: no interference occurs. This fact is used in the simulation law for parallelism.

Law C.7 (Parallelism distribution).

$$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 \preceq B_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

Further laws on simulation can be found in Appendices C and D.

3.3 Process Refinement

The laws for process refinement deal simultaneously with the state and control behaviour. The first law states that we may introduce a new process, assuming it is not used. As in the introduction of fresh variables in an imperative language, the fact that the process is unused is sufficient to guarantee that its introduction has no effect whatsoever.

Law C.10 (Process declaration introduction).

$$cp = pd\ cp$$

provided the process declared in the process declaration pd is not referenced in the sequence of paragraphs of the *Circus* program cp.
This law is apparently innocuous. Its importance, however, becomes evident in the sequel.

In [1], two families of process partitioning laws are presented. The first family apply to processes whose state components are partitioned in such a way that each partition has its own set of paragraphs. By way of illustration, we present the process P below.

$$\begin{array}{l} P = \textbf{begin} \\ \quad \textbf{state}\ State \mathrel{\widehat=} Q.State \land R.State \\ \quad Q.PPar \upharpoonright R.State \\ \quad R.PPar \upharpoonright Q.State \\ \quad \bullet\ F(P.Act, Q.Act) \\ \textbf{end} \end{array}$$

As in Section 2.2.2, the state of the processes P is defined as a conjunction of two other schemas: $Q.State$ and $R.State$. Furthermore, the paragraphs in P are also partitioned in a way that the paragraphs in $Q.PPar$ do not change the components in $R.State$, since they are conjoined with $\Xi R.State$; in a similar way, the paragraphs in $R.PPar$ do not change the components in $Q.State$. Finally, the main action of P is defined as an action context F, which must also make sense as a function on processes, according to the *Circus* syntax (Appendix A).

The law C.11 presented below transforms such partitioned process into three processes: each of the first two includes a partition of the state and the corresponding paragraphs, and, the third process, defined in the terms of the first two, has the same behaviour as the original one.

Law C.11 (Process splitting). 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

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

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

The second family of laws applies to process defined using the well-known Z promotion technique. Using this family of laws, we may refine a specification using a free promotion to an indexed family of processes, each one representing an element of the local type. In [1], the Z promotion technique is extended to *Circus* actions. Firstly, as expected, we have that the promotion of schema expressions is as in Z.

$$\mathbf{promote}(SExp) \mathrel{\widehat=} \exists\, \Delta L.State \bullet SExp \land Promotion$$

$L.State$ stands for the local state, and $Promotion$ for the promotion schema.

The promotion of $Skip$, $Stop$, and $Chaos$ does not change them.

$$\mathbf{promote}(A) \mathrel{\widehat=} A,\ for\ A \in \{Skip, Stop, Chaos\}$$

The promotion of a communication $c.e$, where e stands for a reference to an element of the local state, needs to receive an extra value that indicates the position i of e in the collection. For this reason, a corresponding promoted channel pc, which communicates a pair formed by the identifier and the value, exists for each channel c.

$$\mathbf{promote}(c.e \rightarrow A) \mathrel{\widehat=} pc?i.\mathbf{promote}(e) \rightarrow \mathbf{promote}(A)$$

The definitions of promotion for the other forms of prefixing are very similar. The guards of guarded commands need to be promoted. In the promotion of parallelism and hiding, the channels are replaced with the corresponding promoted channels.

Further laws on promotion of actions and processes can be found in [1].

3.4 Action Refinement

In the second step of the refinement strategy, an algorithmic refinement on actions is proposed. This action refinement is justified by the following theorem, which is proved in [28].

Theorem 3.2 (Soundness of action refinement) *Suppose we have a process P with actions A_1 and A_2. If $A_1 \sqsubseteq_{\mathcal{A}} A_2$, then the identity is a forwards simulation between A_1 and A_2.* □

Using this theorem, we can refine a process by refining its actions.

All the refinement laws on actions can be found in Appendices C and D. In the following, we present some of the new laws required by our case study. They are samples of some groups of laws: laws on prefixing, laws on schemas, laws on variable blocks, laws on guards and assumptions, and laws on parallelism.

3.4.1 Laws on Prefixing

The following law states that a prefixing may be introduced to an action, if it is hidden from the environment.

Law D.5 (Prefixing Introduction).

$$A = (c \rightarrow A) \setminus \{\!|\ c\ |\!\}$$

provided $c \notin usedC(A)$.

3.4.2 Laws on Schemas

The following law applies to an initialisation schema, which operates over a state composed by two disjoint sets of components specified in the schemas S_1 and S_2. Furthermore, its precondition can be expressed as the conjunction of conditions $preS_1$ and $preS_2$ over the different parts of the state. The initialisation of the state is an expression that is a conjunction of conditions CS_1 and CS_2 over the final values of the disjoint parts of the state.

Law D.7 (Schemas/Sequence Introduction 2).

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

provided

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

The application of this law introduces a sequence of two initialisation schemas that initialise the disjoint parts of the state separately. As the initial values of S_1 are potentially changed by the first action, the final values of the state components of S_2 cannot depend on those.

For a given schema *SExp*, $\alpha(SExp)$ gives the set of components of *SExp*; the function *alpha* can also be applied to a declaration. The function FV gives the set of free-variables of a predicate or expression; in a similar way, DFV determines the set of dashed free variables of a given predicate, and $UDFV$ gives the set of undashed free variables of a predicate.

3.4.3 Laws on Variable Blocks

A standard law on variable blocks extension of imperative programming languages is also valid for *Circus*.

Law D.8 (Variable Block Extension).

$$A_1;(\mathbf{var}\ x : T \bullet A_2);A_3 = (\mathbf{var}\ x : T \bullet A_1;A_2;A_3)$$

provided $x \notin FV(A_1) \cup FV(A_3)$.
This law only requires the variable x not to be a free variable in A_1 and A_3.

3.4.4 Laws on Guards and Assumptions

Guards may be distributed to a particular side of the parallelism. The only condition required to make this law sound is that the initial events of the other side of the parallelism are in the synchronisation set of channels.

Law D.10 (Guard/Parallelism Distribution 1).

$$\Box_i\, g_i \ \&\ (A_i \ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ A) = (\Box_i\, g_i \ \&\ A_i) \ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ A$$

provided $initials(A) \subseteq cs$.
The proviso guarantees that the action A can start its behaviour only when one of the guards is *true*.

3.4.5 Laws on Parallelism

The *Skip* is the parallelism unit.

Law D.25 (Parallelism Unit).

$$Skip \ [\![\, cs \,]\!]\ A = A \ [\![\, cs \,]\!]\ Skip = A$$

provided $usedC(A) \cap cs = \emptyset$.

As expected, the other action involved in the parallelism is not expected to synchronise in any of its events. Otherwise, it could deadlock at some point of its behaviour.

This gives us the intuition for the next law presented below. The parallelism of two actions, whose initial events are in the synchronisation channel set and have no initial event in common, is deadlocked.

Law D.27 (Parallelism Zero 2).

$$A_1 \llbracket\, cs \,\rrbracket\, A_2 = Stop$$

provided

- $initials(A_1) \cup initials(A_2) \subseteq cs$
- $initials(A_1) \cap initials(A_2) = \emptyset$ □

In our case study (Chapter 5), we use these laws and those presented in Appendices C and D in order to refine an abstract specification.

Most of the refinement laws of *Circus* are still to be proved. This proof of these laws are in the context of our thesis.

3.5 Conclusions

This section will conclude this chapter. It will be written after all refinement laws have been proved.

Chapter 4

Mechanisation

Chapter 5

Case Study

In this chapter we present a case study on the *Circus* refinement calculus. The case study is a safety-critical fire protection system, that is described in Section 5.1. In Sections 5.2 and 5.3 we describe the types and channels used within the system, respectively. Section 5.4 presents and describes some axiomatic definitions that are used throughout the system definition. In Section 5.5 present an abstract specification for the fire control system. This specification is refined to a concrete one using a refinement strategy presented in Section 5.6. Finally, in Section 5.7 we present some conclusions on the case study.

5.1 Description of the System

The fire control system covers two separate areas. Each area is divided in two zones, and a fire detection can happen in each different zone. If a detection happens, a gas discharge may occur in the area in which the zone where the detection occurred belongs. Besides the four existing zones, two other zones are used for detection only. The system contains a display panel composed by the following lamps:

- System on lamp: indicates that the system is working;
- Fault lamps: indicate that the corresponding fault has been detected by the system;
- Detection lamps: indicate a fire detection in the corresponding zone;
- Silence alarm lamp: indicates that the alarm has been silenced;
- Circuit fault lamp: indicates the need to replace the actuators of the system;
- Discharge lamps: indicate a gas discharge in the corresponding area.

The system can be in one of three modes at any moment: manual, automatic, or disabled. In manual mode, an alarm is sounded if a detection happens; besides, the corresponding detection lamp is lit on the system's display panel. The alarm can be silenced, and, when the reset button is pressed, the system returns to normal. In manual mode, gas discharge needs to be initiated manually.

When the system is running in automatic mode, the detection of fire 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 to automatic gas discharge is started. When the countdown finishes, the gas discharge happens after an exit event. This event represents the latest time people are expected to leave the building before a gas discharge happens. Following a gas discharge, the circuit fault lamp is illuminated in the system's display panel. Besides, if a gas discharge occurs, 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.

Some further requirements should also be satisfied by the system's specification.

- Start up: the system must be started with a *switchOn* event, and, following this event, the system on lamp should be illuminated;
- Mode switching: the system mode can be switched between manual and automatic mode provided no detection happens. Also, when the system is reset, and, in case of a gas discharge has occurred, the actuators are replaced, the system mode is switched to automatic;

- Detection lamps: following a fire detection, the corresponding lamp must be lit; when the system is reset, all the fire detection lamps must be switched off;
- Gas discharge: following a gas discharge, no following discharge may happen before the actuators are replaced.

Of course, the system may not enter a state in which it refuses to participate in any further events (deadlock-free).

5.2 Basic Types

We start by defining the boolean type *Bool*, which can assume values *true* and *false*.

$$Bool ::= true \mid false$$

The two areas and the six zones are identified by the types *AreaId* and *ZoneId* respectively.

$$AreaId ::= 0 \mid 1$$
$$ZoneId ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5$$

As described in Section 5.1, the system can be in three modes at any time. These modes are represented by the type *Mode*. However, the system cannot be manually switched to a disabled mode. For this reason, we define the type *SwitchMode*, which is actually a subset of type *Mode*.

$$Mode ::= automatic \mid manual \mid disabled$$
$$SwitchMode == Mode \setminus \{disabled\}$$

All the lamps and the buzzer of the system's display panel can be either on or off. These two states are represented by the type *OnOff*. Furthermore, the system alarm can be in one of the three states in type *AlarmStage*.

$$OnOff ::= on \mid off$$
$$AlarmStage ::= alarmOff \mid firstStage \mid secondStage$$

The type *LampId* contains individual identifiers for all the existing lamps in the system's display panel.

$$\begin{aligned} LampId ::= {} & zoneFaultLamp \mid earthFaultLamp \mid sounderLineFaultLamp \\ & \mid powerFaultLamp \mid isolateRemoteSignalLamp \mid actuatorLineFaultLamp \\ & \mid circuitFaultLamp \mid alarmSilencedLamp \mid systemOnLamp \end{aligned}$$

The faults that can be detected by the system are as follows.

$$FaultId ::= zoneFault \mid earthFault \mid sounderLineFault \mid powerFault \mid isolateRemoteSignal \mid actuatorLineFault$$

Finally, the system can be in one of the state of type *SystemState* described below.

$$SystemState ::= fireSysStart_s \mid fireSys_s \mid fireSysD_s \mid auto_s \mid countdown_s \mid discharge_s \mid reset_s \mid manual_s$$

This type is used by the system to indicate its current state.

5.3 Used Channels

In order to start working, the system must be started. This is modelled as the event *switchOn*.

channel *switchOn*

The existence of fire is indicated to the system by a smoke detection, which is identified by the zone where the detection happens.

channel *detection* : *ZoneId*

The system can be manually switched between modes *manual* and *automatic*, but not *disabled*.

channel *modeSwitch* : *SwitchMode*

When the system is in *manual* mode, and the conditions that lead to a gas discharge are met, the manual gas discharge can be started in one area only.

channel *externalManualDischarge* : *AreaId*

A system fault can be reported to the system.

channel *fault* : *FaultId*

In manual mode, an alarm is sounded if a detection happens. This alarm, however, can be silenced.

channel *silenceAlarm*

In order to have the system back to its normal, the system must be reset. Furthermore, if a gas discharged occurred, the actuators must be replaced.

channel *reset*, *actuatorsReplaced*

The exit event represents the latest time people are expected to leave the building before a gas discharge happens.

channel *exit*

The system determines the alarm stage to which the alarm must be switched through channel *alarm*.

channel *alarm* : *AlarmStage*

Besides, the system indicates that a lamp must be switched (on or off) using the generic channel *switchLamp*. It provides the type of lamp (*AreaId*,*ZoneId*, or *FaultId*) and the new lamp mode (*OnOff*). The buzzer is also controlled using a specific channel.

channel $[T]$*switchLamp* : $T \times$ *OnOff*
channel *switchBuzzer* : *OnOff*

Finally, in each state change, the system reports its current state.

channel *systemState* : *SystemState*

The fire control system may request a clock to execute the countdown using channel *startClock*. The clock indicates that the countdown is finished using channel *clockFinished*.

channel *startClock*, *clockFinished*

In Figure 5.1 we summarise the abstract fire control system interface.

5.4 Axiomatic Definitions

An area is responsible for 2 zones. The function *getZones* return the zones which are covered by an area. Two zones cannot be covered by two different areas.

$$
\begin{array}{|l}
getZones : AreaId \rightarrowtail \mathbb{P}\, ZoneId \\
\hline
getZones(0) = \{0, 1\} \\
getZones(1) = \{2, 3\}
\end{array}
$$

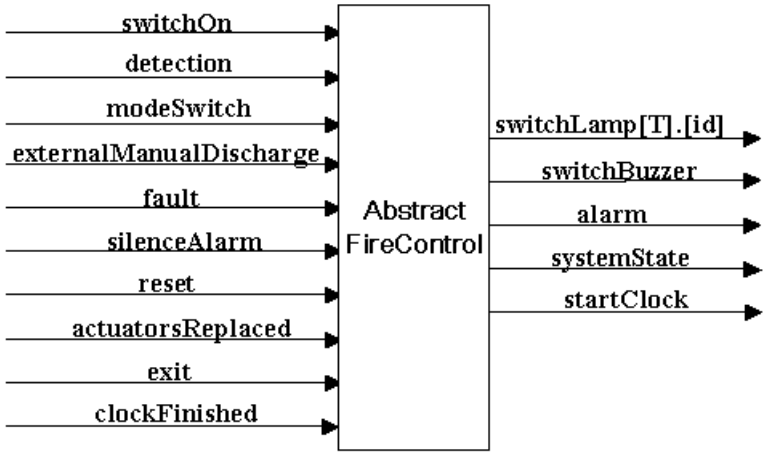

Figure 5.1: Abstract Fire Control

The following axiomatic definition is used to map faults to their respective lamp in the display.

$$getLampId : FaultId \rightarrowtail LampId$$

$$getLampId(zoneFault) = zoneFaultLamp$$
$$getLampId(earthFault) = earthFaultLamp$$
$$getLampId(sounderLineFault) = sounderLineFaultLamp$$
$$getLampId(powerFault) = powerFaultLamp$$
$$getLampId(isolateRemoteSignal) = isolateRemoteSignalLamp$$
$$getLampId(actuatorLineFault) = actuatorLineFaultLamp$$

In the case the system is running in automatic mode, a delay *gasDelay* is given before the gas is released.

$$gasDelay : \mathbb{N}$$

$$gasDelay = 30$$

5.5 Abstract Fire Control System

In this section, we specify process *AbstractFireControl* to formalise the requirements previous described. Its state is composed of five components: the *mode* component indicates the mode in which the fire control is running (*automatic*, *manual*, or *disabled*); the *controlledZones* component maps the areas to their controlled zones; the *activeZones* component maps the areas to the zones in which a

fire detection has occurred; the *discharge* component indicates if a gas discharged has happened (*true*), or not (*false*) within each area; finally, the *active* component indicates if the each area is active (*true*), or not (*false*).

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

$$
\begin{array}{|l}
\hline
AbstractFireControlState \\
\hline
mode : Mode \\
controlledZones : AreaId \pfun \mathbb{P}\, ZoneId \\
activeZones : AreaId \pfun \mathbb{P}\, ZoneId \\
discharge : AreaId \pfun Bool \\
active : AreaId \pfun Bool \\
\hline
controlledZones = \{area : AreaId \bullet area \mapsto getZones(area)\} \\
\forall\, area : AreaId \bullet \\
\quad (mode = automatic) \Rightarrow \\
\quad\quad active(area) = true \Leftrightarrow \\
\quad\quad\quad \exists\, z_1, z_2 : controlledZones(area) \bullet \\
\quad\quad\quad\quad z_1 \neq z_2 \land \{z_1, z_2\} \subseteq activeZones(area) \\
\quad \land\, (mode = manual) \Rightarrow \\
\quad\quad active(area) = true \Leftrightarrow \\
\quad\quad\quad \{area : AreaId \mid \exists\, z : controlledZones(area) \bullet \\
\quad\quad\quad\quad z \in activeZones(area)\} \\
\quad \land\, activeZones(area) \subseteq controlledZones(area) \\
\hline
\end{array}
$$

The invariant determines that, for each area within the system, its controlled zones is defined by the axiomatic definition *getZones*. If running in *manual* mode, an area is *active* if, and only if, any zone controlled by it is active. However, if running in *automatic* mode, an area is active if, and only if, there is more than one active zone controlled by it.

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

$$
\begin{array}{|l}
\hline
InitAbstractFireControl \\
\hline
AbstractFireControlState' \\
\hline
mode' = automatic \\
activeZones' = \{area : AreaId \bullet area \mapsto \emptyset\} \\
discharge' = \{area : AreaId \bullet area \mapsto false\} \\
\hline
\end{array}
$$

We present three operations that are used to switch the system mode. The first one receives the new mode as argument.

$$
\begin{array}{|l}
\hline
\textit{SwitchAbstractFireControlMode} \\
\Delta \textit{AbstractFireControlState} \\
\textit{newMode?} : \textit{Mode} \\
\hline
\textit{mode}' = \textit{newMode?} \\
\textit{activeZones}' = \textit{activeZones} \\
\textit{discharge}' = \textit{discharge} \\
\hline
\end{array}
$$

It changes only the system mode, and leaves the other components unchanged. Two other schemas are used with the same purpose. They, however, do not receive any argument. The first one switches the system to automatic mode.

$$
\begin{array}{|l}
\hline
\textit{SwitchAbstractFireControl2AutomaticMode} \\
\Delta \textit{AbstractFireControlState} \\
\hline
\textit{mode}' = \textit{automatic} \\
\textit{activeZones}' = \textit{activeZones} \\
\textit{discharge}' = \textit{discharge} \\
\hline
\end{array}
$$

The second one switches the system to disabled mode.

$$
\begin{array}{|l}
\hline
\textit{SwitchAbstractFireControl2DisabledMode} \\
\Delta \textit{AbstractFireControlState} \\
\hline
\textit{mode}' = \textit{disabled} \\
\textit{activeZones}' = \textit{activeZones} \\
\textit{discharge}' = \textit{discharge} \\
\hline
\end{array}
$$

Both, as the operation *SwitchAbstractFireControlMode* leave the other state components unchanged.

Next, we describe the schema *AbstractActivateZone*. It receives a zone as input and changes the *activeZones* state component by including the received zone in the set of active zones mapped by the area that controls the given zone. The *active* component is also changed as specified by the state invariant. All other state components are left unchanged.

$$
\begin{array}{|l}
\hline
\textit{AbstractActivateZone} \\
\Delta \textit{AbstractFireControlState} \\
\textit{newZone}? : \textit{ZoneId} \\
\hline
\textit{mode}' = \textit{mode} \\
\textit{discharge}' = \textit{discharge} \\
\textit{activeZones}' = \textit{activeZones} \oplus \\
\quad \{\textit{area} : \textit{AreaId} \mid \textit{newZone}? \in \textit{controlledZones}(\textit{area}) \bullet \\
\qquad \textit{area} \mapsto \textit{activeZones}(\textit{area}) \cup \{\textit{newZone}?\}\} \\
\hline
\end{array}
$$

The last schema operation activates the discharge in the active areas. Only the state component *discharge* is changed.

$$
\begin{array}{|l}
\hline
\textit{AbstractActivateDischarge} \\
\Delta \textit{AbstractFireControlState} \\
\hline
\textit{mode}' = \textit{mode} \\
\textit{activeZones}' = \textit{activeZones} \\
\textit{discharge}' = \textit{discharge} \oplus \\
\quad \{\textit{area} : \textit{AreaId} \mid \textit{area} \in \mathrm{dom}\, \textit{active} \rhd \{\textit{true}\} \bullet \textit{area} \mapsto \textit{true}\} \\
\hline
\end{array}
$$

As all other actions of the fire control system presented below, the action *AbstractFireSysStart* starts by communicating the current system state. Then, it waits for the system to be switched on, switches the corresponding lamp on, initialises the system state and, finally, behaves like action *AbstractFireSys*.

$$
\begin{array}{l}
\textit{AbstractFireSysStart} \mathrel{\widehat=} \\
\quad \textit{systemState}!\textit{fireSysStart}_s \rightarrow \textit{switchOn} \rightarrow \\
\qquad \textit{switchLamp}[\textit{LampId}].\textit{systemOnLamp}!\textit{on} \rightarrow \\
\qquad\quad \textit{InitAbstractFireControl}; \textit{AbstractFireSys}
\end{array}
$$

The action *AbstractFireSys* represents the system running before any detection. After communicating the system state, the mode can be switched between *automatic* and *manual*. Furthermore, if any detection occurs, the zone in which the detection occurred is active, and then, the system behaves like *Auto* or *Manual*, depending on the current system mode. If any fault is identified, the corresponding lamp is lit. Finally, if the system is requested to reset, its state is initialised, and all the lamps (except the circuit fault and the system on lamps) and the buzzer are

switched off (action *AbstractSwitchLampsOff*).

$$
\begin{array}{l}
AbstractFireSys \mathrel{\widehat=} \\
\quad systemState!fireSys_s \rightarrow \\
\qquad modeSwitch?newMode : SwitchMode \rightarrow \\
\qquad\quad SwitchAbstractFireControlMode;\ AbstractFireSys \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow AbstractActivateZone; \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\qquad\qquad (mode = manual)\ \&\ AbstractManual \\
\qquad\qquad \Box\ (mode = automatic)\ \&\ AbstractAuto \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow AbstractFireSys \\
\qquad \Box\ reset \rightarrow alarm!alarmOff \rightarrow InitAbstractFireControl; \\
\qquad\quad AbstractSwitchLampsOff; AbstractFireSys
\end{array}
$$

$$
\begin{array}{l}
AbstractSwitchLampsOff \mathrel{\widehat=} \\
\quad (switchBuzzer!off \rightarrow Skip \\
\quad \|\!\| (|\!|\!| id : (LampId \setminus \{circuitFaultLamp, systemOnLamp\}) \bullet \\
\qquad switchLamp[LampId].id!off \rightarrow Skip) \\
\quad \|\!\| (|\!|\!| zone : ZoneId \bullet switchLamp[ZoneId].zone!off \rightarrow Skip) \\
\quad \|\!\| (|\!|\!| area : AreaId \bullet switchLamp[AreaId].area!off \rightarrow Skip)
\end{array}
$$

After a detection, and if running in *manual* mode, the system behaves like action *AbstractManual*. Any detection leads to the activation of the zone in which the detection happened. If it is requested to silence the alarm, it does so, and behaves like the *AbstractReset* action. If any manual discharge is requested, the action verifies whether the given area is active or not. If it is active, it switches the corresponding gas discharged lamp, discharges the gas in the given area, switches the system mode to disabled and waits for the system to be reset (action *AbstractReset*). However, if the given area is not active, the discharge request is ignored. Finally,

any fault identification leads to the corresponding lamp to be lit.

$$
\begin{array}{l}
AbstractManual \mathrel{\widehat=} \\
\quad systemState!manual_s \rightarrow \\
\qquad detection?newZone : ZoneId \rightarrow AbstractActivateZone; \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow AbstractManual \\
\qquad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow AbstractReset \\
\qquad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad\quad (area \in \mathrm{dom}\ active \rhd \{true\})\ \& \\
\qquad\qquad switchLamp[AreaId].area!on \rightarrow AbstractActivateDischarge; \\
\qquad\qquad\quad SwitchAbstractFireControl2DisabledMode; \\
\qquad\qquad\qquad AbstractReset \\
\qquad\quad \Box\ (area \notin \mathrm{dom}\ active \rhd \{true\})\ \&\ AbstractManual \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow AbstractManual
\end{array}
$$

After a detection, and if running in *automatic* mode, the system behaves like *AbstractAuto*. After reporting the system state, it verifies whether there is any active area or not. If there is any active area, the second stage alarm is sounded and the countdown starts (action *AbstractCountdown*). However, if no area is active, the it is possible to reset the system, by switching the alarm, all the lamps (except the circuit fault and the system on lamps), and the buzzer off, initialising the system state, and starting to behave as *AbstractFireSys*. Detections are also possible but they do not change the behaviour of this action. They only lead to an activation of the zone in which the detection occurred. Finally, as the *AbstractManual* action, any fault identification leads to the corresponding lamp to be lit.

$$
\begin{array}{l}
AbstractAuto \mathrel{\widehat=} \\
\quad systemState!auto_s \rightarrow \\
\qquad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad\quad alarm!secondStage \rightarrow AbstractCountdown \\
\qquad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad\quad reset \rightarrow alarm!alarmOff \rightarrow AbstractSwitchLampsOff; \\
\qquad\qquad InitAbstractFireControl;\ AbstractFireSys \\
\qquad\quad \Box\ detection?newZone : ZoneId \rightarrow AbstractActivateZone; \\
\qquad\qquad switchLamp[ZoneId].newZone!on \rightarrow AbstractAuto \\
\qquad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad\quad switchBuzzer!on \rightarrow AbstractAuto
\end{array}
$$

In action *Reset*, the actuators may be replaced. In this case, the circuit fault lamp is switched off, and the system mode is switched to *automatic*. The behaviour

in the case of any detection or fault is the same as for action *AbstractManual*. Finally, if it is requested to reset, it behaves like *FireSys* or *FireSysD*, depending on the current system mode.

$$
\begin{array}{l}
AbstractReset \mathrel{\widehat=} \\
\quad systemState!reset_s \rightarrow \\
\quad\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad\quad SwitchAbstractFireControl2AutomaticMode;\ AbstractReset \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow AbstractActivateZone; \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow AbstractReset \\
\quad\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow AbstractSwitchLampsOff; \\
\quad\quad\quad (mode = disabled)\ \&\ AbstractFireSysD \\
\quad\quad\quad \Box\ (mode \neq disabled)\ \&\ InitAbstractFireControl;AbstractFireSys \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow AbstractReset
\end{array}
$$

The action *AbstractCountdown* requests the start of a countdown and waits for the clock to finish the countdown (action *AbstractWaitingClock*). While waiting for the clock to be finish (event *clockFinished*), detections and faults lead to the same behaviour as for action *AbstractReset*. When the clock is finished, it discharges the gas (action *AbstractDischarge*).

$$
\begin{array}{l}
AbstractCountdown \mathrel{\widehat=} \\
\quad systemState!countdown_s \rightarrow startClock \rightarrow AbstractWaitingClock \\
\\
AbstractWaitingClock \mathrel{\widehat=} \\
\quad clockFinished \rightarrow AbstractDischarge \\
\quad \Box\ detection?newZone : ZoneId \rightarrow AbstractActivateZone; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow AbstractWaitingClock \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow AbstractWaitingClock
\end{array}
$$

The action *AbstractFireSysD* indicates that the system was reset without the actuators being replaced, and, for this reason, it is disabled. In this action, only two events are accepted: first, the actuators can be replaced, in which case, the alarm is switched off, as are all the lamps (except the circuit fault and the system on lamps) and the buzzer, and the system returns to the action *AbstractFireSys*; second, if

any fault is identified, the corresponding lamp is lit.

$$
\begin{array}{l}
AbstractFireSysD \mathrel{\widehat=} systemState!fireSysD_s \rightarrow \\
\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad AbstractSwitchLampsOff ; InitAbstractFireControl ; AbstractFireSys \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow AbstractFireSysD
\end{array}
$$

Finally, the action *AbstractDischarge* represents an automatic discharge. The action waits for an indication that there is no one in the area (event *exit*). When this indication is received, it switches the gas discharge lamps of all areas in which a gas discharge should occur (action *AbstractSwitchLampsDischarge*). If any area is actually active, the system mode is switched to *disabled*; otherwise, it is switched to *automatic*. Finally, it activates any discharge that should occur, and then it waits for the system to be reset (action *AbstractReset*).

$$
\begin{array}{l}
AbstractDischarge \mathrel{\widehat=} systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\quad\quad AbstractSwitchLampsDischarge ; \\
\quad\quad ((\mathrm{dom}\ active \rhd \{true\} \neq \emptyset)\ \& \\
\quad\quad\quad SwitchAbstractFireControlSystem2DisabledMode \\
\quad\quad \Box\ (\mathrm{dom}\ active \rhd \{true\} = \emptyset)\ \& \\
\quad\quad\quad SwitchAbstractFireControlSystem2AutomaticMode) ; \\
\quad\quad AbstractActivateDischarge ; AbstractReset
\end{array}
$$

$$
\begin{array}{l}
AbstractSwitchLampsDischarge \mathrel{\widehat=} \\
\quad (\fatsemi\ area : \mathrm{dom}\ active \rhd \{true\} \bullet switchLamp[AreaId].area!on \rightarrow Skip)
\end{array}
$$

Finally, the main action of process *AbstractFireSys* is defined below.

- *AbstractFireSysStart*

end

5.5.1 External Devices

Clock

Before discharging the gas, the fire control system requires a clock to countdown. We present below the specification of the process *Clock*. Basically, when requested to start counting (event *startClock*), it counts as many *tock*s as specified by *gasDelay*.

Then, it indicates that it is finished (event *clockFinished*), and waits for the next request for counting.

channel *tock*

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

$ClockCycle \mathrel{\widehat=} startClock \rightarrow ClockCycle(gasDelay); ClockCycle$

$ClockInstance \mathrel{\widehat=} (tocks : \mathbb{N} \bullet$
$(tocks = 0)\ \&\ clockFinished \rightarrow Skip$
$\Box\ (tocks \neq 0)\ \&\ tock \rightarrow ClockInstance(tocks - 1))$

$\bullet\ ClockCycle \setminus \{\!|\ tock\ |\!\}$

end

Output

Display The display is composed by 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*. This generic process has just one component (*status*) that indicates if the lamp is switched on or switched off.

process $[T]GenericLamp \mathrel{\widehat=} (id : T \bullet$ **begin**
state $GenericLampState \mathrel{\widehat=} [\, status : OnOff \,]$

The schema *InitGenericLamp* initialises the lamp with an *off* *status*.

$$InitGenericLamp \mathrel{\widehat=} [\, GenericLampState' \mid status' = off \,]$$

The schema *SwitchLampStatus* switches the *status* to the given new status.

$SwitchLampStatus \mathrel{\widehat=}$
$[\, \Delta GenericLampState;\ status? : OnOff \mid status' = status? \,]$

The generic lamp is initialised, and then it can continuously be switched *on* or *off*.

$\bullet\ InitGenericLamp;$
$(\mu\, X \bullet switchLamp[T].id?status \rightarrow SwitchLampStatus;\ X)$

end)

A generic lamp can be instantiated with *LampId* for the simple lamps, with *ZoneId* for the fire lamps, or with *AreaId* for the lamps that indicate that gas was released in the respective area. The process *SimpleLamps* represents all the simple lamps (fault lamps, alarm silenced lamp, and system on lamp); the process *FireLamps* represents

all the fire detection lamps; and the process *GasReleasedLamps* represents all the gas released lamps. Finally, process *DisplayLamps* represents all the display's lamps.

$$\textbf{process}\ SimpleLamps \mathrel{\widehat=} \Vert\!\Vert\ id : LampId \bullet GenericLamp[LampId](id)$$

$$\textbf{process}\ FireLamps \mathrel{\widehat=} \Vert\!\Vert\ zone : ZoneId \bullet GenericLamp[ZoneId](zone)$$

$$\textbf{process}\ GasReleasedLamps \mathrel{\widehat=} \Vert\!\Vert\ area : AreaId \bullet GasReleasedLamp[AreaId](area)$$

$$\textbf{process}\ DisplayLamps \mathrel{\widehat=} SimpleLamps \Vert\!\Vert FireLamps \Vert\!\Vert GasReleasedLamps$$

The display buzzer has a very simple specification. As the generic lamps, it has a state component *status* that indicates if the buzzer is switched on or switched off. This component is also initialised with the value *off*, and can also be switched between *on* and *off*. All the process offers is the *switchBuzzer* event that can be used to switch the buzzer status.

$$\textbf{process}\ DisplayBuzzer \mathrel{\widehat=} \textbf{begin}$$
$$\textbf{state}\ DisplayBuzzerState \mathrel{\widehat=} [\, status : OnOff \,]$$
$$InitDisplayBuzzer \mathrel{\widehat=} [\, DisplayBuzzerState' \mid status' = off \,]$$
$$SwitchDisplayBuzzerStatus \mathrel{\widehat=} [\, \Delta DisplayBuzzerState;\ status? : OnOff \mid status' = status? \,]$$
$$\bullet\ InitDisplayBuzzer;\ \mu X \bullet switchBuzzer?status \rightarrow SwitchDisplayBuzzerStatus;\ X$$
$$\textbf{end}$$

Finally, the display can be specified as the interleave of all display lamps and the display buzzer.

$$\textbf{process}\ Display \mathrel{\widehat=} DisplayLamps \Vert\!\Vert DisplayBuzzer$$

Alarm The alarm has also a very simple specification. It has a state component *stage*, which records the current state of the alarm (*firstStage*, *secondStage*, or *alarmOff*). This component is initialised with the value *alarmOff*, and can also be switched between the different alarm modes. All this process offers is the *alarm* event, which can be used to trigger a change to the stage of the alarm.

$$\textbf{process}\ Alarm \mathrel{\widehat=} \textbf{begin}$$
$$\textbf{state}\ AlarmState \mathrel{\widehat=} [\, stage : AlarmStage \,]$$
$$InitAlarm \mathrel{\widehat=} [\, AlarmState' \mid stage' = alarmOff \,]$$
$$SwitchStage \mathrel{\widehat=} [\, \Delta AlarmState;\ newStage? : AlarmStage \mid stage' = newStage? \,]$$
$$\bullet\ InitAlarm;\ \mu X \bullet alarm?newStage \rightarrow SwitchStage;\ X$$
$$\textbf{end}$$

The processes *Display* and *Alarm* can be interleaved to represent all the processes that receive any output from the abstract fire control system.

process $Output \mathrel{\widehat=} Display \mid\mid\mid Alarm$

Input Keyboard

In order to simplify the specification of the input to the abstract timed fire control system, we consider a single keyboard as the only means of interaction. This keyboard contains one button for each possible input event acceptable by the abstract fire control system.

$$\begin{aligned} Button == {} & DETECTION_0 \mid DETECTION_1 \\ & \mid DETECTION_2 \mid DETECTION_3 \\ & \mid DETECTION_4 \mid DETECTION_5 \\ & \mid ZONE\_FAULT \mid EARTH\_FAULT \\ & \mid SOUNDER\_LINE\_FAULT \mid POWER\_FAULT \\ & \mid ISOLATE\_REMOTE\_SIGNAL \mid ACTUATOR\_LINE\_FAULT \\ & \mid MANUAL\_DISCHARGE_0 \mid MANUAL\_DISCHARGE_1 \\ & \mid MODE\_SWITCH\_MANUAL \\ & \mid MODE\_SWITCH\_AUTOMATIC \\ & \mid RESET \mid SWITCH\_ON \mid SILENCE\_ALARM \\ & \mid ACTUATORS\_REPLACED \mid EXIT \end{aligned}$$

Some of these buttons can be grouped in four different categories: detection buttons, fault identification buttons, manual discharge buttons, and mode switch buttons.

$$\begin{aligned} DetectionBts == {} & \{DETECTION_0, DETECTION_1, \\ & \quad DETECTION_2, DETECTION_3, \\ & \quad DETECTION_4, DETECTION_5\} \\ FaultBts == {} & \{ZONE\_FAULT, EARTH\_FAULT, \\ & \quad SOUNDER\_LINE\_FAULT, POWER\_FAULT, \\ & \quad ISOLATE\_REMOTE\_SIGNAL, \\ & \quad ACTUATOR\_LINE\_FAULT\} \\ ManDischargeBts == {} & \{MANUAL\_DISCHARGE_0, \\ & \quad MANUAL\_DISCHARGE_1\} \\ ModeSwitchBts == {} & \{MODE\_SWITCH\_MANUAL, \\ & \quad MODE\_SWITCH\_AUTOMATIC\} \end{aligned}$$

The channel *actionPerformed* represents the action of pressing a button. Besides, the event *btnPressed* is used internally by the keyboard.

channel $actionPerfomed, btnPressed : Button$

The process keyboard has two state components: the enabled buttons and the disabled buttons. The set of disabled buttons contains any button that is not enabled.

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

$$
\begin{array}{|l}
\hline
KeyboardState \\
enabledButtons : \mathbb{P}\, Button \\
disabledButtons : \mathbb{P}\, Button \\
\hline
disabledButtons = \{b : Button \mid b \notin enabledButtons\} \\
\hline
\end{array}
$$

In the keyboard's initial state all buttons are disabled.

$$
\begin{array}{|l}
\hline
InitKeyboard \\
enabledButtons' \\
disabledButtons' \\
\hline
enabledButtons' = \emptyset \\
\hline
\end{array}
$$

The operation *AvailableButtons* receives a system state as argument, and changes the state component *enabledButtons* in order to enable the buttons that should be available, and, therefore, disable the buttons that should not be available for the given system state.

AvailableButtons

$\Delta KeyboardState$
$state? : SystemState$

$state? = fireSysStart'_s \Rightarrow$
$\quad enabledButtons' = \{SWITCH\_ON\}$
$state? = fireSys_s \Rightarrow$
$\quad enabledButtons' = DetectionBts \cup FaultBts \cup$
$\quad\quad ModeSwitchBts \cup \{RESET\}$
$state? = FireSysPrime'_s \Rightarrow$
$\quad enabledButtons' = FaultBts \cup$
$\quad\quad \{ACTUATORS\_REPLACED\}$
$state? = auto_s \Rightarrow$
$\quad enabledButtons' = DetectionBts \cup FaultBts \cup$
$\quad\quad \{RESET\}$
$state? = countdown_s \Rightarrow$
$\quad enabledButtons' = DetectionBts \cup FaultBts$
$state? = discharge_s \Rightarrow$
$\quad enabledButtons' = \{EXIT\}$
$state? = reset_s \Rightarrow$
$\quad enabledButtons' = DetectionBts \cup FaultBts \cup$
$\quad\quad \{ACTUATORS\_REPLACED,$
$\quad\quad\ RESET\}$
$state? = manual_s \Rightarrow$
$\quad enabledButtons' = DetectionBts \cup FaultBts \cup$
$\quad\quad ManDischargeBts \cup$
$\quad\quad \{SILENCE\_ALARM, RESET\}$

The action *EventHandler* receives the indication that a button was pressed and send this to the action *KeyboardCycle* (event *btnPressed*).

$$EventHandler \mathrel{\widehat=}$$
$$\quad actionPerfomed?button : enabledButtons \rightarrow$$
$$\quad\quad btnPressed!button \rightarrow EventHandler$$

For a given system state, the action *KeyboardCycle* uses the operation *AvailableButtons* to make available only those buttons that should be available to the user. Then, either a button can be pressed, or the system can change the state. If a button is pressed, the action synchronises in the event corresponding to the button that was pressed. This attempt can be interrupted by a change on the system state. At any time, if the system changes the state, the action *KeyboardCycle* starts again with

the new state.

$$
\begin{array}{l}
KeyboardCycle \mathrel{\widehat=} (state? : SystemState \bullet \\
\quad AvailableButtons; btnPressed?bt : Button \rightarrow \\
\quad \left(\begin{array}{l}
(bt = DETECTION_0)\ \&\ detection!0 \rightarrow Skip \\
\Box\ (bt = DETECTION_1)\ \&\ detection!1 \rightarrow Skip \\
\Box\ (bt = DETECTION_2)\ \&\ detection!2 \rightarrow Skip \\
\Box\ (bt = DETECTION_3)\ \&\ detection!3 \rightarrow Skip \\
\Box\ (bt = DETECTION_4)\ \&\ detection!4 \rightarrow Skip \\
\Box\ (bt = DETECTION_5)\ \&\ detection!5 \rightarrow Skip \\
\Box\ (bt = ZONE\_FAULT)\ \&\ fault!zoneFault \rightarrow Skip \\
\Box\ (bt = EARTH\_FAULT)\ \&\ fault!earthFault \rightarrow Skip \\
\Box\ (bt = SOUNDER\_LINE\_FAULT)\ \& \\
\qquad fault!sounderLineFault \rightarrow Skip \\
\Box\ (bt = POWER\_FAULT)\ \&\ fault!powerFault \rightarrow Skip \\
\Box\ (bt = ISOLATE\_REMOTE\_SIGNAL)\ \& \\
\qquad fault!isolateRemoteSignal \rightarrow Skip \\
\Box\ (bt = ACTUATOR\_LINE\_FAULT)\ \& \\
\qquad fault!actuatorLineFault \rightarrow Skip \\
\Box\ (bt = MANUAL\_DISCHARGE_0)\ \& \\
\qquad externalManualDischarge!0 \rightarrow Skip \\
\Box\ (bt = MANUAL\_DISCHARGE_1)\ \& \\
\qquad externalManualDischarge!1 \rightarrow Skip \\
\Box\ (bt = MODE\_SWITCH\_MANUAL)\ \& \\
\qquad modeSwitch!manual \rightarrow Skip \\
\Box\ (bt = MODE\_SWITCH\_AUTOMATIC)\ \& \\
\qquad modeSwitch!automatic \rightarrow Skip \\
\Box\ (bt = RESET)\ \&\ reset \rightarrow Skip \\
\Box\ (bt = SWITCH\_ON)\ \&\ switchOn \rightarrow Skip \\
\Box\ (bt = SILENCE\_ALARM)\ \&\ silenceAlarm \rightarrow Skip \\
\Box\ (bt = ACTUATORS\_REPLACED)\ \& \\
\qquad actuatorsReplaced \rightarrow Skip \\
\Box\ (bt = EXIT)\ \&\ exit \rightarrow Skip
\end{array} \right) ; \\
\qquad KeyboardCycle(state?) \\
\Box\ systemState?newState : SystemState \rightarrow KeyboardCycle(newState))
\end{array}
$$

The *RunKeyboard* action receives the current system state and starts the keyboard cycle.

$$RunKeyboard \mathrel{\widehat=} systemState?state : SystemState \rightarrow KeyboardCycle(state)$$

Finally, we have the keyboard's main action: after initialising the keyboard, it runs

in a parallel the *RunKeyboard* and the *EventHandler* actions.

- $InitKeyboard;$
  $$\left(\begin{array}{l} RunKeyboard \\ [\![\alpha(KeyboardState) \cup \alpha(KeyboardState')\{\!|\ btnPressed\ |\!\} \mid \emptyset]\!] \\ EventHandler \end{array} \right) \setminus \{\!|\ btnPressed\ |\!\}$$

end

The external devices are represented by the interleaving between the *Output* and the *Keyboard* processes.

process $ExternalDevices \mathrel{\widehat=} Output \ ||| \ Keyboard$

We may encapsulate all the input and output events in two different set of channels. First, we define the set of all switch events.

$$\begin{array}{l} \textbf{chanset}\ InputEvents == \{\!|\ switchLamp[LampId], switchLamp[ZoneId], \\ \qquad\qquad\qquad\qquad switchLamp[AreaId], switchBuzzer, \\ \qquad\qquad\qquad\qquad detection, switchOn, reset, \\ \qquad\qquad\qquad\qquad externalManualDischarge, modeSwitch, \\ \qquad\qquad\qquad\qquad silenceAlarm, actuatorsReplaced, exit, fault\ |\!\} \\ \textbf{chanset}\ OutputEvents == SwitchEvents\ \cup\ \{\!|\ alarm\ |\!\} \end{array}$$

Finally, we specify the set of all external events as follows.

$$\textbf{chanset}\ ExternalSignals == InputSignals \cup\ OutputEvents$$

The Abstract Main Process

First, we combine the *AbstractFireControl* and the clock processes in order to have the timed system *AbstractTimedFireControl*. They synchronise in the events used by the clock process, which we group in the set of channels *ClockSignals*.

$$\begin{array}{l} \textbf{chanset}\ ClockSignals == \{\!|\ startClock, clockFinished\ |\!\} \\ \textbf{process}\ AbstractTimedFireControl \mathrel{\widehat=} \\ \qquad (AbstractFireControl\ [\![\ ClockSignals\ ]\!]\ Clock) \setminus ClockSignals \end{array}$$

Now, we are ready to specify the abstract main system process, which is a parallel composition between the abstract timed fire control system and the external devices.

$$\textbf{process}\ AbstractMain \mathrel{\widehat=} \left(\begin{array}{l} AbstractTimedFireControl \\ [\![ExternalSignals]\!] \\ ExternalDevices \end{array} \right) \setminus ExternalSignals$$

In the next section, we refine the abstract fire control.

5.6 A Refinement Strategy for the Fire Control System

In this section, we aim to refine the main component of the *AbstractMain* process: the process *AbstractFireControl*. Section 5.6.1 presents the target of our refinement, the concrete fire control system. Then, in the following sections, we present the refinement steps summarised graphically in Figure 5.2. Our strategy is based on that proposed in [1] for *Circus*.

The refinement strategy presented for *Circus* is based on laws of simulation, and action and process refinement. It includes (possibly several) iterations of three steps: use of simulation to introduce elements of the concrete system state, action refinement for partitioning the process state and actions, and a process refinement to decompose the original process in two or more processes.

The motivation for the fire control system refinement is the distribution of the areas. This leads to a need of distributed processing in order to get a more efficient system.

First, in Section 5.6.2, we present a data refinement to introduce a new state component ($mode_A$) to the system. The second refinement step, presented in Section 5.6.3, consists in an action refinement that allows the partitioning of the state space and the accompanying actions, in such a way that each partition groups some state elements and the actions which access (read or update) these elements. The idea is to separate the management of the areas from the fire control system. Following this action refinement, in Section 5.6.4 we actually make a process refinement in order to upgrade each partition into separated processes (*InternalSystem* and *Areas*). Next, in Sections 5.6.5 and 5.6.7, we refine the *Areas* process in order to get separated processes to model each existing area within the system; and finally, in Sections 5.6.7 and 5.6.8, we refine the *InternalSystem*, in the same way we do with process $FireControl_1$, in order to get two processes: the fire control system itself (process *FireControl*), and a display controller (process *DisplayController*).

First, however, in Section 5.6.1 below, we present the final concrete system we want to obtain. Further refinement of this system to code only requires refinement of actions.

5.6.1 Concrete Fire Control System

The concrete fire control system is composed of three components: the fire control system, the display controller, and the detection system. In the following sections we present each process separately, and then, we compose them to get the final concrete fire control system. Before proceeding, however, we introduce the internal channels used by the fire control system.

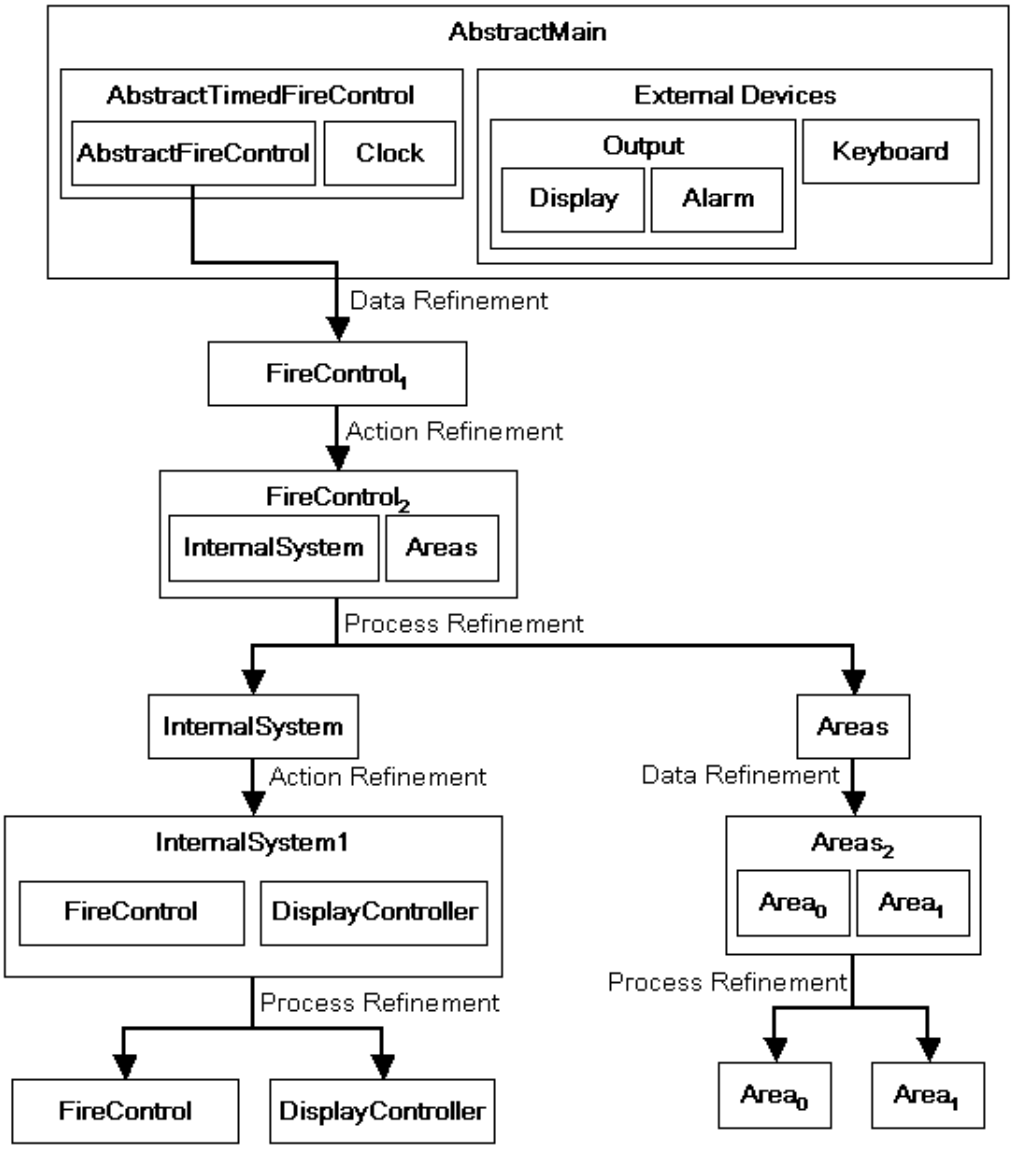

Figure 5.2: Refinement Strategy for the Fire Control System

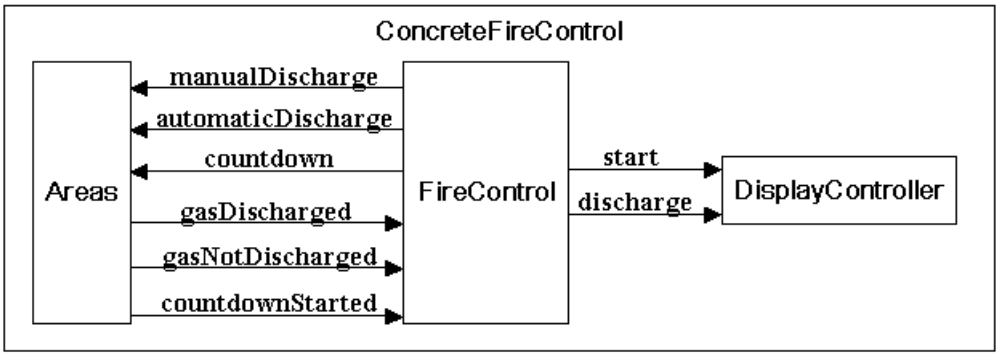

Figure 5.3: Concrete Fire Control

The fire control system may request to the display controller to restart using channel *start*. Besides, the fire control system indicates discharges to the display controller using channel *discharge*.

channel *start*
channel *discharge* : *AreaId*

The fire control system request a gas discharge to the detection process through channels *manualDischarge* and *automaticDischarge*.

channel *manualDischarge*, *automaticDischarge* : *AreaId*

The detection process may reply to these requests indicating if the gas has been discharged or not.

channel *gasDischarged*, *gasNotDischarged* : *AreaId*

When an area is active in *automatic* mode, it may request a countdown using the channel below.

channel *countdown*

The fire control system then replies indicating if the countdown has started (*true*) or not (*false*) through channel *countdownStarted*.

channel *countdownStarted* : *Bool*

In Figure 5.3 we summarise the internal communication of the concrete fire control system.

Fire Control

The process *FireControl* is similar to the its abstract specification (*AbstractFireControl*). However, all the state components and events related to the detection areas and to the display controller are removed from the fire control. They now compose the state of the process presented later.

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

state

The state of the concrete fire control is composed by only one component, *mode*, which indicates the mode in which the fire control is running (*automatic*, *manual*, or *disabled*).

FireControlState

$$mode : Mode$$

As in the abstract specification, the system is initialised in *automatic* mode.

InitFireControl

$$FireControlState'$$

$$mode' = automatic$$

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

SwitchFireControlMode

$$\Delta FireControlState$$
$$newMode? : Mode$$

$$mode' = newMode?$$

The second and third operations, that switch the system mode do not receive any argument. They simply switch the system mode to *automatic* and *disabled*, respectively.

SwitchFireControl2AutomaticMode

$$\Delta FireControlState$$

$$mode' = automatic$$

SwitchFireControl2DisabledMode

$\Delta FireControlState$

$mode' = disabled$

The fire control system is responsible for communicating the current system state. For this reason, almost all actions start with this communication. After being switched on, the fire control only initialises its state and behaves like action *FireSys*. All the events related to the display lamps are no longer controlled by the fire control, but by the display controller that we shall specify later in this section.

$$FireSysStart \mathrel{\widehat=} systemState!fireSysStart_s \rightarrow switchOn \rightarrow InitFireControl; FireSys$$

The action *FireSys* is slightly different from its abstract version. As explained before, it does not make any reference to the display lamps. For this reason, the concrete fire control does not engage in the *fault* events. These are now part of the display controller. After communicating the system state, the mode can be switched between *automatic* and *manual*. Furthermore, if any detection occurs, the system behaves like *Auto* or *Manual*, depending on the current system mode. Since the areas are the processes which have the area-zone information, following a *detection* communication, the zone activation is no longer part of the fire control behaviour. Finally, if the system is requested to reset, its state is initialised.

$$\begin{aligned}
&FireSys \mathrel{\widehat=} \\
&\quad systemState!fireSys_s \rightarrow \\
&\qquad modeSwitch?newMode : SwitchMode \rightarrow SwitchFireControlMode; \\
&\qquad\quad FireSys \\
&\qquad \Box\ detection?newZone : ZoneId \rightarrow alarm!firstStage \rightarrow \\
&\qquad\quad (mode = manual)\ \&\ Manual \\
&\qquad\quad \Box\ (mode = automatic)\ \&\ Auto \\
&\qquad \Box\ reset \rightarrow alarm!alarmOff \rightarrow InitFireControl; FireSys
\end{aligned}$$

After a detection, and if running in *manual* mode, the system behaves like action *Manual*. For the reasons explained above, fault detection events (*fault*) are no longer part of its specification. As the information of the active zones and areas are not within the fire control anymore, in other to decide what to do next, it actually has to communicate (*manualDischarge*) with the process that corresponds to the area in which a gas discharge was manually requested. It then receives an answer from this area: if the discharge actually has taken place (*gasDischarged*), the fire control is set to *disabled* mode, and waits to be reset; otherwise, if the gas

was not discharged (*gasNotDischarged*), it recurses.

$$
\begin{array}{l}
Manual \mathrel{\widehat=} \\
\quad systemState!manual_s \rightarrow \\
\quad\quad detection?newZone : ZoneId \rightarrow Manual \\
\quad\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset \\
\quad\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad\quad manualDischarge.area \rightarrow \\
\quad\quad\quad\quad gasDischarged.area \rightarrow discharge!area \rightarrow \\
\quad\quad\quad\quad\quad SwitchFireControl2DisabledMode;\ Reset \\
\quad\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Manual
\end{array}
$$

After a detection, and if running in *automatic* mode, the concrete system behaves like *Auto*. However, it no longer verifies if any area is active in order to start a countdown, or not. If there is any active area, this area requests the start of a countdown (*countdown*). The fire control confirms the countdown start with the event *countdownStarted*!*true*. Then, as the abstract system, the second stage alarm is sounded and the countdown starts (action *Countdown*). However, if no area is active, the event *countdown* does not happen. In this case, the system may still be reset, in which case the alarm is switched off and the system is initialised. Detections are also allowed, but do not change the behaviour of this action.

$$
\begin{array}{l}
Auto \mathrel{\widehat=} \\
\quad systemState!auto_s \rightarrow \\
\quad\quad countdown \rightarrow countdownStarted!true \rightarrow alarm!secondStage \rightarrow \\
\quad\quad\quad Countdown \\
\quad\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow InitFireControl;\ FireSys \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow Auto
\end{array}
$$

As in the abstract fire control system, the actuators may be replaced when the system is waiting to be reset (action *Reset*). However, the only action that follows such a communication is the switching of the system mode to *automatic*. The behaviour in the case of any detection is the same as for action *Manual*. Finally, if the system is reset, it behaves like *FireSys* or *FireSysD*, depending on the current system mode. Any request to start a countdown (*countdown*) originated from any area is not confirmed (*countdownStarted*!*false*) since no countdown has actually

started.

$$
\begin{array}{l}
Reset \mathrel{\widehat{=}} \\
\quad systemState!reset_s \rightarrow \\
\quad\quad actuatorsReplaced \rightarrow SwitchFireControl2AutomaticMode;\ Reset \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow Reset \\
\quad\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad\quad (mode = disabled)\ \&\ FireSysD \\
\quad\quad\quad \Box\ (mode \neq disabled)\ \&\ InitFireControl;\ FireSys \\
\quad\quad \Box\ countDown \rightarrow countdownStarted!false \rightarrow Reset
\end{array}
$$

The action *Countdown* has almost the same definition as the abstract action *AbstractCountdown*. The only difference is that, after indicating the current system state and requesting the start of a countdown, it behaves as the concrete action *WaitingClock*. While waiting for the clock to finish (event *clockFinished*), detections lead to the same behaviour as for action *Reset*. Any request to start a countdown is confirmed, since this action is already waiting a countdown. When the clock is finished, the fire control discharges the gas (action *Discharge*).

$$
\begin{array}{l}
Countdown \mathrel{\widehat{=}}\ systemState!countdown_s \rightarrow startClock \rightarrow WaitingClock \\
\\
WaitingClock \mathrel{\widehat{=}} \\
\quad clockFinished \rightarrow Discharge \\
\quad \Box\ detection?newZone : ZoneId \rightarrow WaitingClock \\
\quad \Box\ countDown \rightarrow countdownStarted!true \rightarrow WaitingClock
\end{array}
$$

In action *FireSysD*, if the actuators are replaced (*actuatorsReplaced*) the alarm is switched off, and the display controller (*start*) and the fire control system are restarted. Again, any request to start a countdown is rejected.

$$
\begin{array}{l}
FireSysD \mathrel{\widehat{=}} \\
\quad systemState!fireSysD_s \rightarrow \\
\quad\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow start \rightarrow \\
\quad\quad\quad InitFireControl;\ FireSys \\
\quad\quad \Box\ countdown \rightarrow countdownStarted!false \rightarrow FireSysD
\end{array}
$$

Finally, the action *Discharge* represents an automatic discharge. It waits for an indication that there is no one in the area (event *exit*). However, when this indication is received, it sequentially requests an gas discharge to each area process (*automaticDischarge*) and receives an answer: if the gas has been discharged in the area (*gasDischarged*), it indicates the discharge to the display controller (*discharge*), and increments a local variable *log* by one; otherwise, nothing is done. After communicating with all areas, it verifies if the local variable *log* is greater than 0, in

which case, a discharge has happened in, at least, one area. In this case, the system mode is switched to *disabled*. Nevertheless, if *log* is equals to 0, no gas has been discharged, and the system mode is switched to *automatic*. Finally, the system waits to be reset (*Reset*).

$$
\begin{array}{l}
Discharge \mathrel{\widehat=} \\
\quad systemState!discharge_s \rightarrow exit \rightarrow \\
\quad\quad (\textbf{var}\ log : \mathbb{N} \bullet \\
\quad\quad\quad log := 0; \\
\quad\quad\quad (\fatsemi\ area : AreaId \bullet \\
\quad\quad\quad\quad automaticDischarge.area \rightarrow \\
\quad\quad\quad\quad\quad gasDischarged.area \rightarrow discharge!area \rightarrow log := log + 1 \\
\quad\quad\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip); \\
\quad\quad\quad ((log = 0)\ \&\ SwitchFireControlSystem2AutomaticMode \\
\quad\quad\quad \Box\ (log > 0)\ \&\ SwitchFireControlSystem2DisabledMode); \\
\quad\quad\quad Reset
\end{array}
$$

Display Controller

The display controller is responsible for, given an event, make the request to the corresponding lamp to switch on or off.

$$\textbf{process}\ DisplayController \mathrel{\widehat=} \textbf{begin}$$

It has no state: only actions are defined within it. The first action, *InitDisplay*, waits for the system to be switched on, and then, it switches on the system on lamp.

$$
\begin{array}{l}
InitDisplay \mathrel{\widehat=} \\
\quad switchOn \rightarrow switchLamp[LampId].systemOnLamp!on \rightarrow Skip
\end{array}
$$

The next action, *SwitchLampsOff*, is invoked if the system is reset or if the fire control requests the display controller to restart. It switches off the display buzzer and all the lamps (except the circuit fault and the system on lamps).

$$
\begin{array}{l}
SwitchLampsOff \mathrel{\widehat=} \\
\quad switchBuzzer!off \rightarrow Skip \\
\quad ||| (|||\ id : (LampId \setminus \{circuitFaultLamp, systemOnLamp\}) \bullet \\
\quad\quad switchLamp[LampId].id!off \rightarrow Skip) \\
\quad ||| (|||\ zone : ZoneId \bullet switchLamp[ZoneId].zone!off \rightarrow Skip) \\
\quad ||| (|||\ area : AreaId \bullet switchLamp[AreaId].area!off \rightarrow Skip)
\end{array}
$$

Finally, the *DisplayCycle* consists of receiving an event and switching on the corresponding lamp in the display. Besides, if the fire control requests the display

controller to restart (*start*), or if the system is reset (*reset*), the display controller behaves like action *SwitchLampsOff*.

$$
\begin{array}{l}
DisplayCycle \mathrel{\widehat=} \\
\quad detection?zone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].zone!on \rightarrow DisplayCycle \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow DisplayCycle \\
\quad \Box\ silenceAlarm \rightarrow \\
\qquad switchLamp[LampId].alarmSilencedLamp!on \rightarrow DisplayCycle \\
\quad \Box\ discharge?area : AreaId \rightarrow switchLamp[AreaId].area!on \rightarrow \\
\qquad switchLamp[LampId].circuitFaultLamp!on \rightarrow DisplayCycle \\
\quad \Box\ reset \rightarrow SwitchLampsOff ; DisplayCycle \\
\quad \Box\ actuatorsReplaced \rightarrow \\
\qquad switchLamp[LampId].circuitFaultLamp!off \rightarrow DisplayCycle \\
\quad \Box\ start \rightarrow SwitchLampsOff ; DisplayCycle
\end{array}
$$

The main action of the display controller waits the system to be switched on. Then, it switches on the system on lamp, and starts the display cycle.

$$\bullet\ InitDisplay;\ DisplayCycle$$

end

Areas

An area's state is composed of the controlled zones of the area, the active zones in which a fire detection has been made, a boolean *discharge* that records whether a gas discharge has been made in this area, a boolean *active* that records whether a gas discharge may occur in this area, and finally, the mode in which the area is running (*manual*, *automatic*, *disabled*).

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

$$
\begin{array}{|l}
\textit{AreaState} \\
\hline
mode : Mode \\
controlledZones : \mathbb{P}\, ZoneId \\
activeZones : \mathbb{P}\, ZoneId \\
discharge : Bool \\
active : Bool \\
\hline
controlledZones = getZones(id) \\
mode = automatic \Rightarrow \\
\quad active = true \Leftrightarrow \\
\quad\quad \exists\, z_1, z_2 : controlledZones \bullet \\
\quad\quad\quad z_1 \neq z_2 \land \{z_1, z_2\} \subseteq activeZones \\
mode = manual \Rightarrow \\
\quad active = true \Leftrightarrow \\
\quad\quad \exists\, z : controlledZones \bullet z \in activeZones \\
activeZones \subseteq controlledZones \\
\hline
\end{array}
$$

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

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

$$
\begin{array}{|l}
\textit{InitArea} \\
\hline
AreaState' \\
\hline
activeZones' = \emptyset \\
discharge' = false \\
mode' = automatic \\
\hline
\end{array}
$$

A zone can be active using the operation *ActivateZone*. If the given zone is a controlled zone of this area, it is included in the *activeZones* state component.

ActivateZone

$\Delta AreaState$
$newZone? : ZoneId$

$newZone? \in controlledZones$
$activeZones' = activeZones \cup \{newZone?\}$
$discharge' = discharge$
$mode' = mode$

In order to activate the discharge within an area, we must use the operation *ActivateDischarge* that is defined below.

ActivateDischarge

$\Delta AreaState$

$activeZones' = activeZones$
$discharge' = true$
$mode' = mode$

Finally, the last schema allows the area mode to be switched.

SwitchAreaMode

$\Delta AreaState$
$newMode? : Mode$

$activeZones' = activeZones$
$discharge' = discharge$
$mode' = newMode?$

In order to start, an area must synchronise in the *switchOn* event. Its state is then initialised, and the area actually starts working.

$$StartArea \mathrel{\widehat=} switchOn \rightarrow InitArea; AreaCycle$$

In its initial stage, if the *reset* event occurs, the state is initialised; if the system mode is switched, so is the area mode; if an automatic gas discharge is requested, it refuses the discharge (*gasNotDischarge*); and finally, any detection leads to an

analysis of this detection.

$$
\begin{array}{l}
AreaCycle \mathrel{\widehat{=}} \\
\quad \left(\begin{array}{l}
reset \rightarrow InitArea \\
\Box\ modeSwitch?newMode : SwitchMode \rightarrow SwitchAreaMode \\
\Box\ automaticDischarge.id \rightarrow gasNotDischarged.id \rightarrow Skip \\
\Box\ manualDischarge.id \rightarrow gasNotDischarged.id \rightarrow Skip \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad AnalyseDetection(newZone); NextAction(newZone)
\end{array} \right) ; \\
\quad AreaCycle
\end{array}
$$

$$
\begin{array}{l}
AnalyseDetection \mathrel{\widehat{=}} (newZone : ZoneId \bullet \\
\quad (newZone \in controlledZones)\ \&\ ActivateZone \\
\quad \Box\ (newZone \notin controlledZones)\ \&\ Skip)
\end{array}
$$

$$
\begin{array}{l}
NextAction \mathrel{\widehat{=}} (newZone : ZoneId \bullet \\
\quad (newZone \in controlledZones)\ \&\ ActiveArea \\
\quad \Box\ (newZone \notin controlledZones)\ \&\ Skip)
\end{array}
$$

If the zone in which the detection occurred is controlled by this area, it activates the detected zone and takes the area to the active stage; otherwise, no further action is taken. After the analysis of the detection, if any zone was active, the area becomes active. Otherwise, it recurses in the *AreasCycle*.

The behaviour of an active area depends on the mode in which the area is running. If it is running in *automatic* mode, it checks whether the area is active or not. If the area is active, it requests to the fire control system to start a countdown. If the fire control system starts the countdown, the area waits the order to discharge the gas; otherwise, it goes to a disabled stage. If the area is running in *automatic* mode, but it is not active, it may be reset. Furthermore, any detection is analysed as in the *AreaCycle* action, and any request to discharge gas is refused (*gasNotDischarged*).

If the area is running in a *manual* mode, the *reset* and *detection* events are treated in the same way as if the area were not active and running in *automatic* mode. However, if a gas discharge is manually requested, it checks whether the area is active or not. If it is active, then it accepts the discharge (*gasDischarged*), activates the discharge, and goes to a disabled stage; otherwise, it refuses the dis-

charge (*gasNotDischarged*), and remains in the active stage.

$$
\begin{array}{l}
ActiveArea \mathrel{\widehat{=}} \\
\quad (mode = automatic)\ \&\ \\
\quad\quad (active = true)\ \&\ \\
\quad\quad\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad\quad\quad (answer = true)\ \&\ WaitingDischarge \\
\quad\quad\quad\quad \Box\ (answer = false)\ \&\ DisabledArea \\
\quad\quad \Box\ (active = false)\ \&\ \\
\quad\quad\quad reset \rightarrow InitArea \\
\quad\quad\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad\quad\quad AnalyseDetection(newZone); ActiveArea \\
\quad\quad\quad \Box\ automaticDischarge.id \rightarrow gasNotDischarged.id \rightarrow ActiveArea \\
\quad \Box\ (mode = manual)\ \&\ \\
\quad\quad reset \rightarrow InitArea \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad\quad AnalyseDetection(newZone); ActiveArea \\
\quad\quad \Box\ manualDischarge.id \rightarrow \\
\quad\quad\quad (active = true)\ \&\ \\
\quad\quad\quad\quad gasDischarged.id \rightarrow ActivateDischarge;\ DisabledArea \\
\quad\quad\quad \Box\ (active = false)\ \&\ \\
\quad\quad\quad\quad gasNotDischarged.id \rightarrow ActiveArea
\end{array}
$$

When waiting for discharge, any new detection in a controlled zone leads to its activation. When the gas discharge request is sent by the fire system, a *gasDischarged* answer is sent back to the fire system, the gas is discharged, and the area becomes disabled.

$$
\begin{array}{l}
WaitingDischarge \mathrel{\widehat{=}} \\
\quad detection?newZone : ZoneId \rightarrow \\
\quad\quad AnalyseDetection(newZone); WaitingDischarge \\
\quad \Box\ automaticDischarge.id \rightarrow gasDischarged.id \rightarrow \\
\quad\quad ActivateDischarge;\ DisabledArea
\end{array}
$$

In the disabled stage, detections in any zone controlled by the area leads to its activation, and a *reset* event initialises the area state. Furthermore, any request to a gas discharge is refused (*gasNotDischarged*).

$$
\begin{array}{l}
DisabledArea \mathrel{\widehat{=}} \\
\quad reset \rightarrow InitArea \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad AnalyseDetection(newZone); DisabledArea \\
\quad \Box\ automaticDischarge.id \rightarrow gasNotDischarged.id \rightarrow DisabledArea
\end{array}
$$

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

• *StartArea*

end)

The process *Areas* represents all the areas within the system. Basically, it is a parallel composition of all areas. They synchronise in the start up of the system (*switchOn*), and in the *reset*, *modeSwitch* and *detection* events.

process $Areas \mathrel{\widehat=}$
$\quad \Vert\ id : AreaId \ [\![\{\!|\ switchOn, reset, switchMode, detection\ |\!\}]\!] \bullet Area(id)$

The Concrete Main Process

First, we define the internal system, which is defined as the parallel composition of the fire control and the display panel. All the communication between the fire control and the display controller (**chanset** *DisplayComm*) is hidden.

chanset $DisplayComm == \{\!|\ discharge, start\ |\!\}$
chanset $Intervention == \{\!|\ silenceAlarm, actuatorsReplaced\ |\!\}$
chanset $\Sigma_1 == \{\!|\ switchOn, reset, detection\ |\!\} \cup$
$\qquad DisplayComm \ \cup \ Detection \ \cup \ Intervention$

process $InternalSystem \mathrel{\widehat=}$
$\quad FireControl \ [\![\Sigma_1]\!] \ DisplayController \setminus DisplayComm$

Next, we define the concrete fire control as the combination of the internal system and the areas. All the gas discharge synchronisation signals (*GasDischargeSync*) are hidden, since they are communications between the internal system and the areas only.

chanset $GasDischargeSync ==$
$\quad \{\!|\ manualDischarge, automaticDischarge, countdown,$
$\quad\ countdownStarted, gasDischarged, gasNotDischarged\ |\!\}$
chanset $\Sigma_2 ==$
$\quad \{\!|\ switchOn, reset, detection, switchMode\ |\!\} \cup GasDischargeSync$

process $ConcreteFireControl \mathrel{\widehat=}$
$\quad (InternalSystem \ [\![\Sigma_2]\!] \ Areas) \setminus GasDischargeSync$

The concrete timed fire control system is defined as the abstract one, but replacing the abstract fire control by the concrete fire control defined above.

process $ConcreteTimedFireControl \mathrel{\widehat=}$
$\quad (ConcreteFireControl \ [\![ClockSignals]\!] \ Clock) \setminus ClockSignals$

Now, we are ready to specify the concrete main system process, which is a parallel composition between the timed fire control system and the external devices.

$$\textbf{process}\ ConcreteMain \mathrel{\widehat=} \left(\begin{array}{l} ConcreteTimedFireControl \\ [\![ExternalSignals]\!] \\ ExternalDevices \end{array} \right) \setminus ExternalSignals$$

Since the refinement calculus is compositional, in order to prove that the *ConcreteMain* process is a refinement of the *AbstractMain* process, it is enough to prove that process *ConcreteFireControl* is a refinement of process *AbstractFireControl*. In the following sections, we aim to prove this refinement.

5.6.2 Data refinement: including a new state component

In this step we make a data refinement in order to introduce a state component that is used by the fire control. The new $mode_A$ component indicates the mode in which the areas are running. The process *AbstractFireControl* is refined by process $FireControl_1$ presented below.

$$\textbf{process}\ FireControl_1 \mathrel{\widehat=} \textbf{begin}$$

state

$FireControlState_1$

$$\begin{array}{l} mode_1 : Mode \\ controlledZones_1 : AreaId \rightarrow \mathbb{P}\, ZoneId \\ activeZones_1 : AreaId \rightarrow \mathbb{P}\, ZoneId \\ discharge_1 : AreaId \rightarrow Bool \\ active_1 : AreaId \rightarrow Bool \\ mode_A : Mode \\ \hline controlledZones_1 = \{area : AreaId \bullet area \mapsto getZones(area)\} \\ \forall\, area : AreaId \bullet \\ \quad (mode_1 = automatic) \Rightarrow \\ \quad\quad active_1(area) = true \Leftrightarrow \\ \quad\quad\quad \exists\, z_1, z_2 : controlledZones_1(area) \bullet \\ \quad\quad\quad\quad z_1 \neq z_2 \land \{z_1, z_2\} \subseteq activeZones_1(area) \\ \quad \land\, (mode_1 = manual) \Rightarrow \\ \quad\quad active_1(area) = true \Leftrightarrow \\ \quad\quad\quad \{area : AreaId \mid \exists\, z : controlledZones_1(area) \bullet \\ \quad\quad\quad\quad z \in activeZones_1(area)\} \\ \quad \land\, activeZones_1(area) \subseteq controlledZones_1(area) \end{array}$$

The state $FireControlState_1$ is the same as that of *AbstractFireControl*, except that it includes the extra component $mode_A$; the state invariant is the same. In order

to prove that the $FireControl_1$ is a refinement of the *AbstractFireControl* we have to prove that there exists a forwards simulation between the $FireControl_1$ main action and the *AbstractFireControl* main action. The retrieve relation is very simple: it relates each component in the *AbstractFireControlState* to a corresponding component in the $FireControlState_1$. It is defined as follows.

RetrFireControl

$$
\begin{array}{l}
AbstractFireControlState \\
FireControlState_1 \\
\hline
mode_1 = mode \\
controlledZones_1 = controlledZones \\
activeZones_1 = activeZones \\
discharge_1 = discharge \\
active_1 = active
\end{array}
$$

Now, we refine each schema, using the schema expressions simulation law (C.1), in order to deal with the new state. The first schema to be refined is the initialisation schema. In the concrete initialisation, the new state component $mode_A$ is initialised in *automatic* mode.

$InitFireControl_1$

$$
\begin{array}{l}
FireControlState_1' \\
\hline
mode_1' = automatic \\
activeZones_1' = \{area : AreaId \bullet area \mapsto \emptyset\} \\
discharge_1' = \{area : AreaId \bullet area \mapsto false\} \\
mode_A' = automatic
\end{array}
$$

We state the simulations as lemmas.

Lemma 5.1 $InitAbstractFireControl \preceq InitFireControl_1$
Proof.

As already mentioned, this result can be established with an application of Law C.1. It raises two proof obligations. The first one concerns the preconditions of both schemas.

$$
\forall AbstractFireControlState;\ FireControlState_1 \bullet \\
\quad RetrFireControl \land \text{pre } InitAbstractFireControl \Rightarrow \text{pre } InitFireControl_1
$$

This proof obligation is easily proved, since the precondition of both schemas are *true*. The second proof obligation concerns the finalisation of both schemas.

$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet \\ RetrFireControl \land \text{pre}\ InitAbstractFireControl \land InitFireControl_1 \Rightarrow \\ \exists\, AbstractFireControlState' \bullet \\ RetrFireControl' \land InitAbstractFireControl$$

This proof obligation can be also easily discarded using the one-point rule. When this rule is applied, we may remove the universal quantifier, and then, we are left with a predicate in which the consequent of the implication is present in its antecedent. The proof of this lemma, and all the simulation lemmas that follow, are very similar and simple. They can be found in Appendix B.

The next schema to be refined is the *SwitchAbstractFireControlMode*. In the concrete operation, besides the update of component $mode_1$, we have also the update of the component $mode_A$.

$SwitchFireControlMode_1$

$$\Delta FireControlState_1 \\ newMode? : Mode$$

$$mode_1' = newMode? \\ activeZones_1' = activeZones_1 \\ discharge_1' = discharge_1 \\ mode_A' = newMode?$$

Lemma 5.2 $SwitchAbstractFireControlMode \preceq SwitchFireControlMode_1$

The refinement of schema *SwitchAbstractFireControl*2*AutomaticMode* is very similar. However, the new state component is not changed by this schema operation.

$SwitchFireControl2AutomaticMode_1$

$$\Delta FireControlState_1$$

$$mode_1' = automatic \\ activeZones_1' = activeZones_1 \\ discharge_1' = discharge_1 \\ mode_A' = mode_A$$

Lemma 5.3

$$SwitchAbstractFireControl2AutomaticMode \preceq \\ SwitchFireControl2AutomaticMode_1$$

The refinement of schema *SwitchAbstractFireControl2DisabledMode*, as that of the previous schema, is very simple. It also does not change the new state component $mode_A$.

$$
\begin{array}{|l}
\hline
SwitchFireControl2DisabledMode_1 \\
\Delta FireControlState_1 \\
\hline
mode'_1 = disabled \\
activeZones'_1 = activeZones_1 \\
discharge'_1 = discharge_1 \\
mode'_A = mode_A \\
\hline
\end{array}
$$

Lemma 5.4

$$SwitchAbstractFireControl2DisabledMode \preceq SwitchFireControl2DisabledMode_1$$

Next, we have the refinement of the schema *AbstractActivateZone*.

$$
\begin{array}{|l}
\hline
ActivateZone_1 \\
\Delta FireControlState_1 \\
newZone? : ZoneId \\
\hline
mode'_1 = mode_1 \\
activeZones'_1 = activeZones_1 \oplus \\
\quad \{area : AreaId \mid newZone? \in controlledZones_1(area) \bullet \\
\qquad area \mapsto activeZones_1(area) \cup \{newZone?\}\} \\
discharge'_1 = discharge_1 \\
mode'_A = mode_A \\
\hline
\end{array}
$$

Lemma 5.5 $AbstractActivateZone \preceq ActivateZone_1$

The last schema to be refined is the schema *AbstractActivateDischarge*. It also leaves the new state component $mode_A$ unchanged.

$$
\begin{array}{|l}
\hline
ActivateDischarge_1 \\
\Delta FireControlState_1 \\
\hline
mode'_1 = mode_1 \\
activeZones'_1 = activeZones_1 \\
discharge'_1 = discharge_1 \oplus \\
\quad \{area : AreaId \mid area \in \mathrm{dom}\, active_1 \rhd \{true\} \bullet area \mapsto true\} \\
mode'_A = mode_A \\
\hline
\end{array}
$$

Lemma 5.6 $AbstractActivateDischarge \preceq ActivateDischarge_1$

Now, we refine the actions of the *AbstractFireControl*. In this part of the refinement, we rely on the fact that forwards simulation distributes through action constructors (laws of simulation in Appendix C and Appendix D). The new actions have the same structure as the original ones, but use new schema actions (based on Lemmas 5.1 to 5.6). The complete definition of the process $FireControl_1$ can be found in Appendix G. By way of illustration, we present the action $FireSysStart_1$, which simulates of the abstract action *AbstractFireSysStart*.

$$
\begin{array}{l}
FireSysStart_1 \mathrel{\widehat=} \\
\quad systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad\quad InitFireControl_1 ; \; FireSys_1
\end{array}
$$

In this data refinement step, all the output and input values are not changed. For this reason, only the second proviso (action simulation) of the application of Law C.3 must be proved. This law states that, in order to simulate an output prefix, the output of the abstract action must be related to the output of the concrete action using the retrieve relation. Furthermore, the subsequent action in the abstract action must be simulated by the subsequent action in the concrete action. Also, as the guards are not changed, the provisos raised in the application of Law C.4 are only those related to action simulations.

Finally, the main action of the new fire control system, $FireControlSystem_1$, is the simulation of the original action.

- $FireSysStart_1$

end

This concludes this data refinement step.

5.6.3 Action Refinement: decomposing the $FireControl_1$ in two partitions

In this refinement step we change the $FireControl_1$ so that its state is composed of two partitions: one that models the internal system and another one that models the areas. We also change the actions so that the state partitions are handled separately. The final aim, as already discussed, is to split the fire control system into concurrent processes.

process $FireControl_2 \mathrel{\widehat=}$ **begin**

The System State

The internal system state is composed only by the mode in which the internal system is running.

InternalSystemState

$$mode_1 : Mode$$

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

AreasState

$$\begin{aligned}
&mode_A : Mode \\
&controlledZones_1 : AreaId \rightarrow \mathbb{P}\, ZoneId \\
&activeZones_1 : AreaId \rightarrow \mathbb{P}\, ZoneId \\
&discharge_1 : AreaId \rightarrow Bool \\
&active_1 : AreaId \rightarrow Bool \\
\hline
&controlledZones_1 = \{area : AreaId \bullet area \mapsto getZones(area)\} \\
&\forall\, area : AreaId \bullet \\
&\quad (mode_A = automatic) \Rightarrow \\
&\qquad active_1(area) = true \Leftrightarrow \\
&\qquad\quad \exists\, z_1, z_2 : controlledZones_1(area) \bullet \\
&\qquad\qquad z_1 \neq z_2 \land \{z_1, z_2\} \subseteq activeZones_1(area) \\
&\quad \land\, (mode_A = manual) \Rightarrow \\
&\qquad active_1(area) = true \Leftrightarrow \\
&\qquad\quad \{area : AreaId \mid \exists\, z : controlledZones_1(area) \bullet \\
&\qquad\qquad z \in activeZones_1(area)\} \\
&\quad \land\, activeZones_1(area) \subseteq controlledZones_1(area)
\end{aligned}$$

The state of the $FireControlState_1$ is declared as the conjunction of the two previous defined state schemas.

state $FireControlState_1 \mathrel{\widehat=} InternalSystemState \land AreasState$

Internal System Paragraphs

Now, we may declare the first group of paragraphs that access only the internal system components. First, the internal system is initialised in *automatic* mode.

InitInternalSystem

$$\begin{aligned}
&InternalSystemState' \\
&AreasState' \\
\hline
&mode_1' = automatic
\end{aligned}$$

In order to switch the internal system mode, we may use the schema operation *SwitchInternalSystemMode* that receives the new mode as argument.

SwitchInternalSystemMode

$$\Delta InternalSystemState$$
$$\Xi AreasState$$
$$newMode? : Mode$$

$$mode_1' = newMode?$$

The following schema operations switch the internal system mode to *automatic* and to *disabled*, respectively.

SwitchInternalSystem2AutomaticMode

$$\Delta InternalSystemState$$
$$\Xi AreasState$$

$$mode_1' = automatic$$

SwitchInternalSystem2DisabledMode

$$\Delta InternalSystemState$$
$$\Xi AreasState$$

$$mode_1' = disabled$$

The behaviour of this intermediate fire control is very similar to that of the abstract one. However, after being switched on, it only initialises the fire control state components and behaves like action $FireSys_2$. All the operations related to the areas components are no longer controlled by the fire control action, but by the areas actions that we specify later in this section.

$$\begin{aligned} &FireSysStart_2 \mathrel{\widehat=} \\ &\quad systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\ &\quad\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\ &\quad\quad\quad InitInternalSystem;\ FireSys_2 \end{aligned}$$

The action $FireSys_2$ is slightly different from action $FireSys_1$. When a synchronisation on the event *modeSwitch* happens, this action only switches the internal system mode. Furthermore, since the information about the areas are no longer part of this partition, following a *detection* communication, this action does not activate the area in which the detection occurred. If the system is reset, this action

only initialises the internal system.

$$
\begin{array}{l}
FireSys_2 \mathrel{\widehat=} \\
\quad systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\qquad SwitchInternalSystemMode;\ FireSys_2 \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\qquad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\qquad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow FireSys_2 \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\qquad InitInternalSystem;\ SwitchLampsOff_2; FireSys_2
\end{array}
$$

The action $SwitchLampsOff_2$ is the same the action $SwitchLampsOff_1$. We have changed its name just for a standardisation of action names.

$$
\begin{array}{l}
SwitchLampsOff_2 \mathrel{\widehat=} \\
\quad (switchBuzzer!off \rightarrow Skip \\
\quad |||\ id : (LampId \setminus \{circuitFaultLamp, systemOnLamp\}) \bullet \\
\qquad switchLamp[LampId].id!off \rightarrow Skip \\
\quad |||\ zone : ZoneId \bullet switchLamp[ZoneId].zone!off \rightarrow Skip \\
\quad |||\ area : AreaId \bullet switchLamp[AreaId].area!off \rightarrow Skip)
\end{array}
$$

In the detection of fire, the action $Manual_2$ also differs from action $Manual_1$: the former does not activate the zone in which the fire is detected. However, its behaviour in case the alarm is silenced or a fault is detected, is almost the same as action $Manual_1$. The main difference is in a manual request for gas discharge: differently from action $Manual_1$, which has the areas information, the action $Manual_2$ requests the gas discharge using a communication through channel *manualDischarge*. If a gas discharge has actually occurred, $Manual_2$ switches on the corresponding lamp, switches the internal system mode to *disabled*, and waits to be reset. If a gas

discharged has not occurred, it recurses.

$$
\begin{array}{l}
Manual_2 \mathrel{\widehat{=}} \\
\quad systemState!manual_s \rightarrow \\
\quad\quad detection?newZone : ZoneId \rightarrow \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\quad\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\quad\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad\quad manualDischarge.area \rightarrow \\
\quad\quad\quad\quad gasDischarged.area \rightarrow switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad\quad SwitchInternalSystem2DisabledMode;\ Reset_2 \\
\quad\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Manual_2 \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow Manual_2
\end{array}
$$

As the areas information is not available in the internal system state, the action $Auto_2$ does not verify if any area is active in order to start a countdown. If there is any active area, it requests the start of a countdown (*countdown*) itself, in which case the fire control confirms the countdown has started. Then, as in action $Auto_1$, the second stage alarm is sounded, and the countdown starts (action $Countdown_2$). The system may still be reset, in which case the alarm is switched off, and the internal system is initialised. Detections and faults are only followed by the corresponding lamp being lit and the buzzer (for faults) being sounded.

$$
\begin{array}{l}
Auto_2 \mathrel{\widehat{=}} \\
\quad systemState!auto_s \rightarrow \\
\quad\quad countdown \rightarrow countdownStarted!true \rightarrow \\
\quad\quad\quad alarm!secondStage \rightarrow Countdown_2 \\
\quad\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad\quad InitInternalSystem;\ FireSys_2 \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}
$$

When the system is waiting to be reset (action $Reset_2$), the actuators may be replaced. In this case, the 'circuit fault' is switched off and the internal system mode is switched to automatic. In the case of any detection or fault, the corresponding lamp is switched on. Finally, if the system is reset, it behaves like $FireSys_2$ or $FireSysD_2$, depending on the current system mode. At any time, the areas may request a countdown to be started. Such requests are not confirmed (*countdownStarted*!*false*), since

no countdown may start here.

$$
\begin{array}{l}
Reset_2 \mathrel{\widehat{=}} \\
\quad systemState!reset_s \rightarrow \\
\quad\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\quad\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad\quad\quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;\ FireSys_2 \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow Reset_2 \\
\quad\quad \Box\ countdown \rightarrow countdownStarted!false \rightarrow Reset_2
\end{array}
$$

As $Countdown_1$, the action $Countdown_2$ indicates the current system state, requests the start of a countdown, and waits for the countdown to be finished. Throughout this waiting time, detections and faults lead to the same behaviour as for action $Reset_2$. Any request to start a countdown is confirmed, since this action is waiting a countdown. When the clock is finished, the fire control discharges the gas.

$$
\begin{array}{l}
Countdown_2 \mathrel{\widehat{=}}\ systemState!countdown_s \rightarrow startClock \rightarrow WaitingClock_2 \\
\\
WaitingClock_2 \mathrel{\widehat{=}} \\
\quad clockFinished \rightarrow Discharge_2 \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow WaitingClock_2 \\
\quad \Box\ countdown \rightarrow countdownStarted!true \rightarrow WaitingClock_2
\end{array}
$$

In action $FireSysD_2$, if the actuators are replaced, the alarm and the display lamps are switched off, and the internal system state is initialised. Any fault switches the corresponding lamp on, and, finally, any request to start a countdown

is rejected.

$$
\begin{array}{l}
FireSysD_2 \mathrel{\widehat=} \\
\quad systemState!fireSysD_s \rightarrow \\
\quad\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad\quad SwitchLampsOff_2; InitInternalSystem;\ FireSys_2 \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow FireSysD_2 \\
\quad\quad \Box\ countdown \rightarrow countdownStarted!false \rightarrow FireSysD_2
\end{array}
$$

The last paragraph of the internal system partition is the action *Discharge*. It waits for an indication that there is no one in the areas (event *exit*), after which, it sequentially requests a gas discharge to each area process, and waits for an answer. A local variable *log* is used to register how many areas have actually discharged gas. After all communications with the areas, if *log* is greater than 0, a discharge has happened. In this case, the system mode is switched to *disabled*. Nevertheless, if *log* is equals to 0, no gas has been discharged, and the system mode is switched to *automatic*. Finally, the system waits to be reset (*Reset*).

$$
\begin{array}{l}
Discharge_2 \mathrel{\widehat=} \\
\quad systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\quad\quad (\mathbf{var}\ log : \mathbb{N} \bullet \\
\quad\quad\quad log := 0; \\
\quad\quad\quad (\fatsemi_{9}\ area : AreaId \bullet \\
\quad\quad\quad\quad automaticDischarge.area \rightarrow \\
\quad\quad\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad\quad\quad switchLamp[AreaId].area!on \rightarrow log := log + 1 \\
\quad\quad\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip); \\
\quad\quad\quad ((log = 0)\ \&\ SwitchInternalSystem2AutomaticMode \\
\quad\quad\quad \Box\ (log > 0)\ \&\ SwitchInternalSystem2DisabledMode)); \\
\quad\quad Reset_2
\end{array}
$$

This concludes the paragraphs of the internal system partition.

The Areas Paragraphs

Now, we present the paragraphs related to the areas. First, the areas are initialised in *automatic* mode. Initially, there are no active zones, no discharge has occurred, and no area is active.

$$
\begin{array}{|l}
\hline
\textit{InitAreas} \\
\textit{AreasState}' \\
\textit{InternalSystemState}' \\
\hline
\textit{mode}'_A = \textit{automatic} \\
\textit{activeZones}'_1 = \{\textit{area} : \textit{AreaId} \bullet \textit{area} \mapsto \emptyset\} \\
\textit{discharge}'_1 = \{\textit{area} : \textit{AreaId} \bullet \textit{area} \mapsto \textit{false}\} \\
\hline
\end{array}
$$

The areas mode $mode_A$ can be switched with the schema operation *SwitchAreasMode* defined below.

$$
\begin{array}{|l}
\hline
\textit{SwitchAreasMode} \\
\Delta \textit{AreasState} \\
\Xi \textit{InternalSystemState} \\
\textit{newMode}? : \textit{Mode} \\
\hline
\textit{mode}'_A = \textit{newMode}? \\
\textit{activeZones}'_1 = \textit{activeZones}_1 \\
\textit{discharge}'_1 = \textit{discharge}_1 \\
\hline
\end{array}
$$

In order to activate any zone, we may use the schema operation *ActivateZoneAS* defined below. It includes the given zone in the set of zones mapped by the area that controls the given zone in the $activeZones_1$ state component.

$$
\begin{array}{|l}
\hline
\textit{ActivateZoneAS} \\
\Delta \textit{AreasState} \\
\Xi \textit{InternalSystemState} \\
\textit{newZone}? : \textit{ZoneId} \\
\hline
\textit{mode}'_A = \textit{mode}_A \\
\textit{activeZones}'_1 = \textit{activeZones}_1 \oplus \\
\quad \{\textit{area} : \textit{AreaId} \mid \textit{newZone}? \in \textit{controlledZones}_1(\textit{area}) \bullet \\
\qquad \textit{area} \mapsto \textit{activeZones}_1(\textit{area}) \cup \{\textit{newZone}?\}\} \\
\textit{discharge}'_1 = \textit{discharge}_1 \\
\hline
\end{array}
$$

The following schema operation can be use to activate the gas discharge within the areas. It overrides the $discharge_1$ component by mapping each area in the $active_1$ component that is active (*true*) to *true*.

$$
\begin{array}{|l}
\hline
\textit{ActivateDischargeAS} \\
\Delta \textit{AreasState} \\
\Xi \textit{InternalSystemState} \\
\hline
\textit{mode}'_A = \textit{mode}_A \\
\textit{activeZones}'_1 = \textit{activeZones}_1 \\
\textit{discharge}'_1 = \textit{discharge}_1 \oplus \\
\quad \{\textit{area} : \textit{AreaId} \mid \textit{area} \in \operatorname{dom} \textit{active}_1 \rhd \{\textit{true}\} \bullet \textit{area} \mapsto \textit{true}\} \\
\hline
\end{array}
$$

In order to start, the areas synchronise in the *switchOn* event and then, the state is initialised and the areas actually start working.

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

Initially, the areas can be reset, in which case the state is initialised; if the system mode is switched, so is the areas mode; any detection leads to the activation of the zone in which fire was detected, if it is controlled by any area, and takes the area of the zone to the active stage; finally, if an automatic or manual gas discharge is requested, it refuses the discharge (*gasNotDischarge*).

$$
\begin{array}{l}
\textit{AreasCycle} \mathrel{\widehat=} \\
\quad (\textit{reset} \rightarrow \textit{InitAreas} \\
\quad \Box\ \textit{modeSwitch?newMode} : \textit{SwitchMode} \rightarrow \textit{SwitchAreasMode} \\
\quad \Box\ \textit{detection?newZone} : \textit{ZoneId} \rightarrow \textit{ActivateZoneAS}; \textit{ActiveAreas} \\
\quad \Box\ \Box\ \textit{area} : \textit{AreaId} \bullet \textit{automaticDischarge.area} \rightarrow \\
\qquad \textit{gasNotDischarged.area} \rightarrow \textit{Skip} \\
\quad \Box\ \Box\ \textit{area} : \textit{AreaId} \bullet \textit{manualDischarge.area} \rightarrow \\
\qquad \textit{gasNotDischarged.area} \rightarrow \textit{Skip}); \\
\quad \textit{AreasCycle}
\end{array}
$$

In action *ActiveAreas*, if the areas are running in *automatic* mode, it checks whether there is any active area. The existence of any active area leads to a request to the fire control system to start a countdown. If the request is granted, the area waits the order to discharge the gas; otherwise, the system goes to a disabled stage. If the area is running in *automatic* mode, but is not active, it may be reset. Furthermore, any detection is analysed as in the *AreasCycle* action, and any request to discharge gas is refused (*gasNotDischarged*).

In *manual* mode, the *reset* and *detection* events are treated in the same way as if the areas were not active and running in *automatic* mode. For any gas discharge manual request, it checked whether the area in which the gas discharge was requested is active or not. If it is active, the area accepts the discharge (*gasDischarged*), activates the discharge, and goes to a disabled stage; otherwise, the area refuses the

discharge (*gasNotDischarged*), and remains in the active stage.

$$
\begin{array}{l}
ActiveAreas \mathrel{\widehat=} \\
\quad (mode_A = automatic)\ \& \\
\quad\quad (active_1 \rhd \{true\} \neq \emptyset)\ \& \\
\quad\quad\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad\quad\quad (answer = true)\ \&\ WaitingDischarge \\
\quad\quad\quad\quad \Box\ (answer = false)\ \&\ DisabledAreas \\
\quad\quad \Box\ (active_1 \rhd \{true\} = \emptyset)\ \& \\
\quad\quad\quad reset \rightarrow InitAreas \\
\quad\quad\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; ActiveAreas \\
\quad\quad\quad \Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\quad\quad\quad\quad gasNotDischarged.area \rightarrow ActiveAreas \\
\quad \Box\ (mode_A = manual)\ \& \\
\quad\quad reset \rightarrow InitAreas \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; ActiveAreas \\
\quad\quad \Box\ \Box\ area : AreaId \bullet \\
\quad\quad\quad manualDischarge.area \rightarrow \\
\quad\quad\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad\quad\quad activateDischargeAS;\ DisabledAreas \\
\quad\quad\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad\quad\quad gasNotDischarged.area \rightarrow ActiveAreas
\end{array}
$$

When waiting for discharge, any new detection in a controlled zone leads to its activation. If any automatic gas discharge is requested by the internal system, the action checks if the area is active or not. If it is active, the discharge is accepted. However, if the area is not active, the discharge is not active. After, receiving and answering all the discharge requests, discharge is active and then the system behaves like *DisabledAreas*.

$$
\begin{array}{l}
WaitingDischarge \mathrel{\widehat=} \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; WaitingDischarge \\
\quad \Box\ ReplyDischarge; DisabledAreas
\end{array}
$$

$$
\begin{array}{l}
ReplyDischarge \mathrel{\widehat=} \\
\quad (|||\ area : AreaId \bullet \\
\quad\quad (automaticDischarge.area \rightarrow \\
\quad\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad\quad gasNotDischarged.area \rightarrow Skip)); \\
\quad ActivateDischargeAS
\end{array}
$$

The *DisabledAreas* action activates any area in which a detection happens, and initialises the areas state if the system is reset. Furthermore, any request to a gas discharge is refused (*gasNotDischarged*).

$$
\begin{array}{l}
DisabledAreas \mathrel{\widehat=} \\
\quad reset \rightarrow InitAreas \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; DisabledAreas \\
\quad \Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\qquad gasNotDischarged.area \rightarrow DisabledAreas
\end{array}
$$

Finally, the main action of the process $FireControl_2$ 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 channels in set Σ_2, which contains all the events *switchOn*, *reset*, *detection*, *switchMode*, and all the channels used only for communication between both partitions; these are the events *manualDischarge*, *automaticDischarge*, *countdown*, *countdownStarted*, *gasDischarged*, and *gasNotDischarged*. They are hidden in the main action as defined below.

$$
\bullet \left(\begin{array}{l} FireSysStart_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ StartAreas \end{array} \right) \setminus GasDischargeSync
$$

end

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 $FireControl_2$ refines the main action of process $FireControl_1$.

Mutual Recursion.

Before proving this refinement, it very is important to notice that throughout this chapter we actually use a syntactic sugaring for mutual recursive actions.

In order to improve the presentation of the refinements and of the processes themselves, we have used a syntactic sugaring for mutual recursive actions. By way of illustration consider the following mutual recursive processes definitions S, S', and S''.

$$
\left\{ \begin{array}{l} S = \mu\, X, Y \bullet F(X, Y) \\ F(X, Y) = [a \rightarrow SExp_1; X \ \Box\ b \rightarrow Y, c \rightarrow SExp_2; Y \ \Box\ d \rightarrow X] \end{array} \right.
$$

$$
\left\{ \begin{array}{l} S' = \mu\, X, Y \bullet F'(X, Y) \\ F'(X, Y) = [a \rightarrow X \ \Box\ b \rightarrow Y, c \rightarrow Y \ \Box\ d \rightarrow X] \end{array} \right.
$$

$$
\left\{ \begin{array}{l} S'' = \mu\, X, Y \bullet F''(X, Y) \\ F''(X, Y) = [a \rightarrow SExp_1; X \ \Box\ b \rightarrow Y, c \rightarrow SExp_2; Y \ \Box\ d \rightarrow X] \end{array} \right.
$$

Now, suppose we want to prove that

$$S \sqsubseteq_{\mathcal{V}} [S'.1 \parallel S''.1, S'.2 \parallel S''.2]$$

where $\sqsubseteq_{\mathcal{V}}$ represents the vectorial refinement defined as follows.

Definition 5.1 (Vectorial Refinement) *For two vector of actions*

$$V_1 = [a_{1_1}, \ldots, a_{1_n}]$$
$$V_2 = [a_{2_1}, \ldots, a_{2_n}]$$

we have that $V_1 \sqsubseteq_{\mathcal{V}} V_2$ *provided* $a_{1_i} \sqsubseteq_{\mathcal{A}} a_{2_i}$ *for all* i *in* $1 \,..\, n$.

The proof of the vectorial refinement presented above can be presented as follows. First, we apply the definition of S, S' and S''.

$$\begin{array}{l} S \sqsubseteq_{\mathcal{V}} [S'.1 \parallel S''.1, S'.2 \parallel S''.2] \\ \widehat{=} [\textit{Definitions of } S, S', \textit{ and } S''] \\ \mu\, X, Y \bullet F(X, Y) \sqsubseteq_{\mathcal{V}} \left[\begin{array}{l} (\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1, \\ (\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2 \end{array} \right] \end{array}$$

Next, we may use a vectorial version of the recursion-least fixed point law.

$$\begin{array}{l} \Leftarrow [\textit{Vectorial version of law } C.56\,(\textit{Recursion} - \textit{Least Fixed Point})] \\ F \left(\begin{array}{l} (\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1, \\ (\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2 \end{array} \right) \\ \sqsubseteq_{\mathcal{V}} \\ \left[\begin{array}{l} (\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1, \\ (\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2 \end{array} \right] \end{array}$$

We start the proof of this refinement by applying the definition of F.

$$\begin{array}{l} F \left(\begin{array}{l} (\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1, \\ (\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2 \end{array} \right) \\ \widehat{=} [\textit{Definition of } F] \\ \left[\begin{array}{l} \left(\begin{array}{l} a \rightarrow \textit{SExp}_1;((\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1) \\ \Box\ b \rightarrow ((\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2) \end{array} \right), \\ \left(\begin{array}{l} c \rightarrow \textit{SExp}_2;((\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2) \\ \Box\ d \rightarrow ((\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1) \end{array} \right) \end{array} \right] \end{array}$$

Next, we distribute the schema over the parallelism as follows.

$$\begin{array}{l} = [D.29, D.28] \\ \{\bigcup_i \textit{wrtV}(\textit{SExp}_i) \subseteq \textit{ns}_1 \cup \textit{ns}'_1\} \\ \{\bigcup_i \textit{wrtV}(\textit{SExp}_i) \cap \textit{usedV}(A_2) = \emptyset\} \\ \left[\begin{array}{l} \left(\begin{array}{l} a \rightarrow ((\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\textit{SExp}_1;(\mu\, X, Y \bullet F''(X, Y)).1)) \\ \Box\ b \rightarrow ((\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2) \end{array} \right), \\ \left(\begin{array}{l} c \rightarrow ((\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\textit{SExp}_2;(\mu\, X, Y \bullet F''(X, Y)).2)) \\ \Box\ d \rightarrow ((\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1) \end{array} \right) \end{array} \right] \end{array}$$

Then, as the channels a, b, c, and d are in the synchronisation channel set, we may apply the distribution of prefix over parallelism law.

$$
\begin{array}{l}
= [C.51] \\
\{\{a, b, c, d\} \subseteq cs\} \\
\left[\begin{array}{l}
\left(\begin{array}{l}
(a \rightarrow (\mu\, X, Y \bullet F'(X, Y)).1) \parallel (a \rightarrow SExp_1;(\mu\, X, Y \bullet F''(X, Y)).1) \\
\Box\ (b \rightarrow (\mu\, X, Y \bullet F'(X, Y)).2) \parallel (b \rightarrow (\mu\, X, Y \bullet F''(X, Y)).2)
\end{array} \right), \\
\left(\begin{array}{l}
(c \rightarrow (\mu\, X, Y \bullet F'(X, Y)).2) \parallel (c \rightarrow SExp_2;(\mu\, X, Y \bullet F''(X, Y)).2) \\
\Box\ (d \rightarrow (\mu\, X, Y \bullet F'(X, Y)).1) \parallel (d \rightarrow (\mu\, X, Y \bullet F''(X, Y)).1)
\end{array} \right)
\end{array} \right]
\end{array}
$$

Next, we apply the exchange of parallelism and external choice law. This application is valid since the initials of all actions are in the synchronisation channel set.

$$
\begin{array}{l}
= [C.45] \\
\{\{a, b, c, d\} \subseteq cs\} \\
\left[\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
a \rightarrow (\mu\, X, Y \bullet F'(X, Y)).1 \\
\Box\ b \rightarrow (\mu\, X, Y \bullet F'(X, Y)).2
\end{array} \right) \\
\parallel \\
\left(\begin{array}{l}
a \rightarrow SExp_1;(\mu\, X, Y \bullet F''(X, Y)).1 \\
\Box\ b \rightarrow (\mu\, X, Y \bullet F''(X, Y)).2
\end{array} \right)
\end{array} \right), \\
\left(\begin{array}{l}
\left(\begin{array}{l}
c \rightarrow (\mu\, X, Y \bullet F'(X, Y)).2 \\
\Box\ d \rightarrow (\mu\, X, Y \bullet F'(X, Y)).1)
\end{array} \right) \\
\parallel \\
\left(\begin{array}{l}
c \rightarrow SExp_2;(\mu\, X, Y \bullet F''(X, Y)).2 \\
\Box\ d \rightarrow (\mu\, X, Y \bullet F''(X, Y)).1
\end{array} \right)
\end{array} \right)
\end{array} \right]
\end{array}
$$

The definition of array allows us to extend the action above as follows.

$$
\begin{array}{l}
\widehat{=} [A = [A, B].1, B = [A, B].2] \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} a \to (\mu\, X, Y \bullet F'(X, Y)).1 \\ \Box\ b \to (\mu\, X, Y \bullet F'(X, Y)).2 \end{array} \right), \\
\left[\begin{array}{l} c \to (\mu\, X, Y \bullet F'(X, Y)).2 \\ \Box\ d \to (\mu\, X, Y \bullet F'(X, Y)).1 \end{array} \right)
\end{array} \right] .1 \\
\quad \| \\
\left[\begin{array}{l}
\left[\begin{array}{l} a \to SExp_1;(\mu\, X, Y \bullet F''(X, Y)).1 \\ \Box\ b \to (\mu\, X, Y \bullet F''(X, Y)).2 \end{array} \right), \\
\left[\begin{array}{l} c \to SExp_2;(\mu\, X, Y \bullet F''(X, Y)).2 \\ \Box\ d \to (\mu\, X, Y \bullet F''(X, Y)).1 \end{array} \right)
\end{array} \right] .1
\end{array} \right], \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} a \to (\mu\, X, Y \bullet F'(X, Y)).1 \\ \Box\ b \to (\mu\, X, Y \bullet F'(X, Y)).2 \end{array} \right), \\
\left[\begin{array}{l} c \to (\mu\, X, Y \bullet F'(X, Y)).2 \\ \Box\ d \to (\mu\, X, Y \bullet F'(X, Y)).1 \end{array} \right)
\end{array} \right] .2 \\
\quad \| \\
\left[\begin{array}{l}
\left[\begin{array}{l} a \to SExp_1;(\mu\, X, Y \bullet F''(X, Y)).1 \\ \Box\ b \to (\mu\, X, Y \bullet F''(X, Y)).2, \end{array} \right), \\
\left[\begin{array}{l} c \to SExp_2;(\mu\, X, Y \bullet F''(X, Y)).2 \\ \Box\ d \to (\mu\, X, Y \bullet F''(X, Y)).1 \end{array} \right)
\end{array} \right] .2
\end{array} \right]
\end{array} \right]
\end{array}
$$

Finally, using a vectorial version of the recursion unfolding law, we conclude our proof.

$$
\begin{array}{l}
= [\textit{Vectorial version of law } C.55\,(\textit{Recursion Unfolding})] \\
\left[\begin{array}{l} (\mu\, X, Y \bullet F'(X, Y)).1 \parallel (\mu\, X, Y \bullet F''(X, Y)).1, \\ (\mu\, X, Y \bullet F'(X, Y)).2 \parallel (\mu\, X, Y \bullet F''(X, Y)).2 \end{array} \right]
\end{array}
$$

□

Proving Refinement on Mutual Recursive Systems. The system and refinement proved above are quite simple to be presented and understood. However, it may be the case that the system is quite complicated to be presented in the notation presented above. Our case study is a good example of such a system. For this reason, we aim at the presentation of all mutual recursive systems and of refinements on these systems in a more concise way.

First, let us generalise the syntactic sugaring for the definitions of mutual recursive systems: every mutual recursive system of the form

$$
\left\{ \begin{array}{l} S \mathrel{\widehat{=}} \mu\, X_0, \ldots, X_n \bullet F(X_0, \ldots, X_n) \\ F(X_0, \ldots, X_n) \mathrel{\widehat{=}} [F_0(X_0, \ldots, X_n), \ldots, F_n(X_0, \ldots, X_n)] \end{array} \right.
$$

can be presented using the following syntactic sugar of S.

$$S_S \mathrel{\widehat{=}} [N_0, \ldots, N_n]$$

For each index i in $0..n$, the action N_i is defined as $N_i \mathrel{\widehat{=}} G_i$, where G_i is defined as its corresponding $F_i(X_0, \ldots, X_n)$, but replacing all the occurrences of the variables $X_0, \ldots, X_n$ by the corresponding syntactic sugaring $N_0, \ldots, N_n$. Furthermore, the new names N_i are fresh names.

$$G_i = F_i[X_0, \ldots, X_n \setminus N_0, \ldots, N_n]$$

Now, we may present the syntactic sugaring for proving refinements on these systems. We want to prove a refinement of the following form, where $Y_0, \ldots, Y_n$ are actions.

$$S \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n]$$

Actually, what happens is that, when trying to prove this property, we apply the vectorial version of the recursion-least fixed point Law ($C.56$) as follows.

$$\begin{array}{l} \Leftarrow [\textit{Vectorial version of law } C.56\,(\textit{Recursion} - \textit{Least Fixed Point})] \\ F(Y_0, \ldots, Y_n) \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n] \end{array}$$

Applying the definition of F we get the following proof obligation.

$$\begin{array}{l} = [\textit{Definition of } F] \\ [F_0(Y_0, \ldots, Y_n), \ldots, F_n(Y_0, \ldots, Y_n)] \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n] \end{array}$$

From the term rewriting theory, we have the following property that we shall use later in this proof.

Property 5.1 *Given a arbitrary term A, we have that:*
$A[X_0, \ldots, X_n \setminus Y_0, \ldots, Y_n][Y_0, \ldots, Y_n \setminus Z_0, \ldots, Z_n] \equiv A[X_0, \ldots, X_n \setminus Z_0, \ldots, Z_n]$
provided $Y_0, \ldots, Y_n \cap FV(A) = \emptyset$.

The previous proof obligation can then be transformed as follows.

$$\begin{array}{l} F_i(Y_0, \ldots, Y_n) \sqsubseteq_{\mathcal{A}} Y_i \\ \equiv [\textit{Function Invocation}] \\ F_i[X_0, \ldots, X_n \setminus Y_0, \ldots, Y_n] \sqsubseteq_{\mathcal{A}} Y_i \\ \equiv [\textit{Property } 5.1]\{N_i \textit{ are fresh names}\} \\ F_i[X_0, \ldots, X_n \setminus N_0, \ldots, N_n][N_0, \ldots, N_n \setminus Y_0, \ldots, Y_n] \sqsubseteq_{\mathcal{A}} Y_i \\ = [\textit{Definition of } G_i] \\ G_i[N_0, \ldots, N_n \setminus Y_0, \ldots, Y_n] \sqsubseteq_{\mathcal{A}} Y_i \end{array}$$

We have then the following proof obligation.

$$
\begin{array}{l}
[G_0[N_0, \ldots, N_n \setminus Y_0, \ldots, Y_n], \ldots, G_n[N_0, \ldots, N_n \setminus Y_0, \ldots, Y_n]] \\
\sqsubseteq_{\mathcal{V}} \\
[Y_0, \ldots, Y_n]
\end{array}
$$

Finally, by the definition of vectorial refinement (Definition 5.1), this refinement is valid if the refinement holds for each corresponding element in both vectors. This concludes our syntactic sugaring for proving refinements on mutual recursive systems.

Definition 5.2 (Refinement on Mutual Recursive Actions) *For a given vector of actions S_S defined in the form $S_S \mathrel{\widehat{=}} [N_0, \ldots, N_n]$, we have that:*

$$
\begin{array}{l}
S_S \sqsubseteq_{\mathcal{A}} [Y_0, \ldots, Y_n] \\
\Leftarrow \\
G_0[N_0, \ldots, N_n \setminus Y_0, \ldots, Y_n] \sqsubseteq_{\mathcal{A}} Y_0, \ldots, G_n[N_0, \ldots, N_n \setminus Y_0, \ldots, Y_n] \sqsubseteq_{\mathcal{A}} Y_n
\end{array}
$$

We are then left with separated proofs that can be presented in a much better way than the original vectorial one.

Example in a Friendly Notation. In our example, we may apply the strategy of syntactic sugaring presented above to get the following syntactic sugaring S_S of the process S.

$$
\begin{array}{l}
S_S \mathrel{\widehat{=}} [N_0, N_1] \\
\\
N_0 \mathrel{\widehat{=}} G_0 \ \mathit{where}\ G_0 = a \rightarrow \mathit{SExp}_1; N_0 \ \Box\ b \rightarrow N_1 \\
N_1 \mathrel{\widehat{=}} G_1 \ \mathit{where}\ G_1 = c \rightarrow \mathit{SExp}_2; N_1 \ \Box\ d \rightarrow N_0
\end{array}
$$

We may also apply the strategy to get the syntactic sugaring S'_S of the processes S'.

$$
\begin{array}{l}
S'_S \mathrel{\widehat{=}} [N'_0, N'_1] \\
\\
N'_0 \mathrel{\widehat{=}} G'_0 \ \mathit{where}\ G'_0 = a \rightarrow N'_0 \ \Box\ b \rightarrow N'_1 \\
N'_1 \mathrel{\widehat{=}} G'_1 \ \mathit{where}\ G'_1 = c \rightarrow N'_1 \ \Box\ d \rightarrow N'_0
\end{array}
$$

In a similar way, we also apply the strategy to get the syntactic sugaring S''_S of the processes S''.

$$
\begin{array}{l}
S''_S \mathrel{\widehat{=}} [N''_0, N''_1] \\
\\
N''_0 \mathrel{\widehat{=}} G''_0 \ \mathit{where}\ G''_0 = a \rightarrow \mathit{SExp}_1; N''_0 \ \Box\ b \rightarrow N''_1 \\
N''_1 \mathrel{\widehat{=}} G''_1 \ \mathit{where}\ G''_1 = c \rightarrow \mathit{SExp}_2; N''_1 \ \Box\ d \rightarrow N''_0
\end{array}
$$

What we want to prove is that

$S_S \sqsubseteq_{\mathcal{V}} [S'_S.1 \parallel S''_S.1, S'_S.2 \parallel S''_S.2]$
$= [Definition\ of\ S'_S\ and S''_S]$
$S_S \sqsubseteq_{\mathcal{V}} [N'_0 \parallel N''_0, N'_1 \parallel N''_1]$

Our refinement strategy, however, gives us the following proving obligations for this refinement.

$\Leftarrow [Definition\, 5.2]$
$[1]\ G_0[N_0, N_1 \setminus N'_0 \parallel N''_0, N'_1 \parallel N''_1] \sqsubseteq_{\mathcal{A}} N'_0 \parallel N''_0$
and
$[2]\ G_1[N_0, N_1 \setminus N'_0 \parallel N''_0, N'_1 \parallel N''_1] \sqsubseteq_{\mathcal{A}} N'_1 \parallel N''_1$

These can be easily proved as follows.

$[1] G_0[N_0, N_1 \setminus N'_0 \parallel N''_0, N'_1 \parallel N''_1] \sqsubseteq_{\mathcal{A}} N'_0 \parallel N''_0$
$= [Definition\ of\ G_0]$
$(a \rightarrow SExp_1; N_0 \,\Box\, b \rightarrow N_1)[N_0, N_1 \setminus N'_0 \parallel N''_0, N'_1 \parallel N''_1]$
$= [Substitution]$
$a \rightarrow SExp_1; (N'_0 \parallel N''_0) \,\Box\, b \rightarrow (N'_1 \parallel N''_1)$
$= [D.28\,(Schemas/Parallelism\ Distribution)]$
$\{wrtV(SExp_1) \subseteq ns_2 \cup ns'_2\}$
$\{wrtV(SExp_1) \cap usedV(N'_0) = \emptyset\}$
$a \rightarrow (N'_0 \parallel (SExp_1; N''_0)) \,\Box\, b \rightarrow (N'_1 \parallel N''_1)$
$= [C.51\,(Prefix/Parallelism\ Distribution)]$
$\{\{a, b\} \subseteq cs\}$
$((a \rightarrow N'_0) \parallel (a \rightarrow SExp_1; N''_0)) \,\Box\, ((b \rightarrow N'_1) \parallel (b \rightarrow N''_1))$
$= [C.45\,(Parallelism/ExternalChoice\ Exchange)]$
$(a \rightarrow N'_0 \,\Box\, b \rightarrow N'_1) \parallel (a \rightarrow SExp_1; N''_0 \,\Box\, b \rightarrow N''_1)$
$= [Definition\ of\ N'_0\ and\ N''_0]$
$N'_0 \parallel N''_0$

$[2]\, G_1[N_0, N_1 \setminus N_0' \parallel N_0'', N_1' \parallel N_1'']$
$\widehat{=}\, [\mathit{Definition\ of}\ G_1]$
$(c \rightarrow \mathit{SExp}_2; N_1 \,\Box\, d \rightarrow N_0)[N_0, N_1 \setminus N_0' \parallel N_0'', N_1' \parallel N_1'']$
$= [\mathit{Substitution}]$
$c \rightarrow \mathit{SExp}_2; (N_1' \parallel N_1'') \,\Box\, d \rightarrow (N_0' \parallel N_0'')$
$= [D.28\,(\mathit{Schemas/Parallelism\ Distribution})]$
$\{wrtV(\mathit{SExp}_2) \subseteq ns_2 \cup ns_2'\}$
$\{wrtV(\mathit{SExp}_2) \cap usedV(N_1') = \emptyset\}$
$c \rightarrow (N_1' \parallel (\mathit{SExp}_2; N_1'')) \,\Box\, d \rightarrow (N_0' \parallel N_0'')$
$= [C.51\,(\mathit{Prefix/Parallelism\ Distribution})]$
$\{\{a, b\} \subseteq cs\}$
$((c \rightarrow N_1') \parallel (c \rightarrow \mathit{SExp}_2; N_1'')) \,\Box\, ((d \rightarrow N_0') \parallel (d \rightarrow N_0''))$
$= [C.45\,(\mathit{Parallelism/ExternalChoice\ Exchange})]$
$(c \rightarrow N_1' \,\Box\, d \rightarrow N_0') \parallel (c \rightarrow \mathit{SExp}_2; N_1'' \,\Box\, d \rightarrow N_0'')$
$= [\mathit{Definition\ of}\ N_1'\ \mathit{and}\ N_1'']$
$N_1' \parallel N_1''$

□

Proving the Refinement

In view of the fact that $\mathit{FireSysStart}_1$, $\mathit{FireSysStart}_2$, and *StartAreas* are defined using mutual recursion, we have to use the definition of vectorial refinement presented above. We prove the following vectorial refinement, in order to establish that the

main action of $FireControl_1$ is refined by that of $FireControl_2$.

$$
\left[\begin{array}{l}
FireSysStart_1[Act_1 \setminus Act_2], \\
FireSys_1[Act_1 \setminus Act_2], \\
Manual_1[Act_1 \setminus Act_2], \\
Auto_1[Act_1 \setminus Act_2], \\
Reset_1[Act_1 \setminus Act_2], \\
Countdown_1[Act_1 \setminus Act_2], \\
WaitingClock_1[Act_1 \setminus Act_2], \\
FireSysD_1[Act_1 \setminus Act_2], \\
Discharge_1[Act_1 \setminus Act_2], \\
Disabled_1[Act_1 \setminus Act_2]
\end{array}\right] (i)
$$

$$
\sqsubseteq_{\mathcal{V}}
$$

$$
\left[\begin{array}{l}
(FireSysStart_2 \parallel StartAreas) \setminus GasDischargeSync, \\
(FireSys_2 \parallel AreasCycle) \setminus GasDischargeSync, \\
(Manual_2 \parallel (ActiveAreas;AreasCycle)) \setminus GasDischargeSync, \\
(Auto_2 \parallel (ActiveAreas;AreasCycle)) \setminus GasDischargeSync, \\
\left(\begin{array}{l}
Reset_2 \\
\parallel \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\qquad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\qquad DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\qquad \setminus GasDischargeSync, \\
(Countdown_2 \parallel (WaitingDischarge;AreasCycle)) \\
\qquad \setminus GasDischargeSync, \\
(WaitingClock_2 \parallel (WaitingDischarge;AreasCycle)) \\
\qquad \setminus GasDischargeSync, \\
(FireSysD_2 \parallel (InitAreas;AreasCycle)) \setminus GasDischargeSync, \\
(Discharge_2 \parallel (WaitingDischarge;AreasCycle)) \setminus GasDischargeSync, \\
(Disabled_2 \parallel (DisabledAreas;AreasCycle)) \setminus GasDischargeSync
\end{array}\right] (ii)
$$

Here, two abbreviations are used: the parallelism $\parallel$ is actually the alphabetised parallelism $[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]$; and the actions lists Act_1 and Act_2 are defined as follows.

$$
\begin{array}{l}
Act_1 = FireSysStart_1, FireSys_1, Manual_1, Auto_1, Reset_1, \\
\qquad Countdown_1, WaitingClock_1, FireSysD_1, Discharge_1, Disabled_1
\end{array}
$$

$$
\begin{array}{l}
Act_2 = (FireSysStart_2 \parallel StartAreas) \setminus GasDischargeSync, \\
\quad (FireSys_2 \parallel AreasCycle) \setminus GasDischargeSync, \\
\quad (Manual_2 \parallel (ActiveAreas; AreasCycle)) \setminus GasDischargeSync, \\
\quad (Auto_2 \parallel (ActiveAreas; AreasCycle)) \setminus GasDischargeSync, \\
\quad \left(\begin{array}{l} Reset_2 \\ \parallel \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \qquad ActiveAreas; AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas; AreasCycle \end{array} \right) \end{array} \right) \\
\qquad \setminus GasDischargeSync, \\
\quad (Countdown_2 \parallel (WaitingDischarge; AreasCycle)) \\
\qquad \setminus GasDischargeSync, \\
\quad (WaitingClock_2 \parallel (WaitingDischarge; AreasCycle)) \\
\qquad \setminus GasDischargeSync, \\
\quad (FireSysD_2 \parallel (InitAreas; AreasCycle)) \setminus GasDischargeSync, \\
\quad (Discharge_2 \parallel (WaitingDischarge; AreasCycle)) \setminus GasDischargeSync, \\
\quad (Disabled_2 \parallel (DisabledAreas; AreasCycle)) \setminus GasDischargeSync
\end{array}
$$

The left-hand side of the refinement (i) can be refined using Law D.18 to introduce some assumptions that are required in the proof of the refinement of some elements of this array.

$$
\begin{array}{l}
LHS(i) \\
\sqsubseteq_{\mathcal{V}}\ [D.18] \\
\left[\begin{array}{l}
FireSysStart_1[Act_1 \setminus Act_2], \\
FireSys_1[Act_1 \setminus Act_2], \\
\{mode_A = manual \wedge mode_1 = manual\}; Manual_1[Act_1 \setminus Act_2], \\
\{mode_A = automatic \wedge mode_1 = automatic\}; Auto_1[Act_1 \setminus Act_2], \\
\{(mode_A = manual \vee mode_1 = manual) \vee mode_1 \neq manual\}; \\
\qquad Reset_1[Act_1 \setminus Act_2], \\
Countdown_1[Act_1 \setminus Act_2], \\
WaitingClock_1[Act_1 \setminus Act_2], \\
\{mode_1 = Disabled\}; FireSysD_1[Act_1 \setminus Act_2], \\
Discharge_1[Act_1 \setminus Act_2], \\
\{mode_1 = Disabled\}; Disabled_1[Act_1 \setminus Act_2]
\end{array} \right] \quad (iii)
\end{array}
$$

The proof obligations raised by this application can be easily proved. Law D.18 states that, in order to introduce an assumption $\{g\}$ before an action A within the vector of actions, we have to prove, that for each action B within the vector, if A_B is the behaviour of action B before the invocation of A, the following condition

holds.

$$\{g\};A_B \sqsubseteq_{\mathcal{A}} A_B;\{g\}$$

For instance, action $Manual_1$ is invoked by action $FireSys_1$ and recursively by itself. For this reason, in order to introduce the assumption above before the action $Manual_1$ in the vector of actions, we have to prove two conditions. The first one is related to action $FireSys_1$. In the following proof and throughout this document we use the following notation for the refinement steps.

$$A_1 \sqsubseteq_{\mathcal{A}} [law_1, \ldots, law_n]\{op_1\} \ldots \{op_n\} A_2$$

This denotes that, in order to refine the action A_1 to A_2, we have applied laws $law_1, \ldots, law_n$. These law applications have raised the proof obligations $op_1, \ldots, op_n$.

We follow with the refinement proof as follows.

$$\begin{array}{l}
\{mode_A = manual \land mode_1 = manual\};(FireSys_1 \ before \ Manual_1) \\
= [Definition \ of \ before] \\
\{mode_A = manual \land mode_1 = manual\}; \\
systemState!fireSys_s \rightarrow detection?newZone : ZoneId \rightarrow \\
\quad ActivateZone_1;switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow (mode_1 = manual) \ \& \ Skip \\
= [D.19, D.20, D.21] \\
\{newZone \notin \{mode_A, mode_1\}\} \\
\{mode_A = manual \land mode_1 = manual \land mode'_A = mode_A \land mode'_1 = mode_1 \Rightarrow \\
\quad mode'_A = manual \land mode'_1 = manual\} \\
systemState!fireSys_s \rightarrow detection?newZone : ZoneId \rightarrow \\
\quad ActivateZone_1 \rightarrow switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad \{mode_A = manual \land mode_1 = manual\};(mode_1 = manual) \ \& \ Skip \\
\sqsubseteq_{\mathcal{A}} [D.16, C.36, C.57] \\
\{mode_A = manual \land mode_1 = manual \land mode_1 = manual \Rightarrow \\
\quad mode_A = manual \land mode_1 = manual\} \\
systemState!fireSys_s \rightarrow detection?newZone : ZoneId \rightarrow \\
\quad ActivateZone_1 \rightarrow switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad (mode_1 = manual) \ \& \ Skip;\{mode_A = manual \land mode_1 = manual\} \\
= [Definition \ of \ before] \\
(FireSys_1 \ before \ Manual_1);\{mode_A = manual \land mode_1 = manual\}
\end{array}$$

In the same way, as action $Manual_1$ does not change the values of the variables

$mode_1$ and $mode_A$, we can easily prove the second condition described below.

$$\{mode_A = manual \wedge mode_1 = manual\};(Manual_1 \ before \ Manual_1)$$
$$\sqsubseteq_{\mathcal{A}}$$
$$(Manual_1 \ before \ Manual_1);\{mode_A = manual \wedge mode_1 = manual\}$$

This concludes the proof of the conditions that must hold in order to introduce the assumption before the action $Manual_1$ in the vector of actions. The remaining assumption introductions can be proved in a very similar way.

Finally, as discussed above, the refinement from vector (iii) to vector (ii) is valid if we can prove that the refinement is valid for each corresponding element of each vector. In this section, we prove some of these refinements. The remaining proofs can be found in Appendix B.

First, we have the lemma for the refinement of the action $FireSysStart_1$.

Lemma 5.7

$$FireSysStart_1[Act_1 \setminus Act_2]$$
$$\sqsubseteq_{\mathcal{A}}$$
$$\left(\begin{array}{l} FireSysStart_2 \\ \|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\ StartAreas \end{array} \right) \setminus GasDischargeSync$$

Proof. We start this refinement by applying the definition of action $FireSysStart_1$ and by applying the substitution $[Act_1 \setminus Act_2]$.

$$\begin{array}{l} FireSysStart_1[Act_1 \setminus Act_2] \\ = [Definition \ of \ FireSysStart_1, Substitution] \\ systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\ \quad switchLamp[LampId].systemOnLamp!on \rightarrow \\ \quad\quad InitFireControl_1; \\ \quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ \|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\ AreasCycle \end{array} \right) \\ \quad\quad\quad \setminus GasDischargeSync \end{array}$$

The schema operation $InitFireControl_1$ can be written as the sequential composition of two other schema operation as follows.

$$
\begin{array}{l}
= [D.7] \\
\{\alpha(InternalSystemState) \cap \alpha(AreasState) = \emptyset\} \\
\{FV(true) \subseteq \alpha(InternalSystemState)\} \\
\{FV(true) \subseteq \alpha(AreasState)\} \\
\{\{mode'_1, dischargedOcurred'_1\} \subseteq \alpha(InternalSystemState')\} \\
\{\{mode'_A, controlledZones'_1, activeZones'_1, discharge'_1, active'_1\} \subseteq \\
\quad \alpha(AreasState')\} \\
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad InitInternalSystem; InitAreas; \\
\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync
\end{array}
$$

Each one of the new inserted schema operations writes in different partitions of the parallelism that follows them. For this reason, we may distribute them over the parallelism.

$$
\begin{array}{l}
= [D.28, D.29] \\
\{wrtV(InitAreas) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(InitAreas) \cap usedV(FireSys_2) = \emptyset\} \\
\{wrtV(InitInternalSystem) \subseteq \\
\quad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(InitInternalSystem) \cap usedV(AreasCycle) = \emptyset\} \\
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad \left(\begin{array}{l} (InitInternalSystem; FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas; AreasCycle) \end{array} \right) \\
\quad\quad \setminus GasDischargeSync
\end{array}
$$

Next, we move the *switchLamp* event to the internal system side of the parallelism.

$$
\begin{array}{l}
= [C.44] \\
\{initials(AreasCycle) \subseteq \Sigma_2\} \\
\{switchLamp \notin \Sigma_2\} \\
\{wrtV(switchLamp[LampId].systemOnLamp!on \rightarrow Skip) \cap \\
\quad usedV(InitAreas;AreasCycle) = \emptyset\} \\
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad \left(\begin{array}{l} \left(\begin{array}{c} switchLamp[LampId].systemOnLamp!on \rightarrow \\ InitInternalSystem;FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad \setminus GasDischargeSync
\end{array}
$$

Now, we distribute the prefixing *switchOn* over the parallelism. Furthermore, we expand the hiding to the whole action body.

$$
\begin{array}{l}
= [C.54, C.51] \\
\{\{switchOn, systemState\} \cap GasDischargeSync = \emptyset\} \\
\{switchOn \in \Sigma_2\} \\
\left(\begin{array}{l} systemState!fireSysStart_s \rightarrow \\ \quad \left(\begin{array}{l} \left(\begin{array}{c} switchOn \rightarrow switchLamp[LampId].systemOnLamp!on \rightarrow \\ InitInternalSystem;FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (switchOn \rightarrow InitAreas;AreasCycle) \end{array} \right) \end{array} \right) \\
\setminus GasDischargeSync
\end{array}
$$

As we did with the *switchLamp*, we move the *systemState* event to the internal system side of the parallelism.

$$
\begin{array}{l}
= [D.4, C.44] \\
\{switchOn \in \Sigma_2\} \\
\{systemState \notin \Sigma_2\} \\
\{wrtV(systemState!fireSysStart_s \rightarrow Skip) \cap \\
\quad usedV(switchOn \rightarrow InitAreas;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l} \left(\begin{array}{l} systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\ \quad \left(\begin{array}{c} switchLamp[LampId].systemOnLamp!on \rightarrow \\ InitInternalSystem;FireSys_2 \end{array} \right) \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (switchOn \rightarrow InitAreas;AreasCycle) \end{array} \right) \\
\setminus GasDischargeSync
\end{array}
$$

Finally, by applying the definitions of the actions $FireSysStart_2$ and $StartAreas$, we conclude the proof of this lemma.

$$
\begin{array}{l}
= [\textit{Definition of } FireSysStart_2 \textit{ and } StartAreas] \\
\left(\begin{array}{l} FireSysStart_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ StartAreas \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

□

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

Lemma 5.8

$$
\begin{array}{l}
FireSys_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof. As for the previous lemma, we start the proof of this refinement by applying the definition of $FireSys_1$ and the substitution.

$$
\begin{array}{l}
FireSys_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ FireSys_1, Substitution] \\
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad SwitchFireControlMode_1; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad \setminus GasDischargeSync \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \& \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\quad\quad\quad\quad \setminus GasDischargeSync \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \& \\
\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\quad\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow switchBuzzer!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow InitFireControl_1; SwitchLampsOff_1; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad \setminus GasDischargeSync
\end{array}
$$

Next, we expand the hiding to the whole action.

$$
\begin{array}{l}
= [C.54] \\
\{GasDischargeSync \cap \\
\quad \{systemState, modeSwitch, detection, switchLamp, \\
\quad\ alarm, fault, switchBuzzer, reset\} = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad SwitchFireControlMode_1; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \& \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \& \\
\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow InitFireControl_1; SwitchLampsOff_1; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Now, we use the schema calculus to replace all the occurrences of the schema expression $ActivateZone_1$ by the schema expression $ActivateZoneAS$. The two versions of the schema/sequence introduction laws (C.28 and D.7) are used to replace the schema operations $SwitchFireControlMode_1$ and $InitFireControl_1$ by a sequential composition of two schema operations. Finally, by definition, we may replace

the action $SwitchLampsOff_1$ by $SwitchLampsOff_2$.

$$
\begin{array}{l}
= [Schema\ Calculus, Definition\ of\ SwitchLampsOff_2, C.28, D.7] \\
\{\alpha(InternalSystemState) \cap \alpha(AreasState) = \emptyset\} \\
\{\{mode'_1, dischargedOcurred'_1\} \subseteq \alpha(InternalSystemState')\} \\
\{\{controlledZones_1, activeZones_1, discharged_1, active_1\} \cap \\
\ \{mode'_1, dischargedOcurred'_1\} = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad SwitchInternalSystemMode;\ SwitchAreasMode; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \& \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \& \\
\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad InitInternalSystem; InitAreas; SwitchLampsOff_2; \\
\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

We use an auxiliary lemma in order to transform the second choice branch into a parallelism. Also, we want the guards $mode_1 = manual$ and $mode_1 = automatic$ to

be placed only in the internal system side of this parallelism.

$$
\begin{array}{l}
= [B.1] \\
\{initials(ActiveAreas) \subseteq \Sigma_2\} \\
\{\{detection, switchLamp, alarm\} \cap GasDischargeSync = \emptyset\} \\
\{\{switchLamp, alarm\} \cap \Sigma_2 = \emptyset\} \\
\{detection \in \Sigma_2\} \\
\{wrtV(ActivateZoneAS) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(ActivateZoneAS) \cap (usedV(Auto_2) \cup usedV(Manual_2)) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad SwitchInternalSystemMode;\ SwitchAreasMode; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow \\ \quad\quad alarm!firstStage \rightarrow \\ \quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\ \quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad InitInternalSystem;InitAreas;SwitchLampsOff_2; \\
\quad\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

We refine the third and forth alternative branches in order to move the non-synchronising events to one of the parallel actions; in our case, the internal system

action.

$$
\begin{array}{l}
= [C.44] \\
\{initials(AreasCycle) \subseteq \Sigma_2\} \\
\{\{switchLamp, fault, switchBuzzer\} \cap \Sigma_2 = \emptyset\} \\
\{\{faultId\} \cap usedV(AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad SwitchInternalSystemMode;\ SwitchAreasMode; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow \\ \quad\quad alarm!firstStage \rightarrow \\ \quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\ \quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad InitInternalSystem;InitAreas; \\
\quad\quad\quad \left(\begin{array}{l} (SwitchLampsOff_2;FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

The next step consists in distributing the schema operations over the respective parallelism.

$$
\begin{array}{l}
= [D.29, D.28] \\
\{wrtV(SwitchAreasMode) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(SwitchAreasMode) \cap usedV(FireSys_2) = \emptyset\} \\
\{wrtV(SwitchInternalSystemMode) \subseteq \\
\qquad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(SwitchInternalSystemMode) \cap usedV(AreasCycle) = \emptyset\} \\
\{wrtV(InitAreas) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(InitAreas) \cap usedV(FireSys_2) = \emptyset\} \\
\{wrtV(InitInternalSystem) \subseteq \\
\qquad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(InitInternalSystem) \cap usedV(AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\qquad \left(\begin{array}{l} (SwitchInternalSystemMode;\ FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (SwitchAreasMode; AreasCycle) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow \\ \qquad alarm!firstStage \rightarrow \\ \qquad\quad (mode_1 = manual)\ \&\ Manual_2 \\ \qquad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS; ActiveAreas; AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \qquad switchBuzzer!on \rightarrow FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\qquad \left(\begin{array}{l} (InitInternalSystem; SwitchLampsOff_2; FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas; AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Now, we move the *alarm* event, in the forth choice branch, to the internal system action.

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{initials(InitAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{alarm \notin \Sigma_2\} \\
\{wrtV(alarm!alarmOff \rightarrow Skip) \cap usedV(InitAreas;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\qquad \left(\begin{array}{l}
(SwitchInternalSystemMode;\ FireSys_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(SwitchAreasMode;AreasCycle)
\end{array} \right) \\
\qquad \square \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \square\ (mode_1 = automatic)\ \&\ Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\qquad \square \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right) \\
\quad \square\ reset \rightarrow \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

The first and the forth branches of the alternative can be refined by a distribution of the prefixing over the parallelism as shown below. The associativity of the external

choice is also used to move the *reset* branch of the alternative.

$$
\begin{array}{l}
= [D.6, C.51, D.22] \\
\{modeSwitch \in \Sigma_2\} \\
\{reset \in \Sigma_2\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} modeSwitch?newMode : SwitchMode \rightarrow \\ \quad SwitchInternalSystemMode;\ FireSys_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} modeSwitch?newMode : SwitchMode \rightarrow \\ \quad SwitchAreasMode;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow \\ \quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow \\ \quad\quad alarm!firstStage \rightarrow \\ \quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\ \quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow FireSys_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

In the next refinement step, we use the parallelism/external choice exchange law (C.45) in order to transform a external choice of parallelism in a parallelism of external

choices.

$$
\begin{array}{l}
= [D.22, C.45] \\
\{\{modeSwitch, detection, reset\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array}\right) \\
|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]| \\
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchAreasMode;AreasCycle \\
\Box\ reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \\
|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]| \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

We may introduce a *Stop* branch to any external choice, since the *Stop* is the external choice unit.

$$
\begin{array}{l}
= [C.58] \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchAreasMode;AreasCycle \\
\Box\ reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\quad \Box\ Stop \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

However, using the second parallelism zero law, we may transform the *Stop* to a parallelism as follows.

$$
\begin{array}{l}
= [D.27] \\
\{\{modeSwitch, reset, detection, automaticDischarge, manualDischarge\} \subseteq \Sigma_2\} \\
\{\{modeSwitch, reset, detection\} \cap \{automaticDischarge, manualDischarge\} = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array} \right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchAreasMode;AreasCycle \\
\Box\ reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array} \right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
\Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\quad gasNotDischarged.area \rightarrow AreasCycle \\
\Box\ \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad gasNotDischarged.area \rightarrow AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array} \right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Since we have a deterministic system, and the first and the second branches of the alternative have the left-hand side of the parallelism in common, the distribution of parallelism over external choice law can be applied, as we present below.

$$
\begin{array}{l}
= [C.46] \\
\{\{modeSwitch, reset, detection\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem; SwitchLampsOff_2; FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchAreasMode; AreasCycle \\
\Box\ reset \rightarrow InitAreas; AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS; ActiveAreas; AreasCycle \\
\Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\quad gasNotDischarged.area \rightarrow AreasCycle \\
\Box\ \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad gasNotDischarged.area \rightarrow AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Still refining the first branch of the external choice, we may notice that all the choices of the right-hand side of the parallelism finishes with *AreasCycle*. In this

case, we may use the distribution of sequence over external choice as follows.

$$
\begin{array}{l}
= [C.24, D.24, D.22] \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchAreasMode \\
\Box\ reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\quad gasNotDischarged.area \rightarrow Skip \\
\Box\ \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right) \\
AreasCycle
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Now, we have actually that the right-hand side of the parallelism in the first branch of the external choice, given the associativity of the external choice, is the action

$AreasCycle$.

$$
\begin{array}{l}
= [D.22, \mathit{Definition\ of\ AreasCycle}] \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

One more time, as we have the same action in the right-hand side of the parallelism, we may apply the distribution of the parallelism over external choice law (C.46).

$$
\begin{array}{l}
= [C.46] \\
\{initials(AreasCycle) \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
modeSwitch?newMode : SwitchMode \rightarrow \\
\quad SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad InitInternalSystem;SwitchLampsOff_2;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Finally, we may move the communication of the system state to the internal system side of the parallelism as presented below.

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{initials(AreasCycle) \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{systemState\} = \emptyset\} \\
\{wrtV(systemState!fireSys_s \rightarrow Skip) \cap usedV(AreasCycle) = \emptyset\} \\
\left(\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad SwitchInternalSystemMode;\ FireSys_2 \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad InitInternalSystem; SwitchLampsOff_2; FireSys_2 \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array} \right) \right. \\
\left. \begin{array}{l}
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

This finishes our proof since, by the associativity of external choice, we have that the right-hand side of the parallelism is the definition of the action $FireSys_2$.

$$
\begin{array}{l}
= [D.22,\, Definition\ of\ FireSys_2] \\
\left(\begin{array}{l}
FireSys_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right) \setminus GasDischargeSync
\end{array}
$$

□

For conciseness, the proof of the remaining lemmas are not presented here. They can be found in Appendix B.

In the following section we make a process refinement in order to upgrade each partition into a separate process (*InternalSystem* and *Areas*).

5.6.4 Process Refinement: upgrading the partitions into separated processes (*InternalSystem* and *Areas*)

In the previous section, we partitioned the state of the process $FireControl_1$ into *InternalSystemState* and *AreasState*. Each partition has its own set of process paragraphs. These sets are disjoint, since, no command nor action expression in one set refers to state components in the other partition state or paragraph names. Furthermore, we define the main action of the refined process in terms of these two partitions. Therefore, we may apply Law C.11 in order to split process $FireControl_2$ into two independent processes: *InternalSystem* and *Areas*, which are described in Appendix G.

The process $FireControl_2$ becomes the parallel composition of the *InternalSystem* and the *Areas* as defined below.

$$\textbf{process}\ FireControl_2 \mathrel{\widehat=} \left(\begin{array}{l} InternalSystem \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ Areas \end{array} \right) \setminus GasDischargeSync$$

In the next section, we refine the process *Areas* in order to describe the areas as an interleaving of individual processes that model each of the areas.

5.6.5 Data Refinement: the *Areas* process as a promotion of individual areas

5.6.6 Process Refinement: split the *Areas* into separated *Area* processes

5.6.7 Action Refinement: decomposing the *InternalSystem* in two partitions

5.6.8 Process Refinement: split the *InternalSystem* into a *FireControl* and a *DisplayController*

5.7 Conclusions

Chapter 6

Implementation Using JCSP

In this chapter we present a strategy for implementing *Circus* programs in JCSP [25, 24]. The strategy is based on a number of translation laws, which, if applied exhaustively, transforms a *Circus* program into a Java program that uses the JCSP library. We assume that, before applying the translation strategy presented in this chapter, the specification of the system we want to implement has been already refined, using the *Circus* refinement strategy (Chapter 3), to meet the translation strategy's requirements discussed in Section 6.2.

First, Section 6.1 presents JCSP and some examples. Section 6.2 presents the strategy to implement *Circus* programs using JCSP. In Section 6.3 we extend the types of communication considered in our strategy, and in Section 6.4 we present the translation strategy for the *Circus* indexed operator. In Section 6.5 we present an example. Finally, in Section 6.6 we conclude with some considerations about the strategy.

6.1 JCSP

JCSP [25, 24] is a Java library that provides tools for implementing communicating processes based on the CSP model of communicating systems.

In JCSP, a CSP process is an instance of a class that implements the interface `CSProcess` defined below.

```
public interface CSProcess { public void run(); }
```

The method `run` defines the process behaviour. By way of illustration, let us consider the following process that outputs (in the standard output) the Fibonacci sequence.

```
public class Fibonacci implements CSProcess {
    private int x = 1, y  = 1;
    public void run(){
        System.out.println(x);
        System.out.println(y);
        while(true) {
            int next = x+y;
            System.out.println(next);
            x = y;
            y = next;
        }
    }
}
```

This process has two integer components: `x` and `y`. The later stores the last value output by the process, and the former stores the value output before that. First, the process `Fibonacci` outputs the number 1 twice. These values are stored in the variables `x` and `y`. Then, it iterates: in each iteration, the `next` output corresponds to the sum of the last two outputs `x` and `y`, which are updated after the output. However, in this example, no interaction happens with any other process.

All the interaction with a process is made via CSP synchronising channels. In JCSP, the simplest form of such channels is the point-to-point channel, which is implemented by class `One2OneChannel`; multiple readers and writers are not allowed. On the other hand, `Any2AnyChannel` allows multiple reader and writers. However, for any type of channel in JCSP, when a communication happens, it happens between one writer and one reader (point-to-point).

Channels in JCSP communicate instances of a `java.lang.Object`. However, for each channel type in JCSP, there is a corresponding channel type that communicates values of the basic type `int`, which are used in our example.

The `Fibonacci` process described above can be rewritten as follows.

```
public class Fibonacci implements CSProcess {
    private int first = 1, second  = 1;
    private One2OneChannelInt out;
    public Fibonacci(One2OneChannelInt out){ this.out = out; }
    public void run(){
        out.write(first); out.write(second);
        while(true) {
            int next = first+second;
            out.write(next);
            first = second;
            second = next;
        }
    }
}
```

Instead of printing the fibonacci sequence, the process `Fibonacci` communicates the sequence through the channel `out`, which is received as an argument in the constructor of the class. In order to write an object to a channel, a process must invoke the public method `write`; the public method `read` is used to read an object from a channel. The invocation of `write` is blocked until the communication happens (some other process reads the written value). In a similar way, `read` is blocked until a value is written to the channel. In our example, as we are using `One2OneChannelInt` channels, we may write `int`s to the channel.

The choice is a very important operator in CSP. In JCSP, this operator is provided by an object of class `Alternative`, which waits the possibility to synchronise in some of its events, and then, chooses one of those. The choice can be arbitrary, which is a random choice; user-prioritised, in which the user set priority for each of the channels involved in the choice; and fair, in which available channels are fairly chosen.

Our example can be extended in order to give the environment the choice between a new value of the sequence or a new start of the sequence, after the first two elements of the sequence have been output. This choice is represented by two new input channels, which are also declared as private attributes of the process, and are taken as arguments in the constructor.

```
public class Fibonacci implements CSProcess {
    private int first = 1, second  = 1;
    private One2OneChannelInt out;
    private One2OneChannel nextValue, restart;
    public Fibonacci(One2OneChannelInt out,
                     One2OneChannel nextValue,
                     One2OneChannel restart){
```

```
            this.out = out;
            this.nextValue = nextValue;
            this.restart = restart;
        }
    }
```

The method `run` is changed in order to introduce the choice, after the output of the first two elements of the sequence. An array guard of all channels that are involved in the choice, and integer constants corresponding to the indexes of these channels in the array are declared.

```
        public void run(){
            nextValue.read(); out.write(first);
            nextValue.read(); out.write(second);
            final Guard[] guard = { nextValue, restart };
            final int NEXT_VALUE = 0;
            final int RESTART    = 1;
```

This array of channels is given as argument to the constructor of the `Alternative`. In our example, we make a fair choice using the method `fairSelect`, which returns the index of the chosen channel in the array of channel of the `Alternative`. A `switch` block is used to verify the value returned by the invocation of method `fairSelect`: if the channel `nextValue` is chosen, the sequence is restarted; otherwise, the process outputs the next value.

```
            final Alternative alt = new Alternative (guard);
            while(true){
                switch (alt.fairSelect()) {
                    case NEXT_VALUE:
                        int next = first+second;
                        nextValue.read(); out.write(next);
                        first = second; second = next;
                        break;
                    case RESTART:
                        restart.read();
                        first = second = 1;
                        nextValue.read(); out.write(first);
                        nextValue.read(); out.write(second);
                        break;
                }
            }
        }
    }
```

CSP parallel processes are written using the class `Parallel`. Its constructor takes an array of `CSProcesses` and returns a `CSProcess` that is the parallel composition of its process arguments. A `run` of a `Parallel` process terminates when, and only when, all its component processes terminate.

By way of illustration, suppose we have a class `Reader` which defines a reader that communicates with the process `Fibonacci`. All the channels used in the communication, as in the class `Fibonacci`, are declared attributes of the class and are given as arguments to its constructor.

```
public class Reader implements CSProcess {
    private One2OneChannelInt out;
    private One2OneChannel nextValue, restart;
    public Reader(One2OneChannelInt out,
                  One2OneChannelInt nextValue,
                  One2OneChannelInt restart){
        this.out = out;
        this.nextValue = nextValue;
        this.restart = restart;
    }
```

However, the method `run` is quite different. We suppose our `Reader` restarts the fibonacci sequence each time it reads ten values of the sequence.

```
    public void run(){
        int count = 1;
        while(true){
            if(count<=10){
                nextValue.write(null);
                System.out.println(out.read()); count++;
            } else {
                restart.write(null); count = 1;
            }
        }
    }
}
```

No values are communicated through the channels `nextValue` and `restart`; they are only synchronising events. For this reason, we write the `null` value to these channels.

Finally, we can declare the class `FibonacciReader` below, which brings both the `Fibonacci` and the `Reader` processes in parallel. It instantiates the channels that are used by both processes, and uses these channels to create an instance of each process.

```
public class FibonacciReader implements CSProcess {
    private One2OneChannelInt out = new One2OneChannelInt();
    private One2OneChannel nextValue = new One2OneChannel();
    private One2OneChannel restart = new One2OneChannel();
    private Fibonacci fibonacci =
        new Fibonacci(out,nextValue,restart);
    private Reader reader = new Reader(out,nextValue,restart);
```

Its execution consists of an instantiation of a `Parallel` of both process, and its execution.

```
    public void run() {
        CSProcess[] processes =
            new CSProcess[]{fibonacci,reader};
        Parallel parallel = new Parallel(processes);
        parallel.run();
    }
}
```

This executes both the `Fibonacci` and the `Reader` in parallel.

The CSP constructors *Skip* and *Stop* can be written using the corresponding classes `Skip` and `Stop`. Finally, JCSP provides some additional features as `Barrier`, which enables barrier synchronisation between a set of processes, and `Delta`, which has an infinite loop that waits for objects to be sent to it, and then sends the reference to the object in parallel to an array of processes. These facilities are not used in our work; details can de found in [25, 24].

6.2 From *Circus* to JCSP

Our strategy for implementing *Circus* programs considers each single paragraph individually, and in sequence. *Circus* paragraphs include Z paragraphs, channel declarations, and process declarations.

We assume that, in the *Circus* program to be implemented, these three types of paragraphs grouped in this order. Our strategy considers that neither free types nor abbreviations are defined within processes definitions. Our strategy also assumes that the Z paragraphs in a *Circus* program are axiomatic definitions of the form $v : T \mid v = e_1$, free types, or abbreviations. The types used in the *Circus* program should all be already implemented in Java. Moreover, we also consider that, at this stage, all operation schemas and specification constructs have already been refined.

Z paragraphs used to group channel declarations, as well as channel sets must be expanded before the translation strategy is applied. We consider only communications of the following types: non-typed inputs, outputs, or synchronisation events.

This means that channels can be declared to be only of a single type or to have no type. In summary, the acceptable communications are all of the following form.

$$\mathsf{Comm} ::= \mathsf{N?N} \mid \mathsf{N!Expression} \mid \mathsf{N}$$

Communications of the form $c?N : T$ and $c?N : Predicate$ are not directly implementable in JCSP. Their semantics in *Circus* is that the communication does not happen if the transmitted value is not in the set T or does not satisfy the *Predicate*. In JCSP, a communication happens when some process writes in a channel, and some other process reads from this channel. JCSP does not allow to backtrack this communication if the read value does not satisfy a constraint. Our strategy, however, considers that such kind of communication has been refined to some protocol that removes them. Furthermore, since in JCSP, we may write/read an object to/from a channel, communications may have only one input or one output value: multiple inputs/outputs must be encapsulated in Java objects.

Multi-synchronisation channels are also not implementable in JCSP. Our strategy considers that multi-synchronisation has been refined to some protocol, as the one presented in [32]. Besides, guarded output channel must have been removed using the strategy presented in **??**[1].

The output of our translation strategy is Java code that contains some classes definitions. These definitions can be split into different Java files and allocated in their respective packages. For a given project name `proj`, the translation strategy generates six packages: the package `proj` contains the main system class; the package `proj.axiomaticDefinitions` contains the class that encapsulates all the axiomatic definitions within the translated *Circus* program; all the processes in the *Circus* program are declared in the package `proj.processes`; the package `proj.typing` contains all the typing classes of the system; and the package `proj.util` contains all the utilities classes used in the system.

Parallelism of actions and processes must take into account the set of synchronisation channels and the partitions of the variables in scope. JCSP, however, does not allow the user to determine the synchronisation channel set. For this reason, when using JCSP, the intersection of the alphabets determines the synchronisation channels set. If it is not empty, we have a parallelism in the intersection channels; otherwise, we have actually an interleaving. Therefore, all parallelism in the *Circus* program have to be in the form $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$ for actions, or $P_1 \llbracket cs \rrbracket P_2$ for processes, where cs is the intersection of the sets of channels used by A_1 and A_2, or P_1 and P_2, respectively.

The translation strategy uses a channel environment $ChanTypeEnv : ChanEnv$, where *ChanEnv* is an extension of the channel environment presented in Section 2.2. It associates a channel name to a pair of sequences: the first is a sequence of possible

[1]This work was done by Jim Woodcock, but has not yet been published anywhere

generic expressions used to define a family of channels; and the second is a sequence of types of the channel.

$$ChanEnv == ChanName \pfun (\seq \mathsf{Expression} \times \seq \mathsf{Expression})$$

We consider that this environment is available during the translation. Its definitions is as in Section 2.2 with a small extension for considering generic channels: the definition of function $[\![\_]\!]^{\mathcal{C}}$ is extended as follows to consider the generic channels declaration as follows.

$$\begin{aligned}
&[\![_]\!]^{\mathcal{C}} : \mathsf{CDeclaration} \fun ChanEnv \\
&[\![n]\!]^{\mathcal{C}} = \{n \mapsto ([\,], [Sync])\} \\
&[\![n : T_1 \times \ldots \times T_m]\!]^{\mathcal{C}} = \{n \mapsto ([\,], [T_1, \ldots, T_m])\} \\
&[\![[G_1, \ldots, G_n] n : T_1 \times \ldots \times T_m]\!]^{\mathcal{C}} = \{n \mapsto ([G_1, \ldots, G_n], [T_1, \ldots, T_m])\}
\end{aligned}$$

For generic channels, we have that the used generic types are also stored in the channel environment.

Besides the environment *ChanTypeEnv*, we consider that, for each process, two additional environments are available throughout its translation. These environments store information about how each channel is used within each process: the environment $VisChanEnv : ChanUseEnv$ stores the information of the visible channels of the process, and the environment $HidChanEnv : ChanUseEnv$ stores the information of the hidden channels of the process. The type *ChanUseEnv* maps channel names to its use.

$$ChanUseEnv == \mathsf{N} \pfun ChanUse$$

The type *ChanUse* indicates how the channel is used within the process: I for input channels, O for output channels, or A for input channels that take part in external choices.

$$ChanUse ::= I \mid O \mid A$$

An type environment is also considered available in the translation: the environment $TypesEnv : \seq \mathsf{Expression}$ lists all the types that are used in the *Circus* program which is being translated. This list includes all the basic types, free types, abbreviations, and possible types created for encapsulating multiple inputs and outputs.

Some auxiliary functions are considered throughout the translation: the function *JType* defines the Java type corresponding to each of the *Circus* types used in the program; the function *JExp* translates a given expression into Java code; and, finally, the function *CType* returns the *Circus* type of the given variable.

Our strategy considers that all values transported through channels are Java objects. Despite the existence of `int` channels in JCSP, we consider only Object channels. Java primitive integer values are transmitted through the channels using the class `java.lang.Integer`.

In Section 6.2.1 we present the translation for processes declarations and in Section 6.2.2 we present the translation for basic processes. Sections 6.2.3 and 6.2.4 present the translation of processes paragraphs and CSP actions, respectively. In Section 6.2.5 we present how to translate processes that are defined in terms of other processes. The translation of existent Z paragraphs is presented in Section 6.2.6, and the declaration of utilities classes is presented in Section 6.2.7. Finally, Sections 6.2.8 and 6.2.9 present the translation and the execution of *Circus* programs.

6.2.1 Processes Declarations

For each process declaration, we create a new Java class that implements this process. All Java classes representing processes implement the JCSP interface `jcsp.lang.CSProcess`.

First, if the translations reaches the end of the processes declaration it returns the empty Java code.

$$[\![\_]\!]^{ProcDecls} : \mathsf{Program} \pfun \mathsf{N} \pfun \mathsf{JCode}$$
$$[\![\epsilon]\!]^{ProcDecls}\ proj = \epsilon$$

The type JCode represents a Java code. Throughout this chapter, we use the symbol ϵ to represent empty entities: *Circus* programs, Java code, or others.

Otherwise, for a given process P and project name $proj$, we declare a new Java class `P` in the package $proj$`.processes`. This class imports the java utilities package, the JCSP package that contains its most important classes, and all the packages within the project. The body of the class is determined by the translation of the paragraphs of P.

$$[\![\textbf{process}\ P \mathrel{\widehat=} ParProc\ ProcDecls]\!]^{ProcDecls}\ proj =$$

```
    package proj.processes;
    import java.util.*;
    import jcsp.lang.*;
    import proj.axiomaticDefinitions.*;
    import proj.typing.*;
    import proj.util.*;
    public class P implements CSProcess {
        [[ParProc]]^ParProc
    }
    [[ ProcDecls ]]^ProcDecls proj
```

If the process is parameterised, this is reflected in the attributes and in the constructor of the corresponding process Java class. Furthermore, hidden channels are declared as attributes of the class, and initialised in the class constructor, as the parameters and the visible channels of the process.

The translation the body of parametrised processes is captured by the following function.

$$
\begin{array}{l}
[\![\_]\!]^{ParProc} : \mathsf{ParProc} \pfun \mathsf{JCode} \\
[\![Decl \bullet Proc]\!]^{ParProc} = \\
\quad ParamsDecl\ Decl \\
\quad VisibleCDecl\ VisChanEnv \\
\quad HiddenCDecl\ HidChanEnv \\
\quad \texttt{public}\ P\texttt{(}ParamsArgs\ Decl\texttt{,}VisibleCArgs\ VisChanEnv\texttt{)\{} \\
\quad\quad MultiAssign\ (ParamsDecl\ Decl)\ (ParamsArgs\ Decl) \\
\quad\quad MultiAssign\ (VisibleCDecl\ VisChanEnv) \\
\quad\quad\quad\quad (VisibleCArgs\ VisChanEnv) \\
\quad\quad HiddenCCreation\ HidChanEnv \\
\quad \texttt{\}} \\
\\
\quad \texttt{public void run()\{}\ \ [\![\, Proc \,]\!]^{Proc}\ \ \texttt{\}}
\end{array}
$$

The parameters, visible and hidden channels are declared as class attributes. They are initialised within the class constructor: the parameters and the visible channels are initialised with the value given as argument to the constructor, and the hidden channels are instantiated. Finally, the body of the method `run`, which represents the process running, is the translation of the process body *Proc*. The following sections describe these steps of the translation.

Class Attributes

Each of the parameters of the process is declared as an attribute of the Java class that represents the process. The function *ParamsDecl* transforms a declaration of parameters of a *Circus* process into a `;`-separated list of Java attribute declarations.

$$
\begin{array}{l}
ParamsDecl : \mathsf{Decl} \pfun \mathsf{JCode} \\
ParamsDecl\ x_1 : T_1; \ldots; x_n : T_n = \\
\quad \texttt{private}\ (JTypeT_1)\ \texttt{x\_1;} \ldots \texttt{;}\ \texttt{private}\ (JTypeT_n)\ \texttt{x\_n;}
\end{array}
$$

The visible channels within a *Circus* process need also to be declared. The function *VisibleCDecl* receives these channels along with their respective usage, and declares a `;`-separated list of Java attribute declarations.

$$
\begin{array}{l}
VisibleCDecl : ChanUseEnv \pfun \mathsf{JCode} \\
VisibleCDecl\ \emptyset = \epsilon \\
VisibleCDecl\ \{c \mapsto t\} \cup \upsilon = \texttt{private}\ (TypeChan\ t)\ \texttt{c;}\ \ VisibleCDecl\ \upsilon
\end{array}
$$

It uses an auxiliary function *TypeChan* that returns a type of channel given a type of use.

$$
\begin{array}{l}
\mathit{TypeChan} : \{I, O, A\} \pfun \mathsf{JCode} \\
\mathit{TypeChan}(I) = \texttt{ChannelInput} \\
\mathit{TypeChan}(O) = \texttt{ChannelOutput} \\
\mathit{TypeChan}(A) = \texttt{AltingChannelInput}
\end{array}
$$

The hidden channels used within a *Circus* process need also to be declared as class attributes. The function *HiddenCDecl* receives these channels and their respective usage, and declares a `;`-separated list of Java attribute declarations.

$$
\begin{array}{l}
\mathit{HiddenCDecl} : \mathit{ChanUseEnv} \pfun \mathsf{JCode} \\
\mathit{HiddenCDecl}\ \emptyset = \epsilon \\
\mathit{HiddenCDecl}\ \{c \mapsto t\} \cup \upsilon = \texttt{private Any2OneChannel c;}\ \mathit{HiddenCDecl}\ \upsilon
\end{array}
$$

The hidden channels are instantiated within this process. For this reason, we declare then as `Any2OneChannel`, which can be instantiated. In contrast, the visible channels are declared using function *TypeChan* since they are not instantiated within this class.

Class Constructor

Basically, the constructor has to initialise all the visible channels and the parameters of the process with the respective constructor arguments, and to instantiate each of the hidden channels used within the paragraphs and the main action of the process.

The function *ParamsArgs* is similar to *ParamsDecl*, but prefixes each parameter name of the process with `new` and returns a `,`-separated list of method parameters declarations.

$$
\begin{array}{l}
\mathit{ParamsArgs} : \mathsf{Decl} \pfun \mathsf{JCode} \\
\mathit{ParamsArgs}\ x_1 : T_1;\ \ldots;\ x_n : T_n = \\
\quad (\mathit{JType}T_1)\ \texttt{newx\_1,}\ \ldots,\ (\mathit{JType}T_n)\ \texttt{newx\_n}
\end{array}
$$

The next arguments of the constructor are the channels used within the process which are not hidden. The function *VisibleCArgs* is very similar to the function *VisibleCDecl*, but prefixes each channel name with `new` and returns a `,`-separated list of Java method arguments declarations. Its definition is as follows.

$$
\begin{array}{l}
\mathit{VisibleCArgs} : \mathit{ChanUseEnv} \pfun \mathsf{JCode} \\
\mathit{VisibleCArgs}\ \emptyset = \epsilon \\
\mathit{VisibleCArgs}\ \{c \mapsto t\} \cup \upsilon = \\
\quad \textbf{if}\ (\upsilon \neq \emptyset)\ \textbf{then}\ (\mathit{TypeChan}\ t)\ \texttt{newc,}\ (\mathit{VisibleCArgs}\ \upsilon) \\
\quad \textbf{else}\ (\mathit{TypeChan}\ t)\ \texttt{newc}
\end{array}
$$

The initialisation is done using the function *MultiAssign*. This function receives two arguments: a `;`-separated list of Java attribute declarations, and a `,`-separated list or Java methods arguments. It can be defined as shown below.

$$
\begin{array}{l}
\mathit{MultiAssign} : \mathsf{JCode} \pfun \mathsf{JCode} \pfun \mathsf{JCode} \\
\mathit{MultiAssign}\ (\texttt{private type\_1 v\_1 ;...; private type\_n v\_n;}) \\
\qquad\qquad (\texttt{type\_1 newv\_1 ;..., type\_n newv\_n;}) = \\
\quad \texttt{this.v\_1 = newv\_1; ...; this.v\_n = newv\_n;}
\end{array}
$$

The instantiation of the hidden channels uses the function *HiddenCCreation*.

$$
\begin{array}{l}
\mathit{HiddenCCreation} : \mathit{ChanUseEnv} \pfun \mathsf{JCode} \\
\mathit{HiddenCCreation}\ \emptyset = \epsilon \\
\mathit{HiddenCCreation}\ \{c \mapsto t\} \cup \upsilon = \\
\quad \texttt{this.c = new Any2OneChannel();}\ \mathit{HiddenCCreation}\ \upsilon
\end{array}
$$

For a non-parametrised process *Proc*, we have that $[\![\mathit{Proc}]\!]^{\mathit{ParProc}}$ is similar. The only difference is that attributes corresponding to parameters (and their initialisation) are not needed.

At this stage, we have declared the class that represents a given process, its attributes, and its constructor. We are now left, with the definition of the method `run`'s body. This is determined by the process definition, which we consider next.

6.2.2 Transformation of Basic Processes

Each process is translated to an execution of an inner class that implements the class `jcsp.lang.CSProcess`. The first kind of process that we translate is

$$\textbf{begin}\ \mathit{PPars}_1\ \textbf{state}\ \mathit{PSt}\ \mathit{PPars}_2 \bullet \mathit{Main}.$$

The inner class definition starts by declaring all the state components of the process as attributes of the process class. In the translation of the process paragraphs, each schema expression and CSP action gives rise to a private method. Finally, the body of the method `run` is the result of the translation of the main action. We present below the definition of the function $[\![\_]\!]^{\mathit{Proc}}$.

$$
\begin{array}{l}
[\![\_]\!]^{\mathit{Proc}} : \mathsf{Proc} \pfun \mathsf{JCode} \\
[\![\textbf{begin}\ \mathit{PPars}_1\ \textbf{state}\ \mathit{PSt}\ \mathit{PPars}_2 \bullet \mathit{Main}]\!]^{\mathit{Proc}} = \\
\quad \texttt{(new CSProcess()\{} \\
\qquad (\mathit{StateDecl}\ \mathit{PSt}) \\
\qquad ([\![\mathit{PPars}_1\ \mathit{PPars}_2]\!]^{\mathit{PPars}}) \\
\qquad \texttt{public void run() \{}\ [\![\mathit{Main}]\!]^{\mathit{Action}}\ \texttt{\}} \\
\quad \texttt{\}).run();}
\end{array}
$$

The state declaration is defined as a schema expression. We use the function *StateDecl* to transform this schema expression into a `;`-separated list of Java attribute declarations.

$$
\begin{array}{l}
StateDecl : \mathsf{SchemaExp} \pfun \mathsf{JCode} \\
StateDecl\ [\, x_1 : T_1;\ \ldots;\ x_n : T_n \mid inv\,] = \\
\quad \texttt{private}\ (JType\ T_1)\ \texttt{x\_1;} \ldots \texttt{;}\ \texttt{private}\ (JType\ T_n)\ \texttt{x\_n;}
\end{array}
$$

Our strategy ignores the invariant since it has already been considered in the refinement of the process. Once all the actions are refined to code, in the presence of the invariant, it can be eliminated. It is kept in a *Circus* program just for documentation purposes.

6.2.3 Process Paragraphs.

The function $[\![\_]\!]^{PPars}$ translates the paragraphs within a *Circus* process. These paragraphs can either be axiomatic definitions, or (parametrised) actions.

Axiomatic definitions within processes are defined as private methods of the class that defines this process.

$$
\begin{array}{l}
[\![\_]\!]^{PPars} : \mathsf{PPar}^* \pfun \mathsf{JCode} \\
[\![\epsilon]\!]^{PPars} = \epsilon \\
[\![v : T \mid v = e_1\ PPars]\!]^{PPars} = \\
\quad \texttt{private}\ (JType\ T)\ \texttt{v() \{ return}\ (JExp\ e_1)\texttt{; \}} \\
\quad [\![PPars]\!]^{PPars}
\end{array}
$$

Both parametrised actions and non-parametrised actions are translated into private methods. However, the former requires that the parameters are declared as arguments of the new method.

$$
\begin{array}{l}
[\![N \mathrel{\widehat=} (Decl \bullet Action)\ PPars]\!]^{PPars} = \\
\quad \texttt{private void N(}ParamsArgs\ Decl\texttt{)\{}\ \ [\![\, Action \,]\!]^{Action}\ \ \texttt{\}} \\
\quad [\![PPars]\!]^{PPars}
\end{array}
$$

$$
\begin{array}{l}
[\![N \mathrel{\widehat=} Action\ PPars]\!]^{PPar} = \\
\quad \texttt{private void N()\{}\ \ [\![\, Action \,]\!]^{Action}\ \ \texttt{\}} \\
\quad [\![PPars]\!]^{PPars}
\end{array}
$$

The function $[\![\_]\!]^{Action}$ translates the body of a given action. These actions can be schema expressions, CSP actions, or commands. As already mentioned, that schema expressions have already been refined. We are then left with CSP actions and commands.

6.2.4 CSP Actions.

In the translation of each action, we consider that a new environment is available. The local variables environment *LocalVarEnv* : *VarEnv* is used to declare copies of the local variables in scope in the translation of parallel and recursive actions.

$$VarEnv == \text{seq}(\mathsf{N} \times \mathsf{Expression})$$

For each local variable in scope, the environment *LocalVarEnv* has a corresponding pair: its first element is the local variable name and the second element is its type. The translation function receives an action as argument and returns a Java code that implements this action.

$$[\![\_]\!]^{Action} : \mathsf{Action} \pfun \mathsf{JCode}$$

Besides, as for processes, we consider that, for each action, the channel environments *VisChanEnv* and *HidChanEnv* are available throughout its translation. However, these environments store information about how each channel is used within each action.

In the next sections, we present the translation of different *Circus* actions types.

Skip, Stop, and Chaos

The translations of *Skip* and *Stop* use basic JCSP classes; *Chaos* is translated to an infinite loop.

$$[\![Skip]\!]^{Action} = \texttt{(new Skip()).run();}$$
$$[\![Stop]\!]^{Action} = \texttt{(new Stop()).run();}$$
$$[\![Chaos]\!]^{Action} = \texttt{while(true)\{\};}$$

Communications

For non-typed input communications, we use the channels environment in order to identify the type of the input variable. Besides, the local variables environment is also used in order to verify if the input variable is already declared or not. We assign to the input variable the value read from the channel. We must also use Java casting, since the type of the objects transmitted through the channels are `java.lang.Object`.

$$[\![c?x \rightarrow Action]\!]^{Action} =$$
$$\textbf{let } commType = last\ (snd\ (ChanTypeEnv\ c)) \textbf{ in}$$
$$\textbf{if } (x \notin (SetFirst\ LocalVarEnv)) \textbf{ then}$$
$$\texttt{\{ (}JType\ commType\texttt{) x = (}JType\ commType\texttt{)c.read();}\ [\![Action]\!]^{Action}\ \texttt{\}}$$
$$\textbf{else } \texttt{x = (}JType\ commType\texttt{)c.read();}\ \ [\![Action]\!]^{Action}$$

The function *last* returns the last element of a given list. The function *SetFirst* receives a sequence of pairs and returns a set containing all the first elements of this sequence.

$$
\begin{array}{l}
SetFirst : \mathrm{seq}(\mathsf{N} \times \mathsf{Expression}) \pfun \mathbb{P}\,\mathsf{N} \\
SetFirst\ [\,] = \emptyset \\
SetFirst\ (x, T) : xs = \{x\} \cup (SetFirst\ xs)
\end{array}
$$

The output communication just writes in the channel the expression to be written.

$$[\![c!e \rightarrow Action]\!]^{Action} = \texttt{c.write(}JExp\ e\texttt{);}\ \ [\![\, Action]\!]^{Action}$$

For synchronisation channels, we need to know wether this channel is declared an input or output channel, in order to read from the channel or to write to the channel, respectively. This information is retrieved either from the *VisChanEnv*, or from the *HidChanEnv* environments.

$$
\begin{array}{l}
[\![c \rightarrow Action]\!]^{Action} = \\
\quad \textbf{if}\ (c \in \mathrm{dom}\ VisChanEnv)\ \textbf{then} \\
\quad\quad \textbf{if}\ (VisChanEnv\ c = I \lor VisChanEnv\ c = A)\ \textbf{then} \\
\quad\quad\quad \texttt{c.read();}\ \ [\![\, Action]\!]^{Action} \\
\quad\quad \textbf{else}\ \texttt{c.write(null);}\ \ [\![\, Action]\!]^{Action} \\
\quad \textbf{else if}\ (c \in \mathrm{dom}\ HidChanEnv)\ \textbf{then} \\
\quad\quad \textbf{if}\ (HidChanEnv\ c = I \lor HidChanEnv\ c = A)\ \textbf{then} \\
\quad\quad\quad \texttt{c.read();}\ \ [\![\, Action]\!]^{Action} \\
\quad\quad \textbf{else}\ \texttt{c.write(null);}\ \ [\![\, Action]\!]^{Action}
\end{array}
$$

In JCSP, the method `write` receives the object that must be written as argument. As we do not have any value to communicate, we communicate the `null` value.

Sequential composition

Sequential compositions can be simply translated to a Java sequential composition.

$$[\![Action_1;Action_2]\!]^{Action} = [\![Action_1\,]\!]^{Action}\ \ ;\ \ \ldots\ \ ;\ \ [\![\, Action_n]\!]^{Action}$$

External Choice

The translation of an external choice uses the `jcsp.lang.Alternative` class. The idea is to create an alternative in which all the initial channels of all actions, that are not hidden, take part. Furthermore, in this case, we consider that all nested guarded actions in the form $\Box_i\, g_i\ \&\ \Box_j\, g_{i_j}\ \&\ A_{i_j}$ have already been refined to

guarded actions in the form $\Box_{i,j}\, g_i \wedge g_{i_j}\ \&\ A_{i_j}$. Besides, unguarded actions have already been refined to *true* guarded actions. For instance, the action $A_1 \Box A_2$ has been refined to *true* $\&\ A_1 \Box$ *true* $\&\ A_2$. These are very simple refinements and they help in the definition of the auxiliary function *Guard* below. The external choice translation is defined as follows.

$[\![Action_1 \Box \ldots \Box Action_n]\!]^{Action} =$

```
    Guard[] guards =
        new Guard[]{InitCAttr Action_1 , ... , InitCAttr Action_n};
    final Alternative alt = new Alternative(guards);
    DeclConstants (ExtractInitChannels Action_1) 0
    ...
    DeclConstants
        (ExtractInitChannels Action_n) (#(ExtractInitChannels Action_{n-1}))
    boolean[] g =
        new boolean[]{Guard Action_1 , ... , Guard Action_n};

    switch(alt.fairSelect(g)) {
        Cases (ExtractInitChannels Action_1) Action_1
        ...
        Cases (ExtractInitChannels Action_n) Action_n
    }
```

First, it defines the events competing for selection by a declared `Alternative` process `alt`. One integer constant is declared for each one of these events. The guards g_i of each action are also declared within a `boolean` array. Finally, the alternative is made, and the corresponding action is executed.

The function *InitCAttr* returns a `,`-separated list of all the visible initials channels of a given action.

$$InitCAttr : \mathsf{Action} \pfun \mathsf{JCode}$$
$$InitCAttr\ Action = DecAttrChannels\ (ExtractInitChannels\ Action)$$

The function $ExtractInitChannels : \mathsf{Action} \pfun \mathrm{seq}(\mathsf{N} \times \mathsf{Predicate})$ returns a list of pairs. For each initial visible channels of the given action, it includes a new pair in this list: the first element is the channel name, and the second element is a predicate that represents its guard (the condition that must hold in order to the channel to become available).

The function *DecAttrChannels* can be defined as

$$DecAttrChannels : \mathrm{seq}(\mathsf{N} \times \mathsf{Predicate}) \pfun \mathsf{JCode}$$
$$DecAttrChannels\ [\,] = \epsilon$$
$$DecAttrChannels\ (c, p) : [\,] = \mathsf{c}$$
$$DecAttrChannels\ (c, p) : cs = \mathsf{c},\ DecAttrChannels\ cs$$

The function *DeclConstants* returns a `;`-separated list of `int` constant declarations, one for each channel in the given channel list. The first constant is initialised with n; each subsequent constant is initialised with the previous constant value incremented by one. These constants are used in the `switch` block to identify each possible choice made by the `fairSelect` method.

$$
\begin{array}{l}
DeclConstants : \mathrm{seq}(\mathsf{N} \times \mathsf{Predicate}) \pfun \mathbb{N} \pfun \mathsf{JCode} \\
DeclConstants\ [\,]\ n = \epsilon \\
DeclConstants\ (c, p) : cs\ n = \\
\quad \texttt{final int CONST\_}(Capitals\ c)\ \texttt{= n;}\ \ DeclConstants\ cs\ (n + 1)
\end{array}
$$

The function *Capitals* returns the given argument in capitals.

The function *Guard* returns the guard of the given action.

$$
\begin{array}{l}
Guard : \mathsf{Action} \pfun \mathsf{JCode} \\
Guard\ (g\ \&\ A) = (JExp\ g)
\end{array}
$$

Finally, the function *Cases* returns a sequence of Java `case` blocks, one for each channel in the given list.

$$
\begin{array}{l}
Cases : \mathrm{seq}(\mathsf{N} \times \mathsf{Predicate}) \pfun \mathsf{Action} \pfun \mathsf{JCode} \\
Cases\ [\,]\ a = \epsilon \\
Cases\ ((c, p) : cs)\ a = \\
\quad \texttt{case CONST\_}(Capitals\ c)\texttt{:} \\
\quad\quad \texttt{\{}\ [\![\, a \,]\!]^{Action}\ \texttt{\}} \\
\quad\quad \texttt{break;} \\
\quad Cases\ cs\ a
\end{array}
$$

As an example of the translation of an external choice we consider the action below.

$$
\begin{array}{l}
((x > 0)\ \&\ a_1 \rightarrow Skip\ \Box\ (x \leq 0)\ \&\ a_2 \rightarrow Skip) \\
\Box\ ((x > 0)\ \&\ b_1 \rightarrow Stop\ \Box\ (x \leq 0)\ \&\ b_2 \rightarrow Stop)
\end{array}
$$

The first part of the translation declares an array containing all the visible channels within the action, and an `Alternative` on this array.

```
Guard[] guards = new Guard[]{a1,a2,b1,b2};
final Alternative alt = new Alternative(guards);
```

Then, it declares all the constants that are used in the `switch` block to identify each possible choice made by the `fairSelect` method.

```
final int CONST_A_1 = 0;
final int CONST_A_2 = 1;
final int CONST_B_1 = 2;
final int CONST_B_2 = 3;
```

Next, it declares the array containing the guards for each branch of the action.

```
boolean[] g = new boolean[]{x>0,x<=0,x>0,x<=0};
```

Finally, we have the `switch` block. For each possible return value from the invocation of the method `fairSelect` in the previously declared `Alternative`, we declare a new `case`: its body consists of reading from the corresponding channel and followed by the translation of the corresponding action.

```
switch(alt.fairSelect(g)) {
    case CONST_A_1: { a_1.read(); (new Skip()).run(); break; }
    case CONST_A_2: { a_2.read(); (new Skip()).run(); break; }
    case CONST_B_1: { b_1.read(); (new Stop()).run(); break; }
    case CONST_B_2: { b_2.read(); (new Stop()).run(); break; }
}
```

For a special form of external choice, in which the guards are mutually exclusive, we may adopt a different strategy to obtain an `if-then-else` block as shown below.

$$[\![g_1 \;\&\; Action_1 \;\Box\; \ldots \;\Box\; g_n \;\&\; Action_n]\!]^{Action} =$$

```
    if((JExp g_1)){
        [[Action_1]]^Action
    } else if
        ...
    } else if((JExp g_n)){
        [[Action_n]]^Action
    } else {
        (new Stop()).run();
    }
```

provided $\forall\, i, j \bullet i \neq j \Rightarrow (g_i \Rightarrow \neg\, g_j)$

This simplifies the generated Java code, and does not require the guarded actions to be explored in the translation of the external choice.

Internal Choice

The internal choice translation randomly chooses an action, and then, starts to behave as such. It uses the static method `int generateNumber(int min, int max)`

of class `proj.util.RandomGenerator` (See Section 6.2.7) to make this random choice.

$[\![Action_1 \sqcap \ldots \sqcap Action_n]\!]^{Action} =$

```
    int choosen = RandomGenerator.generateNumber(0,n);
    switch(choosen) {
        case 0: { [[ Action_1 ]]^Action } break;
        ...
        case n: { [[ Action_n ]]^Action } break;
    }
```

Differently from the external choice, the choice is made by the program.

Parallelism

In the translation of a parallelism, we have to deal with the partition of the variables in scope. For this reason, we use auxiliary variables to make copies of each state component that takes part in one of the partitions. They are declared and initialised using the function *InitAuxVars*. The body of each branch is translated and each reference to a state component is replaced with the corresponding copy. After running the parallelism, we have to merge the values of the variables in each partition respecting the partitions declaration in the parallelism.

Local variables in scope present the same problem as state components. However, since they are not class attributes as the state components, but local variables, they cannot be directly accessed in the inner classes created for each parallel action. For this reason, their copies are not initialised when declared, as the copies of the state components. They are initialised in the constructor of each parallel action class with the value given to the constructor. Nevertheless, these local variables are included in the merge of the variables after the execution of the parallelism.

In the following, the expression **let** $x_1 = exp_1, \ldots, x_n = exp_n$ **in** exp is used to denote the result of substituting each exp_i for the corresponding x_i in exp. For a

fresh index value *index*, the translation of parallelism can be defined as follows.

$[\![Action_1 \; [\![\; NSExp_1 \mid CSExp \mid NSExp_2 \;]\!] \; Action_2]\!]^{Action} =$
let $LName =$ `ParallelLeftBranch_`$index$,
$RName =$ `ParallelRightBranch_`$index$ **in**

```
class LName implements CSProcess {
    InitAuxVars (NSExp1 \ (setFirst LocalVarEnv)) index L
    DeclLocalVars LocalVarEnv index L
    public LName((LocalVarsArg LocalVarEnv)) {
        InitLocalVars LocalVarEnv index L
     }
    public void run () {
        RenameVars [[Action1]]^Action
                   (NSExp1 ∪ (SetFirst LocalVarEnv))
                   index L
    }
}
CSProcess left_index =
    new LName(JList (ListFirst LocalVarEnv));
class RName implements CSProcess {
    InitAuxVars (NSExp2 \ (setFirst LocalVarEnv)) index R
    DeclLocalVars LocalVarEnv index R
    public RName((LocalVarsArg LocalVarEnv)) {
        InitLocalVars LocalVarEnv index R
     }
    public void run () {
        RenameVars [[Action2]]^Action
                   (NSExp2 ∪ (SetFirst LocalVarEnv))
                   index R
    }
}
CSProcess right_index =
    new RName(JList (ListFirst LocalVarEnv));
CSProcess[] processes_index =
    new CSProcess[]{left_index,right_index};
(new Parallel(processes_index)).run ();
MergeVars LName NSExp1 index L
MergeVars RName NSExp2 index R
```

The fresh index value is used to avoid possible clashes in the name of the inner classes and auxiliary variables needed in this translation. For instance, let us consider the action $(A_1 \parallel A_2);(A_3 \parallel A_4)$. The translation os this action is equivalent to the

translation of $A_1 \parallel A_2$ in sequence with the translation of $A_3 \parallel A_4$. If we did not have a new index for each translation, this would lead to the creation of inner classes and auxiliary variables with same names (i.e. `ParallelLeftBranch` and `left`). Instead, we use a fresh *index* when translating both actions. Considering that the indexes are natural numbers, this leads to the declaration of different inner classes names `ParallelLeftBranch_0` and `ParallelLeftBranch_1`, and different auxiliary variables `left_0` and `left_1`.

The translation of the parallelism declares one inner class, and instantiates an object of this class, for each branch of the parallelism. Each branch creates its own copy of the variables in scope. After the execution of the parallelism, a merge is made in order to retrieve the final values of the variables in scope from the corresponding copy.

The following type is used to indicate if the auxiliary variable is on the left (L) or the right (R) side partition of the parallelism.

$$LeftRight == L \mid R$$

The declaration of the auxiliary variables is very simple. For each variable used in a parallelism partition, we declare two new variables; one for each action of the parallelism. The function *InitVarsAux*, declared below, declares and initialises all the existent variables in the partition given as argument, considering the partition side given as argument.

$$\begin{array}{l} InitAuxVars : \mathbb{P}\,\mathsf{N} \pfun \mathbb{N} \pfun LeftRight \pfun \mathsf{JCode} \\ InitAuxVars\ \emptyset\ index\ S = \epsilon \\ InitAuxVars\ (\{x\} \cup xs)\ index\ L = \\ \quad \texttt{public}\ (JType\ (CType\ x))\ \texttt{aux\_left\_x\_}index\ \texttt{= x;} \\ \quad InitAuxVars\ xs\ index\ L \\ InitAuxVars\ (\{x\} \cup xs)\ index\ R = \\ \quad \texttt{public}\ (JType\ (CType\ x))\ \texttt{aux\_right\_x\_}index\ \texttt{= x;} \\ \quad InitAuxVars\ xs\ index\ R \end{array}$$

The function *DeclLocalVars* is very similar to the function *InitAuxVars*; it, however, does not initialise the variable, since its initial value is received in the constructor of the inner class.

$$\begin{array}{l} DeclLocalVars : \mathrm{seq}(\mathsf{N} \times \mathsf{Expression}) \pfun \mathbb{N} \pfun LeftRight \pfun \mathsf{JCode} \\ DeclLocalVars\ [\,]\ index\ x = \epsilon \\ DeclLocalVars\ ((x, T) : xs)\ index\ L = \\ \quad \texttt{public}\ (JType\ T)\ \texttt{aux\_left\_x\_}index\texttt{;}\ DeclLocalVars\ xs\ index\ L \\ DeclLocalVars\ ((x, T) : xs)\ index\ R = \\ \quad \texttt{public}\ (JType\ T)\ \texttt{aux\_right\_x\_}index\texttt{;}\ DeclLocalVars\ xs\ index\ R \end{array}$$

The constructor of each branch receives the values of each local variable in context, and initialises their local copies. The function *LocalVarsArg* declares the arguments of the constructor.

$$
\begin{array}{l}
\mathit{LocalVarsArg} : \mathrm{seq}(\mathsf{N} \times \mathsf{Expression}) \pfun \mathsf{JCode} \\
\mathit{LocalVarsArg}\ [\,] = \epsilon \\
\mathit{LocalVarsArg}\ (x, T) : [\,] = (\mathit{JType}\ T)\ \texttt{x} \\
\mathit{LocalVarsArg}\ (x, T) : xs = (\mathit{JType}\ T)\ \texttt{x, }\ \mathit{LocalVarsArg}\ xs
\end{array}
$$

The function *InitLocalVars* initialises the copies of each local variable in scope within a branch.

$$
\begin{array}{l}
\mathit{InitLocalVars} : \mathrm{seq}(\mathsf{N} \times \mathsf{Expression}) \pfun \mathbb{N} \pfun \mathit{LeftRight} \pfun \mathsf{JCode} \\
\mathit{InitLocalVars}\ [\,]\ \mathit{index}\ x = \epsilon \\
\mathit{InitLocalVars}\ ((x, T) : xs)\ \mathit{index}\ L = \\
\quad \texttt{this.aux\_left\_x\_}\mathit{index}\ \texttt{= x;}\ \mathit{DeclLocalVars}\ xs\ \mathit{index}\ L \\
\mathit{InitLocalVars}\ ((x, T) : xs)\ \mathit{index}\ R = \\
\quad \texttt{this.aux\_right\_x\_}\mathit{index}\ \texttt{= x;}\ \mathit{DeclLocalVars}\ xs\ \mathit{index}\ R
\end{array}
$$

The function *RenameVars* replaces, in the Java code given as arguments, all the occurrences of the variables in the set given as argument by its corresponding copy. We use the notation $\texttt{C}[X \setminus Y]$ to represent the substitution of all variables in Y for the corresponding variable in X in the Java code `C`.

$$
\begin{array}{l}
\mathit{RenameVars} : \mathsf{JCode} \pfun \mathbb{P}\,\mathsf{N} \pfun \mathbb{N} \pfun \mathit{LeftRight} \pfun \mathsf{JCode} \\
\mathit{RenameVars}\ \mathit{jcode}\ \emptyset\ \mathit{index}\ x = \epsilon \\
\mathit{RenameVars}\ \mathit{jcode}\ (\{x\} \cup xs)\ \mathit{index}\ L = \\
\quad \mathit{RenameVars}\ (\mathit{jcode}[\texttt{x} \setminus \texttt{aux\_left\_x\_}\mathit{index}])\ xs\ \mathit{index}\ L \\
\mathit{RenameVars}(\mathit{jcode}, \{x\} \cup xs, \mathit{index}, R) = \\
\quad \mathit{RenameVars}\ (\mathit{jcode}[\texttt{x} \setminus \texttt{aux\_right\_x\_}\mathit{index}])\ xs\ \mathit{index}\ R
\end{array}
$$

The function *MergeVars* defines the value of the auxiliary variables in terms of that of the auxiliary one. If the variable is in the first partition (L), it uses the `aux_left_` value; otherwise (R) it uses the `aux_right_` value.

$$
\begin{array}{l}
\mathit{MergeVars} : \mathsf{N} \pfun \mathbb{P}\,\mathsf{N} \pfun \mathbb{N} \pfun \{L, R\} \pfun \mathsf{JCode} \\
\mathit{MergeVars}\ \mathit{name}\ \emptyset\ \mathit{index}\ x = \epsilon \\
\mathit{MergeVars}\ \mathit{LName}\ (\{x\} \cup xs)\ \mathit{index}\ L = \\
\quad \texttt{x = ((LName)processes\_}\mathit{index}\texttt{[0]).aux\_left\_x\_index;} \\
\quad \mathit{MergeVars}\ \mathit{LName}\ xs\ \mathit{index}\ L \\
\mathit{MergeVars}\ \mathit{RName}\ (\{x\} \cup xs)\ \mathit{index}\ R = \\
\quad \texttt{x = ((RName)processes\_}\mathit{index}\texttt{[1]).aux\_right\_x\_index;} \\
\quad \mathit{MergeVars}\ \mathit{RName}\ xs\ \mathit{index}\ R
\end{array}
$$

By way of illustration, let us consider the translation of a process that has only one state component $x : \mathbb{N}$, with one local variable $local : \mathbb{N}$ in scope. Given these conditions, consider the translation of the following action, given an index 0 as argument.

$$x := 0 \parallel [\{x\} \mid \{local\}] \parallel local := 1$$

The obtained Java program starts by declaring the class that represents the left hand-side branch of the parallelism.

```
class ParallelLeftBranch_0 implements CSProcess {
```

The left hand-side partition contains the variable x. Its copy is declared as an attribute of the inner class that represents the left hand-side branch of the parallelism. Its initial value is the value of the original variable.

```
    public Integer aux_left_x_0 = x;
```

The local variable is also declared as an inner class attribute. However, it is initialised only in the constructor of the inner class, that receives this initial value as argument.

```
    public Integer aux_left_local_0;
    public ParallelLeftBranch_0(Integer local) {
        this.aux_left_local_0 = local;
    }
```

The `run` method contains the execution of the action in the left hand side of the parallelism. However, we replace the references to all state components and local variables in scope with references to their respective copies. This finishes the declaration of the left action class.

```
    public void run () {
        aux_left_x_0 = new Integer(0);
    }
}
```

Next, we instantiate an object of this new class using the local variables in scope as arguments of the constructor.

```
CSProcess left_0 = new ParallelLeftBranch_0(local);
```

In a similar way, we declare and instantiate the class that represents the right hand-side of the parallelism. This, however, does not include the state component x, since it is not declared in the right hand-side partition.

```
class ParallelRightBranch_0 implements CSProcess {
    public Integer aux_right_local_0;

    public ParallelRightBranch_0(Integer local) {
        this.aux_right_local_0 = local;
    }
    public void run () {
        aux_right_local_0 = new Integer(1);
    }
}
CSProcess right_0 = new ParallelRightBranch_0(local);
```

After declaring both branches of the parallelism, we instantiate and run a JCSP `Parallel` object as follows.

```
CSProcess[] processes_0 = new CSProcess[]{left_0,right_0};
(new Parallel(processes_0)).run ();
```

Finally, after the execution of the parallelism, we merge the values of the state components and local variables in scope.

```
x = ((ParallelLeftBranch_0)processes_0[0]).aux_left_x_0;
local =
    ((ParallelRightBranch_0)processes_0[1]).aux_right_local_0;
```

This finishes the Java code corresponding to the previous declared parallel action.

Recursion

The recursion operator is also translated using an inner class to declare the body of the recursion as a process. As for parallelism, the use of an inner class requires that copies of the local variables in scope are declared as attributes of this new inner class, and initialised in the constructor with the values given as arguments. The `run` method of this new inner class executes the body of the recursion and then, where the recursion occurs, it instantiates a new object of this class, and executes it. Again, the references to the local variables are replaced by references to their copies. After the declaration of the recursion class, we instantiate and run it. After its execution, as for parallel actions, a merge happens to restore the values of the local variables in scope. For a fresh index value *index*, the translation of a recursion

can be defined as follows.

$$
\begin{array}{l}
[\![\mu\, X \bullet Action(X)]\!]^{Action} = \\
\quad \texttt{class I\_}index\ \texttt{implements CSProcess \{} \\
\quad\quad DeclLocalVars\ LocalVarEnv\ index\ L \\
\quad\quad \texttt{public I\_}index\texttt{(}LocalVarsArg\ LocalVarEnv\texttt{) \{} \\
\quad\quad\quad InitLocalVars\ LocalVarEnv\ index\ L \\
\quad\quad \texttt{\}} \\
\quad\quad \texttt{public void run() \{} \\
\quad\quad\quad RenameVars \\
\quad\quad\quad\quad [\![Action(\texttt{(}RunRecursion\ index\texttt{)})]\!]^{Action} \\
\quad\quad\quad\quad (SetFirst\ LocalVarEnv)\ index\ L \\
\quad\quad \texttt{\}} \\
\quad \texttt{\};} \\
\quad RunRecursion\ index
\end{array}
$$

For the same reason as for the traduction of parallelism, we use a fresh index in the name of the inner class created for the recursion.

The function *RunRecursion* instantiates a recursion process, invokes its `run` method, and finally collects the values of the auxiliary variables.

$$
\begin{array}{l}
RunRecursion : \mathbb{N} \pfun \mathsf{JCode} \\
RunRecursion\ index = \\
\quad \texttt{I\_}index\ \texttt{i\_}index\texttt{\_}newIndex\ \texttt{=} \\
\quad\quad \texttt{new I\_}index\texttt{(}JList\ (ListFirst\ LocalVarEnv)\texttt{));} \\
\quad \texttt{i\_}index\texttt{\_}newIndex\texttt{.run();} \\
\quad MergeLocalVars\ LocalVarEnv\ index\ newIndex\ L
\end{array}
$$

where *newIndex* is, again, a fresh index. It is used in order to avoid possible clashes between the names of the new auxiliary variables (i.e. in case we have two instantiations in sequence).

The function *ListFirst* receives a sequence of pairs and returns a list containing all the first elements of this sequence.

$$
\begin{array}{l}
ListFirst : \mathrm{seq}(\mathsf{N} \times \mathsf{Expression}) \pfun \mathrm{seq}\, \mathsf{N} \\
ListFirst\ [\,] = [\,] \\
ListFirst\ (x, T) : xs = x : (ListFirst\ xs)
\end{array}
$$

The function *JList* transforms a list in a ,-separated list of Java arguments.

$$
\begin{array}{l}
JList : \mathrm{seq}\, \mathsf{N} \pfun \mathsf{JCode} \\
JList\ [\,] = \epsilon \\
JList\ x : [\,] = \ \texttt{x\_1} \\
JList\ x : xs = \ \texttt{x\_1, }\ (JList\ xs)
\end{array}
$$

The function *MergeLocalVars* is very similar to the previously defined *MergeVars*. It assigns the values of the auxiliary variables to the original ones. If the variable is in the first partition (L), it uses the `aux_left_` value; otherwise (R) it uses the `aux_right_` value. However, as no parallelism was declared in this case, only a new class, it directly accesses the recently created object corresponding to the recursion to get the current values of the local variables copies.

$$
\begin{array}{l}
\mathit{MergeLocalVars} : \mathbb{P}\,\mathsf{N} \pfun \mathbb{N} \pfun \mathbb{N} \pfun \{L, R\} \pfun \mathsf{JCode} \\
\mathit{MergeLocalVars}\ \emptyset\ \mathit{index}\ \mathit{newIndex}\ x = \epsilon \\
\mathit{MergeLocalVars}\ (\{x\} \cup \mathit{xs})\ \mathit{index}\ \mathit{newIndex}\ L = \\
\quad \texttt{x = i\_}\mathit{index}\texttt{\_}\mathit{newIndex}\texttt{.aux\_left\_x\_}\mathit{index}\texttt{;} \\
\quad \mathit{MergeLocalVars}\ \mathit{xs}\ \mathit{index}\ \mathit{newIndex}\ L \\
\mathit{MergeLocalVars}\ (\{x\} \cup \mathit{xs})\ \mathit{index}\ \mathit{newIndex}\ R = \\
\quad \texttt{x = i\_}\mathit{index}\texttt{\_}\mathit{newIndex}\texttt{.aux\_right\_x\_}\mathit{index}\texttt{;} \\
\quad \mathit{MergeLocalVars}\ \mathit{xs}\ \mathit{index}\ \mathit{newIndex}\ R
\end{array}
$$

For instance, let us consider we are translating a process that has no state component, and that we have one local variable $\mathit{local} : \mathbb{N}$ in scope. Given these conditions, consider the translation of the following action.

$$\mu\, X \bullet x := x + 1; X$$

In the resulting program, a class that corresponds to the recursion body is declared. The only attribute of this new class is the a copy of the local variable *local*. This copy is initialised in the constructor of the class with the value given as argument. As for parallel actions, we replace the references to local variables in scope by references to their respective copies. Furthermore, we replace the recursion point X by an instantiation and execution of an object of this class.

```
class I_0 implements CSProcess {
    public Integer aux_left_local_0;
    public I_0(Integer local) {
        this.aux_left_local_0 = local;
    }
    public void run () {
        aux_left_local_0 =
            new Integer(aux_left_local_0.intValue()+1);
        I_0 i_0_1 = new I_0(aux_left_local_0);
        i_0_1.run();
        aux_left_local_0 = i_0_1.aux_left_local_0;
    }
}
```

After declaring the class corresponding to the recursion, we instantiate and run an object of this class. Finally, we restore the values of the local variables in scope.

```
I_0 i_0_2 = new I_0(local);
i_0_2.run();
local = i_0_2.aux_left_local_0;
```

This finishes the translation of the recursive action.

Action Invocation

If we have a *Circus* action invocation, all we have to do is to translate it to a method invocation. If no parameter is given, the method invocation has no parameters.

$$[\![ActName]\!]^{Action} = \texttt{ActName();}$$

However, if any parameter is given, we use a Java expression corresponding to each parameter in the method invocation.

$$[\![ActName(e_1, \ldots, e_n)]\!]^{Action} = \texttt{ActName((}JExp\ e_1\texttt{),...,(}JExp\ e_n\texttt{));}$$

For invocation of a (parametrised) action, the translation is a new inner class. The name of this class also uses a fresh *index* in order to avoid name clashes.

$$\begin{array}{l}
[\![(Decl \bullet Action)\,(e_1, \ldots, e_n)]\!]^{Action} = \\
\quad DeclareActionClass\ Decl\ Action\ index \\
\quad \texttt{I_}index\ \texttt{i_}index\texttt{_}index\ \texttt{=} \\
\qquad \texttt{new I_}index\texttt{((}JExp\ e_1\texttt{),...,(}JExp\ e_n\texttt{),} \\
\qquad\qquad (JList\ (ListFirst\ LocalVarEnv))\texttt{);} \\
\quad \texttt{i_}index\texttt{_}index\texttt{.run();} \\
\quad MergeLocalVars\ index\ index\ L
\end{array}$$

First, it declares the an inner class that corresponds to the parameterised action. Then, it instantiates an object of this class with the given arguments, and invokes its `run` method. Finally, it restores the values of the local variables in scope.

The function *DeclareActionClass* declares a class representing the parametrised action. As for parallel and recursive actions, each of the local variables in scope has a corresponding copy as an attribute of the new class. The action parameters are also declared as attributes of the new class. Both the local variable copies attributes and the parameters attributes are initialised within the class constructor with the corresponding values given as arguments. The `run` method of the new class executes

the parametrised action. However, the references to the local variables are replaced by references to their copies.

$$
\begin{array}{l}
\mathit{DeclareActionClass} : \mathsf{Decl} \pfun \mathsf{Action} \pfun \mathbb{N} \pfun \mathsf{JCode} \\
\mathit{DeclareActionClass}\ \mathit{Decl}\ \mathit{Action}\ \mathit{index} \\
\quad \textbf{let}\ \mathit{sep} = \textbf{if}\ \mathit{LocalVarEnv} = [\,]\ \textbf{then}\ \epsilon\ \textbf{else}\ \texttt{,}\ \textbf{in} \\
\quad\quad \texttt{class I\_}\mathit{index}\ \texttt{implements CSProcess \{} \\
\quad\quad\quad \mathit{ParamsDecl}\ \mathit{Decl} \\
\quad\quad\quad \mathit{DeclLocalVars}\ \mathit{LocalVarEnv}\ \mathit{index}\ L \\
\quad\quad\quad \texttt{public I\_}\mathit{index}\texttt{(}(\mathit{ParamsArgs}\ \mathit{Decl})\ \mathit{sep} \\
\quad\quad\quad\quad\quad\quad (\mathit{LocalVarsArg}\ \mathit{LocalVarEnv})\texttt{)\{} \\
\quad\quad\quad\quad \mathit{MultiAssign}\ (\mathit{ParamsDecl}\ \mathit{Decl})\ (\mathit{ParamsArgs}\ \mathit{Decl}) \\
\quad\quad\quad\quad \mathit{InitLocalVars}\ \mathit{index}\ L \\
\quad\quad\quad \texttt{\}} \\
\quad\quad\quad \texttt{public void run ()\{} \\
\quad\quad\quad\quad \mathit{RenameVars}\ (\llbracket \mathit{Action} \rrbracket^{\mathit{Action}}) \\
\quad\quad\quad\quad\quad\quad (\mathit{SetFirst}\ \mathit{LocalVarEnv})\ \mathit{index}\ L \\
\quad\quad\quad \texttt{\}} \\
\quad\quad \texttt{\}}
\end{array}
$$

Iterated Operators

The iterated operators are translated using `for` loops. Our strategy considers that only finite sets are used to index the operators. Free types, abbreviations, and subsets of $\mathbb{N}$ and $\mathbb{Z}$ are the acceptable sets for typing indexing variables in our strategy. In the case of subsets of $\mathbb{N}$ and $\mathbb{Z}$, the elements must be equally spaced. For a given variable $x : T$, the function *Inc* returns a Java expression that increments the variable x of type T to the next value of T. For instance, we have that $\mathit{Inc}\ x : \{0, 1, 2\}$ =`x=new Integer(x.intValue()+1)` and $\mathit{Inc}\ y : \{1, 11, 21\}$ =`y=new Integer(y.intValue()+10)`. For free types and abbreviations F we have that $\mathit{Inc}\ x : F$ =`x=new` F`(x.getValue()+1)`. We also consider that the declarations are in the form $x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$.

The first iterated operator on actions is the **iterated sequential composition**, $\fatsemi$. In this case we use the auxiliary function *InstActions* to create a vector of actions. Then, we execute each action within this vector.

$$
\begin{array}{l}
\llbracket \fatsemi\ x_1 : T_1;\ \ldots;\ x_n : T_n \bullet \mathit{Action} \rrbracket^{\mathit{Action}} = \\
\quad \mathit{InstActions}\ \texttt{procVec\_}\mathit{index}\ (x_1 : T_1;\ \ldots;\ x_n : T_n)\ \mathit{Action}\ \mathit{index} \\
\quad \texttt{for(int i = 0; i < procVec\_}\mathit{index}\texttt{.size(); i++) \{} \\
\quad\quad \texttt{((CSProcess)procVec\_}\mathit{index}\texttt{.elementAt(i)).run();} \\
\quad \texttt{\}}
\end{array}
$$

The function *InstActions* instantiates a vector containing each of the actions obtained by considering each of the possible values of the indexing variables. First, we declare an inner class representing a parametrised action using function *InstActions*. The parameters of this action are the indexing variables. Then, it declares a nested loop. In each iteration of the loop, it instantiates a process, using the declared class constructor, and stores it a `Vector` variable, which name is given as argument to the function. The current values of the indexing variables are given as arguments to the constructor.

For a fresh *index*, the function *InstActions* can be defined as follows.

$$
\begin{array}{l}
\mathit{InstActions} : \mathsf{N} \pfun \mathsf{Decl} \pfun \mathsf{Action} \pfun \mathbb{N} \pfun \mathsf{JCode} \\
\mathit{InstActions}\ \mathit{procVecName}\ (x_1 : T_1;\ \ldots;\ x_n : T_n)\ \mathit{Action}\ \mathit{index} = \\
\quad \texttt{Vector}\ \mathit{procVecName}\ \texttt{= new Vector();} \\
\quad \mathit{DeclareActionClass}\ (x_1 : T_1;\ \ldots;\ x_n : T_n)\ \mathit{Action}\ \mathit{index} \\
\quad \texttt{for ((}\mathit{JType}\ T_1\texttt{) x\_1 = (}\mathit{Min}\ T_1\texttt{);} \\
\qquad \texttt{x\_1.compareTo((}\mathit{Max}\ T_1\texttt{))<=0; (}\mathit{Inc}\ x_1 : T_1\texttt{)\{} \\
\quad \ldots \\
\quad \texttt{for ((}\mathit{JType}\ T_n\texttt{) x\_n = (}\mathit{Min}\ T_n\texttt{);} \\
\qquad \texttt{x\_n.compareTo((}\mathit{Max}\ T_n\texttt{))<=0; (}\mathit{Inc}\ x_n : T_n\texttt{))\{} \\
\qquad \mathit{procVecName}\texttt{.addElement(new I\_}\mathit{index}\texttt{(x\_1,...,x\_n));} \\
\quad \texttt{\}} \\
\quad \ldots \\
\texttt{\}}
\end{array}
$$

The functions *Min* and *Max* return the minimum and the maximum values that a variable of the type given as argument may assume, respectively. In the case of free types and abbreviations, these are stored in static constants in the classes created when translating the given free type or abbreviation (See Section 6.2.6). For a type F, we have that the constants `F.MIN_F` and `F.MAX_F` represent the minimum and the maximum values of F, respectively.

The **indexed external choice**, however, cannot be directly translated since we need to know which processes take part in the external choice to translate it. For this reason, they must be expanded before being translated.

The **indexed internal choice** chooses a value for each indexing variable, and then runs the action with the randomly chosen values for the indexing variables in scope. It also uses the class `proj.util.RandomGenerator` to make these random choices. For a fresh *index*, its translation is defined as follows.

$$
\begin{array}{l}
[\![\sqcap\, x_1 : T_1;\ \ldots;\ x_n : T_n \bullet \mathit{Action}]\!]^{\mathit{Action}} = \\
\quad \mathit{ChooseIndexVars}\ (x_1 : T_1;\ \ldots;\ x_n : T_n) \\
\quad \mathit{DeclareActionClass}\ (x_1 : T_1;\ \ldots;\ x_n : T_n)\ \mathit{Action}\ \mathit{index} \\
\quad \texttt{(new I\_}\mathit{index}\texttt{(x\_1,...,x\_n)).run();}
\end{array}
$$

The function *ChooseIndexVars* randomly chooses a value for each indexing variable. Then, it instantiates the variable with a chosen value.

$$
\begin{array}{l}
ChooseIndexVars : \mathsf{Decl} \pfun \mathsf{JCode} \\
ChooseIndexVars\ \epsilon = \epsilon \\
ChooseIndexVars\ (x : T;\ Decs) = \\
\quad (JType\ T)\ \texttt{x =} \\
\quad\quad \texttt{new}\ (JType\ T)\texttt{(RandomGenerator.generateNumber(} \\
\quad\quad\quad\quad\quad\quad (Min\ T)\texttt{,}(Max\ T)\texttt{)))}\texttt{;} \\
\quad ChooseIndexVars\ Decs
\end{array}
$$

As the external choice, the **indexed parallelism and interleaving** of actions in the form described below must also be extended before being translated.

$$\|\ i : (Min\ T)..(Max\ T)\}\ [\![\ \alpha_i \mid cs \mid \bigcup_{j:(Inc\ i:T)..(Max\ T)\}} \alpha_j]\!] \bullet A(i)$$

For instance, consider the following indexed parallelism.

$$\|\ i : \{0..2\}\ [\![\ \alpha_i \mid cs \mid \bigcup_{j:\{i+1..2\}} \alpha_j]\!] \bullet A(i)$$

In our strategy, before its translation, we must expand as defined below.

$$A(0)\ [\![\ \alpha_0 \mid cs \mid \alpha_1 \cup \alpha_2\ ]\!]\ (A(1)\ [\![\ \alpha_1 \mid cs \mid \alpha_2\ ]\!]\ A(2))$$

Commands

Assignments are directly translated to Java assignments. However, in the case of multiple assignments, if the right-hand side of the assignment depends on any variable present in the left-hand side of the assignment, we must store all the values of the variables involved in the multiple assignment before it actually happens, and then use the new auxiliary variable in the right-hand side of the Java assignment.

$$
\begin{array}{l}
[\![x := e]\!]^{Action} = \texttt{x=}(JExp\ e)\texttt{;} \\
[\![x_1, \ldots, x_n := e_1, \ldots, e_n]\!]^{Action} = \\
\quad \textbf{if}\ (\{x_1, \ldots, x_n\} \cap (FV(e_1) \cup \ldots \cup FV(e_n)) = \emptyset)\ \textbf{then} \\
\quad\quad \texttt{x\_1=}(JExp\ e_1)\texttt{;}\ \ \ldots\texttt{;}\ \texttt{x\_n=}(JExp\ e_n)\texttt{;} \\
\quad \textbf{else} \\
\quad\quad (JType\ (CType\ x_1))\ \texttt{aux\_x\_1 =}\ (JExp\ e_1)\texttt{;} \\
\quad\quad \ldots\texttt{;} \\
\quad\quad (JType\ (CType\ x_n))\ \texttt{aux\_x\_n =}\ (JExp\ e_n)\texttt{;} \\
\quad\quad \texttt{x\_1=}(JExp\ e_1)\texttt{[x\_1,}\ldots\texttt{,x\_n} \setminus \texttt{aux\_x\_1,}\ldots\texttt{,aux\_x\_n]}\texttt{;} \\
\quad\quad \ldots\texttt{;} \\
\quad\quad \texttt{x\_n=}(JExp\ e_n)\texttt{[x\_1,}\ldots\texttt{,x\_n} \setminus \texttt{aux\_x\_1,}\ldots\texttt{,aux\_x\_n]}\texttt{;}
\end{array}
$$

Alternations(**if_fi**) are translated to `if-then-else` blocks. The possible nondeterminism is removed by choosing the first guard that is *true*. If none of the guards is *true*, the action behaves like *Chaos*.

$$[\![\textbf{if } g_1 \rightarrow A_1 \,\square\, \ldots \,\square\, g_n \rightarrow A_n \textbf{ fi}]\!]^{Action} =$$

```
if((JExp g_1)){
    [[A_1]]^Action
} else if (...
} else if((JExp g_n)){
    [[A_n]]^Action
} else { while(true){} }
```

Variable declarations only introduce the declared variables in scope. Implicitly, this is also reflected in the local variables environment.

$$[\![\textbf{var } x_1 : T_1; \ldots; x_n : T_n \bullet Action]\!]^{Action} =$$

```
(JType T_1) x_1; ... ; (JType T_n) x_n;
[[Action]]^Action
```

6.2.5 Transformation of Compound Processes

We now concentrate in the translation of the processes that are defined in terms of other processes. In the first case, we have a single **process name** N. In order to run the process N, we must instantiate it, and then, invoke its `run` method. The visible channels of the process are given as arguments to the process constructor.

$$[\![N]\!]^{Proc} =$$

```
(new CSProcess(){
    public void run() {
        (new N(ExtractChans VisChanEnv)).run();
    }
}).run();
```

If, however, we use parameters in the process invocation, besides the visible channels, we must also use the corresponding Java expressions to each given parameter in the instantiation of the object that represents this process.

$$[\![N(e_1, \ldots, e_n)]\!]^{Proc} =$$

```
(new CSProcess(){
    public void run() {
        (new PName((JExp e_1),...,(JExp e_n),
                   (ExtractChans VisChanEnv))).run();
    }
}).run();
```

The function *ExtractChans* returns a list of the channel names in the domain of a given *ChanUseEnv* as defined below.

$$
\begin{array}{l}
\mathit{ExtractChans} : \mathit{ChanUseEnv} \nrightarrow \mathsf{JCode} \\
\mathit{ExtractChans}\ \emptyset = \epsilon \\
\mathit{ExtractChans}\ \{c \mapsto t\} = \mathtt{c} \\
\mathit{ExtractChans}\ \{c \mapsto t\} \cup \upsilon = \mathtt{c,}(\mathit{ExtractChans}\ \upsilon),\ \upsilon \neq \emptyset
\end{array}
$$

As for actions, the **invocation of (parametrised) processes** is translated to a new inner class. It runs the parametrised process instantiated with the given arguments. The name of the new inner class is also indexed by a fresh *index* to avoid name clashes, as presented below.

$$
\begin{array}{l}
[\![(\mathit{Decl} \bullet \mathit{Proc})(e_1, \ldots, e_n)]\!]^{\mathit{Proc}} = \\
\quad \mathit{DeclareProcessClass}\ \mathit{Decl}\ \mathit{Proc}\ \mathit{index} \\
\quad \mathtt{I\_}\mathit{index}\ \mathtt{i\_}\mathit{index}\mathtt{\_}\mathit{index}\ \mathtt{=} \\
\quad\quad \mathtt{new\ I\_}\mathit{index}\mathtt{(}(\mathit{JExp}\ e_1)\mathtt{,} \ldots \mathtt{,}(\mathit{JExp}\ e_n)\mathtt{,} \\
\quad\quad\quad\quad\quad (\mathit{JList}\ (\mathit{ListFirst}\ \mathit{LocalVarEnv}))\mathtt{);} \\
\quad \mathtt{i\_}\mathit{index}\mathtt{\_}\mathit{index}\mathtt{.run();}
\end{array}
$$

The function *DeclareProcessClass* declares a class representing the parametrised process. The only attributes of this new class are the process parameters, which are given as a declaration list to the function. These attributes are initialised in the class constructor with the values given as arguments to the constructor. The body of method `run` is the Java code obtained with the translation of the process given as argument to the function.

$$
\begin{array}{l}
\mathit{DeclareProcessClass} : \mathsf{Decl} \nrightarrow \mathsf{Proc} \nrightarrow \mathbb{N} \nrightarrow \mathsf{JCode} \\
\mathit{DeclareProcessClass}\ \mathit{Decl}\ \mathit{Proc}\ \mathit{index} = \\
\quad \mathtt{class\ I\_}\mathit{index}\ \mathtt{implements\ CSProcess\ \{} \\
\quad\quad \mathit{ParamsDecl}\ \mathit{Decl} \\
\quad\quad \mathtt{public\ I\_}\mathit{index}\mathtt{(}\mathit{ParamsArgs}\ \mathit{Decl}\mathtt{)\{} \\
\quad\quad\quad \mathit{MultiAssign}\ (\mathit{ParamsDecl}\ \mathit{Decl})\ (\mathit{ParamsArgs}\ \mathit{Decl}) \\
\quad\quad \mathtt{\}} \\
\quad\quad \mathtt{public\ void\ run\ (\ )\{} \\
\quad\quad\quad [\![\mathit{Proc}]\!]^{\mathit{Proc}} \\
\quad\quad \mathtt{\}} \\
\quad \mathtt{\}}
\end{array}
$$

The **sequential composition** is also easily translated to the execution of the

second process after the execution of the first one finishes.

$$[\![Proc_1; \ldots; Proc_n]\!]^{Proc} =$$

```
(new CSProcess(){
    public void run() { [[ Proc_1 ]]^Proc }
 }).run();
...
(new CSProcess(){
    public void run() { [[ Proc_n ]]^Proc }
 }).run();
```

External choice has a similar solution to that presented for actions. The idea is to create an alternative in which all the initial channels of both processes, that are not hidden, take part. However, all auxiliary functions used in the previous definitions take actions into account. All we have to do is create similar functions that take processes into account.

As the **internal choice** for actions, the internal choice for processes randomly chooses a process, and then, starts to behave as such.

$$[\![Proc_1 \sqcap \ldots \sqcap Proc_n]\!]^{Proc} =$$

```
int choosen = RandomGenerator.generateNumber(0,n);
switch(choosen) {
    case 0:
        { [[ Proc_1 ]]^Proc }
        break;
    ...
    case n:
        { [[ Proc_n ]]^Proc }
        break;
}
```

The **renaming** operation is translated by a simple substitution in the translated Java code.

$$[\![Proc[x_1, \ldots, x_n := y_1, \ldots, y_n]]\!]^{Proc} =$$
$$[\![Proc]\!]^{Proc} [\texttt{x\_1}, \ldots, \texttt{x\_n} \setminus \texttt{y\_1}, \ldots, \texttt{y\_n}]$$

As for actions, the **iterated operators** are translated using `for` loops. The same restrictions apply for processes. The first iterated operator on processes is the **sequential composition** $\fatsemi$. As for actions, we use an auxiliary function. However, as we are translating processes, this auxiliary function creates a vector of processes.

Then, we execute each process within this vector. For a fresh *index*, the translation can be defined as follows.

$$[\![⨾_9\ x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Proc]\!]^{Proc} =$$

```
    (new CSProcess(){
        public void run() {
            InstProcesses procVec_index (x_1 : T_1; ...; x_n : T_n) Proc index
            for (int i = 0; i<procVec_index.size(); i++){
                ((CSProcess)procVec_index.get(i)).run();
            }
        }
    }).run();
```

Similar to the auxiliary function used for action, the function *InstProcesses* instantiates a vector containing each of the processes obtained by considering each of the possible values of the indexing variables. First, we declare an inner class representing a parametrised process using function *InstProcesses*. The parameters of this process are the indexing variables. Then, it declares a nested loop. In each iteration of the loop, it instantiates a process, using the declared class constructor, and stores it a `Vector` variable, which name is given as argument to the function. The current values of the indexing variables are given as arguments to the constructor.

$$InstProcesses : \mathsf{N} \pfun \mathsf{Decl} \pfun \mathsf{Proc}\mathbb{N} \pfun \mathsf{JCode}$$

```
InstProcesses procVecName (x_1 : T_1; ...; x_n : T_n) Proc index =
    Vector procVecName = new Vector();
    DeclareProcessClass (x_1 : T_1; ...; x_n : T_n) Proc index
    for ((JType T_1) x_1 = (Min T_1);
         x_1.compareTo((Max T_1))<=0; (Inc x_1 : T_1)){
        ...
        for ((JType T_n) x_n = (Min T_n);
             x_n.compareTo((Max T_n))<=0; (Inc x_n : T_n)){
            procVecName.add(new I_index(x_1,...,x_n);
        }
        ...
    }
```

As for actions, the **iterated external choice** must be expanded before being translated. The **iterated internal choice** chooses a value for each indexing variable, and then, runs the process with the randomly chosen values for the indexing

variables in scope.

$$[\![\sqcap x_1 : T_1; \ldots; x_n : T_n \bullet Proc]\!]^{Proc} =$$

```
(new CSProcess(){
    public void run() {
        ChooseIndexVars (x_1 : T_1; ...; x_n : T_n)
        DeclareProcessClass (x_1 : T_1; ...; x_n : T_n) Proc index
        (new I_index(x_1,...,x_n)).run();
    }
}).run();
```

Again, we consider *index* as fresh index.

The **iterated parallelism** of processes are simpler than the iterated parallelism of actions. The fact that, for processes, we do not need to deal with partitions of variables in scope is the reason for that. Its translation uses the function *InstProcesses* to instantiate the vector containing each of the processes obtained by considering each of the possible values of the indexing variables. Then, it transforms this `Vector` of processes in an array of processes, which is given to the constructor of a `Parallel` process. Finally, we run the `Parallel` process.

$$[\![\; \| \; x_1 : T_1; \ldots; x_n : T_n \; [\![CSExp]\!] \bullet Proc]\!]^{Proc} =$$

```
(new CSProcess(){
    public void run() {
        InstProcesses procVec_index (x_1 : T_1; ...; x_n : T_n) Proc index
        CSProcess[] processes_index =
            new CSProcess[procVec_index.size()];
        for (int i = 0; i < procVec_index.size(); i++){
            processes_index[i] =
                (CSProcess)procVec_index.get(i);
        }
        (new Parallel(processes_index)).run();
    }
}).run();
```

The **indexed operator** translation needs the concept of array of channels, which is introduced in Section 6.3. For this reason, its presentation is left to Section 6.4.

6.2.6 Z Paragraphs

Free Types and Abbreviations.

Our strategy takes into account only free types or abbreviations defined in terms of, at most, one other type. Furthermore, two different types should not extend the

same type, since, in our strategy, this would lead to multiple inheritance, which is not allowed in Java.

One new class is created for each declared type. They extend the class defined by function *DeclareTypeClass* presented below.

DeclareTypeClass : Program $\rightarrow\!\!\!\!\!\!+$ N $\rightarrow\!\!\!\!\!\!+$ JCode
DeclareTypeClass prog proj =

```
    package proj.typing;
    public abstract class Type {
        private int value;
        DeclTypeConstants TypesEnv 0
        public int getValue() { return this.value; }
        protected void setValue(int value) {
        this.value = value;
        }
        public boolean equals(Type other) {
            boolean equals = false;
            if (other != null) {
                boolean sameClass =
                    this.getClass().equals(other.getClass());
                boolean sameValue =
                    (this.getValue()==other.getValue());
                equals = (sameClass && sameValue);
            }
            return equals;
        }
        public int compareTo(Type other) {
            int compare = -1;
            if (other != null) {
                if (this.getValue() == other.getValue()) {
                    compare = 0;
                } else if (this.getValue() > other.getValue()) {
                    compare = 1;
                }
            }
            return compare;
        }
    }
```

Each type object contains one `int` value. This value is declared `private`, but it can be accessed using the `getValue` method. For each type T used within the system, the class declares a constant identifier for this type. The function

DeclTypeConstants declares these constants and two special static constants used to declare the range of the types identifiers. The motivation for the existence of these constants is related to the use of generic channels (See Section 6.3).

$$
\begin{array}{l}
\mathit{DeclareTypeConstants} : \mathsf{seq}\,\mathsf{N} \pfun \mathbb{N} \pfun \mathsf{JCode} \\
\mathit{DeclareTypeConstants}\ [\,]\ n = \\
\quad \texttt{public static final int MIN\_TYPE\_ID = 0;} \\
\quad \texttt{public static final int MAX\_TYPE\_ID =}\ n\texttt{;} \\
\mathit{DeclareTypeConstants}\ (T : TS)\ n = \\
\quad \texttt{public static final int}\ (\mathit{Capitals}\ (\mathit{JType}\ T))\ \texttt{=}\ n\texttt{;} \\
\quad \mathit{DeclareTypeConstants}\ TS\ (n+1)
\end{array}
$$

The method `equals` from class `java.lang.Object` is overwritten within the `Type` class: two type objects are equal only if they are of the same class and have the same value within them. The method `compareTo` is used in the nested loops created by the translation of iterated operators. It returns the value 0, if the argument `Type` is equal to this `Type`; a value less than 0, if this `Type` is numerically less than the `Type` argument; and a value greater than 0, if this `Type` is numerically greater than the `Type` argument (signed comparison).

To translate a free type or an abbreviation, we have to check if it was extended by a previously defined type or not. If this is not the case, we have that, for each possible value of the given free type, we declare a static constant that can be used throughout the classes in the project.

When the free type is extended by a previous defined type, it is declared an specialisation (`extends`) of the extension type; otherwise, it extends the class `Type`. For instance, consider the following types.

$$
\begin{array}{l}
T_A == T_B \cup \{3, 4\} \\
T_B ::= 0 \mid 1 \mid 2
\end{array}
$$

The translation of these two types creates two classes: the class `T_A`, which represents the type T_A, extends the class `Type`; and the class `T_B`, which represents the type T_B, extends the class `T_A`.

We consider that a type inheritance environment $\mathit{tInheritance} : N \pfun N$ is available throughout the translation. It maps type classes names to their super classes. In our example above, $T_B \mapsto T_A$ would be an element of this environment.

The translation of a free type declaration verifies if a declared type *FTName* is mapped to any type in the type environment. If it is not mapped to any declared type, this class is declared as an extension of class `Type`. Besides, a constant is declared for each possible value of this type, and the range values of these constants are also declared. However, if *FTName* is mapped to any other declared type, its corresponding class is declared as an extension of the class corresponding to the

type it is mapped to. Besides, as the constants of typing subclasses are already declared in their super classes, we need only to declare its range constants.

$$
\begin{array}{l}
[\![\_]\!]^{Types} : \mathsf{Program} \pfun \mathsf{N} \pfun \mathsf{JCode} \\
[\![\epsilon]\!]^{Types}\ proj = \epsilon \\
[\![FTName ::= V_0 \mid \ldots \mid V_n\ \ Types]\!]^{Types}\ proj = \\
\quad \textbf{let}\ superClass = \\
\qquad\quad \textbf{if}\ (FTName \notin (\mathrm{dom}\ \ tInheritance)) \\
\qquad\quad \textbf{then}\ \texttt{Type} \\
\qquad\quad \textbf{else}\ tInheritance(FTName), \\
\qquad constants = \\
\qquad\quad \textbf{if}\ (FTName \notin (\mathrm{dom}\ \ tInheritance)) \\
\qquad\quad \textbf{then}\ (DeclFTConstants\ ([V_0, \ldots, V_n])\ 0\ FTName) \\
\qquad\quad \textbf{else}\ (DeclFTRange\ 0\ n\ FTName)\ \textbf{in} \\
\qquad \texttt{package}\ proj\texttt{.typing;} \\
\qquad \texttt{public class FTName extends}\ superClass\ \texttt{\{} \\
\qquad\quad constants \\
\qquad\quad \texttt{protected FTName()\{\}} \\
\qquad\quad \texttt{public FTName(int value) \{ this.setValue(value); \}} \\
\qquad \texttt{\}} \\
\qquad [\![Types]\!]^{Types}\ proj
\end{array}
$$

The constructors of possible existing subclasses of *FTName*, by default, invoke the *FTName* empty constructor. This constructor would be implicitly declared if no other constructor were declared. This, however, is not the case. For this reason, the empty constructor must be declared. We declare it as `protected` to limit the access to such constructor, since we intend it to be used only by subclasses constructors.

The function *DeclFTConstants* declares a static constant for each possible value within the given type.

$$
\begin{array}{l}
DeclFTConstants : \mathrm{seq}\,\mathsf{N} \pfun \mathbb{N} \pfun \mathsf{N} \pfun \mathsf{JCode} \\
DeclFTConstants\ (V : []\ )\ n\ name = \\
\quad \texttt{public static final int V = n;} \\
\quad DeclFTRange\ 0\ n\ name \\
DeclFTConstants\ (V : VS)\ n\ name = \\
\quad \texttt{public static final int V = n;} \\
\quad DeclFTConstants\ VS\ (n+1)\ name
\end{array}
$$

Finally, it declares two static constants to represent the range of the values of this

type.

$$
\begin{array}{l}
DeclFTRange : \mathbb{N} \pfun \mathbb{N} \pfun \mathsf{JCode} \\
DeclFTRange\ min\ max\ name = \\
\quad \texttt{public static final int MIN\_name = min;} \\
\quad \texttt{public static final int MAX\_name = max;}
\end{array}
$$

In the case of abbreviations $\mathsf{N} == \mathsf{Expression}$, we have two cases: either it extends a declared type (i.e. $T_A == T_B \cup S$) or it restricts a declared type(i.e. $T_A == T_B \setminus S$). In both cases, the type that expands is declared as a superclass of the class representing the expanded type. We consider two possible cases: the first case is that of a type $TName_{exp}$ being declared as an expansion of a type $TName$. We consider that the expansion of the declaration of the type $TName$ is $V_0 \mid \ldots \mid V_n$. As in the translation of a free type declaration, the new class inheritance and constants declared within it depend on whether this type is mapped to any other type in the typing environment or not. However, its constructor checks if the given value is one of the values in the sequence $V_0, \ldots, V_m$.

$$
\begin{array}{l}
[\![TName_{exp} == TName \cup \{ V_{n+1}, \ldots, V_m \}\ Types]\!]^{Types}\ proj = \\
\quad \textbf{let}\ superClass = \\
\qquad \textbf{if}\ (TName_{ext} \notin (\mathrm{dom}\ tInheritance)) \\
\qquad \textbf{then}\ \texttt{Type} \\
\qquad \textbf{else}\ tInheritance(TName_{ext}) \\
\quad constants = \\
\qquad \textbf{if}\ (TName_{ext} \notin (\mathrm{dom}\ tInheritance)) \\
\qquad \textbf{then}\ (DeclFTConstants\ ([V_0, \ldots, V_n])\ 0\ TName_{ext}) \\
\qquad \textbf{else}\ (DeclFTRange\ 0\ n\ TName_{ext})\ \textbf{in} \\
\quad \texttt{package}\ proj\texttt{.typing;} \\
\quad \texttt{public class TName\_exp extends}\ superClass\ \texttt{\{} \\
\qquad constants \\
\qquad \texttt{public TName\_exp()\{\}} \\
\qquad \texttt{public TName\_exp(int value) \{ this.setValue(value); \}} \\
\quad \texttt{\}} \\
\quad [\![Types]\!]^{Types}\ proj
\end{array}
$$

The second case considers a type $TName$ being declared as a restriction of a type $TName_{exp}$. Our strategy considers that if a type T_2 restricts a type T_1, the values of the type T_1 that are removed in the type T_2 are declared in the end of T_1 declaration. For instance, suppose we want to define a type $T_2 ::= T_1 \setminus \{C, D\}$. In our strategy, T_1 must be declared before T_2 as $T_1 ::= A \mid B \mid C \mid D$. In the definition below, we consider that the expansion of the declaration of the type $TName_{exp}$ is $V_0 \mid \ldots \mid V_n \mid V_{n+1} \mid \ldots \mid V_n$.

The created class is declared as an extension of the class `TName_exp`. As all the attributes of class `TName_exp` are visible from class `TName`, we do not declare a new constant for each value of *TName*. However, since type *TName* has a different range of values, the range constants are redeclared.

$$
\begin{array}{l}
[\![TName == TName_{exp} \setminus \{ V_{n+1}, \ldots, V_m \}\ Types]\!]^{Types}\ proj = \\
\quad \texttt{package}\ proj\texttt{.typing;} \\
\quad \texttt{public class TName extends TName\_exp \{} \\
\quad\quad DeclFTRange\ 0\ n\ TName \\
\quad\quad \texttt{public TName()\{\}} \\
\quad\quad \texttt{public TName(int value) \{ this.setValue(value); \}} \\
\quad \texttt{\}} \\
\quad [\![Types]\!]^{Types}\ proj
\end{array}
$$

Axiomatic Definitions.

The class `proj.axiomaticDefinitions.AxiomaticDefinitions` encapsulates all axiomatic definitions as static methods within it. The function *DeclareAxDefClass* declares this class and is defined below.

$$
\begin{array}{l}
DeclareAxDefClass : \mathsf{Program} \pfun \mathsf{N} \pfun \mathsf{JCode} \\
DeclareAxDefClass\ AxDefs\ proj = \\
\quad \texttt{package}\ proj\texttt{.axiomaticDefinitions;} \\
\quad \texttt{import}\ proj\texttt{.typing.*;} \\
\quad \texttt{public class AxiomaticDefinitions \{}\ [\![AxDefs]\!]^{AxDefs}\ \texttt{\}}
\end{array}
$$

For each axiomatic definition, a different static method is created within the class `AxiomaticDefinitions`. We present below the definition of the function $[\![\_]\!]^{AxDefs}$.

$$
\begin{array}{l}
[\![\_]\!]^{AxDefs} : \mathsf{Program} \pfun \mathsf{JCode} \\
[\![\epsilon]\!]^{AxDefs} = \epsilon \\
[\![v : T \mid v = e_1\ AxDefs]\!]^{AxDefs} = \\
\quad \texttt{public static}\ (JType\ T)\ v\texttt{() \{ return}\ (JExp\ e_1)\texttt{; \}} \\
\quad [\![AxDefs]\!]^{AxDefs}
\end{array}
$$

The return expression of the new method is a Java expression that implements the expression e_1.

6.2.7 Utilities Classes

A package `proj.util` for utilities classes is also generated in our translation strategy. When created, it contains only one class `RandomGenerator` used to generate

random numbers. The Java API contains some random utilities. However, we need a method that, given two integers, it returns a random number between these two numbers. Such a method is not yet provided, and for this reason, we declare this utility class.

$DeclareRandomGenerator : \mathsf{N} \pfun \mathsf{JCode}$
$DeclareRandomGenerator\ proj =$

```
    package proj.util;
    public class RandomGenerator {
        public static int generateNumber(int min, int max){
            int randomNumber = min;
            Long randomLong =
            new Long(Math.round(Math.random()*100));
            int randomInt = randomLong.intValue();
            int numberOfIntervals = max - min + 1;
            int interval = 100 / numberOfIntervals;
            boolean ready = false;
            for (int i=1; i<=numberOfIntervals && !ready ; i++){
                if (randomInt <= i*interval) {
                    randomNumber = min+i-1;
                    ready=true;
                }
            }
            return randomNumber;
        }
    }
```

6.2.8 Circus Programs

Our translation strategy is summarised by a translation function $[\![\_]\!]^{Program}$. Besides the *Circus* program, this function also receives a project name, which is used to declare the package for each new class.

As discussed in Section 6.2, our strategy assumes that the paragraphs within a *Circus* program are grouped in three types and declared in this order: Z paragraphs,

channel declarations, and process declarations.

$$
\begin{array}{l}
[\![\_]\!]^{Program} : \mathsf{Program} \pfun \mathsf{N} \pfun \mathsf{JCode} \\
[\![Types\ AxDefs\ ChanDecls\ ProcDecls]\!]^{Program}\ proj = \\
\quad DeclareTypeClass\ (Types\ AxDefs\ ChanDecls\ ProcDecls)\ proj \\
\quad DeclareRandomGenerator\ proj \\
\quad [\![Types]\!]^{Types}\ proj \\
\quad DeclareAxDefClass\ proj\ AxDefs \\
\quad [\![ProcDecls]\!]^{ProcDecls}\ proj
\end{array}
$$

The function $[\![\_]\!]^{Program}$ declares the class `RandomGenerator`. In the sequel, it declares the Java classes representing each free type and abbreviation declared in the program, and the Java class that encapsulates all the axiomatic definitions. Finally, it translates all the declared processes.

6.2.9 Running the program.

The code generated by $[\![\_]\!]^{Program}$ is a sequence of class definitions, that implement all the processes of the *Circus* program. The function $[\![\_]\!]^{Run}$ can be used to generate a class with a `main` method, which can be used to execute a given process. This function is applied to a *Circus* process, and a project name. It creates a Java class named `Main`, which is created in the package *proj*. After the package declaration, the class imports the package `java.util`, that contains some Java utilities; and the package `jcsp.lang`, that contains all JCSP basic classes. Then, it imports all the existing packages within the project. The `main` method, which is invoked when running the class `Main`, has the translation of the given process as its body.

$$
\begin{array}{l}
[\![\_]\!]^{Run} : \mathsf{Proc} \pfun \mathsf{N} \pfun \mathsf{JCode} \\
[\![Proc]\!]^{Run}\ proj = \\
\quad \texttt{package}\ proj\texttt{;} \\
\quad \texttt{import java.util.*;} \\
\quad \texttt{import jcsp.lang.*;} \\
\quad \texttt{import}\ proj\texttt{.axiomaticDefinitions.*;} \\
\quad \texttt{import}\ proj\texttt{.processes.*;} \\
\quad \texttt{import}\ proj\texttt{.typing.*;} \\
\quad \texttt{import}\ proj\texttt{.util.*;} \\
\quad \texttt{public class Main \{} \\
\quad\quad \texttt{public static void main(String args[]) \{}\ [\![Proc]\!]^{Proc}\ \texttt{\}} \\
\quad \texttt{\}}
\end{array}
$$

6.3 Other types of communications

In this section, we extend the types of communications considered in our strategy; we deal with generic channels ($\mathsf{N}[\mathsf{N}]$) and communication events of the form $\mathsf{N.Expression}$. We consider that the uses of such channels first declare possible synchronisation values, and finally the input/output value. By way of illustration, consider the following channel declaration.

$$\textbf{channel}[T]\, c : \mathbb{N} \times T$$

This declaration declares family of channels c, which, given a type T synchronises in a natural number and communicates a value of type T. As in Section 6.2, our strategy still constrains the channels to have only one input/output value. Multiple inputs/outputs must be encapsulated in Java objects.

Two other important constraints are: firstly, channels may synchronise only values of finite types; further, all the types used within the system must be finite. These constraints arise from the fact that our strategy uses possible multi-dimensional arrays of channels for representing such generic channels and synchronisation events; infinite types would lead to infinite arrays.

For the purpose of characterising the kind of communication contemplated by our strategy, our definition of Comm can be extended as follows.

$$\begin{aligned}
\mathsf{Comm} &::= \mathsf{Chann?N} \mid \mathsf{Chann!Expression} \mid \mathsf{Chann} \\
\mathsf{Chann} &::= \mathsf{N}\ \mathsf{Typing}^*\ \mathsf{Sync}^* \\
\mathsf{Typing} &::= [\mathsf{Expression}] \\
\mathsf{Sync} &::= .\mathsf{Expression}
\end{aligned}$$

A very important change in this extension is the use of a new channel environment $SyncCommEnv : ChanSyncEnv$. It maps each channel used within the system to a value of type SC, which indicates if the channel is a communication channel (C), or a synchronisation channel (S).

$$\begin{aligned}
&SC ::= S \mid C \\
&ChanSyncEnv == \mathsf{N} \rightarrow SC
\end{aligned}$$

The previous defined function $[\![\_]\!]^{Program}$(pg. 155) can then be substituted by its extended version defined below.

$$\begin{aligned}
&[\![_]\!]^{Program_{ext}} : \mathsf{Program} \nrightarrow \mathsf{N} \nrightarrow \mathsf{JCode} \\
&[\![Types\ AxDefs\ ChanDecls\ ProcDecls]\!]^{Program}\ proj = \\
&\quad DeclareTypeClass\ (Types\ AxDefs\ ChanDecls\ ProcDecls)\ proj \\
&\quad DeclareRandomGenerator\ proj \\
&\quad [\![Types]\!]^{Types}\ proj \\
&\quad DeclareAxDefClass\ proj\ AxDefs \\
&\quad [\![ProcDecls]\!]^{ProcDecls_{ext}}\ proj
\end{aligned}$$

The function $[\![\_]\!]^{ProcDecls_{ext}}$ is an extended version of function $[\![\_]\!]^{ProcDecls}$ (pg. 122). Its definition is similar to its original version, but it uses the extended version of function $[\![\_]\!]^{ParProc}$ (pg. 123) defined below.

$$
\begin{array}{l}
[\![\_]\!]^{ParProc_{ext}} : \mathsf{ParProc} \pfun \mathsf{JCode} \\
[\![Decl \bullet Proc]\!]^{ParProc_{ext}} = \\
\quad ParamsDecl\ Decl \\
\quad VisibleCDecl_{ext}\ VisChanEnv\ ChanTypeEnv\ SyncCommEnv \\
\quad HiddenCDecl_{ext}\ HidChanEnv\ ChanTypeEnv\ SyncCommEnv \\
\quad \texttt{public } P\texttt{(}ParamsArgs\ Decl\texttt{,} \\
\qquad\qquad (VisibleCArgs_{ext}\ VisChanEnv\ ChanTypeEnv \\
\qquad\qquad\qquad SyncCommEnv)\texttt{)\{} \\
\qquad MultiAssign\ (ParamsDecl\ Decl)\ (ParamsArgs\ Decl) \\
\qquad MultiAssign\ (VisibleCDecl_{ext}\ VisChanEnv\ ChanTypeEnv \\
\qquad\qquad\qquad SyncCommEnv) \\
\qquad\qquad (VisibleCArgs_{ext}\ VisChanEnv\ ChanTypeEnv \\
\qquad\qquad\qquad SyncCommEnv) \\
\qquad HiddenCCreation_{ext}\ HidChanEnv\ ChanTypeEnv \\
\qquad\qquad\qquad SyncCommEnv\ TypesEnv \\
\quad \texttt{\}} \\
\quad \texttt{public void run()\{ } [\![\ Proc\ ]\!]^{Proc_{ext}} \texttt{ \}}
\end{array}
$$

The following sections describe the steps of the translation that are extended.

6.3.1 Declaration of visible channels.

The function $VisibleCDecl_{ext}$ extends the function $VisibleCDecl$. Its basic difference to the original definition is the possibility of channel array declaration.

$$
\begin{array}{l}
VisibleCDecl_{ext} : ChanUseEnv \pfun ChanEnv \pfun ChanSyncEnv \pfun \mathsf{JCode} \\
VisibleCDecl_{ext}\ \emptyset\ \delta\ \zeta = \epsilon \\
VisibleCDecl_{ext}\ (\{c \mapsto t\} \cup \upsilon)\ \delta\ \zeta = \\
\quad \texttt{private } (TypeChan\ t) \\
\qquad\qquad (ArrayDimension\ (fst\,(\delta\ c))\ (snd\,(\delta\ c))\ (\zeta\ c)\ 0)\ \texttt{c;} \\
\quad VisibleCDecl_{ext}\ \upsilon\ \delta\ \zeta
\end{array}
$$

The visible channels may be declared as an array of channels. The auxiliary function *ArrayDimension* defines this dimension of possible array of channels.

If the channel declaration just gives a channel name, but not type, the channel types environment maps this channel name to a pair which has the empty list as its first element (*gen*) and the list [*Sync*] as its the second element (*types*). In this case, the channel is only a synchronising event, and it is not declared as an array

of channels. Otherwise, it may be an array of channels ($Dimension = 0$). We store in *genExp* the number of types used within the channel declaration (*types*) that are declared as generic (*gen*). For each such channel, two dimensions are added to the final dimension of this array. Any further type $((\#types) - genExp)$ adds one dimension to the array. However, if the channel is a communication channel ($sc = C$), one dimension is removed from the final array dimension since the last type is a communication and not a synchronisation. Furthermore, an extra argument *gap* can be used to decrease the final array dimension.

In the following definition, we use the notation $(\texttt{code})^n$ to represent n repetitions of `code`; if $n \leq 0$, $(\texttt{code})^n$ is the empty string ϵ.

$$
\begin{array}{l}
ArrayDimension : \mathrm{seq}\,\mathsf{Expression} \pfun \mathrm{seq}\,\mathsf{Expression} \pfun SC \pfun \mathbb{N} \pfun \mathsf{JCode} \\
ArrayDimension\ gen\ types\ sc\ gap = \\
\quad \textbf{let}\ genExp = (Count\ types\ gen)\ \textbf{in} \\
\quad\quad \textbf{let}\ Dimension = \\
\quad\quad\quad\quad \textbf{if}\ (types = [Sync])\ \textbf{then}\ 0 \\
\quad\quad\quad\quad \textbf{else}\ if\ (sc = C)\ \textbf{then} \\
\quad\quad\quad\quad\quad\quad (genExp * 2) + ((\#types) - genExp) - 1 \\
\quad\quad\quad\quad \textbf{else}\ (genExp * 2) + ((\#types) - genExp)\ \textbf{in} \\
\quad\quad\quad (\texttt{[]})^{Dimension - gap}
\end{array}
$$

The expression *genExp* represents the number of types used in the channel declaration that are declared as generic. It is defined using the function *Count*, which returns the number of expressions in the first list that are present in the second list, and can be recursively defined as follows.

$$
\begin{array}{l}
Count : \mathrm{seq}\,\mathsf{Expression} \pfun \mathrm{seq}\,\mathsf{Expression} \pfun \mathbb{N} \\
Count\ ts\ [\,] = Count\ [\,]\ ns = 0 \\
Count\ [t](n : ns) = \textbf{if}\ (t = n)\ \textbf{then}\ 1\ \textbf{else}\ (Count\ [t]\ ns) \\
Count\ (t : ts)\ ns = (Count\ [t]\ ns) + (Count\ ts\ ns)
\end{array}
$$

6.3.2 Declaration of Hidden Channels.

As for the used channels, the function $HiddenCDecl_{ext}$ extends the function *HiddenCDecl*. The definition of the dimension of possible arrays of channels is the same as for the visible channels. However, we declare the channels as `Any2OneChannel` channels, since they are instantiated within this process.

$$
\begin{array}{l}
HiddenCDecl_{ext} : ChanUseEnv \pfun ChanEnv \pfun ChanSyncEnv \pfun \mathsf{JCode} \\
HiddenCDecl_{ext}\ \emptyset\ \delta\ \zeta = \epsilon \\
HiddenCDecl_{ext}\ (\{c \mapsto t\} \cup \upsilon)\ \delta\ \zeta = \\
\quad \texttt{private Any2OneChannel} \\
\quad\quad (ArrayDimension\ (fst\,(\delta\ c))\ (snd\,(\delta\ c))\ (\zeta\ c)\ 0)\ \texttt{c;} \\
\quad HiddenCDecl_{ext}\ \upsilon\ \delta\ \zeta
\end{array}
$$

6.3.3 Channel arguments in the constructor.

The function *VisibleCArgs* is also extended. Its extension is very similar to the original one. However, it also takes into account the existence of possible channel arrays, using the auxiliary function *ArrayDimension*.

$$
\begin{array}{l}
\mathit{VisibleCArgs}_{ext} : \mathit{ChanUseEnv} \nrightarrow \mathit{ChanEnv} \nrightarrow \mathit{ChanSyncEnv} \nrightarrow \mathsf{JCode} \\
\mathit{VisibleCArgs}_{ext}\ \emptyset\ \delta\ \zeta = \epsilon \\
\mathit{VisibleCArgs}_{ext}\ (\{c \mapsto t\} \cup \upsilon)\ \delta\ \zeta = \\
\quad (\mathit{TypeChan}\ t)(\mathit{ArrayDimension}\ (\mathit{fst}\,(\delta\ c))\ (\mathit{snd}\,(\delta\ c))\ (\zeta\ c)\ 0)\ \texttt{newc,} \\
\quad \mathit{VisibleCArgs}_{ext}\ \upsilon\ \delta\ \zeta
\end{array}
$$

6.3.4 Instantiation of hidden channels.

If the hidden channel is not declared as an array, the function *ArrayDimension* returns the empty string ϵ. In this case, the channel is instantiated as a `Any2OneChannel` channel. However, if the function *ArrayDimension* does not return the empty string, we use the auxiliary function *InstArray* to instantiate the channel as an array of channels.

$$
\begin{array}{l}
\mathit{HiddenCCreation}_{ext} : \mathit{ChanUseEnv} \nrightarrow \mathit{ChanEnv} \nrightarrow \mathit{ChanSyncEnv} \nrightarrow \\
\qquad\qquad\qquad \mathsf{seq}\,\mathsf{Expression} \nrightarrow \mathsf{JCode} \\
\mathit{HiddenCCreation}_{ext}\ \emptyset\ \delta\ \zeta\ \mathit{types} = \epsilon \\
\mathit{HiddenCCreation}_{ext}\ (\{c \mapsto t\} \cup \upsilon)\ \delta\ \zeta\ \mathit{types} = \\
\quad \textbf{let}\ \mathit{brackets} = (\mathit{ArrayDimension}\ (\mathit{fst}\,(\delta\ c))\ (\mathit{snd}\,(\delta\ c))\ (\zeta\ c)\ 0)\ \textbf{in} \\
\qquad \textbf{if}\ (\mathit{brackets} = \epsilon)\ \textbf{then}\ \texttt{this.c = new Any2OneChannel();} \\
\qquad \textbf{else}\ \texttt{this.}c\ \texttt{=}\ (\mathit{InstArray}\ (\mathit{fst}\,(\delta\ c))\ (\mathit{snd}\,(\delta\ c))\ (\zeta\ c)\ \mathit{types})\texttt{;} \\
\qquad \mathit{HiddenCCreation}_{ext}\ \upsilon\ \delta\ \zeta\ \mathit{types}
\end{array}
$$

The function *InstArray* instantiates an array of channels. It uses the auxiliary function *ArrayDimension* to determine the dimension of the array that is being instantiated. If the first type of the channel declaration (*head types*) is a generic type, we have that it is in the sequence of generic types *gen* used in the channel declaration. For this reason, we know that $\mathit{Count}\ [\mathit{head\ types}]\ \mathit{gen} > 0$. In this case, we instantiate an array of channel with a dimension determined by function *ArrayDimension*. This instantiation uses an auxiliary function *GenericInst*, which declare as many arrays as the number of types used within the system. However, if we do not have a generic type, and it is the last (or maybe the only) type in the channel declaration, we use the function *BaseCase* to declare either a channel instantiation, or an array of channels creation. Finally, if neither of the previous conditions hold, we instantiate an array of channels with dimension defined by the function *ArrayDimension*. In this case, the function *InstArray* recurses in order to

consider the remaining types used in the channel declaration.

```
InstArray : N ⇸ seq Expression ⇸ seq Expression ⇸
                SC ⇸ seq Expression ⇸ JCode
InstArray gen types sc typesEnv =
    let brackets = (ArrayDimension gen types sc 0) in
        if (Count [head types] gen > 0) then
            new Any2OneChannel brackets{
                GenericInst gen types sc typesEnv typesEnv
            }
        else if (#types = 1) then
            BaseCase (head types) sc
        else new Any2OneChannel brackets{
                InstArray gen (tail types) sc typesEnv
            }
```

The base case instantiates a single channel, if the channel is a communication channel, and an array of channels, if the channel is a synchronisation channel.

```
BaseCase : Expression ⇸ SC ⇸ JCode
BaseCase T C = new Any2OneChannel()
BaseCase T S = One2OneChannel.create((Max T)-(Min T)+1)
```

The instantiation of a generic channel declares an element for each type used within the system. For each one of these types, it invokes the function *InstArrayType* to instantiate the corresponding array for that type.

```
GenericInst : seq Expression ⇸ seq Expression ⇸ SC ⇸
                seq Expression ⇸ seq Expression ⇸ JCode
GenericInst gen types sc typesEnv [T] =
    InstArrayType gen types sc typesEnv
GenericInst gen types sc typesEnv (T : TS) =
    InstArrayType gen types sc typesEnv,
    GenericInst gen types sc typesEnv TS
```

The function *InstArrayType* verifies if this is the last type in the declaration of the channel. In this case, it invokes the *BaseCase* function to instantiate the channels for the given type. However, if there are further types in the channel declaration, it declares an array with one dimension less then the current array dimension, and then invokes the function *TypeInst* to make a instantiation for each element of the

given type.

$$
\begin{array}{l}
InstArrayType : \mathrm{seq}\,\mathsf{Expression} \pfun \mathrm{seq}\,\mathsf{Expression} \pfun SC \pfun \\
\qquad\qquad \mathrm{seq}\,\mathsf{Expression} \pfun \mathsf{JCode} \\
InstArrayType\ gen\ ([T])\ sc\ typesEnv = BaseCase\ T\ sc \\
InstArrayType\ gen\ types\ sc\ typesEnv = \\
\quad \texttt{new One2OneChannel}(ArrayDimension\ gen\ types\ sc\ 1)\texttt{\{} \\
\quad\quad TypeInst\ gen\ types\ sc\ typesEnv \\
\quad\quad\quad ((Max\ (head\ types)) - (Min\ (head\ types)) + 1) \\
\quad \texttt{\}}
\end{array}
$$

The function *TypeInst* invokes the function *InstArray* for the other types of the channel declaration for each element in the current type.

$$
\begin{array}{l}
TypeInst : \mathrm{seq}\,\mathsf{Expression} \pfun \mathrm{seq}\,\mathsf{Expression} \pfun SC \pfun \\
\qquad\qquad \mathrm{seq}\,\mathsf{Expression} \pfun \mathbb{N} \pfun \mathsf{JCode} \\
TypeInst\ gen\ types\ sc\ types\ 1 = InstArray\ gen\ (tail\ types)\ sc\ typesEnv \\
TypeInst\ gen\ types\ sc\ types\ n = \\
\quad InstArray\ gen\ (tail\ types)\ sc\ typesEnv\,, \\
\quad TypeInst\ gen\ types\ sc\ typesEnv\ (n-1)
\end{array}
$$

6.3.5 Using the channels.

The extended version of function $[\![\_]\!]^{Proc}$ uses the extended versions of functions $[\![\_]\!]^{PPars}$ and $[\![\_]\!]^{Action}$. The function $[\![\_]\!]^{PPars_{ext}}$ differs from the original function because it also uses the extended version of function $[\![\_]\!]^{Action}$. The function $[\![\_]\!]^{Action_{ext}}$ is equal to the original one, except for two *Circus* constructions: communication and external choice.

For communications, the extended version uses an extension of the auxiliary function $[\![\_]\!]^{Comm}$. I

$$
\begin{array}{l}
[\![\_]\!]^{Action_{ext}} : \mathsf{Action} \pfun \mathsf{JCode} \\
[\![Comm \fun Action]\!]^{Action_{ext}} = \\
\quad [\![Comm]\!]^{Comm_{ext}} \\
\quad [\![Action]\!]^{Action_{ext}}
\end{array}
$$

Our strategy still restricts the communication to have only one input or one output value. The definition of function $[\![\_]\!]^{Comm_{ext}}$ is very similar to that of function $[\![\_]\!]^{Comm}$. However, it also takes into account synchronisation channels using the

auxiliary function *SyncC* as follows.

$$
\begin{aligned}
&[\![\_]\!]^{Comm_{ext}} : \mathsf{Comm} \nrightarrow \mathsf{JCode} \\
&[\![c\ [T_0] \ldots [T_n].e_0\ \ldots\ .e_m?x]\!]^{Comm_{ext}} = \\
&\quad \textbf{let}\ commType = last\ (snd\ (ChanTypeEnv\ c))\ \textbf{in} \\
&\qquad \textbf{let}\ syncC = SyncC\ ([T_0] \ldots [T_n].e_0\ \ldots\ .e_m) \\
&\qquad\qquad\qquad\qquad (fst\ (ChanTypeEnv\ c)) \\
&\qquad\qquad\qquad\qquad (snd\ (ChanTypeEnv\ c))\ \textbf{in} \\
&\qquad\quad (JType\ commType)\ \texttt{x = (}JType\ commType\texttt{)c}syncC\texttt{.read();} \\
&[\![c\ [T_0] \ldots [T_n].e_0\ \ldots\ .e_m!x]\!]^{Comm_{ext}} = \\
&\quad \textbf{let}\ syncC = SyncC\ ([T_0] \ldots [T_n].e_0\ \ldots\ .e_m) \\
&\qquad\qquad\qquad\qquad (fst\ (ChanTypeEnv\ c))\ (snd\ (ChanTypeEnv\ c))\ \textbf{in} \\
&\qquad \texttt{c}syncC\texttt{.write((}JExp\ x\texttt{));} \\
&[\![c\ [T_0] \ldots [T_n].e_0\ \ldots\ .e_m]\!]^{Comm_{ext}} = \\
&\quad \textbf{let}\ syncC = SyncC\ ([T_0] \ldots [T_n].e_0\ \ldots\ .e_m) \\
&\qquad\qquad\qquad\qquad (fst\ (ChanTypeEnv\ c))\ (snd\ (ChanTypeEnv\ c))\ \textbf{in} \\
&\qquad \textbf{if}\ (c \in \mathrm{dom}\ VisChanEnv)\ \textbf{then} \\
&\qquad\quad \textbf{if}\ (VisChanEnv\ c = I \vee VisChanEnv\ c = A)\ \textbf{then} \\
&\qquad\qquad \texttt{c}syncC\texttt{.read();} \\
&\qquad\quad \textbf{else}\ \texttt{c}syncC\texttt{.write(null);} \\
&\qquad \textbf{else if}\ (c \in \mathrm{dom}\ HidChanEnv)\ \textbf{then} \\
&\qquad\quad \textbf{if}\ (HidChanEnv\ c = I \vee HidChanEnv\ c = A)\ \textbf{then} \\
&\qquad\qquad \texttt{c}syncC\texttt{.read();} \\
&\qquad\quad \textbf{else}\ \texttt{c}syncC\texttt{.write(null);}
\end{aligned}
$$

The new auxiliary function *SyncC* identifies which channel, in an (multi-dimensional) array, must be used in the communication. The base case is the empty string.

$$
\begin{aligned}
&SyncC : \mathsf{Typing}^*\mathsf{Sync}^* \nrightarrow \mathrm{seq}\,\mathsf{Expression} \nrightarrow \mathrm{seq}\,\mathsf{Expression} \nrightarrow \mathsf{JCode} \\
&SyncC\ \epsilon\ gen\ types = \epsilon
\end{aligned}
$$

If the only synchronisation is a generic type, we have that we must access the element corresponding to the type identifier.

$$SyncC\ [T]\ gen\ types = \texttt{[Type.(}Capitals\ T\texttt{)]}$$

However, if any other synchronisation value is declared, we ignore the instantiation of the generic type.

$$SyncC\ ([T]\ sync)\ gen\ types = SyncC\ sync\ gen\ types$$

Finally, if we have a synchronisation expression, we verify if the type corresponding tho this value is an instantiation of a generic type: if it is, we must take into

account the indexes corresponding to type and to the value; otherwise, we take into account only the index corresponding to the value. Then, we translate the remaining synchronisation values, considering the *tail* of the types sequence.

$$
\begin{array}{l}
\mathit{SyncC}\ (.e\ \mathit{sync})\ \mathit{gen}\ \mathit{types} = \\
\quad \textbf{if}\ (\mathit{Count}\ (\mathit{head}\ \mathit{types})\ \mathit{gen} > 0)\ \textbf{then} \\
\quad\quad \texttt{[Type.(}\mathit{Capitals}\ (\mathit{JType}\ (\mathit{head}\ \mathit{types}))\texttt{)][(}\mathit{JExp}\ e\texttt{)]} \\
\quad \textbf{else}\ \texttt{[(}\mathit{JExp}\ e\texttt{)]} \\
\quad \mathit{SyncC}\ \mathit{sync}\ \mathit{gen}\ (\mathit{tail}\ \mathit{types})
\end{array}
$$

The extended translation of the external choice uses extended versions of the auxiliary functions to take into account the existence of arrays of channels.

$$
\begin{array}{l}
[\![\mathit{Action}_1 \,\Box\, \ldots \,\Box\, \mathit{Action}_2]\!]^{\mathit{Action}_{ext}} = \\
\quad \texttt{Guard[] guards =} \\
\quad\quad \texttt{new Guard[]\{}\mathit{InitCAttr}_{ext}\ \mathit{Action}_1\texttt{,}\mathit{InitCAttr}_{ext}\ \mathit{Action}_2\texttt{\};} \\
\quad \texttt{final Alternative alt = new Alternative(guards);} \\
\quad \mathit{DeclConstants}_{ext}\ (\mathit{ExtractInitChannels}_{ext}\ \mathit{Action}_1)\ 0 \\
\quad \ldots \\
\quad \mathit{DeclConstants}_{ext}\ (\mathit{ExtractInitChannels}_{ext}\ \mathit{Action}_n) \\
\quad\quad\quad\quad (\#(\mathit{ExtractInitChannels}_{ext}\ \mathit{Action}_{n-1})) \\
\quad \texttt{boolean[] g =} \\
\quad\quad \texttt{new boolean[]\{}\mathit{Guard}\ \mathit{Action}_1\texttt{,}\ldots\texttt{,}\mathit{Guard}\ \mathit{Action}_n\texttt{\};} \\
\\
\quad \texttt{switch(alt.fairSelect(g)) \{} \\
\quad\quad \mathit{Cases}_{ext}\ (\mathit{ExtractInitChannels}_{ext}\ \mathit{Action}_1)\ \mathit{Action}_1 \\
\quad\quad \ldots \\
\quad\quad \mathit{Cases}_{ext}\ (\mathit{ExtractInitChannels}_{ext}\ \mathit{Action}_2)\ \mathit{Action}_n \\
\quad \texttt{\}}
\end{array}
$$

As its original version, the function *InitCAttr* returns a `,`-separated list of all the visible initials channels of a given action. However, it takes into account possible synchronisation values in the channels.

$$
\begin{array}{l}
\mathit{InitCAttr}_{ext} : \mathsf{Action} \nrightarrow \mathsf{JCode} \\
\mathit{InitCAttr}_{ext}\ \mathit{Action} = \mathit{DecAttrChannels}_{ext}\ (\mathit{ExtractInitChannels}_{ext}\ \mathit{Action})
\end{array}
$$

The function $\mathit{ExtractInitChannels}_{ext}$ returns a list of pairs. For each initial visible channel of the given action, it includes a new pair in this list: the first element is the channel (possibly with its synchronisation values) and the second element is a predicate that represents its guard.

The extended function $DecAttrChannels_{ext}$ can be defined as

$$
\begin{array}{l}
DecAttrChannels_{ext} : \mathrm{seq}(\mathsf{Chann} \times \mathsf{Predicate}) \pfun \mathsf{JCode} \\
DecAttrChannels_{ext}\ [\,] = \epsilon \\
DecAttrChannels_{ext}\ (c[T_0] \ldots [T_n].x_0 \ldots .x_m, p) : [\,] = \\
\quad \texttt{c[Type.T\_0]} \ldots \texttt{[Type.T\_n]}\,\texttt{[}(JExp\ x_0)\texttt{]} \ldots \texttt{[}(JExp\ x_m)\texttt{]} \\
DecAttrChannels_{ext}\ (c[T_0] \ldots [T_n].x_0 \ldots .x_m, p) : cs = \\
\quad \texttt{c[Type.T\_0]} \ldots \texttt{[Type.T\_n]}\,\texttt{[}(JExp\ x_0)\texttt{]} \ldots \texttt{[}(JExp\ x_m)\texttt{],} \\
\quad DecAttrChannels_{ext}\ cs
\end{array}
$$

As the original one, the function $DeclConstants_{ext}$ returns a `;`-separated list of `int` constant declarations, one for each channel in the given channel list. The first one is initialised with the given natural number; each subsequent declaration initialises the current constant with the previous constant value incremented by one. Its definition, however, takes into account the possible existent arrays of channels.

$$
\begin{array}{l}
DeclConstants_{ext} : \mathrm{seq}(\mathsf{Chann} \times \mathsf{Predicate}) \pfun \mathbb{N} \pfun \mathsf{JCode} \\
DeclConstants_{ext}\ [\,]\ n = \epsilon \\
DeclConstants_{ext}\ ((c[T_0] \ldots [T_n].x_0 \ldots .x_m, p) : cs)\ n = \\
\quad \texttt{final int CONST\_(}Capitals\ c\texttt{)\_T\_0} \ldots \texttt{T\_N\_X\_0\_} \ldots \texttt{X\_m = n;} \\
\quad DeclConstants_{ext}\ cs\ (n+1)
\end{array}
$$

Finally, the extended function $Cases_{ext}$ returns a sequence of Java `case` blocks, one for each initial channel in given channel list.

$$
\begin{array}{l}
Cases_{ext} : \mathrm{seq}(\mathsf{Chann} \times \mathsf{Predicate}) \pfun \mathrm{seq}\,\mathsf{Action} \pfun \mathsf{JCode} \\
Cases_{ext}\ [\,]\ as = \epsilon \\
Cases_{ext}\ ((c[T_0] \ldots [T_n].x_0 \ldots .x_m, p) : cs)\ (a : as) = \\
\quad \texttt{case CONST\_(}Capitals\ c\texttt{)\_T\_0} \ldots \texttt{T\_N\_X\_0\_} \ldots \texttt{X\_m = n:} \\
\quad\quad [\![a]\!]^{Action_{ext}} \\
\quad\quad \texttt{break;} \\
\quad Cases_{ext}\ cs\ as
\end{array}
$$

As an example of the extended translation of an external choice, we have the external choice below. Again, for simplicity, we consider all channel as input channels. We consider the type $T_1 == \{0..1\}$ as the only existing type within the *Circus* program, and that the channels a and b are generic channels declared as $\mathbf{channel}[T]\ a, b : T$.

$$
\begin{array}{l}
(a[T_1].0 \then Skip \extchoice a[T_1].1 \then Skip) \extchoice (b[T_1].0 \then Stop \extchoice b[T_1].1 \then Stop) = \\
\quad ((x > 0)\ \&\ a[T_1].0 \then Skip \extchoice (x > 0)\ \&\ a[T_1].1 \then Skip) \\
\quad \extchoice ((x \leq 0)\ \&\ b[T_1].2 \then Stop \extchoice (x \leq 0)\ \&\ b[T_1].1 \then Stop)
\end{array}
$$

First, we declare the array that contains all the visible channels within the action, and an `Alternative` on this array. Notice that each element of the channel arrays are considered as different visible channels.

```
Guard[] guards =
    new Guard[]{a[Type.T_1][0],a[Type.T_1][1],
                b[Type.T_1][0],b[Type.T_1][1]};
final Alternative alt = new Alternative(guards);
```

Then, all the constants that are used in the `switch` block are declared. They identify each possible choice that can be made by the `fairSelect` method.

```
final int CONST_A_T_1_0 = 0; final int CONST_A_T_1_1 = 0;
final int CONST_B_T_1_0 = 0; final int CONST_B_T_1_1 = 0;
```

Next, we declare the array containing the guards for each branch of the action.

```
boolean[] g = new boolean[]{x>0,x>0,x<=0,x<=0};
```

Finally, we have a `switch` block. For each value that can be returned by the method `fairSelect` invocation, we have a `case`, which reads from the corresponding channel, and then behaves like the translation of the corresponding action.

```
switch(alt.fairSelect(g)) {
    case CONST_A_T_1_0:
        a[Type.T_1][0].read(); (new Skip()).run(); break;
    case CONST_A_T_1_1:
        a[Type.T_1][1].read(); (new Skip()).run(); break;
    case CONST_B_T_1_0:
        b[Type.T_1][0].read(); (new Stop()).run(); break;
    case CONST_B_T_1_1:
        b[Type.T_1][1].read(); (new Stop()).run(); break;
}
```

6.4 Indexing Operator

An indexed process can be seen as a kind of parametrised process. The difference, however, is that, before its translation, a syntactic substitution on the channels, as defined in [33], is made. It is very important to notice that the creation of the channels environment already takes into account the indexed processes. So, at this point, the channels implicitly created by the indexed operator are already within the channels environment. A renaming in the channels within a given indexed process $Decl \bullet Proc$ is reflected in the way the channels are instantiated, referenced, and used.

$$[\![\_]\!]^{ParProc_{ext}} : \mathsf{ParProc} \pfun \mathsf{JCode}$$
$$[\![x_1 : T_1;\ \ldots;\ x_n : T_n \odot Proc]\!]^{ParProc_{ext}} =$$
$$\quad [\![(x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Proc)[c : used(Proc) \bullet c\_x\_1 \ldots x\_n]]\!]^{ParProc_{ext}}$$

where, the process $P[c : used(Proc) \bullet c\_x\_1 \ldots x\_n]$ is that obtained from P by changing, in the process, all the references to a used channel c by a reference to the channel $c\_x\_1 \ldots x\_n$.

We may also have an instantiation of a indexed process. It is translated as an invocation of a parameterised process. However, the same syntactic substitution is made before the translation.

$$
\begin{lmath}
[\![\_]\!]^{Proc_{ext}} : \mathsf{Proc} \pfun \mathsf{JCode} \\
[\![(x_1 : T_1; \ \ldots; \ x_n : T_n \odot Proc) \lfloor v_1, \ldots, v_n \rfloor]\!]^{Proc_{ext}} = \\
\quad [\![((x_1 : T_1; \ \ldots; \ x_n : T_n \bullet Proc)[c : used(Proc) \bullet c\_x\_1 \ldots x\_n]) \\
\qquad (v_1, \ldots, v_n)]\!]^{Proc_{ext}}
\end{lmath}
$$

If the instantiation uses the process name, we may translate it as follows.

$$[\![N \lfloor v_1, \ldots, v_n \rfloor]\!]^{Proc_{ext}} = [\![N(v_1, \ldots, v_n)]\!]^{Proc_{ext}}$$

The iterated sequential composition over the indexed operator can be simply translated as an iterated sequential composition. However, the process has all the references to the channels within it changed before the translation. The first iterated indexed operator is the **iterated indexed sequential composition**. For a fresh *index*, its translation can be defined as follows.

$$
\begin{lmath}
[\![(⨟ \ x_1 : T_1; \ \ldots x_2 : T_2 \odot Proc)]\!]^{Proc_{ext}} = \\
\quad [\![(⨟ \ x_1 : T_1; \ \ldots x_2 : T_2 \bullet (Proc[c : used(Proc) \bullet c\_x\_1 \ldots x\_n]))]\!]^{Proc_{ext}}
\end{lmath}
$$

The **indexed external choice** must be expanded before being translated. For instance, consider the following process.

process $Reader \mathrel{\widehat=} \Box \ i : \{0, 1\} \odot read?x \then Skip$

Our strategy considers that, before being translated, the process above is extended as presented below.

process $Reader \mathrel{\widehat=}$
begin state $\bullet$
$out\_i.0?x \then Skip$
$\Box \ out\_i.1?x \then Skip$
end

As the iterated indexed sequential composition, the **iterated indexed internal choice** can be simply translated as an iterated internal choice with the process having all the references to the channels within it changed before the translation.

$$
\begin{lmath}
[\![\sqcap \, x_1 : T_1; \ \ldots; \ x_n : T_n \odot Proc]\!]^{Proc_{ext}} = \\
\quad [\![\sqcap \, x_1 : T_1; \ \ldots; \ x_n : T_n \bullet (Proc[c : used(Proc) \bullet c\_x\_1 \ldots x\_n])]\!]^{Proc_{ext}}
\end{lmath}
$$

Finally, the same applies to the translation of the **iterated indexed parallelism**.

$$[\![\ \|\| x_1 : T_1; \ \ldots; \ x_n : T_n \ [\![\ CSExp \ ]\!] \odot Proc]\!]^{Proc_{ext}} =$$
$$[\![\ \|\| x_1 : T_1; \ \ldots; \ x_n : T_n \ [\![\ CSExp]\!] \bullet$$
$$(Proc[c : used(Proc) \bullet c\_x\_1 \ldots x\_n])]\!]^{Proc_{ext}}$$

6.5 Example

This section will present some parts of the translation of our case study.

6.6 Conclusions

The translation strategy presented in this Chapter has already been used not only to implement some simple *Circus* programs, but also, some quite complex *Circus* programs, as our case study presented in Section 5.

Throughout the translation we assume that the specification has been refined into a specification that meets the translation strategy's requirements. For instance, all operation schemas and specification constructs have already been refined.

One of these requirements is the order of the paragraphs: we assume that, in the *Circus* program to be implemented, we have first Z paragraphs, then, channel declarations, and finally, process declarations. This, however, can be achieved with a simple reordering of the paragraphs.

The Z paragraphs are considered to be only axiomatic definitions of the form $v : T \mid v = e_1$, free types, or abbreviations. The considerations of other types of paragraphs is left as future work.

The next requirement concerns the Z paragraphs used to group channel declarations, and channel sets. Our strategy requires they have already been expanded. This can also be achieved with a simple refinement.

An important restriction is that over communication. In Section 6.2 we consider a very restrict subset of possible communications which is expanded in Section 6.3. The availability of strategies for refine out communications of the form $c?N : T$, multi-synchronisation, and guarded outputs, which are not contemplated by our strategy, would be very useful and is left as future work. As a matter of fact, in [32], Woodcock presents a refinement strategy for a special case of multi-way synchronisation, in which, it is not part of an external choice.

JCSP itself restricts our strategy, for instance, in the translation of parallelism. As discussed in this Chapter, JCSP does not allow the user to determine the synchronisation channel set. Actually, when using JCSP, the intersection of the alpha-

bets determines the synchronisation channels set. A refinement strategy to deal with this problem is also left as future work.

The types of all indexing variables of all iterated operators are considered to be finite. We believe a different approach in the translation could make it possible to remove this restriction in some cases. Lazy evaluation in the generated code would be possibly the way the translation could be changed. Further investigations and research on this topic are also left as future work.

The translation of the iterated parallelism and interleaving of actions is defined as the translation of the its expansion. This, however, could be defined in a similar way as for processes. However, the partitions of the variables in scope must also be taken in scope.

Finally, two very interesting topics for research are left as future work. The first one is the implementation of a tool that supports the translation strategy presented here. In order to prove the soundness of such a tool, the proof of the transformation rules presented here would be completely necessary. This, however, is a very complex task, as it involves the Java and the *Circus* semantics.

Chapter 7

Conclusion

In this chapter we present an overview of the contributions of our work. Furthermore, related works are also brought into the context of *Circus*. A comparison of *Circus* and these works is provided. Finally, topics for future work are presented in two contexts: within the scope of the thesis, and beyond it.

7.1 Contributions

Circus has been suggested as a link between two different schools of formal methods for software engineering: the state-based school and the process algebraic. The former is strongly represented by VDM [15] and Z [34], and the latter is strongly represented by CCS [19] and CSP [13, 26]. Besides providing a link between these two schools, *Circus* also presents a refinement strategy in a calculational style as in [20].

Basically, *Circus* programs are characterised by processes, which group paragraphs that describe data and control behaviour. Mainly, we use the Z notation [30] to define data, and actions, which are defined using Z, CSP, and guarded commands constructs, to characterise behaviour.

The *Circus* semantics is based on the Unifying Theory of Programming [14]: the semantics of a *Circus* program is represented as a Z specification, in which the model of a process is itself a Z specification, and the model of an action is a schema. Since Z is the notation for a *Circus* program's semantics, tools as Z-EVES [17] and ProofPower [3] can be used to analyse its behaviour.

As in the Unifying Theory of Programming, the central notion in *Circus* is refinement. A refinement strategy for *Circus*, based on laws of simulation, and action and process refinements (Appendices C and D), has been proposed in [1] and is also presented in this thesis. Each iteration within this strategy includes three steps: simulation, action refinement, and process refinement.

Our thesis aim at the development of a refinement calculus for *Circus*. At the end of this work, we intend to provide a significant set of refinement laws that, together, can be used in formal developments of *Circus* programs. The proof of the soundness of these laws are also in the scope of this work. In order to point the expressiveness of *Circus* and the significance of its refinement calculus, a case study is presented. Finally, a translation strategy for *Circus* programs into Java is also in our plans.

In this mini-thesis we have already proposed a significant set of new laws (see Appendix D), extending the set of laws presented in [1] (see Appendix C). Furthermore, the translation strategy from *Circus* programs to Java is also finished. It will be used in our case study, which is about to be finished.

Our case study on *Circus*, a safety-critical fire protection system, is described in Chapter 5. As far as we know, this is the largest case study on the refinement of *Circus* programs. We have used this case study to verify the usefulness and soundness of the refinement laws discussed in the previous chapter. Throughout this refinement, we have found a few mistakes on the laws presented in [1]. Using our set of laws, we were able to refine the abstract and centralised specification of the system into a concrete and distributed specification. Furthermore, the case study was also used to analyse the expressiveness of the language. Throughout the

development of the case study, some different ways on how some features could be differently expressed were identified.

In Chapter 6, a translation strategy, which makes possible the Java implementation of *Circus* programs, was presented. In spite of the restrictions on the *Circus* programs discussed in that chapter, this translation strategy is already an important tool in the implementation of *Circus* programs, and has been used to implement our case study. Further, it can be used as guideline for a mechanisation of a translation tool.

Some limitations of JCSP have been raised in Chapter 6. Some of them, however, have a related refinement strategy for solving them. Removing multi-synchronisation is one of these refinement strategy. In [32], a refinement strategy for a special case of multi-way synchronisation is presented: multi-synchronisation events are not part of an external choice. However, in our case study, we have such case, and further, we have multiple multi-synchronisation as part of external choices.

7.2 Related Works

Some other works have already presented the integration of Z or one of its extensions with a process algebra. The main objective of *Circus* is not to be another language like those, but to provide support for the formal development of concurrent programs in a calculational style.

Fischer [10] presents a survey of several integrations of Z with process algebras. Combinations of Z with CCS [11, 31], Z with CSP [27], and Object-Z with CSP [9] are considered in this survey, which also discusses issues involved in the integration of Z with a process algebra. All the approaches above are analysed with respect to these issues.

Two different approaches for the combination of Z and a process algebra are pointed by Fischer: syntactic, in which the combination has a single syntax, with semantic definitions lifted from the two languages; and semantic, in which a Z specification is identified with a process.

Differently from all other combinations mentioned above, *Circus* adopts the first approach: as previously discussed, the unifying theory of programming [14] is used as the model for *Circus*. The syntactic approach adopted by *Circus*, provides a deeper integration of the notations. However, with this approach, the semantics of both the Z and CSP operators must be redefined. Fortunately, this is not a major problem for us because we are using an existing semantic model: UTP. Furthermore, as we express this model using Z, it is possible to analyse *Circus* specifications using Z tools.

Refinement has been studied for combinations of Object-Z and CSP [29]. How-

ever, as far as we know, nothing has been proposed in a calculational style like ours.

The work with action systems is the most closely related to *Circus*. In an action system, systems are described as a state and a set of guarded commands. Its behaviour is given by a simple interpreter for the program that repeatedly selects an enabled action and executes it. Parallelism is modelled as the sequential interleaving of atomic steps. Concurrency with shared variables is modelled by partitioning the variables amongst different processes; a model for distributed systems is obtained by partitioning the variables amongst the processes.

The combination of the refinement calculus and action system in the derivation of parallel and distributed algorithms is described in [4]: from a purely sequential algorithm, a stepwise refinement is accomplished until an efficient parallel program is derived. Most steps involves sequential refinements; the parallelism is introduced only through the decomposition of atomic actions.

The very basic nature of action systems formalism in comparison with process algebra is the main difference between action systems and *Circus*. Action system have a very flat structure, where auxiliary variables simulating program counters are needed to guarantee the proper sequencing of actions. This is due to the simple control flow of action systems: select an enable guard, execute it, repeat. In *Circus* a much more rich control flow is provided using CSP operators.

In [22], stepwise development of correct programs is supported by a design calculus for occam-like [16] communicating programs. Specifications are given in terms of assertions. Both program and specifications semantics are uniformly presented in a predicative style similar to that adopted in the unifying theories of programming. Actually, both works are based in the Esprit ProCos project. Another source of inspiration for further refinement laws for *Circus* actions is the design rules in [22].

7.3 Future Work

This mini-thesis presents just part of the work we aim at presenting as a PhD thesis. Some more adventurous and exciting work is still to be done. However, as expected, some new interesting topics of research have been raised throughout the last year. For this reason, we have divided all the future work topics into two groups: those we intend to investigate in the time left for the conclusion of this thesis, and those left to be investigated in later research projects.

Within the Scope of our Thesis

As discussed before, the semantics of *Circus* has been defined using the Unifying Theory of Programming. However, a shallow embedding of the semantics of *Circus* programs is Z is currently provided. This makes it harder to prove any property of

the language. As we intend to use this semantics to proof all the laws of refinement proposed for *Circus*, a deep embedding semantics must be provided. The definition of this extended semantics for *Circus* and its mechanisation in ProofPower [3] is the next step of our work. This mechanisation will allow us to prove all the existing refinement law of *Circus* refinement calculus. Furthermore, the mechanisation will make it possible to reason about properties of the language, and about the semantics itself; it can also be used as a basis for a theorem prover. The extension of *Circus* semantics and its mechanisation are intended to be the final reference of *Circus* semantics.

Chapter 4 is yet to be done. We intend to implement a prototype of a tool that supports the *Circus* refinement calculus presented in this thesis. Together, the refinement strategy and this tool seem to be a very powerful for formal development of concurrent reactive systems.

Our case study has not already been completed: two more iterations of the refinement strategy are still needed. The first one intends to split the process *Areas* into separated processes, one for each individual area. Finally, the *InternalSystem* has to be decomposed into two separated processes: one for the control itself, and another that represents a display controller (see Figure 5.2). The notation for the presentation of the refinement must also be changed in order to make it simpler to be understood. We intend to use a notation for refinement similar to the one adopted in [20].

The translation strategy presented in Chapter 6 some points must yet be addressed. The first one concerns the translation of iterated parallelism and interleaving of actions. In the work presented here we demand the previous extension of such operators before the translation strategy be applied. In order to remove this requirement, we intend to find a way to generalise the solution provided for the simple parallelism of actions. Furthermore, throughout the translation strategy, some considerations were made concerning the *Circus* programs. Some of these requirements could be satisfied simply by applying some refinement strategies to the *Circus* programs. Some of these refinement strategies are: a refinement strategy to deal with the restriction on the synchronisation set of channels for parallelism and interleaving described in Chapter 6; a refinement strategy for removing guarded outputs; and finally, a refinement strategy for removing multi-synchronisation in a more generic way than that presented in [32], in which, multi-synchronisation is not part of an external choice.

As discussed above, in our case study, multiple multi-synchronisation take part on external choices. An implementation of a protocol based on that presented in [32] has already been done. However, proving the refinement that leads to this implementations is a task still to be done.

Beyond the Scope of our Thesis

The refinement strategy and its laws presented in Chapter 3 do not take into account timed *Circus*, object-oriented *Circus*, and mobile *Circus* processes. Some work has already been done in these areas. Nevertheless, a lot of interesting work is still to be done: the suggestion and proof of new refinement laws that covers these variations of *Circus* is the one that seems most interesting for us.

Furthermore, the mechanisation of the translation strategy presented in Chapter 6 would be very useful in the process of development of *Circus* programs.

Finally, a very interesting topic of research is the support of tactics of *Circus* refinement in a similar way to that presented in [23]. This provides an optimisation in time and effort spent throughout a development of a *Circus* program.

Appendix A

Syntax of Circus.

Program	::=	CircusPar*
CircusPar	::=	Par $\mid$ **channel** CDecl $\mid$ **chanset** N == CSExp
	$\mid$	ProcessDefinition
CDecl	::=	SimpleCDecl $\mid$ SimpleCDecl; CDecl
SimpleCDecl	::=	N^+ $\mid$ N^+ : Exp $\mid$ $[N^+]N^+$: Exp $\mid$ Schema-Exp
CSExp	::=	$\{\!\mid \mid\!\}$ $\mid$ $\{\!\mid N^+ \mid\!\}$ $\mid$ N $\mid$ CSExp $\cup$ CSExp $\mid$ CSExp $\cap$ CSExp
	$\mid$	CSExp $\setminus$ CSExp
ProcessDefinition	::=	**process** N $\widehat{=}$ ParProc $\mid$ **process** N $\widehat{=}$ IndexProc
ParProc	::=	Decl $\bullet$ Proc $\mid$ Proc
IndexProc	::=	Decl $\odot$ Proc
Proc	::=	**begin** PPar* **state** Schema-Exp PPar* $\bullet$ Action **end** $\mid$ N
	$\mid$	Proc; Proc $\mid$ Proc $\Box$ Proc $\mid$ Proc $\sqcap$ Proc
	$\mid$	Proc $[\![$ CSExp $]\!]$ Proc $\mid$ Proc $\mid\!\mid\!\mid$ Proc $\mid$ Proc $\setminus$ CSExp
	$\mid$	IndexProc$\lfloor \text{Exp}^+ \rfloor$ $\mid$ Proc$[N^+ := N^+]$
	$\mid$	$\fatsemi$ Decl $\odot$ Proc $\mid$ $\Box$ Decl $\odot$ Proc $\mid$ $\sqcap$ Decl $\odot$ Proc
	$\mid$	$\parallel$ Decl $[\![$ CSExp $]\!]$ $\odot$Proc $\mid$ $\parallel\!\parallel$ Decl $\odot$ Proc
	$\mid$	ParProc(Exp$^+$)
	$\mid$	$\fatsemi$ Decl $\bullet$ Proc $\mid$ $\Box$ Decl $\bullet$ Proc $\mid$ $\sqcap$ Decl $\bullet$ Proc
	$\mid$	$\parallel$ Decl $[\![$ CSExp$]\!]$ $\bullet$ Proc $\mid$ $\parallel\!\parallel$ Decl $\bullet$ Proc
NSExp	::=	$\{\,\}$ $\mid$ $\{N^+\}$ $\mid$ N $\mid$ NSExp $\cup$ NSExp $\mid$ NSExp $\cap$ NSExp
	$\mid$	NSExp $\setminus$ NSExp
PPar	::=	Par $\mid$ N $\widehat{=}$ ParAction $\mid$ **nameset** N == NSExp
ParAction	::=	Decl $\bullet$ action $\mid$ Action
Action	::=	Schema-Exp $\mid$ CSPAction $\mid$ Command $\mid$ N

$$
\begin{array}{lcl}
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \mathbin{\&} \mathsf{Action} \\
& \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
& \mid & \mathsf{Action} \ [\![\mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp}]\!] \ \mathsf{Action} \\
& \mid & \mathsf{Action} \ \|[\mathsf{NSExp} \mid \mathsf{NSExp}]\| \ \mathsf{Action} \\
& \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \mu\, \mathsf{N} \bullet \mathsf{Action} \mid \mathsf{ParAction}(\mathsf{Exp}^+) \\
& \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Action} \mid \Box\, \mathsf{Decl} \bullet \mathsf{Action} \mid \sqcap\, \mathsf{Decl} \bullet \mathsf{Action} \\
& \mid & \|\ \mathsf{Decl} \ [\![\mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp}]\!] \bullet \mathsf{Action} \\
& \mid & |||\ \mathsf{Decl} \ \|[\mathsf{NSExp} \mid \mathsf{NSExp}]\| \bullet \mathsf{Action} \\
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= & ?\,\mathsf{N} \mid ?\,\mathsf{N} : \mathsf{Predicate} \mid !\,\mathsf{Expression} \mid .\,\mathsf{Expression} \\
\mathsf{Command} & ::= & \mathsf{N}^+ : [\,\mathsf{Pred}, \mathsf{Pred}\,] \mid \mathsf{N}^+ := \mathsf{Exp}^+ \\
& \mid & \mathbf{if}\ \mathsf{GActions}\ \mathbf{fi} \mid \mathbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \\
\mathsf{GActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \Box \mathsf{GActions}
\end{array}
$$

Appendix B

Proof of Lemmas.

Throughout this chapter we use the following notation for the refinement steps.

$$A_1 \sqsubseteq_{\mathcal{A}} [law_1, \ldots, law_n]\{op_1\} \ldots \{op_n\} A_2$$

This denotes that, in order to refine the action A_1 to A_2, we have applied laws $law_1, \ldots, law_n$. These law applications have raised the proof obligations $op_1, \ldots, op_n$.

General Lemmas

Lemma B.1

$$c_1?x \rightarrow SExp_1;c_2 \rightarrow \left(\begin{array}{l} g_1 \;\&\; (A_1 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A) \\ \Box\; g_2 \;\&\; (A_2 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A) \end{array} \right)$$
$$=$$
$$\left(c_1?x \rightarrow c_2 \rightarrow \left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) \right) [\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, (c_1?x \rightarrow SExp_1;A)$$

provided

- $initials(A) \subseteq cs_1$
- $c_2 \notin cs_1$
- $c_1 \notin usedC(A_1) \cup usedC(A_2)$ *or* $c_1 \in cs_1$
- $wrtV(Sexp_1) \subseteq ns_2 \cup ns_2'$
- $wrtV(SExp_1) \cap (usedV(A_1) \cup usedV(A_2)) = \emptyset$

Proof.

$$
\begin{array}{l}
c_1?x \to SExp_1;c_2 \to \left(\begin{array}{l} g_1 \;\&\; (A_1 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A) \\ \Box\; g_2 \;\&\; (A_2 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A) \end{array} \right) \\
= [D.10] \\
\{initials(A) \subseteq cs\} \\
c_1?x \to SExp_1;c_2 \to \left(\left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) [\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A \right) \\
= [D.4, C.44] \\
\{initials(A) \subseteq cs_1, c_2 \notin cs_1\} \\
c_1?x \to SExp_1; \left(\left(c_2 \to \left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) \right) [\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A \right) \\
= [D.29] \\
c_1?x \to SExp_1; \left(A \,[\![\, ns_2 \mid cs_1 \mid ns_1 \,]\!] \left(c_2 \to \left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) \right) \right) \\
= [D.4, D.28] \\
\{wrtV(SExp_1) \subseteq ns_2 \cup ns_2'\} \\
\{wrtV(SExp_1) \cap (usedV(A_1) \cup usedV(A_2)) = \emptyset\} \\
c_1?x \to \left((SExp_1;A) \,[\![\, ns_2 \mid cs_1 \mid ns_1 \,]\!] \left(c_2 \to \left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) \right) \right) \\
= [D.6] \\
\{c_1 \notin usedC(A_1) \cup usedC(A_2) \;\; or \;\; c_1 \in cs_1\} \\
(c_1?x \to SExp_1;A) \,[\![\, ns_2 \mid cs_1 \mid ns_1 \,]\!] \left(c_1?x \to c_2 \to \left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) \right) \\
= [D.29] \\
\left(c_1?x \to c_2 \to \left(\begin{array}{l} g_1 \;\&\; A_1 \\ \Box\; g_2 \;\&\; A_2 \end{array} \right) \right) [\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, (c_1?x \to SExp_1;A)
\end{array}
$$

□

Lemma B.2 *For any variable $x : T$ in scope.*

$$
\begin{array}{l}
(g_1(x) \;\&\; (A_{1_1}(x) \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; A_{1_2}(x)) \\
\Box\; g_2(x) \;\&\; (A_{2_1}(x) \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; A_{2_2}(x))) \\
= \\
\left(\begin{array}{l}
\left(\; comm.x \rightarrow \left(\begin{array}{l} c_1.x \rightarrow A_{1_1}(x) \\ \Box\; c_2.x \rightarrow A_{2_1}(x) \end{array} \right) \right) \\
[\![ns_1 \mid cs \cup \{comm, c_1, c_2\} \mid ns_2]\!] \\
\left(\; comm.x \rightarrow \left(\begin{array}{l} g_1(x) \;\&\; c_1.x \rightarrow A_{1_2}(x) \\ \Box\; g_2(x) \;\&\; c_2.x \rightarrow A_{2_2}(x) \end{array} \right) \right)
\end{array} \right) \setminus \{comm, c_1, c_2\}
\end{array}
$$

provided

- $\{c_1, c_2, comm\} \cap (usedC(A_{1_1}) \cup usedC(A_{1_2}) \cup usedC(A_{2_1}) \cup usedC(A_{2_2})) = \emptyset$

Proof.

$$
\begin{array}{l}
g_1(x) \;\&\; (A_{1_1}(x) \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; A_{1_2}(x)) \\
\Box\; g_2(x) \;\&\; (A_{2_1}(x) \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; A_{2_2}(x)) \\
\sqsubseteq_{\mathcal{A}} [D.5] \\
\{c_1 \notin usedC(A_{1_1}(x)) \cup usedC(A_{1_2}(x))\} \\
\{c_2 \notin usedC(A_{2_1}(x)) \cup usedC(A_{2_2}(x))\} \\
g_1(x) \;\&\; (c_1.x \rightarrow (A_{1_1}(x) \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; A_{1_2}(x))) \setminus \{\!|\; c_1 \;|\!\} \\
\Box\; g_2(x) \;\&\; (c_2.x \rightarrow (A_{2_1}(x) \;[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\; A_{2_2}(x))) \setminus \{\!|\; c_2 \;|\!\} \\
\sqsubseteq_{\mathcal{A}} [C.51] \\
\{c_1 \notin usedC(A_{1_1}(x)) \cup usedC(A_{1_2}(x))\} \\
\{c_2 \notin usedC(A_{2_1}(x)) \cup usedC(A_{2_2}(x))\} \\
g_1(x) \;\&\; ((c_1.x \rightarrow A_{1_1}(x)) \;[\![\, ns_1 \mid cs \cup \{c_1\} \mid ns_2 \,]\!]\; (c_1.x \rightarrow A_{1_2}(x))) \setminus \{\!|\; c_1 \;|\!\} \\
\Box\; g_2(x) \;\&\; ((c_2.x \rightarrow A_{2_1}(x)) \;[\![\, ns_1 \mid cs \cup \{c_2\} \mid ns_2 \,]\!]\; (c_2.x \rightarrow A_{2_2}(x))) \setminus \{\!|\; c_2 \;|\!\} \\
\sqsubseteq_{\mathcal{A}} [C.54, C.41] \\
\{c_1 \notin usedC(A_{2_1}(x)) \cup usedC(A_{2_2}(x))\} \\
\{c_2 \notin usedC(A_{1_1}(x)) \cup usedC(A_{1_2}(x))\} \\
\left(\begin{array}{l}
g_1(x) \;\& \\
\qquad \left(\begin{array}{l} (c_1.x \rightarrow A_{1_1}(x)) \\ [\![ns_1 \mid cs \cup \{c_1, c_2\} \mid ns_2]\!] \\ (c_1.x \rightarrow A_{1_2}(x)) \end{array} \right) \\
\Box\; g_2(x) \;\& \\
\qquad \left(\begin{array}{l} (c_2.x \rightarrow A_{2_1}(x)) \\ [\![ns_1 \mid cs \cup \{c_1, c_2\} \mid ns_2]\!] \\ (c_2.x \rightarrow A_{2_2}(x)) \end{array} \right)
\end{array} \right) \setminus \{\!|\; c_1, c_2 \;|\!\}
\end{array}
$$

$$
\begin{array}{l}
= [D.11] \\
\{\{c_1, c_2\} \subseteq cs \cup \{c_1, c_2\} \equiv true\} \\
\{\{c_1\} \cup usedC(A_{1_2}(x)) \cap \{c_2\} = \emptyset \equiv c_2 \notin usedC(A_{1_2}(x))\} \\
\{\{c_2\} \cup usedC(A_{2_2}(x)) \cap \{c_1\} = \emptyset \equiv c_1 \notin usedC(A_{2_2}(x))\} \\
\left(\begin{array}{l}
\left(\begin{array}{l} c_1.x \rightarrow A_{1_1}(x) \\ \Box\ c_2.x \rightarrow A_{2_1}(x) \end{array} \right) \\
[\![ns_1 \mid cs \cup \{c_1, c_2\} \mid ns_2]\!] \\
\left(\begin{array}{l} g_1(x)\ \&\ c_1.x \rightarrow A_{1_2}(x) \\ \Box\ g_2(x)\ \&\ c_2.x \rightarrow A_{2_2}(x) \end{array} \right)
\end{array} \right) \setminus \{\!|\ c_1, c_2\ |\!\} \\
= [C.41, C.54, D.37] \\
\{comm \notin \{c_1, c_2\} \cup usedC(A_{1_1}(x)) \cup usedC(A_{1_2}(x)) \cup \\
\qquad\qquad usedC(A_{2_1}(x)) \cup usedC(A_{2_2}(x))\} \\
\{comm \in cs \cup \{comm, c_1, c_2\} \equiv true\} \\
\left(\begin{array}{l}
\left(\begin{array}{l} comm.x \rightarrow \left(\begin{array}{l} c_1.x \rightarrow A_{1_1}(x) \\ \Box\ c_2.x \rightarrow A_{2_1}(x) \end{array} \right) \end{array} \right) \\
[\![ns_1 \mid cs \cup \{comm, c_1, c_2\} \mid ns_2]\!] \\
\left(\begin{array}{l} comm.x \rightarrow \left(\begin{array}{l} g_1(x)\ \&\ c_1.x \rightarrow A_{1_2}(x) \\ \Box\ g_2(x)\ \&\ c_2.x \rightarrow A_{2_2}(x) \end{array} \right) \end{array} \right)
\end{array} \right) \setminus \{comm, c_1, c_2\}
\end{array}
$$

□

Lemma B.3

$$
\begin{array}{l}
(\fatsemi\, x : S_2 \bullet ((x \in S_1)\ \&\ A_1(x)) \,\Box\, ((x \notin S_1)\ \&\ A_2(x)));\\
(S_1 = \emptyset)\ \&\ A_3 \,\Box\, (S_1 \neq \emptyset)\ \&\ A_4\\
\sqsubseteq_{\mathcal{A}}\\
\left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet\\
\quad log := 0;\\
\quad (\fatsemi\, x : S_2 \bullet ((x \in S_1); A_1(x); log := log + 1) \,\Box\, ((x \notin S_1)\ \&\ A_2(x)));\\
\quad ((log = 0)\ \&\ A_3 \,\Box\, (log > 0)\ \&\ A_4)
\end{array}\right)
\end{array}
$$

provided log is a fresh variable name.

Lemma B.4

$$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; B_2)$$
$$=$$
$$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; (B_2 \;\Box\; B_3))$$

provided

- $initials(A_1) \cup initials(A_2) \subseteq cs$
- $initials(A_1) \cap initials(B_3) = \emptyset$

Proof.

$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; B_2)$
$= [C.58](A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; B_2)) \;\Box\; Stop$
$= [D.27]$
$\{initials(A_1) \cup initials(A_2) \subseteq cs\}$
$\{initials(A_1) \cap initials(B_3) = \emptyset\}$
$(A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; B_2)) \;\Box\; (A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, g_2 \;\&\; B_3)$
$= [C.46]$
$\{initials(A_1) \subseteq cs\}$
$(A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; B_2 \;\Box\; g_2 \;\&\; B_3)$
$= [C.18]$
$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \;\&\; B_1 \;\Box\; g_2 \;\&\; (B_2 \;\Box\; B_3))$

$\Box$

Lemmas of Section 5.6.2

Lemma 5.1.

$$\forall\, AbstractFireControlState;\ FireControlState_1 \bullet \\ RetrFireControl \land \text{pre}\ InitAbstractFireControl \Rightarrow \text{pre}\ InitFireControl_1$$

Since the precondition of the initialisation is *true*, this implication is directly *true* as well.

$$\begin{array}{l} \forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet \\ \quad RetrFireControl \land \text{pre}\ InitAbstractFireControl \land InitFireControl_1 \Rightarrow \\ \quad\quad \exists\, AbstractFireControlState' \bullet \\ \quad\quad\quad RetrFireControl' \land InitAbstractFireControl \\ \equiv [Definition] \\ \forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet \\ \quad mode_1 = mode \land \\ \quad controlledZones_1 = controlledZones \land activeZones_1 = activeZones \\ \quad \land\ discharge_1 = discharge \land active_1 = active \\ \quad \land\ true \\ \quad \land\ mode_1' = automatic \\ \quad \land\ activeZones_1' = \{area : AreaId \bullet area \mapsto \emptyset\} \\ \quad \land\ discharge_1' = \{area : AreaId \bullet area \mapsto false\} \\ \quad \land\ mode_A' = automatic \Rightarrow \\ \quad\quad \exists\, AbstractFireControlState' \bullet \\ \quad\quad\quad mode_1' = mode' \\ \quad\quad\quad \land\ controlledZones_1' = controlledZones' \\ \quad\quad\quad \land\ activeZones_1' = activeZones' \\ \quad\quad\quad \land\ discharge_1' = discharge' \land active_1' = active' \\ \quad\quad\quad \land\ mode' = automatic \\ \quad\quad\quad \land\ activeZones' = \{area : AreaId \bullet area \mapsto \emptyset\} \\ \quad\quad\quad \land\ discharge' = \{area : AreaId \bullet area \mapsto false\} \end{array}$$

$\equiv [E.1, E.2]$
$mode_1 = mode \land$
$controlledZones_1 = controlledZones \land activeZones_1 = activeZones$
$\land\ discharge_1 = discharge \land active_1 = active$
$\land\ true$
$\land\ mode'_1 = automatic$
$\land\ activeZones'_1 = \{area : AreaId \bullet area \mapsto \emptyset\}$
$\land\ discharge'_1 = \{area : AreaId \bullet area \mapsto false\}$
$\land\ mode'_A = automatic \Rightarrow$
$\quad mode'_1 = automatic$
$\quad \land\ activeZones'_1 = \{area : AreaId \bullet area \mapsto \emptyset\}$
$\quad \land\ discharge'_1 = \{area : AreaId \bullet area \mapsto false\}$
$\equiv [E.3]$
$true$

□

Lemma 5.2.

$$\forall\, AbstractFireControlState;\ FireControlState_1 \bullet$$
$$RetrFireControl \land \text{pre}\ SwitchAbstractFireControlMode \Rightarrow$$
$$\text{pre}\ SwitchFireControlMode_1$$

Since the precondition of the schemas are *true*, this implication is directly *true* as well.

$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$RetrFireControl \land \text{pre}\ SwitchAbstractFireControlMode \land$$
$$SwitchFireControlMode_1 \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$RetrFireControl' \land SwitchAbstractFireControlMode$$
$$\equiv [Definition]$$
$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$mode_1 = mode \land$$
$$controlledZones_1 = controlledZones \land activeZones_1 = activeZones$$
$$\land\ discharge_1 = discharge \land active_1 = active$$
$$\land\ true$$
$$\land\ mode_1' = newMode?$$
$$\land\ activeZones_1' = activeZones_1$$
$$\land\ discharge_1' = discharge_1$$
$$\land\ mode_A' = newMode? \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$mode_1' = mode'$$
$$\land\ controlledZones_1' = controlledZones'$$
$$\land\ activeZones_1' = activeZones'$$
$$\land\ discharge_1' = discharge' \land active_1' = active'$$
$$\land\ mode' = newMode?$$
$$\land\ activeZones' = activeZones$$
$$\land\ discharge' = discharge$$

$\equiv [E.1, E.2]$
$mode_1 = mode \wedge$
$controlledZones_1 = controlledZones \wedge activeZones_1 = activeZones$
$\wedge\ discharge_1 = discharge \wedge active_1 = active$
$\wedge\ true$
$\wedge\ mode_1' = newMode?$
$\wedge\ activeZones_1' = activeZones_1$
$\wedge\ discharge_1' = discharge_1$
$\wedge\ mode_A' = newMode? \Rightarrow$
$\quad mode_1' = newMode?$
$\quad \wedge\ activeZones_1' = activeZones$
$\quad \wedge\ discharge_1' = discharge$
$\equiv [E.4, E.3]$
$true$

□

Lemma 5.3.

$$\forall\, AbstractFireControlState;\ FireControlState_1 \bullet$$
$$RetrFireControl \land \text{pre } SwitchAbstractFireControl2AutomaticMode \Rightarrow$$
$$\text{pre } SwitchFireControl2AutomaticMode_1$$

Since the precondition of the schemas are *true*, this implication is directly *true* as well.

$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$RetrFireControl \land \text{pre } SwitchAbstractFireControl2AutomaticMode \land$$
$$SwitchFireControl2AutomaticMode_1 \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$RetrFireControl'$$
$$\land\ SwitchAbstractFireControl2AutomaticMode$$
$$\equiv [Definition]$$
$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$mode_1 = mode \land$$
$$controlledZones_1 = controlledZones \land activeZones_1 = activeZones$$
$$\land\ discharge_1 = discharge \land active_1 = active$$
$$\land\ true$$
$$\land\ mode_1' = automatic$$
$$\land\ activeZones_1' = activeZones_1$$
$$\land\ discharge_1' = discharge_1$$
$$\land\ mode_A' = mode_A \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$mode_1' = mode'$$
$$\land\ controlledZones_1' = controlledZones'$$
$$\land\ activeZones_1' = activeZones'$$
$$\land\ discharge_1' = discharge' \land active_1' = active'$$
$$\land\ mode' = automatic$$
$$\land\ activeZones' = activeZones$$
$$\land\ discharge' = discharge$$

$\equiv [E.1, E.2]$
$mode_1 = mode \wedge$
$controlledZones_1 = controlledZones \wedge activeZones_1 = activeZones$
$\wedge\ discharge_1 = discharge \wedge active_1 = active$
$\wedge\ true$
$\wedge\ mode'_1 = automatic$
$\wedge\ activeZones'_1 = activeZones_1$
$\wedge\ discharge'_1 = discharge_1$
$\wedge\ mode'_A = mode_A \Rightarrow$
$\quad mode'_1 = automatic$
$\quad \wedge\ activeZones'_1 = activeZones$
$\quad \wedge\ discharge'_1 = discharge$
$\equiv [E.4, E.3]$
$true$

□

Lemma 5.4.

$$\forall\, AbstractFireControlState;\ FireControlState_1 \bullet$$
$$RetrFireControl \land \text{pre } SwitchAbstractFireControl2DisabledMode \Rightarrow$$
$$\text{pre } SwitchFireControl2DisabledMode_1$$

Since the precondition of the schemas are *true*, this implication is directly *true* as well.

$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$RetrFireControl \land \text{pre } SwitchAbstractFireControl2DisabledMode \land$$
$$SwitchFireControl2DisabledMode_1 \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$RetrFireControl'$$
$$\land\ SwitchAbstractFireControl2DisabledMode$$
$$\equiv [Definition]$$
$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$mode_1 = mode \land$$
$$controlledZones_1 = controlledZones \land activeZones_1 = activeZones$$
$$\land\ discharge_1 = discharge \land active_1 = active$$
$$\land\ true$$
$$\land\ mode_1' = Disabled$$
$$\land\ activeZones_1' = activeZones_1$$
$$\land\ discharge_1' = discharge_1$$
$$\land\ mode_A' = mode_A \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$mode_1' = mode'$$
$$\land\ controlledZones_1' = controlledZones'$$
$$\land\ activeZones_1' = activeZones'$$
$$\land\ discharge_1' = discharge' \land active_1' = active'$$
$$\land\ mode' = Disabled$$
$$\land\ activeZones' = activeZones$$
$$\land\ discharge' = discharge$$

$\equiv [E.1, E.2]$
$mode_1 = mode \wedge$
$controlledZones_1 = controlledZones \wedge activeZones_1 = activeZones$
$\wedge\ discharge_1 = discharge \wedge active_1 = active$
$\wedge\ true$
$\wedge\ mode'_1 = Disabled$
$\wedge\ activeZones'_1 = activeZones_1$
$\wedge\ discharge'_1 = discharge_1$
$\wedge\ mode'_A = mode_A \Rightarrow$
$\quad mode'_1 = Disabled$
$\quad \wedge\ activeZones'_1 = activeZones$
$\quad \wedge\ discharge'_1 = discharge$
$\equiv [E.4, E.3]$
$true$

□

Lemma 5.5.

$$\forall\, AbstractFireControlState;\ FireControlState_1 \bullet$$
$$RetrFireControl \land \text{pre}\ AbstractActivateZone \Rightarrow$$
$$\text{pre}\ ActivateZone_1$$

Since the precondition of the schemas are *true*, this implication is directly *true* as well.

$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$RetrFireControl \land \text{pre}\ AbstractActivateZone \land$$
$$ActivateZone_1 \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$RetrFireControl' \land AbstractActivateZone$$
$$\equiv [Definition]$$
$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$mode_1 = mode \land$$
$$controlledZones_1 = controlledZones$$
$$\land\ activeZones_1 = activeZones$$
$$\land\ discharge_1 = discharge \land active_1 = active$$
$$\land\ true$$
$$\land\ mode_1' = mode_1$$
$$\land\ activeZones_1' = activeZones_1 \oplus$$
$$\{area : AreaId \mid newZone? \in controlledZones_1(area) \bullet$$
$$area \mapsto activeZones_1(area) \cup \{newZone?\}\}$$
$$\land\ discharge_1' = discharge_1$$
$$\land\ mode_A' = mode_A \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$mode_1' = mode'$$
$$\land\ controlledZones_1' = controlledZones'$$
$$\land\ activeZones_1' = activeZones'$$
$$\land\ discharge_1' = discharge' \land active_1' = active'$$
$$\land\ mode' = mode$$
$$\land\ activeZones' = activeZones \oplus$$
$$\{area : AreaId \mid newZone? \in controlledZones(area) \bullet$$
$$area \mapsto activeZones(area) \cup \{newZone?\}\}$$
$$\land\ discharge' = discharge$$

$\equiv [E.1, E.2]$
$mode_1 = mode \land$
$controlledZones_1 = controlledZones \land activeZones_1 = activeZones$
$\land\ discharge_1 = discharge \land active_1 = active$
$\land\ true$
$\land\ mode'_1 = mode_1$
$\land\ activeZones'_1 = activeZones_1 \oplus$
$\quad \{area : AreaId \mid newZone? \in controlledZones_1(area) \bullet$
$\quad\quad area \mapsto activeZones_1(area) \cup \{newZone?\}\}$
$\land\ discharge'_1 = discharge_1$
$\land\ mode'_A = mode_A \Rightarrow$
$\quad mode'_1 = mode$
$\quad \land\ activeZones'_1 = activeZones \oplus$
$\quad\quad \{area : AreaId \mid newZone? \in controlledZones(area) \bullet$
$\quad\quad\quad area \mapsto activeZones(area) \cup \{newZone?\}\}$
$\quad \land\ discharge'_1 = discharge$
$\equiv [E.4, E.3]$
$true$

□

Lemma 5.6.

$$\forall\, AbstractFireControlState;\ FireControlState_1 \bullet$$
$$RetrFireControl \land \text{pre}\ AbstractActivateDischarge \Rightarrow$$
$$\text{pre}\ ActivateDischarge_1$$

Since the precondition of the schemas are *true*, this implication is directly *true* as well.

$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$RetrFireControl \land \text{pre}\ AbstractActivateDischarge \land$$
$$ActivateDischarge_1 \Rightarrow$$
$$\exists\, AbstractFireControlState' \bullet$$
$$RetrFireControl' \land AbstractActivateDischarge$$
$$\equiv [Definition]$$
$$\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet$$
$$mode_1 = mode \land$$
$$controlledZones_1 = controlledZones$$
$$\land\ activeZones_1 = activeZones$$
$$\land\ discharge_1 = discharge \land active_1 = active$$
$$\land\ true$$
$$\land\ mode_1' = mode_1$$
$$\land\ activeZones_1' = activeZones_1$$
$$\land\ discharge_1' = discharge_1 \oplus$$
$$\{area : AreaId \mid area \in \text{dom}\ active_1 \rhd \{true\} \bullet area \mapsto true\}$$
$$\land\ mode_A' = mode_A$$
$$\Rightarrow \exists\, AbstractFireControlState' \bullet$$
$$mode_1' = mode'$$
$$\land\ controlledZones_1' = controlledZones'$$
$$\land\ activeZones_1' = activeZones'$$
$$\land\ discharge_1' = discharge' \land active_1' = active'$$
$$\land\ mode' = mode$$
$$\land\ activeZones' = activeZones$$
$$\land\ discharge' = discharge \oplus$$
$$\{area : AreaId \mid area \in \text{dom}\ active \rhd \{true\}$$
$$\bullet\ area \mapsto true\}$$

$$
\begin{array}{l}
\equiv [E.1, E.2] \\
mode_1 = mode \wedge \\
controlledZones_1 = controlledZones \wedge activeZones_1 = activeZones \\
\wedge\ discharge_1 = discharge \wedge active_1 = active \\
\wedge\ true \\
\wedge\ mode_1' = mode_1 \\
\wedge\ controlledZones_1' = controlledZones_1 \\
\wedge\ activeZones_1' = activeZones_1 \\
\wedge\ discharge_1' = discharge_1 \oplus \\
\qquad \{area : AreaId \mid area \in \mathrm{dom}\ active_1 \rhd \{true\} \bullet area \mapsto true\} \\
\wedge\ active_1' = active_1 \\
\wedge\ mode_A' = mode_A \\
\Rightarrow mode_1' = mode \\
\quad \wedge\ activeZones_1' = activeZones \\
\quad \wedge\ discharge_1' = discharge \oplus \\
\qquad \{area : AreaId \mid area \in \mathrm{dom}\ active \rhd \{true\} \bullet area \mapsto true\} \\
\equiv [E.4, E.3] \\
true
\end{array}
$$

□

Lemmas of Section 5.6.3

Lemma B.5

$$
\begin{array}{l}
\{mode_A = manual \wedge mode_1 = manual\};Manual_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} Manual_2 \\ \|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
\{mode_A = manual \land mode_1 = manual\};Manual_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ \ Manual_1, Substitution] \\
\{mode_A = manual \land mode_1 = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad \setminus GasDischargeSync \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow ActivateDischarge_1; \\
\quad\quad\quad\quad SwitchFireControl2DisabledMode_1; \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad\quad\quad\quad \setminus GasDischargeSync \\
\quad\quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad\quad\quad\quad \setminus GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.54, \mathit{Schema\ \ Calculus}] \\
\left(\begin{array}{l}
\{mode_A = manual \land mode_1 = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode; \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.13, D.19, C.30, C.36, C.57, C.33, C.35] \\
\{mode_A = manual \wedge mode_1 = manual \Rightarrow mode_A = manual \wedge mode_1 = manual\} \\
\left(\begin{array}{l}
\{mode_A = manual \wedge mode_1 = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \{mode_A = manual \wedge mode_1 = manual\}; \\ (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode; \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \{mode_1 = disabled\}; \\ (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.17, C.36, C.57] \\
\{mode_A = manual \wedge mode_1 = manual \Rightarrow mode_A = manual \wedge mode_1 = manual\} \\
\{mode_A = manual \wedge mode_1 = manual \Rightarrow \neg\, (mode_1 \neq manual)\} \\
\{mode_1 = disabled \Rightarrow mode_1 = disabled\} \\
\{mode_1 = disabled \Rightarrow \neg\, (mode_A = manual \wedge mode_1 = manual)\} \\
\left(\begin{array}{l}
\{mode_A = manual \wedge mode_1 = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode; \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (DisabledAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [D.14] \\
\{mode_A = manual \wedge mode_1 = manual \Rightarrow mode_A = manual\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode; \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (DisabledAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.28, D.29] \\
\{wrtV(SwitchInternalSystem2DisabledMode) \subseteq \\
\quad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(SwitchInternalSystem2DisabledMode) \cap \\
\quad usedV(DisabledAreas;AreasCycle) = \emptyset\} \\
\{wrtV(ActivateDischargeAS) \subseteq \\
\quad \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(ActivateDischargeAS) \cap usedV(Reset_2) = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} (SwitchInternalSystem2DisabledMode;Reset_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActivateDischargeAS;DisabledAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4, C.44] \\
\{initials(DisabledAreas) \subseteq \Sigma_2\} \\
\{switchLamp \notin \Sigma_2\} \\
\{initials(ActiveAreas) \subseteq \Sigma_2\} \\
\{\{silenceAlarm, alarm\} \cap \Sigma_2 = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} (silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} \left(\begin{array}{l} switchLamp[AreaId].area!on \rightarrow \\ \quad SwitchInternalSystem2DisabledMode;Reset_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActivateDischargeAS;DisabledAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4, C.44, D.29, D.28] \\
\{initials(ActiveAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{\{fault, switchLamp, switchBuzzer\} \cap \Sigma_2 = \emptyset\} \\
\{\{faultId\} \cap usedV(ActiveAreas;AreasCycle) = \emptyset\} \\
\{wrtV(ActivateZoneAS) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(ActivateZoneAS) \cap usedV(X) = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow \\
\quad\quad \left(\begin{array}{l} (switchLamp[ZoneId].newZone!on \rightarrow Manual_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActivateZoneAS;ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} (silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} \left(\begin{array}{l} switchLamp[AreaId].area!on \rightarrow \\ \quad SwitchInternalSystem2DisabledMode;Reset_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActivateDischargeAS;DisabledAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad \left(\begin{array}{l} Manual_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.6] \\
\{detection \in \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
(silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad \left(\begin{array}{l}
\left(\begin{array}{l} switchLamp[AreaId].area!on \rightarrow \\ \quad SwitchInternalSystem2DisabledMode;Reset_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS;DisabledAreas;AreasCycle)
\end{array} \right) \\
\qquad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad \left(\begin{array}{l}
Manual_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [B.2, C.54, C.53] \\
\{\{gasDischarged, gasNotDischarged, manualDischarge\} \cap \\
\quad (\{switchLamp\} \cup usedC(Reset_2) \cup usedC(DisabledAreas;AreasCycle) \\
\qquad \cup usedC(Manual_2) \cup usedC(ActiveAreas;AreasCycle)) = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
(silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\
\quad gasDischarged.area \rightarrow \\
\qquad switchLamp[AreaId].area!on \rightarrow \\
\qquad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \\
\qquad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\qquad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\qquad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\qquad\quad DisabledAreas;AreasCycle \\
\quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\qquad gasNotDischarged.area \rightarrow \\
\qquad\quad ActiveAreas;AreasCycle
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \qquad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.32] \\
\{manualDischarge \in \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} (silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad \left(\begin{array}{l} manualDischarge.area \rightarrow \\ \quad gasDischarged.area \rightarrow \\ \quad\quad switchLamp[AreaId].area!on \rightarrow \\ \quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\ \quad \Box\ gasNotDischarged.area \rightarrow Manual_2 \end{array} \right) \\
\qquad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\qquad \left(\begin{array}{l} \Box\ area : AreaId \bullet \\ \quad manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas;AreasCycle \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas;AreasCycle \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.22, C.46] \\
\{initials(ActiveAreas) \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\
\quad gasDischarged.area \rightarrow \\
\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \\
\quad\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad\quad \left(\begin{array}{l}
\Box\ area : AreaId \bullet \\
\quad manualDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas;AreasCycle \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad ActiveAreas;AreasCycle
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\ \Box\ fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.58, D.27, C.46] \\
\{\{detection, reset, manualDischarge\} \subseteq \Sigma_2\} \\
\{\{detection\} \cap \{reset, manualDischarge\} = \emptyset\} \\
\{\{detection\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \\ \Box\ reset \rightarrow InitAreas;AreasCycle \\ \Box\ \Box\ area : AreaId \bullet \\ \quad manualDischarge.area \rightarrow \\ \quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad DisabledAreas;AreasCycle \\ \quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad gasNotDischarged.area \rightarrow \\ \quad ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} manualDischarge.area \rightarrow \\ \quad gasDischarged.area \rightarrow \\ \quad switchLamp[AreaId].area!on \rightarrow \\ \quad SwitchInternalSystem2DisabledMode;Reset_2 \\ \Box\ gasNotDischarged.area \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} \Box\ area : AreaId \bullet \\ \quad manualDischarge.area \rightarrow \\ \quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad DisabledAreas;AreasCycle \\ \quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad gasNotDischarged.area \rightarrow \\ \quad ActiveAreas;AreasCycle \end{array} \right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l} silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\ \Box\ fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.24, D.24] \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas \\
\Box\ reset \rightarrow InitAreas \\
\Box\ \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad\quad\quad DisabledAreas \\ \quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad ActiveAreas
\end{array}\right) \\
AreasCycle
\end{array}\right) ; \\
\Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\ \quad gasDischarged.area \rightarrow \\ \quad\quad switchLamp[AreaId].area!on \rightarrow \\ \quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\ \quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array}\right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
\Box\ area : AreaId \bullet \\ \quad manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas;AreasCycle \\ \quad\quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas;AreasCycle
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.13, D.19, C.30, C.36, C.57] \\
\{mode_A = manual \Rightarrow mode_A = manual\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas \\
\Box\ reset \rightarrow InitAreas \\
\Box\ \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad\quad\quad DisabledAreas \\ \quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad ActiveAreas
\end{array} \right); \\
AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\ \quad gasDischarged.area \rightarrow \\ \quad\quad switchLamp[AreaId].area!on \rightarrow \\ \quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \\
\quad\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad\quad \left(\begin{array}{l}
\Box\ area : AreaId \bullet \\ \quad manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas;AreasCycle \\ \quad\quad \Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas;AreasCycle
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\ \Box\ fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Manual_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.17, C.36, C.57, D.22, \mathit{Definition\ of\ ActiveAreas}] \\
\{mode_A = manual \Rightarrow mode_A = manual\} \\
\{mode_A = manual \Rightarrow \neg\ mode_A = automatic\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\
\quad gasDischarged.area \rightarrow \\
\qquad switchLamp[AreaId].area!on \rightarrow \\
\qquad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \\
\qquad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\qquad \left(\begin{array}{l}
\Box\ area : AreaId \bullet \\
\quad manualDischarge.area \rightarrow \\
\qquad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\qquad\qquad DisabledAreas;AreasCycle \\
\qquad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\qquad ActiveAreas;AreasCycle
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad switchBuzzer!on \rightarrow Manual_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$= [D.22, C.46]$

$\{initials(ActiveAreas;AreasCycle) \subseteq \Sigma_2\}$

$$
\left(\begin{array}{l}
\{mode_A = manual\};\\
systemState!manual_s \rightarrow\\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow\\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2\\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2\\
\Box\ fault?faultId : FaultId \rightarrow\\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow\\
\quad\quad switchBuzzer!on \rightarrow Manual_2
\end{array}\right)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
(ActiveAreas;AreasCycle)
\end{array}\right)\\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow\\
\quad\quad \left(\begin{array}{l}
manualDischarge.area \rightarrow\\
\quad gasDischarged.area \rightarrow\\
\quad\quad switchLamp[AreaId].area!on \rightarrow\\
\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2\\
\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array}\right)\\
\quad\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
\quad\quad \left(\begin{array}{l}
\Box\ area : AreaId \bullet\\
\quad manualDischarge.area \rightarrow\\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \&\\
\quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS;\\
\quad\quad\quad\quad DisabledAreas;AreasCycle\\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \&\\
\quad\quad\quad gasNotDischarged.area \rightarrow\\
\quad\quad\quad\quad ActiveAreas;AreasCycle
\end{array}\right)
\end{array}\right)\\
\setminus\ GasDischargeSync
$$

$$
\begin{array}{l}
= [C.58, D.27, C.46] \\
\{\{manualDischarge, reset, detection\} \subseteq \Sigma_2\} \\
\{\{manualDischarge\} \cap \{reset, detection\} = \emptyset\} \\
\{\{manualDischarge\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Manual_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad \left(\begin{array}{l}
manualDischarge.area \rightarrow \\
\quad gasDischarged.area \rightarrow \\
\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \\
\quad\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad\quad \left(\begin{array}{l}
\Box\ area : AreaId \bullet \\
\quad manualDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas;AreasCycle \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad ActiveAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4, C.44] \\
\{\{manualDischarge, reset, detection\} \subseteq \Sigma_2\} \\
\{\{externalManualDischarge\} \cap \Sigma_2 = \emptyset\} \\
\{\{area\} \cap usedV() = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
systemState!manual_s \rightarrow \\
\quad \left(\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Manual_2
\end{array} \right) \right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad (ActiveAreas;AreasCycle) \\
\quad \Box \left(\left(\begin{array}{l}
externalManualDischarge?area : AreaId \rightarrow \\
\quad manualDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \right) \\
\quad\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad\quad \left(\begin{array}{l}
\Box\ area : AreaId \bullet \\
\quad manualDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas;AreasCycle \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad ActiveAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.19, C.30, C.33, C.36, C.57] \\
\left(\begin{array}{l}
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad switchBuzzer!on \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
externalManualDischarge?area : AreaId \rightarrow \\
\quad manualDischarge.area \rightarrow \\
\qquad gasDischarged.area \rightarrow \\
\qquad\quad switchLamp[AreaId].area!on \rightarrow \\
\qquad\qquad SwitchInternalSystem2DisabledMode;Reset_2 \\
\qquad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
\left(\begin{array}{l}
\Box\ area : AreaId \bullet \\
\quad manualDischarge.area \rightarrow \\
\qquad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\qquad\qquad DisabledAreas;AreasCycle \\
\qquad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\qquad ActiveAreas;AreasCycle \\
\Box\ reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad ActiveAreas;AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.24, D.24] \\
\left(\begin{array}{l}
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad switchBuzzer!on \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
externalManualDischarge?area : AreaId \rightarrow \\
\quad manualDischarge.area \rightarrow \\
\qquad gasDischarged.area \rightarrow \\
\qquad\quad switchLamp[AreaId].area!on \rightarrow \\
\qquad\qquad SwitchInternalSystem2DisabledMode;Reset_2 \\
\qquad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{mode_A = manual\}; \\
\left(\begin{array}{l}
\Box\ area : AreaId \bullet \\
\quad manualDischarge.area \rightarrow \\
\qquad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad gasDischarged.area \rightarrow ActivateDischargeAS; \\
\qquad\qquad DisabledAreas \\
\qquad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\qquad ActiveAreas \\
\Box\ reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad ActiveAreas
\end{array}\right); \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$ [$D.17, C.36, C.57, D.22, Definition\ of\ ActiveAreas$]
$\{mode_A = manual \Rightarrow mode_A = manual\}$
$\{mode_A = manual \Rightarrow \neg\ mode_A = automatic\}$

$$
\left(\begin{array}{l}
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
externalManualDischarge?area : AreaId \rightarrow \\
\quad manualDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right)
\end{array}\right)
\setminus GasDischargeSync
$$

$\sqsubseteq_{\mathcal{A}}$ [$C.46$]
$\{initials(ActiveAreas;AreasCycle) \subseteq \Sigma_2\}$

$$
\left(\begin{array}{l}
systemState!manual_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Manual_2 \\
\Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad manualDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right)
\end{array}\right)
\setminus GasDischargeSync
$$

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{initials(ActiveAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{systemState \notin \Sigma_2\} \\
\{wrtV(systemState!manual_s \rightarrow Skip) \cap usedV(ActiveAreas;AreasCycle) = \emptyset\} \\
\left(\left(\begin{array}{l}
systemState!manual_s \rightarrow \\
\quad detection?newZone : ZoneId \rightarrow \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\quad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow Manual_2 \\
\quad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\quad\quad manualDischarge.area \rightarrow \\
\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad\quad SwitchInternalSystem2DisabledMode;Reset_2 \\
\quad\quad\quad \Box\ gasNotDischarged.area \rightarrow Manual_2
\end{array} \right) \right) \\
\left(\begin{array}{l}
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
= [D.22,\, Definition\ \ of\ \ Manual_2] \\
\left(\begin{array}{l}
Manual_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Lemma B.6

$$
\begin{array}{l}
\{mode_A = automatic \wedge mode_1 = automatic\};Auto_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
\{mode_A = automatic \wedge mode_1 = automatic\};Auto_1[Act_1 \setminus Act_2] \\
= [D.14] \\
\{mode_A = automatic \wedge mode_1 = automatic \Rightarrow mode_A = automatic\} \\
\{mode_A = automatic\};Auto_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ Auto_1, Substitution] \\
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad alarm!secondStage \rightarrow \\
\qquad\quad \left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\qquad\quad \setminus GasDischargeSync \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_1;InitFireControl_1; \\
\qquad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\qquad\quad \setminus GasDischargeSync \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\qquad\qquad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\qquad\qquad \setminus GasDischargeSync \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow \\
\qquad\qquad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\qquad\qquad\quad \setminus GasDischargeSync
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$ $[D.7, D.29, D.28]$
$\{wrtV(InitAreas) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\}$
$\{wrtV(InitAreas) \cap usedV(FireSys_2) = \emptyset\}$
$\{wrtV(InitInternalSystem) \subseteq \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\}$
$\{wrtV(InitInternalSystem) \cap usedV(WaitingDischarge;AreasCycle) = \emptyset\}$
$\{mode_A = automatic\};$
$systemState!auto_s \rightarrow$
$(active \rhd \{true\} \neq \emptyset)$ &
$alarm!secondStage \rightarrow$
$\left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync$
$\Box$ $(active \rhd \{true\} = \emptyset)$ &
$reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_1;$
$\left(\begin{array}{l} (InitInternalSystem;FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \setminus GasDischargeSync$
$\Box$ $detection?newZone : ZoneId \rightarrow ActivateZone_1;$
$switchLamp[ZoneId].newZone!on \rightarrow$
$\left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \setminus GasDischargeSync$
$\Box$ $fault?faultId : FaultId \rightarrow$
$switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$switchBuzzer!on \rightarrow$
$\left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \setminus GasDischargeSync$

$$
\begin{array}{l}
= [Schema\ Calculus, Definition\ of\ SwitchLampsOff_2] \\
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad alarm!secondStage \rightarrow \\
\qquad\quad \left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge; AreasCycle) \end{array} \right) \\
\qquad\quad \setminus\ GasDischargeSync \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad\quad \left(\begin{array}{l} (InitInternalSystem; FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas; AreasCycle) \end{array} \right) \\
\qquad\quad \setminus\ GasDischargeSync \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\qquad\qquad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\qquad\qquad \setminus\ GasDischargeSync \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow \\
\qquad\qquad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas; AreasCycle) \end{array} \right) \\
\qquad\qquad\quad \setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.54, C.44] \\
\{initials(AreasCycle) \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{alarm, switchLamp, switchBuzzer\} = \emptyset\} \\
\{wrtV(SwitchLampsOff_2) \cap usedV(InitAreas;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\quad\quad alarm!secondStage \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad\quad reset \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} \left(\begin{array}{c} alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ InitInternalSystem;FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.51] \\
\{reset \in \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\quad\quad alarm!secondStage \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad\quad \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad InitInternalSystem;FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (reset \rightarrow InitAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\quad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad\quad \left(\begin{array}{l} Auto_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{initials(WaitingDischarge) \subseteq \Sigma_2\} \\
\{\{alarm, switchLamp, switchBuzzer, fault\} \cap \Sigma_2 = \emptyset\} \\
\{wrtV(alarm!secondStage \rightarrow Skip) \cap \\
\quad usedV(WaitingDischarge;AreasCycle) = \emptyset\} \\
\{initials(ActiveAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{wrtV(switchLamp[ZoneId].newZone!on \rightarrow Skip) \cap \\
\quad usedV(ActiveAreas;AreasCycle) = \emptyset\} \\
\{\{faultId\} \cap usedV(ActiveAreas;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \&\ \\
\qquad \left(\begin{array}{l} (alarm!secondStage \rightarrow Countdown_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \&\ \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \qquad InitInternalSystem;FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (reset \rightarrow InitAreas;AreasCycle) \end{array} \right) \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\qquad\quad \left(\begin{array}{l} (switchLamp[ZoneId].newZone!on \rightarrow Auto_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right) \\
\qquad\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \qquad\quad switchBuzzer!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [D.28] \\
\{wrtV(ActivateZoneAS) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(ActivateZoneAS) \cap \\
\quad usedV(switchLamp[ZoneId].newZone!on \rightarrow Auto_2) = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \&\ \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \&\ \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad\quad \left(\begin{array}{l}
(switchLamp[ZoneId].newZone!on \rightarrow Auto_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateZoneAS;ActiveAreas;AreasCycle)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.6] \\
\{detection \in \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.45] \\
\{\{detection, reset\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l} (alarm!secondStage \rightarrow Countdown_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad InitInternalSystem;FireSys_2 \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} reset \rightarrow InitAreas;AreasCycle \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad switchBuzzer!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActiveAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.13, D.19, C.30, C.36, C.57] \\
\{mode_A = automatic \Rightarrow mode_A = automatic\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ \{mode_A = automatic\}; \\
\quad\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\ InitInternalSystem;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad\ switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad\ ActivateZoneAS;ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad\ switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\ switchBuzzer!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [D.16, C.36, C.57] \\
\{mode_A = automatic \wedge active \rhd \{true\} = \emptyset \Leftrightarrow \\
\quad mode_A = automatic \wedge active \rhd \{true\} = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \{mode_A = automatic \wedge active \rhd \{true\} = \emptyset\}; \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad InitInternalSystem;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad switchBuzzer!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [C.30, C.33, C.36, C.57] \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad InitInternalSystem;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad switchBuzzer!on \rightarrow Auto_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{mode_A = automatic \wedge active \rhd \{true\} = \emptyset\}; \\
ActiveAreas; \\
AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$ [*Definition of ActiveAreas*, $D.17$, $C.36$, $C.57$]

$\{(mode_A = automatic \wedge active \rhd \{true\} = \emptyset) \Rightarrow (mode_A = automatic)\}$

$\{(mode_A = automatic \wedge active \rhd \{true\} = \emptyset) \Rightarrow \neg\, (mode_A = manual)\}$

$$
\left(\begin{array}{l}
\{mode_A = automatic\};\\
systemState!auto_s \rightarrow\\
\quad (active \rhd \{true\} \neq \emptyset)\ \&\\
\quad\quad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
(WaitingDischarge;AreasCycle)
\end{array}\right)\\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \&\\
\quad\quad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2;\\
\quad InitInternalSystem;FireSys_2\\
\Box\ detection?newZone : ZoneId \rightarrow\\
\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array}\right)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
\left(\begin{array}{l}
reset \rightarrow InitAreas;AreasCycle\\
\Box\ detection?newZone : ZoneId \rightarrow\\
\quad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array}\right)
\end{array}\right)\\
\quad\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow\\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow\\
\quad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}\right)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
\left(\begin{array}{l}
\{mode_A = automatic \wedge active \rhd \{true\} = \emptyset\};\\
\left(\begin{array}{l}
(mode_A = automatic)\ \&\\
\quad (active_1 \rhd \{true\} \neq \emptyset)\ \&\\
\quad\quad countdown \rightarrow\\
\quad\quad\quad countdownStarted?answer : Bool \rightarrow\\
\quad\quad\quad\quad (answer = true)\ \&\\
\quad\quad\quad\quad\quad WaitingDischarge\\
\quad\quad\quad\quad \Box\ (answer = false)\ \&\\
\quad\quad\quad\quad\quad DisabledAreas\\
\quad \Box\ (active_1 \rhd \{true\} = \emptyset)\ \&\\
\quad\quad reset \rightarrow InitAreas\\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow\\
\quad\quad\quad ActivateZoneAS;ActiveAreas\\
\quad\quad \Box\ \Box\ area : AreaId \bullet\\
\quad\quad\quad automaticDischarge.area \rightarrow\\
\quad\quad\quad\quad gasNotDischarged.area \rightarrow\\
\quad\quad\quad\quad\quad ActiveAreas
\end{array}\right);\\
AreasCycle
\end{array}\right)
\end{array}\right)\\
\setminus\ GasDischargeSync
\end{array}\right)
$$

$$
\begin{array}{l}
= [C.32, C.16, C.24] \\
\{(mode_A = automatic \wedge active \rhd \{true\} = \emptyset) \Rightarrow mode_A = automatic)\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l} (alarm!secondStage \rightarrow Countdown_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad InitInternalSystem;FireSys_2 \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} reset \rightarrow InitAreas;AreasCycle \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \{mode_A = automatic \wedge active \rhd \{true\} = \emptyset\}; \\ \left(\begin{array}{l} (active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad countdown \rightarrow \\ \quad\quad countdownStarted?answer : Bool \rightarrow \\ \quad\quad\quad (answer = true)\ \& \\ \quad\quad\quad\quad WaitingDischarge \\ \quad\quad\quad \Box\ (answer = false)\ \& \\ \quad\quad\quad\quad DisabledAreas \\ \Box\ (active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad reset \rightarrow InitAreas \\ \quad \Box\ detection?newZone : ZoneId \rightarrow \\ \quad\quad ActivateZoneAS;ActiveAreas \\ \quad \Box\ \Box\ area : AreaId \bullet \\ \quad\quad automaticDischarge.area \rightarrow \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right) \end{array} \right) \\
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$ [$D.17$, $C.36$, $C.57$]
$\{(mode_A = automatic \wedge active \rhd \{true\} = \emptyset) \Rightarrow active_1 \rhd \{true\} = \emptyset\}$
$\{(mode_A = automatic \wedge active \rhd \{true\} = \emptyset) \Rightarrow \neg\ (active_1 \rhd \{true\} \neq \emptyset)\}$

$$
\left(
\begin{array}{l}
\{mode_A = automatic\};\\
systemState!auto_s \rightarrow\\
\quad (active \rhd \{true\} \neq \emptyset)\ \&\\
\qquad \left(
\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
(WaitingDischarge;AreasCycle)
\end{array}
\right)\\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \&\\
\qquad \left(
\begin{array}{l}
\left(
\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2;\\
\qquad InitInternalSystem;FireSys_2\\
\Box\ detection?newZone : ZoneId \rightarrow\\
\qquad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array}
\right)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
\left(
\begin{array}{l}
reset \rightarrow InitAreas;AreasCycle\\
\Box\ detection?newZone : ZoneId \rightarrow\\
\qquad ActivateZoneAS;ActiveAreas;AreasCycle
\end{array}
\right)
\end{array}
\right)\\
\qquad \Box \left(
\begin{array}{l}
\left(
\begin{array}{l}
fault?faultId : FaultId \rightarrow\\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow\\
\qquad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}
\right)\\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!]\\
\left(
\begin{array}{l}
\left(
\begin{array}{l}
reset \rightarrow InitAreas\\
\Box\ detection?newZone : ZoneId \rightarrow\\
\qquad ActivateZoneAS;ActiveAreas\\
\Box\ \Box\ area : AreaId \bullet\\
\qquad automaticDischarge.area \rightarrow\\
\qquad\quad gasNotDischarged.area \rightarrow\\
\qquad\qquad ActiveAreas
\end{array}
\right);\\
AreasCycle
\end{array}
\right)
\end{array}
\right)
\end{array}
\right)
\setminus GasDischargeSync
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [C.58, D.27] \\
\{\{reset, detection, automaticDischarge\} \subseteq cs\Sigma_2\} \\
\{\{reset, detection\} \cap \{automaticDischarge\} = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l} (alarm!secondStage \rightarrow Countdown_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad InitInternalSystem;FireSys_2 \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} reset \rightarrow InitAreas;AreasCycle \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad \Box \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad InitInternalSystem;FireSys_2 \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\ \quad gasNotDischarged.area \rightarrow ActiveAreas; \\ \quad AreasCycle \end{array} \right) \end{array} \right) \\
\qquad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad switchBuzzer!on \rightarrow Auto_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow InitAreas \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas \\ \Box\ \Box\ area : AreaId \bullet \\ \quad automaticDischarge.area \rightarrow \\ \quad gasNotDischarged.area \rightarrow \\ \quad ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right) \end{array} \right) \end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [C.46] \\
\{\{reset, detection\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array}\right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad InitInternalSystem;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
reset \rightarrow InitAreas;AreasCycle \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;ActiveAreas;AreasCycle \\
\Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\qquad gasNotDischarged.area \rightarrow ActiveAreas; \\
\qquad\quad AreasCycle
\end{array}\right)
\end{array}\right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\qquad automaticDischarge.area \rightarrow \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\qquad ActiveAreas
\end{array}\right); \\
AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.24, D.24] \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array}\right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad InitInternalSystem;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\qquad automaticDischarge.area \rightarrow \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\qquad ActiveAreas
\end{array}\right); \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\qquad automaticDischarge.area \rightarrow \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\qquad ActiveAreas
\end{array}\right); \\
AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.46] \\
\{\{reset, detection, automaticDischarge\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
(alarm!secondStage \rightarrow Countdown_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array}\right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad InitInternalSystem;FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\qquad automaticDischarge.area \rightarrow \\
\qquad\quad gasNotDischarged.area \rightarrow \\
\qquad\quad\quad ActiveAreas
\end{array}\right);\right) \\
AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.35, \mathit{answer} \neq \mathit{true} \Rightarrow \mathit{answer} = \mathit{false}, C.54] \\
\{\mathit{countdownStarted} \in \Sigma_2\} \\
\{\mathit{countdownStarted} \in \mathit{GasDischargeSync}\} \\
\{\mathit{answer} \notin FV(\mathit{WaitingDischarge;AreasCycle})\} \\
\left(\begin{array}{l}
\{\mathit{mode}_A = \mathit{automatic}\}; \\
\mathit{systemState}!\mathit{auto}_s \rightarrow \\
\quad (\mathit{active} \rhd \{\mathit{true}\} \neq \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{countdownStarted}!\mathit{true} \rightarrow \mathit{alarm}!\mathit{secondStage} \rightarrow \\
\quad \mathit{Countdown}_2
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
\left(\begin{array}{l}
\mathit{countdownStarted}?\mathit{answer} : \mathit{Bool} \rightarrow \\
\quad (\mathit{answer} = \mathit{true})\ \&\ \mathit{WaitingDischarge;AreasCycle} \\
\quad \Box\ (\mathit{answer} = \mathit{false})\ \&\ \mathit{DisabledAreas;AreasCycle}
\end{array}\right)
\end{array}\right) \\
\quad \Box\ (\mathit{active} \rhd \{\mathit{true}\} = \emptyset)\ \& \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{reset} \rightarrow \mathit{alarm}!\mathit{alarmOff} \rightarrow \mathit{SwitchLampsOff}_2; \\
\quad \mathit{InitInternalSystem;FireSys}_2 \\
\Box\ \mathit{detection}?\mathit{newZone} : \mathit{ZoneId} \rightarrow \\
\quad \mathit{switchLamp}[\mathit{ZoneId}].\mathit{newZone}!\mathit{on} \rightarrow \mathit{Auto}_2 \\
\Box\ \mathit{fault}?\mathit{faultId} : \mathit{FaultId} \rightarrow \\
\quad \mathit{switchLamp}[\mathit{LampId}].\mathit{getLampId}(\mathit{faultId})!\mathit{on} \rightarrow \\
\quad\quad \mathit{switchBuzzer}!\mathit{on} \rightarrow \mathit{Auto}_2
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{reset} \rightarrow \mathit{InitAreas} \\
\Box\ \mathit{detection}?\mathit{newZone} : \mathit{ZoneId} \rightarrow \\
\quad \mathit{ActivateZoneAS;ActiveAreas} \\
\Box\ \Box\ \mathit{area} : \mathit{AreaId} \bullet \\
\quad \mathit{automaticDischarge}.\mathit{area} \rightarrow \\
\quad\quad \mathit{gasNotDischarged}.\mathit{area} \rightarrow \\
\quad\quad\quad \mathit{ActiveAreas}
\end{array}\right); \\
\mathit{AreasCycle}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ \mathit{GasDischargeSync}
\end{array}
$$

$$
\begin{array}{l}
= [D.37] \\
\{countdown \in \Sigma_2\} \\
\{countdown \in GasDischargeSync\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad (active \rhd \{true\} \neq \emptyset)\ \& \\
\quad\quad \left(\begin{array}{l}
\left(\begin{array}{l} countdown \rightarrow countdownStarted!true \rightarrow \\ \quad alarm!secondStage \rightarrow Countdown_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\ \quad (answer = true)\ \&\ WaitingDischarge;AreasCycle \\ \quad \Box\ (answer = false)\ \&\ DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad\quad \left(\begin{array}{l}
\left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad InitInternalSystem;FireSys_2 \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \\ \Box\ fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Auto_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} \left(\begin{array}{l} reset \rightarrow InitAreas \\ \Box\ detection?newZone : ZoneId \rightarrow \\ \quad ActivateZoneAS;ActiveAreas \\ \Box\ \Box\ area : AreaId \bullet \\ \quad automaticDischarge.area \rightarrow \\ \quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.11] \\
\{(\{reset, switchLamp, switchBuzzer, detection, fault\} \cup usedC(FireSys_2) \cup \\
\quad usedC(Auto_2)) \cap \{countdown\} = \emptyset\} \\
\{(\{reset, detection\} \cup usedC(ActiveAreas)) \cap \{countdown\} = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
systemState!auto_s \rightarrow \\
\quad \left(\begin{array}{l}
countdown \rightarrow countdownStarted!true \rightarrow \\
\quad alarm!secondStage \rightarrow Countdown_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad InitInternalSystem; FireSys_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Auto_2
\end{array} \right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
(active \rhd \{true\} \neq \emptyset)\ \& \\
\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad (answer = true)\ \&\ WaitingDischarge; AreasCycle \\
\quad\quad \Box\ (answer = false)\ \&\ DisabledAareas; AreasCycle \\
\Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad \left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS; ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad ActiveAreas
\end{array} \right); \\
\quad AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{\{countdown, reset, detection\} \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{systemState\} = \emptyset\} \\
\{wrtV(systemState!auto_s \rightarrow Skip) \cap A_3 = \emptyset\} \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
\quad \left(\begin{array}{l}
systemState!auto_s \rightarrow \\
\quad countdown \rightarrow countdownStarted!true \rightarrow \\
\quad\quad alarm!secondStage \rightarrow Countdown_2 \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad InitInternalSystem; FireSys_2 \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow Auto_2
\end{array}\right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
(active \rhd \{true\} \neq \emptyset)\ \& \\
\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad (answer = true)\ \&\ WaitingDischarge; AreasCycle \\
\quad\quad \Box\ (answer = false)\ \&\ DisabledAreas; AreasCycle \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad\quad \left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad ActivateZoneAS; ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad ActiveAreas
\end{array}\right); \\
\quad\quad AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$ [*Definition of* $Auto_2$, $C.33$, $C.36$, $C.57$]

$$
\left(\begin{array}{l}
Auto_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
(active \rhd \{true\} \neq \emptyset)\ \& \\
\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad (answer = true)\ \&\ WaitingDischarge; AreasCycle \\
\quad\quad \Box\ (answer = false)\ \&\ DisabledAreas; AreasCycle \\
\Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad \left(\begin{array}{l}
reset \rightarrow InitAreas \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad ActivateZoneAS; ActiveAreas \\
\Box\ \Box\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad ActiveAreas
\end{array}\right); \\
\quad AreasCycle
\end{array}\right)
\end{array}\right)
\setminus GasDischargeSync
$$

$=$ [$D.24$]

$$
\left(\begin{array}{l}
Auto_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{mode_A = automatic\}; \\
\left(\begin{array}{l}
(active \rhd \{true\} \neq \emptyset)\ \& \\
\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad (answer = true)\ \&\ WaitingDischarge \\
\quad\quad \Box\ (answer = false)\ \&\ DisabledAreas \\
\Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\quad reset \rightarrow InitAreas \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad ActivateZoneAS; ActiveAreas \\
\quad \Box\ \Box\ area : AreaId \bullet \\
\quad\quad automaticDischarge.area \rightarrow \\
\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad ActiveAreas
\end{array}\right); \\
AreasCycle
\end{array}\right)
\end{array}\right)
\setminus GasDischargeSync
$$

$\sqsubseteq_{\mathcal{A}}$ [$D.17, C.36, C.57$]
$\{mode_A = automatic \Rightarrow mode_A = automatic\}$
$\{mode_A = automatic \Rightarrow \neg\ mode_A = manual\}$

$$
\left(\begin{array}{l}
Auto_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(mode_A = automatic)\ \&\ \\
\quad (active \rhd \{true\} \neq \emptyset)\ \&\ \\
\quad\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\quad\quad\quad (answer = true)\ \&\ WaitingDischarge \\
\quad\quad\quad \Box\ (answer = false)\ \&\ DisabledAreas \\
\quad \Box\ (active \rhd \{true\} = \emptyset)\ \&\ \\
\quad\quad reset \rightarrow InitAreas \\
\quad\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad\quad ActivateZoneAS;ActiveAreas \\
\quad\quad \Box\ \Box\ area : AreaId \bullet \\
\quad\quad\quad automaticDischarge.area \rightarrow \\
\quad\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad\quad ActiveAreas \\
\Box\ (mode_A = manual)\ \&\ \\
\quad reset \rightarrow InitAreas \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad ActivateZoneAS;ActiveAreas \\
\quad \Box\ \Box\ area : AreaId \bullet \\
\quad\quad manualDischarge.area \rightarrow \\
\quad\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \&\ \\
\quad\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad\quad ActivateDischargeAS;\ DisabledAreas \\
\quad\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \&\ \\
\quad\quad\quad\quad gasNotDischarged.area \rightarrow ActiveAreas
\end{array}\right) \\
AreasCycle
\end{array}\right)
\end{array}\right)
$$

$\setminus\ GasDischargeSync$
$\widehat{=}$ [*Definition of ActiveAreas*]

$$
\left(\begin{array}{l}
Auto_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActiveAreas;AreasCycle)
\end{array}\right) \setminus GasDischargeSync
$$

Lemma B.7

$$
\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
\quad Reset_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\};Reset_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ \ Reset_1, Substitution] \\
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchFireControl2AutomaticMode_1; \\
\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_1; \\
\quad\quad (mode_1 = disabled)\ \& \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad\quad\quad\quad\quad\quad \setminus GasDischargeSync \\
\quad\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad\quad\quad InitFireControl_1; \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\quad\quad\quad\quad\quad\quad \setminus GasDischargeSync \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow switchBuzzer!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad\quad\quad \setminus GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [Schema\ Calculus, Definition\ of\ SwitchLampsOff_2, D.15] \\
\{(mode_A = manual \land mode_1 = manual) \lor (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\qquad SwitchInternalSystem2AutomaticMode; \\
\qquad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad\quad \setminus\ GasDischargeSync \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow \\
\qquad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad\quad \setminus\ GasDischargeSync \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad (mode_1 = disabled)\ \& \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\qquad\qquad \setminus\ GasDischargeSync \\
\qquad \Box\ (mode_1 \neq disabled)\ \& \\
\qquad\quad InitFireControl_1; \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ AreasCycle \end{array} \right) \\
\qquad\qquad \setminus\ GasDischargeSync \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow switchBuzzer!on \rightarrow \\
\qquad\quad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad\quad \setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.54] \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode; \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual) \; \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box \; (mode_1 \neq manual) \; \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \; detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual) \; \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box \; (mode_1 \neq manual) \; \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \; reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad (mode_1 = disabled) \; \& \\
\quad\quad\quad \left(\begin{array}{l}
FireSysD_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \\
\quad\quad \Box \; (mode_1 \neq disabled) \; \& \; InitFireControl_1; \\
\quad\quad\quad \left(\begin{array}{l}
FireSys_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right) \\
\quad \Box \; fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow switchBuzzer!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual) \; \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box \; (mode_1 \neq manual) \; \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus \; GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.7] \\
\{\alpha(InternalSystemState) \cap \alpha(AreasState) = \emptyset\} \\
\{FV(true) \subseteq \alpha(InternalSystemState)\} \\
\{FV(true) \subseteq \alpha(AreasState)\} \\
\{\{mode_1', dischargedOcurred_1'\} \subseteq \alpha(InternalSystemState')\} \\
\{\{mode_A', controlledZones_1', activeZones_1', discharge_1', active_1'\} \subseteq \\
\quad \alpha(AreasState')\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode; \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad (mode_1 = disabled)\ \& \\
\quad\quad\quad \left(\begin{array}{l}
FireSysD_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \\
\quad\quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;InitAreas; \\
\quad\quad\quad \left(\begin{array}{l}
FireSys_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
AreasCycle
\end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow switchBuzzer!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.28, D.29] \\
\{wrtV(InitAreas) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(InitAreas) \cap usedV(FireSys_2) = \emptyset\} \\
\{wrtV(InitInternalSystem) \subseteq \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(InitInternalSystem) \cap usedV(AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode; \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad (mode_1 = disabled)\ \& \\
\quad\quad\quad \left(\begin{array}{l}
FireSysD_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \\
\quad\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad\quad \left(\begin{array}{l}
(InitInternalSystem;FireSys_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow switchBuzzer!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.10] \\
\{initials(AreasCycles) \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode; \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad \left(\begin{array}{l}
(mode_1 = disabled)\ \&\ FireSysD_2 \\
\Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2
\end{array} \right) \\
\quad\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad\quad (InitAreas;AreasCycle) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.44, C.51] \\
\{initials(AreasCycles) \subseteq \Sigma_2\} \\
\{\{alarm, switchLamp, switchBuzzer\} \cap \Sigma_2 = \emptyset\} \\
\{wrtV(alarm!alarmOff \rightarrow SwitchLampsOff_2) \cap \\
\quad usedV(InitAreas;AreasCycle) = \emptyset\} \\
\{reset \in \Sigma_2\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode; \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad\quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{initials(ActiveAreas;AreasCycle) \cup \\
\quad initials(DisabledAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{\{fault, switchLamp, switchBuzzer\} \cap \Sigma_2 = \emptyset\} \\
\{\{faultId\} \cap (usedV(ActiveAreas;AreasCycle) \cup \\
\quad usedV(DisabledAreas;AreasCycle)) = \emptyset\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode; \\
\quad\quad\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad \left(\begin{array}{l}
(switchLamp[ZoneId].newZone!on \rightarrow Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad\quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.28] \\
\{wrtV(SwitchInternalSystem2AutomaticMode) \subseteq \\
\quad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(SwitchInternalSystem2AutomaticMode) \cap \\
\quad (usedV(ActiveAreas;AreasCycle) \cup \\
\qquad usedV(DisabledAreas;AreasCycle)) = \emptyset\} \\
\{wrtV(ActivateZoneAS) \subseteq \\
\quad \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(ActivateZoneAS) \cap \\
\quad usedV(switchLamp[ZoneId].newZone!on \rightarrow Reset_2) = \emptyset\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\qquad \left(\begin{array}{l}
(SwitchInternalSystem2AutomaticMode;\ Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\qquad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\qquad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad \left(\begin{array}{l}
(switchLamp[ZoneId].newZone!on \rightarrow Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
ActivateZoneAS; \\
\qquad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\qquad\quad ActiveAreas;AreasCycle \\
\qquad \Box\ (mode_1 \neq manual)\ \& \\
\qquad\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\qquad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\qquad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\qquad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{initials(ActiveAreas;AreasCycle) \cup \\
\quad initials(DisabledAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{\{switchLamp,\, actuatorsReplaced\} \cap \Sigma_2 = \emptyset\} \\
\{\{actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow Skip\} \cap \\
\quad (usedV(ActiveAreas;AreasCycle) \cup \\
\qquad usedV(DisabledAreas;AreasCycle)) = \emptyset\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l} actuatorsReplaced \rightarrow \\ \quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\ \quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad\quad \left(\begin{array}{l}
(switchLamp[ZoneId].newZone!on \rightarrow Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} ActivateZoneAS; \\ \quad (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \quad \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad (mode_1 = disabled)\ \&\ FireSysD_2 \\ \quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\qquad \Box \left(\begin{array}{l}
\left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Reset_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.6] \\
\{detection \in \Sigma_2 = \emptyset\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} actuatorsReplaced \rightarrow \\ \quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\ \quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\ \quad (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \quad \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad (mode_1 = disabled)\ \&\ FireSysD_2 \\ \quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(reset \rightarrow InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow Reset_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.22, C.45, C.58] \\
\{\{detection, reset\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle \\
\Box\ reset \rightarrow InitAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.58, D.26, C.45] \\
\{\{manualDischarge\} \subseteq \Sigma_2\} \\
\{\{detection, reset\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{(mode_A = manual \land mode_1 = manual) \lor (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\qquad \left(\left(\begin{array}{l} actuatorsReplaced \rightarrow \\ \qquad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\ \qquad\qquad SwitchInternalSystem2AutomaticMode;\ Reset_2 \end{array} \right) \right) \\
\qquad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\qquad \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \qquad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\ \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \qquad (mode_1 = disabled)\ \&\ FireSysD_2 \\ \qquad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2 \end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l} detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\ \qquad (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad\qquad ActiveAreas;AreasCycle \\ \qquad \Box\ (mode_1 \neq manual)\ \& \\ \qquad\qquad DisabledAreas;AreasCycle \\ \Box\ reset \rightarrow InitAreas;AreasCycle \\ \qquad \Box \left(\begin{array}{l} \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \qquad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \qquad\qquad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \qquad\qquad\qquad gasDischarged.area \rightarrow \\ \qquad\qquad\qquad\qquad ActivateDischargeAS;\ DisabledAreas \\ \qquad\qquad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \qquad\qquad\qquad gasNotDischarged.area \rightarrow ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right) \end{array} \right) \\
\Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \qquad\qquad switchBuzzer!on \rightarrow Reset_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \&\ ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.22, C.46] \\
\{initials(ActiveAreas;AreasCycle) \cup \\
\quad initials(DisabledAreas;AreasCycle) \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
systemState!reset_s \rightarrow \\
\quad \left(\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array} \right) \right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right) \\
\quad \left(\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array} \right) \right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \Box \left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad ActiveAreas;AreasCycle \\
\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad DisabledAreas;AreasCycle \\
\Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad \Box \left(\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad ActiveAreas
\end{array} \right) ; \right) \\
AreasCycle
\end{array} \right) \right) \\
\end{array} \right) \\
\setminus\ GasDischargeSync
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.19, C.30, C.33, C.36, C.57] \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\{(mode_A = manual \wedge mode_1 = manual) \vee (mode_1 \neq manual)\}; \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad\quad\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas \\
\quad\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad\quad ActiveAreas
\end{array}\right); \\
AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [C.13, C.36, C.57] \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \land mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
((mode_A = manual \land mode_1 = manual) \lor (mode_1 \neq manual))\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \land mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
(mode_A = manual \land mode_1 = manual)\ \& \\
\quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad\quad ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad\quad ActiveAreas
\end{array}\right) \\
AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

We refine the right-hand side action in the parallelism of the second alternative branch. In the following, we present only the refinement of this action.

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [D.12] \\
(mode_A = manual \land mode_1 = manual)\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \land mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas; AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas; AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas; AreasCycle \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad\quad ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right) \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \land mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas; AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas; AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas; AreasCycle \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad\quad ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right)
\end{array}
$$

$= [D.15]$

$(mode_A = manual \land mode_1 = manual)\ \&\ \{mode_A = manual \land mode_1 = manual\}$

$detection?newZone : ZoneId \rightarrow ActivateZoneAS;$

$(mode_A = manual \land mode_1 = manual)\ \&$

$ActiveAreas;AreasCycle$

$\Box\ (mode_1 \neq manual)\ \&$

$DisabledAreas;AreasCycle$

$\Box\ reset \rightarrow InitAreas;AreasCycle$

$$\Box \left(\begin{array}{l} \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad\quad ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas \end{array} \right) ; \\ AreasCycle \end{array} \right)$$

$\Box\ (mode_1 \neq manual)\ \&\ \{mode_1 \neq manual\}$

$detection?newZone : ZoneId \rightarrow ActivateZoneAS;$

$(mode_A = manual \land mode_1 = manual)\ \&$

$ActiveAreas;AreasCycle$

$\Box\ (mode_1 \neq manual)\ \&$

$DisabledAreas;AreasCycle$

$\Box\ reset \rightarrow InitAreas;AreasCycle$

$$\Box \left(\begin{array}{l} \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad\quad ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas \end{array} \right) ; \\ AreasCycle \end{array} \right)$$

$\sqsubseteq_{\mathcal{A}}$ [$C.30$, $C.36$, $C.57$, $D.20$, $D.21$]
$\{newZone \notin \{mode_A, mode_1\}\}$
$\{mode_A = manual \land mode_1 = manual \land mode'_1 = mode_1 \land mode'_A = mode_A \Rightarrow$
$\quad mode'_A = manual \land mode'_1 = manual\}$
$\{mode_1 \neq manual \land mode'_1 = mode_1 \Rightarrow mode'_1 \neq manual\}$
$(mode_A = manual \land mode_1 = manual)$ &
$\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS;$
$\quad\quad (mode_A = manual \land mode_1 = manual)$ &
$\quad\quad\quad ActiveAreas;AreasCycle$
$\quad\quad \Box\ (mode_1 \neq manual)$ &
$\quad\quad\quad DisabledAreas;AreasCycle$
$\quad \Box\ reset \rightarrow InitAreas;AreasCycle$
$\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} \{mode_A = manual \land mode_1 = manual\}; \\ (mode_A = manual \land mode_1 = manual)\ \& \\ \quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad\quad ActivateDischargeAS; \\ \quad\quad\quad\quad DisabledAreas \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad\quad ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right)$
$\Box\ (mode_1 \neq manual)$ &
$\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS;$
$\quad\quad (mode_A = manual \land mode_1 = manual)$ & $ActiveAreas;AreasCycle$
$\quad\quad \Box\ (mode_1 \neq manual)$ & $DisabledAreas;AreasCycle$
$\quad \Box\ reset \rightarrow InitAreas;AreasCycle$
$\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} \{mode_1 \neq manual\}; \\ (mode_A = manual \land mode_1 = manual)\ \& \\ \quad \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad\quad ActivateDischargeAS;\ DisabledAreas \\ \quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad\quad gasNotDischarged.area \rightarrow ActiveAreas \end{array} \right); \\ AreasCycle \end{array} \right)$

$\sqsubseteq_{\mathcal{A}}$ [$D.17$, $C.36$, $C.57$, $C.32$, $C.31$, $C.58$]
$\{mode_A = manual \land mode_1 = manual \Rightarrow mode_A = manual \land mode_1 = manual\}$
$\{mode_A = manual \land mode_1 = manual \Rightarrow \neg\ mode_1 \neq manual\}$
$\{mode_1 \neq manual \Rightarrow mode_1 \neq manual\}$
$\{mode_1 \neq manual \Rightarrow \neg\ (mode_A = manual \land mode_1 = manual)\}$
$(mode_A = manual \land mode_1 = manual)$ &
$detection?newZone : ZoneId \rightarrow ActivateZoneAS;$
$(mode_A = manual \land mode_1 = manual)$ &
$ActiveAreas;AreasCycle$
$\Box\ (mode_1 \neq manual)$ &
$DisabledAreas;AreasCycle$
$\Box\ reset \rightarrow InitAreas;AreasCycle$
$\Box \left(\begin{array}{l} \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\ \quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad gasDischarged.area \rightarrow \\ \quad\quad\quad ActivateDischargeAS; \\ \quad\quad\quad DisabledAreas \\ \quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\ \quad\quad gasNotDischarged.area \rightarrow \\ \quad\quad\quad ActiveAreas \end{array} \right);$
$AreasCycle$
$\Box\ (mode_1 \neq manual)$ &
$detection?newZone : ZoneId \rightarrow ActivateZoneAS;$
$(mode_A = manual \land mode_1 = manual)$ &
$ActiveAreas;AreasCycle$
$\Box\ (mode_1 \neq manual)$ &
$DisabledAreas;AreasCycle$
$\Box\ reset \rightarrow InitAreas;AreasCycle$

Now, we return to the refinement of the whole action.

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [C.13, C.36, C.57] \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad\quad \Box \left(\begin{array}{l}
\Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad ActivateDischargeAS;\ DisabledAreas \\
\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasNotDischarged.area \rightarrow ActiveAreas
\end{array}\right); \\
\quad\quad AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [B.4] \\
\{\{detection, reset\} \subseteq cs \\
\{\{detection, reset\} \cap \{automaticDischarge\} = \emptyset \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array} \right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;FireSys_2
\end{array} \right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad \Box \left(\begin{array}{l}
\Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad ActivateDischargeAS;\ DisabledAreas \\
\quad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow ActiveAreas
\end{array} \right); \\
\quad AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad \Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\quad\quad gasNotDischarged.area \rightarrow ActiveAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.24, D.24, \mathit{Definition\ of\ DisabledAreas}] \\
\left(\begin{array}{l}
\mathit{systemState!reset_s} \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{actuatorsReplaced} \rightarrow \\
\quad \mathit{switchLamp[LampId].circuitFaultLamp!off} \rightarrow \\
\quad\quad \mathit{SwitchInternalSystem2AutomaticMode;\ Reset_2} \\
\Box\ \mathit{fault?faultId : FaultId} \rightarrow \\
\quad \mathit{switchLamp[LampId].getLampId(faultId)!on} \rightarrow \\
\quad\quad \mathit{switchBuzzer!on} \rightarrow \mathit{Reset_2}
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
\left(\begin{array}{l}
(\mathit{mode_A} = \mathit{manual} \wedge \mathit{mode_1} = \mathit{manual})\ \&\ \\
\quad \mathit{ActiveAreas;AreasCycle} \\
\Box\ (\mathit{mode_1} \neq \mathit{manual})\ \&\ \mathit{DisabledAreas;AreasCycle}
\end{array}\right)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{detection?newZone : ZoneId} \rightarrow \\
\quad \mathit{switchLamp[ZoneId].newZone!on} \rightarrow \mathit{Reset_2} \\
\Box\ \mathit{reset} \rightarrow \mathit{alarm!alarmOff} \rightarrow \mathit{SwitchLampsOff_2;} \\
\quad (\mathit{mode_1} = \mathit{disabled})\ \&\ \mathit{FireSysD_2} \\
\quad \Box\ (\mathit{mode_1} \neq \mathit{disabled})\ \& \\
\quad\quad \mathit{InitInternalSystem;FireSys_2}
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
\left(\begin{array}{l}
(\mathit{mode_A} = \mathit{manual} \wedge \mathit{mode_1} = \mathit{manual})\ \& \\
\quad \mathit{detection?newZone : ZoneId} \rightarrow \mathit{ActivateZoneAS;} \\
\quad\quad (\mathit{mode_A} = \mathit{manual} \wedge \mathit{mode_1} = \mathit{manual})\ \& \\
\quad\quad\quad \mathit{ActiveAreas;AreasCycle} \\
\quad\quad \Box\ (\mathit{mode_1} \neq \mathit{manual})\ \& \\
\quad\quad\quad \mathit{DisabledAreas;AreasCycle} \\
\quad \Box\ \mathit{reset} \rightarrow \mathit{InitAareas;AreasCycle} \\
\quad\quad \Box \left(\begin{array}{l}
\Box\ \mathit{area : AreaId} \bullet \mathit{manualDischarge.area} \rightarrow \\
\quad (\mathit{area} \in \mathrm{dom}\ \mathit{active_1} \rhd \{\mathit{true}\})\ \& \\
\quad\quad \mathit{gasDischarged.area} \rightarrow \\
\quad\quad\quad \mathit{ActivateDischargeAS;} \\
\quad\quad\quad\quad \mathit{DisabledAreas} \\
\quad\quad \Box\ (\mathit{area} \notin \mathrm{dom}\ \mathit{active_1} \rhd \{\mathit{true}\})\ \& \\
\quad\quad\quad \mathit{gasNotDischarged.area} \rightarrow \\
\quad\quad\quad\quad \mathit{ActiveAreas}
\end{array}\right) ; \\
\quad\quad \mathit{AreasCycle} \\
\Box\ (\mathit{mode_1} \neq \mathit{manual})\ \&\ \mathit{DisabledAreas;AreasCycle}
\end{array}\right)
\end{array}\right) \\
\setminus\ \mathit{GasDischargeSync}
\end{array}
$$

$$
\begin{array}{l}
= [D.15] \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\{mode_A = manual \wedge mode_1 = manual\}; \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad (mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAareas;AreasCycle \\
\quad\quad \Box \left(\begin{array}{l}
\Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas \\
\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad ActiveAreas
\end{array}\right); \\
\quad AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$\sqsubseteq_{\mathcal{A}}$ [$D.13$, $C.30$, $C.36$, $C.57$]

$\{mode_A = manual \land mode_1 = manual \Rightarrow mode_A = manual \land mode_1 = manual\}$

$$
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \land mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \land mode_1 = manual)\ \& \\
\{mode_A = manual \land mode_1 = manual\}; \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad \{mode_A = manual \land mode_1 = manual\}; \\
\quad\quad (mode_A = manual \land mode_1 = manual)\ \& \\
\quad\quad\quad ActiveAreas;AreasCycle \\
\quad\quad \Box\ (mode_1 \neq manual)\ \& \\
\quad\quad\quad DisabledAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad\quad \Box \left(\begin{array}{l}
\Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas \\
\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad ActiveAreas
\end{array}\right); \\
\quad AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}\right)
$$

$\sqsubseteq_{\mathcal{A}}$ [D.17]

$\{mode_A = manual \wedge mode_1 = manual \Rightarrow mode_A = manual \wedge mode_1 = manual\}$

$\{mode_A = manual \wedge mode_1 = manual \Rightarrow \neg\, (mode_1 \neq manual)\}$

$$
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\{mode_A = manual \wedge mode_1 = manual\}; \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad\quad ActiveAreas;AreasCycle \\
\quad \Box\ reset \rightarrow InitAreas;AreasCycle \\
\quad\quad \Box \left(\begin{array}{l}
\Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad ActivateDischargeAS; \\
\quad\quad\quad\quad DisabledAreas \\
\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasNotDischarged.area \rightarrow \\
\quad\quad\quad ActiveAreas
\end{array}\right); \\
\quad\quad AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}\right)
$$

$\sqsubseteq_{\mathcal{A}}$ [$D.17, C.36, C.57, D.24, Definition\ of\ ActiveAreas$]
$\{mode_A = manual \wedge mode_1 = manual \Rightarrow mode_A = manual\}$
$\{mode_A = manual \wedge mode_1 = manual \Rightarrow \neg\ mode_A = automatic\}$

$$
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
$$

$$
\begin{array}{l}
= [C.58, D.27] \\
\{\{countdown\} \cup initials(ActiveAreas;AreasCycles) \\
\quad \cup initials(DisabledAreas;AreasCycles) \subseteq \Sigma_2\} \\
\{\{countdown\} \cap (initials(ActiveAreas;AreasCycles) \\
\quad \cup initials(DisabledAreas;AreasCycles)) = \emptyset\} \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
(countdown \rightarrow countdownStarted!false \rightarrow Reset_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.46] \\
\{initials(ActiveAreas;AreasCycle) \cup \\
\quad initials(DisabledAreas;AreasCycle) \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Reset_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad InitInternalSystem;FireSys_2 \\
\Box\ countdown \rightarrow countdownStarted!false \rightarrow Reset_2
\end{array} \right) \\
\|[\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\| \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4,\, C.44,\, D.22] \\
\{initials(ActiveAreas;AreasCycle) \cup \\
\quad initials(ActiveAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap systemState = \emptyset\} \\
\{wrtV(systemState!reset_s \rightarrow Skip) \cap (usedV(ActiveAreas;AreasCycle) \cup \\
\quad usedV(DisabledAreas;AreasCycle)) = \emptyset\} \\
\left(\left(\left(\begin{array}{l}
systemState!reset_s \rightarrow \\
\quad actuatorsReplaced \rightarrow \\
\quad\quad switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\quad\quad\quad SwitchInternalSystem2AutomaticMode;\ Reset_2 \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\quad\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_2 \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad (mode_1 = disabled)\ \&\ FireSysD_2 \\
\quad\quad \Box\ (mode_1 \neq disabled)\ \& \\
\quad\quad\quad InitInternalSystem;FireSys_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow Reset_2 \\
\quad \Box\ countdown \rightarrow countdownStarted!false \rightarrow Reset_2
\end{array} \right) \right) \right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \&\ DisabledAreas;AreasCycle
\end{array} \right) \\
\setminus\ GasDischargeSync \\
= [Definition\ of\ Reset_2] \\
\left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
(mode_A = manual \wedge mode_1 = manual)\ \& \\
\quad ActiveAreas;AreasCycle \\
\Box\ (mode_1 \neq manual)\ \& \\
\quad DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \setminus\ GasDischargeSync
\end{array}
$$

Lemma B.8

$$
\begin{array}{l}
Countdown_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
Countdown_1 \\
= [Definition\ of\ \ Countdown_1, Substitution] \\
systemState!countdown_s \rightarrow startClock \rightarrow \\
\qquad \left(\begin{array}{l} WaitingClock_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync \\
= [D.4, C.54, C.44] \\
\{\{detection, automaticDischarge\} \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{systemState, startClock\} = \emptyset\} \\
\{\emptyset \cap usedV(WaitingDischarge;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l} (systemState!countdown_s \rightarrow startClock \rightarrow \\ \qquad WaitingClock_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync \\
= [Definition\ of\ \ Countdown_2] \\
\left(\begin{array}{l} Countdown_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Lemma B.9

$$
\begin{array}{l}
WaitingClock_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} WaitingClock_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
WaitingClock_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ \ WaitingClock_1, Substitution] \\
clockFinished \rightarrow \\
\quad \left(\begin{array}{l} Discharge_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \setminus GasDischargeSync \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\qquad \left(\begin{array}{l} WaitingClock_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\qquad \setminus GasDischargeSync \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad switchBuzzer!on \rightarrow \\
\qquad\quad \left(\begin{array}{l} WaitingClock_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\qquad\quad \setminus GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [Schema\ \ Calculus, C.54] \\
\{\{clockFinished, detection, switchLamp, fault, switchBuzzer\} \cap \\
\quad GasDischargeSync = \emptyset\} \\
\left(\begin{array}{l}
clockFinished \rightarrow \\
\quad \left(\begin{array}{l} Discharge_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge ; AreasCycle) \end{array} \right) \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS ; \\
\quad switchLamp[ZoneId].newZone!on \rightarrow \\
\quad\quad \left(\begin{array}{l} WaitingClock_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge ; AreasCycle) \end{array} \right) \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad \left(\begin{array}{l} WaitingClock_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge ; AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.4,\, C.44] \\
\{\{automaticDischarge\} \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{clockFinished\} = \emptyset\} \\
\{wrtV(clockFinished \rightarrow Skip) \cap \\
\qquad usedV(WaitingDischarge;AreasCycle) = \emptyset\} \\
\{usedC(WaitingDischarge;AreasCycle) \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{switchLamp\} = \emptyset\} \\
\{wrtV(switchLamp[ZoneId].newZone!on \rightarrow Skip) \cap \\
\qquad usedV(WaitingDischarge;AreasCycle) = \emptyset\} \\
\{\Sigma_2 \cap \{fault, switchLamp, switchBuzzer\} = \emptyset\} \\
\{\{faultId\} \cap usedV(WaitingDischarge;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(clockFinished \rightarrow Discharge_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\qquad \left(\begin{array}{l}
(switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus GasDischargeSync \\
\sqsubseteq_{\mathcal{A}} [D.28] \\
\{wrtV(ActivateZoneAS) \subseteq (\alpha(AreasState) \cup \alpha(AreasState'))\} \\
\{wrtV(ActivateZoneAS) \cap \\
\qquad usedV(switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2) = \emptyset\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(clockFinished \rightarrow Discharge_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad \left(\begin{array}{l}
(switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateZoneAS;WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.6] \\
\{detection \in \Sigma_2\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(clockFinished \rightarrow Discharge_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;WaitingDischarge;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus GasDischargeSync \\
= [C.58, D.27] \\
\{\{detection, automaticDischarge\} \subseteq \Sigma_2\} \\
\{\{detection\} \cap \{automaticDischarge\} = \emptyset\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(clockFinished \rightarrow Discharge_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad ActivateZoneAS;WaitingDischarge;AreasCycle
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ReplyDischarge;DisabledAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow WaitingClock_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.46] \\
\{\{detection\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
\left(\begin{array}{l} (clockFinished \rightarrow Discharge_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \qquad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \qquad ActivateZoneAS;WaitingDischarge;AreasCycle \\ \Box\ ReplyDischarge;DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \qquad\quad switchBuzzer!on \rightarrow WaitingClock_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
= [C.24, D.24] \\
\left(\begin{array}{l}
\left(\begin{array}{l} (clockFinished \rightarrow Discharge_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \qquad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} detection?newZone : ZoneId \rightarrow \\ \qquad ActivateZoneAS;WaitingDischarge \\ \Box\ ReplyDischarge;DisabledAreas \end{array} \right); \\ AreasCycle \end{array} \right) \end{array} \right) \\
\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \qquad\quad switchBuzzer!on \rightarrow WaitingClock_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [\mathit{Definition\ of\ \ WaitingDischarge}] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(\mathit{clockFinished} \rightarrow \mathit{Discharge}_2) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{detection}?\mathit{newZone} : \mathit{ZoneId} \rightarrow \\
\qquad \mathit{switchLamp}[\mathit{ZoneId}].\mathit{newZone}!\mathit{on} \rightarrow \mathit{WaitingClock}_2
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{fault}?\mathit{faultId} : \mathit{FaultId} \rightarrow \\
\qquad \mathit{switchLamp}[\mathit{LampId}].\mathit{getLampId}(\mathit{faultId})!\mathit{on} \rightarrow \\
\qquad\quad \mathit{switchBuzzer}!\mathit{on} \rightarrow \mathit{WaitingClock}_2
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right)
\end{array}\right) \\
\setminus\ \mathit{GasDischargeSync} \\
= [C.58, D.27] \\
\{\{\mathit{countdown}\} \cup \mathit{initials}(\mathit{WaitingDischarge}; \mathit{AreasCycles}) \subseteq \Sigma_2\} \\
\{\{\mathit{countdown}\} \cap \mathit{initials}(\mathit{WaitingDischarge}; \mathit{AreasCycles}) = \emptyset\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
(\mathit{clockFinished} \rightarrow \mathit{Discharge}_2) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{detection}?\mathit{newZone} : \mathit{ZoneId} \rightarrow \\
\qquad \mathit{switchLamp}[\mathit{ZoneId}].\mathit{newZone}!\mathit{on} \rightarrow \mathit{WaitingClock}_2
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right) \\
\Box \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{fault}?\mathit{faultId} : \mathit{FaultId} \rightarrow \\
\qquad \mathit{switchLamp}[\mathit{LampId}].\mathit{getLampId}(\mathit{faultId})!\mathit{on} \rightarrow \\
\qquad\quad \mathit{switchBuzzer}!\mathit{on} \rightarrow \mathit{WaitingClock}_2
\end{array}\right) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right) \\
\Box \left(\begin{array}{l}
(\mathit{countdown} \rightarrow \mathit{countdownStarted}!\mathit{true} \rightarrow \mathit{WaitingClock}_2) \\
[\![\alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState})]\!] \\
(\mathit{WaitingDischarge}; \mathit{AreasCycle})
\end{array}\right)
\end{array}\right) \\
\setminus\ \mathit{GasDischargeSync}
\end{array}
$$

$$
\begin{array}{l}
= [C.46] \\
\{initials(WaitingDischarge;AreasCycle) \subseteq \Sigma_2\} \\
\left(\left(\begin{array}{l}
clockFinished \rightarrow Discharge_2 \\
\Box\ detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow WaitingClock_2 \\
\Box\ countdown \rightarrow countdownStarted!true \rightarrow WaitingClock_2
\end{array} \right) \right) \\
\left(\begin{array}{l}
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
\sqsubseteq_{\mathcal{A}} [Definition\ of\ \ WaitingClock_2] \\
\left(\begin{array}{l}
WaitingClock_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge;AreasCycle)
\end{array} \right) \setminus\ GasDischargeSync
\end{array}
$$

Lemma B.10

$$
\begin{array}{l}
FireSysD_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
FireSysD_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ FireSysD_1, Substitutions] \\
systemState!fireSysD_s \rightarrow \\
\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_1; \\
\quad\quad InitFireControl_1; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (AreasCycle) \end{array} \right) \\
\quad\quad \setminus GasDischargeSync \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad\quad\quad\quad \setminus GasDischargeSync \\
= [Definition\ of\ SwitchLampsOff_2] \\
systemState!fireSysD_s \rightarrow \\
\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad InitFireControl_1; \\
\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (AreasCycle) \end{array} \right) \\
\quad\quad \setminus GasDischargeSync \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad\quad\quad\quad \setminus GasDischargeSync
\end{array}
$$

$= [D.7]$
$\{\alpha(InternalSystemState) \cap \alpha(AreasState) = \emptyset\}$
$\{FV(true) \subseteq \alpha(InternalSystemState)\}$
$\{FV(true) \subseteq \alpha(AreasState)\}$
$\{\{mode'_1, dischargedOcurred'_1\} \subseteq \alpha(InternalSystemState')\}$
$\{\{mode'_A, controlledZones'_1, activeZones'_1, discharge'_1, active'_1\} \subseteq$
$\quad \alpha(AreasState')\}$
$systemState!fireSysD_s \rightarrow$
$\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2;$
$\quad\quad InitInternalSystem;InitAreas;$
$\quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (AreasCycle) \end{array} \right)$
$\quad\quad \setminus GasDischargeSync$
$\quad \Box\ fault?faultId : FaultId \rightarrow$
$\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$\quad\quad\quad switchBuzzer!on \rightarrow$
$\quad\quad\quad\quad \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right)$
$\quad\quad\quad\quad \setminus GasDischargeSync$
$= [C.54]$
$\{\{systemState, detection, switchLamp, alarm,$
$\quad actuatorsReplaced, fault, switchBuzzer\} \cap gasDischargeSync = \emptyset\}$
$\left(\begin{array}{l} systemState!fireSysD_s \rightarrow \\ \quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\ \quad\quad InitInternalSystem;InitAreas; \\ \quad\quad \left(\begin{array}{l} FireSys_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (AreasCycle) \end{array} \right) \\ \quad \Box\ fault?faultId : FaultId \rightarrow \\ \quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad\quad switchBuzzer!on \rightarrow \\ \quad\quad\quad\quad \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \end{array} \right)$
$\setminus GasDischargeSync$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}} [D.28, D.29] \\
\{wrtV(InitAreas) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\} \\
\{wrtV(InitAreas) \cap usedV(FireSys_2) = \emptyset\} \\
\{wrtV(InitInternalSystem) \subseteq \\
\quad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(InitInternalSystem) \cap usedV(AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSysD_s \rightarrow \\
\quad actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\quad\quad \left(\begin{array}{l} (InitInternalSystem;FireSys_2) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow \\
\quad\quad\quad\quad \left(\begin{array}{l} FireSysD_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync = [D.4, C.44] \\
\{initials(InitAreas;AreasCycle) \subseteq \Sigma_2\} \\
\{\{actuatorsReplaced, switchLamp, switchBuzzer, alarm\} \cap \Sigma_2 = \emptyset\} \\
\{wrtV(actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2) \cap \\
\quad usedV(InitAreas;AreasCycle) = \emptyset\} \\
\{\{fault, switchLamp, switchBuzzer\} \cap \Sigma_2 = \emptyset\} \\
\{wrtV(fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow Skip) \cap usedV(InitAreas;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSysD_s \rightarrow \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow \\ \quad SwitchLampsOff_2;InitInternalSystem;FireSys_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right) \\
\quad \Box \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} fault?faultId : FaultId \rightarrow \\ \quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\ \quad\quad switchBuzzer!on \rightarrow FireSysD_2 \end{array} \right) \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (InitAreas;AreasCycle) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.58, D.27] \\
\{\{countdown\} \cup initials(AreasCycle) \subseteq \Sigma_2\} \\
\{\{countdown\} \cap initials(AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!fireSysD_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow \\
\quad SwitchLampsOff_2;InitInternalSystem;FireSys_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSysD_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
(countdown \rightarrow countdownStarted!false \rightarrow FireSysD_2) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus GasDischargeSync \\
= [C.46] \\
\{initials(InitAreas;AreasCycle) \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
systemState!fireSysD_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow \\
\quad SwitchLampsOff_2;InitInternalSystem;FireSys_2 \\
\Box\ fault?faultId : FaultId \rightarrow \\
\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSysD_2 \\
\Box\ countdown \rightarrow \\
\quad countdownStarted!false \rightarrow FireSysD_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus GasDischargeSync \\
= [Definition\ of\ FireSysD_2] \\
\left(\begin{array}{l}
FireSysD_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(InitAreas;AreasCycle)
\end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Lemma B.11

$$
\begin{array}{l}
Discharge_1[Act_1 \setminus Act_2] \\
\sqsubseteq_{\mathcal{A}} \\
\left(\begin{array}{l} Discharge_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (WaitingDischarge;AreasCycle) \end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Proof.

$$
\begin{array}{l}
Discharge_1[Act_1 \setminus Act_2] \\
= [Definition\ of\ Discharge_1, Substitution] \\
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l} \fatsemi\ area : \mathrm{dom}\ active_1 \rhd \{true\} \bullet \\ \qquad switchLamp[AreaId].area!on \rightarrow Skip \end{array} \right); \\
\qquad \left(\begin{array}{l} (\mathrm{dom}\ active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \qquad SwitchFireControlSystem2DisabledMode_1 \\ \Box\ (\mathrm{dom}\ active_1 \rhd \{true\} = \emptyset)\ \& \\ \qquad SwitchFireControlSystem2AutomaticMode_1 \end{array} \right); \\
\qquad ActivateDischarge_1; \\
\qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \qquad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \qquad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right) \\
\qquad \setminus GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [Schema\ \ Calculus, C.54] \\
\{\{systemState, exit, switchLamp\} \cap GasDischargeSync = \emptyset\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l} {}^{9}_{9}\ area : \mathrm{dom}\ active_1 \rhd \{true\} \bullet \\ \quad switchLamp[AreaId].area!on \rightarrow Skip \end{array} \right); \\
\qquad \left(\begin{array}{l} (\mathrm{dom}\ active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad SwitchInternalSystem2DisabledMode \\ \Box\ (\mathrm{dom}\ active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad SwitchInternalSystem2AutomaticMode \end{array} \right); \\
\qquad ActivateDischargeAS; \\
\qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
= [C.35] \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l} {}^{9}_{9}\ area : \mathrm{dom}\ active_1 \rhd \{true\} \bullet \\ \quad switchLamp[AreaId].area!on \rightarrow Skip \end{array} \right); \\
\qquad \left(\begin{array}{l} (\mathrm{dom}\ active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad SwitchInternalSystem2DisabledMode; \\ \quad \{mode_1 = disabled\} \\ \Box\ (\mathrm{dom}\ active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad SwitchInternalSystem2AutomaticMode; \\ \quad \{mode_1 = automatic\} \end{array} \right); \\
\qquad ActivateDischargeAS; \\
\qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [D.14] \\
\{mode_1 = disabled \Rightarrow mode_1 \neq manual\} \\
\{mode_1 = automatic \Rightarrow mode_1 \neq manual\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l} {}^{\circ}_{9}\ area : \mathrm{dom}\ active_1 \rhd \{true\} \bullet \\ \quad switchLamp[AreaId].area!on \rightarrow Skip \end{array} \right); \\
\qquad \left(\begin{array}{l} (\mathrm{dom}\ active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad SwitchInternalSystem2DisabledMode; \\ \quad \{mode_1 \neq manual\} \\ \Box\ (\mathrm{dom}\ active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad SwitchInternalSystem2AutomaticMode; \\ \quad \{mode_1 \neq manual\} \end{array} \right); \\
\qquad ActivateDischargeAS; \\
\qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
=\ [D.24] \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l} {}^{\circ}_{9}\ area : \mathrm{dom}\ active_1 \rhd \{true\} \bullet \\ \quad switchLamp[AreaId].area!on \rightarrow Skip \end{array} \right); \\
\qquad \left(\begin{array}{l} (\mathrm{dom}\ active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad SwitchInternalSystem2DisabledMode \\ \Box\ (\mathrm{dom}\ active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad SwitchInternalSystem2AutomaticMode \end{array} \right); \\
\qquad \{mode_1 \neq manual\}; \\
\qquad ActivateDischargeAS; \\
\qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \wedge mode_1 = manual)\ \& \\ \quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad DisabledAreas;AreasCycle \end{array} \right) \end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$= [D.21]$

$\{mode_1 \neq manual \land mode_1' = mode_1 \Rightarrow mode_1' \neq manual\}$

$$\left(\begin{array}{l} systemState!discharge_s \rightarrow \\ \quad exit \rightarrow \\ \qquad \left(\begin{array}{l} \fatsemi\, area : \mathrm{dom}\, active_1 \rhd \{true\} \bullet \\ \quad\quad switchLamp[AreaId].area!on \rightarrow Skip \end{array}\right); \\ \qquad \left(\begin{array}{l} (\mathrm{dom}\, active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad\quad SwitchInternalSystem2DisabledMode \\ \Box\ (\mathrm{dom}\, active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad\quad SwitchInternalSystem2AutomaticMode \end{array}\right); \\ \qquad ActivateDischargeAS; \\ \qquad \{mode_1 \neq manual\}; \\ \qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array}\right) \end{array}\right) \end{array}\right) \setminus GasDischargeSync$$

$\sqsubseteq_{\mathcal{A}} [C.33, C.36, C.57]$

$$\left(\begin{array}{l} systemState!discharge_s \rightarrow \\ \quad exit \rightarrow \\ \qquad \left(\begin{array}{l} \fatsemi\, area : \mathrm{dom}\, active_1 \rhd \{true\} \bullet \\ \quad\quad switchLamp[AreaId].area!on \rightarrow Skip \end{array}\right); \\ \qquad \left(\begin{array}{l} (\mathrm{dom}\, active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \quad\quad SwitchInternalSystem2DisabledMode \\ \Box\ (\mathrm{dom}\, active_1 \rhd \{true\} = \emptyset)\ \& \\ \quad\quad SwitchInternalSystem2AutomaticMode \end{array}\right); \\ \qquad ActivateDischargeAS; \\ \qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ \left(\begin{array}{l} \{mode_1 \neq manual\}; \\ \left(\begin{array}{l} (mode_A = manual \land mode_1 = manual)\ \& \\ \quad\quad ActiveAreas;AreasCycle \\ \Box\ (mode_1 \neq manual)\ \& \\ \quad\quad DisabledAreas;AreasCycle \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right) \setminus GasDischargeSync$$

$\sqsubseteq_{\mathcal{A}}$ [$D.17$, $C.36$, $C.57$]
$\{mode_1 \neq manual \Rightarrow mode_1 \neq manual\}$
$\{mode_1 \neq manual \Rightarrow \neg\,(mode_A = manual \wedge mode_1 = manual)\}$

$$\left(\begin{array}{l} systemState!discharge_s \rightarrow \\ \quad exit \rightarrow \\ \qquad \left(\begin{array}{l} \fatsemi\, area : \mathrm{dom}\, active_1 \rhd \{true\} \bullet \\ \qquad switchLamp[AreaId].area!on \rightarrow Skip \end{array}\right); \\ \qquad \left(\begin{array}{l} (\mathrm{dom}\, active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \qquad SwitchInternalSystem2DisabledMode \\ \Box\ (\mathrm{dom}\, active_1 \rhd \{true\} = \emptyset)\ \& \\ \qquad SwitchInternalSystem2AutomaticMode \end{array}\right); \\ \qquad ActivateDischargeAS; \\ \qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (DisabledAreas;AreasCycle) \end{array}\right) \end{array}\right)$$

$\setminus\ GasDischargeSync$
$\sqsubseteq_{\mathcal{A}}$ [$D.28$]
$\{wrtV(ActivateDischargeAS) \subseteq \alpha(AreasState) \cup \alpha(AreasState')\}$
$\{wrtV(ActivateDischargeAS) \cap usedV(Reset_2) = \emptyset\}$

$$\left(\begin{array}{l} systemState!discharge_s \rightarrow \\ \quad exit \rightarrow \\ \qquad \left(\begin{array}{l} \fatsemi\, area : \mathrm{dom}\, active_1 \rhd \{true\} \bullet \\ \qquad switchLamp[AreaId].area!on \rightarrow Skip \end{array}\right); \\ \qquad \left(\begin{array}{l} (\mathrm{dom}\, active_1 \rhd \{true\} \neq \emptyset)\ \& \\ \qquad SwitchInternalSystem2DisabledMode \\ \Box\ (\mathrm{dom}\, active_1 \rhd \{true\} = \emptyset)\ \& \\ \qquad SwitchInternalSystem2AutomaticMode \end{array}\right); \\ \qquad \left(\begin{array}{l} Reset_2 \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ (ActivateDischargeAS;DisabledAreas;AreasCycle) \end{array}\right) \end{array}\right)$$

$\setminus\ GasDischargeSync$

$$
\begin{array}{l}
= [D.38] \\
\{\mathrm{dom}\ active_1 \rhd \{true\} \subseteq AreaId\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l}
\fatsemi\ area : AreaId \bullet \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad switchLamp[AreaId].area!on \rightarrow Skip \\
\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad Skip
\end{array} \right) ; \\
\qquad \left(\begin{array}{l}
(\mathrm{dom}\ active_1 \rhd \{true\} \neq \emptyset)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (\mathrm{dom}\ active_1 \rhd \{true\} = \emptyset)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array} \right) ; \\
\qquad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS\,;DisabledAreas\,;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
= [B.3, D.4] \\
\{log\ is\ fresh\ variable\ name\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l}
\mathbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\qquad \left(\begin{array}{l}
\fatsemi\ area : AreaId \bullet \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad switchLamp[AreaId].area!on \rightarrow \\
\qquad\quad log := log + 1 \\
\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad Skip
\end{array} \right) ; \\
\qquad \left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array} \right)
\end{array} \right) ; \\
\qquad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS\,;DisabledAreas\,;AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.25, D.34] \\
\{\{switchLamp\} \cap \Sigma_2 = \emptyset\} \\
\{log \notin FV(Reset_2) \cup FV(ActivateDischargeAS;DisabledAreas;AreasCycle)\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow \\
\quad exit \rightarrow \\
\qquad \left(\begin{array}{l}
\textbf{var } log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
{}^{9}_{9}\ area : AreaId \bullet \\
\quad (area \in \operatorname{dom} active_1 \rhd \{true\}) \ \& \\
\qquad \left(\begin{array}{l} switchLamp[AreaId].area!on \rightarrow \\ \quad log := log + 1 \end{array}\right) \\
\qquad [\![\alpha(InternalSystemState) \cup \{log\} \\
\qquad \mid \Sigma_2 \mid \\
\qquad \alpha(AreasState)]\!] \\
\qquad Skip \\
\quad \Box\ (area \notin \operatorname{dom} active_1 \rhd \{true\}) \ \& \\
\qquad Skip \\
\qquad [\![\alpha(InternalSystemState) \cup \{log\} \\
\qquad \mid \Sigma_2 \mid \\
\qquad \alpha(AreasState)]\!] \\
\qquad Skip
\end{array}\right) \\
\quad \left(\begin{array}{l}
(log = 0) \ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0) \ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right)
\end{array}\right) \\
\qquad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS;DisabledAreas;AreasCycle)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [B.2] \\
\{\{gasDischarged, gasNotDischarged, automaticDischarge\} \cap \{switchLamp\} = \emptyset\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var } log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\S\ area : AreaId \bullet \\
\quad \left(\begin{array}{l}
automaticDischarge.area \rightarrow \\
\quad \left(\begin{array}{l}
gasDischarged.area \rightarrow \\
\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad log := log + 1 \\
\Box\ gasNotDischarged.area \rightarrow Skip
\end{array} \right)
\end{array} \right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \\
\mid \Sigma_2 \mid \\
\alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
automaticDischarge.area \rightarrow \\
\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasDischarged.area \rightarrow Skip \\
\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array} \right)
\end{array} \right) \\
\quad \left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array} \right)
\end{array} \right) \\
\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS; DisabledAreas; AreasCycle)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$= [D.39]$

$\{automaticDischarge \notin \{gasDischarged, gasNotDischarged, switchLamp\}\}$

$$
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathop{\fatsemi}_{\S}\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array}\right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \\
\mid \Sigma_2 \mid \\
\alpha(AreasState)]\!] \\
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \operatorname{dom} active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \operatorname{dom} active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right)
\end{array}\right) \\
\quad \left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right)
\end{array}\right) \\
\quad \left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS\,;DisabledAreas\,;AreasCycle)
\end{array}\right)
\end{array}\right) \setminus GasDischargeSync
$$

$$
\begin{array}{l}
= [D.8] \\
\{log \notin FV(Reset_2) \cup FV(ActivateDischargeAS;DisabledAreas;AreasCycle)\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
{}^{\circ}_{9}\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array} \right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \\
\mid \Sigma_2 \mid \\
\alpha(AreasState)]\!] \\
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array} \right)
\end{array} \right) \\
\left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array} \right); \\
\left(\begin{array}{l}
Reset_2 \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(ActivateDischargeAS;DisabledAreas;AreasCycle)
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq_{\mathcal{A}}\ [D.28] \\
\{wrtV(SwitchInternalSystem2AutomaticMode) \cup \\
\quad wrtV(SwitchInternalSystem2DisabledMode) \subseteq \\
\qquad \alpha(InternalSystemState) \cup \alpha(InternalSystemState')\} \\
\{wrtV(SwitchInternalSystem2AutomaticMode) \cup \\
\quad wrtV(SwitchInternalSystem2DisabledMode) \cap \\
\qquad usedV(ActivateDischargeAS\,;DisabledAreas\,;AreasCycle) = \emptyset\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\fatsemi\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\qquad gasDischarged.area \rightarrow \\
\qquad\quad switchLamp[AreaId].area!on \rightarrow \\
\qquad\qquad log := log + 1 \\
\qquad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array}\right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \\
\ |\ \Sigma_2\ | \\
\ \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\qquad (area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\qquad\quad gasDischarged.area \rightarrow Skip \\
\qquad (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \& \\
\qquad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right)
\end{array}\right)\ ; \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right)\ ; \\
Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \cup \{log\}\ |\ \Sigma_2\ |\ \alpha(AreasState)]\!] \\
(ActivateDischargeAS\,;DisabledAreas\,;AreasCycle)
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.33, C.54] \\
\{\emptyset \cup usedC(DisabledAreas;AreasCycle) \subseteq cs\} \\
\{\{automaticDischarge, gasDischarged, gasNotDischarged, switchLamp\} \cap \\
\quad usedC(DisabledAreas;AreasCycle) = \emptyset\} \\
\{\{automaticDischarge, gasDischarged, gasNotDischarged\} \cap \emptyset = \emptyset\}
\end{array}
$$

$$
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\fatsemi\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
\left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right); \\
Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
ActivateDischargeAS;DisabledAreas;AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
$$

$$
\begin{array}{l}
= [C.58, D.27] \\
\{\{automaticDischarge\} \cup \{detection\} \subseteq \Sigma_2\} \\
\{\{automaticDischarge\} \cap \{detection\} = \emptyset\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\left(\begin{array}{l}
\textbf{var } log : \mathbb{N} \bullet \\
log := 0; \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathop{;}_{9} area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array} \right) ; \\
\left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array} \right) ; \\
Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array} \right) ; \\
ActivateDischargeAS;DisabledAreas;AreasCycle
\end{array} \right)
\end{array} \right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathop{;}_{9} area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array} \right) ; \\
\left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array} \right) ; \\
Reset_2
\end{array} \right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad WaitingDischarge;AreasCycle
\end{array} \right)
\end{array} \right)
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.46] \\
\{\{automaticDischarge\} \subseteq \Sigma_2\} \\
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathop{\fatsemi}_{\S}\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
\left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right); \\
Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \cup \{log\} \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
ActivateDischargeAS;DisabledAreas;AreasCycle
\end{array}\right) \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad WaitingDischarge;AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.9, D.4, C.44] \\
\{log \notin \{area\} \cup usedV(ActivateDischargeAS;DisabledAreas;AreasCycle)\} \\
\{\{automaticDischarge, detection\} \subseteq \Sigma_2\} \\
\{\Sigma_2 \cap \{systemState, exit\} = \emptyset\} \\
\{wrtV(systemState!discharge_s \rightarrow exit \rightarrow Skip) \cap usedV(A_3)\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
{}^{8}_{9}\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
\quad \left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right); \\
\quad Reset_2
\end{array}\right)
\end{array}\right) \\
\quad [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
ActivateDischargeAS;DisabledAreas;AreasCycle
\end{array}\right) \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad WaitingDischarge;AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [D.8] \\
\{log \notin usedV(Reset_2)\} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
systemState!discharge_s \rightarrow exit \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ log : \mathbb{N} \bullet \\
\quad log := 0; \\
\quad \left(\begin{array}{l}
\fatsemi\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad gasDischarged.area \rightarrow \\
\quad\quad\quad switchLamp[AreaId].area!on \rightarrow \\
\quad\quad\quad\quad log := log + 1 \\
\quad\quad \Box\ gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
\quad \left(\begin{array}{l}
(log = 0)\ \& \\
\quad SwitchInternalSystem2DisabledMode \\
\Box\ (log > 0)\ \& \\
\quad SwitchInternalSystem2AutomaticMode
\end{array}\right)
\end{array}\right); \\
Reset_2
\end{array}\right) \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
ActivateDischargeAS; DisabledAreas; AreasCycle
\end{array}\right) \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad WaitingDischarge; AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync \\
= [Definition\ of\ Discharge_2] \\
\left(\begin{array}{l}
Discharge_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array}\right); \\
ActivateDischargeAS; DisabledAreas; AreasCycle
\end{array}\right) \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; \\
\quad WaitingDischarge; AreasCycle
\end{array}\right)
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

$$
\begin{array}{l}
= [C.24, D.24] \\
\left(\begin{array}{l}
Discharge_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
|||\ area : AreaId \bullet \\
\quad automaticDischarge.area \rightarrow \\
\quad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasDischarged.area \rightarrow Skip \\
\quad\quad (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\quad\quad\quad gasNotDischarged.area \rightarrow Skip
\end{array} \right) ; \\
ActivateDischargeAS ; DisabledAreas
\end{array} \right) \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS ; \\
\quad WaitingDischarge
\end{array} \right) ; \\
AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync = [Definition\ of\ ReplyDischarge] \\
\left(\begin{array}{l}
Discharge_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
ReplyDischarge ; DisabledAreas \\
\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS ;
\end{array} \right) ; \\
\quad WaitingDischarge \\
AreasCycle
\end{array} \right)
\end{array} \right) \\
\setminus\ GasDischargeSync \\
= [D.22, Definition\ of\ WaitingDischarge] \\
\left(\begin{array}{l}
Discharge_2 \\
[\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\
(WaitingDischarge ; AreasCycle)
\end{array} \right) \setminus GasDischargeSync
\end{array}
$$

Appendix C

Existing Refinement Laws

Laws of simulation

Law C.1 (Schema Expressions)

$$ASExp \preceq CSExp$$

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 C.2 (Input prefix distribution)

$$c?x \rightarrow A_1 \preceq c?x \rightarrow A_2$$

provided $A_1 \preceq A_2$ □

Law C.3 (Output prefix distribution)

$$c!ae \rightarrow A_1 \preceq c!ce \rightarrow A_2$$

provided

- $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow ae = ce$
- $A_1 \preceq A_2$ □

Law C.4 (Guard distribution)

$$ag \mathbin{\&} A_1 \preceq cg \mathbin{\&} A_2$$

provided

- $\forall\, P_1.st;\ P_2.st;\ L \bullet R \Rightarrow (ag \Leftrightarrow cg)$
- $A_1 \preceq A_2$ □

Law C.5 (Sequence distribution)

$$A_1;\ A_2 \preceq B_1;\ B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

Law C.6 (External choice distribution)

$$A_1 \,\Box\, A_2 \preceq B_1 \,\Box\, B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

Law C.7 (Parallelism distribution)

$$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 \preceq B_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ □

Law C.8 (Recursion distribution)

$$\mu\, X \bullet F(A) \preceq \mu\, X \bullet F'(A)$$

provided $F \preceq F'$ □

Law C.9 (Schema Expressions)

$$ASExp \preceq CSExp$$

provided

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

Process Refinement

Law C.10 (Process declaration introduction)

$$cp = pd\ cp$$

provided *the process declared in the process declaration pd is not referenced in the sequence of paragraphs of the* Circus *program cp.* □

Law C.11 (Process splitting)
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.* □

Law C.12 (Process indexing)
Let gd and ild be the process declarations above, and let ld be the declaration of the process L. Then,

$$gd = ld\ ild\ \textbf{process}\ G \mathrel{\widehat=} |||\ i : Range \odot IL\lfloor i \rfloor$$

provided *L.pps and G.pps are disjoint with respect to L.st and G.st.* □

Action Refinement

Guards

Law C.13 (Guard Introduction—Assumption)

$$\{ g \};\ A = \{ g \};\ g\ \&\ A$$

Law C.14 (Assumption/Guard—Elimination 1)

$$\{\, g_1 \,\};\ (g_2\ \&\ A) \sqsubseteq_{\mathcal{A}} \{\, g_1 \,\};\ A$$

provided $g_1 \Rightarrow g_2$ □

Law C.15 (Guard Introduction—Schema Expression)

$$SExp \ \sqsubseteq_{\mathcal{A}}\ \Box\, i \bullet g_i\ \&\ SExp \wedge [State \mid g_i]$$

provided $\text{pre}\ SExp \Rightarrow \bigvee i \bullet g_i$ □

Law C.16 (Guard combination)

$$g_1\ \&\ (g_2\ \&\ A) = (g_1 \wedge g_2)\ \&\ A$$

Law C.17 (Guard/Sequence—Association)

$$(g\ \&\ A_1);\ A_2 = g\ \&\ (A_1;\ A_2)$$

Law C.18 (Guard/External choice—Distribution)

$$g\ \&\ (A_1 \,\Box\, A_2) = (g\ \&\ A_1) \,\Box\, (g\ \&\ A_2)$$

Law C.19 (Guard/Internal choice—Distribution)

$$g\ \&\ (A_1 \sqcap A_2) = (g\ \&\ A_1) \sqcap (g\ \&\ A_2)$$

Law C.20 (Guard/Parallelism—Distribution 1)

$$g\ \&\ (A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2) = (g\ \&\ A_1) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g\ \&\ A_2)$$

Law C.21 (Guard/Parallelism—Distribution 2)

$$\begin{array}{l}(g_1\ \&\ A_1) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_2\ \&\ A_2)\\ =\\ (g_1 \vee g_2)\ \&\ ((g_1\ \&\ A_1) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_2\ \&\ A_2))\end{array}$$

Law C.22 (Guard/Interleaving—Distribution 1)

$$g\ \&\ (A_1 \,|||\, A_2) = (g\ \&\ A_1) \,|||\, (g\ \&\ A_2)$$

Law C.23 (Guard/Interleaving—Distribution 2)

$$(g_1 \mathbin{\&} A_1) \mathbin{[\![} [ns_1 \mid ns_2] \mathbin{]\!]} (g_2 \mathbin{\&} A_2)$$
$$=$$
$$(g_1 \lor g_2) \mathbin{\&} ((g_1 \mathbin{\&} A_1) \mathbin{[\![} [ns_1 \mid ns_2] \mathbin{]\!]} (g_2 \mathbin{\&} A_2))$$

Law C.24 (True Guard)

$$\mathit{true} \mathbin{\&} A = A$$

Law C.25 (False Guard)

$$\mathit{false} \mathbin{\&} A = \mathit{Stop}$$

Law C.26 (Guarded Stop)

$$g \mathbin{\&} \mathit{Stop} = \mathit{Stop}$$

Schema Expressions

Law C.27 (Schema Disjunction Elimination)

$$\text{pre } \mathit{SExp}_1 \mathbin{\&} (\mathit{SExp}_1 \lor \mathit{SExp}_2) \sqsubseteq_{\mathcal{A}} \text{pre } \mathit{SExp}_1 \mathbin{\&} \mathit{SExp}_1$$

Law C.28 (Sequence Introduction—Schema Expression)

$$[\Delta S_1;\ \Delta S_2;\ i? : T \mid preS_1 \land preS_2 \land CS_1 \land CS_2]$$
$$=$$
$$[\Delta S_1;\ \Xi S_2;\ i? : T \mid preS_1 \land CS_1];\ [\Xi S_1;\ \Delta S_2;\ i? : T \mid preS_2 \land CS_2]$$

syntactic restrictions

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$;
- $FV(preS_1) \subseteq \alpha(S_1) \cup \{i?\}$;
- $FV(preS_2) \subseteq \alpha(S_2) \cup \{i?\}$;
- $DFV(CS_1) \subseteq \alpha(S_1')$;
- $DFV(CS_2) \subseteq \alpha(S_2')$;
- $UDFV(CS_2) \cap DFV(CS_1) = \emptyset$. □

Law C.29 (Parallelism Introduction—Schema Expression)

$$
\begin{array}{l}
\quad [\Delta S_1;\ \Delta S_2;\ i? : T \mid CS_1(i?, s_2) \land CS_2] \\
= \\
\quad (c?j?s \rightarrow [\Delta S_1;\ \Delta S_2;\ j? : T;\ s? : U \mid CS_1(j?, s?)] \\
\qquad [\![\alpha(S_1) \mid \{\!|c|\!\} \mid \alpha(S2)]\!] \\
\quad c!i!s_2 \rightarrow [\Delta S_1;\ \Delta S_2 \mid CS_2]) \setminus \{\!|c|\!\}
\end{array}
$$

syntactic restrictions

- $\alpha(S_1) \cap \alpha(S_2) = \emptyset$;
- i *is an input variable in scope;*
- $s_2 \in \alpha(S_2)$ *and* s_2 *has type* U;
- $FV(CS_1) \subseteq \alpha(\Delta S_1) \cup \{i?, s_2\}$;
- $FV(CS_2) \subseteq \alpha(\Delta S_2)$;
- c *is a valid channel of type* $T \times U$. □

Assumptions

Law C.30 (Assumption/External Choice—Distribution)

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

Law C.31 (Assumption/Guard—Elimination 2)

$$\{\ g_1\ \};\ (g_2\ \&\ A) = \{\ g_1\ \};\ \mathit{Stop}$$

provided $g_1 \Rightarrow \neg\ g_2$ □

Law C.32 (Assumption/Guard—Replacement)

$$\{\ g_1\ \};\ (g_2\ \&\ A) = \{\ g_1\ \};\ (g_3\ \&\ A)$$

provided $g_1 \Rightarrow (g_2 \Leftrightarrow g_3)$ □

Law C.33 (Assumption/Parallelism—Distribution)

$$\{p\};\ (A_1 \,[\![\ ns_1 \mid cs \mid ns_2\ ]\!]\, A_2) = (\{p\};\ A_1) \,[\![\ cs\ ]\!]\, (\{p\};\ A_2)$$

Law C.34 (Assumption/Interleaving—Distribution)

$$\{p\};\ (A_1 \,[\![ns_1 \mid ns_2]\!]\, A_2) = (\{p\};\ A_1) \,[\![ns_1 \mid ns_2]\!]\, (\{p\};\ A_2)$$

In the following law we refer to a predicate $assump'$. In general, for any predicate p, the predicate p' is formed by dashing all its free undecorated variables.

Law C.35 (Assumption Introduction—Schema Expression)

$$
\begin{array}{l}
[\Delta State;\ i? : T_i;\ o! : T_o \mid p \land assump'] \\
= \\
[\Delta State;\ i? : T_i;\ o! : T_o \mid p \land assump']; \{assump\}
\end{array}
$$

The schema in this law is an arbitrary schema that specifies an action in *Circus*: it acts on a state schema *State* and, optionally, has input variables $i?$ of type T_i, and output variables $o!$ of type T_o.

Law C.36 (Assumption Elimination)

$$\{p\} \sqsubseteq_{\mathcal{A}} Skip$$

Parallelism

Law C.37 (Parallelism Introduction—Sequence 1)

$$
\begin{array}{l}
A_1;\ A_2(e) \\
\sqsubseteq_{\mathcal{A}} \\
((A_1;\ c!e \rightarrow Skip) \llbracket \overline{wrtV(A_2)} \mid \{\!|c|\!\} \mid wrtV(A_2) \rrbracket\ c?y \rightarrow A_2(y)) \setminus \{\!|c|\!\}
\end{array}
$$

syntactic restrictions

- c *is a valid channel of type* T;
- $c \notin usedC(A_1) \cup usedC(A_2)$;
- $y \notin FV(A_2)$.

provided

- $wrtV(A_1) \cap usedV(A_2) = \emptyset$;
- $FV(e) \cap wrtV(A_2\ before\ e) = \emptyset$. □

Law C.38 (Parallelism Introduction—Sequence 2)

$$
\begin{array}{l}
A_1(x);\ A_2(x) \\
= \\
(c!x \rightarrow A_1(x) \ [\![\ \overline{wrtV(A_2)} \mid \{\!|c|\!\} \mid wrtV(A_2)\ ]\!]\ c?y \rightarrow A_2(y)) \setminus \{\!|c|\!\}
\end{array}
$$

syntactic restrictions

- $wrtV(A_1) \cap usedV(A_2) = \emptyset$;
- c *is a valid channel of type* T;
- $c \notin usedC(A_1) \cup usedC(A_2)$;
- $y \notin FV(A_2)$. □

Law C.39 (Parallelism Introduction—Sequence 3)

$$
\begin{array}{l}
A_1(x);\ A_2(x) \\
\sqsubseteq_{\mathcal{A}} \\
((A_1(x);\ c!x \rightarrow \mathit{Skip})\ [\![\ \overline{wrtV(A_2)} \mid \{\!|c|\!\} \mid wrtV(A_2)\ ]\!]\ (c?y \rightarrow A_2(y))) \setminus \{\!|c|\!\}
\end{array}
$$

syntactic restrictions

- c *is a valid channel of type* T;
- $c \notin usedC(A_1) \cup usedC(A_2)$;
- $y \notin FV(A_2)$.

provided $wrtV(A_1) \cap usedV(A_2) = \{x\}$ □

Law C.40 (Channel Combination)

$$
\begin{array}{l}
\left(\begin{array}{l}
A_1[c_1.com_1 \rightarrow c_2.com_2 \rightarrow B_1] \\
[\![ns_1 \mid \{\!|c_1, c_2|\!\} \mid ns_2]\!] \\
A_2[c_1.com_3 \rightarrow c_2.com_4 \rightarrow B_2]
\end{array}\right) \setminus \{\!|c_1, c_2|\!\} \\
= \\
\left(\begin{array}{l}
A_1[c_3.com_1.com_2 \rightarrow B_1] \\
[\![ns_1 \mid \{\!|c_3|\!\} \mid ns_2]\!] \\
A_2[c_3.com_3.com_4 \rightarrow B_2]
\end{array}\right) \setminus \{\!|c_3|\!\}
\end{array}
$$

syntactic restrictions

- *all occurrences of* c_1 *and* c_2 *in* A_1 *and* A_2 *are as explicitly stated;*
- c_3 *is a valid channel of the appropriate type.* □

Law C.41 (Channel Extension 1)

$$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 = A_1 \,[\![\, ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \,]\!]\, A_2$$

provided $c \notin usedC(A_1) \cup usedC(A_2)$ □

Law C.42 (Channel Extension 2)

$$\begin{array}{l} A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2(e) \\ = \\ (c!e \rightarrow A_1 \,[\![\, ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \,]\!]\, c?x \ \rightarrow A_2(x)) \setminus \{\!|c|\!\} \end{array}$$

syntactic restrictions

- c *is a valid channel of the appropriate type;*
- $c \notin usedC(A_1) \cup usedC(A_2)$*;*
- $x \notin FV(A_2)$*.*

provided $FV(e) \cap wrtV(A_2 \ before \ e) = \emptyset$ □

Law C.43 (Synchronisation Elimination)

$$\begin{array}{l} (\Box\, i \bullet g_i \ \&\ c_i.ccom_i \rightarrow d_i.acom_i \rightarrow A_i) \\ \qquad [\![ns_1 \mid cs \cup \{\!|i \bullet c_i|\!\} \mid ns_2]\!] \\ (\Box\, i \bullet g_i \ \&\ c_i.ccom_i \rightarrow d_i.bcom_i \rightarrow B_i) \\ = \\ (\Box\, i \bullet d_i.acom_i \rightarrow A_i) \\ \qquad [\![ns_1 \mid cs \mid ns_2]\!] \\ (\Box\, i \bullet g_i \ \&\ c_i.ccom_i \rightarrow d_i.bcom_i \rightarrow B_i) \end{array}$$

provided $\{i \bullet c_i\} \cap usedC(A_i) \cup usedC(B_i) = \emptyset$ □

Law C.44 (Parallelism/Sequence—Step)

$$(A_1;\ A_2) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_3 = A_1;\ (A_2 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_3)$$

provided

- $initials(A_3) \subseteq cs$*;*
- $cs \cap usedC(A_1) = \emptyset$*;*
- $wrtV(A_1) \cap usedV(A_3) = \emptyset$ □

Law C.45 (Parallelism/External Choice—Exchange)

$$(A_1 \llbracket cs \rrbracket A_2) \Box (B_1 \llbracket cs \rrbracket B_2) = (A_1 \Box B_1) \llbracket cs \rrbracket (A_2 \Box B_2)$$

provided $A_1 \llbracket cs \rrbracket B_2 = A_2 \llbracket cs \rrbracket B_1 = Stop$ □

Law C.46 (Parallelism/External Choice - Distribution)

$$A_1 \llbracket cs \rrbracket (A_2 \Box A_3) = (A_1 \llbracket cs \rrbracket A_2) \Box (A_1 \llbracket cs \rrbracket A_3)$$

provided

- $initials(A_1) \subseteq cs$;
- A_1 *is deterministic.*

□

Law C.47 (Parallelism Deadlock)

$$g1 \;\&\; c_1 \rightarrow A_1 \llbracket ns_1 \mid cs \cup \{\!|c_1, c_2|\!\} \mid ns_2 \rrbracket g_2 \;\&\; c_2 \rightarrow A_2 = Stop$$

provided $c_1 \neq c_2$ □

Prefixing

Law C.48 (Prefix/Sequential Composition—Association)

$$c \rightarrow (A_1;\ A_2) = (c \rightarrow A_1);\ A_2$$

syntactic restriction $FV(A_2) \cap \alpha(c) = \emptyset$ □

The following are laws for distribution.

Law C.49 (Prefix/External choice—Distribution)

$$c \rightarrow \Box\ i \bullet g_i \;\&\; A_i = \Box\ i \bullet g_i \;\&\; c \rightarrow A_i$$

provided $\vee\ i \bullet g_i$

syntactic restriction $FV(g_i) \cap \alpha(c) = \emptyset$, *for all* i □

The proviso is needed to ensure that at least one guard is valid, so that in the right-hand side action the communication does take place.

Law C.50 (Prefix/Internal choice—Distribution)

$$c \rightarrow (A_1 \sqcap A_2) = (c \rightarrow A_1) \sqcap (c \rightarrow A_2)$$

Law C.51 (Prefix/Parallelism—Distribution)

$$c \rightarrow (A_1 \,[\![\, cs \,]\!]\, A_2) = (c \rightarrow A_1) \,[\![\, ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \,]\!]\, (c \rightarrow A_2)$$

syntactic restriction $c \notin usedC(A_1) \cup usedC(A_2)$ *or* $c \in cs$ □

External choice

Law C.52 (External choice/Sequence—Distribution)

$$(\Box\, i \bullet g_i \;\&\; c_i \rightarrow A_i);\; B = \Box\, i \bullet g_i \;\&\; c_i \rightarrow A_i;\; B$$

Hiding

Law C.53 (Hide combination)

$$(A \setminus cs_1) \setminus cs_2 = A \setminus (cs_1 \cup cs_2)$$

Law C.54 (Hide expansion)

$$F(A \setminus cs) = F(A) \setminus cs$$

provided $cs \cap usedC(F(\_)) = \emptyset$ □

Recursion

Law C.55 (Recursion Unfold)

$$\mu\, X \bullet F(X) = F(\mu\, X \bullet F(X))$$

Law C.56 (Recursion—Least Fixed Point)

$$F(Y) \sqsubseteq_{\mathcal{A}} Y \Rightarrow \mu\, X \bullet F(X) \sqsubseteq_{\mathcal{A}} Y$$

Unit and Zero Laws

Law C.57 (Sequence—Unit)

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

Law C.58 (External Choice—Unit)

$$Stop \,\Box\, A = A$$

Law C.59 (Sequence—Zero)

$$Stop;\ A = Stop$$

Appendix D

New Refinement Laws

Laws of Simulation.

Law D.1 (Prefixing/Simulation)

$$c \rightarrow A_1 \preceq c \rightarrow A_2$$

provided $A_1 \preceq A_2$ □

Law D.2 (Interleave/Simulation)

$$A_1 \,|||\, A_2 \preceq B_1 \,|||\, B_2$$

provided

- $A_1 \preceq A_2$
- $B_1 \preceq B_2$

□

Law D.3 (Internal Choice/Simulation)

$$A_1 \sqcap A_2 \preceq B_1 \sqcap B_2$$

provided

- $A_1 \preceq A_2$
- $B_1 \preceq B_2$

□

Laws on Prefixing.

Law D.4 (Prefixing/Skip)

$$c \rightarrow A = (c \rightarrow Skip);A$$

□

Law D.5 (Prefixing Introduction)

$$A = (c \rightarrow A) \setminus \{\!|\ c\ |\!\}$$

provided $c \notin usedC(A)$ □

Law D.6 (Prefixing/Parallelism 2)

$$c?x \rightarrow (A_1(x) \ [\![\ cs\ ]\!]\ A_2(x)) = (c?x \rightarrow A_1(x)) \ [\![\ cs\ ]\!]\ (c?x \rightarrow A_2(x))$$

provided $c \notin usedC(A_1) \cup usedC(A_2)$ *or* $c \in cs$ □

Laws on Schemas.

Law D.7 (Schemas/Sequence Introduction 2)

$$\begin{array}{l}[\ S_1';\ S_2' \mid preS_1 \wedge preS_2 \wedge CS_1 \wedge CS_2\ ] \\ = \\ [\ S_1' \mid preS_1 \wedge CS_1\ ];[\ S_2' \mid preS_2 \wedge CS_2\ ]\end{array}$$

provided

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

□

Laws on Variable Blocks.

Law D.8 (Variable Block Extension)

$$A_1;(\textbf{var}\ x : T \bullet A_2);A_3 = (\textbf{var}\ x : T \bullet A_1;A_2;A_3)$$

provided $x \notin FV(A_1) \cup FV(A_3)$ □

Law D.9 (Variable Block Extension 2)

$$\begin{array}{l}(\textbf{var}\ x : T \bullet A_1) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 \\ = \\ (\textbf{var}\ x : T \bullet A_1 \,[\![\, ns_1 \cup \{x\} \mid cs \mid ns_2 \,]\!]\, A_2)\end{array}$$

provided $x \notin FV(A_2)$ □

Laws on Guards and Assumptions.

Law D.10 (Guard/Parallelism Distribution 1)

$$\Box_i\, g_i \,\&\, (A_i \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A) = (\Box_i\, g_i \,\&\, A_i) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A$$

provided

- $initials(A) \subseteq cs$

□

Law D.11 (Guards/Communication Substitution)

$$\begin{array}{l}g_1 \,\&\, (A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, B_1) \\ \Box\ g_2 \,\&\, (A_2 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, B_2) \\ = \\ (A_1 \,\Box\, A_2) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (g_1 \,\&\, B_1 \,\Box\, g_2 \,\&\, B_2)\end{array}$$

provided

- $initials(A_1) \cup initials(A_2) \subseteq cs$
- $usedC(B_1) \cap initials(A_2) = \emptyset$
- $usedC(B_2) \cap initials(A_1) = \emptyset$

□

Law D.12 (Guards Expansion)

$$(g_1 \vee g_2) \,\&\, A = g_1 \,\&\, A \,\Box\, g_2 \,\&\, A$$

□

Law D.13 (Assumption Introduction 2)

$$\{g\} = \{g\};\{g_1\}$$

provided $g \Rightarrow g_1$ □

Law D.14 (Assumption Substitution 1)

$$\{g_1\} \sqsubseteq_{\mathcal{A}} \{g_2\}$$

provided $g_1 \Rightarrow g_2$ □

Law D.15 (Guard/Assumption Introduction 1)

$$g_1 \,\&\, A_1 = g_1 \,\&\, \{g_1\};A_1$$

□

Law D.16 (Guard/Assumption Introduction 2)

$$\{g_1\};\Box_i\, g_i \,\&\, A_i = \{g_1\};\Box_i\, g_i \,\&\, \{g_{2_i}\};A_i$$

provided *For all* i, $g_1 \wedge g_i \equiv g_{2_i}$
Proof. See F. □

Law D.17 (Guard/Assumption Introduction 3)

$$\{g\};A_1 = \{g\};(g_1 \,\&\, A_1 \,\Box\, g_2 \,\&\, A_2)$$

provided

- $g \Rightarrow g_1$
- $g_1 \Rightarrow \neg\, g_2$

Proof. See F. □

Law D.18 (Assumption/Recursion Distribution – Mutual Recursion)

In this law we use the following definitions.

(Parameters). $P = X_1, \ldots, X_n$

(Vector of functions). $V(P) = F_1(X_1, \ldots, X_n), \ldots, F_n(X_1, \ldots, X_n)$

(Substitution of elements). $V(P)[F_i(P) \setminus exp]$ *express the substitution of the* i*-th element of the vector* $V(P)$ *by the expression* exp*.*

$$\mu\, P \bullet V(P) \sqsubseteq_{\mathcal{A}} \mu\, P \bullet V(P)[F_i(P) \setminus \{g\};F_i(P)]$$

provided $\{g\};(F(P)$ *before* $X_i) \sqsubseteq_{\mathcal{A}} (F(P)$ *before* $X_i);\{g\}$ *for all* $F(P)$ *in* $V(P)$
□

Law D.19 (Assumption/Prefixing 1)

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

□

Law D.20 (Assumption/Prefixing 2)

$$\{g\};c?x \rightarrow A = c?x \rightarrow \{g\};A$$

provided $x \notin FV(g)$ □

Law D.21 (Assumption/Schema)

$$\{g\};[d \mid p] = [d \mid p];\{g\}$$

provided $g \wedge p \Rightarrow g'$ □

Laws on External Choice.

Law D.22 (Associativity of External Choice)

$$A_1 \,\Box\, A_2 = A_2 \,\Box\, A_1$$

□

Law D.23 (External Choice Elimination)

$$A \,\Box\, A \sqsubseteq_{\mathcal{A}} A$$

□

Law D.24 (External Choice/Sequence - Distribution.)

$$\Box_i(g_i \;\&\; A_i;A) = (\Box\, g_i \;\&\; A_i);A$$

□

Laws on Parallelism.

Law D.25 (Parallelism Unit)

$$\mathit{Skip} \,[\![\, cs \,]\!]\, A = A \,[\![\, cs \,]\!]\, \mathit{Skip} = A$$

provided $\mathit{usedC}(A) \cap cs = \emptyset$ □

Law D.26 (Parallelism Zero 1)

$$A \,[\![\, cs \,]\!]\, \mathit{Stop} = \mathit{Stop} \,[\![\, cs \,]\!]\, A = \mathit{Stop}$$

provided $\mathit{initials}(A) \subseteq cs$ □

Law D.27 (Parallelism Zero 2)

$$A_1 \,[\![\, cs \,]\!]\, A_2 = \mathit{Stop}$$

provided

- $\mathit{initials}(A_1) \cup \mathit{initials}(A_2) \subseteq cs$
- $\mathit{initials}(A_1) \cap \mathit{initials}(A_2) = \emptyset$

□

Law D.28 (Schemas/Parallelism - Distribution.)

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

provided

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

□

Law D.29 (Parallelism Commutativity)

$$A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ A_2 = A_2 \llbracket\, ns_2 \mid cs \mid ns_1 \,\rrbracket\ A_1$$

□

Law D.30 (Parallelism Introduction 1)

$$c \rightarrow A = (c \rightarrow A \llbracket\, ns_1 \mid \{\!|\ c\ |\!\} \mid ns_2 \,\rrbracket\ c \rightarrow Skip)$$

provided $wrtV(A) \subseteq ns_1 \cup ns_1'$ □

Law D.31 (Parallelism Introduction 2)

$$c?x \rightarrow A_1(x);A_2(x) = c?x \rightarrow A_1(x) \llbracket\, ns_1 \mid \{\!|\ c\ |\!\} \mid ns_2 \,\rrbracket\ c?x \rightarrow A_2(x)$$

provided

- $wrtV(A_1) \cap usedV(A_2) = \emptyset$
- $wrtV(A_1) \subseteq ns_1 \cup ns_1'$
- $wrtV(A_2) \subseteq ns_2 \cup ns_2'$
- $c \notin usedC(A_1) \cup usedC(A_2)$

Proof. See F. □

Law D.32 (Parallelism/External Choice Introduction)
For any variable $x : T$ in scope,

$$
\begin{array}{l}
(c.x \rightarrow A_1(x)) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ (c.x \rightarrow A_2(x)) \\
= \\
(c.x \rightarrow A_1(x)) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ (\Box\, x : T \bullet c.x \rightarrow A_2(x))
\end{array}
$$

provided $c \in cs$ □

Law D.33 (Parallelism/Sequence Distribution)

$$(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2);(B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2)$$
$$\sqsubseteq_{\mathcal{A}}$$
$$(A_1;B_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_2;B_2)$$

provided

- $usedC(B_1) \cup usedC(B_2) \subseteq cs$
- $usedC(A_1) \cap usedC(B_2) = \emptyset$
- $usedC(A_2) \cap usedC(B_1) = \emptyset$

□

Law D.34 (Parallelism Partition Extension)

$$(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (A_1 \llbracket ns_1 \cup \{x\} \mid cs \mid ns_2 \rrbracket A_2)$$

provided

- $x \notin FV(A_1) \cup FV(A_2)$

□

Laws on Communications.

Law D.35 (Communication Introduction)

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2$$
$$=$$
$$((c?x \rightarrow ((x = e)\ \&\ A_1 \,\Box\, (x \neq e)\ \&\ A)) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (c!e \rightarrow A_2)) \setminus cs_2$$

provided

- $c \in cs_1$
- $c \in cs_2$
- $x \notin FV(A_2)$

Proof. See F. □

Law D.36 (Channel Extension 3.)

$$(A_1 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A_2(e)) \setminus cs_2$$
$$=$$
$$((c!e \rightarrow A_1) \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, (c?x \rightarrow A_2(x))) \setminus cs_2$$

provided

- $c \in cs_1$
- $c \in cs_2$
- $x \notin FV(A_2)$

□

Law D.37 (Channel Extension 4.)

$$(A_1 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A_2(e)) \setminus cs_2$$
$$=$$
$$((c \rightarrow A_1) \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, (c \rightarrow A_2)) \setminus cs_2$$

provided

- $c \in cs_1$
- $c \in cs_2$

□

Laws on Indexed Sequential Composition.

Law D.38 (Indexed Sequential Composition/Guards Introduction)

$$\fatsemi\, x : S_1 \bullet A_1(x) = \fatsemi\, x : S_2 \bullet ((x \in S_1)\ \&\ A_1(x)) \,\Box\, ((x \notin S_1)\ \&\ \mathit{Skip})$$

provided $S_1 \subseteq S_2$ □

Law D.39 (Indexed Sequential Composition 1)

$$\begin{array}{l}
\mathop{\fatsemi}_{9}\ x : T \bullet ((c.x \rightarrow A_1(x)) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (c.x \rightarrow A_2(x))) \\
= \\
(\mathop{\fatsemi}_{9}\ x : T \bullet c.x \rightarrow A_1(x)) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (|||\ x : T \bullet c.x \rightarrow A_2(x))
\end{array}$$

provided

- $c \notin usedC(A_1(x)) \cup usedC(A_2(x))$

□

Appendix E

Laws of Logical Calculi

Law E.1 (One Point Rule)

$$\forall\, x \bullet x = t \Rightarrow A \equiv A[x \setminus t] \equiv \exists\, x \bullet x = t \wedge A$$

Law E.2 (Universal Quantifier Elimination)

$$\forall\, x : T \bullet P \equiv P$$

Law E.3 (Implication Elimination)

$$A \Rightarrow A \equiv \mathit{true}$$

Law E.4 (Transitivity of Equality)

$$A = B \wedge B = C \equiv A = C$$

Appendix F

Proof of Some Derived Laws

Guard/Assumption Introduction 2 (D.16).

$\{g_1\};\Box_i\, g_i \;\&\; A_i$
$= [C.30]$
$\Box_i\{g_1\};g_i \;\&\; A_i$
$= [C.13, C.16]$
$\Box_i\{g_1\};(g_1 \wedge g_i) \;\&\; A_i$
$= [D.15]$
$\Box_i\{g_1\};(g_1 \wedge g_i) \;\&\; \{g_1 \wedge g_i\};A_i$
$= [C.16, C.13]$
$\Box_i\{g_1\};g_i \;\&\; \{g_1 \wedge g_i\};A_i$
$= [\forall\, i \bullet g_1 \wedge g_i \equiv g_{2_1}]$
$\{g_1\};\Box_i\, g_i \;\&\; \{g_{2_i}\};A_i$

Guard/Assumption Introduction 3 (D.17).

$\{g\};A_1$
$= [D.13]$
$\{g \Rightarrow g_1\}$
$\{g\};\{g_1\};A_1$
$= [C.13]$
$\{g\};\{g_1\};g_1 \;\&\; A_1$
$= [C.58]$
$\{g\};\{g_1\};((g_1 \;\&\; A_1) \;\Box\; \mathit{Stop})$
$= [C.30]$
$\{g\};(\{g_1\};g_1 \;\&\; A_1) \;\Box\; (\{g_1\};\mathit{Stop})$
$= [C.31]$
$\{g_1 \Rightarrow \neg\, g_2\}$
$\{g\};(\{g_1\};g_1 \;\&\; A_1) \;\Box\; (\{g_1\};g_2;A_2)$
$= [C.30]$
$\{g\};\{g_1\};(g_1 \;\&\; A_1 \;\Box\; g_2;A_2)$
$= [D.13]$
$\{g \Rightarrow g_1\}$
$\{g\};(g_1 \;\&\; A_1 \;\Box\; g_2;A_2)$

Parallelism Introduction 2 (D.31).

$c?x \rightarrow A_1(x);A_2(x)$
$= [C.38]$
$\{c_1 \notin \mathit{usedC}(A_1) \cup \mathit{usedC}(A_2)\}$
$\{\mathit{wrtV}(A_1) \cap \mathit{usedV}(A_2) = \emptyset\}$
$\{\mathit{wrtV}(A_1) \subseteq ns_1 \cup ns_1'\}$
$\{\mathit{wrtV}(A_2) \subseteq ns_2 \cup ns_2'\}$
$c?x \rightarrow ((c_1!x \rightarrow A_1(x)) \;[\![\, \{\!|\; c_1 \;|\!\} \,]\!]\; (c_1?y \rightarrow A_2(y))) \setminus \{\!|\; c_1 \;|\!\}$
$= [C.42]$
$\{c_1 \notin \mathit{usedC}(A_1) \cup \mathit{usedC}(A_2)\}$
$c?x \rightarrow (A_1 \;[\![\, \{\!|\; \emptyset \;|\!\} \,]\!]\; A_2)$
$= [D.6]$
$\{c \notin \mathit{usedC}(A_1) \cup \mathit{usedC}(A_2)\}$
$= (c?x \rightarrow A_1) \;[\![\, ns_1 \mid \{\!|\; c \;|\!\} \mid ns_2 \,]\!]\; (c?x \rightarrow A_2)$

Communication Introduction (D.35).

$(A_1 \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A_2) \setminus cs_2$
$= [C.25, C.58]$
$(((\mathit{true} \;\&\; A_1) \,\Box\, (\mathit{false} \;\&\; A)) \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A_2) \setminus cs_2$
$= [\mathit{Logical\ Calculus}]$
$(((e = e \;\&\; A_1) \,\Box\, (e \neq e \;\&\; A)) \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, A_2) \setminus cs_2$
$= [D.36]$
$\{c \in cs_1\}$
$\{c \in cs_2\}$
$\{x \notin FV(A_2)\}$
$((c?x \to (x = e \;\&\; A_1) \,\Box\, (x \neq e \;\&\; A)) \,[\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!]\, (c!e \to A_2)) \setminus cs_2$

Appendix G

Case Study - Some Refinement Steps

Data refinement: including a new state component

process $FireControl_1 \mathrel{\widehat=} \textbf{begin}$

state

$FireControlState_1$

$mode_1 : Mode$
$controlledZones_1 : AreaId \to \mathbb{P}\, ZoneId$
$activeZones_1 : AreaId \to \mathbb{P}\, ZoneId$
$discharge_1 : AreaId \to Bool$
$active_1 : AreaId \to Bool$
$log : \mathbb{N}$
$mode_A : Mode$

$controlledZones_1 = \{area : AreaId \bullet area \mapsto getZones(area)\}$
$\forall\, area : AreaId \bullet$
$\quad (mode_1 = automatic) \Rightarrow$
$\quad\quad active_1(area) = true \Leftrightarrow$
$\quad\quad\quad \exists\, z_1, z_2 : controlledZones_1(area) \bullet$
$\quad\quad\quad\quad z_1 \neq z_2 \land \{z_1, z_2\} \subseteq activeZones_1(area)$
$\quad (mode_1 = manual) \Rightarrow$
$\quad\quad active_1(area) = true \Leftrightarrow$
$\quad\quad\quad \{area : AreaId \mid \exists\, z : controlledZones_1(area) \bullet$
$\quad\quad\quad\quad z \in activeZones_1(area)\}$
$\quad activeZones_1(area) \subseteq controlledZones_1(area)$

RetrFireControl

$AbstractFireControlState$
$FireControlState_1$

$mode_1 = mode$
$controlledZones_1 = controlledZones$
$activeZones_1 = activeZones$
$discharge_1 = discharge$
$active_1 = active$

InitFireControl$_1$

$FireControlState'_1$

$mode'_1 = automatic$
$controlledZones'_1 = \{area : AreaId \bullet area \mapsto getZones(area)\}$
$activeZones'_1 = \{area : AreaId \bullet area \mapsto \emptyset\}$
$discharge'_1 = \{area : AreaId \bullet area \mapsto false\}$
$active'_1 = \{area : AreaId \bullet area \mapsto false\}$
$log = 0$
$mode'_A = automatic$

SwitchFireControlMode$_1$

$\Delta FireControlState_1$
$newMode? : Mode$

$mode'_1 = newMode?$
$controlledZones'_1 = controlledZones_1$
$activeZones'_1 = activeZones_1$
$discharge'_1 = discharge_1$
$active'_1 = active_1$
$log' = log$
$mode'_A = newMode?$

SwitchFireControl2AutomaticMode$_1$

$\Delta FireControlState_1$

$mode'_1 = automatic$
$controlledZones'_1 = controlledZones_1$
$activeZones'_1 = activeZones_1$
$discharge'_1 = discharge_1$
$active'_1 = active_1$
$log' = log$
$mode'_A = mode_A$

SwitchFireControl2DisabledMode$_1$

$\Delta FireControlState_1$

$mode'_1 = disabled$
$controlledZones'_1 = controlledZones_1$
$activeZones'_1 = activeZones_1$
$discharge'_1 = discharge_1$
$active'_1 = active_1$
$log' = log$
$mode'_A = mode_A$

ActivateZone$_1$

$\Delta FireControlState_1$
$newZone? : ZoneId$

$mode'_1 = mode_1$
$controlledZones'_1 = controlledZones_1$
$activeZones'_1 = activeZones_1 \oplus$
$\quad \{area : AreaId \mid newZone? \in controlledZones_1(area) \bullet$
$\quad\quad area \mapsto activeZones_1(area) \cup \{newZone?\}\}$
$discharge'_1 = discharge_1$
$log' = log$
$mode'_A = mode_A$

$$
\begin{array}{|l}
\hline
ActivateDischarge_1 \\
\Delta FireControlState_1 \\
\hline
mode'_1 = mode_1 \\
controlledZones'_1 = controlledZones_1 \\
activeZones'_1 = activeZones_1 \\
discharge'_1 = discharge_1 \oplus \\
\quad \{area : AreaId \mid area \in \mathrm{dom}\, active_1 \rhd \{true\} \bullet area \mapsto true\} \\
active'_1 = active_1 \\
mode'_A = mode_A \\
\hline
\end{array}
$$

$$
\begin{array}{l}
FireSysStart_1 \mathrel{\widehat=} \\
\quad systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad\quad InitFireControl_1; \; FireSys_1
\end{array}
$$

$$
\begin{array}{l}
SwitchLampsOff_1 \mathrel{\widehat=} \\
\quad (switchBuzzer!off \rightarrow Skip \\
\quad \lll id : (LampId \setminus \{circuitFaultLamp, systemOnLamp\}) \bullet \\
\quad\quad switchLamp[LampId].id!off \rightarrow Skip \\
\quad \lll zone : ZoneId \bullet switchLamp[ZoneId].zone!off \rightarrow Skip \\
\quad \lll area : AreaId \bullet switchLamp[AreaId].area!off \rightarrow Skip)
\end{array}
$$

$$
\begin{array}{l}
SwitchLampsDischarge_1 \mathrel{\widehat=} \\
\quad (\fatsemi \; area : \mathrm{dom}\, active_1 \rhd \{true\} \bullet switchLamp[AreaId].area!on \rightarrow Skip)
\end{array}
$$

$$
\begin{array}{l}
FireSys_1 \mathrel{\widehat=} \\
\quad systemState!fireSys_s \rightarrow \\
\quad\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\quad\quad\quad SwitchFireControlMode_1; \; FireSys_1 \\
\quad\quad \Box \; detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\quad\quad\quad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\quad\quad\quad\quad (mode_1 = manual) \;\&\; Manual_1 \\
\quad\quad\quad\quad \Box \; (mode_1 = automatic) \;\&\; Auto_1 \\
\quad\quad \Box \; fault?faultId : FaultId \rightarrow \\
\quad\quad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\quad\quad\quad\quad switchBuzzer!on \rightarrow FireSys_1 \\
\quad\quad \Box \; reset \rightarrow alarm!alarmOff \rightarrow \\
\quad\quad\quad InitFireControl_1; \; SwitchLampsOff_1; FireSys_1
\end{array}
$$

$$
\begin{array}{l}
Manual_1 \mathrel{\widehat=} \\
\quad systemState!manual_s \rightarrow \\
\qquad detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_1 \\
\qquad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_1 \\
\qquad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad\quad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\qquad switchLamp[AreaId].area!on \rightarrow ActivateDischarge_1; \\
\qquad\qquad\quad SwitchFireControl2DisabledMode_1;\ Reset_1 \\
\qquad\quad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \&\ Manual_1 \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow Manual_1
\end{array}
$$

$$
\begin{array}{l}
Auto_1 \mathrel{\widehat=} \\
\quad systemState!auto_s \rightarrow \\
\qquad (active \rhd \{true\} \neq \emptyset)\ \& \\
\qquad\quad alarm!secondStage \rightarrow Countdown_1 \\
\qquad \Box\ (active \rhd \{true\} = \emptyset)\ \& \\
\qquad\quad reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_1; \\
\qquad\qquad InitFireControl_1;\ FireSys_1 \\
\qquad\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\qquad\qquad switchLamp[ZoneId].newZone!on \rightarrow Auto_1 \\
\qquad\quad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad\quad switchBuzzer!on \rightarrow Auto_1
\end{array}
$$

$$
\begin{array}{l}
Reset_1 \mathrel{\widehat=} \\
\quad systemState!reset_s \rightarrow \\
\qquad actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow \\
\qquad\quad SwitchFireControl2AutomaticMode_1;\ Reset_1 \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1; \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow Reset_1 \\
\qquad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_1; \\
\qquad\quad (mode_1 = disabled)\ \&\ FireSysD_1 \\
\qquad\quad \Box\ (mode_1 \neq disabled)\ \&\ InitFireControl_1;\ FireSys_1 \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow Reset_1
\end{array}
$$

$$
Countdown_1 \mathrel{\widehat=} systemState!countdown_s \rightarrow startClock \rightarrow WaitingClock_1
$$

$WaitingClock_1 \mathrel{\widehat=}$
$clockFinished \rightarrow Discharge_1$
$\Box\ detection?newZone : ZoneId \rightarrow ActivateZone_1;$
$switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_1$
$\Box\ fault?faultId : FaultId \rightarrow$
$switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$switchBuzzer!on \rightarrow WaitingClock_1$

$FireSysD_1 \mathrel{\widehat=}$
$systemState!fireSysD_s \rightarrow$
$actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow$
$SwitchLampsOff_1; InitFireControl_1;\ FireSys_1$
$\Box\ fault?faultId : FaultId \rightarrow$
$switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$switchBuzzer!on \rightarrow FireSysD_1$

$Discharge_1 \mathrel{\widehat=}$
$systemState!discharge_s \rightarrow$
$exit \rightarrow$
$SwitchLampsDischarge_1;$
$((\mathrm{dom}\ active \rhd \{true\} \neq \emptyset)\ \&$
$SwitchFireControlSystem2DisabledMode_1$
$\Box\ (\mathrm{dom}\ active \rhd \{true\} = \emptyset)\ \&$
$SwitchFireControlSystem2AutomaticMode_1);$
$ActivateDischarge_1; Reset_1$

• $FireSysStart_1$

end

Process refinement: upgrading the partitions into separated processes (*InternalSystem* and *Areas*)

Process *InternalSystem*

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

InternalSystemState

$mode_1 : Mode$

InitInternalSystem

$InternalSystemState'$

$mode_1' = automatic$

SwitchInternalSystemMode

$\Delta InternalSystemState$

$newMode? : Mode$

$mode_1' = newMode?$

SwitchInternalSystem2AutomaticMode

$\Delta InternalSystemState$

$mode_1' = automatic$

SwitchInternalSystem2DisabledMode

$\Delta InternalSystemState$

$mode_1' = disabled$

$$
\begin{array}{l}
FireSysStart_2 \mathrel{\widehat=} \\
\quad systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\qquad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\qquad\quad InitInternalSystem;\ FireSys_2
\end{array}
$$

$$
\begin{array}{l}
FireSys_2 \mathrel{\widehat=} \\
\quad systemState!fireSys_s \rightarrow \\
\quad modeSwitch?newMode : SwitchMode \rightarrow \\
\qquad SwitchInternalSystemMode;\ FireSys_2 \\
\quad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad switchLamp[ZoneId].newZone!on \rightarrow alarm!firstStage \rightarrow \\
\qquad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\qquad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\quad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\quad switchBuzzer!on \rightarrow FireSys_2 \\
\quad \Box\ reset \rightarrow alarm!alarmOff \rightarrow \\
\qquad InitInternalSystem;\ SwitchLampsOff_2; FireSys_2
\end{array}
$$

$$
\begin{array}{l}
SwitchLampsOff_2 \mathrel{\widehat=} \\
\quad (switchBuzzer!off \rightarrow Skip \\
\quad \lll id : (LampId \setminus \{circuitFaultLamp, systemOnLamp\}) \bullet \\
\qquad switchLamp[LampId].id!off \rightarrow Skip \\
\quad \lll zone : ZoneId \bullet switchLamp[ZoneId].zone!off \rightarrow Skip \\
\quad \lll area : AreaId \bullet switchLamp[AreaId].area!off \rightarrow Skip)
\end{array}
$$

$$
\begin{array}{l}
Manual_2 \mathrel{\widehat=} \\
\quad systemState!manual_s \rightarrow \\
\qquad detection?newZone : ZoneId \rightarrow \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow Manual_2 \\
\qquad \Box\ silenceAlarm \rightarrow alarm!alarmOff \rightarrow Reset_2 \\
\qquad \Box\ externalManualDischarge?area : AreaId \rightarrow \\
\qquad\quad manualDischarge.area \rightarrow \\
\qquad\qquad gasDischarged.area \rightarrow switchLamp[AreaId].area!on \rightarrow \\
\qquad\qquad\quad SwitchInternalSystem2DisabledMode;\ Reset_2 \\
\qquad\qquad \Box\ gasNotDischarged.area \rightarrow Manual_2 \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow Manual_2
\end{array}
$$

$$
\begin{array}{l}
Auto_2 \mathrel{\widehat=} \\
\quad systemState!auto_s \rightarrow \\
\qquad countdown \rightarrow countdownStarted!true \rightarrow \\
\qquad\quad alarm!secondStage \rightarrow Countdown_2 \\
\qquad \Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2; \\
\qquad\quad InitInternalSystem;\ FireSys_2 \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow \\
\qquad\quad switchLamp[ZoneId].newZone!on \rightarrow Auto_2 \\
\qquad \Box\ fault?faultId : FaultId \rightarrow \\
\qquad\quad switchLamp[LampId].getLampId(faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow Auto_2
\end{array}
$$

$Reset_2 \mathrel{\widehat{=}}$
$systemState!reset_s \rightarrow$
$actuatorsReplaced \rightarrow switchLamp[LampId].circuitFaultLamp!off \rightarrow$
$SwitchInternalSystem2AutomaticMode;\ Reset_2$
$\Box\ detection?newZone : ZoneId \rightarrow$
$switchLamp[ZoneId].newZone!on \rightarrow Reset_2$
$\Box\ reset \rightarrow alarm!alarmOff \rightarrow SwitchLampsOff_2;$
$(mode_1 = disabled)\ \&\ FireSysD_2$
$\Box\ (mode_1 \neq disabled)\ \&\ InitInternalSystem;\ FireSys_2$
$\Box\ fault?faultId : FaultId \rightarrow$
$switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$switchBuzzer!on \rightarrow Reset_2$
$\Box\ countdown \rightarrow countdownStarted!false \rightarrow Reset_2$

$Countdown_2 \mathrel{\widehat{=}}\ systemState!countdown_s \rightarrow startClock \rightarrow WaitingClock_2$

$WaitingClock_2 \mathrel{\widehat{=}}$
$clockFinished \rightarrow Discharge_2$
$\Box\ detection?newZone : ZoneId \rightarrow$
$switchLamp[ZoneId].newZone!on \rightarrow WaitingClock_2$
$\Box\ fault?faultId : FaultId \rightarrow$
$switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$switchBuzzer!on \rightarrow WaitingClock_2$
$\Box\ countdown \rightarrow countdownStarted!true \rightarrow WaitingClock_2$

$FireSysD_2 \mathrel{\widehat{=}}$
$systemState!fireSysD_s \rightarrow$
$actuatorsReplaced \rightarrow alarm!alarmOff \rightarrow$
$SwitchLampsOff_2; InitInternalSystem;\ FireSys_2$
$\Box\ fault?faultId : FaultId \rightarrow$
$switchLamp[LampId].getLampId(faultId)!on \rightarrow$
$switchBuzzer!on \rightarrow FireSysD_2$
$\Box\ countdown \rightarrow countdownStarted!false \rightarrow FireSysD_2$

$Discharge_2 \mathrel{\widehat=}$
$systemState!discharge_s \rightarrow$
$exit \rightarrow$
(**var** $log : \mathbb{N} \bullet$
$log := 0;$
$(\fatsemi\ area : AreaId \bullet$
$automaticDischarge.area \rightarrow$
$gasDischarged.area \rightarrow$
$switchLamp[AreaId].area!on \rightarrow log := log + 1$
$\Box\ gasNotDischarged.area \rightarrow Skip);$
$((log = 0)\ \&\ SwitchInternalSystem2AutomaticMode$
$\Box\ (log > 0)\ \&\ SwitchInternalSystem2DisabledMode));$
$Reset_2$

$\bullet\ FireSysStart_2$

end

Process *Areas*

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

AreasState

$mode_A : Mode$
$controlledZones_1 : AreaId \rightarrow \mathbb{P}\ ZoneId$
$activeZones_1 : AreaId \rightarrow \mathbb{P}\ ZoneId$
$discharge_1 : AreaId \rightarrow Bool$
$active_1 : AreaId \rightarrow Bool$

$controlledZones_1 = \{area : AreaId \bullet area \mapsto getZones(area)\}$
$\forall\ area : AreaId \bullet$
$(mode_A = automatic) \Rightarrow$
$active_1(area) = true \Leftrightarrow$
$\exists\ z_1, z_2 : controlledZones_1(area) \bullet$
$z_1 \neq z_2 \land \{z_1, z_2\} \subseteq activeZones_1(area)$
$\land\ (mode_A = manual) \Rightarrow$
$active_1(area) = true \Leftrightarrow$
$\{area : AreaId \mid \exists\ z : controlledZones_1(area) \bullet$
$z \in activeZones_1(area)\}$
$\land\ activeZones_1(area) \subseteq controlledZones_1(area)$

InitAreas

$$AreasState'$$

$$mode'_A = automatic$$
$$activeZones'_1 = \{area : AreaId \bullet area \mapsto \emptyset\}$$
$$discharge'_1 = \{area : AreaId \bullet area \mapsto false\}$$

SwitchAreasMode

$$\Delta AreasState$$
$$newMode? : Mode$$

$$mode'_A = newMode?$$
$$activeZones'_1 = activeZones_1$$
$$discharge'_1 = discharge_1$$

ActivateZoneAS

$$\Delta AreasState$$
$$newZone? : ZoneId$$

$$mode'_A = mode_A$$
$$activeZones'_1 = activeZones_1 \oplus$$
$$\quad \{area : AreaId \mid newZone? \in controlledZones_1(area) \bullet$$
$$\quad\quad area \mapsto activeZones_1(area) \cup \{newZone?\}\}$$
$$discharge'_1 = discharge_1$$

ActivateDischargeAS

$$\Delta AreasState$$

$$mode'_A = mode_A$$
$$activeZones'_1 = activeZones_1$$
$$discharge'_1 = discharge_1 \oplus$$
$$\quad \{area : AreaId \mid area \in \mathrm{dom}\ active_1 \rhd \{true\} \bullet area \mapsto true\}$$

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

$$
\begin{array}{l}
AreasCycle \mathrel{\widehat{=}} \\
\quad (reset \rightarrow InitAreas \\
\quad \Box\ modeSwitch?newMode : SwitchMode \rightarrow SwitchAreasMode \\
\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; ActiveAreas \\
\quad \Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\qquad gasNotDischarged.area \rightarrow Skip \\
\quad \Box\ \Box\ area : AreaId \bullet manualDischarge.area \rightarrow \\
\qquad gasNotDischarged.area \rightarrow Skip); \\
\quad AreasCycle
\end{array}
$$

$$
\begin{array}{l}
ActiveAreas \mathrel{\widehat{=}} \\
\quad (mode_A = automatic)\ \& \\
\qquad (active_1 \rhd \{true\} \neq \emptyset)\ \& \\
\qquad\quad countdown \rightarrow countdownStarted?answer : Bool \rightarrow \\
\qquad\qquad (answer = true)\ \&\ WaitingDischarge \\
\qquad\qquad \Box\ (answer = false)\ \&\ DisabledAreas \\
\qquad \Box\ (active_1 \rhd \{true\} = \emptyset)\ \& \\
\qquad\quad reset \rightarrow InitAreas \\
\qquad\quad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; ActiveAreas \\
\qquad\quad \Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow \\
\qquad\qquad gasNotDischarged.area \rightarrow ActiveAreas \\
\quad \Box\ (mode_A = manual)\ \& \\
\qquad reset \rightarrow InitAreas \\
\qquad \Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; ActiveAreas \\
\qquad \Box\ \Box\ area : AreaId \bullet \\
\qquad\quad manualDischarge.area \rightarrow \\
\qquad\qquad (area \in \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\qquad\quad gasDischarged.area \rightarrow \\
\qquad\qquad\qquad ActivateDischargeAS;\ DisabledAreas \\
\qquad\qquad \Box\ (area \notin \mathrm{dom}\ active_1 \rhd \{true\})\ \& \\
\qquad\qquad\quad gasNotDischarged.area \rightarrow ActiveAreas
\end{array}
$$

$$
\begin{array}{l}
WaitingDischarge \mathrel{\widehat{=}} \\
\quad detection?newZone : ZoneId \rightarrow ActivateZoneAS; WaitingDischarge \\
\quad \Box\ ReplyDischarge; DisabledAreas
\end{array}
$$

$ReplyDischarge \mathrel{\widehat=}$
$(\|\|\| area : AreaId \bullet$
$(automaticDischarge.area \rightarrow$
$(area \in \mathrm{dom}\, active_1 \rhd \{true\})\ \&$
$gasDischarged.area \rightarrow Skip$
$\Box\ (area \notin \mathrm{dom}\, active_1 \rhd \{true\})\ \&$
$gasNotDischarged.area \rightarrow Skip));$
$ActivateDischargeAS$

$DisabledAreas \mathrel{\widehat=}$
$reset \rightarrow InitAreas$
$\Box\ detection?newZone : ZoneId \rightarrow ActivateZoneAS; DisabledAreas$
$\Box\ \Box\ area : AreaId \bullet automaticDischarge.area \rightarrow$
$gasNotDischarged.area \rightarrow DisabledAreas$

$\bullet\ StartAreas$

end

Redefinition of Process $FireControl_2$

process $FireControl_2 \mathrel{\widehat=}$

$$\left(\begin{array}{l} InternalSystem \\ [\![\alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState)]\!] \\ Areas \end{array}\right) \setminus GasDischargeSync$$

Bibliography

[1] J. C. P. Woodcock A. L. C. Cavalcanti, A. C. A. Sampaio. A Refinement Strategy for Circus. *Formal Aspects of Computing*. To appear.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] Rob Arthan. Powerproof reference page. http://www.lemma-one.com/ProofPower/index/index.html.

[4] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2-3):133 – 180, 1990.

[5] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[6] ALC Cavalcanti, ACA Sampaio, and JCP Woodcock. Refinement of Actions in Circus. In *Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science, unknown 2002. Invited Paper.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[9] 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.

[10] C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*. Springer-Verlag, 1998.

[11] 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.

[12] J. He, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In G. Goos and H. Hartmants, editors, *ESOP'86 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187 – 196, March 1986.

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

[14] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[15] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[16] G. Jones and M. Goldsmith. *Programming in occam 2*. Prentice-Hall, 1988.

[17] I. Meisels. *Software Manual for Windows Z/EVES Version 2.1*. ORA Canada, 2000. TR-97-5505-04g.

[18] A. J. R. G. Milner. Is Computing an Experimental Science? Technical Report ECS-LFCS-86-1, University of Edinburgh, Department of Computer Science, Edinburgh - UK, August 1986.

[19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[20] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[21] Carroll Morgan and P. H. B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481 – 503, 1990.

[22] 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.

[23] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28 – 47, 2003.

[24] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[25] P.H.Welch, G.S.Stiles, G.H.Hilderink, and A.P.Bakkers. CSP for Java : Multithreading for All. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[26] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[27] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In D. Gollmann, editor, *ESORICS 94*, volume 1214 of *Lecture Notes in Computer Science*, pages 33 – 54. Springer-Verlag, 1994.

[28] ACA Sampaio, JCP Woodcock, and ALC Cavalcanti. Refinement in Circus. In L Eriksson and PA Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, unknown 2002.

[29] 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.

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

[31] 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.

[32] J. C. P. Woodcock. Using Circus for Safety-Critical Applications. In *VI Brazilian Workshop on Formal Methods*, pages 1–15, Campina Grande, Brazil, 12th–14st October 2003.

[33] Jim Woodcock and Ana Cavalcanti. Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.

[34] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall, 1996.