UNIVERSIDADE DO PORTO
FACULDADE DE ENGENHARIA

Cooperação entre Processos Distribuídos em Ambiente Industrial:

Especificação, Distribuição e Supervisão.

Sérgio Adriano Fernandes Lopes

Novembro de 1998

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
Departamento de Engenharia Electrotécnica e de Computadores

Cooperação entre Processos Distribuídos em Ambiente Industrial: Especificação, Distribuição e Supervisão.

Sérgio Adriano Fernandes Lopes
Licenciado em Engenharia Electrónica Industrial pela Universidade do Minho

Dissertação submetida para satisfação parcial dos
requisitos do grau de Mestre em
Engenharia Electrotécnica e de Computadores
(área de especialização de Informática Industrial)

catalogada

Dissertação realizada sob a supervisão do
Professor Doutor Raul Moreira Vidal,
do Departamento de Engenharia Electrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto,
e co-orientado pelo
Professor Doutor João Luis Pereira Monteiro,
do Departamento de Engenharia Electrónica Industrial
da Universidade do Minho

0037921

681.3(043) 1995/000
UNIVERSIDADE DO PORTO
Faculdade de Engenharia
BIBLIOTECA
N.º 42450
CDU
Data / / 19

Guimarães, Novembro de 1998

Resumo

O desenvolvimento de sistemas baseados em *software* não deve ser um exercício desordenado e aleatório de programação sob pena do sistema não funcionar, ou não funcionar correctamente, ou ainda funcionar e não fazer o que se pretende. Os métodos de desenvolvimento não implementam os sistemas, mas organizam e definem regras para o processo de engenharia de sistemas; consequentemente, isto reflecte-se no tempo de desenvolvimento, nos custos e na qualidade final do sistema. É, além disso, vantajoso utilizar ferramentas que automatizem o processo, nomeadamente ferramentas CASE que suportem a metodologia escolhida.

Daqui resulta uma importância acrescida das metodologias de desenvolvimento de sistemas baseados em *software*. Qualquer empresa deste ramo que pretenda ter um posição competitiva no mercado, deve seguir uma metodologia no desenvolvimento dos seus projectos. Existem várias soluções, umas com maior valor, outras com menos capacidades, mas sobretudo é importante adoptar uma. No entanto, esse passo é muitas vezes complicado devido à inércia que existe em mudar os métodos de trabalho e o tempo e esforço necessários para formar os engenheiros de *software*.

Neste trabalho adoptou-se o método OORT (*Object-Oriented Real-time Techniques*), que é orientado para a especificação de sistemas distribuídos de tempo-real. Foi desenvolvido pela Verilog, juntamente com a ferramenta CASE *Object*GEODE que lhe dá suporte, com base em notações formais que são normas internacionais.

Nesta tese descreve-se o método aplicando-o à especificação completa de um sistema de múltiplos elevadores. O método baseia-se nas notações gráficas OMT, MSC e SDL. Primeiro faz-se uma descrição genérica de cada uma das linguagens e discute-se a sua utilização. Define-se o problema, descrevendo textualmente o sistema a implementar e seus requisitos. Segue-se a aplicação de todo o processo OORT. O modelo de análise constrói-se com base nas linguagens OMT e MSC, a primeira para modelar o ambiente onde o sistema se insere, e a segunda para descrever os requisitos dinâmicos do sistema. A arquitectura do sistema é descrita em SDL. O projecto detalhado do sistema emprega SDL para especificar os objectos concorrentes e OMT para descrever os componentes passivos. A linguagem MSC suporta a especificação dos testes do sistema.

Este trabalho insere-se num projecto mais alargado de especificação de sistemas de controlo distribuídos e estudo dos respectivos problemas.

Abstract

The development of software based systems should not be a unguided and random programming exercise, or the resulting application may not work, not work properly, or even work but not the way it was intended. The development methods do not build systems, but they do organise and define rules for the software engineering process; consequently, this has a impact on the developing time, the overall costs and on the final product quality. Additionally, it is advantageous to use tools that automate the process, namely CASE tools that support the chose methodology.

Therefore, methodologies are of greater importance in the development of *software* systems. Any enterprise in this field aspiring to have a competitive position in the market, should follow some methodology in the development of its projects. There are several alternative solutions, some better than others, but the important thing is to adopt one. Nevertheless, this step is often difficult due to the inertia in changing the work methods and the time and effort required to form the engineers.

In the work here presented, it has been adopted the OORT (*Object-Oriented Real-Time Techniques*) method, which is oriented towards the specification of distributed real-time systems. It has been developed by Verilog, together with the *Object*GEODE CASE tool that supports it, and is based on formal notations that are international standards.

This thesis, describes the method applying it to the complete specification of a multiple elevator system. The is based on the OMT, MSC and SDL graphical notations. First it is given a summarised description of each language and it is discussed its use. The problem is defined, by stating textually the system to implement and its requisites. The application of the whole OORT process follows. The analysis model uses OMT to model the system's environment, and MSC to describe the dynamic requisites. The system's architecture is defined in SDL. The detailed design uses SDL for the concurrent objects specification and OMT for the passive components description. The MSC language supports the test design activity.

This work is part of a wider project in the specification of distributed control systems e the study of the related problems.

Prefácio

Existe pouco trabalho documentado sobre a utilização de objectos em sistemas de tempo-real. Menos ainda, em termos de metodologias de especificação. Existe muita investigação feita sobre métodos OO nos domínios de aplicações empresariais e base de dados, mas não são adequados para sistemas de tempo-real. Por outro lado, os trabalhos no domínio dos sistemas TR ignoram em grande parte a tecnologia OO. Este trabalho aborda uma metodologia de especificação OO para sistemas de tempo-real.

A especificação de *sofware* é actualmente uma área de grande interesse devido à dificuldade de lidar com a complexidade, maiores exigências em termos de desempenho e capacidade de integração de diversas tecnologias, dos sistemas que se desenvolvem. Outras razões, prendem-se com as questões económicas empresariais e a redução de custos de desenvolvimento e manutenção dos sistemas. Por isso, a especificação de *software* constitui, internacionalmente, um foco de interesse muito importante no campo da investigação cientifica relacionado com sistemas distribuídos.

Este trabalho enquadra-se num projecto mais amplo de estudo de sistemas industriais e seus problemas associados à distribuição e requisitos de tempo-real. O objectivo central deste trabalho consiste em estudar uma metodologia actualizada, actuante, madura e semi-automática de especificação de sistemas distribuídos de tempo-real. Com o conhecimento obtido desta metodologia de especificação estamos em condições de, futuramente, elaborar e introduzir soluções para os problemas desses sistemas, ou mecanismos que obviem a sua resolução, ao nível das ferramentas de especificação. Deste modo, a implementação deste tipo de sistemas ficará mais simplificada.

Agradecimentos

Não poderia deixar de manifestar a minha gratidão para com os meus orientadores, Prof. Raúl Moreira Vidal e Prof. João Luis Pereira Monteiro, pelo apoio prestado durante o trabalho e pela ajuda na elaboração desta tese.

Um palavra especial para a minha família e amigos, pela paciência que tiveram e incentivos que me deram para a realização do trabalho.

Sérgio Lopes

x

Índice

Resumo iii

Abstract v

Prefácio vii

Agradecimentos ix

Índices de Figuras xvii

Lista de abreviaturas e símbolos xxi

PARTE I MÉTODOS DE ESPECIFICAÇÃO PARA TEMPO-REAL 1

1 Introdução 3

1.1 O desafio 3

1.2 As Tecnologias 4

1.3 Opção pelo OORT 4

1.4 Convenções da Notação Utilizada 5

2 Actividades da Engenharia de Sistemas 6

2.1 Esquema Genérico 6

2.2 Análise de Requisitos 6

2.3 Projecto da Arquitectura 7

2.4 Projecto Detalhado 7

2.5 Projecto de Testes 8

2.6 Implementação (*Targeting*) 8

2.7 Teste 9

2.8 O Processo de Engenharia de Sistemas 9

3 Resumo da Linguagem OMT 11

3.1 Modelação Por Objectos 11

3.1.1 Diagrama de Classes 11

3.1.2 Diagrama de Instâncias 13

3.2 Modelação Dinâmica 13

3.2.1 Traçado de Eventos 14

3.2.2 Diagramas Estado-Transição 15

3.3 Modelação Funcional 15
3.4 Relação Entre os Modelos de Objectos, Dinâmico e Funcional 16
3.5 Utilização dos Diagramas OMT ao longo do PES 17
3.6 OMT no Desenvolvimento de Sistema TR 17
3.7 UML versus OMT 18

4 Breve Descrição de MSC 20
4.1 Hierarquia de Cenários 20
4.2 MSC 21
4.3 Utilização dos Diagramas MSC no decurso do PES 23
4.4 MSCs no Desenvolvimento de Sistemas TR 23

5 Sumário de SDL 24
5.1 Diagrama de Hierarquia 24
5.2 Diagrama de Interligação 25
5.3 Diagrama de Processo 26
5.4 Declarações PR 28
5.5 Utilização dos Diagramas SDL ao longo do PES 30
5.6 SDL no Desenvolvimento de Sistemas TR 30
5.7 UML versus MSC/SDL 31

6 Combinação de OMT, MSC e SDL no OORT 32
6.1 Posição Genérica das Três Notações 32
6.2 Consistência Global 33
6.3 Benefícios 35

PARTE II ESPECIFICAÇÃO DE UM SISTEMA DE CONTROLO 37

7 Apresentação do Sistema 39
7.1 Requisitos do Utilizador 39
7.2 Notas e Comentários 41

8 Análise de Requisitos 42
8.1 Introdução 42
8.2 Modelação por Objectos em OMT 42
8.2.1 Identificação de Objectos e Classes 43
8.2.2 Identificação das Associações e Agregações 43
8.2.3 Identificação de Atributos 44
8.2.4 Identificação de Operações 44
8.2.5 Redundância 45

8.2.6 Introdução de Herança 45
8.2.7 Identificação de Diagramas de Instâncias 45
8.2.8 Identificação de Módulos 46
8.2.9 Iterações no Modelo de Objectos 46

8.3 Modelo de Objectos do SES 47
8.3.1 Edifício 48
8.3.2 Arquitectura 48
8.3.3 Acesso do Piso 49
8.3.4 Elevador 49
8.3.5 Eixo do Elevador 49
8.3.6 Passageiro e Potencial Passageiro 50
8.3.7 Distribuição de Chamadas 50
8.3.8 Supervisão de Eventos 51
8.3.9 Botões 51
8.3.10 Edifício de Quatro Pisos com Dois Elevadores 52
8.3.11 Comentários ao Modelo: 52

8.4 Modelação Dinâmica em MSCs 53
8.4.1 Identificação de Cenários 53
8.4.2 Hierarquia de Cenários do SES 54
8.4.3 Descrição de Cenários por MSCs 54

8.5 Modelo Dinâmico do SES 57

8.6 Coerência Entre o Modelo de Objectos e o Modelo de Casos de Utilização 60

8.7 Relações com as outras Actividades de Desenvolvimento 61
8.7.1 Transição para o Projecto da Arquitectura 61
8.7.2 Transição para o Projecto Detalhado 61
8.7.3 Transição para o Projecto de Testes 62

9 Projecto da Arquitectura 63

9.1 Introdução 63

9.2 Projecto da Arquitectura em SDL 63
9.2.1 Descrição do Primeiro Nível da Hierarquia 64
9.2.2 Refinamento e Composição 64
9.2.3 Descrição da Comunicação 65
9.2.4 Declaração de Objectos Passivos 66
9.2.5 Portabilidade da Arquitectura Concebida 67

9.3 Projecto da Arquitectura do SES 67

9.4 Relações com as outras Actividades de Desenvolvimento 71
9.4.1 Transição para a Fase de Projecto Detalhado 71
9.4.2 Relação com a Fase de Projecto de Testes 71

10 Projecto Detalhado 73

10.1 Introdução 73

10.2 Projecto Detalhado dos Objectos Concorrentes 73

10.2.1 Identificação de Estados 74
10.2.2 Descrição das Acções das Transições 74
10.2.3 Simulação do Projecto Detalhado 75
10.2.4 Explicação da Actividade 75

10.3 Projecto Detalhado do SES em SDL 76
10.3.1 Gnomo 76
10.3.2 Supervisor 77
10.3.3 Painel de Chamada 78
10.3.4 Porta do Piso 78
10.3.5 Elevador 78
10.3.6 Painel de Destinos 79
10.3.7 Porta do Elevador 79
10.3.8 Monitor do Cabo 80

10.4 SDL: Problemas e Soluções 80
10.4.1 Identificação de Instâncias de Blocos 80
10.4.2 Correspondência Determinística entre Instâncias de Processos de Diferentes Blocos 81

10.5 Projecto Detalhado dos Objectos Passivos 83
10.5.1 Origem dos Objectos Passivos 83
10.5.2 Especificar ADTs com Diagramas de Classes 83
10.5.3 Enriquecer o Projecto dos Diagramas de Classes 83

10.6 Projecto Detalhado do SES em OMT 86

10.7 Reutilização de Objectos Externos 86
10.7.1 Reutilização de Objectos Concorrentes 86
10.7.2 Reutilização de Objectos Passivos 87
10.7.3 Produção de Componentes Reutilizáveis 88

10.8 Portabilidade do Projecto Detalhado 88

10.9 Relações com as outras Actividades de Desenvolvimento 89
10.9.1 Relação com o Projecto de Testes 89
10.9.2 Transição para a Fase de Implementação (*Targeting*) 89

11 Projecto de Testes **90**

11.1 Introdução 90

11.2 Projecto do Conjunto de Testes 91
11.2.1 Produção de MSCs Detalhados 91
11.2.2 Extensão da Cobertura feita pelos Testes 92
11.2.3 Testes de Integração 92
11.2.4 Testes Unitários 92
11.2.5 Concordância do Projecto com os Requisitos 93

11.3 Testes do SES 93

11.4 Relações com as outras Actividades de Desenvolvimento 96
11.4.1 Relação com a Actividade de Projecto Detalhado 96
11.4.2 Transição para o Teste 96

12 Simulação **97**

12.1 Introdução 97
12.2 Sistema Distribuído de Dois Elevadores 98
12.3 Cooperação de Processos Distribuídos 100

13 Conclusões 102

Lista de referências 105

ANEXO A DIAGRAMAS SDL DO PROJECTO DETALHADO DO SES 107

A.1 SES 109
A.2 Bloco Central 111
A.3 Bloco Acesso do Piso 112
A.4 Bloco Elevador 113
A.5 Processo Gnomo 114
A.6 Processo Supervisor 117
A.7 Processo Painel de Chamada 119
A.8 Processo Porta do Piso 120
A.9 Processo Elevador 121
A.10 Processo Painel de Destinos 126
A.11 Processo Porta do Elevador 127
A.12 Processo Monitor do Cabo 129

ANEXO B MSCS DA MODELAÇÃO DINÂMICA DO SES 131

ANEXO C MSCS DO PROJECTO DE TESTES DO SES

Índices de Figuras

Figura 2.1: Esquema Genérico do PES. 6
Figura 2.2: Ciclo-de-vida V. 9
Figura 2.3: Processo Iterativo. 10
Figura 3.1: Exemplo de um Diagrama de Classes. 12
Figura 3.2: Diagrama de Instâncias relativo à Figura 3.1. 14
Figura 3.3: Exemplo de um Traçado de Eventos. 14
Figura 3.4: Exemplo de um Diagrama Estado-Transição. 15
Figura 3.5: Exemplo de um Modelo Funcional. 16
Figura 4.1: Exemplo de uma Hierarquia de Cenários. 20
Figura 4.2: Exemplo de um MSC Abstracto. 22
Figura 4.3: Parte de um MSC Detalhado. 22
Figura 5.1: Exemplo de uma Hierarquia SDL. 25
Figura 5.2: Diagrama de Interligação SDL de um SES. 26
Figura 5.3: Parte de um Diagrama de Processo SDL. 27
Figura 5.4: Descrição Informal de um Operador ADT. 28
Figura 5.5: Descrição Axiomática de um Operador ADT. 29
Figura 5.6: Descrição Algorítmica Textual de um Operador ADT. 29
Figura 5.7: Descrição Algorítmica Gráfica de um Operador ADT. 29
Figura 5.8: Descrição Externa de um Operador ADT. 29
Figura 6.1: Processo de Engenharia Genérico do OORT. 33
Figura 7.1: Acesso aos Elevadores num dos Pisos Intermédios. 39
Figura 7.2: Interior de um Elevador. 40
Figura 8.1: Diagrama de Classes OMT do Módulo Edifício do SES. 48
Figura 8.2: Diagrama de Classes OMT do Módulo Arquitectura do SES. 49
Figura 8.3: Diagrama de Classes OMT do Módulo Piso do SES. 49
Figura 8.4: Diagrama de Classes OMT do Módulo Elevador do SES. 50
Figura 8.5: Diagrama de Classes OMT do Módulo Eixo do Elevador do SES. 50
Figura 8.6: Diagrama de Classes OMT do Módulo Passageiro e Potencial Passageiro do SES. 51
Figura 8.7: Diagrama de Classes OMT do Módulo Distribuição de Chamadas do SES 51
Figura 8.8: Diagrama de Classes OMT do Módulo Supervisão de Eventos do SES 52
Figura 8.9: Diagrama de Classes OMT do Módulo Botões do SES. 52
Figura 8.10: Diagrama de Instâncias OMT do Módulo Edifício de Quatro Pisos com Dois Elevadores 53

Figura 8.11: Hierarquia de Cenários do SES. 55
Figura 8.12: Hierarquia do Sub-cenário Viagens do SES 56
Figura 8.13: MSC do Arranque de um Elevador do SES. 58
Figura 8.14: MSC de uma Chamada Efectiva. 58
Figura 8.15: MSC de uma Chamada Redundante. 58
Figura 8.16: MSC do Início de uma Viagem de um Elevador Originado pela Recepção de um Destino. 59
Figura 8.17: MSC da Chegada de um Elevador a um Piso. 59
Figura 8.18: MSC da Passagem por um Piso. 60
Figura 8.19: MSC da Porta do Piso Não Abre. 60
Figura 9.1: Diagrama de Interligação do SES. 68
Figura 9.2: Declaração de Constantes ao Nível do SES. 68
Figura 9.3: Declaração de Tipos ao Nível do SES. 68
Figura 9.4: Declaração de Sinais ao Nível do SES. 69
Figura 9.5: Diagrama de Interligação do Bloco Central. 69
Figura 9.6: Declaração PR dos Sinais do Bloco Central. 70
Figura 9.7: Diagrama de Interligação do Tipo de Bloco Piso. 70
Figura 9.8: Diagrama de Interligação do Tipo de Bloco Elevador. 70
Figura 9.9: Declarações PR dos Sinais do Tipo de Bloco Elevadores. 71
Figura 9.10: Hierarquia SDL do SES. 71
Figura 10.1: Diagrama de Estado OMT do Processo PortaPiso. 76
Figura 10.2: Diagrama de Processo SDL do Painel de Chamada. 78
Figura 10.3: Diagrama de Processo SDL da Porta do Piso. 79
Figura 10.4: Hierarquia de Cenários da Correspondência entre Processos 82
Figura 10.5: MSC da Correspondência entre Pais 82
Figura 10.6: MSC da Correspondência entre Filhos 82
Figura 10.7: Diagrama de Classes OMT do projecto detalhado do SES. 86
Figura 11.1: Refinamento Hierárquico de MSCs. 91
Figura 11.2: MSC Detalhado ao Nível dos Blocos do SES. 94
Figura 11.3: MSC Detalhado ao Nível dos Processos do Bloco Pisos. 94
Figura 11.4: MSC Detalhado ao Nível dos Processos do Bloco Central. 95
Figura 11.5: MSC Detalhado ao Nível dos Processos do Bloco Elevadores. 95
Figura A.1 : Diagrama de Interligação do SES. 109
Figura A.2 : Declaração de Sinais ao Nível do SES. 109
Figura A.3 : Declaração das Constantes ao Nível do SES. 110
Figura A.4 : Declaração de Tipos ao Nível do SES. 110
Figura A.5 : Diagrama de Interligação do Bloco Central. 111
Figura A.6 : Declaração de Sinais ao Nível do Bloco Central. 111
Figura A.7 : Diagrama de Interligação do Bloco Acesso do Piso. 112
Figura A.8 : Diagrama de Interligação do Bloco Elevador 113

Figura A.9 : Declarações PR ao Nível do Bloco Elevador. 113
Figura A.10 : Diagrama de Processo SDL do Gnomo (1/3). 114
Figura A.11 : Diagrama de Processo SDL do Gnomo (2/3). 115
Figura A.12 : Diagrama de Processo SDL do Gnomo (3/3). 116
Figura A.13 : Diagrama de Processo SDL do Procedimento Envia Mensagem do Gnomo. 116
Figura A.14 : Diagrama de Processo SDL do Supervisor (1/2). 117
Figura A.15 : Diagrama de Processo SDL do Supervisor (2/2). 118
Figura A.16 : Diagrama de Processo SDL do Painel de Chamada. 119
Figura A.17 : Diagrama de Processo SDL da Porta do Piso. 120
Figura A.18 : Diagrama de Processo SDL do Procedimento Abre da Porta do Piso. 120
Figura A.19 : Diagrama de Processo SDL do Procedimento Fecha da Porta do Piso. 120
Figura A.20 : Diagrama de Processo SDL do Elevador (1/5). 121
Figura A.21 : Diagrama de Processo SDL do Elevador (2/5). 122
Figura A.22 : Diagrama de Processo SDL do Elevador (3/5). 123
Figura A.23 : Diagrama de Processo SDL do Elevador (4/5). 124
Figura A.24 : Diagrama de Processo SDL do Elevador (5/5). 125
Figura A.25 : Diagrama de Processo do Procedimento Pedidos Pendentes do Elevador. 125
Figura A.26 : Diagrama de Processo SDL do Painel de Destinos. 126
Figura A.27 : Diagrama de Processo SDL da Porta do Elevador (1/2). 127
Figura A.28 : Diagrama de Processo SDL da Porta do Elevador (2/2). 128
Figura A.29 : Diagrama de Processo SDL do Monitor do Cabo. 129

Lista de abreviaturas e símbolos

ADT	*Abstract Data Type*
API	*Aplication Programming Interface*
ASN.1	*Abstract Syntax Notation One*
CASE	*Computer-Aided Software Engineering*
DBMS	*Data Base Management System*
GUI	*Gaphical User Interface*
MSC	*Message Sequence Chart*
OMG	*Object Management Group*
OMT	*Object Modelling Technique*
OO	Orientado ao Objecto
OODBMS	*Object-Oriented* DBMS
OORT	*Object-Oriented Real-time Techniques*
OOSE	*Object-Oriented Software Engineering*
PES	Processo de Engenharia de Sistemas
PR	*Phrasal Representaion*
SDL	*Specification and Decription Language*
SES	Sistema Elevatório Simples
SOTR	Sistema Operativo de Tempo-Real
TR	Tempo-Real
TTCN	*Tree and Tabular Combined Notation*
UML	*Unified Modeling Language*

Parte I

Métodos de Especificação para Tempo-Real

Nesta parte, começa-se por referir as características do desenvolvimento de sistemas de tempo-real e das ferramentas de suporte. Depois, e para estabelecer uma referência, apresenta-se um processo genérico de engenharia de sistemas. A seguir faz-se um resumo das notações OMT, MSC e SDL que são utilizadas no OORT. Finalmente, mostra-se como é que estas três notações servem para as diferentes actividades envolvidas no desenvolvimento de *software* para aplicações de tempo-real, e quais os benefícios que se podem obter em cada fase e na globalidade do ciclo-de-vida do projecto.

1 Introdução

1.1 O Desafio

Os sistemas de Tempo-Real (TR) actuais são naturalmente muito complexos. O seu desenvolvimento e manutenção exige uma quantidade enorme de recursos financeiros, humanos e técnicos.

Estes sistemas têm características únicas, tais como, a necessidade de suportar e operar em plataformas heterogéneas distribuídas, executarem em paralelo sob restrições temporais críticas e integrar diversas formas de tecnologia (desde bases de dados a *interfaces* gráficas, redes, *drivers* de *hardware*, *etc.*).

As pressões económicas não são menores. Os requisitos do mercado em termos de melhor qualidade e no menor período de tempo estão constantemente a aumentar. Simultaneamente, os investimentos feitos na construção destes sistemas fazem com que a facilidade de manutenção e reutilização sejam factores críticos na sua rentabilidade.

A componente de *software* dos sistemas de TR é, na maioria dos casos, a parte mais crítica e dispendiosa. No sector das telecomunicações, por exemplo, os custos do *software* representam cerca de 70% do custo total do desenvolvimento do sistema.

A equipa de *software* de tempo-real bem sucedida integra a melhor tecnologia e recursos humanos com um processo de desenvolvimento planeado ao pormenor. As empresas que estiverem aptas a reconhecer e ultrapassar os desafios de controlar este processo, conseguirão manter uma significativa vantagem competitiva no mercado. Por estas razões, a especificação de *software* é uma actividade de importância capital. Consequentemente, a metodologia empregue nesta actividade torna-se decisiva para o (in)sucesso, sendo de esperar que esta ajude a ultrapassar os obstáculos citados. Mais concretamente, um bom método de especificação deve proporcionar soluções para:

- Diminuir a complexidade através da modelação de cada aspecto do sistema (arquitectura, dados, comportamento) com notações gráficas e organização hierárquica.
- De preferência, utilizar normas internacionais apropriadas para responder às características únicas dos sistema TR, de forma a defender o investimento.
- Permitir o desenvolvimento de sistemas de alta qualidade pela aplicação de técnicas de verificação (o sistema funciona?) e validação (o sistema faz o que é suposto fazer?) avançadas o mais cedo possível no ciclo de desenvolvimento.
- Entregar o sistema em prazos reduzidos através da automatização da geração de código fiável e de alto desempenho, pronto a executar em diversos ambientes de tempo-real.

- Fazer a manutenção das aplicações ao nível do projecto gráfica e reduzir a manutenção de código fonte ao mínimo, de modo a permitir maiores níveis de reutilização.
- Automatizar e controlar todo o processo de desenvolvimento através do rastreio (*traceability*) completo desde a análise até à geração de código, teste e manutenção.

1.2 As Tecnologias

É normalmente aceite que a tecnologia OO (Orientada ao Objecto) é **a** solução de desenvolvimento do futuro próximo. Com o constante aumento do tempo-de-vida das aplicações, os novos sistemas são obrigados a evoluir e partes deles são susceptíveis de serem reutilizadas noutros projectos. A aplicação de OO no desenvolvimento ajuda os analistas e projectistas a integrarem a noção de durabilidade do *software* no processo de desenvolvimento e a considerarem a reutilização a todos os níveis graças a conceitos e mecanismos poderosos como encapsulamento, ocultação de informação, herança, ligação (*binding*) dinâmica e polimorfismo. Adicionalmente, as técnicas OO suportam o paradigma cliente/servidor, que facilita a reutilização de partes de aplicações existentes para construir sistemas mais complexos.

Como é obvio, os sistemas TR, designadamente para aplicações distribuídas, são mais difíceis de projectar porque têm que respeitar restrições adicionais, nomeadamente:

- gestão de recursos físicos;
- concorrência interna;
- distribuição de *software* (que requer protocolos de comunicação de controlo e de dados);
- desempenho temporal ou comportamento de segurança-crítica (*safety-critical*).

As técnicas OO puras geralmente não conseguem respeitar estas restrições e não proporcionam aos projectistas construções adequadas aos sistemas TR. Todavia, já existem várias técnicas de desenvolvimento de sistemas TR e que são actualmente utilizadas. Normalmente, baseiam-se em definições não ambíguas, isto é, as regras de leitura são as mesmas para qualquer leitor. Por isso, as descrições de sistemas através dessas técnicas, ao contrário das bem sucedidas mas informais descrições baseadas em OO, podem ser simuladas e validadas antes da implementação, resultando uma geração de código significativamente automatizada. Estas técnicas são geralmente denominadas **técnicas de descrição formal** ou **notações formais**.

1.3 Opção pelo OORT

As notações a utilizar nas diversas actividades que constituem o PES (Processo de Engenharia de Sistemas) têm que ser escolhidas de forma a abrangerem todas as fases e aumentarem a consistência global evitando descontinuidades entre elas. As necessidades dos analistas e de quem desenvolve o

sistema são bastantes diferentes. Durante a fase de análise, os especialistas preferem notações gráficas leves e simples. Por outro lado, os projectistas tiram partido de notações não ambíguas e que apresentem um bom poder de expressão para cobrir toda a gama de problemas susceptíveis de aparecer no desenvolvimento distribuídos TR.

O método *Object-Oriented Real-Time Techniques* (OORT) baseia-se na utilização combinada da notação *Object Modelling Technique* (OMT), que é uma norma de-facto, e as notações *Specification and Decription Language* (SDL) e *Message Sequence Chart* (MSC), que são recomendações ITU-T. Cada uma delas tem propósitos bem definidos, e o objectivo global do OORT é suportar um processo iterativo que cubra todos os aspectos do desenvolvimento de sistemas sem qualquer salto ou descontinuidade de paradigma. Na fase de análise, os diagramas de classes OMT adequam-se bem à modelação por objectos, e um subconjunto da notação MSC pode ser utilizada para a modelação dinâmica. O projecto da arquitectura e o projecto detalhado requerem uma combinação de SDL e de diagramas de classes OMT para, respectivamente, especificar os objectos concorrentes e descrever os objectos passivos. A notação MSC é utilizada na sua totalidade para o projecto de testes.

O método OORT, suportado pela ferramenta *Object*GEODE, satisfaz o que se considera serem os principais requisitos necessários para o desenvolvimento de aplicações distribuídas de tempo-real (enumerados em 1.1).

1.4 Convenções da Notação Utilizada

As palavras que se pretendem salientar pela sua importância, ou porque representam algum conceito introduzido, aparecem a **negrito**. Os termos adoptados de outras nomenclaturas, como a anglo-saxónica ou latim, aparecem a *itálico*. Os nomes referentes a entidades das várias linguagens abordadas aparecem no tipo `Courier New`.

2 Actividades da Engenharia de Sistemas

O propósito deste capítulo não é propor novas fases na engenharia de sistemas, mas antes fornecer uma ideia clara e consensual do que é um processo de engenharia de sistemas. A apresentação do OORT será baseada no esquema genérico aqui descrito.

2.1 Esquema Genérico

O esquema genérico define-se em termos de actividades, conforme ilustra a Figura 2.1. Este esquema é suficientemente genérico e flexível para ser compatível com os PESs tradicionalmente utilizados pela comunidade industrial, onde os tipos de ciclo-de-vida mais divulgados são: a queda-de-água (*waterfall*), o V, espiral e o mais recente X (uma extensão do V que inclui actividades de reutilização). Adicionalmente, este esquema suporta o **desenvolvimento iterativo** (ou **desenvolvimento incremental**) e a **prototipagem rápida**. Também abrange os PESs promovidos pelos principais métodos OO existentes, que são: [Rum91], [Jac92], e [SDL93].

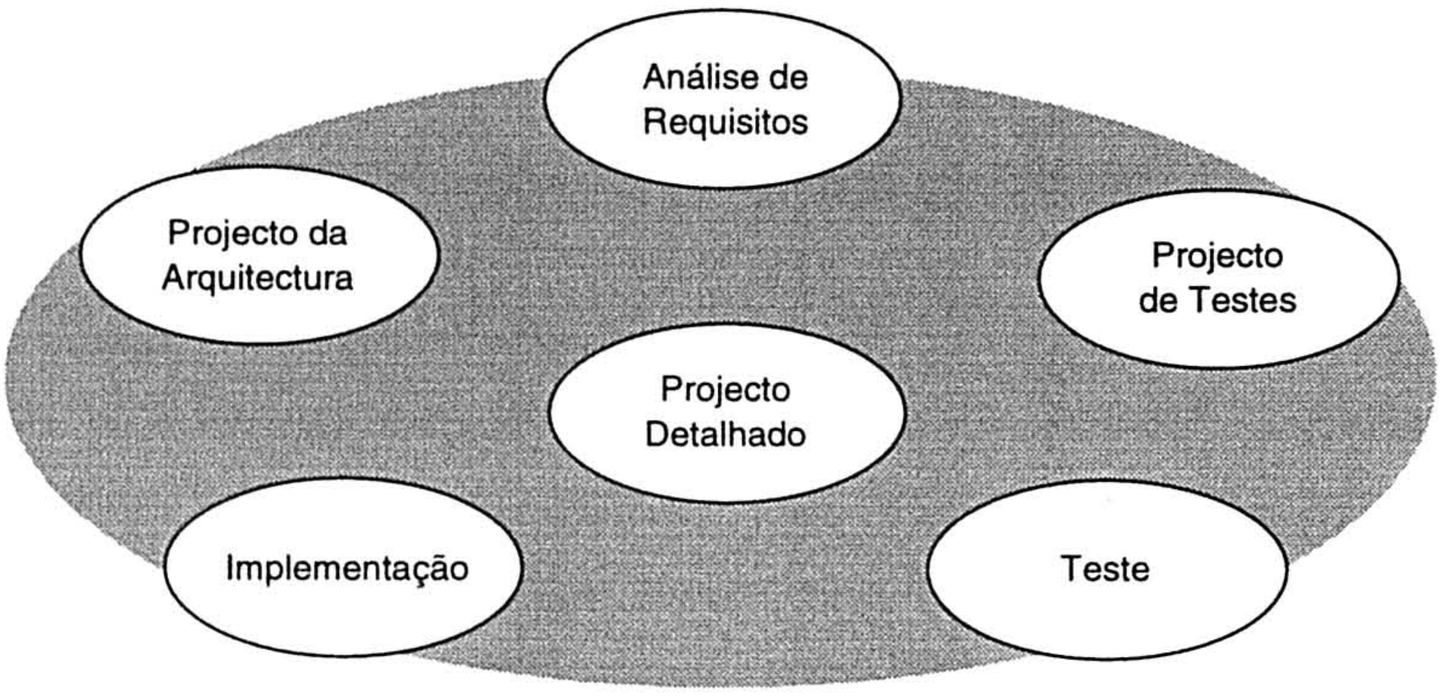


Figura 2.1: Esquema Genérico do PES.

2.2 Análise de Requisitos

A **análise de requisitos** é o estudo inicial do sistema a ser construído. Consiste em modelar os requisitos do utilizador e o ambiente real em que o sistema se vai inserir, e com o qual vai interagir.

No desenvolvimento de sistemas TR, a análise do sistema debruça-se sobre a reacção esperada do **sistema** em resposta a estímulos produzidos por um conjunto de **actores** que compõem o ambiente. As reacções são descritas de dois pontos-de-vista:

- do ponto de vista dos actores influenciadores relacionados com o sistema, com o qual trocam informação;
- do ponto de vista das sequências de controlo trocados entre o sistema e os actores.

Estas descrições são conseguidas, respectivamente, através da **modelação por objectos** e da **modelação dinâmica**. Quando a actividade de análise está completa, dois modelos estão disponíveis: um modelo de objectos e um modelo dinâmico. Uma preocupação importante é manter os dois modelos coerentes. Os modelos de análise servirão como referência entre o cliente e o fornecedor do sistema, bem como entre o fornecedor do sistema e o construtor do mesmo.

Os analistas devem concentrar o seu modelo em **o que** o sistema tem que fazer, os serviços esperados, de forma a poderem apresentar várias soluções possíveis para um dado problema. Têm que evitar fazer assunções ou conjecturas prévias sobre a arquitectura do sistema.

2.3 Projecto da Arquitectura

O objectivo da actividade de **projecto da arquitectura** é especificar a **arquitectura lógica** do sistema. O termo **arquitectura lógica** refere-se à arquitectura do *software* da aplicação, em oposição ao termo **arquitectura física** que significa a arquitectura do plataforma real onde a aplicação será executada.

Os requisitos elaborados na fase anterior não constituem a única entrada a considerar na fase de projecto. Os aspectos económicos e técnicos de sistemas completamente novos também devem ser levados em conta, de forma a optimizar o desempenho e a fiabilidade de acordo com o orçamento disponível. No que diz respeito a aplicações que irão cooperar com sistemas já existentes, os projectistas têm que considerar certas restrições: protocolos de comunicação, soluções para a persistência de dados, *interface* com *Application Program Interfaces* (APIs) existentes, *etc.* Quando se criam variantes de produtos normalizados, as considerações sobre reutilização devem guiar o projecto, de modo a tornar possível a reutilização de uma arquitectura validada, ou então, dos componentes de *software* disponíveis.

2.4 Projecto Detalhado

Uma vez completa a arquitectura do sistema, os objectos que compõem essa arquitectura têm que ser pormenorizados de forma a serem facilmente implementados.

Identificam-se duas categorias de objectos:

- **objectos concorrentes**, que contêm a sua própria sequência de controlo e que é executada em concorrência com a de outros;
- **objectos passivos**, que servem como recursos de computação utilizados pelos objectos anteriores.

Esta distinção conduz a dois tipos de projecto OO. Os objectos concorrentes são normalmente descritos sob a forma de máquinas de estados finitos, ao passo que os objectos passivos são descritos sob a forma de classes compostas por atributos e métodos mas sem sequência interna.

2.5 Projecto de Testes

Tipicamente, definem-se dois tipos de testes:

- **testes unitários**, que permitem aos projectistas assegurarem que os diferentes componentes do sistema comportam-se individualmente conforme o esperado;
- **testes de integração**, que permitem aos projectistas verificarem o funcionamento da aplicação obtido pela integração gradual dos componentes implementados.

O projecto dos testes de integração e o projecto da arquitectura estão intimamente interligados e podem ser realizados proveitosamente em simultâneo. Os testes de integração pormenorizam sequências de interacção que ocorrem entre elementos definidos na arquitectura e, por isso, podem servir para verificar se a arquitectura do sistema está completa. Por outro lado, o processo de refinamento é facilitado, já que os testes de integração disponíveis em determinado nível podem ser utilizados como requisitos a cumprir quando os componentes são refinados em elementos hierarquicamente inferiores. Consequentemente, estes testes servem para verificar a consistência da arquitectura detalhada.

Como o projecto dos testes unitários e o projecto detalhado são actividades fortemente interrelacionadas, também podem ser realizadas em simultâneo de modo que os testes unitários sirvam de requisitos a ser satisfeitos quando se detalha cada componente terminal.

A actividade de projecto de testes pode prolongar-se por muito tempo, particularmente no caso de sistemas de segurança crítica (*safety-critical*) ou de missão crítica (*mission-critical*). O esforço despendido nesta actividade pode ser equivalente ao das actividades de análise e projecto. Por isso, as notações a utilizar no PES têm que fornecer uma ajuda significativa nas actividades de teste, como, por exemplo, a geração automática dos testes.

2.6 Implementação (*Targeting*)

A implementação de sistemas TR, *e.g.* sistemas embebidos, depende muito da arquitectura física do ambiente no qual o sistema será inserido. Em particular, os objectos concorrentes têm que ser concebidos para operarem perfeitamente, tanto com uma plataforma final de arquitectura distribuída como com o Sistema Operativo de Tempo-Real (SOTR) subjacente. Os objectos passivos são muito menos dependentes da plataforma e a linguagem de programação escolhida afecta somente a fase de implementação.

A utilização de métodos formais ao longo do PES melhora francamente a actividade de implementação, porque as descrições formais podem ser automaticamente traduzidas em código totalmente executável, com desempenho melhorado, testado e verificado em níveis superiores.

2.7 Teste

A fase de teste consiste em executar a aplicação desenvolvida seguindo as sequências de teste definidas nas fases de análise e projecto, primeiro no local de desenvolvimento e depois na plataforma final.

Quem realiza esta actividade tem que verificar que a aplicação passa os testes unitários e de integração, e que cumpre os requisitos expressos pelo utilizador. Para isso, esta actividade utiliza como entradas os resultados da fase de projecto de testes e, também, o modelo dinâmico elaborado na fase de análise dos requisitos.

Esta actividade pode ser parcialmente automatizada, dependendo das notações empregues nas fases de análise e projecto. Por exemplo, produzir planos de teste numa linguagem como TTCN (*Tree and Tabular Combined Notation*) a partir de uma notação de projecto, automatiza os testes e a análise de resultados.

2.8 O Processo de Engenharia de Sistemas

A ordem cronológica de realização das actividades é estipulada pelo processo de desenvolvimento. Actualmente, o ciclo-de-vida mais comum na indústria é o ciclo-de-vida V, ilustrado na Figura 2.2. De acordo com ele, as actividades são realizadas sequencialmente: a aplicação tem que estar completamente analisada antes de ser concebida, e estar totalmente concebida antes de ser implementada.

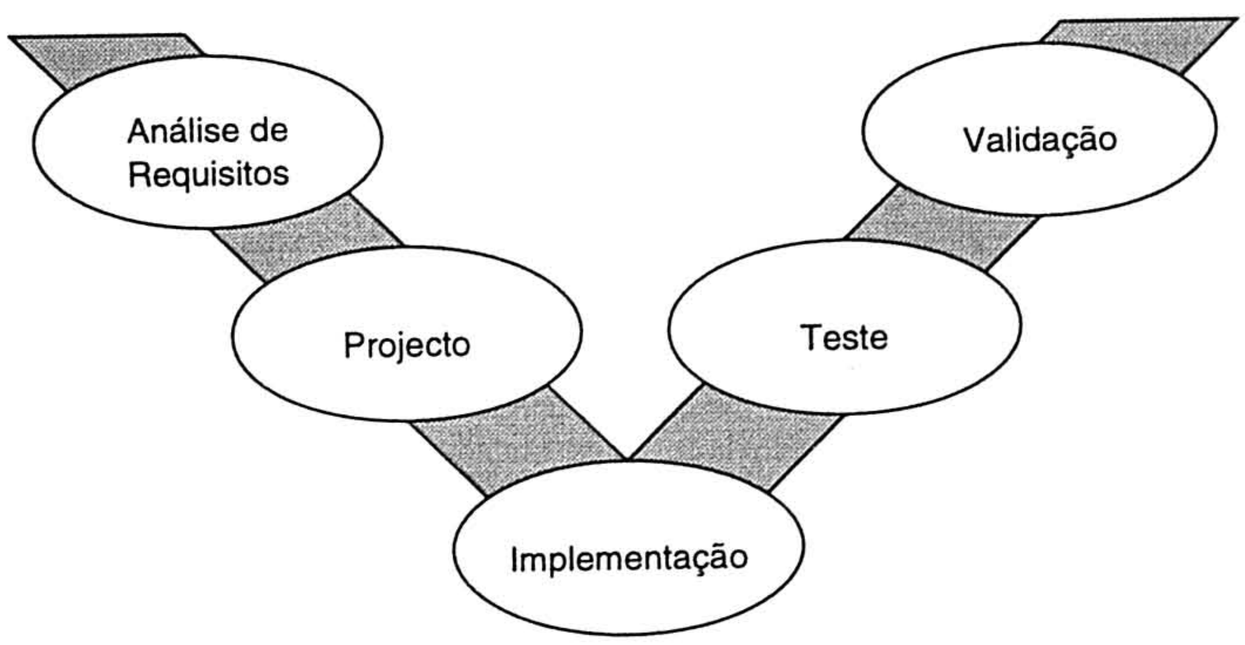


Figura 2.2: Ciclo-de-vida V.

O termo **validação**, empregue no ciclo-de-vida V, corresponde à ultima actividade do processo, que para o utilizador consiste na confrontação do produto final com os requisitos. Este conceito de validação difere da noção de validação antecipada da arquitectura, tal como é suportada pelos processos iterativos (conceito de **Verificação e Validação**).

Um ciclo-de-vida unidireccional não permite aos projectistas verificar a viabilidade técnica genérica da aplicação no início do projecto. Por exemplo, limitações que não permitam que o código fonte proporcione o desempenho especificado, só podem ser detectados durante a fase de implementação. Quanto mais tarde for encontrada e corrigida uma discrepância entre requisitos e produto final, mais afectará o que já foi projectado e maior será o custo global da correcção necessária.

O ciclo-de-vida V, sendo uma abordagem fortemente **de-cima-para-baixo** (*top-down*), não tem flexibilidade suficiente para permitir que componentes reutilizados cumpram as escolhas técnicas definidas mais abaixo no processo de desenvolvimento. Além disso, este ciclo-de-vida pode gerar dificuldades nas passagens de fase e não suporta a prototipagem rápida.

Com o objectivo de manter sob controlo as actividades de desenvolvimento da aplicação recomenda-se seguir um processo iterativo, conforme mostra a Figura 2.3.

Como todas as actividades são realizadas numa parte do sistema ainda no início do processo de desenvolvimento, as dificuldades técnicas são geralmente detectadas de forma a que as respostas apropriadas sejam desde logo fornecidas. Mais ainda, os membros da equipa são treinados mais progressivamente na utilização dos vários métodos e ferramentas seleccionadas para o projecto.

O processo iterativo torna a reutilização mais fácil. A possibilidade de avaliar quanto das arquitecturas de *software* existentes ou dos módulos de código pode ser reutilizado, tem uma influência capital nas fases de projecto e análise: partes da nova aplicação baseadas em código reutilizado podem ser excluídas da lista de componentes a serem iniciados do zero. De igual modo, os objectos desenvolvidos em iterações prévias podem ser reutilizados mais tarde, o que reduz o esforço de projecto necessário em cada iteração.

Além disso, um processo de desenvolvimento iterativo permite realizar prototipagem rápida, ou seja, no plano de trabalho do projecto podem executar-se antecipadamente partes do sistema (manual, ou automaticamente).

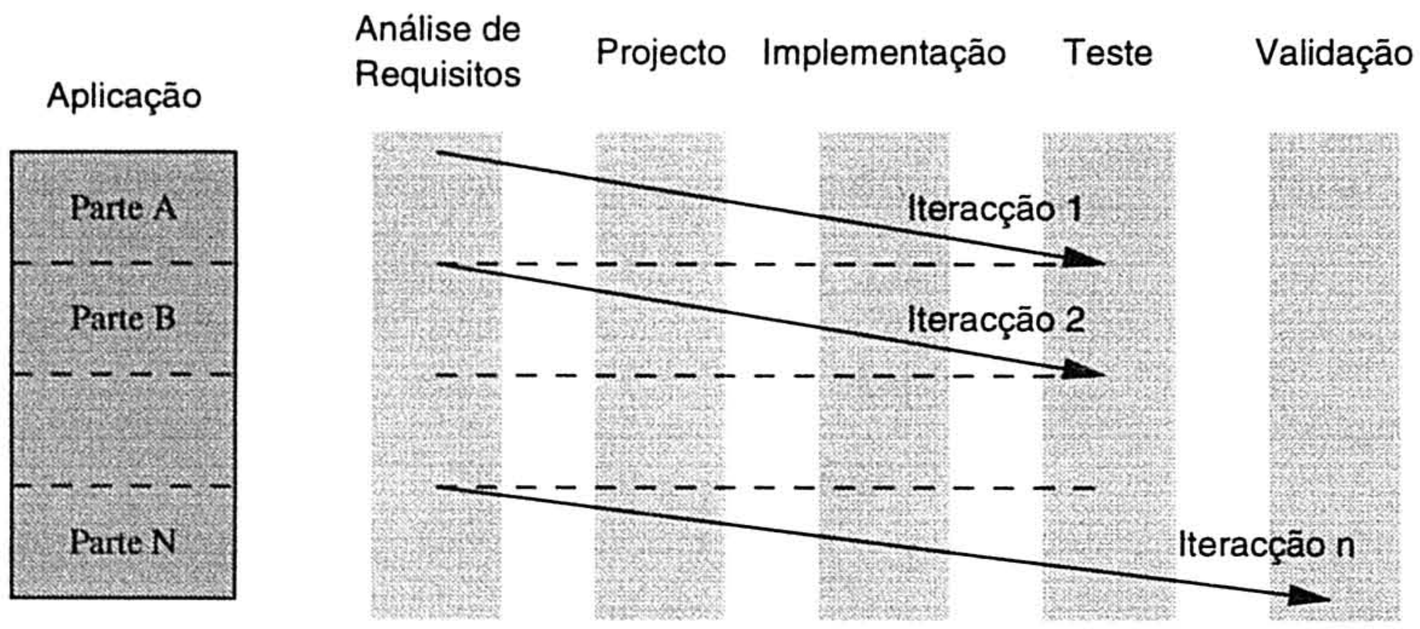


Figura 2.3: Processo Iterativo.

Finalmente, seguir um processo iterativo permite à empresa gerir melhor o seu projecto, de forma mais visível. Porém, a tarefa de divisão da aplicação em sub-partes tem que ser cuidadosamente ponderada, porque a aferição do progresso do projecto depende em larga escala disso. Tipicamente, uma aplicação pode ser dividida de acordo com o seguinte padrão: funções nucleares, situações críticas e funções repetitivas.

3 Resumo da Linguagem OMT

A técnica OMT foi definida por James Rumbaugh *et al.* em [Rum91], e cobre os três aspectos da análise de sistemas: **modelação por objectos**, **dinâmica** e **funcional**. O projecto do *software* é suportado pela mesma notação orientada ao objecto, assegurando, assim, uma boa continuidade entre as duas fases. Contudo, os próprios autores da OMT reconhecem que esta é pouco adequada para o projecto da arquitectura, projecto de testes e implementação dos sistemas. A OMT é melhor para o desenvolvimento de sistemas de informação, não oferecendo items gráficos específicos para o desenvolvimento de tempo real.

3.1 Modelação Por Objectos

Um **modelo de objectos** OMT é composto por dois tipos de diagramas que descrevem o sistema em termos de **classes** e **objectos** (**instâncias de classes**), respectivamente: **diagrama de classes** e **diagrama de instâncias**.

O modelo de objectos OMT organiza-se em **módulos** que correspondem a diagramas de classes ou diagramas de instâncias. Tal como também Yourdon defende em [You94], a tendência predominante no campo das metodologias OO é a abordagem **multimodal**, constituída por diferentes diagramas que ilustram diversos aspectos do sistema. Isso é bastante conveniente, pois consoante a perspectiva pode dar-se atenção: aos objectos essenciais e à hierarquia a que pertencem; ao comportamento interno de cada objecto e ao ciclo de vida que exibem; e à comunicação entre os objectos.

3.1.1 Diagrama de Classes

Os diagramas de classes baseiam-se na conhecida notação de **relação-entre-entidades** (*entity-relationship*) com a introdução dos conceitos OO: operações e herança. Estes diagramas são utilizados para expressar os domínios da aplicação e dos serviços.

A Figura 3.1 mostra parte de um possível diagrama de classes onde o domínio da aplicação é o transporte de passageiros entre pisos de um edifício. Os diagramas de classes são compostos por:

- **classes**, *e.g.* `Elevador`, com **atributos** `Numero`, `Piso` e `Sentido`, e com operações `ServeChamada` e `ServeDestino`;
- **associações**, *e.g.* `Alberga` entre `Piso` e `PotencialPassageiro`;
- **agregações** entre classes, como entre `Edificio` e `Piso`, e entre `Edificio` e `SES`;

- **hierarquias de herança**, entre `Utilizador` e `Passageiro`, e entre `Utilizador` e `PotencialPassageiro`.

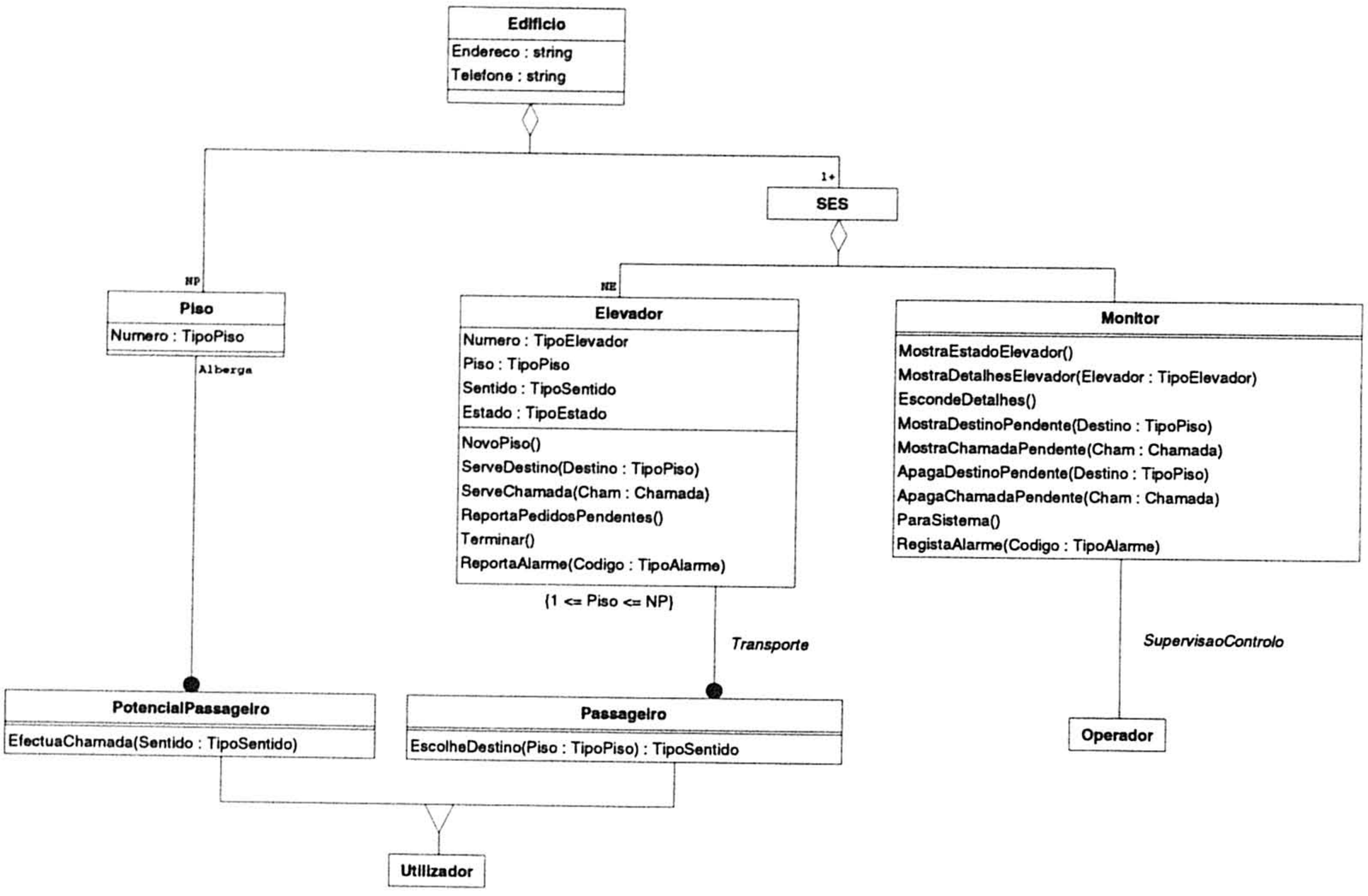


Figura 3.1: Exemplo de um Diagrama de Classes.

Uma classe pode ser **abstracta**, se não puder ser instanciada (utiliza-se para construir a árvore de herança do domínio), ou **derivada**, se é redundante quando comparada com a informação já modelada (geralmente enfatizam a manipulação frequente de dados). Uma classe pode receber texto adicional, **restrições**, como `1<=Piso<=NP`.

Cada atributo pode ter um **tipo**, *e.g.* `Piso` que é do tipo `TipoPiso`. O tipo de um atributo não pode ser uma classe, para isso recorre-se a uma associação entre as classes. Um atributo pode ser **meta** se o seu valor é comum a todas as instâncias existentes da classe, como, por exemplo, o número de instâncias existentes da classe. Um atributo pode ser **derivado** (distinguido pelo prefixo /) se for redundante com informação já modelada. Classes derivadas são geralmente utilizadas para chamar a atenção para valores frequentemente calculados a partir de outros dados existentes no diagrama. Finalmente, um atributo pode ser **chave** se identifica de forma única uma instância entre as outras. Este conceito não é necessário na fase de análise de requisitos, mas sim na fase de projecto, já que na análise assume-se que todas as instâncias podem ser acedidas independentemente, mesmo que a forma de o fazer não seja conhecida nesse momento.

Uma associação representa uma dependência entre duas classes e não tem uma direcção única (as duas classes são acedidas de maneira indiferente). Uma associação pode ter um nome, *e.g.* `Transporte` para a associação `Elevador-Passageiro`, ou então o **papel** de cada classe pode ser expresso: o `Piso Alberga` o `PotencialPassageiro`. Todavia, dar nomes às associações é opcional. Cada associação tem uma **multiplicidade** que é a relação entre o número de instâncias das classes por ela associadas.

Existem 4 multiplicidades predefinidas: 0 ou 1 (bola vazia ○), exactamente 1 (opção por defeito, não representada graficamente), 0 ou mais (bola preenchida ●) e 1 ou mais (1+). Mas também são permitidas outras multiplicidades à escolha do utilizador. Uma associação pode ser derivada (assinalada com /) se for redundante, isto é, se existir outro caminho com o mesmo significado entre as duas classes, possivelmente através de outras classes; normalmente evidenciam o acesso frequente às classes.

Uma agregação representa um tipo de associação em que uma classe contém outras classes como sendo seus componentes. A agregação implica que alterações realizadas sobre a classe contentora afectam as classes contidas. Em particular, apagar um objecto contentor implica apagar todos os seus objectos componentes. As agregações são especialmente úteis para refinar objectos grandes. Na Figura 3.1, o SES contém vários componentes, e quando for desligado todos os seus componentes são também apagados. No entanto, convém esclarecer que a agregação não é uma divisão hierárquica, porque não é uma substituição; em termos de instâncias, os objectos agregadores coexistem com os objectos seus agregados.

A herança é o meio tradicional para implementar a reutilização: `Passageiro` e `PotencialPassageiro` herdam todas as propriedades (atributos, operações, associações e agregações) de `Utilizador`. A herança múltipla é permitida na análise mas não é aconselhável, pois a sua utilização não é consensual[1].

3.1.2 Diagrama de Instâncias

Os diagramas de instâncias podem ser interpretados como fotografias do sistema em que se representam todos os objectos de cada classe que existem num determinado momento. A Figura 3.2 apresenta um diagrama de instâncias típico correspondente ao diagrama de classes anterior. É composto por:

- instâncias de classes chamadas **objectos** (rectângulos com cantos arredondados), *e.g.* `Piso` e `Edificio`;
- instâncias de associações chamadas **ligações** (linhas), *e.g.* `SupervisaoControlo`.

Os diagramas de instâncias podem servir para detalhar estados específicos do sistema, *e.g.* arranque, terminações anormais, *etc.*, ou para clarificar diagramas de classes complexos, principalmente os que contêm redes confusas de associações com diversas multiplicidades.

3.2 Modelação Dinâmica

Um modelo dinâmico OMT é composto por dois tipos de diagramas: **traçado de eventos** para descrever cenários de eventos em sequência cronológica, e **diagramas estado-transição** para descrever completamente o comportamento dos objectos.

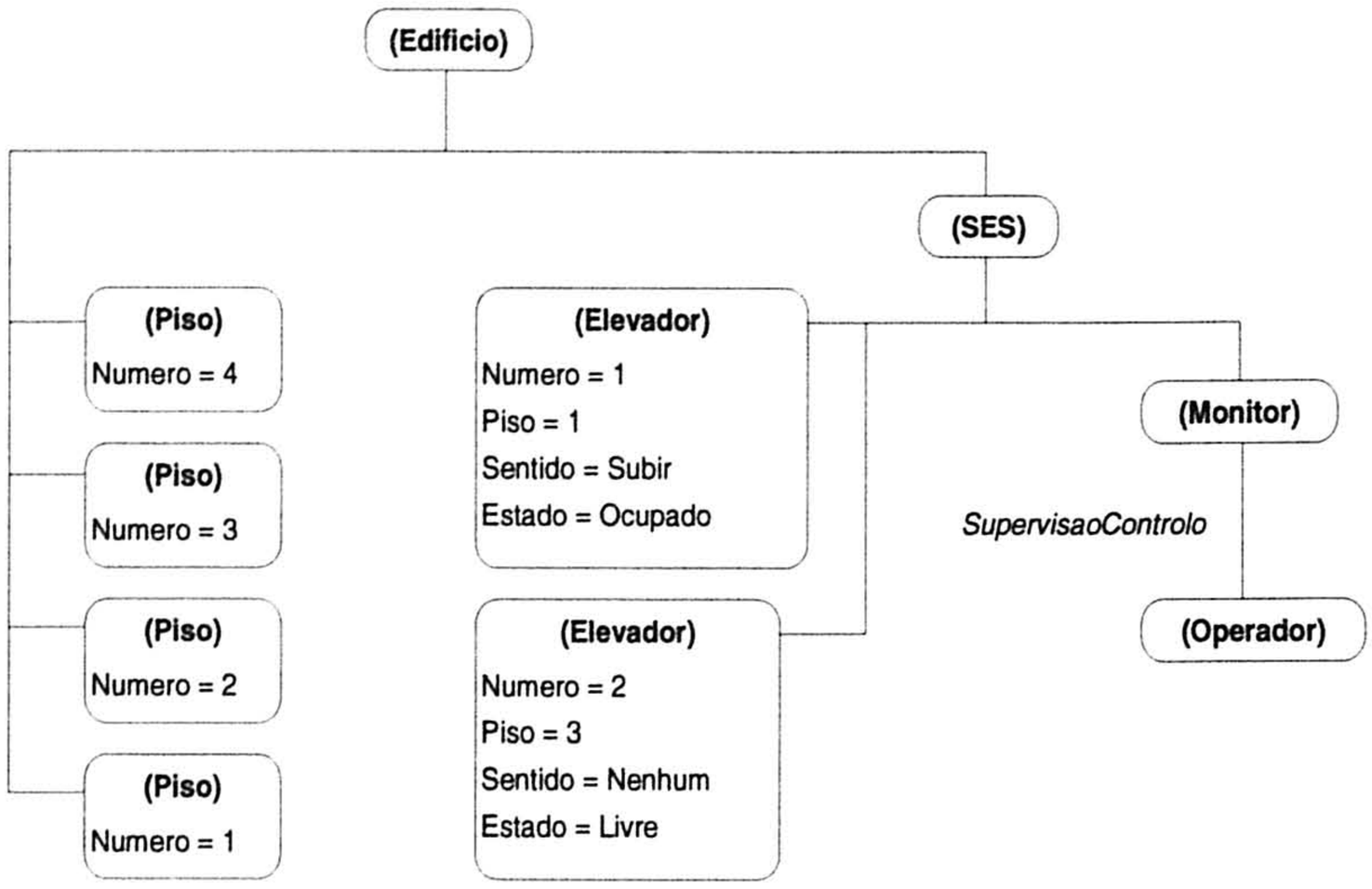


Figura 3.2: Diagrama de Instâncias relativo à Figura 3.1.

3.2.1 Traçado de Eventos

Os traçados de eventos descrevem sequências de eventos que ocorrem entre objectos, fornecendo assim uma perspectiva dinâmica da aplicação.

A Figura 3.3 ilustra o diagrama de eventos típico da recepção de uma chamada pelo SES. Estes diagramas são compostos por:

- **instâncias de classes** (barras verticais), *e.g.* `Utilizador` e `SES`;
- **eventos com informação** (setas horizontais), *e.g.* `BotaoChamadaPremido` e `ComandaLuz`.

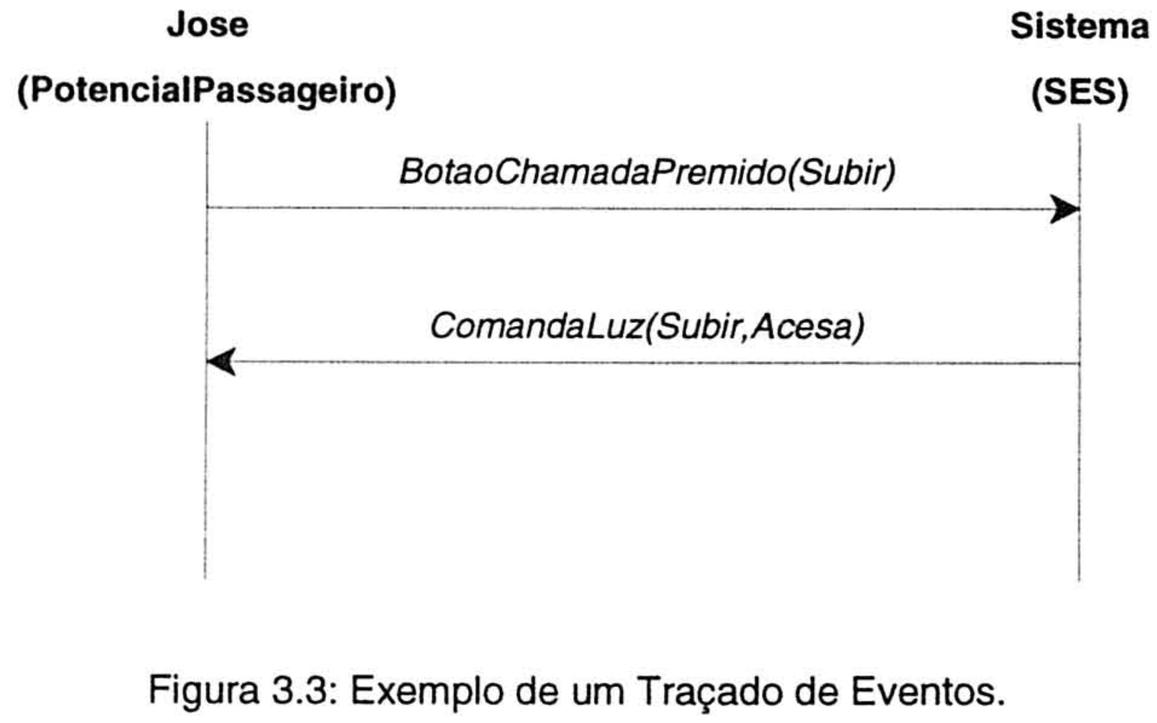


Figura 3.3: Exemplo de um Traçado de Eventos.

---

[1] [You94] A herança múltipla tem as desvantagens de perder simplicidade conceptual e de implementação e, principalmente, a possibilidade de haver conflitos de nomes. Por outro lado, tem as vantagens de tornar mais poderosa a especificação de classes (não restringindo a hierarquia à forma de árvore) e aumentar as possibilidades de reutilização.

3.2.2 Diagramas Estado-Transição

Um diagrama estado-transição descreve completamente o comportamento de um objecto. Existe um diagrama por cada classe concorrente. Estes diagramas complementam os traçados de eventos, fornecendo aos analistas um modelo dinâmico exaustivo.

Os diagramas estado-transição OMT baseiam-se na notação dos diagramas de estado de Harel. A notação de Harel é um melhoramento da tradicional notação de Máquinas de Estados Finitos com refinamentos de estados e concorrência interna. Na OMT foi incluído apenas um subconjunto dessa notação.

A Figura 3.4 é um exemplo de um diagrama estado-transição que descreve o comportamento das portas de cada piso de um sistema de elevadores. Estes diagramas são compostos por:

- **estados** (rectângulos com cantos arredondados), *e.g.* `TensaoCorrecta` e `TensaoExcessiva`;
- **acções de entrada**, **acções de saída** e **actividades** contidas em estados, *e.g.* `DesligaSom()` e `EmiteSom()`;
- **transições** (setas), *e.g.* `CargaCorrecta` e `TensaoAlterada`;
- **recepções** de eventos, **condições** como `Tensao=Correcta` e `ReportaCC=TRUE`, **acções** e **envios de eventos** nas transições como `ConCargaCorrecta` e `ReportaAlarme`.

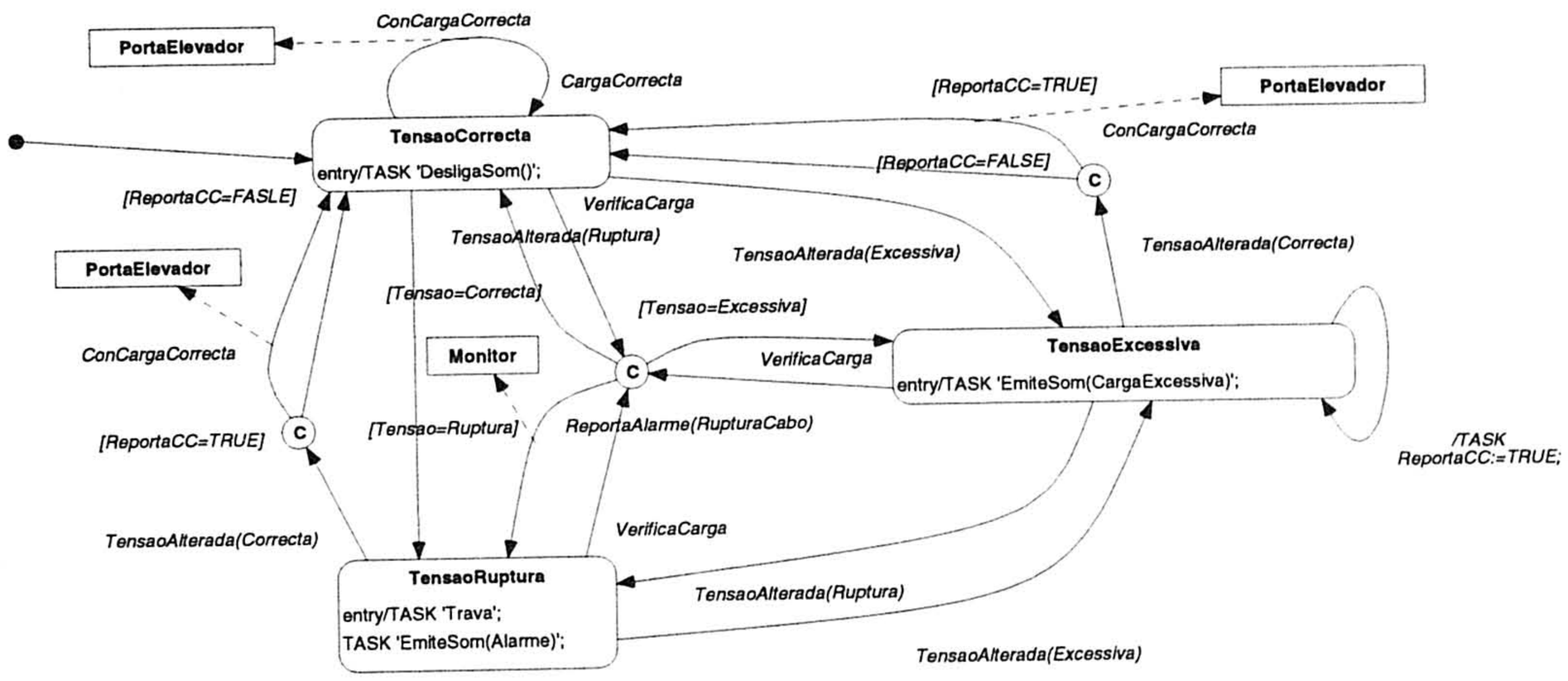


Figura 3.4: Exemplo de um Diagrama Estado-Transição.

3.3 Modelação Funcional

Um modelo funcional é composto por **diagramas de fluxo de dados** que correspondem à notação de **análise estruturada**.

A Figura 3.5 mostra o primeiro nível de uma diagrama de fluxo de dados que descrevem as funções principais esperadas de um SES. Estes diagramas são compostos por:

- **processos** (elipses), *e.g.* `PainelChamada` e `Elevador`;
- **fluxos de dados** (setas), *e.g.* `BotaoChamadaPremido` e `Chamada`;
- **controlos de fluxo** (setas a tracejado), *e.g.* `LuzAcesa` e `LuzApagada`;
- **actores** (rectângulos), *e.g.* `Utilizador`;
- **armazenadores de dados** (linhas duplas).

O modelo funcional deve ser um complemento ao modelo de objectos e ao modelo dinâmico. Os diagramas de fluxo de dados podem ser utilizados em diferentes níveis:

- visão geral dos serviços que o sistema deve fornecer, caso semelhante ao da análise estruturada tradicional;
- detalhar transições complexas dos diagramas de estados definidos no modelo dinâmico.

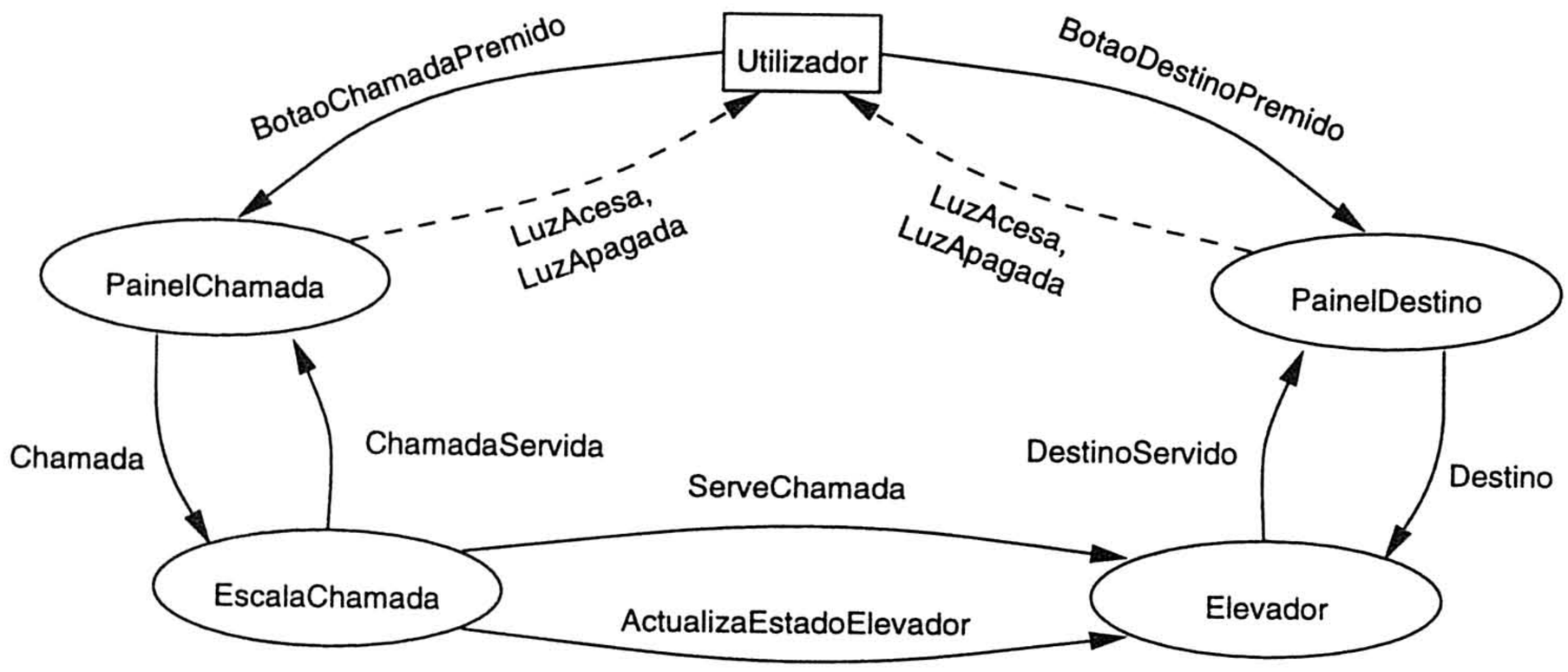


Figura 3.5: Exemplo de um Modelo Funcional.

3.4 Relação Entre os Modelos de Objectos, Dinâmico e Funcional

Na análise de sistemas, o modelo de objectos é o núcleo do método OMT, os outros dois modelos referem-se a ele.

O modelo dinâmico refina o modelo objecto especificando quando é que:

- os objectos são criados;
- as operações das classes são invocadas;
- os atributos são acedidos; e
- as associações são criadas, utilizadas e apagadas.

O comportamento das classes, relevante do ponto de vista dinâmico do sistema, é descrito através de diagramas estado-transição. Os atributos, operações e associações identificadas no modelo objecto são utilizadas nesses diagramas para especificar eventos, acções, actividades, *etc.*

O modelo funcional propõe um aspecto complementar da análise de sistemas. Focando-se nos serviços esperados, este modelo fornece ao analista uma visão das funções do sistema agrupando operações detalhadas das classes, ou detalha os aspectos dinâmicos refinando operações complexas de classes em processos mais pormenorizados. Geralmente, os diagramas de fluxo de dados utilizam as classes como actores ou armazenadores de dados, atributos como fluxos de dados e operações como processos. As sequências de controlo, embora não sejam recomendadas para evitar redundância com o modelo dinâmico, podem corresponder tanto a operações (eventos) como atributos (valores transmitidos pelos eventos).

3.5 Utilização dos Diagramas OMT ao longo do PES

A análise de requisitos (denominada **análise do sistema** em OMT) explora todos os diagramas: de objectos, dinâmico e funcional. Os esforços empenhados em cada um deles depende da natureza do sistema a ser desenvolvido: sistemas de informação requerem que seja dada ênfase à modelação por objectos, enquanto sistemas altamente interactivos são mais orientadas para a modelação dinâmica; sistemas de grande dimensão compostos por diferentes aplicações podem ser inicialmente detalhados por meio de um modelo funcional.

A OMT não providencia diagramas adequados para o projecto da arquitectura (chamado **projecto do sistema** na abordagem OMT), apesar desse assunto ser discutido em [Rum91].

A fase de projecto detalhado (denominado **projecto dos objectos** em OMT) utiliza somente os diagramas de classes. O modelo de objectos da análise deve primeiro ser enriquecido pela exploração dos outros modelos de análise, e depois adaptado e detalhado de acordo com as considerações do projecto da arquitectura.

3.6 OMT no Desenvolvimento de Sistema TR

As notações propostas pela OMT relativamente à modelação dinâmica (*i.e.* traçados de eventos e um subconjunto da notação dos diagramas de estado de Harel) são semiformais (apesar da notação completa de Harel ser formal). A monografia referência da OMT [Rum91] descreve como modelar a dinâmica do sistema através de diagramas gráficos, mas não fornece uma interpretação clara desses items gráficos. Não são definidos mecanismos de instanciação das classes para a transmissão de eventos (fila-de-espera, sincronização, *etc*) ou para múltiplas sequências de execução (*multi-threading*). Mais ainda, as acções e actividades são descritas através de texto plano e, por isso, não podem ser analisadas automaticamente.

A descrição através de diagramas de estado em complemento a traçados de eventos parece redundante, pois o propósito dos diagramas de estado é precisamente implementar os respectivos traçados de eventos. Porém, os diagramas de estado não podem ser utilizados mais tarde no SEP, já que são construídos *a priori* sem ter em conta restrições de projecto. Para o analista, o maior benefício extraído dos diagramas de estado é a capacidade de conseguir uma especificação exaustiva, mas isso obtém-se durante a fase de projecto. Nessa fase, a descrição da dinâmica do sistema baseia-se em considerações de projecto, pelo que os diagramas de estado podem ser explorados mais vantajosamente para a implementação da aplicação.

Por estas razões, em vez dos traçados de eventos OMT, é mais adequado empregar a notação MSC para fazer a modelação dinâmica, sendo as máquinas de estado descritas mais tarde na fase de projecto.

Quanto ao modelo funcional OMT, pode dizer-se que este não constitui nenhum meio específico para o desenvolvimento de *software* TR, porque é menos preciso que o modelo dinâmico para especificar aspectos dinâmicos (a cronologia dos processos é implícita), e porque o modelo de objectos oferece uma forma melhor para descrever o sistema no seu ambiente se a reutilização for considerada um objectivo principal (os objectos são melhores candidatos à reutilização do que processos hierárquicos).

A modelação por objectos é de facto o valor acrescentado da utilização de OMT no desenvolvimento de *sofwtare* TR. A notação é simples, concisa e contém todos os conceitos OO necessários (encapsulamento e herança) numa abordagem que suporta melhor a reutilização. No contexto dos sistemas TR, esta notação é particularmente adequada para especificar aspectos dos objectos durante a fase de análise. Os diagramas de classes também são bastante úteis para os projectistas, pois além de serem uma notação formal dedicada ao projecto de objectos concorrentes, podem ser utilizados para a descrição de objectos passivos. Em alguns sistemas, os objectos passivos podem representar uma parte significativa do sistema a implementar.

3.7 UML versus OMT

Nos últimos anos vários especialistas em metodologias de especificação têm comparado, debatido e finalmente cooperado na formulação de uma linguagem de modelação genérica. Foi assim que surgiu a *Unified Modeling Language* (UML) contando com a participação de autores de alguns dos mais importantes métodos de especificação como Grady Booch, Ivar Jacobson, James Rumbaugh e David Harel. O objectivo foi tentar reunir o melhor das principais notações e métodos OO, mais concretamente: Booch [Boo91], *Object-Oriented Software Engineering* (OOSE) [Jac92] e OMT [Rum91]. A última versão da UML (UML 1.1) surgiu em Setembro de 1997 e, segundo Douglass [Dou98], recentemente o *Object Management Group* (OMG) adoptou-a como uma norma.

A UML vem preencher uma lacuna que há muito os analistas e projectistas anseiam por ver preenchida: uma norma que una a comunidade industrial OO e encorage novos utilizadores a aprenderem modelação visual, acabando com a divisão entre diferentes notações para expressar conceitos semelhantes e com o custo de passar de umas para outras consoante as ferramentas utilizadas.

Como parte da UML assenta na OMT, são naturais as semelhanças existentes entre as duas. Essas semelhanças verificam-se sobretudo ao nível dos diagramas de classes e diagramas de estado-transição, precisamente as partes de OMT utilizadas no método OORT. Por isso, a utilização de UML ou OMT no OORT não levanta problemas: a transição para UML será feita sem problemas, pois a tradução dos referidos diagramas de OMT para UML é simples.

A próxima versão da ferramenta *Object*GEODE suportará a UML e terá um tradutor de OMT para UML.

4 Breve Descrição de MSC

A notação MSC está definida numa recomendação ITU-T [MSC96] e foi inicialmente especificada para complementar a notação SDL nas fases de análise do sistema e projecto de testes. A notação MSC é utilizada tanto por analistas, como por quem desenvolve os sistemas. Os analistas elaboram MSCs para especificar cenários de troca de mensagens (o termo **mensagem** corresponde ao termo **evento** do OMT) entre o sistema e o ambiente em que este se insere. Os implementadores do sistema utilizam a notação MSC para construir planos de teste, refinado os cenários de alto nível anteriores de acordo com a arquitectura do sistema. Os cenários e os planos de teste descritos por MSCs podem ser organizados na forma de uma **hierarquia de cenários**.

4.1 Hierarquia de Cenários

Um sistema tem uma hierarquia de cenários, que engloba todos os diferentes cenários esperados do sistema quando este interage com o respectivo ambiente. Os cenários são identificados através do seguinte processo iterativo: primeiro identificam-se os cenários principais; depois refina-se cada um independentemente; o processo termina quando todos os cenários terminais podem ser descritos através de sequências de mensagens entre instâncias.

A Figura 4.1 apresenta uma hierarquia de cenários parcial para um elevador. A hierarquia é constituída por:

- **cenários** (caixas), como `Operacao` e `RecebePedidos`.
- **operadores de composição** (pequenos símbolos ligados à parte inferior dos cenários);

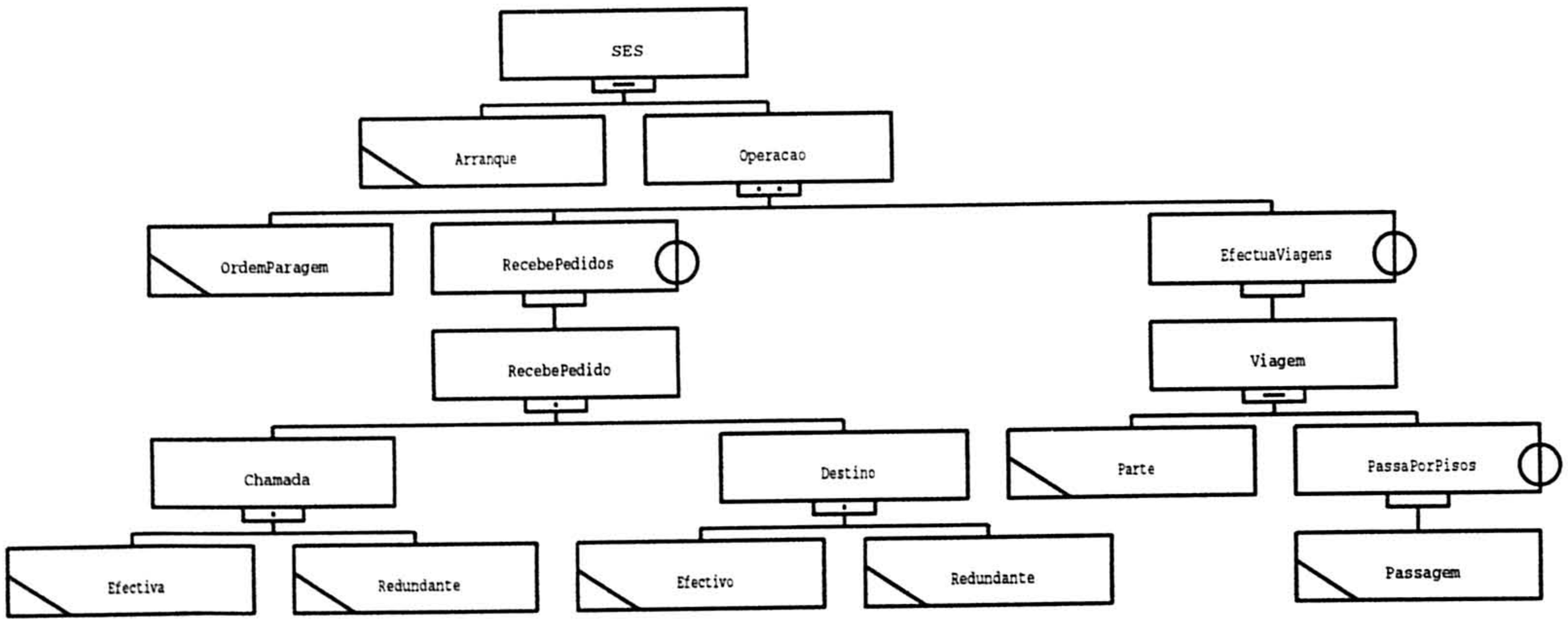


Figura 4.1: Exemplo de uma Hierarquia de Cenários.

Uma das maiores dificuldades na modelação dinâmica com MSCs é a explosão de cenários. Uma maneira de lidar com este problema é utilizar operadores de composição para definir como é que os diferentes cenários são combinados para formar um cenário de nível superior mais abstracto[2]. Os operadores de composição de cenários MSC geralmente aceites são:

- **e**, os cenários são executados em sequência;
- **ou**, um dos cenários é executado (escolha não-determinística);
- **paralelo**, os cenários são executados em paralelo, sem sincronização;
- **repete**, o cenário é executado 0 ou mais vezes;
- **excepção**, cenário que interrompe a execução doutro;
- **é**, que expressa a ausência de composição.

Todavia, os operadores de composição não resolvem o problema, apenas servem para o minorar.

4.2 MSC

Os cenários terminais que compõem a hierarquia de cenários são detalhados por meio de MSCs. Um MSC descreve um cenário como uma sequência cronológica de interacções entre objectos. Há duas categorias de MSCs:

- **MSCs abstractos**, que descrevem as trocas de mensagens que ocorre entre os objectos; e
- **MSCs detalhados**, que adicionalmente fornecem uma descrição das acções levadas a cabo pelos objectos.

A Figura 4.2 mostra um MSC abstracto. Os MSCs abstractos são equivalentes aos traçados de eventos OMT sendo compostos por:

- **instâncias** (barras verticais com nome, cuja classe é mencionada em caixas no topo), como `Jose` cuja classe é `Utilizador`;
- **mensagens** (setas) que transportam informação, como `BotaoChamadaPremido` e `ComandaLuz`.

Um MSC detalhado é um MSC abstracto enriquecido com mais símbolos gráficos que detalham o comportamento interno dos objectos como, por exemplo, estados, procedimentos, temporizadores, *etc.* A Figura 4.3 mostra parcialmente um MSC detalhado que descreve as interacções entre objectos internos do SES. Além dos elementos que compõem os MSCs abstractos, os MSCs detalhados incluem:

- **criações de objectos** (setas a tracejado), *e.g.* `Porta_2`;

---

[2] Esta solução constitui uma extensão à norma [MSC96] que define o MSC.

- **acções** (caixas), *e.g.* `MeuNE:=LeNumPiso();`
- **condições** (caixas com lados quebrados), *e.g.* `MeuNE<NE`;
- **activação**, **desactivação** e **expiração de temporizadores** (setas em forma de L dispostas verticalmente) , *e.g.* `Temporizador`;
- **procedimentos** (caixas alongadas na vertical), como `ProcExe`;
- **regiões de concorrência** (barras verticais a tracejado), *e.g.* a que cobre as mensagens `PidPE`;
- **paragem de processos** (cruzes terminais das barras verticais).

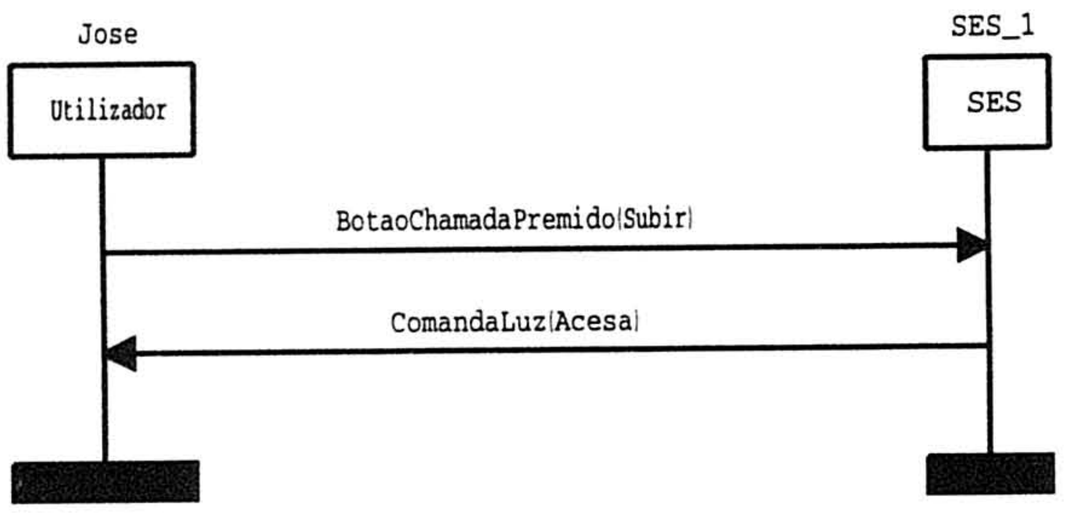


Figura 4.2: Exemplo de um MSC Abstracto.

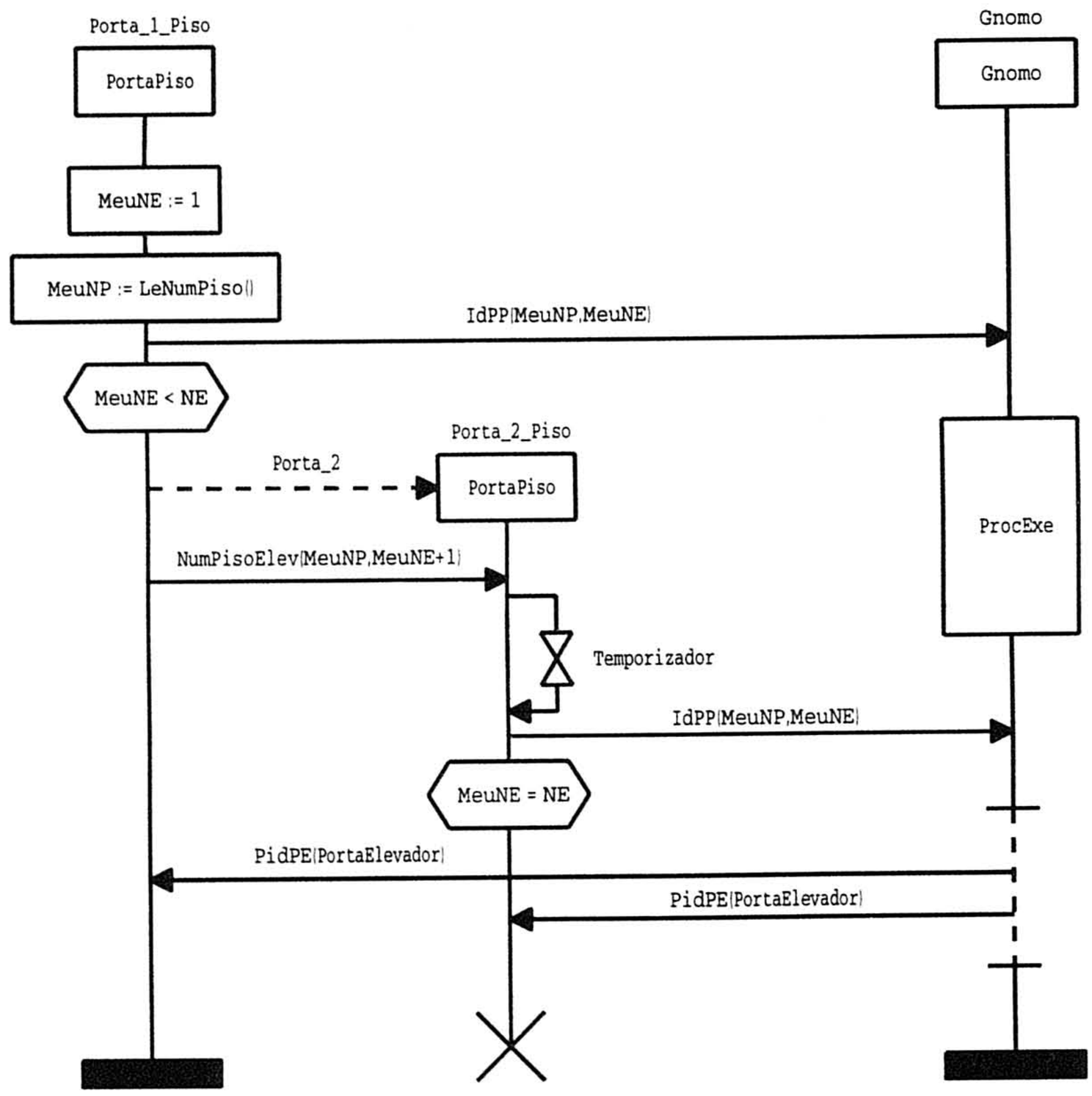


Figura 4.3: Parte de um MSC Detalhado.

4.3 Utilização dos Diagramas MSC no decurso do PES

Na fase de análise, a notação MSC serve para modelar casos de utilização. O comportamento do sistema é descrito através de MSCs abstractos definidos ao nível do sistema (*i.e.*, o sistema aparece como uma das instâncias e as outras são actores do ambiente), que são organizados numa hierarquia de cenários.

Na fase de projecto, o refinamento dos MSCs ajuda a elaboração da arquitectura do sistema através da atribuição clara de funções a subsistemas e a objectos mais pormenorizados. Para além disso, os MSCs são também utilizados para comparar o comportamento da aplicação com os requisitos do utilizador.

Na fase de projecto de testes, os MSCs permitem a produção de planos de teste (MSCs detalhados) que irão servir para verificar independentemente (testes unitários) os objectos que compõem o sistema, e verificar gradualmente a sua integração (teste de integração) no sistema. Estes MSCs detalhados são depois concretamente aplicados à aplicação desenvolvida durante a fase de verificação.

4.4 MSCs no Desenvolvimento de Sistemas TR

A notação MSC é particularmente útil no desenvolvimento de sistemas de tempo-real, ou não tivesse sido definida por especialistas de telecomunicações para lidar com os aspectos específicos do desenvolvimento de sistemas TR.

No método OORT, em vez dos traçados de eventos OMT, utilizam-se as hierarquias de cenários e os MSCs nas fases de análise de requisitos e projecto de testes. Utilizar uma notação única para descrever casos de utilização e testes reforça a habilidade para fazer o rastreio (*traceability*) do sistema através de todo o PES.

5 Sumário de SDL

A linguagem SDL está definida numa recomendação ITU-T e foi criada para o desenvolvimento de sistemas TR. A partir da penúltima versão [SDL93], passou a incluir os conceitos de herança e generalidade (*genericity*). A SDL foi formalmente definida para facilitar a validação antecipada e a prototipagem rápida, e também para automatizar a implementação de sistemas. Adequa-se particularmente à especificação de sistemas reactivos/assíncronos (*event-driven*) discretos[3], já que os objectos SDL comunicam por meio de mensagens assíncronas através de canais de transmissão ponto-a-ponto.

A SDL é composta de quatro tipos de diagramas que cobrem as fases de projecto da arquitectura e projecto detalhado.

5.1 Diagrama de Hierarquia

O **diagrama de hierarquia** é o primeiro a ser construído quando se especifica um sistema em SDL. Ele serve para descrever o sistema como uma hierarquia de objectos, cada um com uma *interface* clara e bem visível para comunicar com as entidades exteriores. Um **objecto pai** esconde do exterior **os objectos filhos** que contém através de encapsulamento hierárquico.

A hierarquia SDL é composta pelos seguintes elementos:

- **tipo de sistemas**,
- **sistema**,
- **tipo de bloco**,
- **bloco**,
- **substrutura**,
- **tipo de processo**,
- **processo**,
- **tipo de serviço**,
- **serviço**,
- **procedimento**,

---

[3] Ver [Olsen94] para uma discussão mais pormenorizada sobre o tipo de sistemas aos quais a linguagem SDL pode ser aplicada com sucesso.

- **macro**,
- **pacote**,
- **operador**, e
- **declaração PR** (*Phrasal Representaion*)[4].

A Figura 5.1 exemplifica uma hierarquia SDL parcial de um SES: `SES` representa um sistema, `Elevador` é um tipo de bloco, `Elevadores` é um bloco do tipo `Elevador`, `Central` é um bloco, `Elevador`, `Porta`, `Supervisor` e `Gnomo` são processos, `EnviaChamada` é um procedimento.

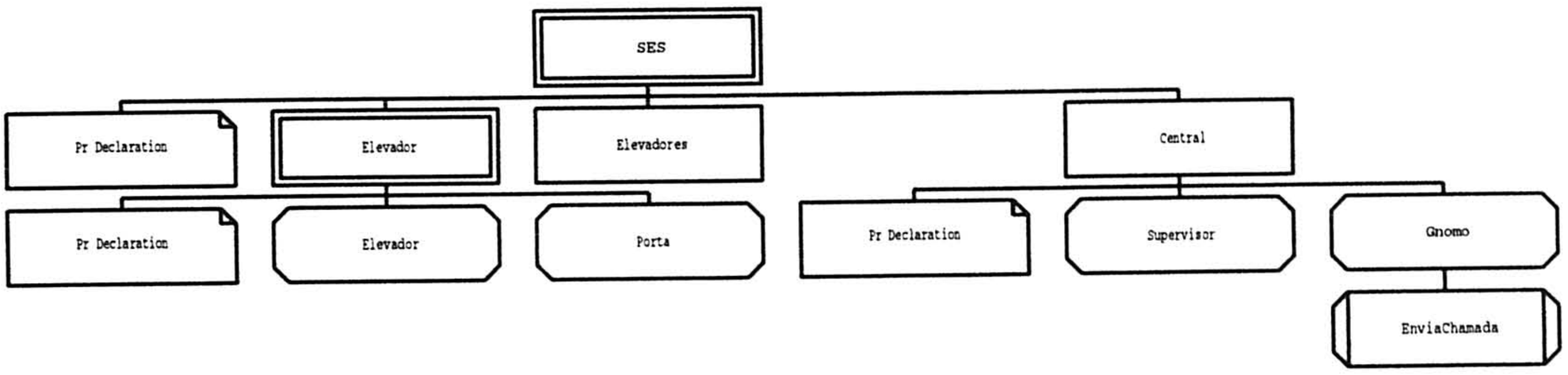


Figura 5.1: Exemplo de uma Hierarquia SDL.

Qualquer objecto estrutural (sistema, bloco, processo, *etc.*) pode conter declarações de **constantes**, **tipos de dados**, **sinais**, *etc.*, definidos em declarações PR (descritas mais adiante).

Os tipos de objectos (tipo de sistema, tipo de bloco, tipo de processo e tipo de serviço) com os seus parâmetros genéricos, permitem ao projectista construir objectos estruturais que são reutilizáveis em múltiplas implementações sem alterações.

5.2 Diagrama de Interligação

Os **diagramas de interligação** completam os diagramas de hierarquia SDL, indicando os fluxos de comunicação entre objectos localizados ao mesmo nível hierárquico e entre esses objectos e o respectivo pai (objecto exterior que os engloba).

A Figura 5.2 mostra um diagrama de interligação de primeiro-nível de um SES. Para além dos objectos disponíveis no diagrama de hierarquia, o diagrama de interligação inclui:

- **ligações** (**canais** ou **rotas**), que simbolizam caminhos de comunicação entre objectos estruturais, como `Chamadas` e `SensorPiso`;

---

[4] A linguagem SDL tem uma representação gráfica chamada SDL-GR (*Graphical Representation*), e uma representação textual chamada SDL-PR, e permite a tradução de uma forma para a outra. Para a troca de especificações SDL geralmente é necessário o formato SDL-PR ou qualquer outra forma normalizada. O *Object*GEODE guarda o ficheiros SDL no formato PR com extensões CIF (*Common Interchange Format*) que são comentários (/* CIF ... */) que incluem informação gráfica (para mais informação ver [CIF96]).

- **sinais**, que transmitem valores e que estão associados às ligações, como `ServeChamada` e `ChamadaServida`; os seus parâmetros formais são definidos em declarações PR;
- **pontos extremos de conexão**, que servem para conectar ligações de filhos e pais (na Figura 5.2 não existem pois ela refere-se ao topo da hierarquia SDL, caso contrário o canais estariam conectados ao rectângulo envolvente do diagrama através de pontos extremos de conexão).
- **portas**, utilizadas em conjunto com tipos de objectos estruturais e que definem *interfaces* de comunicação genéricas, como `GE` e `GSP`.
- **criações de processos**, no nível mais baixo da hierarquia, que expressam a criação de processos entre objectos concorrentes terminais.

Os sinais SDL correspondem aos eventos OMT e às mensagens MSC.

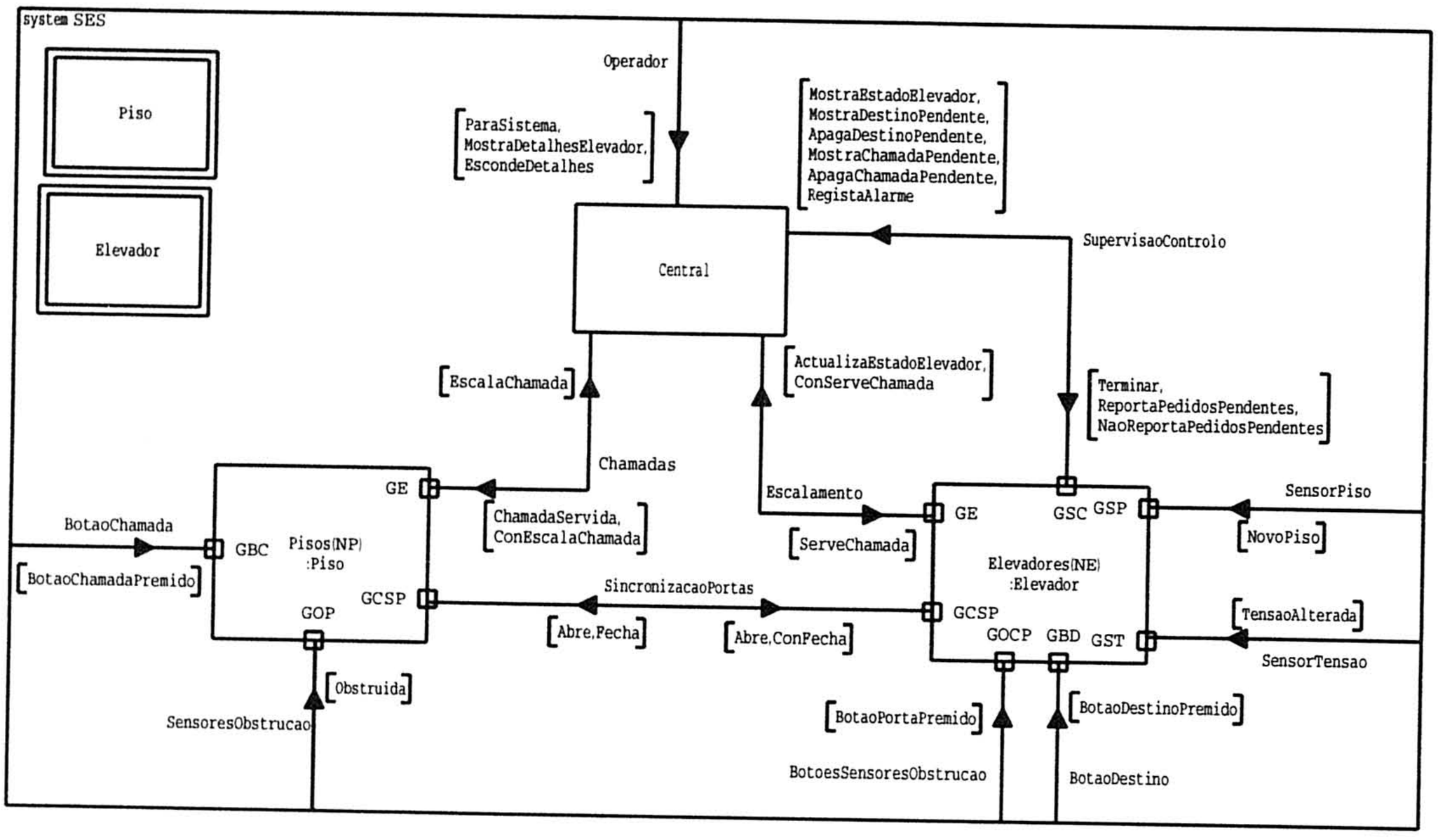


Figura 5.2: Diagrama de Interligação SDL de um SES.

5.3 Diagrama de Processo

Os **diagramas de processo** descrevem objectos terminais de um diagrama de hierarquia e são semelhantes aos tradicionais diagramas estado-transição. Um objecto do último nível da hierarquia SDL é denominado **processo** e corresponde a um objecto concorrente. Um processo tem uma única **fila-de-espera** que guarda os sinais recebidos. De cada vez que um sinal é recebido, o processo fica activo, consome o sinal e executa a transição disparada por esse sinal.

Comparativamente a outras notações de projecto (OOSE, Booch, *etc.*), a SDL tem a grande vantagem de propor um conjunto completo de símbolos gráficos para especificar **formalmente** todas as acções realizadas quando uma transição é disparada.

A Figura 5.3 ilustra parte de um processo que controla a porta do elevador de um SES. A linguagem SDL oferece um conjunto de aproximadamente trinta símbolos gráficos para pormenorizar estados e transições, dos quais os mais utilizados são:

- **estados**, *e.g.* `Fechada` e `AguardaTempoFechar`;
- **entradas**, que correspondem à recepção de sinais, *e.g.* `BotaoPortaPremido` e `ResVerificaCarga`;
- **saídas**, que correspondem à produção de sinais, *e.g.* `CargaCorrecta` e `ConFecha`;
- **tarefas**, que permitem a manipulação de dados através de operadores, *e.g.* `'Abre'`;
- **decisões**, que estruturam as transições de acordo com valores dos dados, *e.g.* `Carga`;
- **activação de temporizadores**, *e.g.* `SET(NOW+IntervaloFecho, TentaFechar)`, **desactivação de temporizadores,** *e.g.* `RESET(TentaFechar)`, e **expiração de temporizadores**, *e.g.* `TentaFechar`, para gerir o tempo;
- **término de processos** (uma cruz terminal como em MSCs) para terminar a actual instância do processo;
- **estado início** (estado sem nome), ao qual se segue a transição inicial (transição sem entrada) que é executada uma vez quando o processo é instanciado (criação do objecto).

Existem ainda os símbolos para **definir** e **invocar procedimentos**, **instanciar processos** (se estes pertencerem ao mesmo objecto pai), **definir** e **unir pontos de execução** (*labels*), **guardar sinais**, **definir** e **chamar macros**, *etc.*

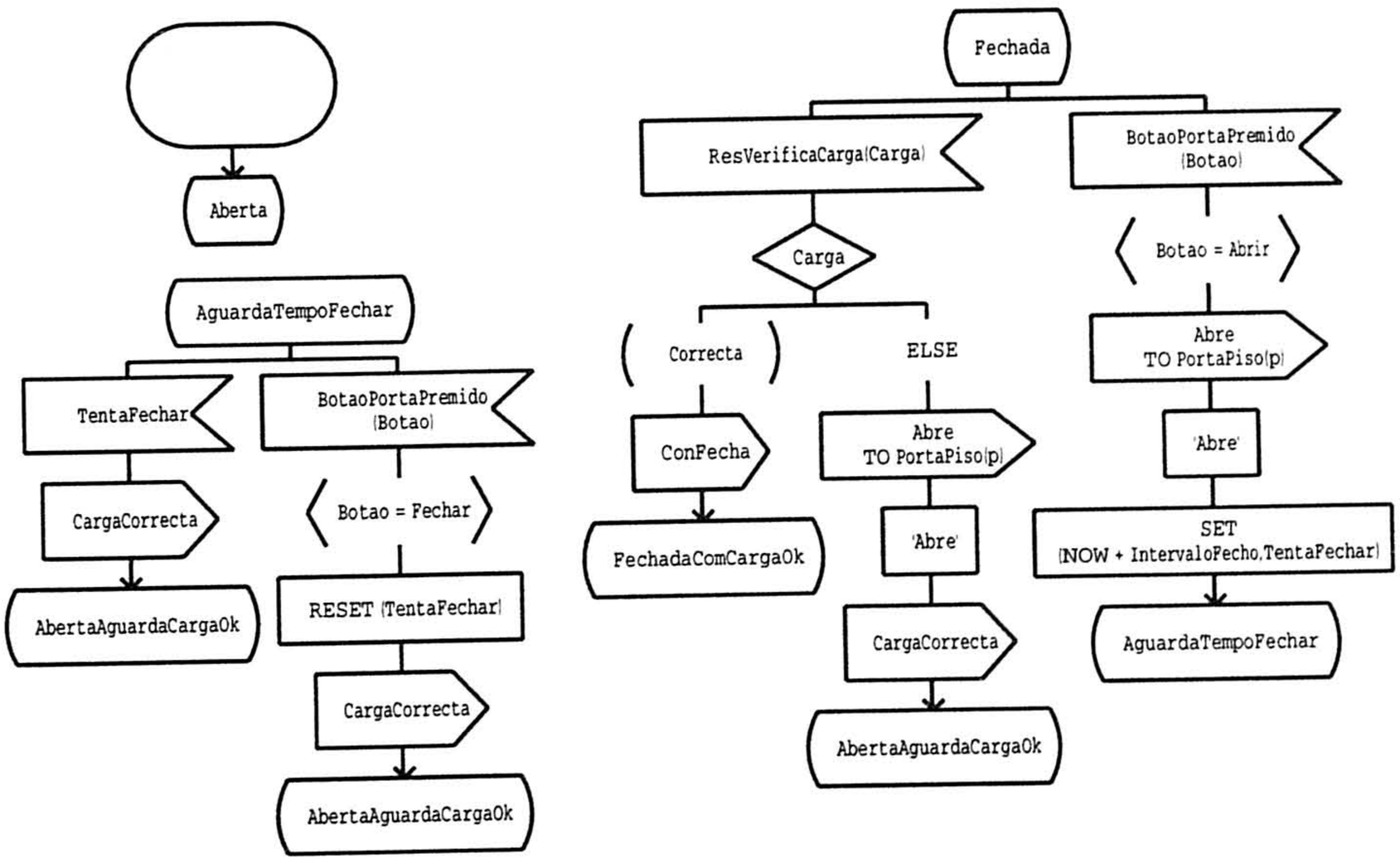


Figura 5.3: Parte de um Diagrama de Processo SDL.

5.4 Declarações PR

Cada objecto contém o seu próprio conjunto de declarações PR, que consistem em definições textuais como **constantes**, **tipos**, **sinais**, **temporizadores**, *etc.* As variáveis podem ser declaradas localmente em cada objecto terminal, enquanto as variáveis globais são fortemente restringidas, já que não têm significado se forem partilhadas por objectos que são executados em plataformas diferentes e/ou remotamente.

Para facilitar a reutilização e, também, para tornar possível a integração de objectos externos, a linguagem SDL oferece o conceito de ***Abstract Data Type*** (ADT). Um ADT, tal como uma classe, é composto por dados internos (semelhantes aos atributos das classes OMT) e operadores (correspondentes às operações das classes OMT). Os operadores ADT são invocados por processos para realizar acções sequenciais durante as transições. Os operadores podem ser especificados de várias formas:

- Informalmente, especificando texto informal; esta opção é adequada para as primeiras fases do processo de desenvolvimento; ver Figura 5.4;
- Axiomaticamente, especificando equações que expressam qual o comportamento dos operadores; *i.e.* definem o valor retornado de acordo com os valores dos parâmetros de entrada; as equações também definem implicitamente um conjunto de valores do tipo de dados; Figura 5.5;
- Algoritmicamente, especificando o algoritmo dos operadores em SDL-PR (Figura 5.6), ou SDL-GR (Figura 5.7), de modo análogo a um procedimento, *i.e.* descrevendo como é que os operadores calculam o resultado;
- Externamente, especificando um comportamento externo através de uma construção especial para expressar outro conceito de dados não SDL; é útil nas situações em que a especificação será implementada numa linguagem de programação específica; ver Figura 5.8;

```
NEWTYPE Matematica
 LITERALS Nenhum;
 OPERATORS
   fact: INTEGER -> INTEGER;
 AXIOMS
   'fact retorna o factorial';
ENDNEWTYPE;
```

Figura 5.4: Descrição Informal de um Operador ADT.

```
NEWTYPE Matematica
LITERALS Nenhum;
OPERATORS
  fact: INTEGER -> INTEGER;
AXIOMS
  FOR ALL x IN INTEGER (
    fact (x) ==
      IF x < 0 THEN ERROR!
      ELSE IF x = 0 THEN 1
        ELSE x * fact (x-1) FI
      FI; );
ENDNEWTYPE;
```

Figura 5.5: Descrição Axiomática de um Operador ADT.

```
NEWTYPE Matematica
LITERALS Nenhum;
OPERATORS
  fact: INTEGER -> INTEGER;
OPERATOR fact;
  FPAR x INTEGER;
  RETURNS f INTEGER;
    START;
    IF x < 0 THEN TASK f := 0;
    ELSE
      IF x = 0 THEN TASK f := 1;
      ELSE TASK f := x * fact (x-1);
      FI;
    FI;
    RETURN f;
 ENDOPERATOR;
ENDNEWTYPE;
```

Figura 5.6: Descrição Algorítmica Textual de um Operador ADT.

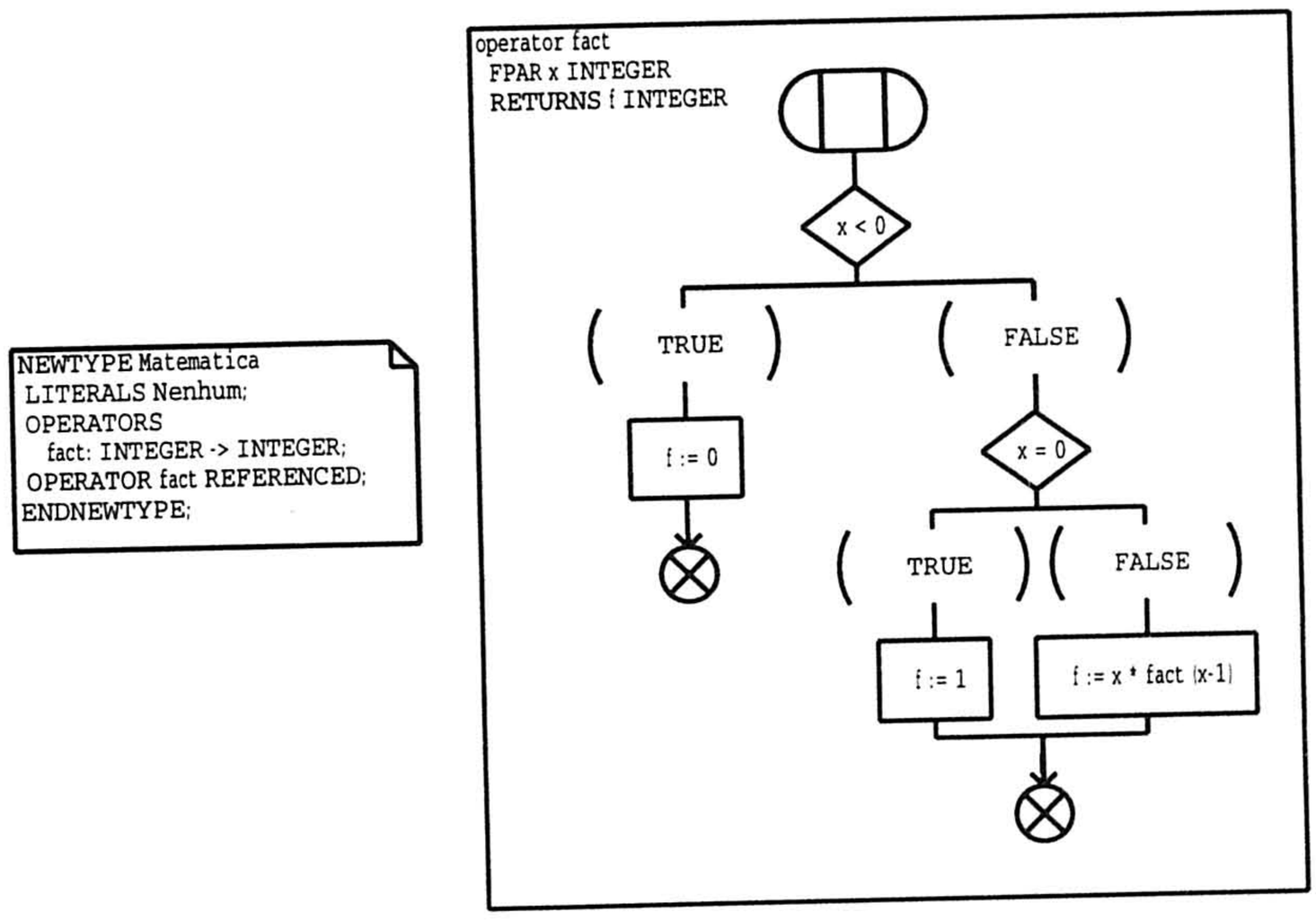


Figura 5.7: Descrição Algorítmica Gráfica de um Operador ADT.

```
NEWTYPE Matematica
LITERALS Nenhum;
OPERATORS
  fact: INTEGER -> INTEGER;
ALTERNATIVE C;
  #include 'mat.c'
ENDALTERNATIVE
ENDNEWTYPE;
```

Figura 5.8: Descrição Externa de um Operador ADT.

As formas axiomática e algorítmica são alternativas nas maior parte dos casos. A abordagem algorítmica é mais simples de utilizar e mais fácil de lidar para as ferramentas do que a axiomática. No entanto, a

abordagem algorítmica não é tão poderosa (*i.e.*, não pode definir os valores para o tipo de dados), e do ponto de vista da especificação, a abordagem axiomática é mais elegante já que, em oposição à forma algorítmica, define qual o comportamento em vez de como o comportamento é obtido.

As actuais evoluções da recomendação SDL visam a utilização da norma *Abstract Syntax Notation One* (ASN.1) para a especificação dos dados (ver [ASN95]).

5.5 Utilização dos Diagramas SDL ao longo do PES

A linguagem SDL é utilizada na actividade de projecto da arquitectura, para definir a arquitectura do sistema (diagramas hierárquico e de interligação), e mais tarde durante a fase de projecto detalhado, para descrever objectos concorrentes através de diagramas de estado-transição (diagramas de processo). Esta linguagem não é a melhor para descrever objectos passivos, logo, não cobre toda a fase de projecto detalhado. Os axiomas e algoritmos que descrevem ADTs são bem mais adequados para integrar objectos passivos externos do que para os implementar.

Geralmente, utiliza-se SDL em conjunto com MSC [SDLS97]. Quando se inicia a fase de projecto, recorre-se aos MSCs abstractos definidos ao nível do sistema e para construir arquitectura do sistema. Simultaneamente, a refinação de MSCs é levada a cabo de modo a assegurar uma coerência óptima entre o comportamento esperado, expresso através de MSCs, e a implementação de facto, expressa em objectos hierárquicos SDL.

5.6 SDL no Desenvolvimento de Sistemas TR

Tal como o MSC, a SDL é particularmente apropriada ao desenvolvimento de sistemas TR, pois também foi definido por especialistas da área das telecomunicações, para especificar e desenvolver arquitecturas desses sistemas. Todavia, a SDL centra-se somente na arquitectura do sistema e na sua dinâmica; os ADTs não são gráficos e estão dispersos pelos diferentes objectos SDL. Consequentemente, a SDL, por si só, não é totalmente adequada para especificar dados complexos que podem requerer modelos sofisticados com tipos de dados fortemente correlacionados.

Por conseguinte, complementam-se os diagramas SDL com a técnica dedicada de modelação por objectos utilizada na OMT, sendo as declarações de dados utilizadas para fazer a *interface* apropriada com os objectos passivos.

5.7 UML versus MSC/SDL

Tal como foi referido em 3.7, a UML foi elaborada com base na experiência passada de outras técnicas de modelação de sistemas. Actualmente discutem-se os méritos do método UML comparativamente aos métodos baseados em SDL/MSC.

Segundo Harel em [Dou98], com mais algum trabalho a UML tornar-se-á provavelmente não só uma norma aprovada oficialmente, mas também na principal metodologia OO de especificação de *software*.

Mas até à data, a linguagem SDL tem as vantagens de ser uma norma internacional e estar totalmente formalizada permitindo a simulação e geração completa de código. Apesar de poder ser considerada como uma norma, a UML não está completamente definida. Tal como a respectiva documentação [Rat97] diz, somente a semântica estática da linguagem está definida, faltando definir a semântica dinâmica. Além disso, a SDL tem, para já, um poder de expressão superior à UML.

Neste momento, a posição dos fabricantes de ferramentas CASE aponta para a utilização de UML em vez de OMT para a análise e a utilização de SDL para o desenho.

6 Combinação de OMT, MSC e SDL no OORT

Tal como foi previamente demonstrado, utilizar uma única notação para todas as actividades do desenvolvimento de *software* TR não é actualmente apropriado, pois ainda não há nenhuma que as suporte de forma adequada. Em vez de inventar uma nova e complexa notação, é vantajoso utilizar uma combinação consistente de notações maduras, cada uma dedicada a problemas específicos. As notações OMT, MSC e SDL encaixam perfeitamente neste propósito.

6.1 Posição Genérica das Três Notações

Para um dado estágio de desenvolvimento da aplicação, escolher entre OMT, MSC e SDL, depende em primeiro lugar da fase do PES e depois da natureza das partes do *software* a serem estudadas.

Na fase de análise de requisitos, combinam-se a modelação por objectos OMT e a notação MSC. Empregam-se os diagramas de classes e de instâncias para descrever os requisitos de dados do utilizador (tais como dados produzidos pelo ambiente e consumidos pelo sistema, ou produzidos pelo sistema e enviados para o ambiente) e, de forma mais genérica, todos os objectos relevantes ao espaço do domínio da aplicação (entidades lógicas tipicamente identificadas no domínio do sistema e entidades físicas constituintes do ambiente e do sistema quando são geridas por este). A notação MSC é utilizada, primeiro, para identificar e organizar cenários (ou funções) esperados e, depois, para descrever os cenários terminais na forma de MSCs. Esta abordagem é semelhante à modelação de **Casos de Utilização** definida em [Jac92].

As necessidades do projecto da arquitectura são totalmente satisfeitas pela notação SDL. A descrição da arquitectura do sistema é constituída por um diagrama hierárquico SDL e o correspondente conjunto de diagramas de interligação. O projecto de testes, utilizando MSCs, realiza-se em simultâneo e ajuda a refinar a arquitectura. Estes MSCs arquitecturais vão mais tarde ser utilizados como testes de integração.

Para a fase de projecto detalhado recorre-se à utilização de SDL e OMT para descrever, respectivamente, objectos concorrentes terminais através de diagramas de processo e objectos passivos por meio de diagramas de classes. Os objectos concorrentes terminais são elementos das extremidades da hierarquia SDL. Os objectos passivos resultam tanto dos diagramas de classes da análise, como de ADTs declarados na arquitectura do sistema. Estes objectos passivos iniciais são refinados em objectos passivos mais detalhados de forma a simplificar a implementação das operações das classes. Geralmente, os objectos passivos correspondem a gestão de dados (*e.g.* através de sistemas de base-de-dados), algoritmos sequenciais (*e.g.* algoritmos matemáticos) ou ligação/encapsulamento de objectos (*e.g. software* de *interface* a elementos físicos). Os MSCs arquitecturais, criados durante o projecto do sistema, refinam-se

em paralelo com a actividade de projecto detalhado, para completarem os testes de integração e produzirem testes unitários.

A implementação (*targeting*) é significativamente automatizada pela utilização da notação formal SDL. Os processos SDL podem ser traduzidos em código fonte totalmente executável. Os objectos passivos descritos por diagramas de classes OMT podem ser traduzidos em esqueletos de código. A maior parte do restante trabalho de implementação consiste em codificar os objectos passivos.

A actividade de teste também é francamente facilitada pela utilização de MSCs para produzir planos de teste. Estes planos podem, por exemplo, ser traduzidos numa linguagem como TTCN e utilizados para analisar os resultados dos testes obtidos no ambiente da plataforma.

A Figura 6.1 apresenta a posição geral da OMT, MSC e SDL dentro do processo genérico do método OORT.

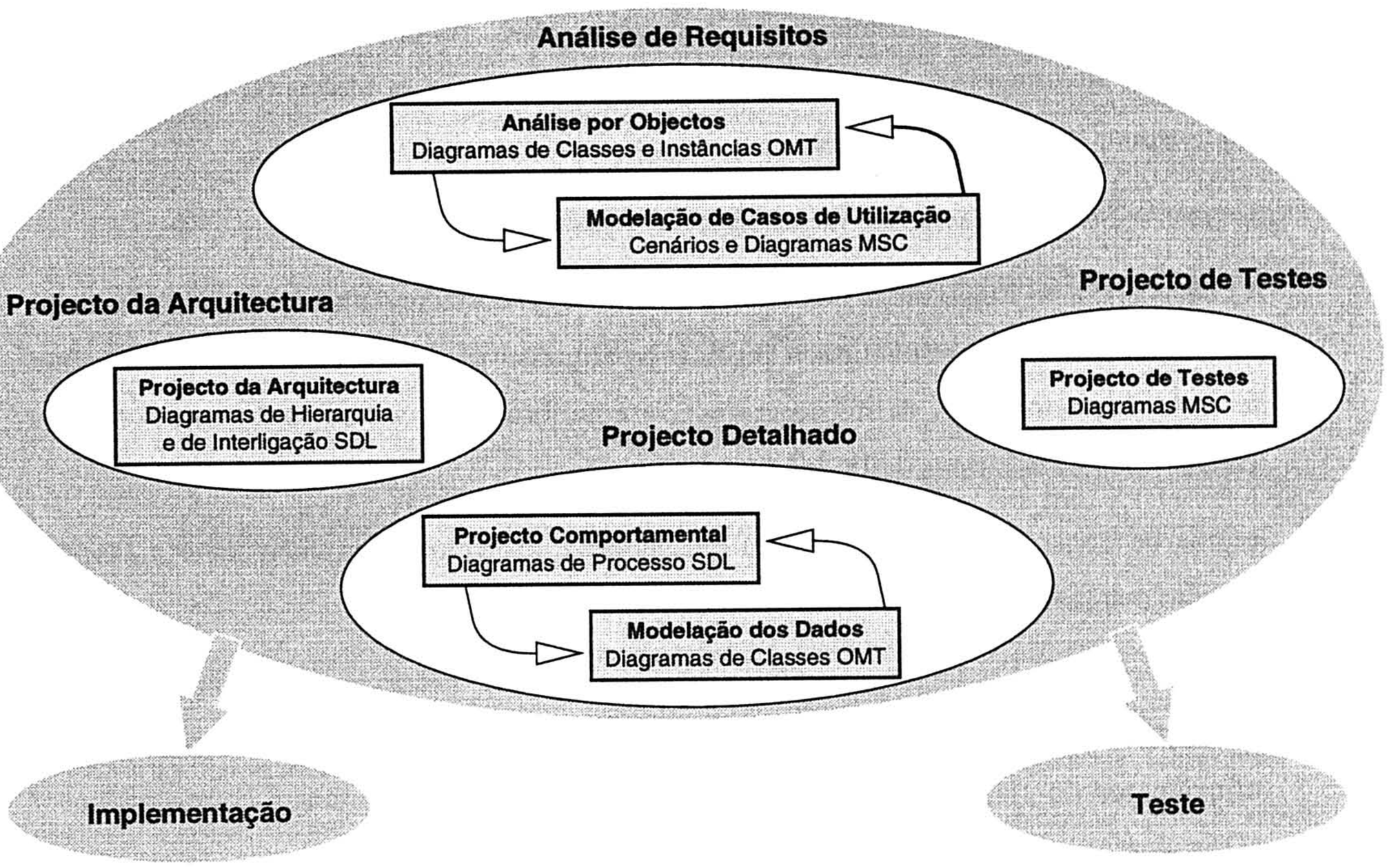


Figura 6.1: Processo de Engenharia Genérico do OORT.

6.2 Consistência Global

A consistência da metodologia OORT é conseguida pela adesão a um conjunto de regras que guiam a utilização combinada das diferentes notações em cada fase e as passagens de fase para fase:

- Regras de consistência na fase de análise de requisitos

A modelação por objectos OMT e a modelação dinâmica MSC ligam-se do seguinte modo: as instâncias de objectos MSC são instâncias de classes OMT, as mensagens MSC têm que aparecer como propriedades das classes OMT (operações, atributos ou associações); a transmissão de uma mensagem MSC tem que ser suportada por um conjunto de associações OMT que ligam os objectos emissor e receptor.

- Regras de consistência entre a análise de requisitos e o projecto da arquitectura

  Os objectos concorrentes são identificados de entre todos os objectos que constituem os diagramas de classes da análise e são transformados em elementos estruturais de SDL (blocos, processos, *etc.*). Algumas mensagens, identificadas em MSCs da análise, são reutilizadas em diagramas de interligação como sinais SDL. As restantes podem ser descritas por ADTs ou sub-hierarquias SDL (por exemplo, porque são implementados por protocolos complexos). As ligações entre objectos SDL têm que ser consistentes com as associações OMT e as transmissões de mensagens MSC. As operações das classes que se encontram nos diagramas de classes da análise podem tornar-se blocos ou processos para implementar concorrência interna.

- Regras de consistência na fase de projecto da arquitectura

  A consistência ao longo desta fase resulta da utilização correcta da notação SDL que inclui um conjunto de regras semânticas a verificar na arquitectura do sistema: utilização coerente de entidades estruturais, definição correcta e equilibrada da comunicação entre objectos (canais, sinais, *etc.*), tipos definidos e utilizados, *etc.*

- Regras de consistência entre as fases de análise e projecto da arquitectura e a fase de projecto detalhado

  No que concerne à linguagem SDL, os diagramas de processo são refinamentos da arquitectura do sistema. Relativamente aos diagramas de classes OMT, alguns objectos são refinamentos de classes passivas identificadas na análise e os restantes correspondem a ADTs identificados em declarações PR de SDL.

- Regras de consistência na fase de projecto detalhado

  Os processos SDL e as classes OMT mantêm-se coerentes através de ADTs: os ADTs SDL e as suas correspondentes classes OMT têm que ter a mesma *interface* pública.

- Regras de consistência entre as fases de análise e projecto e a fase de projecto de testes

  O projecto de testes realiza-se refinando os MSCs obtidos na modelação de casos de utilização, de acordo com a arquitectura SDL.

- Regras de consistência na fase de projecto de testes

Os MSCs arquitecturais iniciais produzidos durante o projecto da aplicação são enriquecidos por novos testes, de forma a ter-se uma abrangência satisfatória de cenários (critério de plenitude - *completeness*). Em particular, dá-se ênfase a condições críticas no sentido de detectar comportamento perigosos. Tanto as sequências esperadas, como as de erro, são identificadas.

- Regras de consistência entre o projecto e a implementação (*targeting*)

  A consistência da implementação depende da linguagem de programação escolhida para OMT e SDL, e também, da forma como as construções OMT e SDL são automaticamente traduzidas em código executável. Resta ainda a tarefa de integração de objectos concorrentes e objectos passivos.

- Regras de consistência entre as fases de projecto detalhado e projecto de testes e a fase de testes

  Os testes elaborados durante a fase de projecto de testes servirão de entrada para a actividade de teste. Esses testes podem ser traduzidos para uma linguagem de teste como a TTCN, de modo a automatizar o teste. A análise dos testes aplicados também pode ser facilitada pela utilização do projecto detalhado como entrada, isto é, o teste é realizado sobre um modelo abstracto: medições de facto são confrontadas com a especificação SDL e OMT.

6.3 Benefícios

Esta técnica permite que os analistas e projectistas de sistemas TR utilizem as notações que melhor satisfazem as necessidades das diferentes fases do processo de engenharia, assegurando simultaneamente a cobertura completa das actividades e a consistência contínua entre as diferentes fases do desenvolvimento. Através da utilização de técnicas OO em conjunto com técnicas formais, os utilizadores obtêm os benefícios da reutilização, validação antecipada e geração automática de código.

Comparativamente a outras abordagens, o OORT lida melhor com as questões de TR (do que, *e.g.* o OMT) e cobre melhor a fase de análise (do que, *e.g.* Booch e OOSE). O método OORT acenta em normas maduras e provadas pela indústria como a SDL e o MSC. A escolha de melhorar uma notação OO com normas existentes em vez de inventar novas extensões está de acordo com os desafios recentes da engenharia de *software*, por exemplo, Jacobson em [Jac95] recomenda a utilização de SDL como complemento à sua técnica de modelação por casos de utilização.

Parte II

Especificação de um Sistema de Controlo

Neste capítulo põe-se em prática o método OORT aplicado à especificação de um sistema de controlo. Começa-se por apresentar o sistema, depois descreve-se cada uma das fases do processo de desenvolvimento do sistema, desde os requisitos do utilizador até à implementação.

7 Apresentação do Sistema

Como exemplo, tomou-se o caso do sistema elevatório composto por múltiplos elevadores. Escolheu-se este caso por ser um típico sistema distribuído com requisitos de tempo-real.

7.1 Requisitos do Utilizador

O requisito geral é controlar e escalar vários elevadores para transportar pessoas entre os pisos de um edifício da forma convencional. O sistema possui ainda um posto de supervisão e comando da frota de elevadores.

Em cada **piso** encontram-se:

- um **botão de chamada** por sentido possível da deslocação pretendida (um para subir no piso inferior, um para descer no piso superior, e um para subir e outro para descer nos restantes pisos), com a respectiva indicação;
- uma **luz** de chamada associada a cada botão de chamada, que deve ficar ligada desde o momento em que o botão é premido até que um elevador pare no respectivo piso com a mesma direcção;
- uma **porta** de acesso a cada um dos elevadores;
- um **indicador de estado** por elevador indicando o piso em que este se encontra e o sentido actual do respectivo movimento.

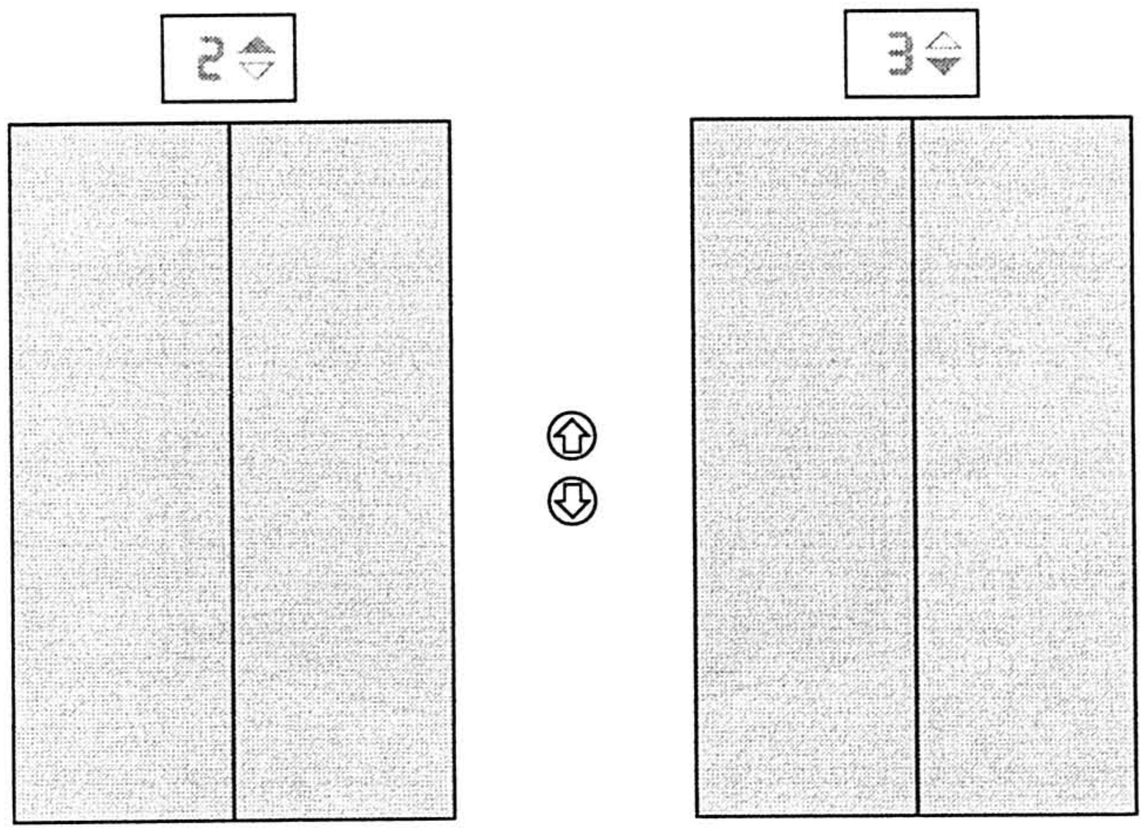


Figura 7.1: Acesso aos Elevadores num dos Pisos Intermédios.

Cada elevador contém:

- um **botão de destino** por cada piso do edifício com a respectiva indicação;
- uma **luz** de destino associada a cada botão de destino, que deve ficar ligada desde que o botão é premido até que o elevador pare no respectivo piso;
- uma **porta** de acesso ao piso;
- um **botão de abrir** e um **botão de fechar** a porta;
- um **sensor de piso** que detecta a proximidade de cada piso a uma distância predeterminada suficiente para permitir a paragem do elevador nesse piso;
- um **indicador de estado** indicando o piso em que este se encontra e o sentido actual do respectivo movimento;
- um **altifalante** para indicar aos passageiros a paragem do elevador num piso.

Na caixa de cada elevador existe:

- um **sensor de tensão** que detecta situações de carga excessiva e ruptura do cabo do elevador;
- um **freio de segurança** que trava o elevador em qualquer posição;
- um **motor** que move o elevador para cima e para baixo e que o pára nos pisos pretendidos.

O posto central de supervisão e comando permite visualizar a informação relativa a cada elevador, registar alarmes e eventos dos sistema, e desligar o sistema.

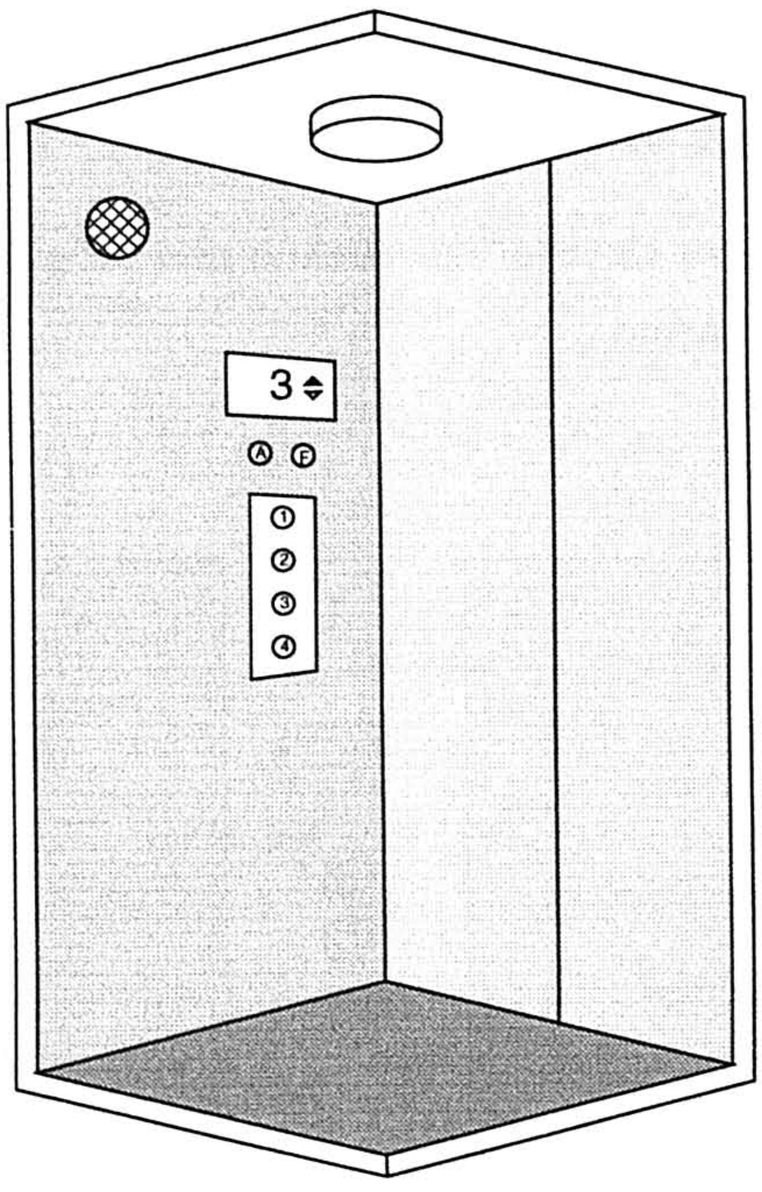


Figura 7.2: Interior de um Elevador.

O sistema recebe pedidos que se dividem em chamadas e destinos. Obviamente, num ambiente de vários elevadores, estes concorrem para servir as chamadas, sendo que os destinos continuam a ser servidos exclusivamente pelo respectivo elevador.

Regras de eficiência:

- se existir uma chamada pendente no piso P no sentido S, o próximo elevador que passe em P com o sentido S deve parar e aceitar passageiros;
- se um elevador não tem qualquer destino pendente, nem é o elevador mais próximo de qualquer chamada pendente, então deve permanecer imóvel no último piso em que parou até que seja necessário;
- um elevador é necessário quando o sistema recebe um destino seu, ou uma chamada da qual ele é o elevador mais próximo;
- nenhum elevador deve inverter o sentido do movimento enquanto houver pedidos (destinos, ou chamadas) que podem ser servidos mantendo o mesmo sentido de movimento.

7.2 Notas e Comentários

Este sistema constitui uma simplificação de um sistema real, daí o seu nome Sistema Elevatório Simplificado (SES), já que não considera certos pormenores:

- um elevador no limite da sua capacidade não deve responder a mais chamadas;
- um botão de alarme dentro de cada elevador para accionar o sinal de emergência;
- um comutador pára/viaja para alternar o seu modo de funcionamento entre parado num piso (no próximo piso por onde passar) e viajar para servir os pedidos pendentes;
- comando do elevador pelo operador no modo fora-de-serviço.

O controlo dos motores é feito por um sistema próprio que obedece a três comandos (subir, descer e parar) e que garante a paragem exacta do elevador nos pisos. Além disso, não responde a comandos inseguros (*e.g.*, se nenhum sensor de piso estiver actuado o controlador ignora o comando de paragem até que o elevador chegue a um piso, ou subir quando o elevador está a descer).

O controlo das portas também é realizado por um sistema externo que aceita os comandos abrir e fechar.

O sensor de piso é um dispositivo que detecta a proximidade de um piso quando o elevador entra numa zona predeterminada de proximidade dos pisos.

Este sistema tem características de tempo-real, contudo não são restrições temporais difíceis de gerir.

8 Análise de Requisitos

8.1 Introdução

A análise de requisitos pode começar antes da identificação dos elementos da arquitectura (apesar de muitos dos novos sistemas serem derivados de arquitecturas existentes). A análise concentra-se em **o que** o sistema deve fazer e não em **como** ele fornecerá os serviços esperados. O modelo de objectos e o modelo de casos de utilização são independentes das soluções que serão escolhidas para a implementação real do sistema.

Serão identificadas as entidades relevantes do domínio do sistema e integradas como objectos em diagramas de classes e instâncias. O comportamento do sistema será definido de um ponto de vista externo identificando interacções que ocorrerão entre o sistema e os elementos que compõem o seu ambiente.

Geralmente, esta actividade é realizada pelos utilizadores finais do sistema e pelos produtores do sistema, que são peritos no respectivo domínio de aplicação, mas não têm necessariamente um grande conhecimento de técnicas de projecto de *software*.

Esta fase é constituída por duas fases: modelação por objectos recorrendo aos diagramas de classes e diagramas de instâncias OMT, e modelação de casos de utilização através de MSCs.

8.2 Modelação por Objectos em OMT

A utilização da OMT limita-se aos diagramas de classes e instâncias. Em primeiro lugar, criam-se diagramas de instâncias que representam o domínio do sistema como um conjunto de objectos distintos relacionados por ligações. Estes objectos são depois ordenados em classes de objectos e as ligações entre eles são agrupadas formando associações entre classes.

Os outro dois aspectos da OMT, nomeadamente a modelação funcional e a modelação dinâmica, são obtidos através da modelação de casos de utilização descrita mais adiante.

A aplicação da OMT é guiada por um conjunto de tarefas que a seguir são descritas:

1. Identificar objectos e classes;
2. Identificar associações e agregações para indicar as relações existentes entre os objectos;
3. Identificar atributos das classes para modelar os dados;

4. Identificar operações de classes para modelar as transformações dos dados e as computações;
5. Introduzir herança para refinar e simplificar o modelo, e tornar certas classes mais genéricas;
6. Suprimir a redundância, ou mantê-la e torná-la explícita;
7. Clarificar aspectos complexos do modelo através de diagramas de instâncias;
8. Identificar os pontos de vista relevantes através de módulos apropriados;
9. Repetir os passos de 1 a 8 tantas vezes quantas as necessárias para conseguir um modelo completo.

Embora os autores do OORT proponham em [Ver96a] uma sequência ligeiramente diferente, considera-se esta uma abordagem mais eficaz para a construção de um modelo final. Está aliás, mais de acordo com o proposto por Rumbaugh [Rum91] e, mais recentemente, Douglass [Dou98] para o caso do UML.

As referências feitas nesta secção a entidades que fazem parte do modelo do sistema, referem-se às figuras apresentadas na secção 8.3.

8.2.1 Identificação de Objectos e Classes

Os objectos e classes identificados na fase de análise tanto podem representar entidades físicas (*e.g.* `Elevador` e `Piso`) como noções lógicas (*e.g.* `Gnomo`).

Durante a análise, tradicionalmente surge o problema de identificar uma arquitectura inicial para o sistema. Para o evitar, recomenda-se avaliar, no domínio do sistema, as classes que representam entidades físicas: as que não pertencem, geralmente, ao domínio devem ser excluídas. Isto também se aplica a noções lógicas, que devem representar conceitos da aplicação bem aceites e não soluções específicas de implementação. Por exemplo, uma base de dados composta pelas chamadas efectuadas deve ser modelada pela associação entre as classes `Chamada` e `Gnomo`, sem que nenhuma classe adicional seja introduzida para representar a base de dados.

Geralmente, as classes são encontradas analisando os nomes encontrados nos documentos textuais que definem os requisitos do utilizador, ou resultantes de entrevistas com o utilizador final. Obviamente, todos os nomes plurais são estritamente proibidos: existe uma classe e instâncias dessa classe (objectos).

8.2.2 Identificação das Associações e Agregações

8.2.2.1 Associações

Qualquer dependência entre classes é modelada como uma associação. As ligações e associações podem representar:

- equipamentos físicos ou dependências físicas, como a ligação entre `PainelDestino` e `Elevador`;
- relações de dependência lógica (produtor/consumidor, utilização), como a relação `Supervisiona` entre `Supervisor` e `Elevador`.

Uma associação representa uma dependência estática entre dois objectos e, por conseguinte, não tem direcção privilegiada: uma associação não representa um caminho de comunicação. O nome dado ao papel de uma classe numa associação deve ser cuidadosamente escolhido, de forma a definir precisamente a função de cada classe na associação, como a função `Escala` do `Gnomo` na sua associação com `Chamada`. É preferível dar nome aos papéis a nomear associações, já que os nomes das associações nem sempre têm significado nos dois sentidos.

As associações são enriquecidas especificando a sua multiplicidade, podendo também ter atributos e operações locais.

As associações são identificadas nos requisitos textuais analisando os verbos que relacionam os nomes.

8.2.2.2 Agregações

As associações que representem uma relação de posse ou de composição são geralmente melhor modeladas como agregações. Uma agregação adiciona uma nova restrição relativamente a uma simples associação: o comportamento do objecto agregador tem uma influência fundamental no comportamento do objecto agregado. Em particular, criar ou apagar objectos agregadores implica criar ou apagar objectos agregados.

Todas as associações com nome "parte de", "composto por", "elemento de", *etc.*, devem ser convertidas em agregações. As agregações geralmente não têm nome.

8.2.3 Identificação de Atributos

Os atributos são propriedades das classes e servem para expressar dados que constam nos requisitos reunidos, como o `Piso` em que se encontra um `Elevador`. Os detalhes de projecto e implementação, tais como apontadores para aceder a estruturas de dados ou classes, têm que ser eliminados.

Atributos complexos, como dados estruturados ou de tamanho ilimitado (*e.g.,* listas ligadas), devem ser transformados em classes e devem ser introduzidas agregações e associações adequadas. Os atributos podem ter um tipo, mas descrições sofisticadas de tipos devem ser evitadas se corresponderem a detalhes de implementação. Os atributos não podem ser apontadores, em vez disso devem ser utilizadas associações.

8.2.4 Identificação de Operações

As operações enriquecem as classes e correspondem a serviços esperados destas, como `Abre` e `Fecha` da classe `Porta`, ou à produção de dados que as classes devem fornecer ao seu ambiente, como

`VerificaCarga` da classe `MonitorCabo`. Também podem definir-se assinaturas (*signatures*) de operações: parâmetros formais e valores de retorno. Os parâmetros e valores de retorno podem ser classes, pois ao contrário dos atributos que expressam propriedades estruturais, uma operação é um evento efémero para o qual não há necessidade de criar uma associação.

Uma operação tem que ser declarada na classe que a fornece como um serviço externo e não na classe que a utiliza.

As operações são difíceis de encontrar aquando da modelação por objectos. No entanto, a modelação dinâmica dos requisitos dá uma grande ajuda na identificação das operações que as classes devem fornecer.

8.2.5 Redundância

As classes redundantes, que representam o mesmo conceito ou entidade física, ou com os mesmos atributos, têm que ser eliminadas. Porém, em alguns casos, as classes redundantes podem ser mantidas como forma de salientar informação que irá requerer acessos frequentes. Essas classes devem ser explicitamente identificadas. Em OMT puro, elas designam-se "derivadas". Por exemplo, uma classe pode exprimir o resultado de uma computação realizada utilizando entradas especificadas noutras classes. Teoricamente, esta noção poderia ser modelada com uma única operação, mas se a computação é complexa e o resultado é frequentemente utilizado, recomenda-se a criação de uma classe redundante. É claro que cabe ao projectista de *software* decidir qual é a melhor forma de implementar a classe redundante: criar dados redundantes persistentes, introduzir valores persistentes intermédios, optimizar o algoritmo da computação, *etc.*

Associações redundantes têm que ser removidas, ou explicitamente assinaladas como tal (derivadas) para indicar acesso frequente a classes ligadas através de caminhos indirectos.

8.2.6 Introdução de Herança

As classes com elementos estruturais comuns, *i.e.* atributos, operações ou associações, podem ser reorganizadas pela introdução de novas classes que incluam essas subestruturas comuns. As primeiras podem então tornar-se subclasses especializadas das novas superclasses. A introdução de superclasses é também favorecida pelo esforço empregue na reutilização de classes do domínio da aplicação que foram especificadas no passado. Contudo, na fase de análise devem evitar-se as árvores de herança muito profundas.

8.2.7 Identificação de Diagramas de Instâncias

Os diagramas de instâncias podem ser introduzidos para clarificar aspectos particulares dos sistemas: uma situação típica, o arranque do sistema, uma terminação anormal, *etc.*

Os diagramas de instâncias também podem ser criados para clarificar conjuntos complexos de classes interrelacionadas. Por exemplo, o diagrama de instâncias `Edifício de Quatro Pisos com Dois Elevadores`" mostra uma possível configuração do diagrama de classes `Arquitectura do SES`.

8.2.8 Identificação de Módulos

Os pontos de vista importantes para a análise de requisitos são expressos por módulos. Um módulo reflecte uma determinada perspectiva do sistema e mostra quais as classes, associações e heranças envolvidas. As classes podem aparecer em vários módulos, que é geralmente o caso, já que os pontos de vista não formam partições rígidas do sistema. Da mesma forma, associações e heranças podem ser encontradas em mais que um módulo, ainda que a repetição seja desaconselhada.

Os módulos que geralmente constituem o modelo por objectos, são:

- módulo de descrição do ambiente, principalmente para identificar os actores, como `Edificio`;
- um módulo para cada tipo de actor, por exemplo o utilizador final e o administrador do sistema, como `Piso` e `Elevador` que, respectivamente, representam as perspectivas do `PotencialPassageiro` e do `Passageiro`;
- um módulo para cada fase funcional, caso a operação do sistema possa ser dividida em diferentes fases;
- modulo descrevendo a arquitectura genérica do sistema, como `Arquitectura`.

Para facilitar a leitura dos diagramas, cada módulo deve conter no máximo vinte classes. Dependendo da complexidade de cada classe (número de atributos, operações e associações), o número ideal de classes por módulo não excede as dez.

8.2.9 Iterações no Modelo de Objectos

A modelação por objectos é um processo iterativo. O diagrama de classes inicial é muito próximo dos requisitos recolhidos: nomes são utilizados como classes, verbos como associações, *etc.* Após iterações sucessivas, o modelo é reorganizado de modo a ser não ambíguo, não redundante, facilmente compreensível, ausente de detalhes de projecto e de implementação, e condizer exaustivamente com os requisitos do utilizador. Em particular, a identificação de atributos e operações para cada classe será tão completa quanto possível. A modelação dinâmica também conduz a modelos de objectos mais enriquecidos, especialmente no que concerne a operações e associações de classes. Portanto, as iterações finais no modelo de objectos devem ser realizadas em paralelo com as iterações finais no modelo dinâmico.

Geralmente, as versões iniciais são maiores e mais complicadas comparativamente às versões finais. O número de classes normalmente diminui com o decorrer deste processo. De facto, na primeira iteração,

recomenda-se criar classes sem que necessariamente se tenha em mente uma ideia clara do que elas representarão no fim. Ao iterar o modelo, algumas dessas classes serão fundidas, outras transformadas em associações, se realmente descreverem relações entre classes existentes, em atributos, se representarem dados com significado somente no contexto de uma classe, ou em operações, se representarem informação transitória. Classes vazias, *i.e.* classes sem atributos e fracamente ligadas com as outras classes do modelo, têm que ser removidas.

A modelação por objectos é conseguida assim que o modelo esteja completo. Há dois aspectos que têm que ser verificados para assegurar a plenitude dos objectos:

- Navegabilidade

  Deve ser possível navegar através do modelo para aceder a todas as porções de informação nele contida. Se um caminho de navegação pretendido não é suportado, mesmo que indirectamente através das associações existentes, novas associações têm que ser criadas. O modelo dinâmico é muito útil para verificar a plenitude dos caminhos de acesso, porque mostra quais os objectos que são postos em interacção e quais as partes de informação (transmitidas pelas mensagens) acedidas por este e aquele objecto.

- Exaustividade dos dados

  Todos os dados pretendidos têm que ser incluídos no modelo. Se uma porção de dados, como um atributo, um conjunto de atributos, ou o resultado de uma operação, falta no modelo, novos atributos ou operações devem ser introduzidos.

A verificação da plenitude do modelo é realizada pela aplicação de perguntas: todos os dados pretendidos têm que aparecer nos objectos e têm que existir caminhos de acesso entre as classes que permitam chegar até aos dados necessários.

8.3 Modelo de Objectos do SES

A partir dos requisitos expressos na secção 7.1, criaram-se quatro módulos de diagramas de classes e um diagrama de instâncias, respectivamente:

- `Edificio`;
- `Arquitectura`;
- `Acesso do Piso`;
- `Elevador`;
- `Eixo de Elevador`;
- `Passageiro e PotencialPassageiro`
- `Distribuicao de Chamadas`

- Supervisao de Eventos
- Botoes
- Edificio de Quatro Pisos com Dois Elevadores.

8.3.1 Edifício

Este módulo reflecte o enquadramento do SES no edifício. Do ponto de vista que interessa ao sistema, um edifício é constituído por NP pisos. Em cada piso existe um acesso ao sistema. O sistema é constituído por NP acessos, NE elevadores cada um com o seu eixo, e por um posto de supervisão. Este módulo está representado na Figura 8.1.

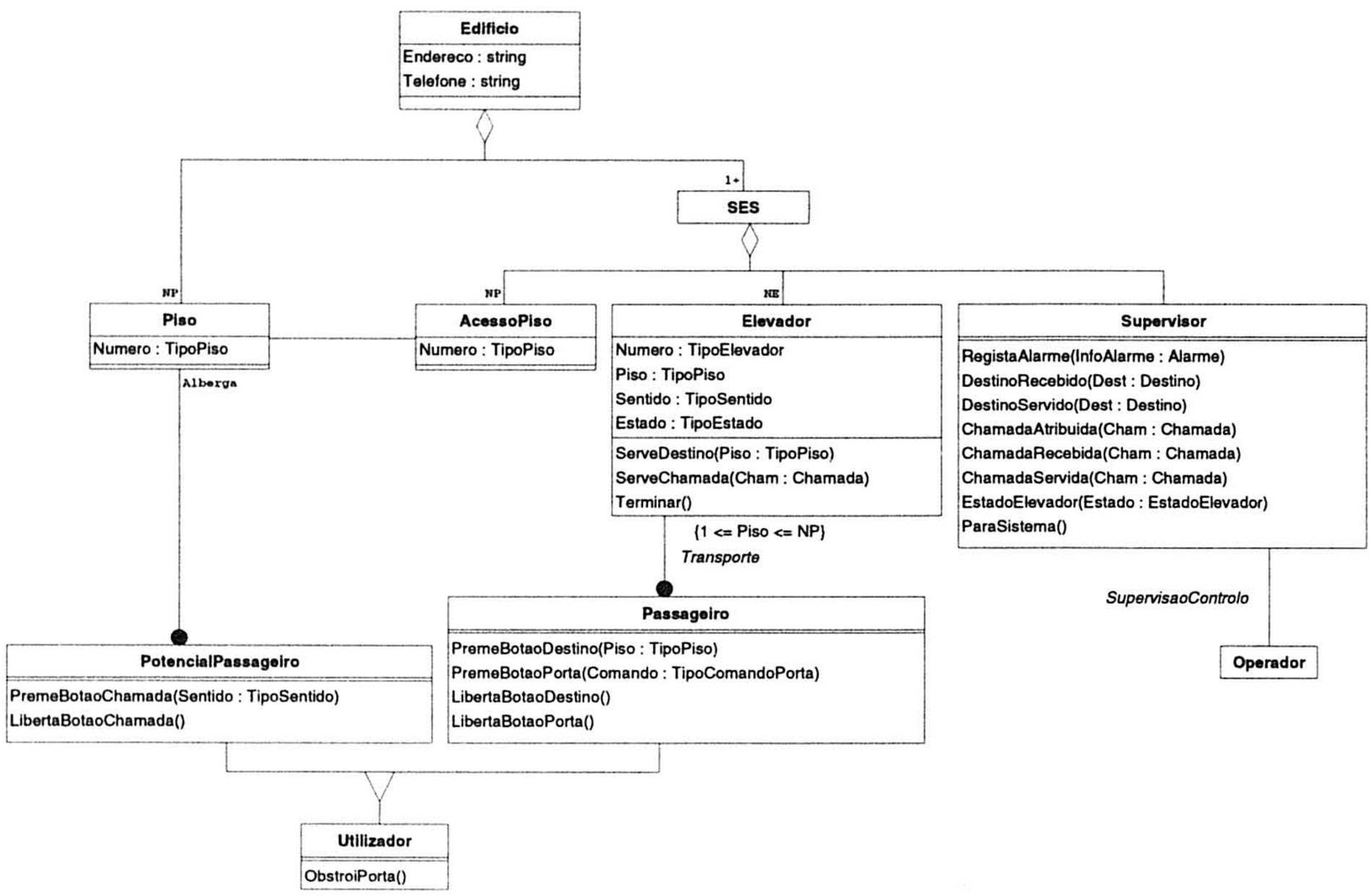


Figura 8.1: Diagrama de Classes OMT do Módulo Edifício do SES.

8.3.2 Arquitectura

Este módulo descreve a arquitectura do SES em termos de classes. A arquitectura do SES é um dos requisitos iniciais. As várias entidades físicas são modeladas como classes interagindo com o utilizador.

De forma a manter a legibilidade dos diagramas, a arquitectura foi partida hierarquicamente em quatro módulos. O módulo da Figura 8.2 é o primeiro e corresponde ao níveis superiores da hierarquia de classes do SES. Os três seguintes correspondem ao níveis inferiores da hierarquia de classes da arquitectura.

8.3.3 Acesso do Piso

Este módulo descreve a parte da arquitectura do sistema respeitante aos componentes da classe `Piso`. Cada piso contém: `NE` portas e indicadores de estado, um por elevador; um ou dois botões de chamada, consoante seja, ou não, um piso extremo, respectivamente. Este módulo está representado na Figura 8.3.

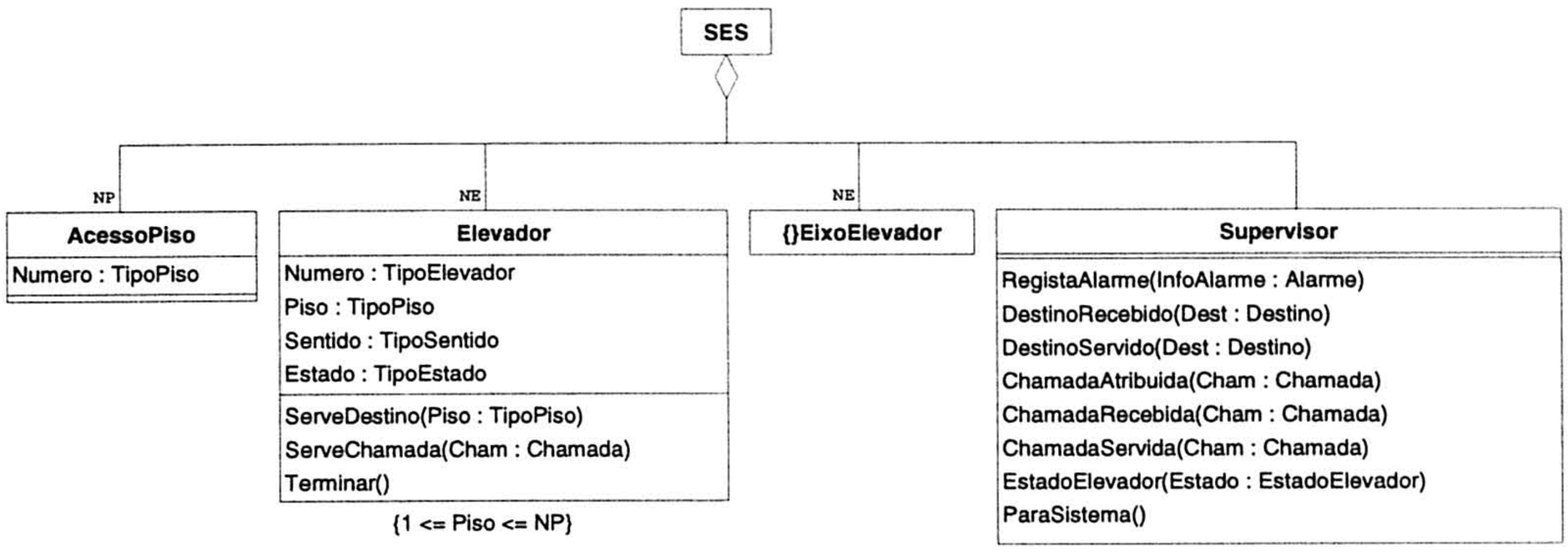


Figura 8.2: Diagrama de Classes OMT do Módulo Arquitectura do SES.

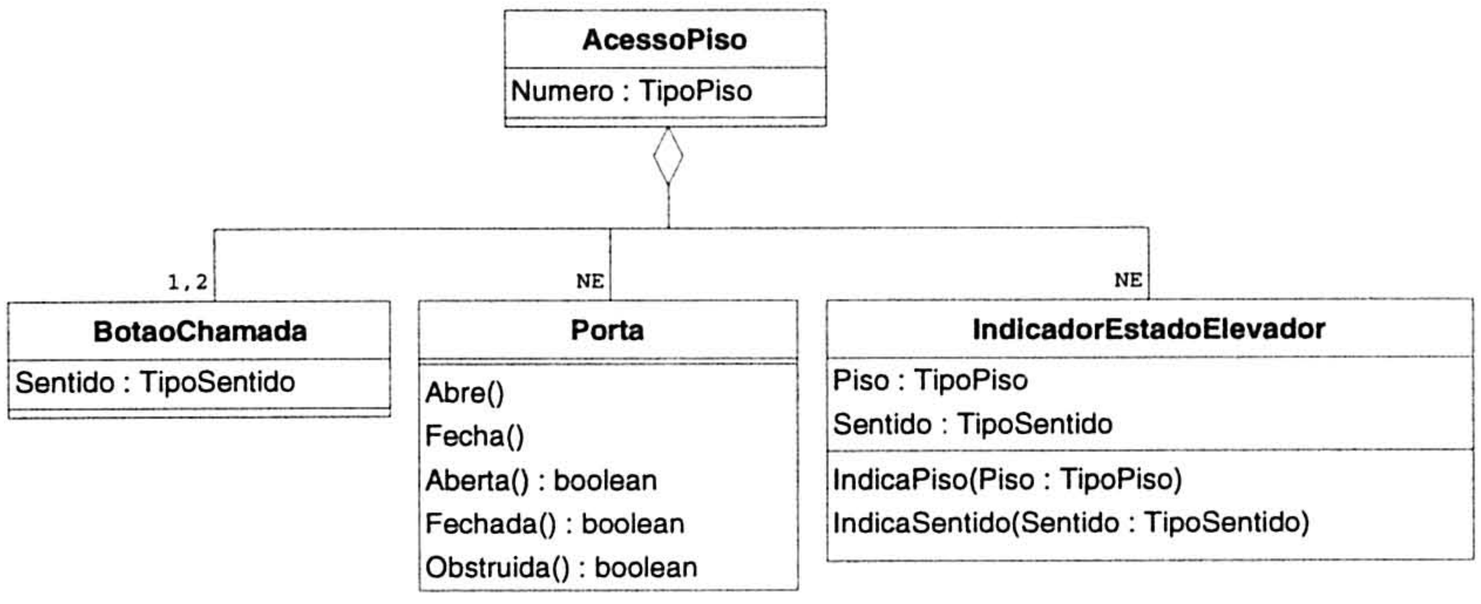


Figura 8.3: Diagrama de Classes OMT do Módulo Piso do SES.

8.3.4 Elevador

Este módulo descreve a parte da arquitectura do sistema respeitante ao agregado da classe `Elevador`. Cada elevador é constituído por: NP botões de destino, um por piso; dois botões de comando da porta, um para abrir e outro para fechar; uma porta; um indicador de estado; um altifalante; e, um sensor de piso. Este módulo está representado na Figura 8.4.

8.3.5 Eixo do Elevador

Este módulo mostra as classes que modelam os dispositivos físicos que constituem o espaço do edifício onde se movimenta cada um dos elevadores. Em cada eixo de elevador existe um motor que move o elevador através de um cabo, um sensor de tensão no cabo e um freio que serve para segurar o elevador caso o cabo ceda. Este módulo está representado na Figura 8.5.

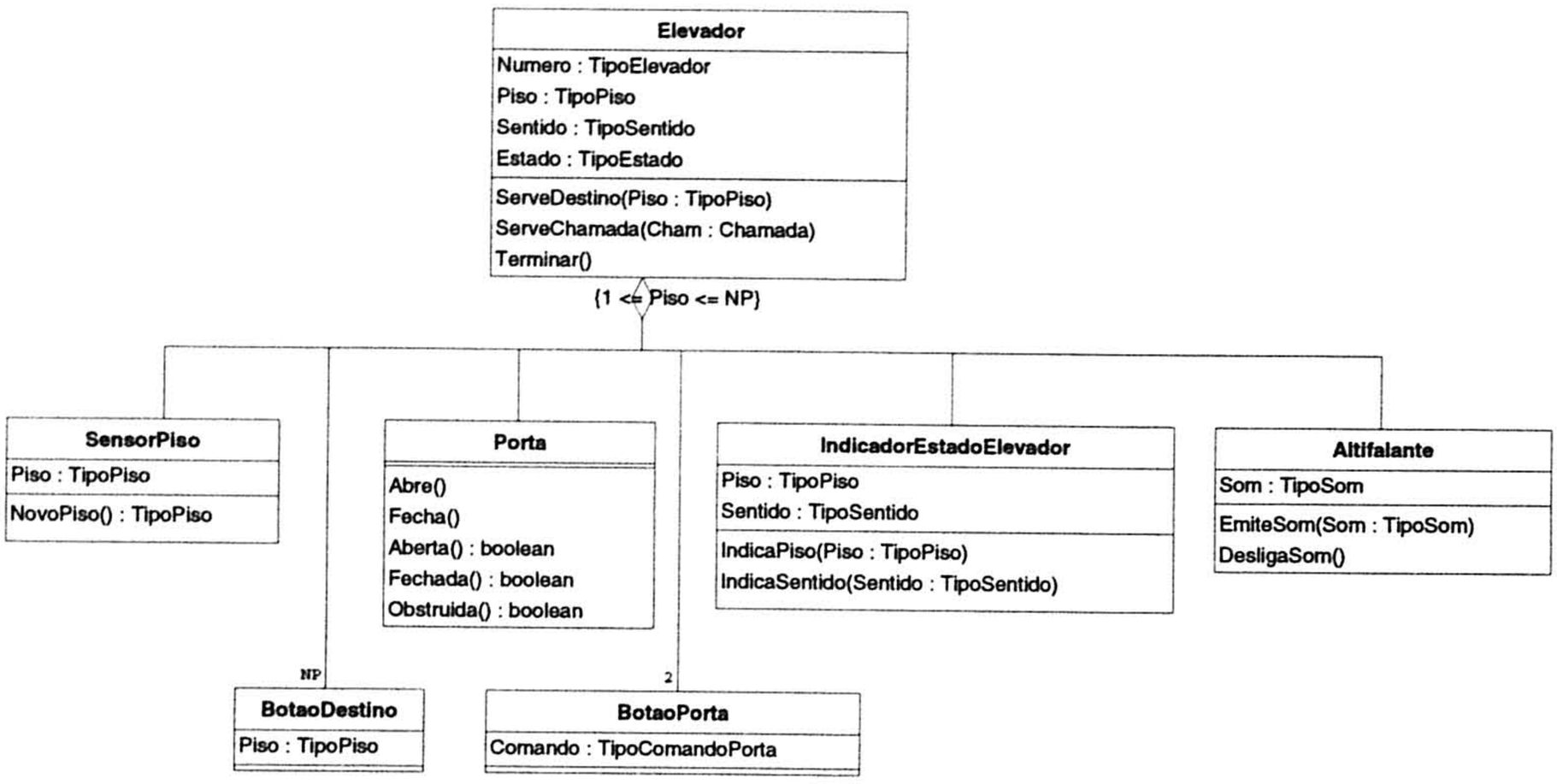


Figura 8.4: Diagrama de Classes OMT do Módulo Elevador do SES.

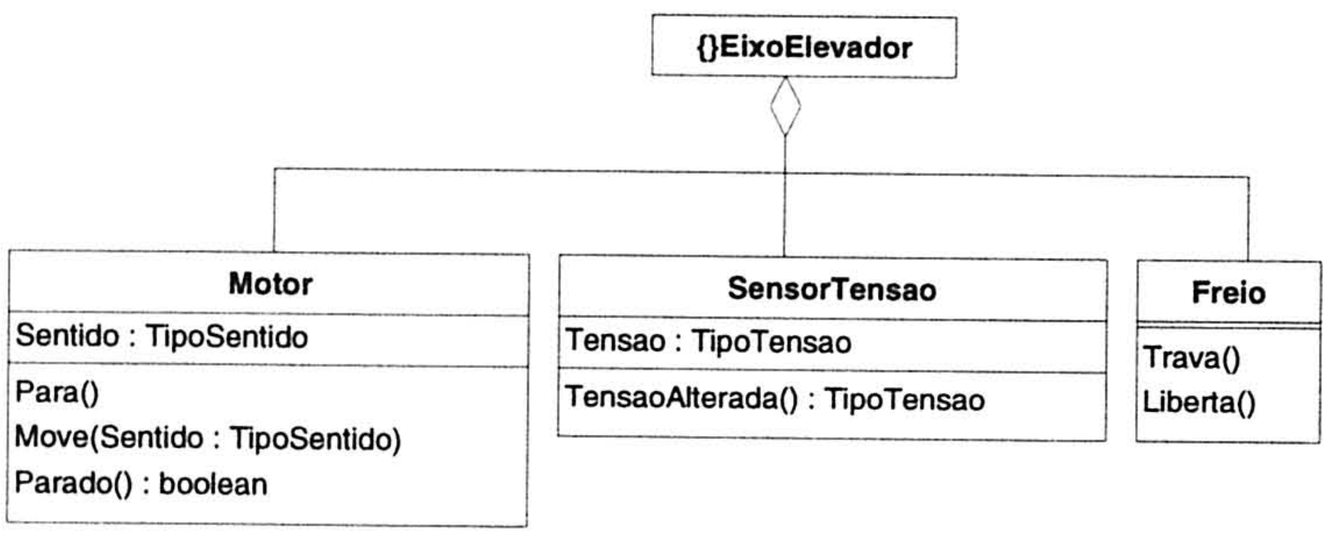


Figura 8.5: Diagrama de Classes OMT do Módulo Eixo do Elevador do SES.

8.3.6 Passageiro e Potencial Passageiro

Este módulo descreve a distinção/união entre passageiros e potenciais passageiros. Reflecte também o ponto de vista que estes têm do SES, modelando a maneira como ambos influenciam o SES e a informação envolvida nessas interacções. Este módulo está representado na Figura 8.6.

8.3.7 Distribuição de Chamadas

Neste módulo manifesta-se a necessidade de distribuir as chamadas entre os elevadores que concorrem para as servir. Para isso introduz-se a classe `Gnomo` como sinónimo de entidade inteligente responsável pela tarefa de decidir o que fazer com as chamadas. Note-se que este é um conceito consensual já que inevitavelmente num sistema de múltiplos elevadores, estes concorrem para servir chamadas que o sistema recebe. Este módulo está descrito na Figura 8.7.

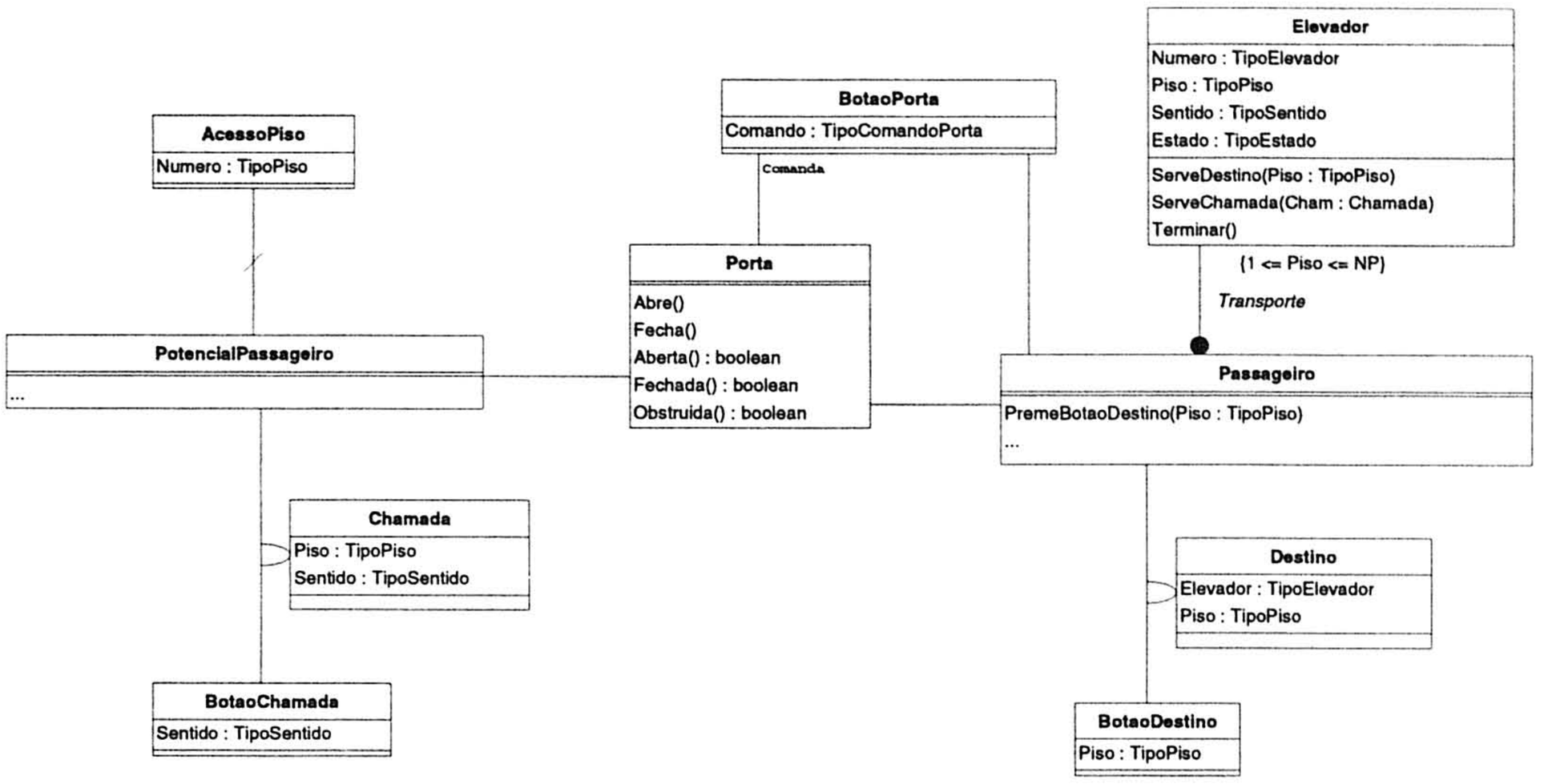


Figura 8.6: Diagrama de Classes OMT do Módulo Passageiro e Potencial Passageiro do SES.

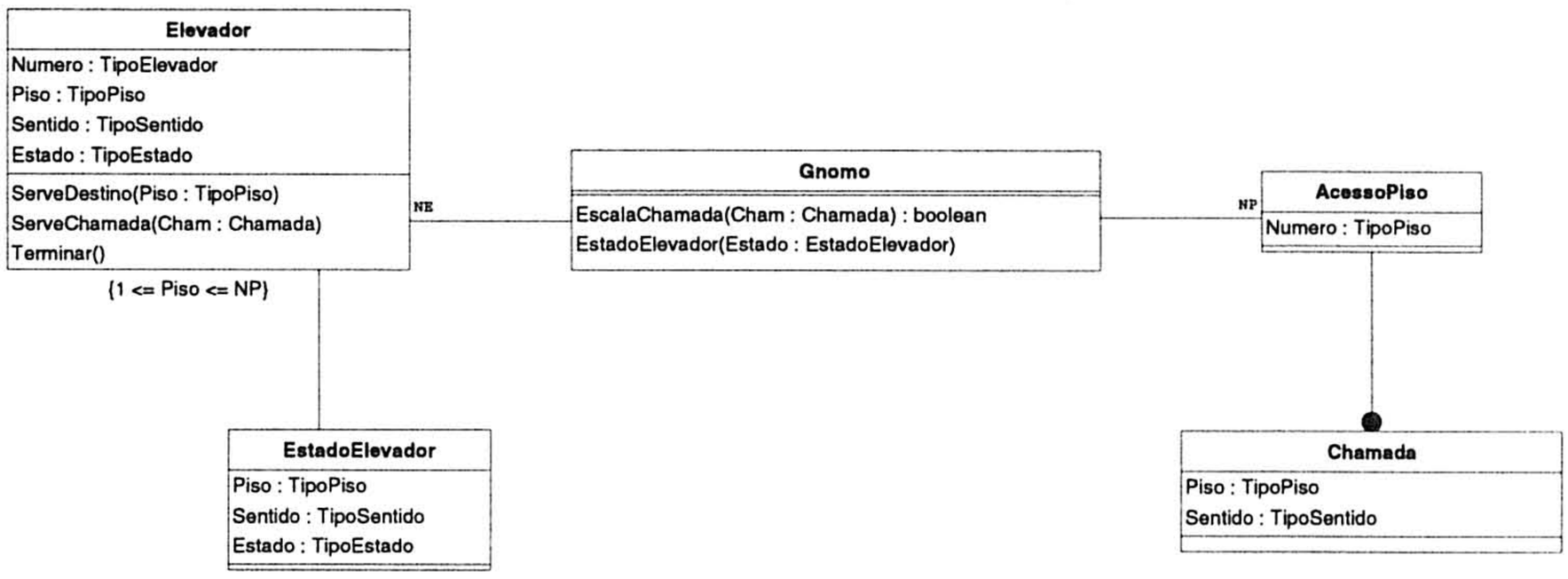


Figura 8.7: Diagrama de Classes OMT do Módulo Distribuição de Chamadas do SES

8.3.8 Supervisão de Eventos

Este módulo da Figura 8.8, descreve as dependências entre as classes da perspectiva da supervisão de eventos, bem como os dados envolvidos.

8.3.9 Botões

Este módulo mostra a generalização possível das classes do domínio dos Elevadores, através das heranças entre as classes relativas ao dispositivo físico `Botao`. Esta é uma parte da modelação onde a reutilização é importante, pois quando se dispõe de um repositório de modelos e classes razoável (que não é o caso), surgem as oportunidades para fazer tirar partido da reutilização através da especialização de classes genéricas. Este módulo está representado na Figura 8.9.

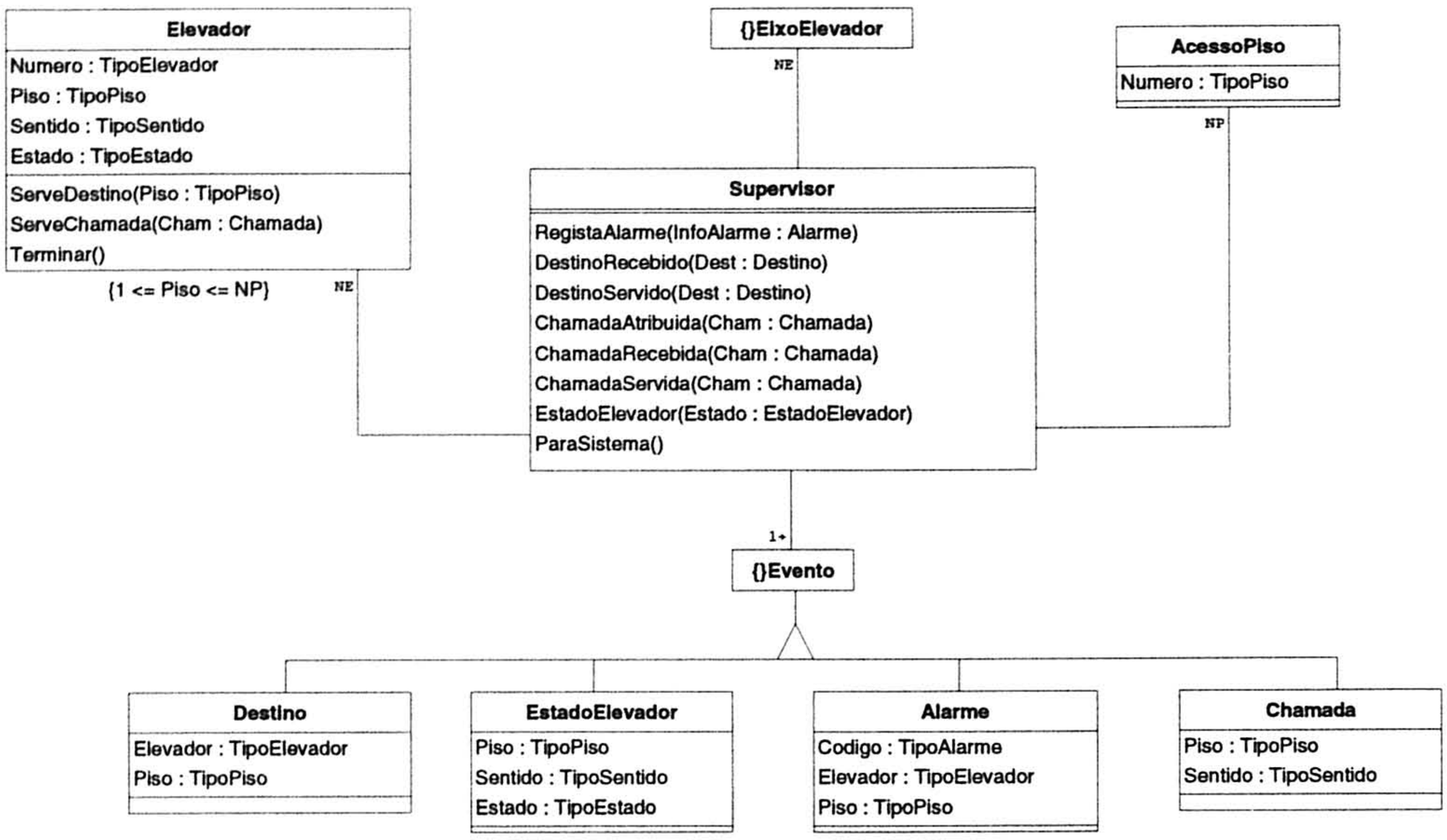


Figura 8.8: Diagrama de Classes OMT do Módulo Supervisão de Eventos do SES

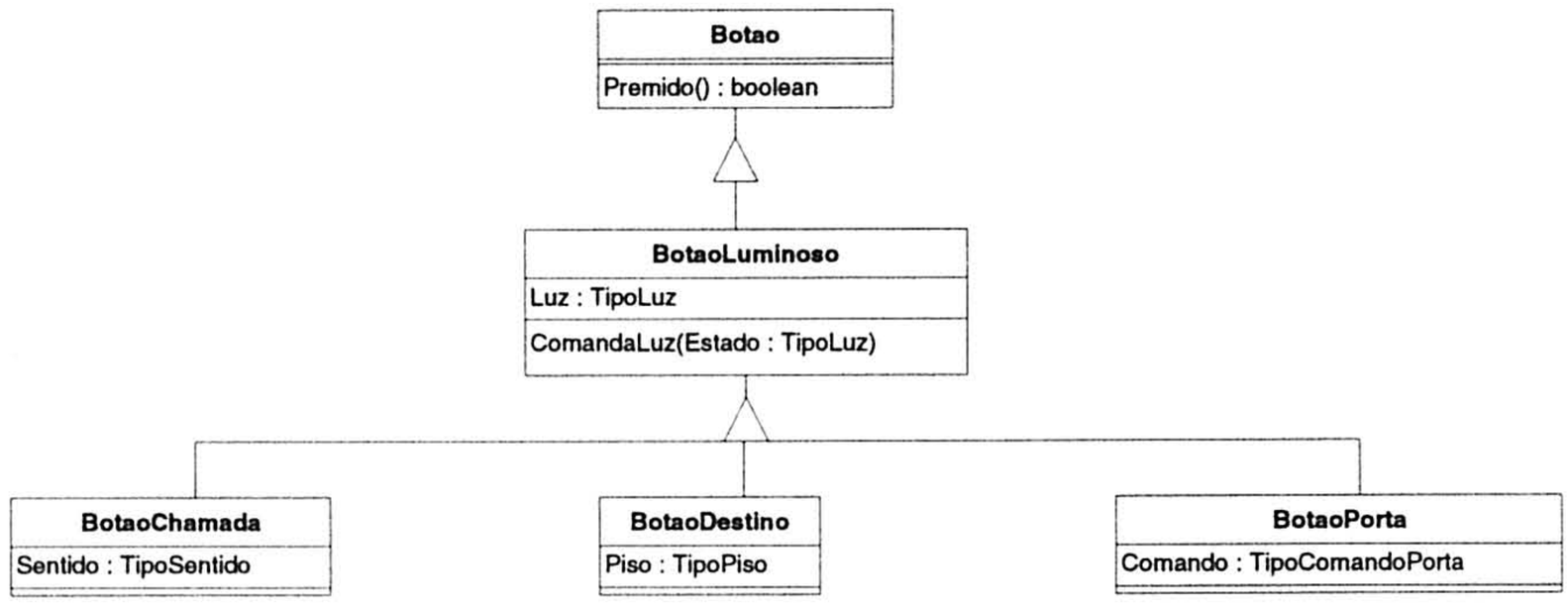


Figura 8.9: Diagrama de Classes OMT do Módulo Botões do SES.

8.3.10 Edifício de Quatro Pisos com Dois Elevadores

Este módulo é um diagrama de instâncias que pretende esclarecer qual o número de instâncias de cada classe da arquitectura de um SES num exemplo concreto. Na Figura 8.10 representam-se apenas algumas instâncias, por uma questão de espaço e facilidade de leitura, mas incluem-se as necessárias ao efeito pretendido.

8.3.11 Comentários ao Modelo:

Como se pode verificar as classes do sistema do elevador têm atributos muitos simples, porque esta aplicação não é caracterizada por ter dados estáticos, mas sim pelo seu comportamento dinâmico.

Obviamente, que pela necessidade de incluir no modelo toda a informação necessária, têm que se fazer algumas incursões em matéria de arquitectura, mas que não impõem limites às opções de arquitectura ou, então, são perfeitamente consensuais. A classe `Gnomo`, é disso um exemplo.

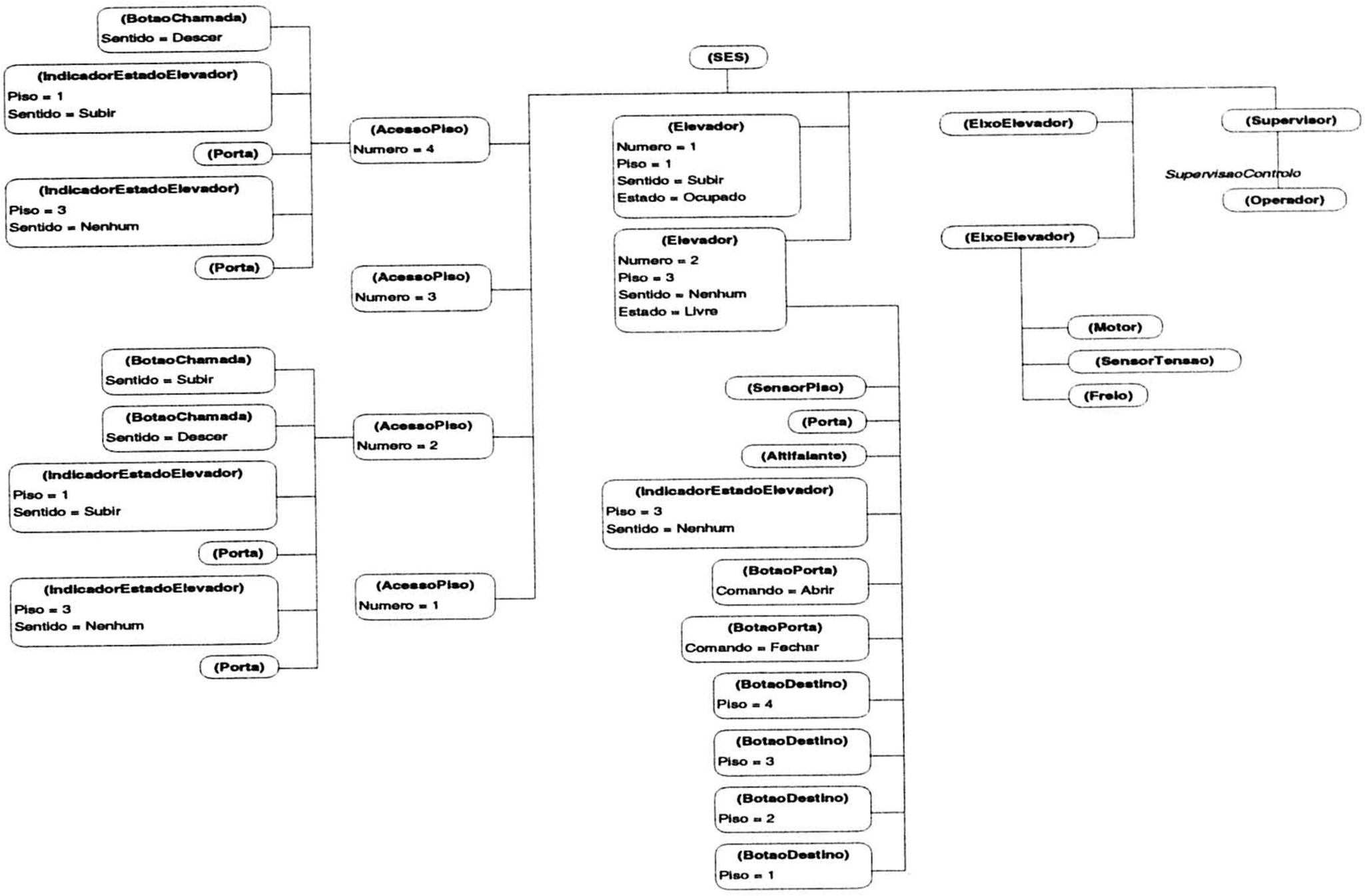


Figura 8.10: Diagrama de Instâncias OMT do Módulo Edifício de Quatro Pisos com Dois Elevadores

8.4 Modelação Dinâmica em MSCs

Durante a fase de análise de requisitos, o MSC é utilizado para modelar casos de utilização com o objectivo de descrever os requisitos funcionais e dinâmicos do sistema. A modelação de casos de utilização é um método em que o sistema é visto como um todo e que nesta fase não se podem assumir considerações sobre a sua arquitectura interna.

A modelação de casos de utilização é realizada em dois passos: primeiro, identificam-se as funções que o sistema deve fornecer e refinam-se estas em funções de nível sucessivamente inferior; depois, detalham-se todas as funções de baixo nível em termos de sequências de interacções entre o sistema e o respectivo ambiente.

Os termos **serviços**, **funções** e **cenários**, são utilizados de forma indiferente nesta secção.

8.4.1 Identificação de Cenários

Os cenários são identificados a partir dos requisitos relativos à dinâmica do sistema. Alguns descrevem o comportamento nominal do sistema, que geralmente é dividido em fases em que cada uma é um ramo da hierarquia de cenários. Refinamentos adicionais no primeiro nível podem surgir da partição dos serviços esperados em:

- Casos de utilização sem estímulos erróneos vindos do ambiente;
- Casos de utilização com erros provocados pelo ambiente que abortam as sequências normais;
- Funções de controlo e gestão utilizadas pelo ambiente ou pelo próprio sistema;
- Casos de utilização que consideram estragos no sistema e no ambiente, e mais genericamente todas as condições anormais.

Os cenários são refinados em cenários de nível inferior cada vez mais simples de modo a reduzir a complexidade. Ao nível mais baixo, todos os cenários devem ser facilmente descritos em termos de conjuntos de interacções. Normalmente, o refinamento de um cenário não deve exceder seis sub-cenários no sentido de preservar a legibilidade.

A duplicação de cenários é estritamente proibida na mesma hierarquia. Um cenário que pode ocorrer em várias situações (depois, ou em paralelo com outros) tem que aparecer na hierarquia a um nível tão alto quanto o necessário para autorizar todas as suas possíveis ocorrências.

Os operadores de composição – e, ou, paralelo, repetir e excepção – são utilizados em cada nível de refinamento para indicar como são agrupados os sub-cenários para formar o cenário de nível superior. Isto constitui uma grande ajuda na interpretação da hierarquia de cenários. É de referir que, se dois cenários podem correr em paralelo, os seus sub-cenários também podem correr em paralelo.

Os nomes a atribuir aos cenários devem ser cuidadosamente escolhidos de forma a informarem o melhor possível as funções que representam.

8.4.2 Hierarquia de Cenários do SES

A Figura 8.11 e a Figura 8.12 apresentam a hierarquia de cenários para o SES, dividida em duas partes para poder ser visível. Foram identificadas duas categorias de alto nível: uma relativa ao funcionamento normal e outra descrevendo situações de erro. No funcionamento sem estímulos erróneos distinguem-se dois cenários: a recepção de pedidos e as viagens dos elevadores. A recepção de pedidos (chamadas e destinos) é claramente uma função paralela às viagens dos elevadores. Os que estão parados (livres) podem iniciar uma viagem quando o sistema recebe um pedido que a eles é dirigido (cenários `ElevadorLivreIniciaViagem`). O cenário `Viagem` foi decomposto em fases de forma reduzir o número de cenários possíveis, através do emprego de operadores de composição. Por uma questão de simplicidade os cenários de erro considerados foram limitados.

8.4.3 Descrição de Cenários por MSCs

Os cenários terminais são descritos por meio de MSCs, denominados nesta fase de MSCs ao **nível do sistema**, que correspondem de forma aproximada aos diagramas de interacção da abordagem de Jacobson [Jac92].

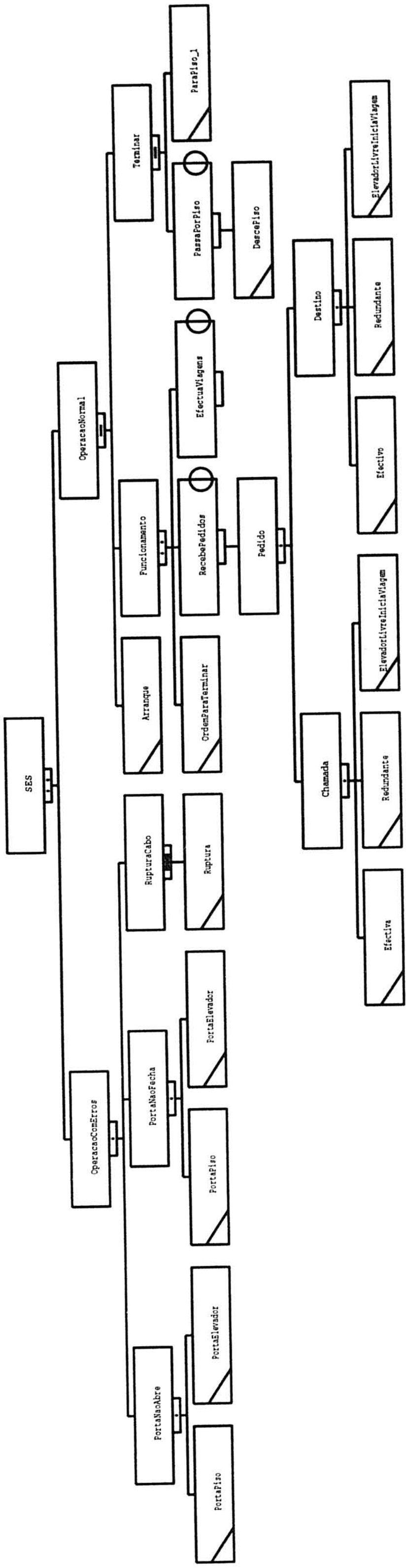


Figura 8.11: Hierarquia de Cenários do SES.

Figura 8.12: Hierarquia do Sub-cenário Viagens do SES

As referências feitas nesta secção a entidades que fazem parte da descrição do sistema, referem-se às figuras apresentadas na secção 8.5.

8.4.3.1 Identificação das Instâncias de um MSC

As instâncias que aparecem nos MSCs devem ser instâncias de actores definidos como classes no modelo de objectos, excepto o sistema, que não necessita de ser modelado como uma classe. Geralmente, as instâncias dos MSCs correspondem a objectos que representam:

- operadores humanos, *e.g.* `Passageiro` e `Operador`;
- entidades físicas externas, *e.g.* `Motor` e `Porta`;

O principal objectivo das entidades lógicas é representar dados (*e.g.*, `Chamada`). Por isso, elas não devem aparecer como instâncias, mas antes serem utilizadas como mensagens, ou dados das mensagens.

Num MSC, existem tantas barras verticais quantas as instâncias concorrentes do sistemas que comunicam entre si no mundo real. Por exemplo, no caso de uma rede de sistemas, os MSCs conterão várias instâncias do sistema. Analogamente, um actor pode ser instanciado mais que uma vez num MSC se essa condição ocorrer no mundo real, por exemplo, se o sistema interagir simultaneamente com instâncias concorrentes diferentes do mesmo actor. As diversas instâncias do mesmo actor devem ter nomes explicativos.

No caso em estudo, o número de dispositivos físicos externos com os quais o sistema interage é bastante elevado, daí a densidade dos MSCs apresentados em 8.5.

8.4.3.2 Identificação e Desenho das Mensagens Trocadas

As mensagens geralmente correspondem a:

- operações de classes, *e.g.* `Fecha da Porta` e `IndicaPiso` do `IndicadorEstadoElevador`;

- valores de retornados pelas operações, *e.g.* a mensagem `NovoPiso` corresponde ao valor retornado pela operação `NovoPiso` do `SensorPiso`.

No caso do resultado de uma operação corresponder a outra operação existente no modelo de objectos, pode ser utilizado o nome da segunda como resposta à primeira. Caso contrário, o nome da mensagem resultante da operação pode ser construído adicionando ao nome da operação os prefixos `Res` para resposta/resultado, ou `Con` para confirmação.

Por vezes, a mensagem recebida não corresponde directamente a uma operação da classe que a recebe, mas a uma operação de uma terceira classe agregada da classe receptora no modelo de casos de utilização. É o que acontece com as mensagens recebidas pelo sistema.

Os dados das mensagens geralmente correspondem a atributos, ou ao resultado das operações das classes.

Na fase de análise, não há necessidade para descrever formalmente os meios utilizados pelos actores e pelo sistema para interagirem. Os analistas devem ter presente a ideia de que as instâncias podem ser acedidas independentemente sem identificação nem problemas de conexão.

Não existe nenhuma relação directa entre as associações no modelo de objectos e as interacções entre actores, porque as associações reflectem propriedades estruturais enquanto as interacções são eventos efémeros. Todavia, para cumprir o critério de acessibilidade da informação, deve existir no modelo de objectos um caminho, directo ou indirecto, através de associações entre as instâncias em diálogo no modelo dinâmico. O conjunto de associações `BotaoChamada-AcessoPiso-SES-Elevador` é um caminho de comunicação entre `BotaoChamada` e `Elevador`.

8.4.3.3 Verificação da Plenitude do Modelo

Cada cenário terminal deve ser descrito por um MSC. O modelo dinâmico fica completo quando todos os cenários possíveis identificados na hierarquia estiverem descritos. Ainda assim, a plenitude é difícil de conseguir devido ao número enorme de interacções possíveis. As iterações no modelo dinâmico cessam quando o analista considerar que um número satisfatório de casos de utilização foram elaborados. Esta é claramente uma opção subjectiva e dependente do sistema em questão.

Para o caso do SES, o número de cenários possíveis facilmente atingiria as largas dezenas caso se tentasse explorar todas as alternativas, ou fosse necessário uma modelação mais exaustiva.

8.5 Modelo Dinâmico do SES

A seguir apresentam-se ao principais MSCs do SES. Pode ver-se que os digramas contêm muitas mensagens o que lhe dá um aspecto denso, no entanto, isso deve-se à natureza do sistema. A comunicação do ambiente com o sistema faz-se por meio de sinais eléctricos (relativos a acender luzes, ler estado de botões, *etc.*), daí os protocolos serem muito simples e os sinais numerosos.

Nas figuras seguintes apresentam-se apenas alguns dos MSCs relativos à hierarquia do SES da Figura 8.11 e da Figura 8.12. Escolheram-se os mais significativos em termos de descrição do funcionamento do sistema. No Anexo B apresentam-se todos os MSCs que descrevem completamente o comportamento dinâmico do sistema.

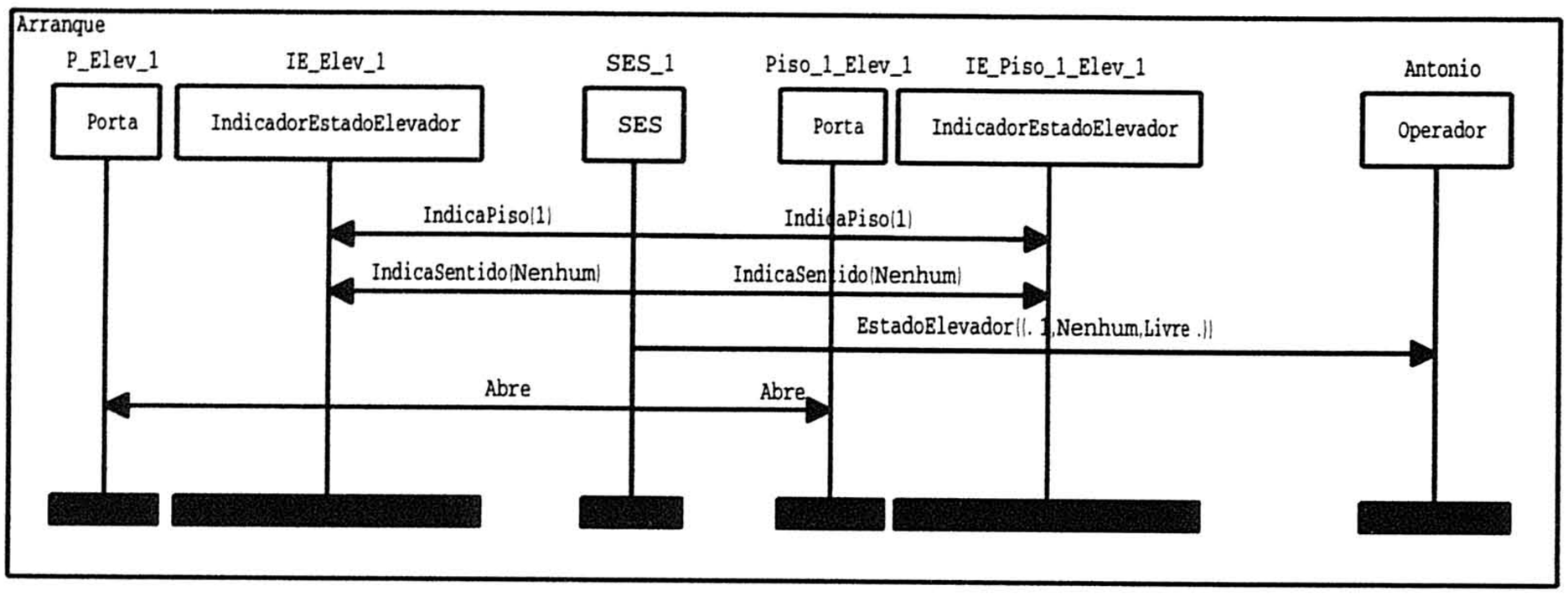


Figura 8.13: MSC do Arranque de um Elevador do SES.

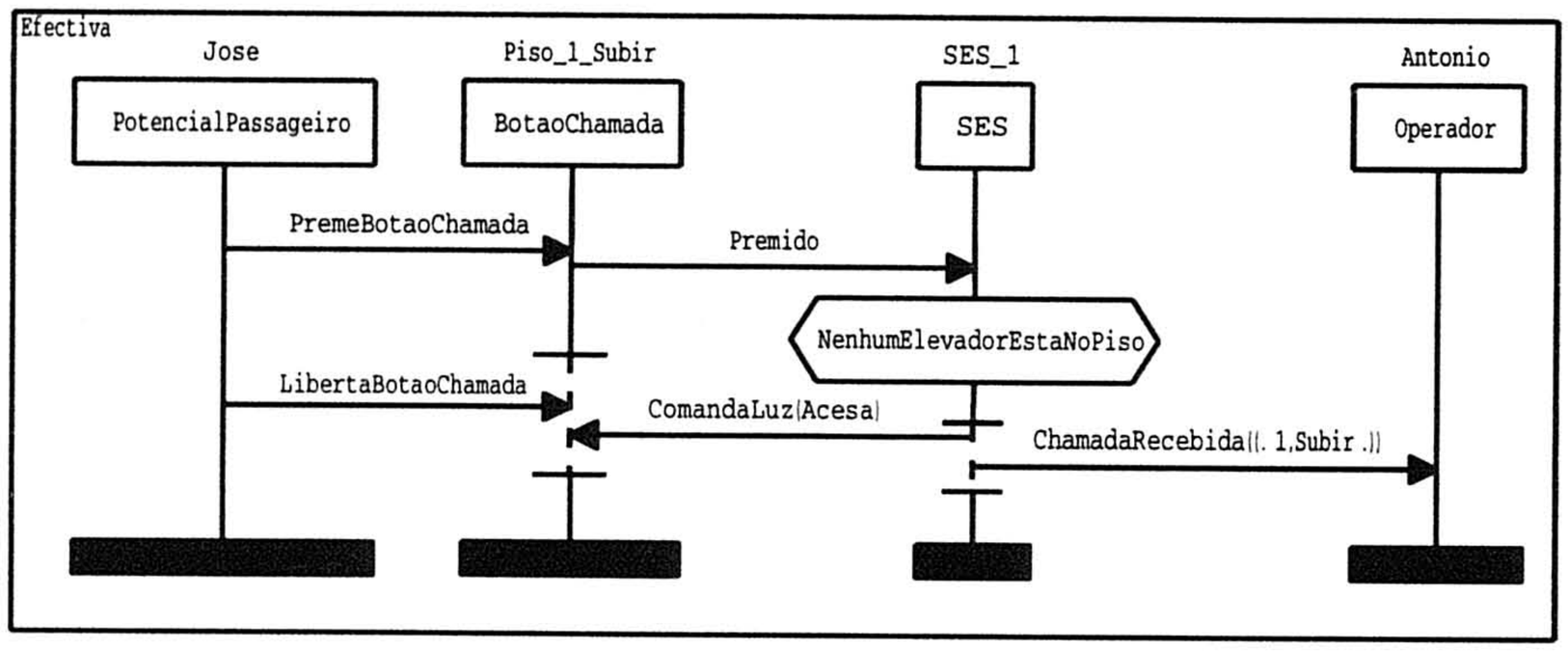


Figura 8.14: MSC de uma Chamada Efectiva.

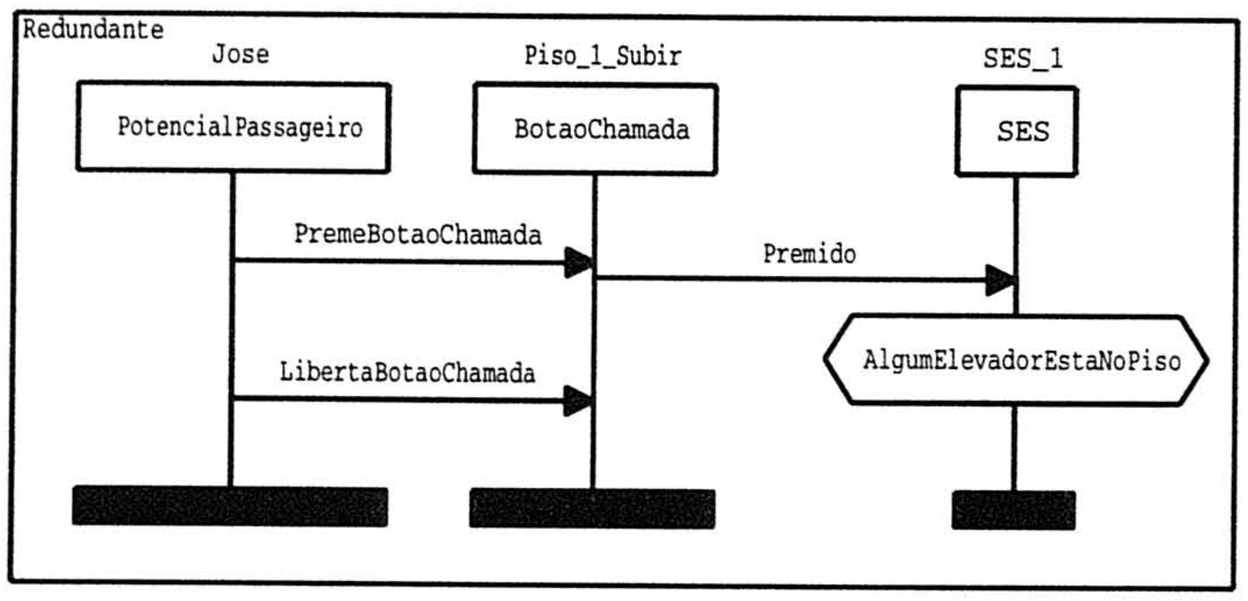


Figura 8.15: MSC de uma Chamada Redundante.

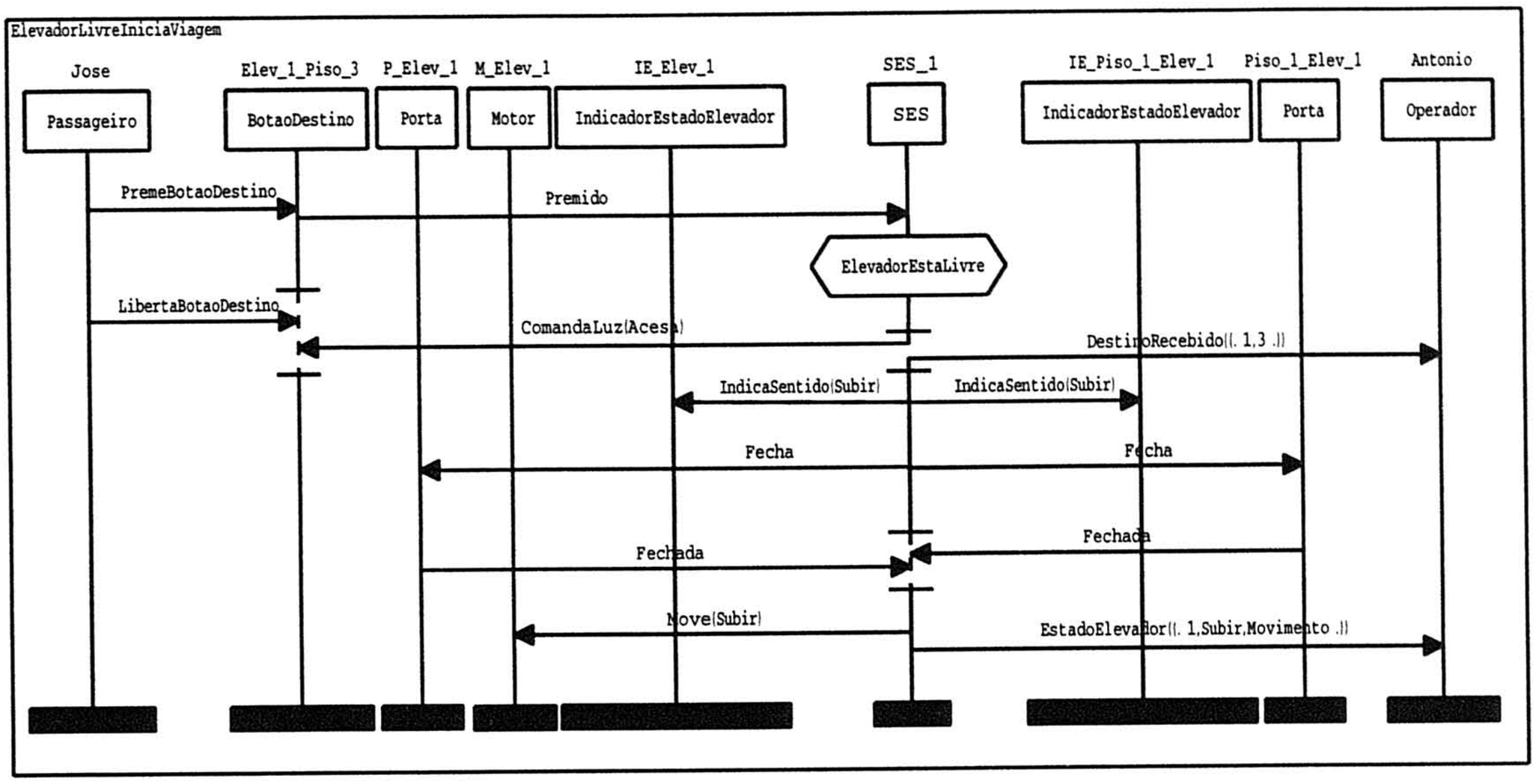


Figura 8.16: MSC do Início de uma Viagem de um Elevador Originado pela Recepção de um Destino.

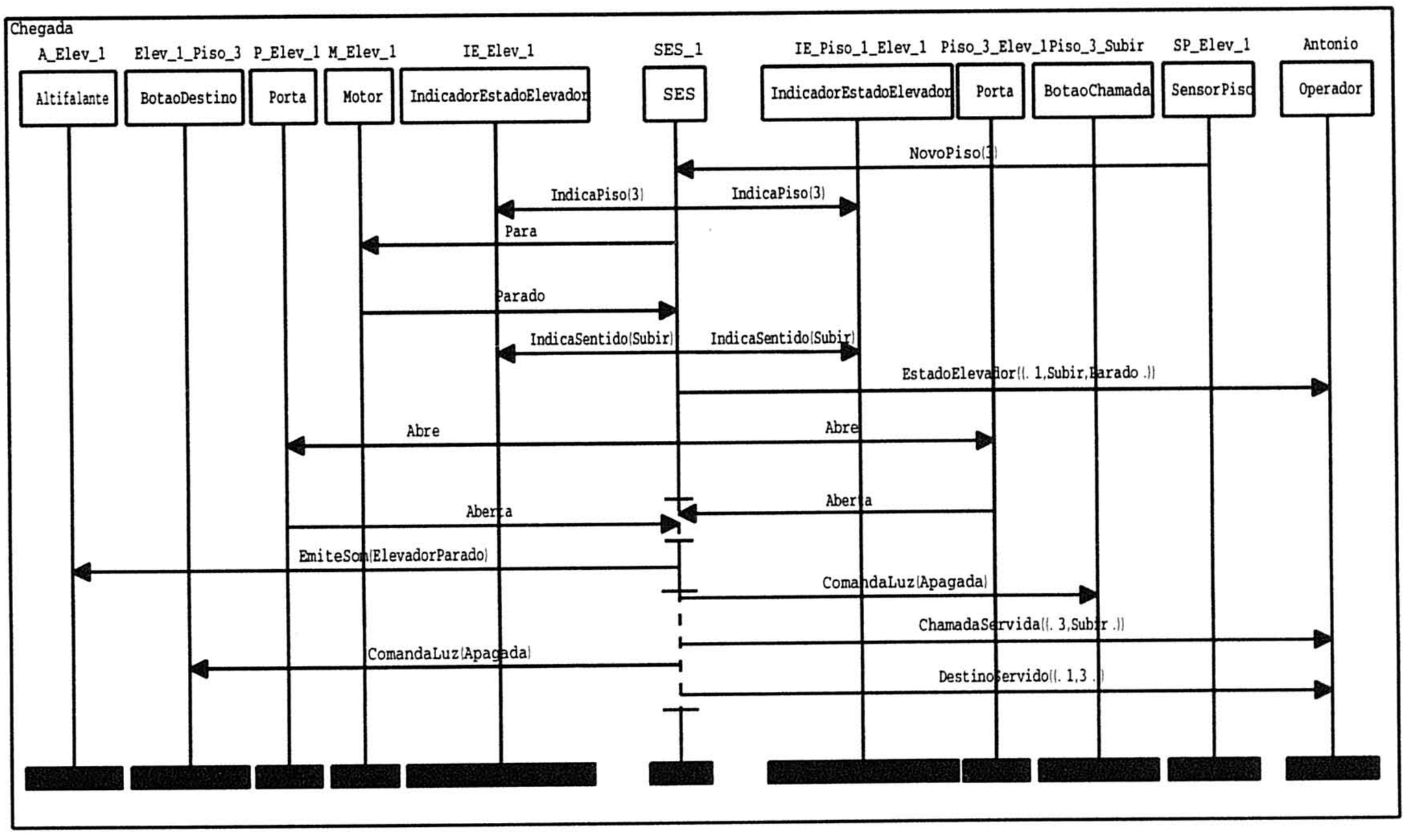


Figura 8.17: MSC da Chegada de um Elevador a um Piso.

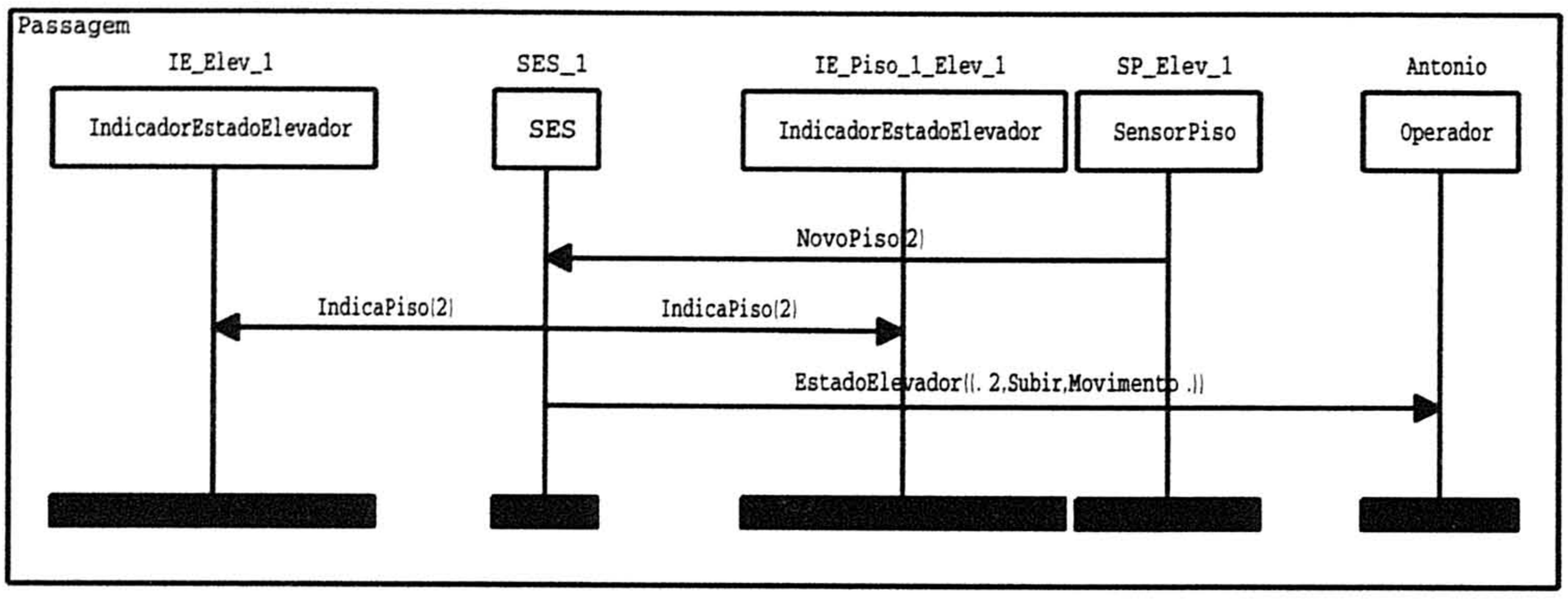


Figura 8.18: MSC da Passagem por um Piso.

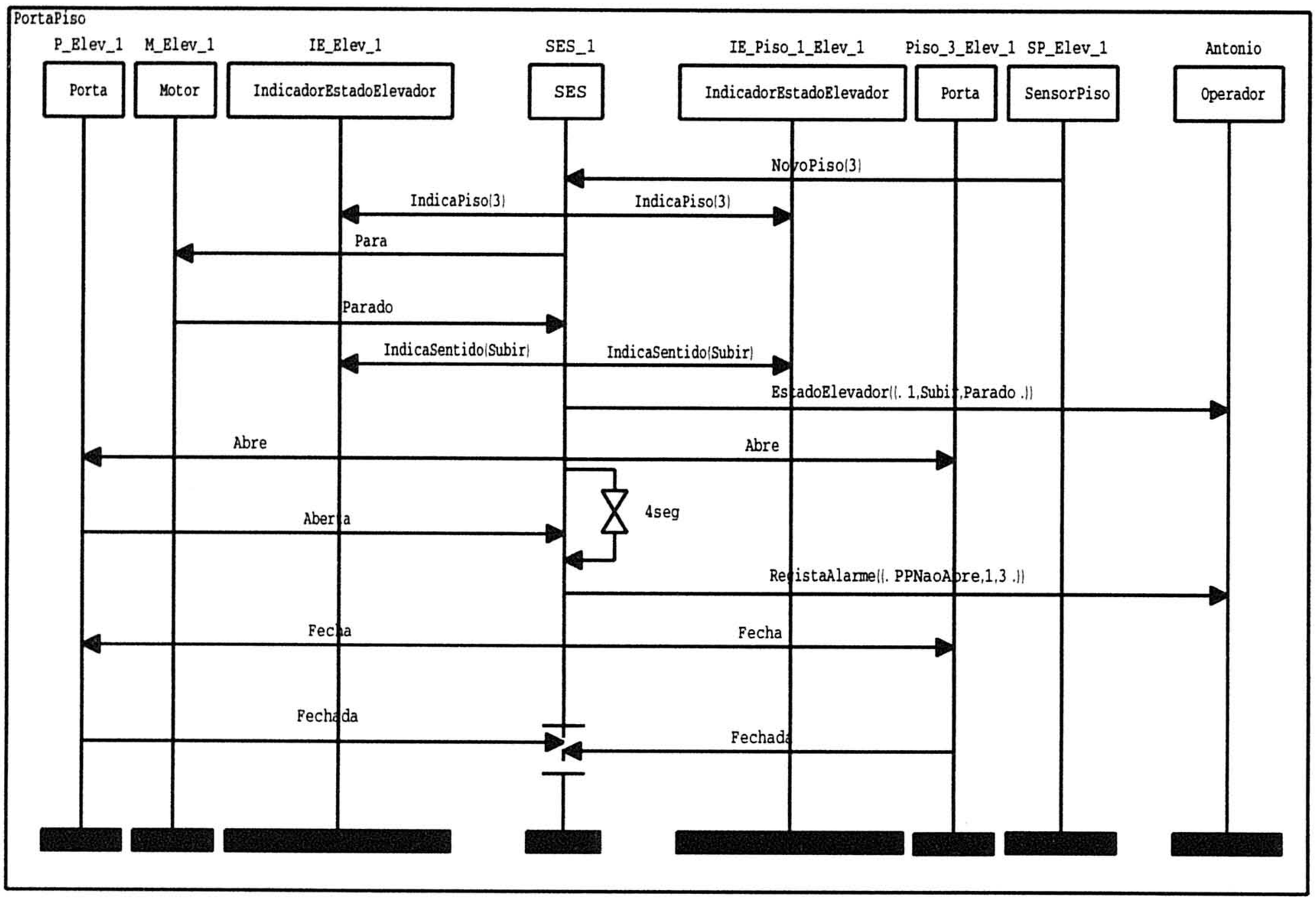


Figura 8.19: MSC da Porta do Piso Não Abre.

8.6 Coerência Entre o Modelo de Objectos e o Modelo de Casos de Utilização

O modelo dinâmico tem que ser verificado relativamente ao modelo de objectos e adicionar a este último a informação que falte, de forma a cumprir as seguintes regras:

- Os cenários identificados na hierarquia podem corresponder a classes ou operações de classes. No entanto, também podem ser uma combinação de operações ou uma parte de uma operação (isto aplica-se especialmente aos cenários terminais).

- As barras verticais correspondem a instâncias do sistema e a instâncias de actores modelados como classes dinâmicas no modelo de objectos.
- As mensagens MSC podem corresponder a operações das classes ou a informação retornada por elas. As operações devem ser declaradas, no primeiro caso, na classe que recebe a mensagem e, no segundo caso, na classe que envia a mensagem (ou numa classe acessível a partir dela através das associações). No entanto, isto só é aplicável se as classes tiverem um papel de importância semelhante. Se a classe destinatária da primeira mensagem fornece um serviço de baixo nível comparativamente à classe remetente, esta última pode ter a operação correspondente. Deste modo, essa operação também pode ser acedida por outras classes sem terem que comunicar directamente com a classe de baixo nível. Contudo, não há qualquer relação automática entre mensagens e operações. Uma mensagem MSC pode corresponder a um conjunto de operações e associações e, reciprocamente, uma operação pode corresponder a uma sequência de mensagens.
- Os valores das mensagens MSC podem corresponder a atributos de classes ou a resultados de operações. Os atributos têm que ser declarados na operação correspondente à mensagem, ou noutra classe acessível pelas associações.
- As interacções entre duas instâncias têm que ser suportadas, directa ou indirectamente, por associações de classes.

8.7 Relações com as outras Actividades de Desenvolvimento

8.7.1 Transição para o Projecto da Arquitectura

A actividade de análise de requisitos produz um modelo de objectos OMT e um modelo MSC. Na fase de projecto da arquitectura, os diagramas de classe serão explorados para identificar objectos concorrentes que serão elementos da arquitectura do sistema. Os MSCs servirão para identificar as sequências de comunicação trocadas por esses objectos. Algumas das sequências encontradas nos MSCs serão modeladas como canais, sinais, ADTs ou até objectos.

8.7.2 Transição para o Projecto Detalhado

O projecto detalhado de objectos passivos começa tanto a partir da arquitectura SDL como do modelo de objectos da análise. Alguns objectos passivos são identificados logo na fase de análise e permanecem no decurso do processo de desenvolvimento. Por isso, o modelo de objectos da análise será explorado mais tarde, durante a especificação em OMT.

8.7.3 Transição para o Projecto de Testes

O projecto dos testes começará pela utilização do modelo dinâmico elaborado na análise de requisitos. Através da pormenorização do modelo de casos de utilização de acordo com a arquitectura SDL serão produzidos testes iniciais de integração.

9 Projecto da Arquitectura

9.1 Introdução

O primeiro objectivo dos projectistas do sistema é definir a arquitectura da aplicação pela identificação dos objectos concorrentes. Esta tarefa baseia-se no modelo de objectos e modelo dinâmico produzidos na fase de análise de requisitos. As características do ambiente real no qual o sistema vai operar têm que ser consideradas. No entanto, para preservar a portabilidade (*portability*) da aplicação, o projecto deve ser independente da distribuição real dos objectos.

A arquitectura é construída pela combinação de uma abordagem de-cima-para-baixo (*top-down*), ou **processo de refinamento**, e de uma abordagem de-baixo-para-cima (*bottom-up*), ou **processo de composição**. O refinamento permite partir um problema noutros menores e mais facilmente resolúveis. A composição torna possível reutilizar componentes já existentes. Os elementos de *software* identificados neste estágio correspondem a objectos concorrentes.

A comunicação entre objectos concorrentes tem que ser definida de forma clara em termos de canais e mensagens transmitidas através desses canais.

As pessoas envolvidas nesta actividade são os projectistas do sistema. É necessária perícia técnica tanto no domínio da aplicação, de modo a considerar arquitecturas reutilizáveis, como em técnicas de desenvolvimento de *software*, para permitir seleccionar as soluções mais eficientes.

O projecto da arquitectura é realizada na notação SDL. Mais concretamente, os diagramas de hierarquia e de interligação do sistema. Esta actividade está intimamente relacionada com o projecto de testes, pelo que os MSCs servem para complementar a SDL. Porém, a notação MSC é estritamente dedicada ao projecto de testes e será discutida apenas no capítulo 11.

9.2 Projecto da Arquitectura em SDL

As referências feitas nesta secção a entidades que fazem parte da arquitectura do sistema, referem-se às figuras apresentadas na secção 9.3.

9.2.1 Descrição do Primeiro Nível da Hierarquia

A actividade de projecto foca-se sobre a aplicação propriamente dita. Os actores modelados na fase de análise são agora considerados unicamente através das suas *interfaces* com o sistema, nomeadamente, o conteúdo da informação que trocam e os meios de comunicação que utilizam.

Cada actor é geralmente modelado sob a forma de um ou mais canais; vários canais podem ser criados para um actor se for possível dividir a sequência de comunicação em subsequências consistentes.

No caso do `SES`, foram identificados vários actores na fase de análise: `BotaoChamada`, `BotaoDestino`, `BotaoPorta`, `SensorPiso`, `SensorTensao`, `SensorObstrucao` e `Operador`. No primeiro nível da arquitectura, fez-se corresponder a cada um o respectivo canal.

O primeiro refinamento da aplicação não é guiado pelo modelo de objectos da análise, mas antes, por considerações técnicas como reutilização de arquitecturas e componentes existentes, encapsulamento de elementos de *hardware*, *etc.* Os elementos de alto-nível identificados têm que ter propósitos bem definidos e as suas interacções devem ser limitadas de forma a proporcionar uma arquitectura modular.

No SES, foram identificados três objectos de alto-nível:

- bloco `Central`, que inclui as funcionalidades de supervisão e comando;
- bloco `Pisos`, que faz o processamento associado a cada piso;
- bloco `Elevadores`, que controla os elevadores e os respectivos dispositivos.

9.2.2 Refinamento e Composição

O refinamento consiste em dividir os objectos de alto-nível em objectos cada vez mais detalhados. De modo a descobrir sub-objectos, os projectistas podem também explorar as agregações e associações entre as classes do sistema definidas no modelo de objectos da fase de análise. O processo de refinamento está completo logo que os objectos concorrentes identificados no último nível possam ser facilmente transcritos para linguagens de programação e implementados na plataforma real. Cada objecto não terminal é descrito sob a forma de um diagrama de interligação SDL.

O projecto da arquitectura tem que ser realizado tendo sempre a preocupação de preservar a facilidade de manutenção e a modularidade, isto é, os objectos identificados devem ter uma função específica e ter interacções reduzidas com os outros objectos. Este objectivo torna-se mais fácil de cumprir pela utilização de SDL, já que esta notação inviabiliza a utilização de campos de dados comuns e os objectos comunicam através de trocas de sinais. Por exemplo, o comportamento de um objecto concorrente não pode ser dependente do estado de outros.

São também introduzidos novos objectos de acordo com as soluções técnicas escolhidas para a aplicação, relativamente a comunicações, persistência de dados, ou *interface* com o utilizador. Geralmente, estes novos objectos servem para encapsular, ou servir de *interface* a recursos externos, como protocolos de comunicação, bases de dados relacionais ou OO, *interface*s gráficas, *etc.* No caso em estudo, nenhum objecto concorrente é introduzido com este propósito, de forma a manter a portabilidade da aplicação.

Os objectos terminais correspondem a processos SDL identificados pelas suas *interfaces* de comunicação e pelo número máximo de instâncias a serem executadas em paralelo.

No SES, a arquitectura é constituída pelos seguintes componentes:

- bloco `Central`, que contém o processo `Supervisor`, que supervisiona e comanda os elevadores, e o processo `Gnomo` que escala as chamadas consoante o estado de cada elevador;
- bloco `Pisos`, que contém os processos `PainelChamada`, que recebem a chamadas de cada piso, e `PortaPiso`, que controla a porta do piso;
- bloco `Elevadores`, que é composto pelos processos `Elevador`, que controla o elevador, `PainelDestino`, que recebe os destinos, `PortaElevador`, que controla a porta do elevador, e `MonitorCabo`, que supervisiona o cabo de tracção do elevador.

9.2.3 Descrição da Comunicação

A comunicação entre objectos é descrita em cada diagrama de interligação sob a forma de:

- canais entre os **objectos irmãos**;
- canais entre os **objectos filho** e os canais do **objecto pai**;
- sinais trocados entre objectos irmãos, e entre estes e o respectivo objecto pai.

Os sinais são inicialmente extraídos dos MSCs, nos quais foram identificadas sequências de mensagens ao nível do sistema. Um conjunto de sinais é declarado no diagrama de interligação de nível mais alto: `Premido`, `NovoPiso`, `TensaoAlterada`, *etc.*

Sendo identificados em MSCs de análise, os sinais normalmente correspondem a propriedades das classes identificadas nos diagramas de classes da análise. Por exemplo, os anteriores sinais correspondem respectivamente às operações `Premido` da classe `Botao`, `NovoPiso` da classe `Elevador` e `TensaoAlterada` de `MonitorCabo`. Os parâmetros dos sinais devem ser compatíveis com as assinaturas das operações.

No entanto, a relação entre os sinais SDL e as mensagens MSC não é automática. Algumas mensagens MSC podem ser transformadas em objectos passivos (descritos na secção 9.2.4), em conjuntos de sinais, ou em objectos concorrentes. Por exemplo, considere-se um protocolo de alto-nível em que cada mensagem enviada é dividida em vários pacotes e depois codificada. A recepção de cada mensagem tem que ser confirmada; se o objecto destinatário não confirma a recepção antes do fim de um determinado tempo, os respectivos pacotes são reenviados. Tal protocolo conduzirá à introdução de novos blocos para lidar com a comunicação, e uma simples mensagem de análise seria transformada numa sequência de sinais de projecto. Por outro lado, alguns sinais SDL podem corresponder a elementos de análise que não sejam mensagens MSC, *e.g.* classes.

No diagrama de interligação do topo da hierarquia, podem ainda ser introduzidos novos sinais trocados com o ambiente do sistema para implementar restrições ambientais (*e.g.* o estado de uma linha de comunicações).

No decorrer do refinamento da arquitectura, a comunicação tem que ser detalhada em cada diagrama de interligação:

- Os sinais trocados pelos objectos pais tornam-se sinais trocados entre os filhos e o respectivo ambiente. Os canais localizados em ambos os níveis hierárquicos têm que ser ligados uns aos outros. No diagrama de interligação do tipo de bloco `Piso`, o canal `Escalamento` transporta os sinais `EscalaChamada`, `ConEscalaChamada` e `ChamadaServida`, entre o processo `PainelChamada` e o canal `Chamadas` do seu objecto pai (o tipo de bloco `Piso`) através da porta `GE`.
- São introduzidos sinais internos para suportar interacções entre objectos irmãos, como o sinal `ConServeDestino` enviado pelo processo `PainelDestino` ao processo `Elevador`. Estes sinais internos não são visíveis para o exterior do objecto pai.

9.2.4 Declaração de Objectos Passivos

Como os sinais transportam informação nos seus parâmetros, estes podem ter tipos complexos. Por conseguinte, os tipos dos dados são definidos em complemento às declarações dos sinais. Estes tipos de dados podem ser declarados ao mesmo nível que os sinais associados. Os tipos de dados definem-se por intermédio de ADTs. Alguns destes ADTs são construídos a partir de tipos básicos, como `TipoSentido` e `TipoSom`, que são listas de literais. Outros correspondem a classes identificadas no modelo de objectos resultante da análise: `IndicadorEstadoElevador` é um tipo novo associado com a classe `IndicadorEstadoElevador`.

Durante o refinamento da arquitectura e da comunicação, são introduzidos novos ADTs associados ao processamento dos sinais internos. Estes correspondem a novos objectos passivos não identificados no modelo de objectos da análise, que resultam de considerações de implementação. Estes objectos passivos irão enriquecer o modelo de objectos na fase de projecto detalhado.

A comunicação entre o sistema e o ambiente em SDL é geralmente descrita através de sinais. No entanto, isso também pode ser feito através de operadores ADT. Por exemplo, uma interacção com o utilizador pode ser implementada por um operador, como por exemplo, apresentar uma mensagem num ecrã, a aquisição de um valor do teclado, uma acção sobre um equipamento em *interface* com o utilizador. Este tipo de implementação só é totalmente descrita na fase de projecto detalhado, já que essas interacções com o utilizador fazem parte das transições dos processos. Isto significa que no projecto de uma arquitectura, a comunicação com o ambiente não está completamente especificada, e que partes dessa comunicação (as que correspondem aos operadores ADTs) serão projectadas mais tarde.

Os objectos passivos devem ser especificados num nível da hierarquia tal que, não seja alto demais para facilitar a reutilização, nem baixo demais para permitir evoluções. Se os objectos passivos forem

introduzidos num nível alto da hierarquia de um sistema, os objectos concorrentes dos níveis inferiores não podem ser reutilizados porque não incluem os seus próprios recursos. Por outro lado, se os objectos passivos são declarados num nível muito baixo, uma pequena restruturação dos objectos concorrentes afectará muito significativamente a localização da declaração dos objectos passivos na hierarquia.

9.2.5 Portabilidade da Arquitectura Concebida

A linguagem SDL fornece aos projectistas um serviço de comunicação que torna a arquitectura do sistema independente da distribuição real dos objectos na plataforma. Graças a isso, não há necessidade de fazer assunções quanto à proximidade dos objectos comunicantes. Mais ainda, os canais de comunicação são dinâmicos, eles são criados e apagados de acordo com a instanciação e término dos processos.

Por exemplo, a arquitectura do SES não depende do número de elevadores do edifício (`NE`). Este número é um parâmetro da arquitectura que define o número de instâncias activas do tipo de bloco `Elevador`, (neste caso, esse parâmetro é uma constante, mas poderia ser uma variável externa definida no arranque do sistema). Mais genericamente, se o sistema incluir uma rede de nós, a sua arquitectura SDL é independente da configuração real da rede.

A correspondência entre a arquitectura lógica SDL e a configuração da plataforma real é definida na fase de implementação (*targeting*).

9.3 Projecto da Arquitectura do SES

O primeiro passo de refinamento do SES requer perícia no domínio do sistema. Tal como a Figura 9.1 documenta, primeiro são identificados três blocos, cada um deles dedicado a um aspecto específico do SES. Os canais e sinais são criados de acordo com o modelo de análise (objecto e dinâmico):

- o bloco `Central` faz o interface com o utilizador, supervisiona, os elevadores e inclui o processo `Gnomo` responsável pela inteligência empregue na afectação das chamadas aos elevadores;
- o tipo de bloco `Piso` recebe e transfere as chamadas, controla as luzes dos botões de chamada e as portas de cada piso;
- o tipo de bloco `Elevadores` recebe e trata os destinos, controla os elevadores e respectivas partes.

Ainda relativo ao primeiro nível da hierarquia SDL tem-se declarações das constantes (Figura 9.2), dos tipos (Figura 9.3) e dos sinais (Figura 9.4).

Note-se que tanto o bloco `Pisos`, como o bloco `Elevadores`, definem instâncias de tipos de bloco, respectivamente, `Piso` e `Elevadores`.

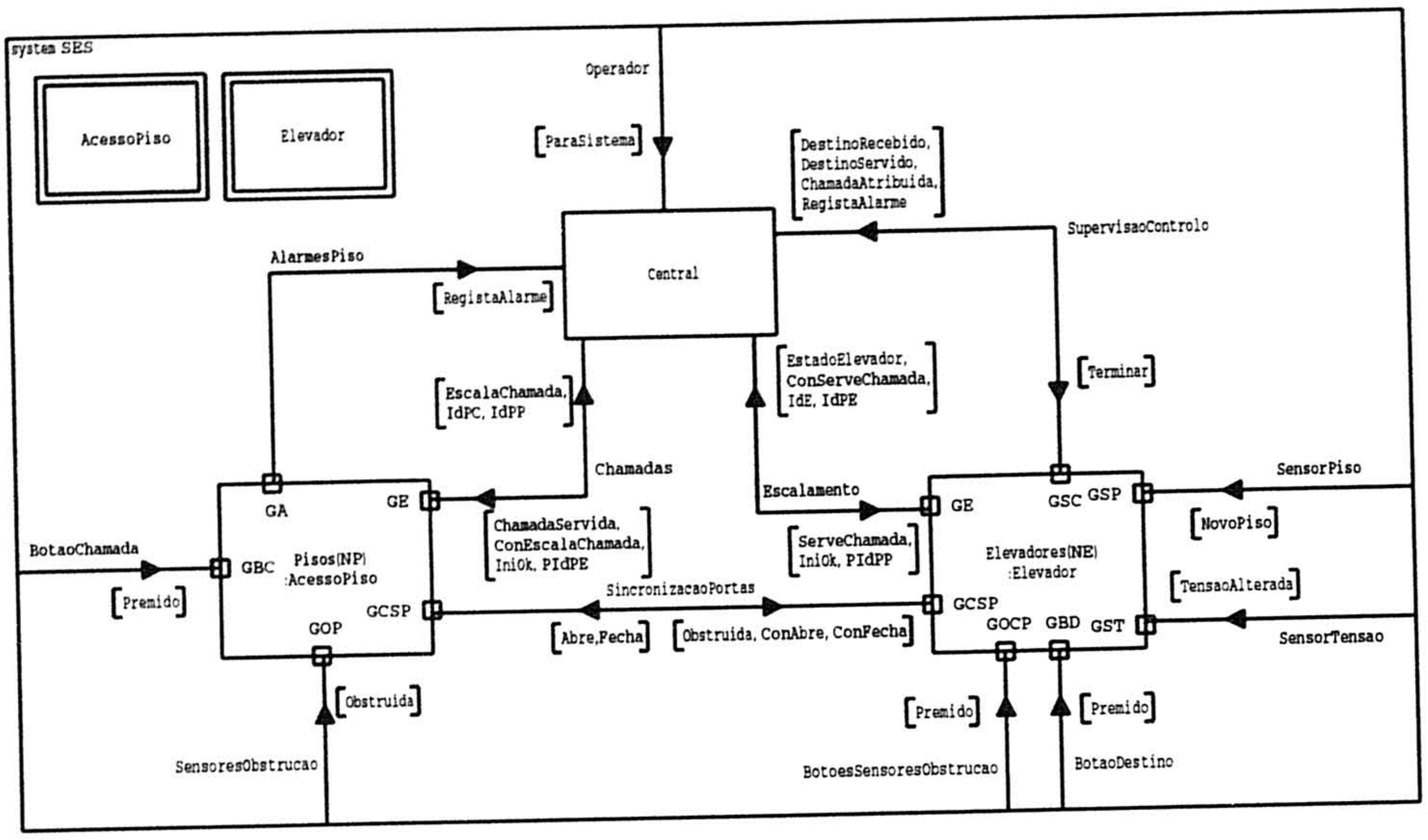


Figura 9.1: Diagrama de Interligação do SES.

```
system SES
/* DEFINICAO DE SINONIMOS */
/*************************************/

SYNONYM NE NATURAL = 2;
SYNONYM NP NATURAL = 4;

SYNONYM Nenhum NATURAL = 0;

/* No ObjectGEODE a escala temporal por defeito e 0.1 segundos */
SYNONYM DecSeg NATURAL = 1;
SYNONYM Seg NATURAL = 10;

SYNONYM TempoActuacaoPorta DURATION = 3*Seg;
```

Figura 9.2: Declaração de Constantes ao Nível do SES.

```
system SES
/* DEFINICAO DE TIPOS */
/*************************************/

SYNTYPE TipoElevador = NATURAL
 CONSTANTS 0:NE
ENDSYNTYPE;

SYNTYPE TipoPiso = NATURAL
 CONSTANTS 0:NP
ENDSYNTYPE;

NEWTYPE TipoSentido
 LITERALS Nenhum, Subir, Descer;
 OPERATORS
   Iterador : TipoSentido -> INTEGER;
   SI : TipoSentido -> TipoSentido;
 AXIOMS
   Iterador (Subir) == 1;
   Iterador (Descer) == -1;
/*   Iterador (Nenhum) == ERROR!;     nao suportao pelo OG */
   SI (Subir) == Descer;
   SI (Descer) == Subir;
/*   SI (Nenhum) == ERROR!;     nao suportado pelo OG */
ENDNEWTYPE;

NEWTYPE TipoAlarme
 LITERALS ErroIniPC, ErroIniPP, ErroIniE, ErroIniPE,
   PPNaoAbre, PPNaoFecha, PENaoAbre, PENaoFecha,
   PPNaoConFechada, PPNaoConAberta, RupturaCabo;
 OPERATORS
   Descricao: TipoAlarme -> CHARSTRING;
 AXIOMS
   Descricao (ErroIniPC) == 'Painel de Chamada com Numero de Piso Repetido';
   Descricao (ErroIniPP) == 'Porta do Piso com Numero de Piso Repetido';
   Descricao (ErroIniE) == 'Elevador com Numero de Elevador Repetido';
   Descricao (ErroIniPE) == 'Porta do Elevador com Numero de Elevador Repetido';
   Descricao (PPNaoAbre) == 'Porta do Piso Nao Abre';
   Descricao (PPNaoFecha) == 'Porta do Piso Nao Fecha';
   Descricao (PENaoAbre) == 'Porta do Elevador Nao Abre';
   Descricao (PENaoFecha) == 'Porta do Elevador Nao Fecha';
   Descricao (PPNaoConAberta) == 'Porta do Piso Nao Confirma Abertura';
   Descricao (PPNaoConFechada) == 'Porta do Piso Nao Confirma Fecho';
   Descricao(RupturaCabo) == 'Ruptura do Cabo do Elevador';
ENDNEWTYPE;

NEWTYPE Alarme
 STRUCT
   Codigo TipoAlarme;
   Elevador TipoElevador;
   Piso TipoPiso;
ENDNEWTYPE;

NEWTYPE TipoTensao
 LITERALS Correcta, Excessiva, Ruptura;
ENDNEWTYPE;

NEWTYPE TipoLuz
 LITERALS Apagada, Acesa;
ENDNEWTYPE;

NEWTYPE TipoSom
 LITERALS Nenhum, ElevadorParado, CargaExcessiva, Alarme;
ENDNEWTYPE;

NEWTYPE TipoComandoPorta
 LITERALS Abrir, Fechar;
ENDNEWTYPE;

NEWTYPE Chamada
 STRUCT
   Piso TipoPiso;
   Sentido TipoSentido;
 DEFAULT
 (. Nenhum, Nenhum .);
ENDNEWTYPE;

NEWTYPE TipoEstadoElevador
 LITERALS ForaServico, Livre, Movimento, Parado;
ENDNEWTYPE;

NEWTYPE DadosElevador
 STRUCT
   Piso TipoPiso;
   Sentido TipoSentido;
   Estado TipoEstadoElevador;
 DEFAULT
   (. Nenhum, Nenhum, ForaServico .);
ENDNEWTYPE ;

NEWTYPE TipoVoid
 LITERALS Nada;
ENDNEWTYPE;
```

Figura 9.3: Declaração de Tipos ao Nível do SES.

```
system SES
  /* DEFINICAO DE SINAIS */
  /*****************************/
  SIGNAL
  /* Do Operador para a Central */
   ParaSistema,
  /* Dos Botoes para o SES */
   Premido,
  /* Dos Sensores de Obstrucao para o Piso */
   Obstruida,
  /* Do Sensor de Tensao para o Elevador */
   TensaoAlterada,
  /* Do Sensor de Piso para o Elevador */
   NovoPiso,
  /* Entre as Portas */
   Abre,Fecha, ConAbre, ConFecha,
  /* Da Central para o Elevador */
   Terminar,
  ServeChamada(Chamada), ConServeChamada(Chamada),
  /* Do Elevador para a Central */
   EstadoElevador(DadosElevador),
   DestinoRecebido(TipoPiso), DestinoServido(TipoPiso),ChamadaAtribuida(Chamada),
  /* Entre Piso e Central */
   ChamadaServida(Chamada), EscalaChamada(Chamada),
   RegistaAlarme(Alarme),
  ConEscalaChamada(Chamada),

  /* Sinais de Inicializacao */
   IdPC(TipoPiso), IdPP(TipoPiso,TipoElevador), IdE(TipoElevador), IdPE(TipoElevador),
   PIdPE(PID), PIdPP(TipoPiso,PID),
   IniOk;
```

Figura 9.4: Declaração de Sinais ao Nível do SES.

Os blocos do primeiro nível são depois refinados em processos correspondentes ao nível terminal. Para sistemas complexos, o processo de refinamento é mais demorado e requer mais níveis intermédios. Neste caso, bastaram dois níveis para conseguir uma arquitectura simples e completa. Quando é necessário, em cada nível de refinamento são criados novos sinais e tipos para suportar as sequências de informação trocadas entre os objectos que compõem esse nível. Para isso, utilizam-se as declarações PR.

A Figura 9.5 mostra os processos do bloco `Central`, estando as declarações dos sinais trocados internamente entre os processos que o constituem na Figura 9.6.

Os processos localizados no tipo de bloco `Piso` são ilustrados na Figura 9.7.

A Figura 9.8 detalha o tipo de bloco `Elevador` e a Figura 9.9 mostra os respectivos sinais internos.

A Figura 9.10 ilustra a hierarquia final do SES na forma de blocos para os níveis não terminais e processos para os níveis terminais. O primeiro índice entre parêntesis de um processo corresponde ao número de instâncias iniciais (no arranque), e o segundo define o número de instâncias máximo desse processo. No caso dos blocos, o índice entre parêntesis designa o número de instâncias do tipo do bloco.

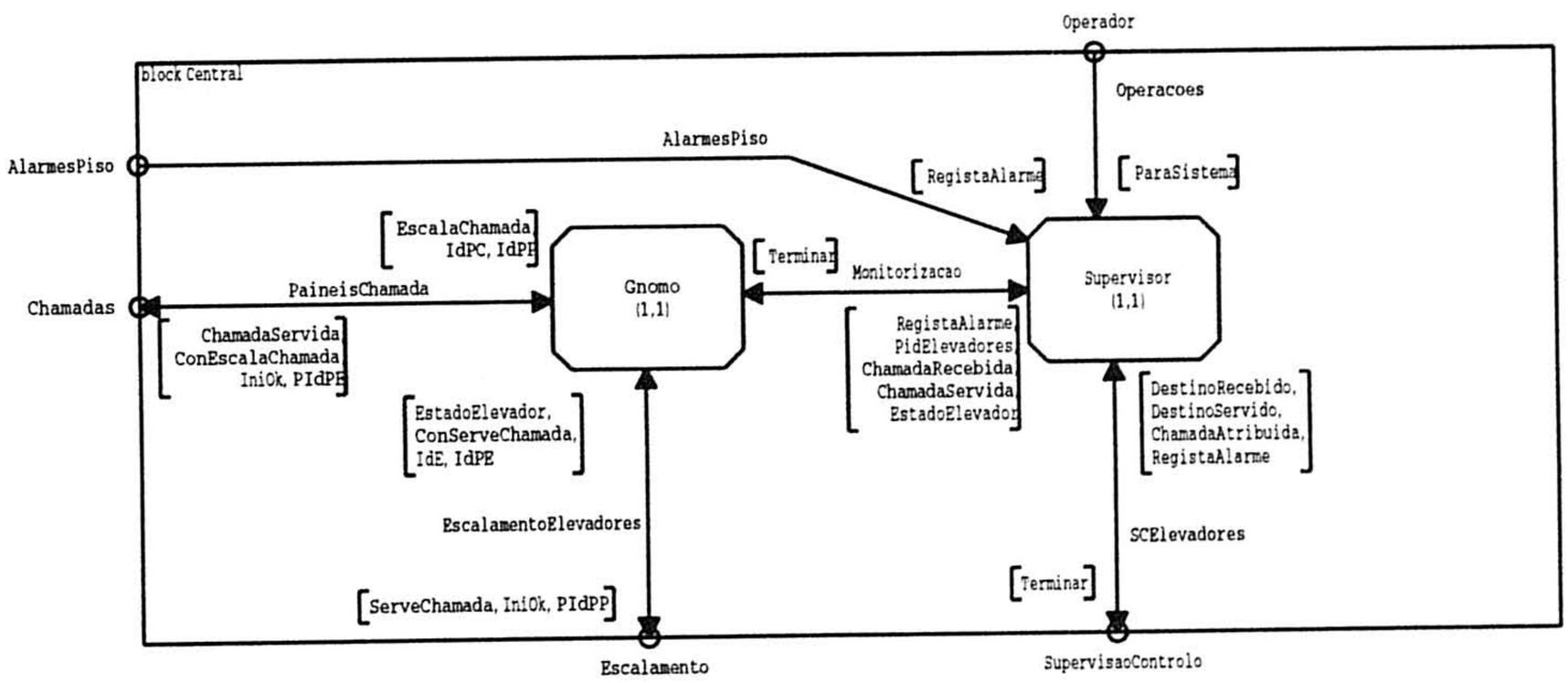


Figura 9.5: Diagrama de Interligação do Bloco Central.

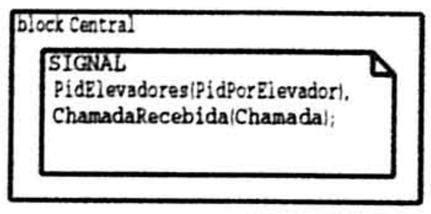


Figura 9.6: Declaração PR dos Sinais do Bloco Central.

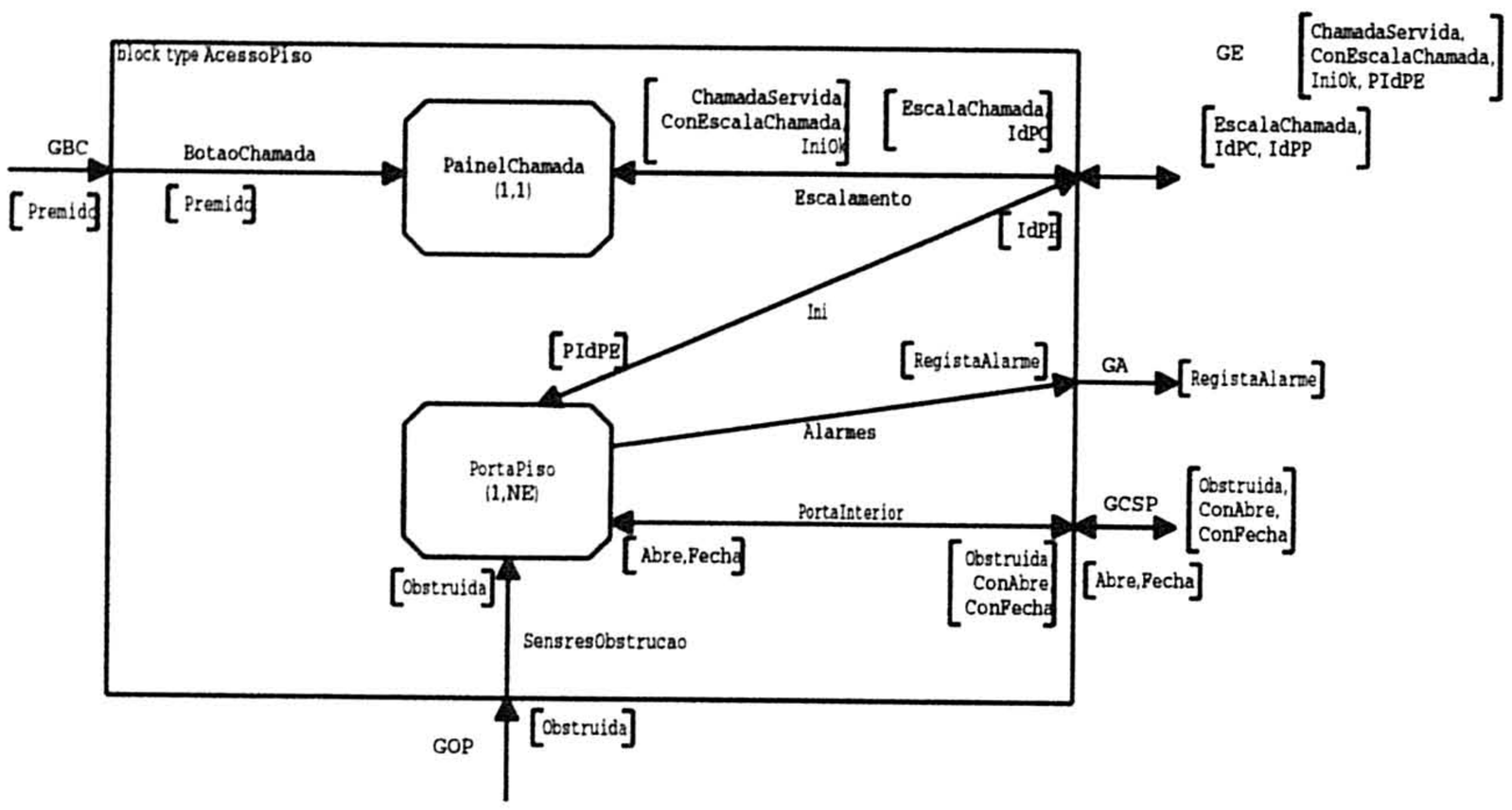


Figura 9.7: Diagrama de Interligação do Tipo de Bloco Piso.

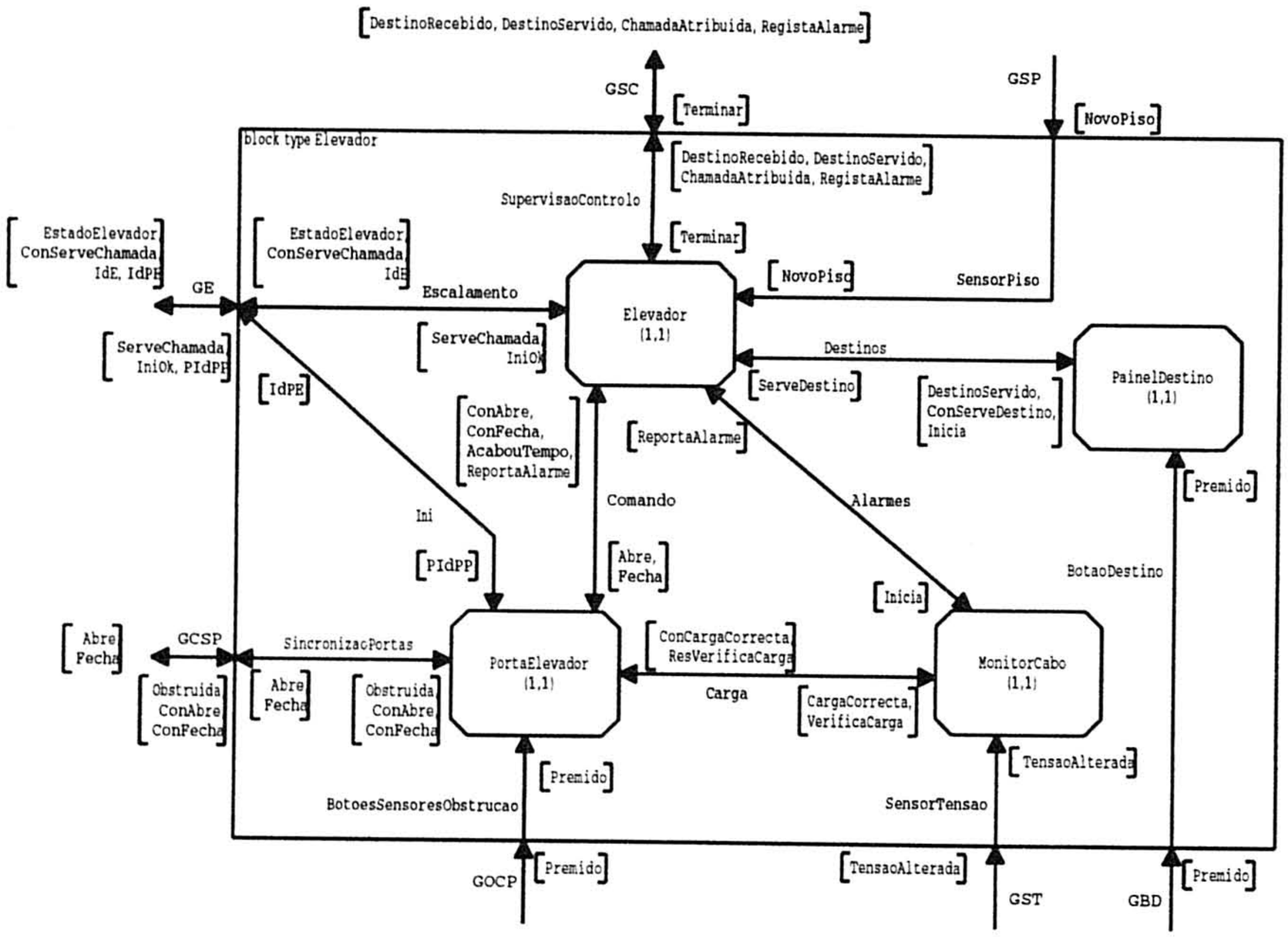


Figura 9.8: Diagrama de Interligação do Tipo de Bloco Elevador.

```
block type Elevador

/* Declaracoes de Sinais */
/*************************/

SIGNAL
 /* Da Porta para o Elevador */
 AcabouTempo,
 /* Entre a Porta e o Monitor do Cabo */
 CargaCorrecta, ConCargaCorrecta,
 VerificaCarga, ResVerificaCarga(TipoTensao),
 /* Entre o Elevador e o Painel de Destino */
 /* DestinoServido(TipoPiso),*/
 ConServeDestino(TipoPiso), Inicia,
 ServeDestino(TipoPiso),
 /* Entre o Elevador e o Monitor Cabo */
 ReportaAlarme(TipoAlarme, TipoPiso);

/* DEFINICAO DE SINONIMOS */
/**************************************/

SYNONYM TempoPortaAberta DURATION = 5*Seg;
SYNONYM TempoConfirmacaoPorta DURATION = 6*Seg;
```

Figura 9.9: Declarações PR dos Sinais do Tipo de Bloco Elevadores.

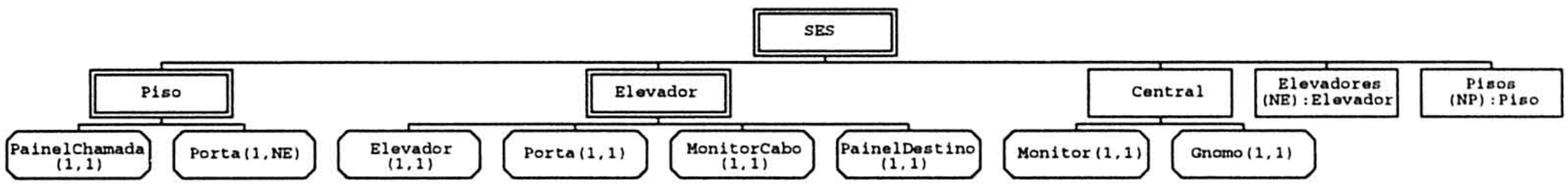


Figura 9.10: Hierarquia SDL do SES.

9.4 Relações com as outras Actividades de Desenvolvimento

9.4.1 Transição para a Fase de Projecto Detalhado

A arquitectura SDL resultante desta actividade inicial de projecto é a base de trabalho para o estágio seguinte de projecto detalhado. Todos os objectos concorrentes terminais (os processos SDL) estão identificados e serão detalhados por meio de diagramas estado-transição. Os objectos passivos identificados na análise e projecto da arquitectura serão detalhados através de diagramas de classes.

9.4.2 Relação com a Fase de Projecto de Testes

Tal como anteriormente referido, é aconselhável realizar o projecto da arquitectura em simultâneo com o projecto de testes. Os casos de utilização constituem um ponto de partida para a definição da comunicação no primeiro nível da arquitectura. O diagrama de interligação só fica completo quando suporta todas as sequências de interacção definidas nos casos de utilização (com a restrição de algumas interacções serem implementadas por meio de operadores, e por isso não podem ser representadas no diagrama de interligação). Quando o primeiro diagrama de interligação está terminado, os casos de utilização podem ser refinados de acordo com os objectos concorrentes nele incluídos. São produzidos novos MSCs, que podem ser considerados como testes de integração. Após o refinamento desses objectos através de diagramas de interligação de nível inferior, os testes de integração podem ser utilizados como requisitos a serem satisfeitos.

Deste modo, no final do projecto da arquitectura, os projectistas obterão um conjunto completo de especificações que descrevem a comunicação esperada ao nível do processo.

Resumindo, a comunicação num diagrama de interligação é expressa por MSCs descrevendo esse nível, e reciprocamente, o refinamento dos MSCs é guiado pelos objectos identificados em níveis de interligação inferiores.

10 Projecto Detalhado

10.1 Introdução

Nesta fase, os projectistas têm que descrever totalmente **como** é que o sistema se comporta para fornecer os serviços esperados: os objectos concorrentes e passivos são detalhados possibilitando, depois, uma fácil implementação.

O projecto detalhado, tal como projecto da arquitectura, é realizado através da combinação das abordagens de-cima-para-baixo e de-baixo-para-cima.

Os recursos externos a serem integrados, tais como primitivas gráficas, bases de dados ou equipamentos físicos, terão uma *interface* com o sistema especificado através da modelação das suas APIs como objectos. Esta tarefa é facilitada se as APIs já existirem na forma de classes OMT, construídas por exemplo através de engenharia inversa (*reverse engineering*).

Para melhorar a portabilidade da aplicação, o projecto detalhado deve ser independente da configuração da plataforma real (microprocessadores, microcontroladores, barramentos, rede, *etc.*).

Esta actividade é realizada pelos engenheiros de *software*, pois requer perícia técnica no desenvolvimento de *software*. Habilidades específicas podem ser utilizadas para resolver problemas específicos, como por exemplo, construir *interfaces* gráficas com o utilizador, integrar bases de dados relacionais, *interface* com a rede e protocolos existentes.

Nesta fase utilizam-se as notações SDL e OMT: os objectos concorrentes são descritos por diagramas de processo SDL e os objectos passivos são por classes OMT. Depois, os objectos passivos OMT são combinados com os objectos concorrentes através de ADTs a serem declarados na arquitectura SDL.

10.2 Projecto Detalhado dos Objectos Concorrentes

As referências feitas nesta secção a entidades da especificação em SDL do `SES`, referem-se às figuras apresentadas em 10.3.

Os objectos concorrentes do último nível da hierarquia SDL têm que ser descritos por diagramas de processo.

10.2.1 Identificação de Estados

A primeira tarefa a fazer é identificar para cada processo todos os estados possíveis. Isto faz-se através da observação dos sinais que são esperados em cada estado e de que forma a resposta do processo a um determinado sinal depende dos seus estados anteriores. A identificação de estados é facilitada pela exploração dos testes produzidos na actividade de projecto de testes.

Por exemplo, o processo `PainelChamada` tem somente um único estado `Ciclo` porque os sinais de entrada não têm uma ordem lógica e a resposta a dar-lhes é independente de qualquer estado.

10.2.2 Descrição das Acções das Transições

Cada processo deve identificar uma transição inicial, a ser disparada quando o processo é instanciado.

Depois, todas as transições têm que ser pormenorizadas por items gráficos de SDL, como recepção de sinais, envio de sinais, tarefas de atribuição (que incluem as chamadas dos operadores ADTs), decisões, activação, desactivação e expiração de temporizadores, invocação de procedimentos, criação de processos, término de processos, *etc.* Introduzem-se novas constantes, tipos e variáveis, bem como operadores e procedimentos. Consequentemente, são criados novos objectos passivos para implementar plenamente o comportamento esperado. As variáveis são locais aos processos, isto é, não podem ser partilhadas por várias instâncias de processos. No entanto, os mecanismos específicos de SDL `Revealed/View` e `Import/Export` permitem que os processos consigam obter valores sem ser (directamente[5]) através de sinais.

O desempenho esperado tem que ser tido em conta, por exemplo, o tempo requerido para processar um sinal, de modo a evitar exceder o tamanho da fila-de-espera. Os temporizadores também devem ser introduzidos na espera de sinais, de forma a evitar impasses (*deadlocks*). O projectista deve também ter presente que, quando um processo muda de estado a sua fila-de-espera de entrada pode não estar vazia, logo os sinais que a preenchem podem ser processados no estado seguinte. Mais ainda, se num determinado estado não houver transições para os sinais que estão em primeiro lugar na fila-de-espera, esses sinais serão perdidos (ignorados) quando for recebido o primeiro sinal que dispare uma transição. Para lidar com este problema existe o mecanismo SDL de guardar (*save*) sinais.

A tarefa de especificação dos processo é iterativa, em vez de sequencial. Alguns processos são detalhados, depois são estaticamente verificados em conjunto, principalmente para verificar a nomenclatura e o balanceamento da recepção/transmissão de sinais. Depois disso, mais processos são criados. Uma primeira versão da especificação é conseguida quando todas as transições são detalhadas e os processos estão correctos do ponto de vista semântico estático.

---

[5] Estes mecanismos são implementados pela máquina virtual SDL através de sinais implícitos.

10.2.3 Simulação do Projecto Detalhado

Como a notação SDL permite a simulação do modelo, pode verificar-se a semântica dinâmica da especificação detalhada dos objectos concorrentes através da simulação. Desta forma, as propriedades dinâmicas, como ausência/existência de impasses (*deadlocks*), sinais recebidos mas não esperados (*false inputs*), *etc.*, podem ser verificadas. Após a detecção de erros de projecto, é possível avaliar a conformidade com os requisitos. Produzem-se novas versões da especificação, que fica completa quando todos os erros detectados pela simulação são corrigidos. Os testes criados durante a actividade de projecto de testes podem ser utilizados para comparar o comportamento actual com o comportamento esperado.

Geralmente, o código fonte dos objectos passivos não está disponível nesta altura, o que torna unicamente possível verificar a coerência da invocação, *i.e.* a conformidade dos valores de facto com os parâmetros formais. Porém, a notação SDL permite expressar uma abstracção do comportamento esperado através de axiomas ou algoritmos. Estas facilidades são fundamentais para investigar de forma mais precisa a coerência do comportamento de facto dos objectos concorrentes e o comportamento esperado dos objectos passivos.

A coerência completa entre o comportamento dos objectos passivos e concorrentes só será verificada na implementação durante os testes de integração.

10.2.4 Explicação da Actividade

O processo `PortaPiso` é um bom exemplo para ilustrar a utilização deste método. Foram identificados quatro estados: `Abrindo`, `Aberta`, `Fechando` e `Fechada`. Inicialmente, a porta está `Fechada`. Pode, portanto, receber o sinal `Abre` (ordem para abrir). Quando recebe o sinal `Abre` vindo da `PortaElevador`, inicia a abertura, lança um temporizador para esperar pela confirmação da abertura e transita para o estado `Abrindo`. No estado `Abrindo`, pode receber a confirmação `Aberta` (este sinal é denominado sinal contínuo, ou seja, na realidade corresponde a uma condição), ou pode receber a expiração do temporizador. No primeiro caso, passa para o estado `Aberta`, enquanto no segundo envia um sinal de alarme ao supervisor e permanece no mesmo estado. No estado `Aberta`, pode receber o sinal para `Fechar` e quando o recebe, inicia o fecho e transita para o estado `Fechando`. Nesse estado, aguarda que termine o fecho passando para o estado `Fechada`, que seja `Obstruída` interrompendo o fecho e voltando a abrir, ou que falhe o fecho reportando um alarme. A Figura 10.1 é o diagrama de estados OMT do processo agora descrito. Esta actividade pode ser iniciada com diagramas de estado OMT, que se não utilizarem a concorrência interna de estados nem a hierarquia de estados, podem ser traduzidos em esqueletos de código SDL, que depois serão mais detalhados e completados com a inclusão de acções nas transições.

10.3 Projecto Detalhado do SES em SDL

Devemos concentrar-nos na descrição do problema no que se refere ao **comportamento** dos objectos. A atenção deve dirigir-se para **como** é que o objecto X comunica com o objecto Y. Facilmente se percebe que o problema dos elevadores é um tipo de sistema cujo modelo é constituído por um conjunto de objectos comunicantes assíncronos.

A seguir faz-se uma descrição de cada um dos processos. Obviamente, esta é uma explicação superficial com o objectivo dar uma ideia do que faz cada processo.

Dada a dimensão dos diagramas de processo SDL respeitantes ao projecto detalhado do SES, estes foram remetidos para o Anexo A com o objectivo de manter a organização do texto e a facilidade de leitura. A seguir apresentam-se somente alguns mais pequenos a título ilustrativo.

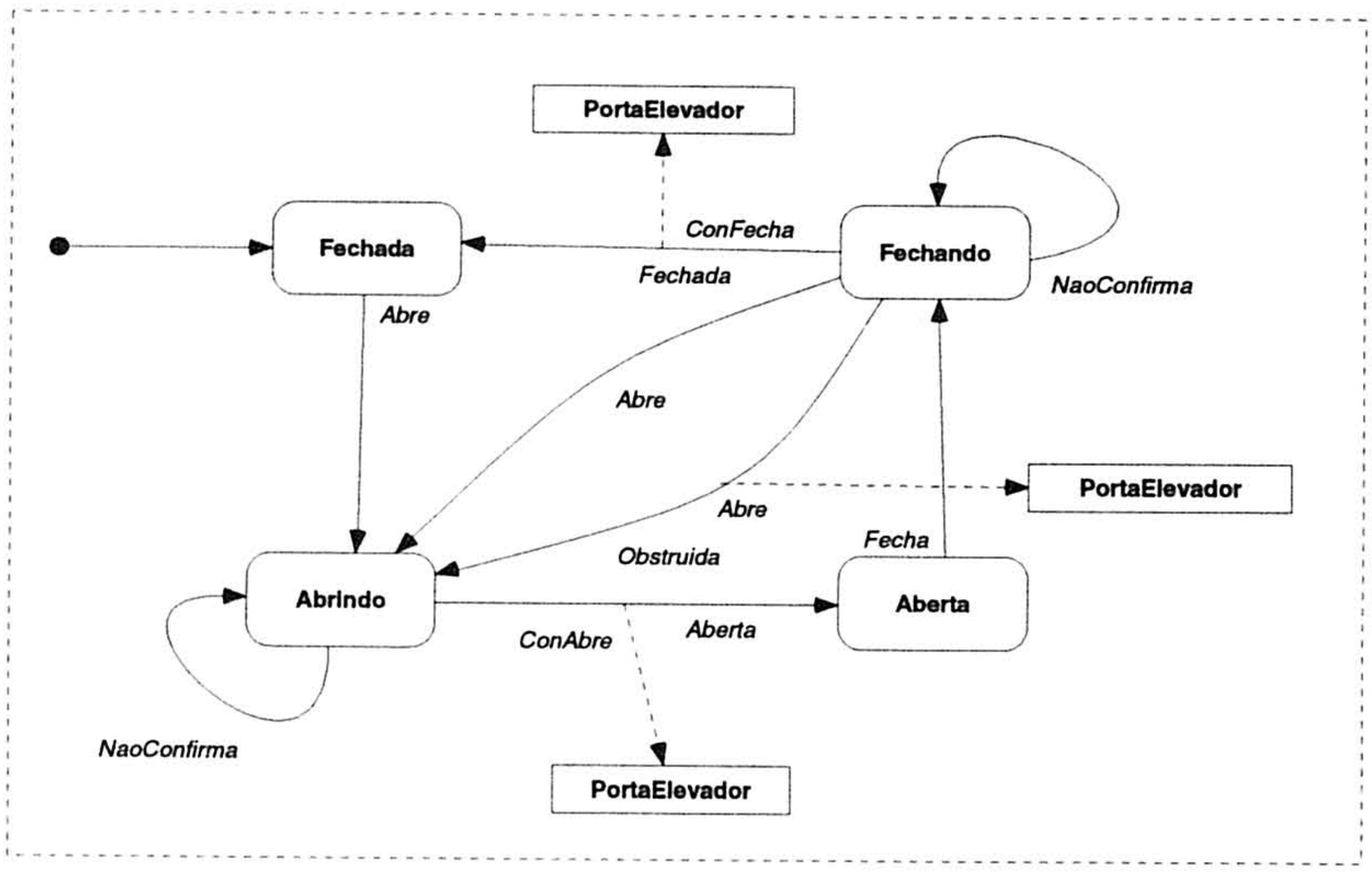


Figura 10.1: Diagrama de Estado OMT do Processo PortaPiso.

10.3.1 Gnomo

O processo Gnomo é responsável pelo atribuição das chamadas a cada um dos elevadores. Como os elevadores concorrem para servir as chamadas, é necessário garantir que a cada elevador em movimento esteja reservada pelo menos uma chamada, de modo a evitar que os elevadores viagem sem servir quaisquer chamada e que mais que um tente servir a mesma chamada. Essa chamada deve ser a chamada não reservada mais próxima do elevador. A distância é medida em pisos tendo em conta o sentido actual do elevador e da chamada. Essa distância para elevadores em movimento pode ser calculada pela equações 1, 2, 3 e 4:

$$D = \begin{cases} |PC - PE| & se \quad SE = SC \wedge Antes(PE, PC, S) \\ |PC + PE - 2PPSC| & se \quad SE \neq SC \\ |PC - PE| + 2(NP - 1) & se \quad SE = SC \wedge (Depois(PE, PC, S) \vee PE = PC) \end{cases} \tag{1}$$

em que, PE é o Piso em que está o Elevador, PC é o piso em que é efectuada a Chamada, SE é o Sentido do Elevador, SC é o Sentido da Chamada, NP é o Numero de Pisos e PPSC é o Primeiro Piso no Sentido da Chamada.

$$PPSC = \begin{cases} 1 & se \quad SC = Subir \\ NP & se \quad SC = Descer \end{cases}, \tag{2}$$

$$Antes(P1, P2, S) = \begin{cases} P1 < P2 & se \quad S = Subir \\ P2 > P1 & se \quad S = Descer \end{cases}, \tag{3}$$

$$Depois(P1, P2, S) = \begin{cases} P1 > P2 & se \quad S = Subir \\ P2 < P1 & se \quad S = Descer \end{cases}, \tag{4}$$

A equação 1 pode também ser expressa de uma outra forma, mais adequada quando é necessário optimizar o tempo de processamento.

$$D = \begin{cases} |PC - PE| & se \quad SE = SC \wedge Antes(PE, PC, S) \\ |PC - PE| + 2|PE - PPSC| & se \quad SE \neq SC \\ |PC - PE| + 2(NP - 1) & se \quad SE = SC \wedge (Depois(PE, PC, S) \vee PE = PC) \end{cases} \tag{1}$$

Quando o `Gnomo` recebe uma chamada (sinal `EscalaChamada`): rejeita-a se houver algum elevador parado no piso de onde é oriunda a chamada; senão guarda-a na base de dados, e se o elevador mais próximo dela estiver livre atribui-a a esse elevador (envia o sinal `ServeChamada` ao `Elevador`). Depois, sempre que um elevador actualiza o seu estado (sinal `EstadoElevador`) verifica qual o novo estado:

- se for parado num piso, informa o respectivo `PainelPiso` sobre a chamada que foi servida (sinal `ChamadaServida`), retira-a da base de dados e fornece ao elevador a chamada seguinte (ver a macro `EnviaChamada` da Figura A.13);
- se for ocupado (em movimento) e houver uma chamada pendente no piso por onde vai passar a seguir, envia-lhe a respectiva chamada;
- se for livre, verifica se entretanto não surgiu nenhuma chamada que ele possa servir., e nesse caso envia-a;
- se for fora-de-serviço simplesmente actualiza a base de dados dos elevadores.

A informação de supervisão relativa aos estados dos elevadores é repetida para o processo `Supervisor`.

As figuras que mostram o diagrama SDL deste processo estão na secção A.1 dos anexos.

10.3.2 Supervisor

O processo `Supervisor` faz o *interface* com o operador do sistema e vigia o sistema. Tem dois modos de registo dos eventos do sistema em tempo-real: um, em que somente guarda os alarmes num ficheiro, e

outro, em que para além do ficheiro de alarmes, tem também um ficheiro onde guarda todos os eventos do sistema (chamadas, destinos, alarmes e estados dos elevadores). O operador pode alternar entre esse dois modos e, ainda, parar o sistema. Sublinhe-se que o registo de alarmes é sempre garantido. Para além disso, este processo apresenta graficamente e em tempo-real a evolução de todo o sistema a partir da informação recebida dos outros processos. Consultar a Figura A.14. Quando o operador escolhe alternar o modo de registo de eventos (sinal de entrada `AlternaRegistoEventos`), troca o valor a uma variável interna, de forma que quando recebe eventos, verifica o seu valor para decidir se os regista ou não. Quando o operador decide parar o sistema (sinal de entrada `ParaSistema`), envia o sinal `Terminar` a todos os elevadores.

10.3.3 Painel de Chamada

Existe uma instância do processo `PainelChamada` em cada instância do bloco `Pisos`. Este processo está sempre em ciclo, e: quando um botão-de-chamada do seu piso é premido, envia a respectiva chamada ao processo `Gnomo`; quando recebe do `Gnomo` a mensagem que confirma a aceitação da chamada, acende a luz do respectivo botão; quando recebe a mensagem de chamada servida apaga a luz do botão correspondente. A Figura 10.2 mostra o diagrama SDL deste processo.

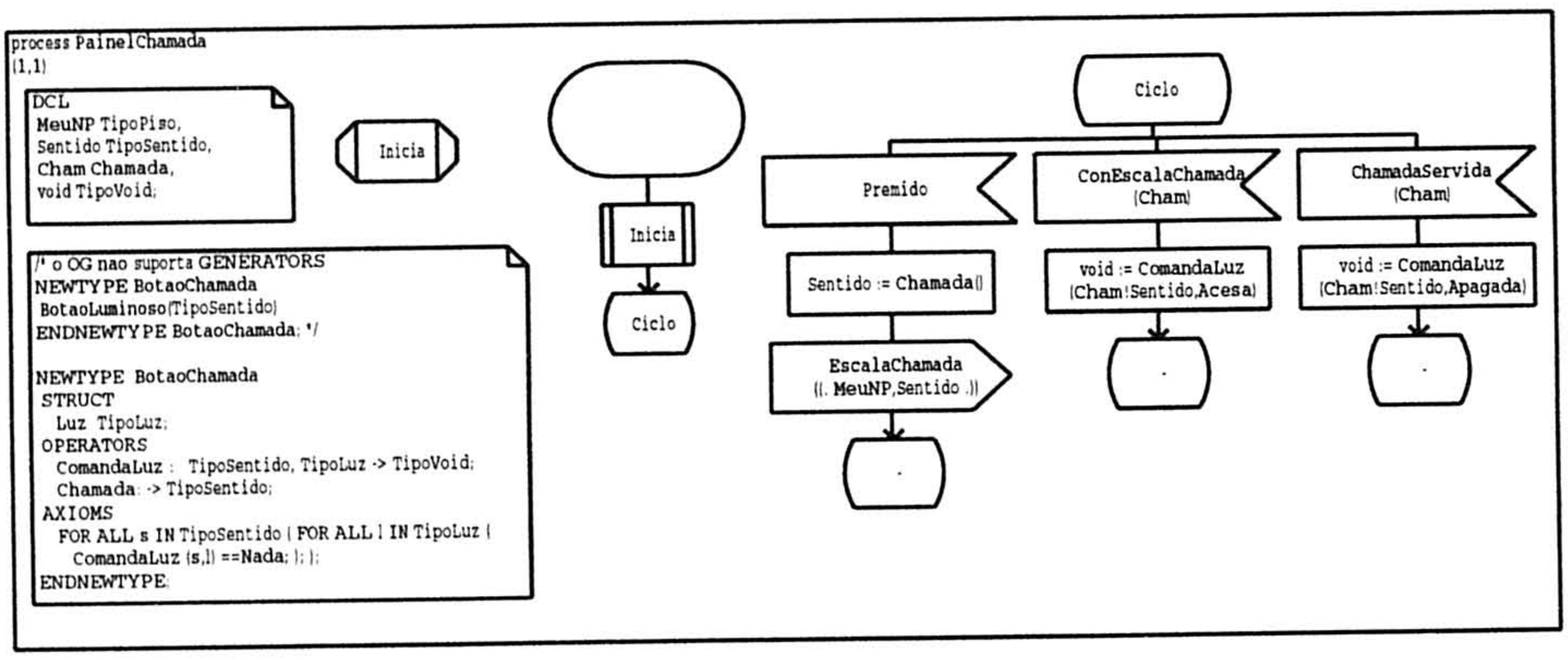


Figura 10.2: Diagrama de Processo SDL do Painel de Chamada.

10.3.4 Porta do Piso

O processo `PortaPiso`, já foi abordado em 10.2.4, por isso apresenta-se somente o diagrama de processo na Figura 10.3.

10.3.5 Elevador

O processo `Elevador`, quando recebe um pedido e está livre serve esse pedido (inicia o movimento para o servir), quando está ocupado guarda o pedido. O elevador só guarda uma chamada (a chamada que lhe está reservada) de cada vez, e se receber uma sem servir a que tem, mantém apenas a mais recente. Sempre que muda de estado (arranca, para, muda de sentido, passa num piso), notifica o `Gnomo` (sinal

ActualizaEstadoElevador). Ao aproximar-se de um piso consulta a base de dados de destinos e a chamada que lhe foi atribuída, para decidir se pára ou não. Quando está parado e termina o tempo de paragem, efectua o mesmo tipo de consulta para decidir se volta a partir. As figuras que mostram o diagrama SDL deste processo, estão na secção A.8 dos anexos.

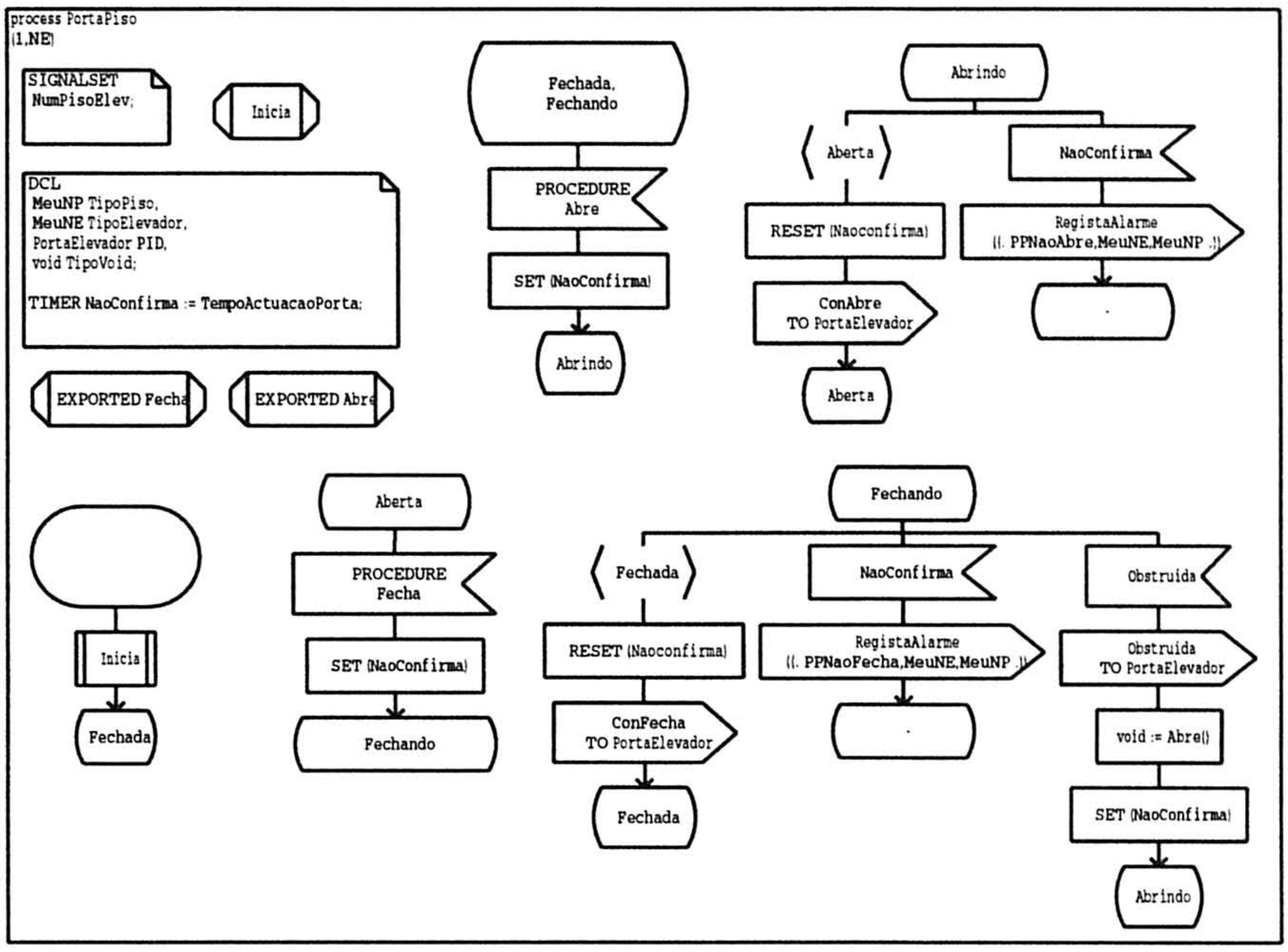


Figura 10.3: Diagrama de Processo SDL da Porta do Piso.

10.3.6 Painel de Destinos

O processo PainelDestinos trata os destinos da mesma forma que o PainelChamada lida com as chamadas. A Figura A.26 mostra o diagrama SDL deste processo.

10.3.7 Porta do Elevador

O processo PortaElevador recebe os sinais dos botões de comando da porta do elevador, serve as ordens do processo Elevador e, por sua vez, comanda o processo PortaPiso. Quando recebe a ordem para abrir um piso (abrir a porta do elevador e a porta do piso em que ele se encontra), envia a ordem para o respectivo processo PortaPiso e inicia a abertura da porta do elevador. A abertura pode não concretizar-se e, nesse caso, o temporizador expira e é gerado um alarme. Quando está aberta e recebe a ordem de fecho, solicita a MonitorCabo que o avise quando a carga for correcta. Quando receber essa indicação, envia a PortaPiso o sinal Fecha e inicia o fecho da porta do elevador. Durante o período de fecho, a porta do piso pode ser obstruída (recebe sinal Abre de PortaPiso) ou pode ser premido o botão de abertura (recebe o sinal BotaoPortaPremido). Em ambos os casos, interrompe o fecho, abrindo a porta e esperando pela expiração de um determinado tempo para tentar fechar de novo. A Figura A.27 mostra o diagrama SDL deste processo.

A complexidade da especificação deste processo advém do facto de ser necessário manter a coerência e sincronização com o processo `PortaPiso`. Tanto o problema da coerência, como o da sincronização, são problemas típicos de sistemas distribuídos. A sua resolução é por vezes complicada, já que os cenários de dessincronização e incoerência têm que ser todos previstos, ou então, tem que se recorrer a mecanismos de comunicação que façam reduzir o número de situações alternativas possíveis, tais como a Chamada de Procedimento Remoto (*Remote Procedure Call*), que é a única primitiva de comunicação síncrona em SDL.

10.3.8 Monitor do Cabo

Este processo pode ser implementado de duas formas: em três estados um para situação da tensão do cabo do elevador, ou num único estado e com uma variável interna com o valor da tensão. Optou-se pela última abordagem por ser mais económica em termos de código. As situações de tensão possíveis são: `Correcta`, se estiver dentro dos limites de bom funcionamento; `Excessiva`, se exceder a carga nominal; e `Ruptura`, se a tensão baixar do limite mínimo. Quando recebe o sinal `TensaoAlterada` lê a nova tensão, guarda o valor e actua o altifalante em conformidade com a situação. No caso de ruptura, actua também o freio de segurança e reporta o respectivo alarme. As figuras que mostram o diagrama SDL deste processo, estão na secção A.11 dos anexos.

10.4 SDL: Problemas e Soluções

Durante o projecto do SES, vários modelos foram construídos e varias versões de cada um foram elaboradas. Nesse período foram encontradas algumas dificuldades específicas da linguagem SDL que conduziram à elaboração de técnicas para as resolver. A seguir, apresentam-se dois problemas e as respectivas soluções técnicas encontradas.

10.4.1 Identificação de Instâncias de Blocos

Quando dentro de um tipo de bloco com N instâncias existe uma instância de um determinado processo, obtêm-se N instâncias desse processo que não têm meio de saber a que instância do bloco pertencem. Actualmente o grupo de peritos de manutenção da norma SDL está a estudar o caso e a considerar a utilização de tipos de portas e/ou a criação dinâmica de blocos para resolver este problema na nova versão (SDL-2000) da norma.

Uma solução é ter um servidor central (único no sistema) que, através de invocação de RPCs ou sinais, reserva números a blocos. Cada bloco teria um Processo de Identificação do Bloco (PIB) que na transição inicial contactaria o servidor para obter o número do seu bloco. Esta solução tem como desvantagem o facto de complicar o modelo devido à introdução de processos inúteis em termos da aplicação.

Para o sistema dos elevadores foi encontrada uma solução mais económica e que se adequa melhor à realidade do problema e a uma possível forma de plataforma de implementação. Em vez dos números serem reservados por um servidor central, existe em cada bloco um operador de um ADT que identifica o respectivo bloco. Cada processo dos blocos `Elevador` e `AcessoPiso` que necessita saber a que instância pertence, chama esse operador para obter respectivamente, o número de elevador e de piso, a que pertence respectivamente. Esse operador não necessita de ser implementado nem por axiomas nem algoritmicamente em SDL-PR, já que na simulação pode escolher-se o valor de retorno para cada bloco. Depois cada processo contacta o processo `Gnomo` (que funciona como o tal servidor único do sistema) comunicando-lhe o seu número (piso/elevador).

O processo `Gnomo` verifica a coerência de todos os números recebidos e guarda-os para os fornecer aos processos que deles necessitem. Esta solução adequa-se bem à implementação do `SES` sob a forma de um sistemas distribuído em que cada bloco corresponde a um nó onde existe uma função de acesso a *hardware* que lê o valor do respectivo número de piso/elevador.

10.4.2 Correspondência Determinística entre Instâncias de Processos de Diferentes Blocos

Um outro problema é quando se têm N instâncias de dois processos que estão em blocos distintos e se pretende estabelecer a comunicação entre elas de forma determinística. Isto é, que a instância 1 do processo do primeiro bloco corresponda à instância 1 do processo do segundo bloco. Como um processo não pode criar um processo noutro bloco, tem que se criar uma outra solução. Existem várias soluções possíveis, das quais se destacam três:

- Ter um servidor central no sistema que reserva números aos processos criados em cada bloco e depois fornece a quem precisa a correspondência entre identificadores de processos e números. Esta solução é pouco modular e pode ser bastante complexa no caso de haver criação dinâmica de processos.

- Introduzir um processo gestor em cada bloco para instanciar os processos locais e trocar os identificadores com o bloco remoto. Esse processo exporta um procedimento de criação de processos locais que retorna o identificador do processo criado, ou então recebe um sinal para o efeito e retorna o identificador através de outro sinal. Esta solução é bastante modular.

- Processo que a partir de uma instância inicial criam as suas próprias instâncias e comunicam em cadeia aos seus filhos os identificadores dos processos correspondentes. Começa-se com uma instância de cada processo em cada bloco. Como só existe uma instância elas comunicam entre si conseguindo o determinismo inicial. Depois de se conhecerem, criam o seu filho e dão a conhecer à instância correspondente do outro bloco o endereço dele. Recebem a mesma informação e fornecem ao filho. Estabelece-se assim a correspondência entre os filhos, e assim sucessivamente. Esta solução pode ser ilustrada pela hierarquia de cenários da Figura 10.4, e pelos respectivos MSCs da Figura 10.5 e da Figura 10.6. Esta é a solução mais optimizada de todas, mas torna-se confusa para o estabelecimento da comunicação entre mais que dois processos em blocos diferentes.

Existem ainda outras soluções, mas têm pouco interesse devido ao seu carácter específico e muito restritivo.

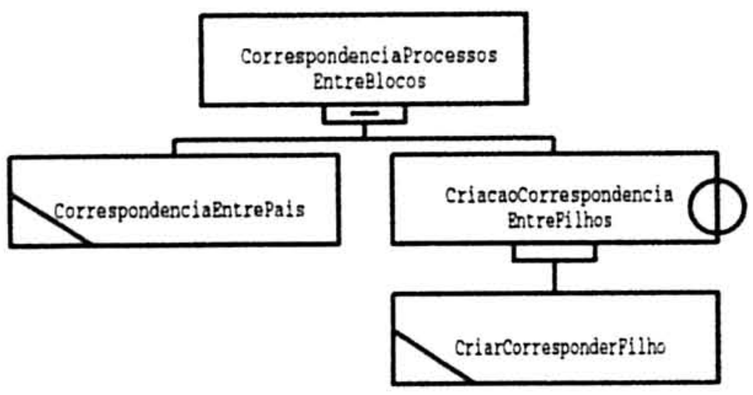


Figura 10.4: Hierarquia de Cenários da Correspondência entre Processos

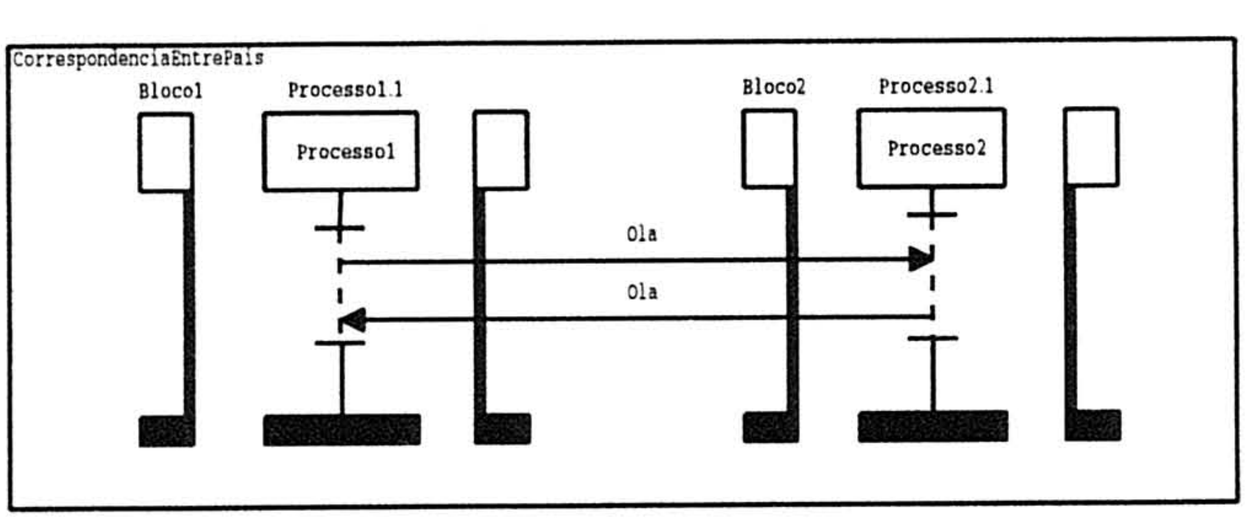


Figura 10.5: MSC da Correspondência entre Pais

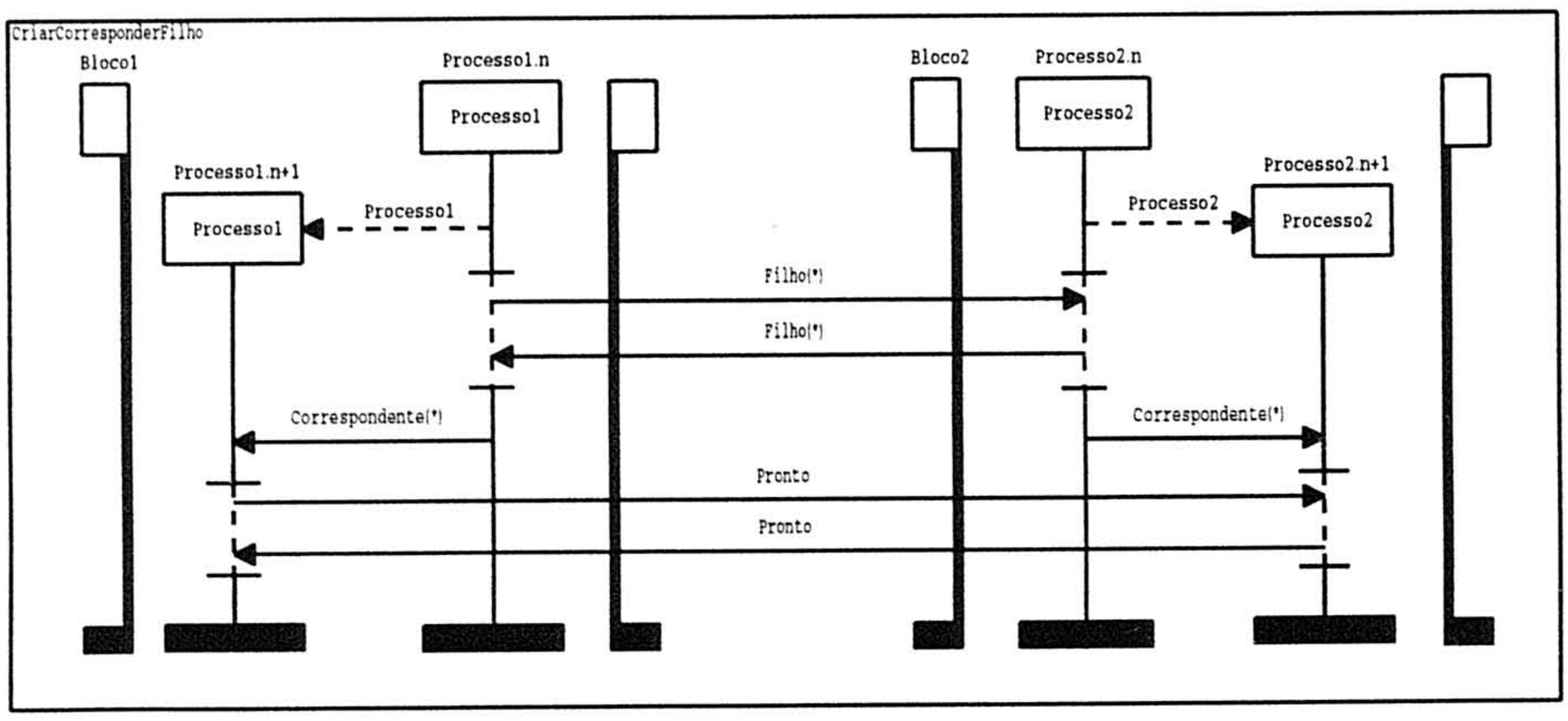


Figura 10.6: MSC da Correspondência entre Filhos

10.5 Projecto Detalhado dos Objectos Passivos

As referências feitas nesta secção a entidades da especificação em SDL do SES, referem-se às figuras apresentadas na secção 10.6.

10.5.1 Origem dos Objectos Passivos

Os objectos passivos foram identificados ao longo das fases de análise dos requisitos, projecto da arquitectura e projecto detalhado.

Os objectos passivos provenientes da fase de análise foram modelados como classes no modelo de objectos OMT. Geralmente correspondem a dados utilizados ou produzidos pelo sistema, e aparecem de novo no projecto detalhado para fornecer serviços aos objectos concorrentes (*e.g.*, para guardar e calcular valores). No caso do SES, tem-se, por exemplo, a classe Chamada que contém a informação relativa a uma chamada e que simboliza a base de dados de chamadas.

Durante o projecto da arquitectura e o projecto detalhado de objectos concorrentes em SDL, identificam-se novos objectos passivos de acordo com opções de projecto que têm que ser feitas. Estes não têm relação directa com os diagramas de análise. Podem resultar de soluções de projecto relativas à gestão de dados (*e.g.* BDElevador do processo Gnomo), *interface* com o utilizador (*e.g.* o ADT IU do processo Gnomo) ou ao *interface* com equipamentos (*e.g.* o ADT ConfigElev do processo Elevador). Estes objectos são definidos como ADTs com uma *interface* composta por vários campos como tipos, variáveis e operadores. No entanto, o conceito de ADT não permite a especificação completa das partes passivas. Deve empregar-se uma técnica orientada ao objecto para esse fim.

10.5.2 Especificar ADTs com Diagramas de Classes

A melhor forma de especificar ADTs é utilizar diagramas de classes OMT. Os ADTs têm que ser modelados como classes em diagramas de classes. Se forem necessários muitos ADTs, será mais adequado colocá-los em diagramas de classes diferentes para uma leitura mais fácil.

Os ADTs e as respectivas classes OMT têm que ter a mesma *interface* pública: as variáveis do ADT são transformadas em atributos da classe, e os operadores do ADT são transformados em operações da classe com as respectivas assinaturas (*signatures*). Para além disso, a linguagem SDL permite que os operadores dos ADTs sejam descritos por axiomas ou especificados por algoritmos, o que também pode ser feito em OMT explorando a definição de algoritmos nos corpos das operações.

10.5.3 Enriquecer o Projecto dos Diagramas de Classes

Os ADTs complexos são descritos através de diagramas de classses. São introduzidas novas classes que são ligadas aos ADTs modelados como classes através de associações, agregações e heranças. O projecto OMT fica completo quando todas as classes podem ser facilmente implementadas.

No caso do SES, são ilustrados três casos gerais: gestão de dados persistentes, *interface* gráfica com o utilizador e gestão de dispositivos físicos através de classes envolventes.

10.5.3.1 Gestão de Dados Persistentes

A modelação por objectos é adequada para desenhar meios de tornar os dados persistentes. No caso de *Relational Data Base Management Systems* (RDBMS), a tradução de diagramas de classes em modelos lógicos da base de dados já foi investigada, inclusivé para a abordagem OMT, e hoje existem soluções, algumas das quais acompanhadas por ferramentas de *software*. Uma primeira solução é traduzir uma classe numa tabela, em que os atributos formam novas colunas. Esta solução pode ser mais refinada de forma a optimizar o tamanho das bases de dados, os tempos de acesso, *etc.* Outra questão importante é encontrar a chave certa para as classes.

A utilização de um sistema de ficheiros é um caso semelhante ao da RDBMS. Os projectistas têm que decidir qual é a organização de ficheiros óptima: quais os atributos, classes ou conjuntos de classes que são guardados e em que ficheiros. Geralmente, a organização correcta é um compromisso entre vários factores, como tempo de acesso, redundância, partilha ou robustez. Implementar associações em ficheiros é equivalente a implementá-las em tabelas.

A utilização de *Object-Oriented* DBMS (OODBMS) torna o problema mais simples. Os OODBMSs comuns disponíveis no mercado proporcionam formas simples de tornar classes permanentes, por exemplo fornecendo uma classe dedicada que gere a persistência dos atributos das classes. Obviamente, esta classe tem que ser herdada pelas classes da aplicações que se pretendem persistentes. Nestes sistemas, as associações normalmente podem ser simplesmente guardadas como apontadores entre classes persistentes.

Um benefício adicional da utilização da mesma técnica OO para desenhar uniformemente os dados persistentes e outros objectos passivos é a facilidade que isso proporciona para integrar bases de dados, ficheiros, *etc.*, com o código que implementa os algoritmos de manipulação dos dados.

Na especificação do SES em SDL, o ADT `RegistoEventosPersistente` guarda os alarmes num ficheiro. No diagrama de classes, isso foi modelado como uma agregação entre as classes `Alarme` e `RegistoEventosPersistente`. De forma a tornar o registo dos alarmes persistente introduziu-se a `ClassePersistente`, à qual foi ligado `RegistoEventosPersistente` por herança.

10.5.3.2 *Interface* Gráfica com o Utilizador

Uma das vantagens da modelação por objectos na especificação das *interfaces* com o utilizador é que uma parte de informação que apareça em diferentes partes da aplicação (arquivo, computação, *interface* com o utilizador, *etc.*) é modelada por uma classe única. As suas diferentes características, ou facetas, são implementadas pelas operações. Deste modo, a integridade dos dados é melhorada por toda a aplicação, desde a *interface* do utilizador até às profundezas do código onde a informação está escondida.

Uma forma de desenhar *Graphical User Interfaces* (GUI) é adicionar a todas as classes cujos atributos estejam envolvidos, as funcionalidades de *interface* gráfica necessárias. Essas funcionalidades podem ser

adicionadas simplesmente por herança ou **delegação** (agregação), no caso de *interfaces* simples, ou criando novos diagramas de classes ligados às classes iniciais por associações ou herança.

As GUIs mais sofisticadas são, apesar disso, bastantes mais simples de desenhar utilizando geradores de *interface* apropriados. Mais ainda, a integração das classes produzidas por essas ferramentas é facilitada na modelação por objectos pela criação de classes envolventes adequadas que são associadas ao modelo original.

No projecto detalhado do modelo OMT do SES, foi escolhida a solução de associar a classe IU com a classe `Janela` que fornece funcionalidades gráficas como `EscreveTexto`.

10.5.3.3 Invólucros

A integração de equipamentos é outro tipo de classes de projecto. Os sistemas TR geralmente manipulam dispositivos como sensores, barramentos, ou dispositivos de E/S ou de comunicação com outros sistemas. Estes equipamentos são geralmente controlados por APIs fornecidas com eles.

Como normalmente os equipamentos são dinâmicos, têm que ser considerados no projecto de objectos concorrentes, e aparecem muitas vezes como objectos terminais da arquitectura SDL. No entanto, o objecto concorrente associado representa somente o comportamento do equipamento, pelo que a implementação necessita de algo mais para fazer o *interface* à API. Os objectos passivos criados para este efeito denominam-se **invólucros** (*wrappers*).

Em regra, um objecto SDL associado a um dispositivo contém um ADT que o representa. Esse ADT tem que fornecer todas as operações necessárias para controlar o dispositivo. Estes operadores são invocados durante as transições de estado. No modelo de objectos, esse ADT é transformado numa classe OMT invólucro que contém todas as propriedades necessárias.

No SES existem vários equipamentos para os quais foram definidos invólucros como, `Motor`, `ActuadorPorta`, `LeNumPiso`, *etc.*

10.5.3.4 Utilização de Outras Técnicas de Projecto

Para certos sistemas, em vez do OMT, ou em complemento, podem ser utilizadas outras técnicas de projecto. Acima discutiram-se as vantagens de geradores de *interfaces* gráficas para criar *interfaces* com o utilizador. Noutros domínios é possível tirar vantagem de técnicas específicas como a ASN.1 para desenvolver protocolos, ou no caso dos sistemas embebidos, de VHDL para desenhar em paralelo os vários componentes físicos constituintes de um sistema. Alias, existem já alguns trabalhos de projecto conjunto de *hardware/software* que a partir de especificações SDL geram programas C e código VHDL sintetizável (ver [Berg97]). Como o método OORT é totalmente baseado em técnicas OO, a utilização de outras técnicas normalmente não constitui problema.

10.6 Projecto Detalhado do SES em OMT

O projecto dos objectos do SES foi parcialmente realizado. As várias categorias identificadas foram incluídas num diagrama de classes OMT representado na Figura 10.7.

10.7 Reutilização de Objectos Externos

Na fase de projecto detalhado, é possível reutilizar componentes de tempo-real (SDL) e passivos (OMT). Em ambas as respectivas notações, existem mecanismos específicos para melhorar a reutilização.

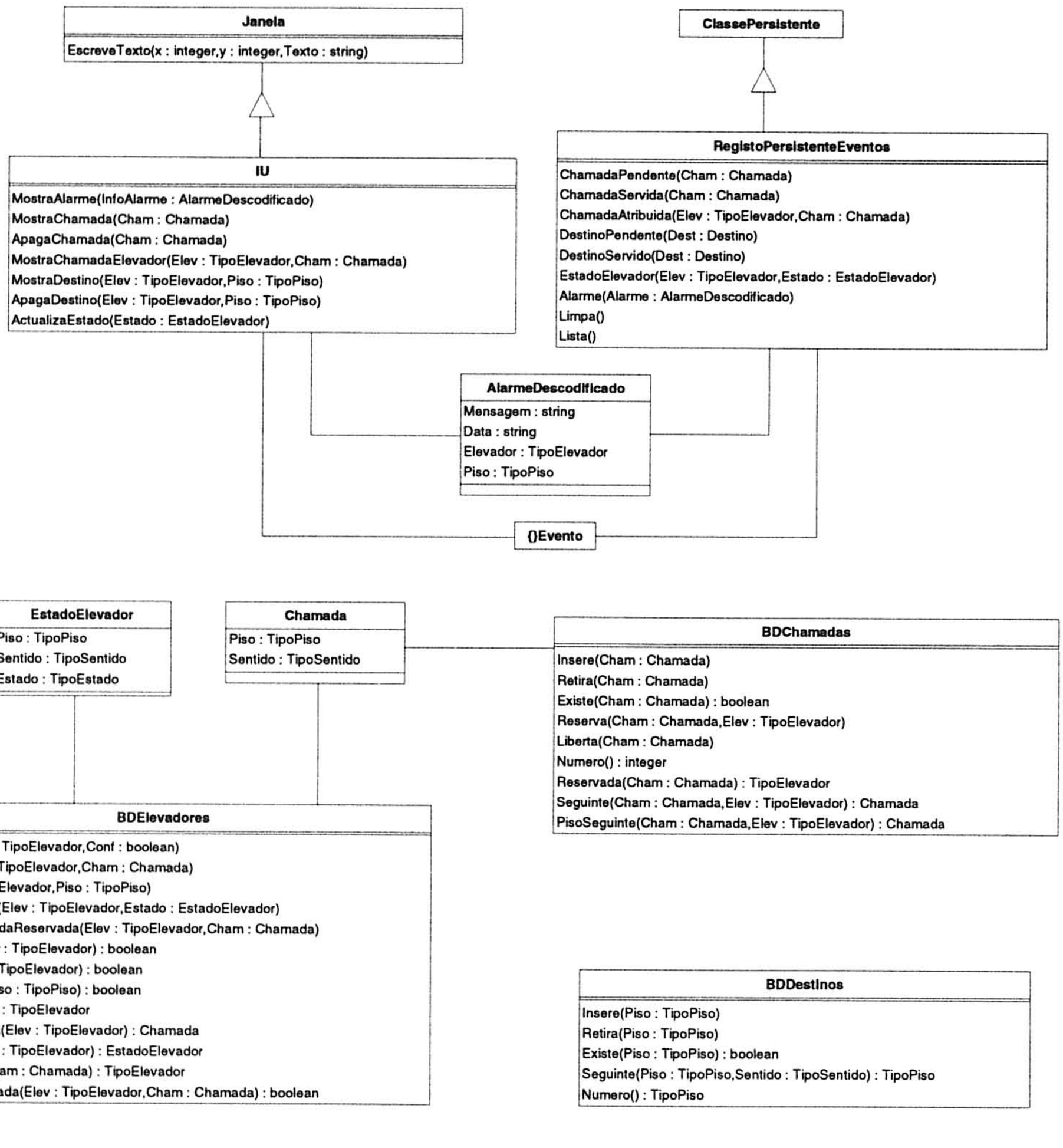


Figura 10.7: Diagrama de Classes OMT do projecto detalhado do SES.

10.7.1 Reutilização de Objectos Concorrentes

Ao nível da descrição SDL, a reutilização dos objectos concorrentes é suportada pelos seguintes mecanismos:

- Encapsulamento

  A implementação de um objecto SDL é totalmente escondida do exterior. Um objecto SDL comunica com outros objectos através da troca de sinais. As estruturas de dados internas e a forma como as respostas são elaboradas não é visível. A única restrição que deve ser respeitada pelos objectos externos dialogantes é a definição dos sinais trocados (parâmetros e declarações dos tipos de dados associados).

- Generalidade (*generacity*)

  As entidades estruturantes de SDL (blocos, processos, *etc*) podem ser genéricas. Este mecanismo permite desenvolver entidades com parâmetros genéricos. Em vez de ter canais ligados a outras entidades, são definidas portas com os seus próprios sinais associados. Para utilizar estas entidades genéricas, os parâmetros genéricos têm que ser instanciados e as portas ligadas aos canais existentes. Deste modo, as entidades que tiverem comportamentos idênticos podem ser substituídos por instâncias de um único tipo genérico.

- Herança

  Tal como outras notações OO, a SDL oferece o mecanismo de herança. Os blocos ou processos podem herdar entidades do mesmo tipo. As entidades especializadas podem redefinir transições particulares de forma a adaptarem-nas a requisitos específicos.

10.7.2 Reutilização de Objectos Passivos

A modelação por objectos OMT também torna a reutilização mais simples, graças aos seguintes mecanismos:

- Encapsulamento e Delegação

  Os objectos OMT oferecem uma *interface* pública através da qual podem interagir com o mundo exterior em que os detalhes de implementação ficam escondidos. Um objecto reutiliza outro simplesmente pelo acesso às suas propriedades públicas. Não existe qualquer limite à visibilidade das *interfaces* públicas. No caso do acesso a atributos, é melhor modelar a interacção pela criação de uma associação, tal como já foi mencionado na actividade de análise. A reutilização também é frequentemente conseguida através de agregações. Este mecanismo é denominado **delegação**: objectos de baixo nível já existentes podem ser integrados em diagramas de classes pela criação de agregações com objectos de nível superior identificados, por exemplo, na fase de análise.

- Herança

  A OMT suporta a herança múltipla. As operações das classes pais podem ser redefinidas através da hierarquia de herança.

A generalidade (*genericity*) não é suportada pela OMT, o que significa que esta facilidade só pode ser utilizada no passo de implementação se a linguagem de programação escolhida o permitir (C++, por exemplo).

10.7.3 Produção de Componentes Reutilizáveis

A produção de componentes reutilizáveis é facilitada principalmente pelo paradigma dos objectos suportado pelas linguagens SDL e OMT. Um componente reutilizável pode corresponder a uma classe OMT, a um processo SDL isolado, a um conjunto de classes OMT interrelacionadas, ou a um bloco SDL contendo uma hierarquia de entidades SDL. A *interface* pública de tal componente é constituída por todas as *interfaces* públicas das classes OMT contidas ou pela *interface* pública da entidade SDL de nível mais alto (*i.e.* sinais e declarações de tipos de dados).

10.8 Portabilidade do Projecto Detalhado

O comportamento de tempo-real da aplicação é completamente especificado através de construções gráficas formais da linguagem SDL. Como resultado, o projecto é independente dos dispositivos físicos que compõem a plataforma destino (*target*), bem como do SOTR sobre o qual a aplicação irá ser executada. A máquina virtual SDL adapta a especificação à plataforma real, ou seja, a multi-tarefa, a comunicação entre processos, ou a gestão do tempo, ficam directamente ao cuidado da máquina virtual SDL. Além disso, como o projecto SDL é totalmente independente da distribuição dos objectos, a arquitectura lógica pode ser traduzida (*mapped*) em diferentes configurações da plataforma, como foi explicado na secção 9.2.5. As comunicações remotas são processadas como comunicações locais. A escolha da configuração final do *hardware* será feita com base em considerações sobre o desempenho.

Os objectos concorrentes estão fracamente ligados à configuração real da plataforma, enquanto os objectos passivos fazem directamente o *interface* com a plataforma (*target*). Como consequência, a portabilidade depende dos meios escolhidos para implementar os objectos passivos: linguagens de programação, *interfaces* com DBMS, GUI, *etc.*

A correspondência (*mapping*) entre a especificação SDL e a configuração real da plataforma é feita na fase de implementação (*targeting*).

Graças aos mecanismos de projecto propostos pelo OORT, a aplicação pode ser desenvolvida e mantida ao nível do projecto, o que melhora substancialmente a flexibilidade e portabilidade.

10.9 Relações com as outras Actividades de Desenvolvimento

10.9.1 Relação com o Projecto de Testes

Os objectos passivos e concorrentes concebidos nesta fase formam os elementos de *software* que têm que ser testados independentemente através de testes unitários.

Os testes unitários, correspondendo ao último nível da arquitectura SDL, podem ser considerados como o conjunto de requisitos que os processos e as classes têm que satisfazer: os processos têm que dar respostas (sinais, temporizadores, mudanças de estado, *etc.*) conforme as sequências de interacção esperadas, e os objectos passivos têm que gerar valores de saída em conformidade.

10.9.2 Transição para a Fase de Implementação (*Targeting*)

A implementação diz respeito aos processos SDL e às classes OMT, mas questões relacionadas com os aspectos de tempo-real do sistema são unicamente ponderados na parte SDL.

A utilização normal de notações formais permite conseguir a geração automática de código. Por conseguinte, a actividade de implementação (*targeting*) será altamente dependente dos resultados da actividade de projecto.

Deverá ser prestada especial atenção à ligação de código fonte proveniente de diferentes descrições. Esta tarefa será mais simples utilizando ADTs declarados na descrição SDL para integrar o código resultante da modelação OMT.

11 Projecto de Testes

11.1 Introdução

O projecto de testes pode ser uma tarefa muito demorada e pode requerer tanto esforço quando o desenvolvimento propriamente dito, em particular nos casos de sistemas de segurança-crítica (*safety-critical*) ou missão-critica (*mission-critical*). Em [You94], Yourdon cita que as proporções de tempo gasto em cada uma das actividades são na ordem de 50-60% para a análise e projecto, 10-15% para a implementação e 25-35% para a integração e teste. Por conseguinte, todos os meios que ajudem o projecto de testes têm que ser utilizados de forma a reduzir o tempo e os custos.

O projecto de testes é realizado em paralelo com o projecto da arquitectura e do *software*. Esta actividade tem por objectivo produzir testes utilizados para a integração de objectos. Os testes ao nível do processo e da classe são utilizados para verificação unitária, e ao nível das entidades estruturantes SDL servem para fazer a verificação da integração.

A validação do sistema é realizada pela execução de testes de aceitação. Estes testes, inicialmente compostos por MSCs nominais de análise, são depois melhorados por um conjunto de testes de desempenho e testes que verificam o comportamento do sistema em situações críticas. Os MSCs de nominais de análise são testes funcionais, que geralmente formam o contracto entre o produtor e o cliente. Os testes de desempenho e os testes de situações críticos são não-funcionais e servem para avaliar propriedades do sistema como segurança contra faltas acidentais (*safety*) e faltas intencionais (*security*), evolução do desempenho em várias situações, *etc.*

Esta actividade envolve diversas equipas: projectistas (os autores da arquitectura), gestão do projecto (responsáveis pelo desenvolvimento do sistema), controlo da qualidade (pessoas encarregues dos assuntos de qualidade) e a equipa de validação (os utilizadores dos testes).

A notação MSC adequa-se bem ao projecto de testes, principalmente aos que são executados em objectos concorrentes.

11.2 Projecto do Conjunto de Testes

11.2.1 Produção de MSCs Detalhados

Os testes são extraídos de MSCs detalhados que refinam os casos de utilização produzidos na fase de análise de requisitos (MSCs abstractos definidos ao nível do sistema), através da descrição de mensagens e acções trocadas pelos objectos identificados na arquitectura do sistema. A Figura 11.1 ilustra o processo de refinamento hierárquico dos MSCs.

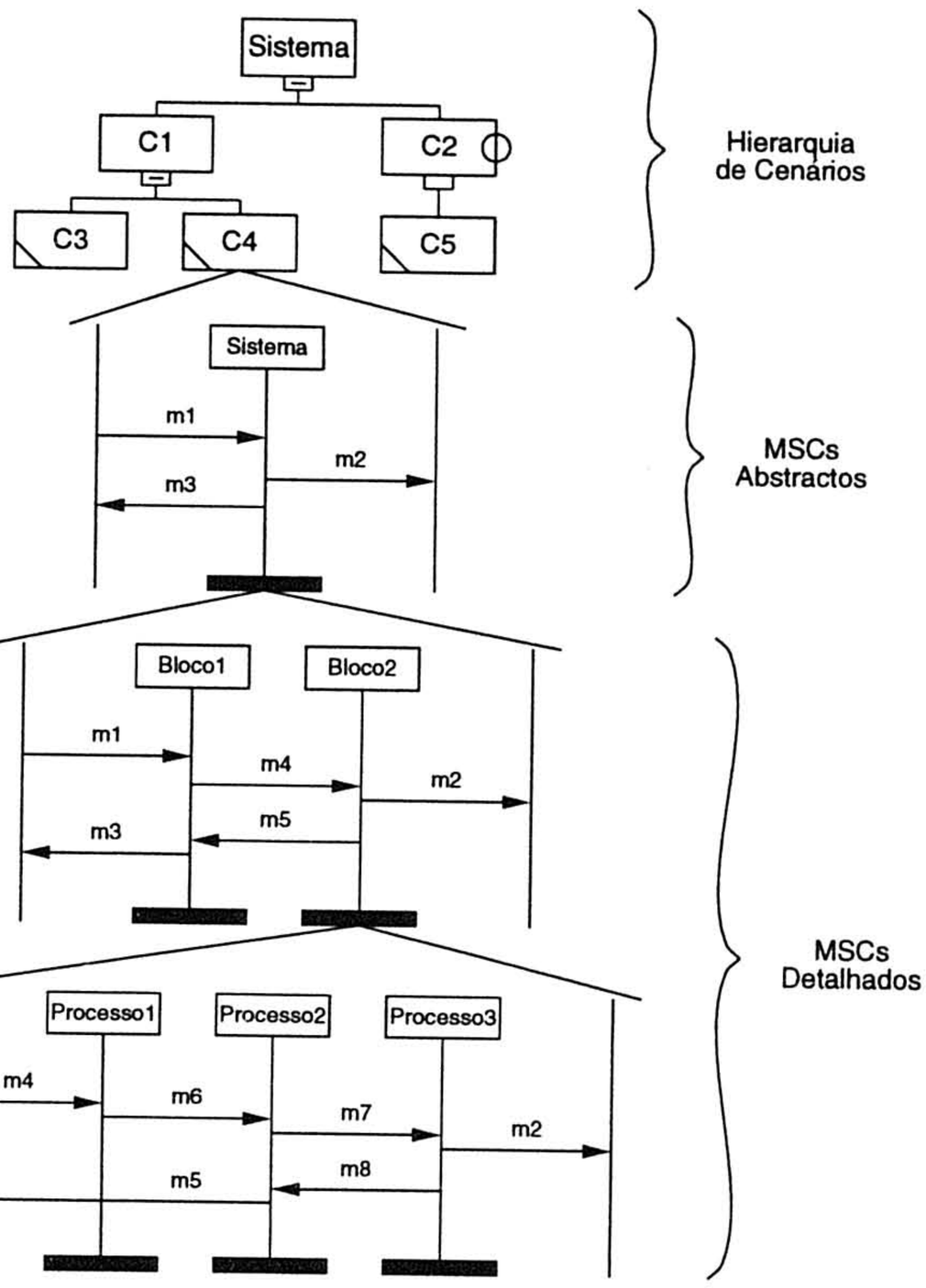


Figura 11.1: Refinamento Hierárquico de MSCs.

Os MSCs detalhados contêm instâncias das entidades que compõem a hierarquia SDL. A comunicação entre entidades descritas nos MSCs tem que ser coerente com a hierarquia SDL. As principais regras de correspondência são as seguintes:

- as barras verticais são instâncias das entidades estruturais SDL.
- as mensagens MSC correspondem a sinais SDL.
- cada transmissão de mensagem tem que ser suportada por um canal que liga as duas entidades em diálogo no modelo SDL.

Adicionalmente, os objectos SDL encontrados nos diagramas de processo, tais como **estados**, **procedimentos** ou **temporizadores**, podem ser utilizados em MSCs detalhados para descrever o comportamento do processo de uma forma mais precisa. De forma inversa, a descrição de objectos concorrentes através de MSCs ajuda a encontrar estados, transições, *etc.*

11.2.2 Extensão da Cobertura feita pelos Testes

Os testes baseiam-se no modelo de casos de utilização e, portanto, reflectem a visão que o utilizador final tem da aplicação. A cobertura feita pelos testes tem que ser alargada pela adição de novos testes que verifiquem aspectos relacionados com a arquitectura: desempenho, robustez, segurança contra faltas acidentais e intencionais, flexibilidade, *etc.* Contudo, estes testes não verificam funções de utilização; ao invés, permitem que o cliente avalie a qualidade do sistema desenvolvido.

Os testes arquitecturais dependem fortemente da natureza da aplicação. Por exemplo, no caso de sistemas distribuídos, a verificação do projecto poderia consistir em:

- medir o tempo médio de resposta a estímulos numa situação normal;
- saturar o sistema pela introdução de informação de entrada a uma taxa anormalmente alta;
- desligar um nó de um sistema distribuído;
- gerar erros num canal de comunicação;
- criar entradas erradas;
- tentar aceder a um nó e ter acesso à base de dados local e a outros nós a ele ligados;
- colocar o sistema a trabalhar em diferentes configurações de *software*;
- colocar o sistema a operar em diferentes configurações de *hardware*;

Obviamente, a cobertura dos testes deve ser mais extensiva quando se testam objectos reutilizáveis, que para além de serem irem ser utilizados em vários projectos, provavelmente, irão ser utilizados em aplicações de outros domínios de aplicação.

11.2.3 Testes de Integração

No processo geral de projecto de testes, os testes de integração correspondem ao refinamento intermédio dos casos de utilização, complementados com os testes de arquitectura. Quando o refinamento é realizado sobre a arquitectura SDL, o último nível SDL é ignorado pois está fora do âmbito.

Os testes de integração concentram-se inteiramente na comunicação entre objectos concorrentes. Os objectos passivos não são considerados. Do ponto de vista do projecto do sistema, os testes de integração fornecem uma referência a cumprir a cada nível hierárquico. Ver na secção 11.3 um exemplo de um MSC com vários níveis hierárquicos.

11.2.4 Testes Unitários

Os testes unitários correspondem ao último refinamento dos testes de integração aplicado ao último nível da hierarquia SDL. Nos MSCs detalhados, utilizam-se elementos gráficos detalhados, como estados, temporizadores, procedimentos, tarefas, *etc.* Estes elementos têm que ser compatíveis com as acções das transições especificadas nos respectivos diagramas de processo.

Os testes unitários também se aplicam aos objectos passivos. Como o comportamento local de um objecto concorrente pode ser detalhado por meio de tarefas, é possível indicar chamadas a operações de classes dentro de uma sequência de interacções. Estas tarefas podem servir como pós-condições a serem satisfeitas pelos objectos passivos utilizados pelos objectos concorrentes.

11.2.5 Concordância do Projecto com os Requisitos

O projecto dos testes complementa a análise e projecto do sistema: a análise expressa **o que** o sistema deve fazer (quais as funções esperadas), enquanto os testes unitários e de integração indicam **como** as funções esperadas são implementadas. O projecto de testes através de MSCs melhora o rastreio (*traceability*) da informação entre os requisitos e o projecto. O PES do OORT é um processo de tal modo integrado que qualquer requisito que o projecto não cumpra pode ser detectado bastante cedo no ciclo de vida do desenvolvimento da aplicação. Por outro lado, o "excesso de projecto" (*i.e.*, projectar funcionalidades não pretendidas) é mais facilmente detectado. Todavia, considerações de projecto, como procurar desenvolver componentes reutilizáveis, ou reutilizar componentes existentes, pode conduzir a concepções mais sofisticadas.

11.3 Testes do SES

O teste detalhado representado na Figura 11.2 descreve como deve ser executada, pelos blocos SDL da arquitectura `SES`, a chegada de um elevador a um piso. Este teste foi construído pela pormenorização, ao nível do bloco do MSC `InicioViagem` do cenário de recepção de chamadas, definido na fase de análise do sistema.

Por sua vez, o MSC ao nível do bloco pode ser refinado ao nível do processo da arquitectura SDL. As barras verticais que anteriormente representaram blocos do SES passam a representar processos. O mesmo caso de utilização está descrito ao nível dos processos na Figura 11.3, para o bloco `Pisos`, na Figura 11.4, para o bloco `Central`, e na Figura 11.5, para o bloco `Elevadores`.

Uma vez mais e por razões já apontadas, encontra-se no Anexo C a listagem completa dos testes descritos por MSCs.

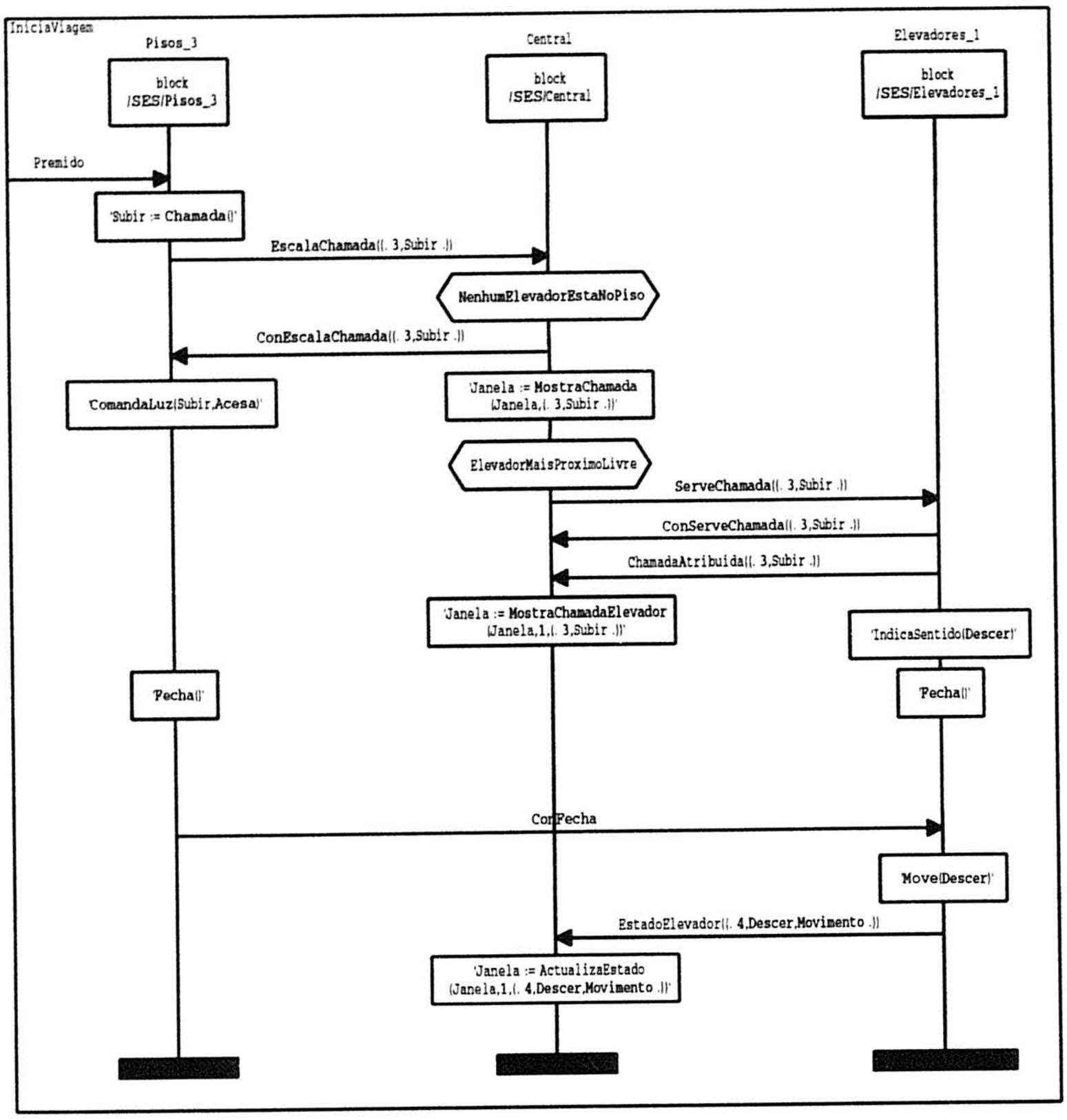


Figura 11.2: MSC Detalhado ao Nível dos Blocos do SES.

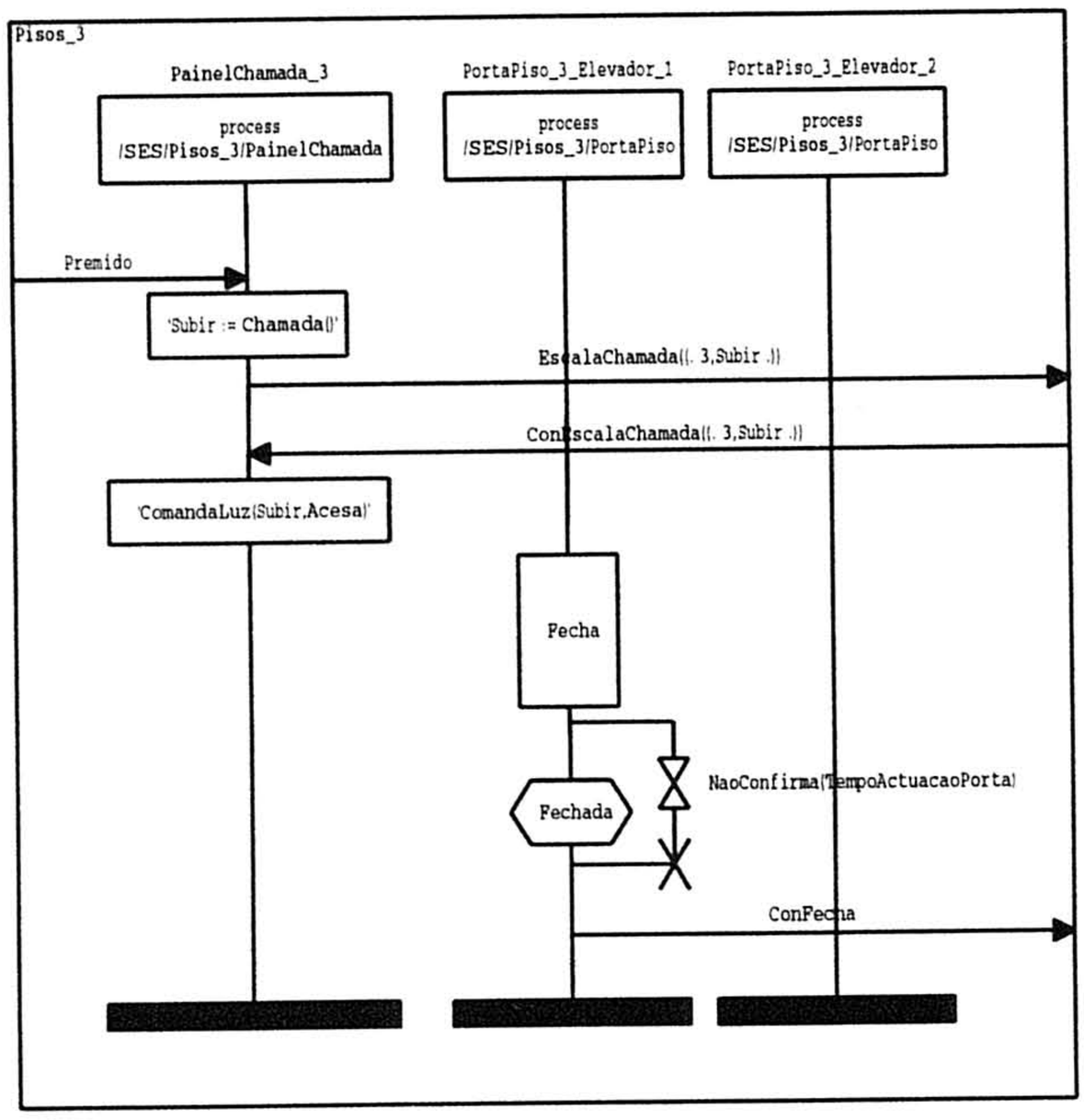


Figura 11.3: MSC Detalhado ao Nível dos Processos do Bloco Pisos.

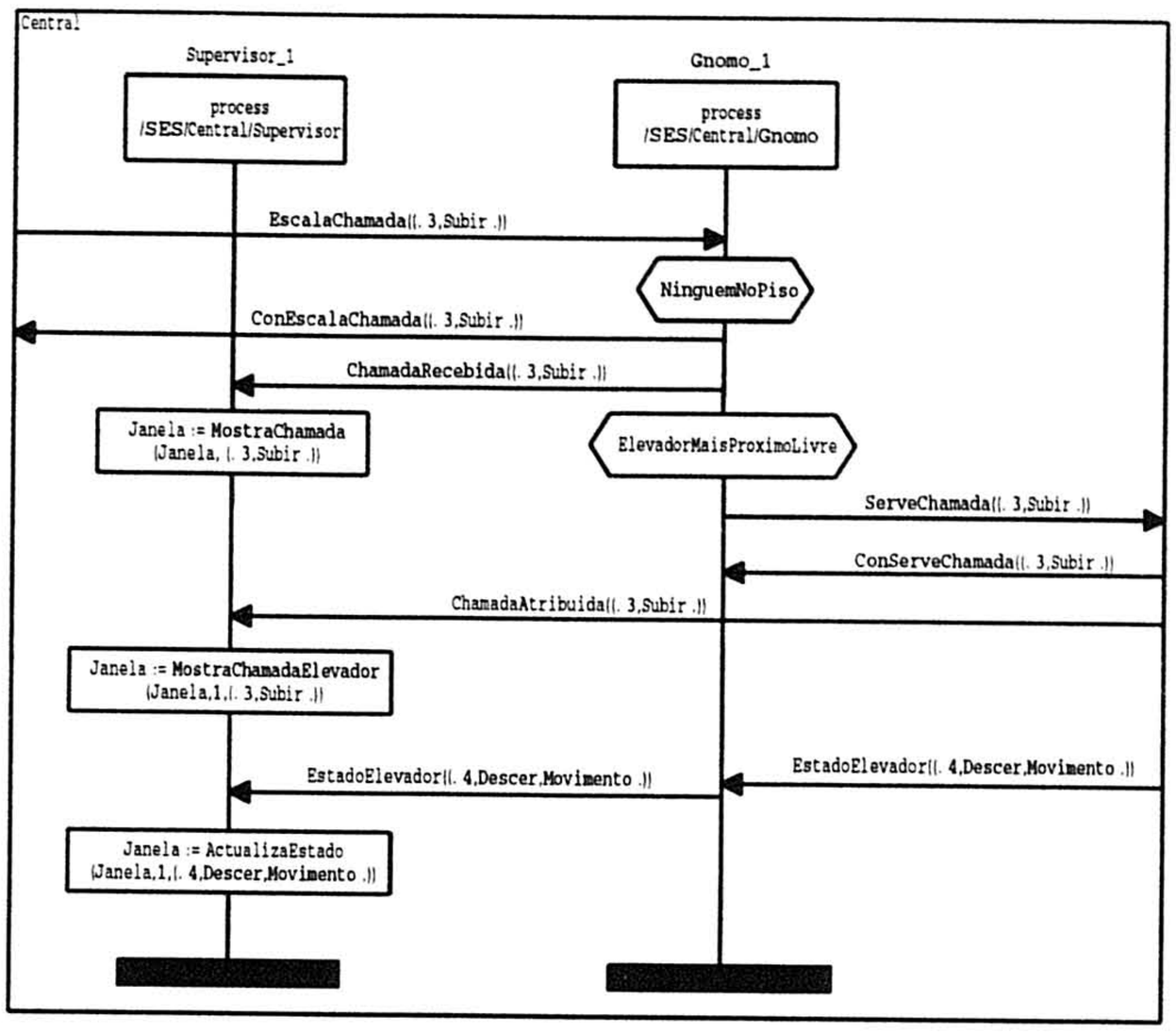


Figura 11.4: MSC Detalhado ao Nível dos Processos do Bloco Central.

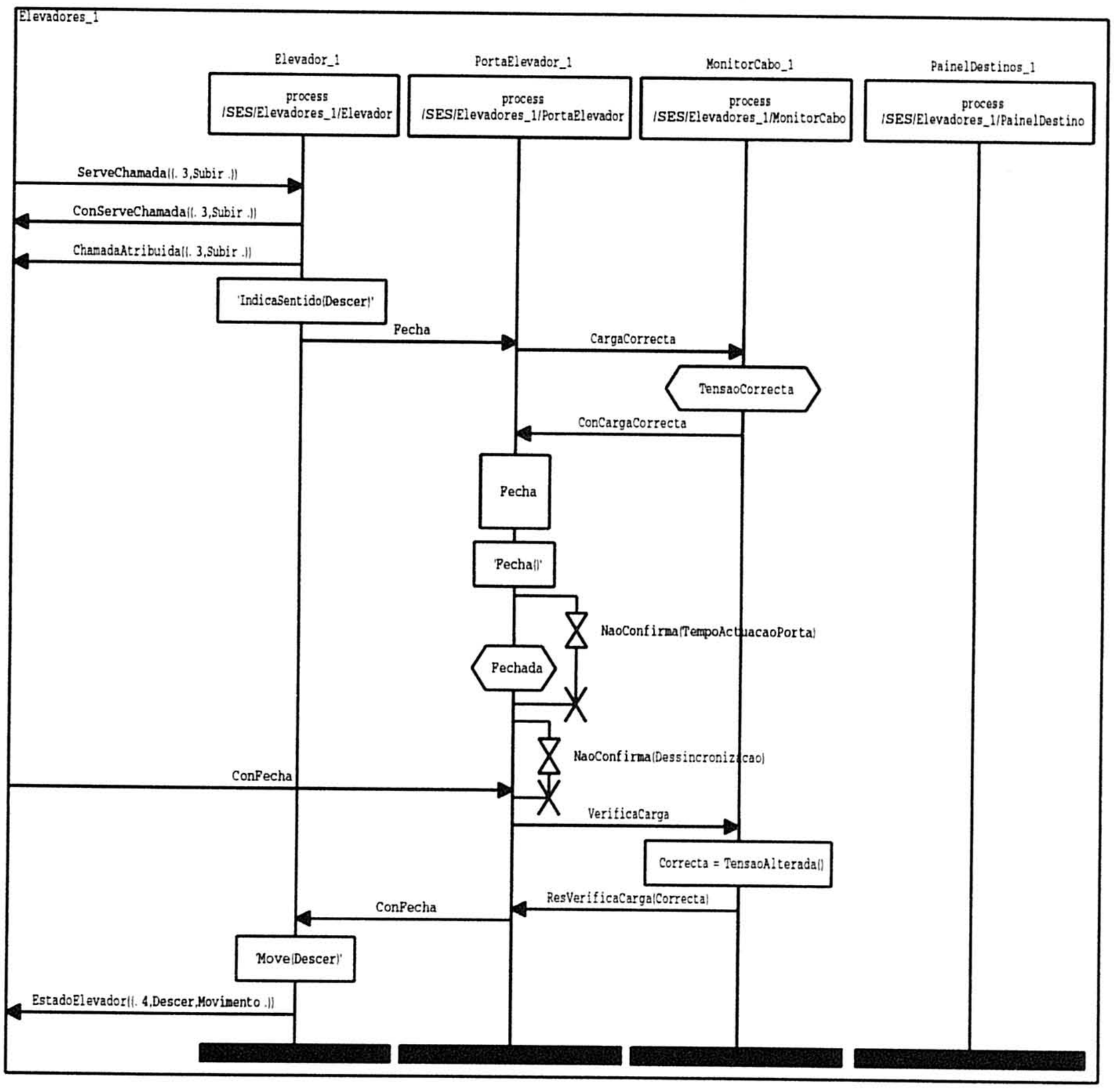


Figura 11.5: MSC Detalhado ao Nível dos Processos do Bloco Elevadores.

11.4 Relações com as outras Actividades de Desenvolvimento

11.4.1 Relação com a Actividade de Projecto Detalhado

Os testes que formam o comportamento de referência dos processos SDL, podem ser utilizados como entradas para simular e validar a descrição desses mesmos processos. Se assim for, os MSCs podem ser vistos como padrões para procurar sequências esperadas ou erradas. Os testes podem ser muito numerosos, por isso devem ser seleccionados os mais significativos para o propósito de validar a arquitectura.

Além disso, os testes produzidos pela simulação, e mais especificamente pela simulação exaustiva, podem ser integrados nos testes construídos manualmente.

Os testes unitários também podem servir para especificar o comportamento dos objectos passivos explicitando o resultado esperado da operação de uma classe invocada no contexto de uma determinada sequência de interacção.

11.4.2 Transição para o Teste

De forma a facilitar o teste da aplicação, é possível gerar testes em formatos apropriados recorrendo aos testes MSC. Por exemplo, no campo das telecomunicações onde a notação TTCN é largamente utilizada, uma solução é produzir casos de teste TTCN a partir das descrições SDL seguindo os objectivos de teste expressos em MSCs.

12 Simulação

12.1 Introdução

A ferramenta *Object*GEODE dispõe de um simulador que permite simular modelos SDL, comparando-os dinamicamente com MSCs do comportamento esperado e gerando MSCs do comportamento verificado na simulação.

Basicamente, a execução de um modelo SDL é constituída por um conjunto de passos, de estado para estado, disparando transições:

- um **estado** corresponde a um estado estável do sistema, ou seja, todos os processos activos estão num determinado estado;
- um **passo** corresponde à execução de uma transição escolhida de entre o conjunto de transições disparáveis, *i.e.* uma transição é disparada de entre todas as transições possíveis dos processos activos.

O simulador suporta três modos de simulação:

- Interactiva – o utilizador toma o papel do ambiente e observa o funcionamento interno do sistema;
- Aleatória – o simulador executa o modelo e escolhe aleatoriamente a transição a disparar, sempre que há mais que uma;
- Exaustiva – o simulador executa automaticamente o modelo e investiga todas as possibilidades, *i.e.* explora todos os estados do sistema e faz *back-tracking* quando um ramo foi totalmente coberto.

A simulação é muito importante, pois permite reduzir os custos de desenvolvimento do sistema reduzindo o número de versões falhadas, ou seja, ajuda os projectistas a aproximarem-se do tão ambicionado "correcto à primeira vez".

A simulação tem três funções principais:

- Prototipagem rápida

  O propósito é, no início do ciclo de vida do projecto, simular os diagramas criados para responder à questão "o sistema realiza alguma função?". Para isso, utilizam-se os modos interactivo e aleatório do simulador.

- Verificação

Pretende detectar erros de projecto de forma a responder à pergunta "está-se a construir o sistema correctamente?". Pode ser realizada nos três modos.

- Validação

  O objectivo é validar o comportamento observado comparando-o com o comportamento esperado, de forma a responder à questão "está-se a construir o sistema pretendido?". Tal como no caso anterior, pode ser realizada nos três modos.

Os três modos de simulação podem ser utilizados pela ordem apresentada, *i.e.* na ordem crescente do nível de exigência do sistema. O modo interactivo pode ser utilizado para fazer os primeiros testes e verificar detalhadamente alguns casos importantes para a correcção e ultimação do funcionamento geral do sistema, *i.e.* verificar que o sistema faz alguma coisa. Mas obviamente, essa não é a forma adequada para cobrir um grande número de casos. Depois de haver certeza quanto ao funcionamento do algoritmo global da aplicação, podem-se testar um maior número de casos com vista à detecção de erros dinâmicos, como impasses, partes do modelo não executadas, recepção de sinais inesperados, sinais sem receptor e ultrapassagens (*overflows*) dos limites dos tipos (nomeadamente no processamento dos dados feito por ADTs). Para isso, recorre-se ao modo aleatório, verificando se o sistema está a ser construído correctamente. Quando a confiança no sistema é boa, chega a altura de fazer a verificação exaustiva do comportamento do sistema e compará-lo com os serviços esperados. Isto permite detectar interacções que violam propriedades definidas, sequências de interacção esperadas, *i.e.* conferir se o sistema faz o pretendido. Finalmente, pode fazer-se a avaliação da robustez e comportamento de segurança, detectando condições críticas. Foi esta a abordagem seguida neste trabalho.

12.2 Sistema Distribuído de Dois Elevadores

Após a simulação interactiva e aleatória do sistema especificado fizeram-se várias simulações exaustivas. A seguir apresenta-se parte dos resultados obtidos de uma dessas simulações, para uma configuração de 2 elevadores e 4 pisos:

```
...
(622592 states 1174150 transitions 2806 seconds, depth=39, breadth=131629)
verify interrupted

Number of states : 622726
Number of transitions : 1174374
Maximum depth reached : 39
Maximum breadth reached : 131629
duration : 46 mn 52 s

Number of exceptions : 0
Number of deadlocks : 0
Number of stop conditions : 0
Transitions coverage rate : 67.38 (46 transitions not covered)
States coverage rate : 98.11 (1 states not covered)
Number of errors : 4848
Number of success : 2400
  observer ses: 4848 errors, 2400 success

Sub-graphs statistics :
process supervisor : 450 states
process gnomo : 442 states
process pisos_1!painelchamada : 4 states
```

```
process pisos_1!portapiso : 10 states
process pisos_2!painelchamada : 15 states
process pisos_2!portapiso : 7 states
process pisos_3!painelchamada : 10 states
process pisos_3!portapiso : 5 states
process pisos_4!painelchamada : 4 states
process pisos_4!portapiso : 5 states
process elevadores_1!elevador : 720 states
process elevadores_1!paineldestino : 14 states
process elevadores_1!monitorcabo : 5 states
process elevadores_1!portaelevador : 54 states
process elevadores_2!elevador : 176 states
process elevadores_2!paineldestino : 14 states
process elevadores_2!monitorcabo : 5 states
process elevadores_2!portaelevador : 22 states
queue supervisor!queue : 103 states
queue gnomo!queue : 41 states
queue pisos_1!painelchamada!queue : 1 states
queue pisos_1!portapiso!queue : 4 states
queue pisos_2!painelchamada!queue : 5 states
queue pisos_2!portapiso!queue : 4 states
queue pisos_3!painelchamada!queue : 5 states
queue pisos_3!portapiso!queue : 3 states
queue pisos_4!painelchamada!queue : 3 states
queue pisos_4!portapiso!queue : 3 states
queue elevadores_1!elevador!queue : 15 states
queue elevadores_1!paineldestino!queue : 8 states
queue elevadores_1!monitorcabo!queue : 3 states
queue elevadores_1!portaelevador!queue : 12 states
queue elevadores_2!elevador!queue : 13 states
queue elevadores_2!paineldestino!queue : 8 states
queue elevadores_2!monitorcabo!queue : 3 states
queue elevadores_2!portaelevador!queue : 10 states
observer observation!ses : 85 states
...
```

Realce-se o facto de não terem ocorrido impasses. Durante o tempo de simulação foram cobertos quase todos os estados mas somente cerca de 70% das transições. O número de erros encontrados traduz o número de situações que não estão previstas nos observadores MSC do projecto de testes ou que são diferentes das sequências lá especificadas. Um cenário é diferente se envolver instâncias diferentes e/ou sinais com valores diferentes. O número de sucessos traduz a quantidade de situações encontradas que estão de acordo com os cenários especificados em MSCs. Como não foram considerados todos os cenários é natural o resultado obtido. Para conseguir expressar todas as ocorrências dos cenários do Anexo C, teriam que ser descritas todas as combinações de interacções possíveis entre instâncias de pisos e elevadores num sistema de 4 pisos e 2 elevadores. Como é fácil de imaginar, isso representaria um volume de testes impraticável. Por isso, para além dos observadores MSC provenientes dos testes adicionaram-se apenas algumas variantes. Tendo isso em conta, o número de situações de conformidade com os MSCs de teste pode considerar-se significativo.

Cada linha começada por *process* informa qual o número de estados por que passou um processo ao longo da simulação. O estado de um processo depende do estado da sua FSM, do tamanho da fila de espera, da ordem das mensagens na fila de espera, *etc*. O mesmo acontece com as linhas referentes às filas de espera.

O simulador fornece ainda o número de vezes que cada transição foi disparada, bem como o número de vezes que cada estado ficou activo. Destes resultados pode extrair-se informação valiosa relativa ao modelo: quais as transições e estados mais repetidos, transições nunca disparadas, estados nunca activados, *etc*. A seguir apresenta-se um excerto desses dados relativo à taxa de cobertura de transições e da taxa de cobertura de estados do processo `Elevador` da instância 1 do bloco `Elevadores`.

```
transitions coverage of elevadores_1!elevador : rate 57.69
```

```
  start : 2
  from_all_but_aguardarespostaporta_aguardaportafechada_input_acaboutempo   :
98
  from_all_but_aguardaportaaberta_input_conabre : 16
  from_aguardarespostaporta_input_acaboutempo : 10
  from_aguardarespostaporta_input_confecha : 0
  from_inicioaguardaportaaberta_input_conabre : 2
  from_inicioaguardaportaaberta_input_portanaoconfirma : 0
  from_aguardaportaaberta_priority_input_conabre : 1114
  from_aguardaportaaberta_input_portanaoconfirma : 0
  from_aguardapiso1_input_novopiso : 0
  from_all_input_reportaalarme : 0
  from_all_input_terminar : 0
  from_aguardapiso_input_novopiso : 13857
  from_all_but_livre_input_servedestino : 11231
  from_all_but_livre_input_servechamada : 699
  parte : 2217
  from_aguardaportafechada_input_acaboutempo : 0
  from_livre_input_servedestino : 21104
  from_livre_input_servechamada : 31603
  from_livre_provided : 0
  from_livre_input_confecha : 55
  call_pedidospendentes_1 : 10
  call_pedidospendentes_2 : 0
  call_inicia : 2
  discard : 0
  timeout_portanaoconfirma : 0

states coverage of elevadores_1!elevador : rate 87.50
  start : 2
  livre : 5457
  aguardaportafechada : 47431
  aguardapiso : 20446
  aguardapiso1 : 0
  aguardaportaaberta : 8513
  inicioaguardaportaaberta : 2
  aguardarespostaporta : 1335
```

12.3 Cooperação de Processos Distribuídos

Sendo os sistemas distribuídos mais complexos, porquê construí-los? Obviamente, que estes têm vantagens como melhor desempenho, maior tolerância a falhas, *etc.* No domínio dos elevadores, isso é claro. Se num edifício existirem dois elevadores, cada um com os seus botões de chamada independentes, *i.e.* dois sistemas não comunicantes, obviamente eles não poderão partilhar os pedidos dos potenciais passageiros. Assim, cada um vai responder aos pedidos isoladamente. Suponha-se que num determinado instante o elevador A está no piso inferior e o elevador B está próximo do piso superior. Admita-se que nesse momento um utilizador no último piso chama o elevador A. O elevador A terá que viajar toda a altura do edifício. Se os dois elevadores fossem parte de um único sistema, o elevador B subiria apenas alguns pisos e serviria o utilizador em muito menos tempo e poupando energia. Esse sistema é composto por componentes que cooperam entre si, pois dividem o trabalho a efectuar. A isto pode chamar-se cooperação de processos. A cooperação de processos distribuídos permite conseguir uma maior eficiência.

Mas pode argumentar-se que um sistema cooperante não tem que ser distribuído. Obviamente que não, mas em certos sistemas é vantajoso que seja. Imagine-se um edifício com muitos pisos, onde um sistema centralizado com tão grande número de entradas e saídas exigiria grandes quantidades de cabos eléctricos e uma capacidade de processamento considerável só para tratar os eventos. O sistema distribuído poderia

ser constituído por uma rede que interligaria pequenos nós com capacidade de processamento muito inferior.

Uma plataforma adequada para a implementação deste sistema é ter uma rede que interligue vários nós:

- um pequeno nó por piso, isto é, por cada instância do tipo de bloco `Piso`, já que a carga computacional associada é pequena;
- um nó de capacidade de processamento intermédio em cada elevador, que executará cada instância do tipo de bloco `Elevador`, por exemplo um microcontrolador;
- um nó central que executará os processo do bloco `Central` que pode muito bem ser um PC.

Os factores a considerar para a definição da plataforma passarão obviamente pela distribuição da carga, a eficiência e a redução de custos.

BIBLIOTECA F.E. PORTO

13 Conclusões

Este trabalho permitiu conhecer as linguagens OMT, MSC e SDL e aferir a sua validade, individual e conjunta, para a implementação de controladores distribuídos em ambiente industrial. Como todas são linguagens orientadas ao objecto podem ser combinadas de forma consistente para o desenvolvimento de sistemas tempo-real. De facto, a utilização conjunta revela o potencial de cada uma das notações. A modelação por objectos da notação OMT é simples e concisa. As linguagens MSC e SDL são utilizadas na sua totalidade. São notações maduras e bem definidas: as construções que proporcionam são essenciais, deixando de fora mecanismos que não sejam úteis ou genéricos. Além disso, são normas que vão sendo constantemente melhoradas, mas sempre dentro de uma filosofia de manter/introduzir apenas o que é realmente necessário.

A metodologia OORT baseia-se nas notações OMT, MSC e SDL, proporcionando uma solução completa que cobre o processo de desenvolvimento de sistemas desde a análise até à implementação. Em particular, a metodologia OORT fornece facilidade de rastreio (*traceability*) ao longo de todo o PES, graças a um conjunto de regras que assegura a consistência de cada fase e das passagens entre fases. Além disso, simplifica a reutilização, porque utiliza notações orientadas ao objecto.

A implementação pode ser automatizada já que as notações empregues são formais, limitando o código manual às operações que não são de tempo-real. A aplicação gerada é adaptável (*scalable*), pois a arquitectura da aplicação é elaborada independentemente da arquitectura física. A correspondência entre os objectos e o *hardware* só é definida na fase de implementação.

Para além das razões apresentadas, a adopção de uma metodologia de especificação "obriga" o projectista a pensar de forma organizada e consistente toda a aplicação, pelo que comparativamente ao desenvolvimento de *software* não formalizado, resultam aplicações melhores em termos de eficiência, erros, flexibilidade e facilidade de manutenção e actualização.

O *Object*GEODE é a ferramenta de suporte ao método OORT e permite gerar o código fonte da aplicação. Esse código é portável já que é gerado de forma a ser ligado a uma biblioteca dinâmica (*run-time*) que forma uma camada entre a aplicação e a plataforma (SOTR e *hardware*). O *Object*GEODE já fornece bibliotecas para alguns SOTRs, das quais somente uma corre em pequenos microcontroladores como os da família 8051, que constituem um elo muito útil com o mundo da indústria. Este trabalho faz parte de um projecto de carácter mais amplo. Integrada nesse projecto, a próxima actividade será desenvolver uma biblioteca que permita distribuir o código gerado por diversos sistemas, distribuídos e interligados por uma rede industrial. Em particular, optou-se por utilizar a ferramenta *Object*GEODE para gerar o código e integrá-lo com a biblioteca de serviços de rede para um barramento CAN de microcontroladores. O objectivo é formar uma base de trabalho que permita estudar outros casos de algoritmos de controlo distribuídos e os seus problemas, de uma forma integrada desde a especificação até à implementação e teste. Esse estudo poderá produzir resultados como métodos de partição

hardware/software, estratégias de distribuição de processos e protocolos de sincronização de processos distribuídos.

Lista de referências

Literatura Técnica

[Ber97] Bergé J-M, Levia O., Rouillard J., *Hardware/Software Co-design and Co-Verification*, Kluwer Academic Publichers, 1997.

[Boo91] Booch G., *Object-Oriented Analysis and Design with Applications, 2nd edition*, Benjamin Cummings, Redwood City, 1993.

[Dou98] Douglass B.P., *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley, , 1998.

[Fae94] Faergemand O., Olsen A., *Introduction to SDL-92*, Computer Networks and ISDN Systems, vol. 26-9, 1994.

[Jac92] Jacobson I., Christerson M., Jonsson P., Overgaard G., *Object-Oriented Software Engineering, A Use Case Driven Approach*, ACM Press, Addison-Wesley, 1992.

[Ols94] Olsen A., Faergemand O., Moller-Pedersen B., Reed R., Smith J.R.W., *Systems Engineering Using SDL-92*, North Holland, 1994.

[Rat97] Rational Software Corporation, *The Unified Modeling Language, Version 1.1*, 1997.

[Rud96] Rudolph E., Graubmann P., Grabowski J., *Tutorial on Message Sequence Charts*, Computer Networks and ISDN Systems, vol. 28-12, 1996.

[Rum91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modeling and Design*, Prentice Hall, Inc., 1991.

[Ver96a] Verilog, *Object*GEODE *Method Guidelines*, Verilog SA, 1996.

[Ver96b] Verilog, *Object*GEODE *Tutorial*, Verilog SA, 1996.

[Ver96c] Verilog, *Object*GEODE *SDL Simulator Reference Manual*, Verilog SA, 1996.

[Enc97] Vincent Encontre, *How to Use Modeling to Implement Verifiable, Scalable, and Efficient Real-Time Application Programs*, Real-Time Engineering, Fall 1997.

[You94] Yourdon E., *Object-Oriented Systems Design: An Integrated Aproach*, Prentice Hall, 1994.

Normas

[ASN95] ITU-T Recommendation Z.105, *SDL Combined with ASN.1*, March 1995.

[CIF96] ITU-T Recommendation Z.106, *Common Interchange Format for SDL*, October 1996.

[MSC96] ITU-T Recommendation Z.120, *Message Sequence Chart (MSC)*, ITU, October 1996.

[SDL93] ITU-T Recommendation Z.100, *Specification and Description Languge (SDL)*, March 1993.

[SDLA94] ITU-T Recommendation Z.100 Appendix 1, *SDL Methodology Guidelines*, March 1994.

[SDLA96] ITU-T Recommendation Z.100 Addendum 1, *Specification and Description Languge (SDL) Addendum 1*, October 1996.

[SDLS97] ITU-T Recommendation Z.100 Supplement 1, *SDL + Methodology: Use of MSC and SDL (with ASN.1)*, May 1997.

Fontes Essenciais de Informação na Internet

MSC	http://www.win.tue.nl/cs/fm/sjouke/msc.html
SDL	http://www.sdl-forum.org
UML	http://www.rational.com/uml
Verilog	http://www.verilog.com

Anexo A Diagramas SDL do Projecto Detalhado do SES

Apresentação de todos os diagramas SDL que constituem a especificação do SES, excluindo somente a implementação algorítmica dos ADTs mais extensos definidos ao nível de alguns processos.

A.1 SES

As figuras seguintes constituem, em conjunto, a descrição do nível mais alto da hierarquia SDL do SES (nível do sistema).

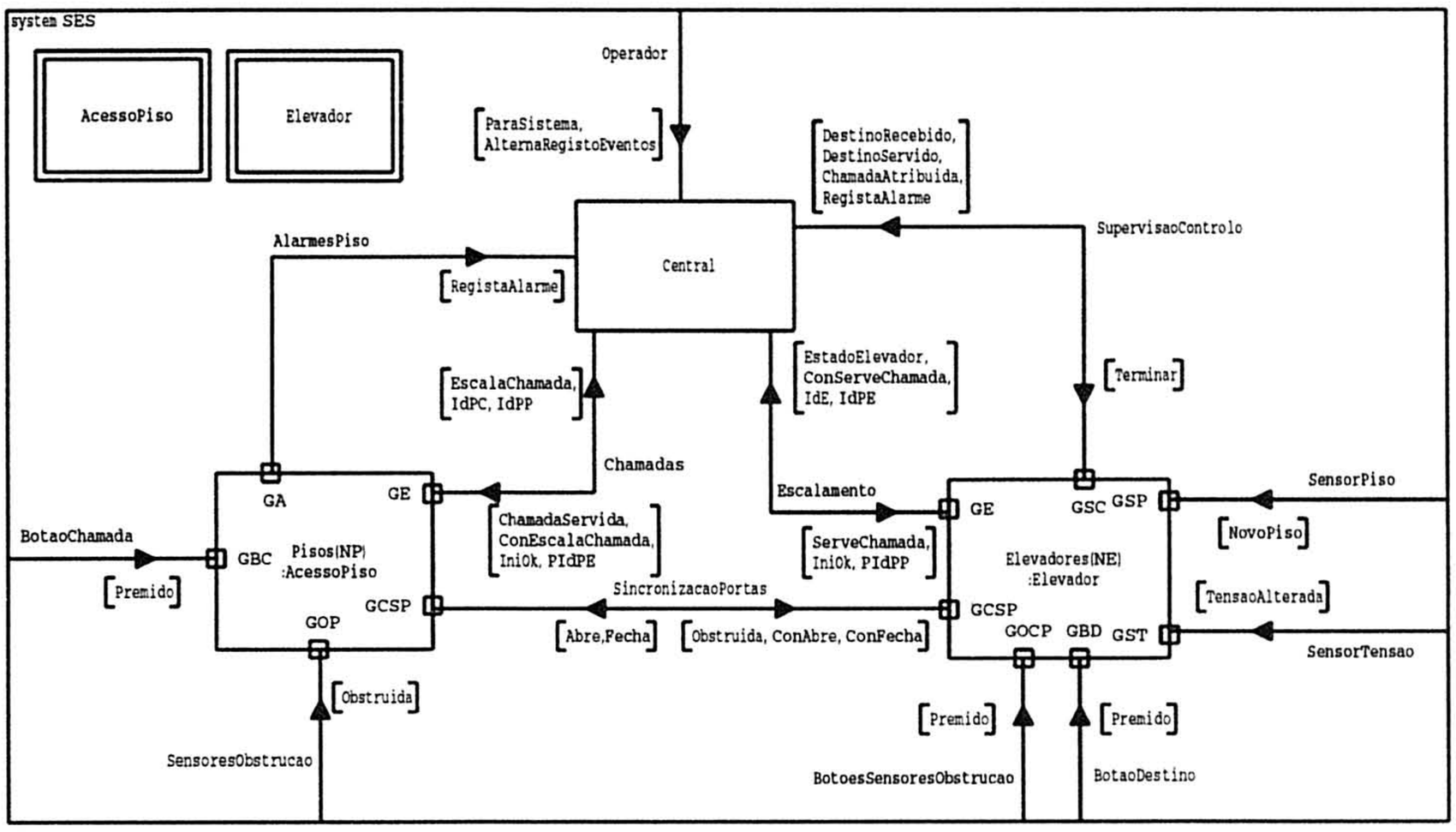


Figura A.1: Diagrama de Interligação do SES.

```
system SES

/* DEFINICAO DE SINAIS */
/*******************************/
SIGNAL
/* Do Operador para a Central */
 ParaSistema, AlternaRegistoEventos,
/* Dos Botoes para o SES */
 Premido,
/* Dos Sensores de Obstrucao para o Piso */
 Obstruida,
/* Do Sensor de Tensao para o Elevador */
 TensaoAlterada,
/* Do Sensor de Piso para o Elevador */
 NovoPiso,
/* Entre as Portas */
 Abre,Fecha, ConAbre, ConFecha,
/* Da Central para o Elevador */
 Terminar,
ServeChamada(Chamada), ConServeChamada(Chamada),
/* Do Elevador para a Central */
 EstadoElevador(DadosElevador),
 DestinoRecebido(TipoPiso), DestinoServido(TipoPiso),ChamadaAtribuida(Chamada),
/* Entre Piso e Central */
 ChamadaServida(Chamada), EscalaChamada(Chamada),
 RegistaAlarme(Alarme),
ConEscalaChamada(Chamada),

/* Sinais de Inicializacao */
 IdPC(TipoPiso), IdPP(TipoPiso,TipoElevador), IdE(TipoElevador), IdPE(TipoElevador),
 PIdPE(PID), PIdPP(TipoPiso,PID),
 IniOk;

/* PROCEDIMENTOS REMOTOS */
/*************************************/
REMOTE PROCEDURE Abre;
REMOTE PROCEDURE Fecha;
```

Figura A.2: Declaração de Sinais ao Nível do SES.

```
system SES

/* DEFINICAO DE SINONIMOS */
/**************************************/

SYNONYM NE NATURAL = 2;
SYNONYM NP NATURAL = 4;

SYNONYM Nenhum NATURAL = 0;

/* No ObjectGEODE a escala temporal por defeito e 0.1 segundos */
SYNONYM DecSeg NATURAL = 1;
SYNONYM Seg NATURAL = 10;

SYNONYM TempoActuacaoPorta DURATION = 3*Seg;
```

Figura A.3: Declaração das Constantes ao Nível do SES.

```
system SES

/* DEFINICAO DE TIPOS */
/******************************/

SYNTYPE TipoElevador = NATURAL
 CONSTANTS 0:NE
ENDSYNTYPE;

SYNTYPE TipoPiso = NATURAL
 CONSTANTS 0:NP
ENDSYNTYPE;

NEWTYPE IteradorPiso
 OPERATORS
   PrimeiroPiso : TipoSentido -> TipoPiso;
   UltimoPiso : TipoSentido -> TipoPiso;
   Seguinte : TipoPiso, TipoSentido -> TipoPiso;
   Anterior : TipoPiso, TipoSentido -> TipoPiso;
   CalculaSentido : TipoPiso,TipoPiso -> TipoSentido;
   Antes : TipoPiso, TipoPiso, TipoSentido -> BOOLEAN;
   Depois : TipoPiso, TipoPiso, TipoSentido -> BOOLEAN;
OPERATOR CalculaSentido;
   FPAR PisoActual TipoPiso, PisoDestino TipoPiso;
   RETURNS Sentido TipoSentido;
   START;
     DECISION PisoActual - PisoDestino;
       (< 0):
         Sentido := Subir;
       (> 0):
         Sentido := Descer;
       (= 0):
         Sentido := Nenhum;
     ENDDECISION;
   RETURN;
 ENDOPERATOR;
AXIOMS
   PrimeiroPiso (Subir) == 1;
   UltimoPiso (Subir) == NP;
   PrimeiroPiso (Descer) == NP;
   UltimoPiso (Descer) == 1;
   FOR ALL p,p1,p2 IN TipoPiso (
      FOR ALL s IN TipoSentido (
        Anterior (p,s) == p - Iterador(s);
        Seguinte (p,s) == p + Iterador(s);
        Antes (p1,p2,s) ==
          IF s = Subir THEN p1 < p2
            ELSE IF s = Descer THEN p1 > p2
              ELSE ERROR! FI FI;
        Depois (p1,p2,s) ==
          IF s = Subir THEN p1 > p2
            ELSE IF s = Descer THEN p1 < p2
              ELSE ERROR! FI FI; ); );
ENDNEWTYPE;

NEWTYPE TipoSentido
 LITERALS Nenhum, Subir, Descer;
 OPERATORS
   Iterador : TipoSentido -> INTEGER;
   SI : TipoSentido -> TipoSentido;
 AXIOMS
   Iterador (Subir) == 1;
   Iterador (Descer) == -1;
/*   Iterador (Nenhum) == ERROR!;      nao suportao pelo OG */
   SI (Subir) == Descer;
   SI (Descer) == Subir;
/*   SI (Nenhum) == ERROR!     nao suportado pelo OG */
ENDNEWTYPE;

NEWTYPE TipoAlarme
 LITERALS ErroIniPC, ErroIniPP, ErroIniE, ErroIniPE,
   PPNaoAbre, PPNaoFecha, PENaoAbre, PENaoFecha,
   PPNaoConFechada, PPNaoConAberta, RupturaCabo;
 OPERATORS
   Descricao: TipoAlarme -> CHARSTRING;
 AXIOMS
   Descricao (ErroIniPC) == 'Painel de Chamada com Numero de Piso Repetido';
   Descricao (ErroIniPP) == 'Porta do Piso com Numero de Piso Repetido';
   Descricao (ErroIniE) == 'Elevador com Numero de Elevador Repetido';
   Descricao (ErroIniPE) == 'Porta do Elevador com Numero de Elevador Repetido';
   Descricao (PPNaoAbre) == 'Porta do Piso Nao Abre';
   Descricao (PPNaoFecha) == 'Porta do Piso Nao Fecha';
   Descricao (PENaoAbre) == 'Porta do Elevador Nao Abre';
   Descricao (PENaoFecha) == 'Porta do Elevador Nao Fecha';
   Descricao (PPNaoConAberta) == 'Porta do Piso Nao Confirma Abertura';
   Descricao (PPNaoConFechada) == 'Porta do Piso Nao Confirma Fecho';
   Descricao(RupturaCabo) == 'Ruptura do Cabo do Elevador';
ENDNEWTYPE;

NEWTYPE Alarme
 STRUCT
   Codigo TipoAlarme;
   Elevador TipoElevador;
   Piso TipoPiso;
ENDNEWTYPE;

NEWTYPE TipoTensao
 LITERALS Correcta, Excessiva, Ruptura;
ENDNEWTYPE;

NEWTYPE TipoLuz
 LITERALS Apagada, Acesa;
ENDNEWTYPE;

NEWTYPE TipoSom
 LITERALS Nenhum, ElevadorParado, CargaExcessiva, Alarme;
ENDNEWTYPE;

NEWTYPE TipoComandoPorta
 LITERALS Abrir, Fechar;
ENDNEWTYPE;

NEWTYPE Chamada
 STRUCT
   Piso TipoPiso;
   Sentido TipoSentido;
 DEFAULT
 (. Nenhum, Nenhum .);
ENDNEWTYPE;

NEWTYPE TipoEstadoElevador
 LITERALS ForaServico, Livre, Movimento, Parado;
ENDNEWTYPE;

NEWTYPE DadosElevador
 STRUCT
   Piso TipoPiso;
   Sentido TipoSentido;
   Estado TipoEstadoElevador;
 DEFAULT
   (. Nenhum, Nenhum, ForaServico .);
ENDNEWTYPE ;

NEWTYPE TipoVoid
 LITERALS Nada;
ENDNEWTYPE;

NEWTYPE PidPorPiso
 ARRAY (TipoPiso,PID);
ENDNEWTYPE;

NEWTYPE PidPorElevador
 ARRAY(TipoElevador, PID);
 OPERATORS
   Numero :  PidPorElevador, PID -> TipoElevador;
 OPERATOR Numero;
   FPAR PidElevador PidPorElevador, Processo PID;
   RETURNS TipoElevador;
   DCL i TipoElevador := 1;
   START;
     CicloNumeroPidPorElevador:
       IF PidElevador(i) /= Processo AND i < NE THEN
         TASK i := i + 1;
         JOIN CicloNumeroPidPorElevador;
       FI;
   RETURN i;
 ENDOPERATOR;
ENDNEWTYPE;

NEWTYPE ActuadorPorta
 OPERATORS
   Abre: -> TipoVoid;
   Fecha: -> TipoVoid;
   Aberta : -> BOOLEAN;
   Fechada: -> BOOLEAN;
 AXIOMS
   Abre == Nada;
   Fecha == Nada;
   Aberta == TRUE;
   Fechada == TRUE;
ENDNEWTYPE;

/* nao suportado pelo OG
GENERATOR BotaoLuminoso (TYPE TipoId)
 STRUCT
   Luz  TipoLuz;
 OPERATORS
   ComandaLuz :  TipoId, TipoLuz -> TipoVoid;
 AXIOMS
   FOR ALL t IN TipoId ( FOR ALL l IN TipoLuz (
     ComandaLuz (t,l) == Nada; ); );
ENDGENERATOR BotaoLuminoso;*/
```

Figura A.4: Declaração de Tipos ao Nível do SES.

A.2 Bloco Central

As figuras seguintes constituem a descrição do bloco `Central` da hierarquia do `SES`.

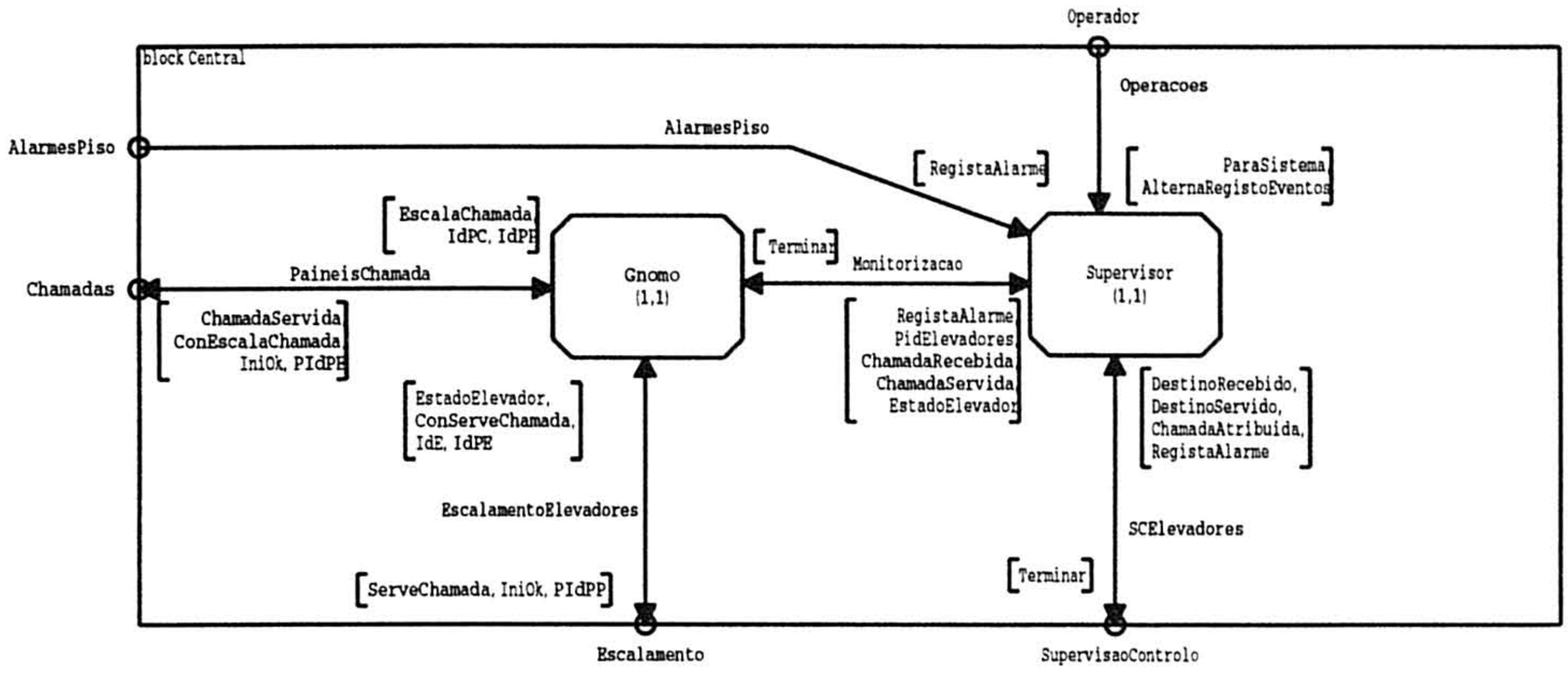


Figura A.5: Diagrama de Interligação do Bloco Central.

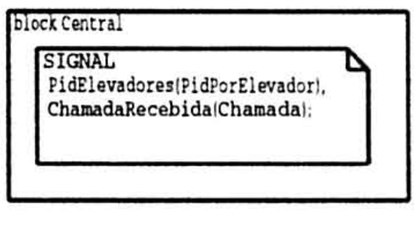


Figura A.6: Declaração de Sinais ao Nível do Bloco Central.

A.3 Bloco Acesso do Piso

A figura seguinte constitui a descrição do tipo de bloco `AcessoPiso` da hierarquia do SES.

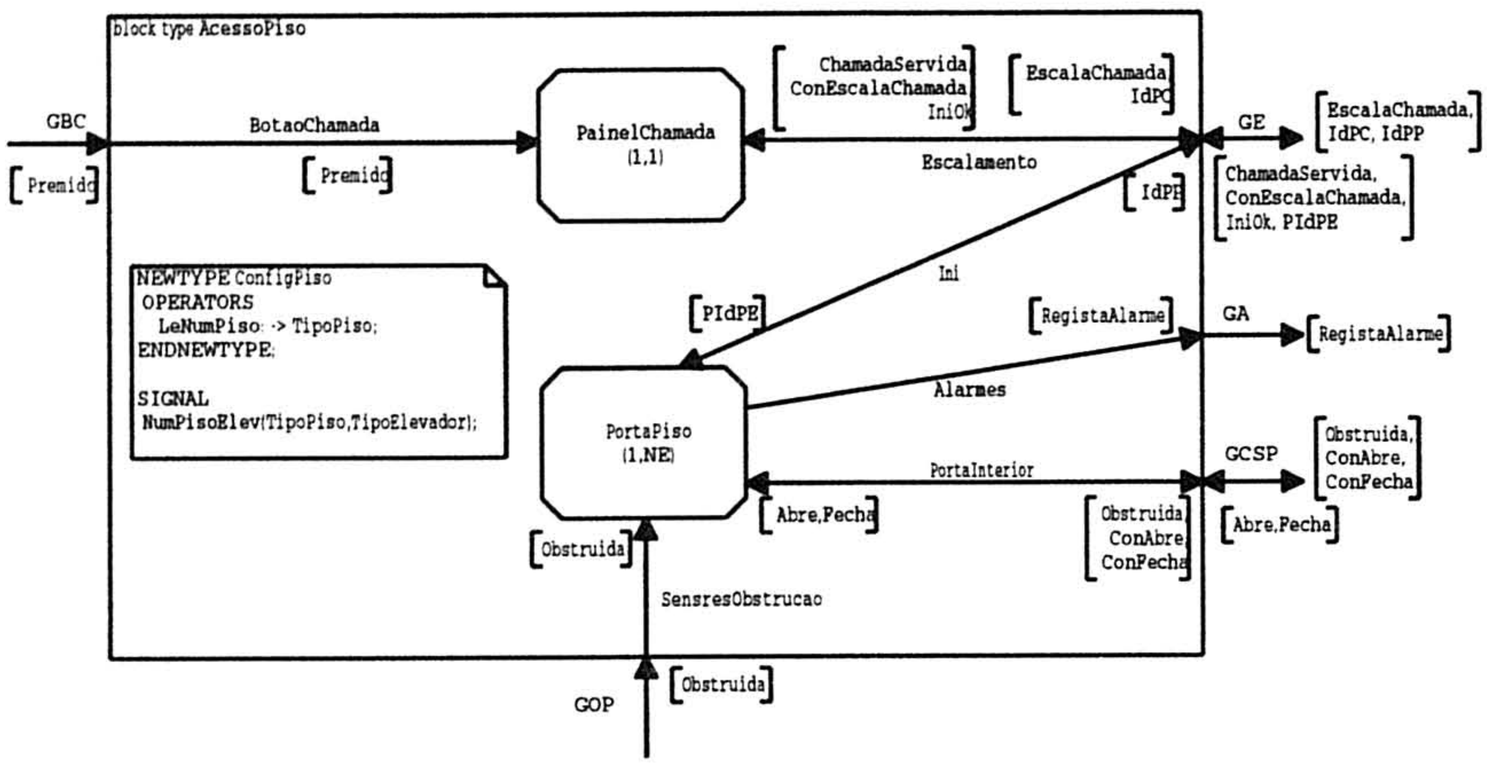


Figura A.7: Diagrama de Interligação do Bloco Acesso do Piso.

A.4 Bloco Elevador

As figuras seguintes constituem a descrição do tipo de bloco `Elevador` da hierarquia do `SES`.

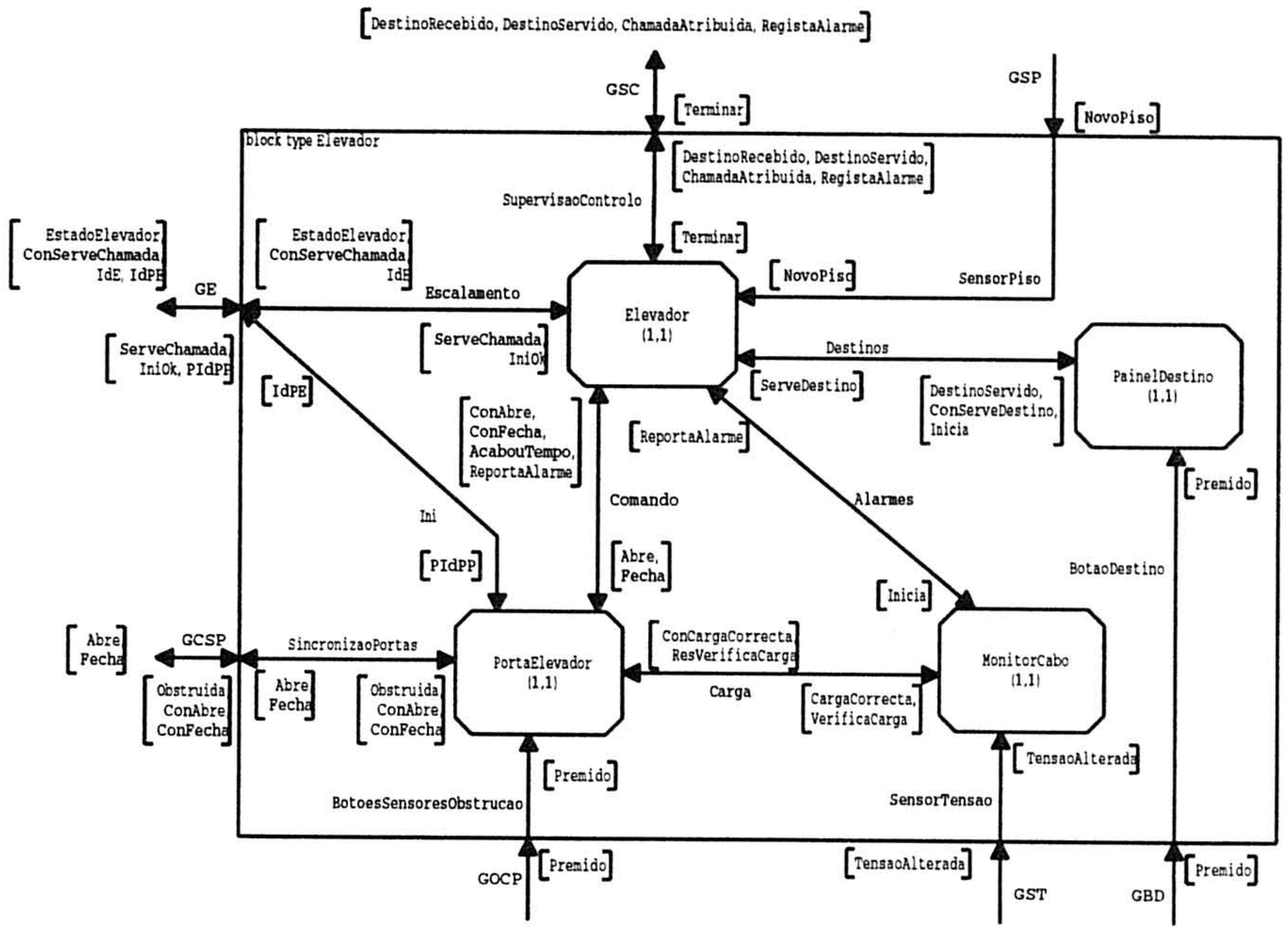


Figura A.8: Diagrama de Interligação do Bloco Elevador

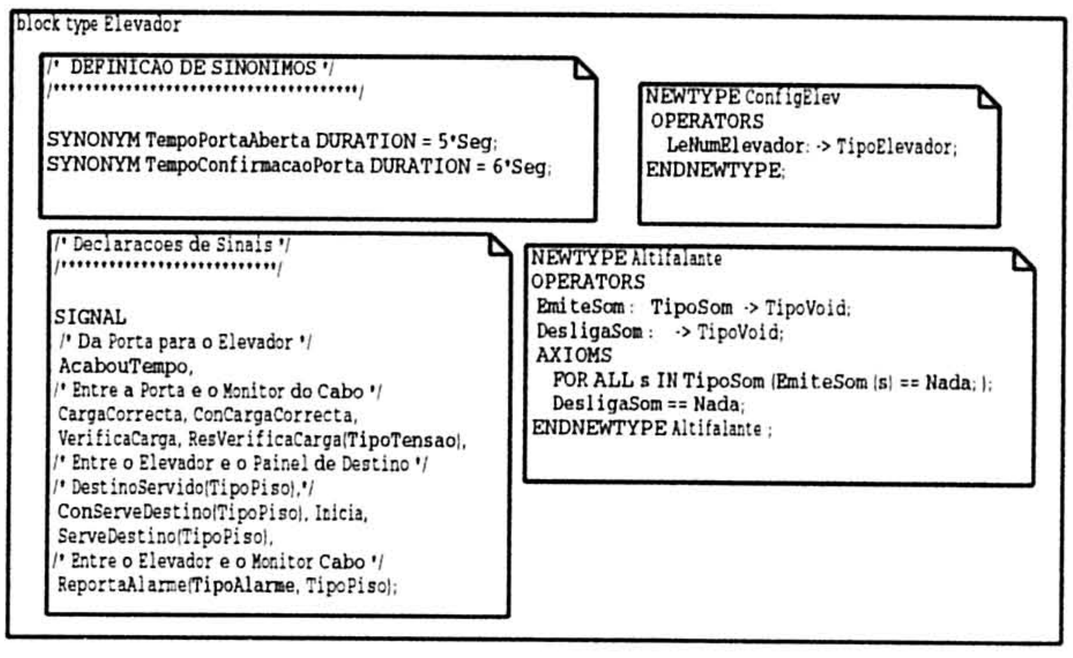


Figura A.9: Declarações PR ao Nível do Bloco Elevador.

A.5 Processo Gnomo

As figuras seguintes constituem, em conjunto, a especificação SDL do processo Gnomo.

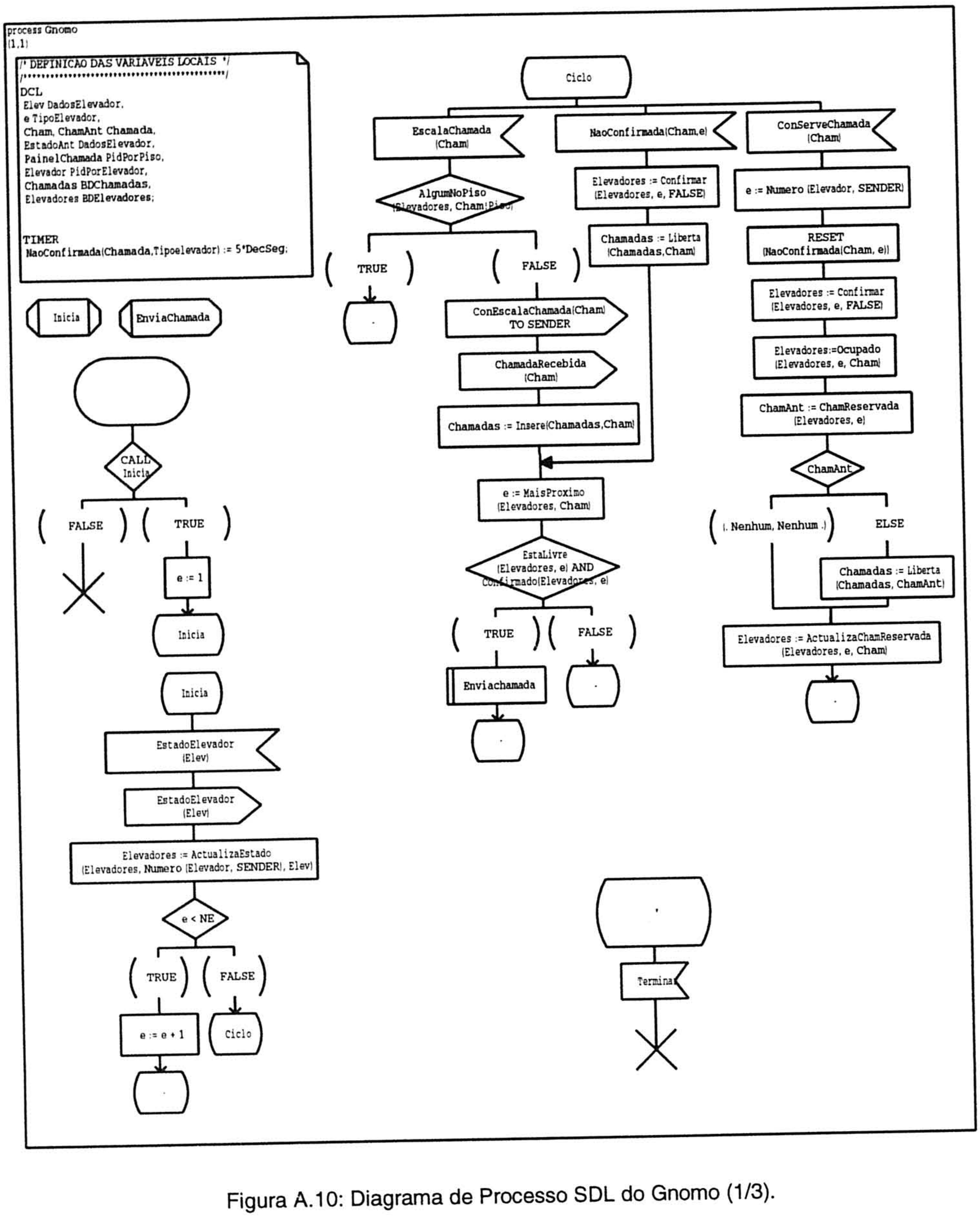


Figura A.10: Diagrama de Processo SDL do Gnomo (1/3).

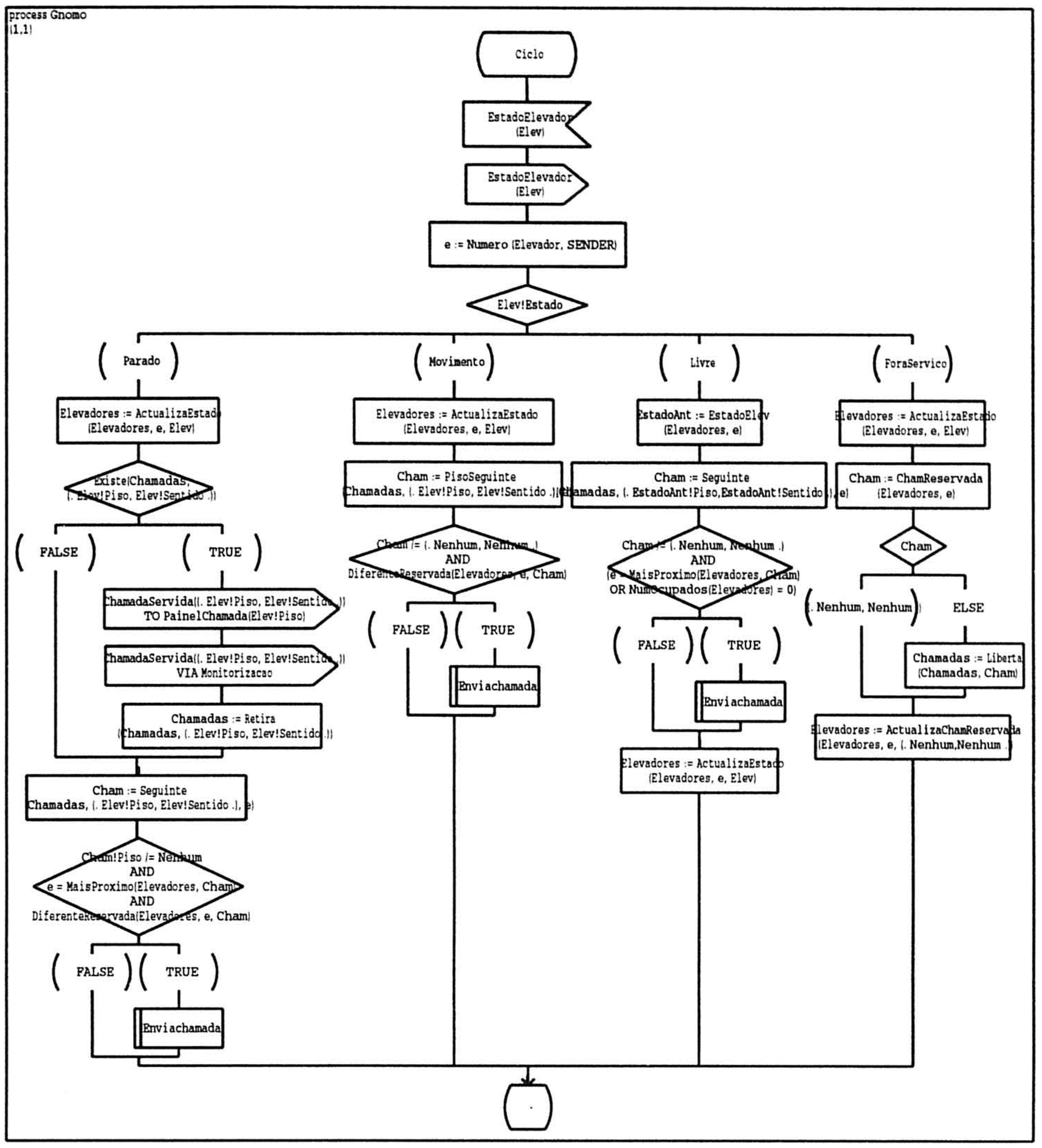


Figura A.11: Diagrama de Processo SDL do Gnomo (2/3).

```
process Gnomo
(1,1)

NEWTYPE AtributosChamada
 STRUCT
   Pendente BOOLEAN;
   Reservada TipoElevador;
 DEFAULT
   (. FALSE, Nenhum .);
ENDNEWTYPE;

NEWTYPE ChamadasNumSentido
 ARRAY (TipoPiso, AtributosChamada);
 OPERATORS
   SeguinteSentido :  ChamadasNumSentido, TipoSentido, TipoPiso, TipoPiso, TipoElevador -> Chamada;
   PisoSegSentido :  ChamadasNumSentido, TipoSentido, TipoPiso, TipoElevador -> Chamada;
ENDNEWTYPE;

NEWTYPE ArrayChamadasNumSentido
 STRUCT
   Atributos ChamadasNumSentido;
   Numero TipoPiso;
ENDNEWTYPE;

NEWTYPE BDChamadas
 ARRAY (TipoSentido, ArrayChamadasNumSentido);
 OPERATORS
   Insere :  BDChamadas, Chamada     -> BDChamadas;
   Retira :  BDChamadas, Chamada     -> BDChamadas;
   Reserva :  BDChamadas, Chamada, TipoElevador   -> BDChamadas;
   Liberta :  BDChamadas, Chamada     -> BDChamadas;
   Existe :  BDChamadas, Chamada   -> BOOLEAN;
   Reservada :  BDChamadas, Chamada   -> TipoElevador;
   Seguinte :  BDChamadas, Chamada, TipoElevador   -> Chamada;
   PisoSeguinte :  BDChamadas, Chamada, TipoElevador   -> Chamada;
   Numero :  BDChamadas -> NATURAL;
ENDNEWTYPE;

NEWTYPE AtributosElevador
 STRUCT
   Estado DadosElevador;
   ConfirmarCham BOOLEAN;
   ChamadaReservada Chamada;
 DEFAULT
   (. (. Nenhum, Nenhum, ForaServico .), FALSE, (. Nenhum,Nenhum .) .);
ENDNEWTYPE;

NEWTYPE ArrayAtributosElevadores
 ARRAY (TipoElevador, AtributosElevador);
ENDNEWTYPE;

NEWTYPE BDElevadores
 STRUCT
   Elevador ArrayAtributosElevadores;
   NumLivres TipoElevador;
 OPERATORS
   Confirmar :      BDElevadores, TipoElevador, BOOLEAN -> BDElevadores; /* Muda o estado de confirmar de recepcao de chamada de um elevador */
   Ocupado :      BDElevadores, TipoElevador, Chamada -> BDElevadores;  /* Coloca o elevador como ocupado */
   Livre :      BDElevadores, TipoElevador, TipoPiso -> BDElevadores;
   ActualizaEstado :     BDElevadores, TipoElevador, DadosElevador -> BDElevadores;  /* Actualiza o estado do elevador */
   ActualizaChamReservada :  BDElevadores, TipoElevador, Chamada -> BDElevadores;     /* Guarda no elevador a chamada como reservada */
   Confirmado :      BDElevadores, TipoElevador -> BOOLEAN;  /* Retorna se o elevador conformou ou nao a ultima chamada */
   EstaLivre :      BDElevadores, TipoElevador -> BOOLEAN;          /* Verifica se o elevador esta livre */
   AlgumNoPiso :      BDElevadores, TipoPiso -> BOOLEAN;          /* Verifica se ha elevadores parados no piso */
   NumOcupados :      BDElevadores -> TipoElevador;          /* Retorna o numero de elevadores ocupados */
   ChamReservada :      BDElevadores, TipoElevador -> Chamada;          /* Retorna a chamada reservada ao elevador */
   EstadoElev :      BDElevadores, TipoElevador -> DadosElevador;          /* Retorna o estado do elevador */
   MaisProximo :      BDElevadores, Chamada -> TipoElevador;          /* Retorna o elevador mais proximo da chamada */
   DiferenteReservada :     BDElevadores, TipoElevador, Chamada -> BOOLEAN;        /* Verifica se a chamada e diferente da reservada */
ENDNEWTYPE;
```

Figura A.12: Diagrama de Processo SDL do Gnomo (3/3).

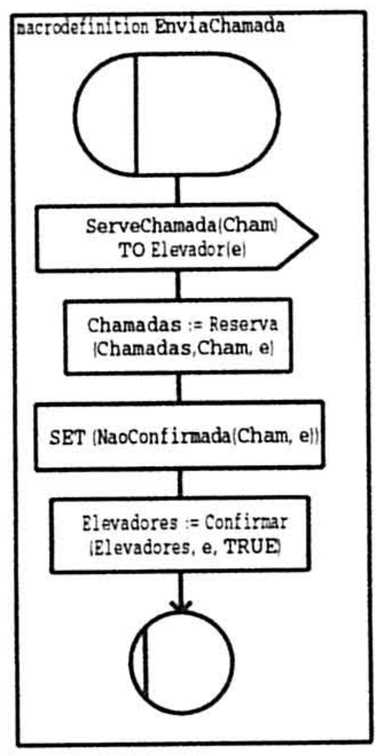


Figura A.13: Diagrama de Processo SDL do Procedimento Envia Mensagem do Gnomo.

A.6 Processo Supervisor

As figuras seguintes constituem a especificação SDL do processo `Supervisor`.

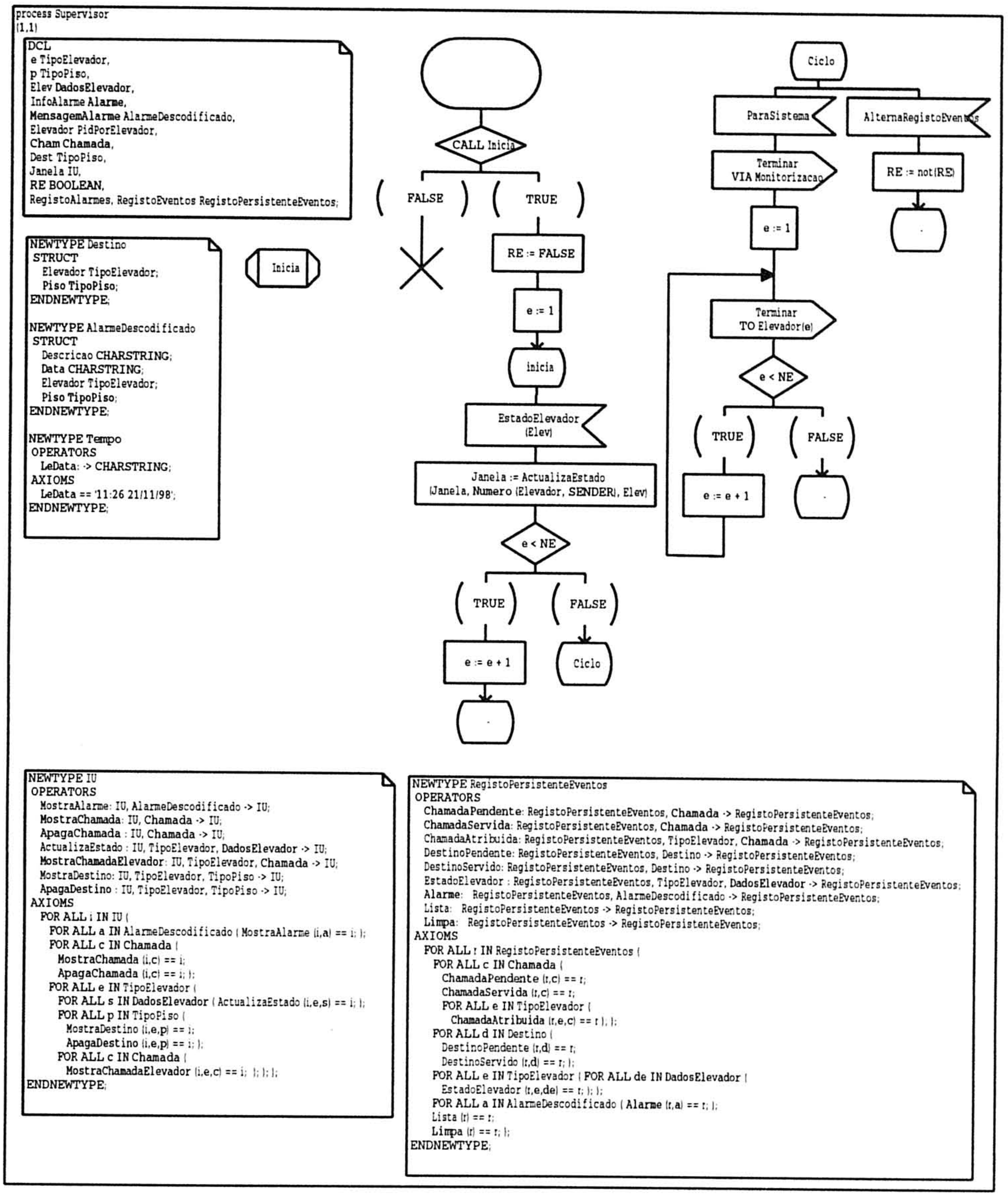


Figura A.14: Diagrama de Processo SDL do Supervisor (1/2).

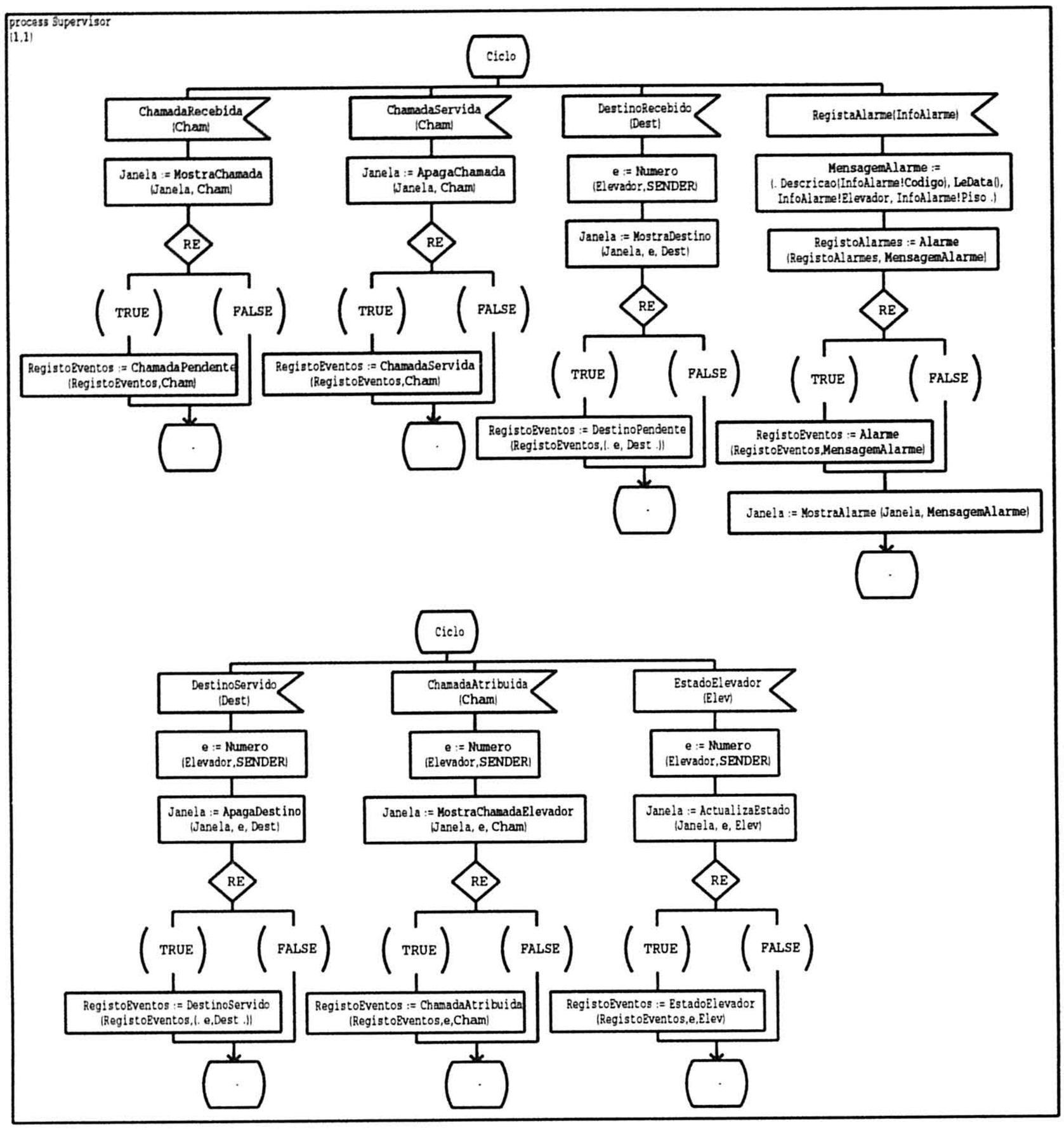


Figura A.15: Diagrama de Processo SDL do Supervisor (2/2).

A.7 Processo Painel de Chamada

A figura seguinte constitui a especificação SDL do processo `PainelChamada`.

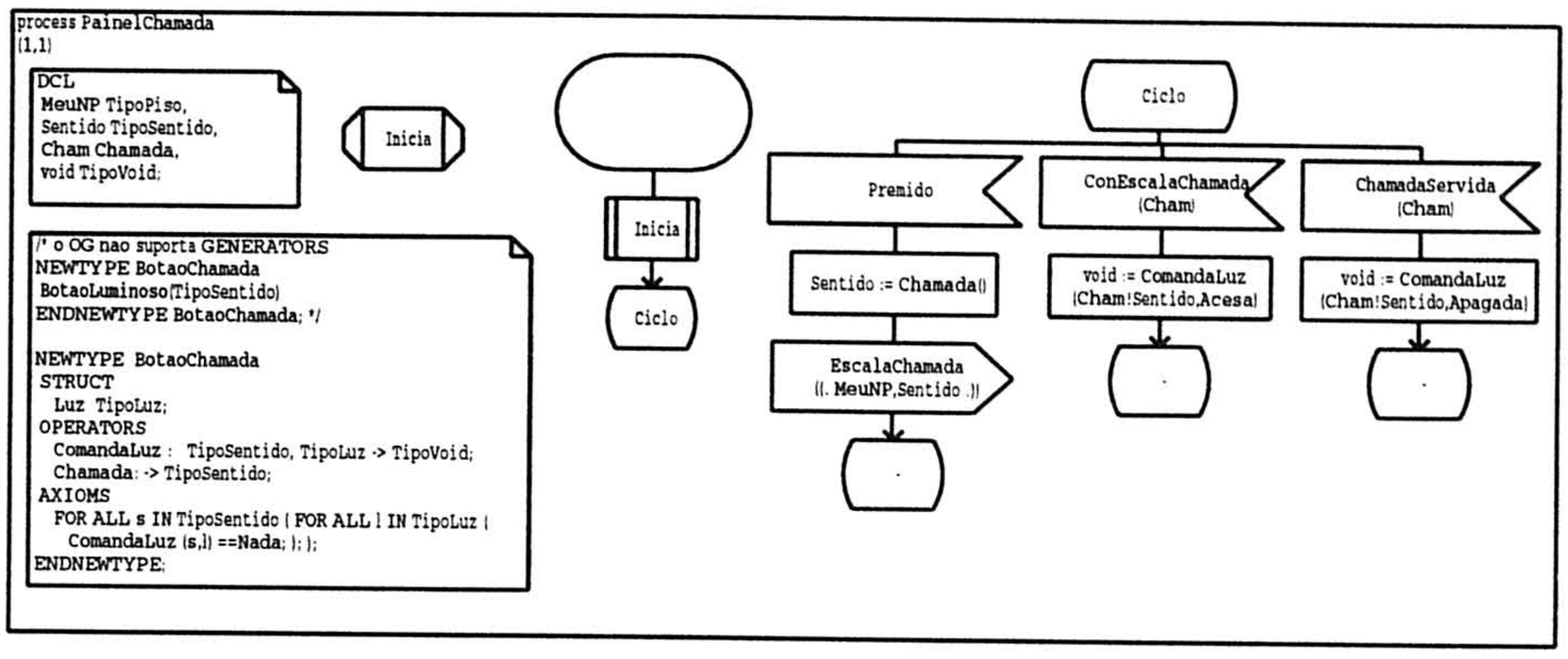


Figura A.16: Diagrama de Processo SDL do Painel de Chamada.

A.8 Processo Porta do Piso

As figuras seguintes constituem a especificação SDL do processo `PortaPiso`.

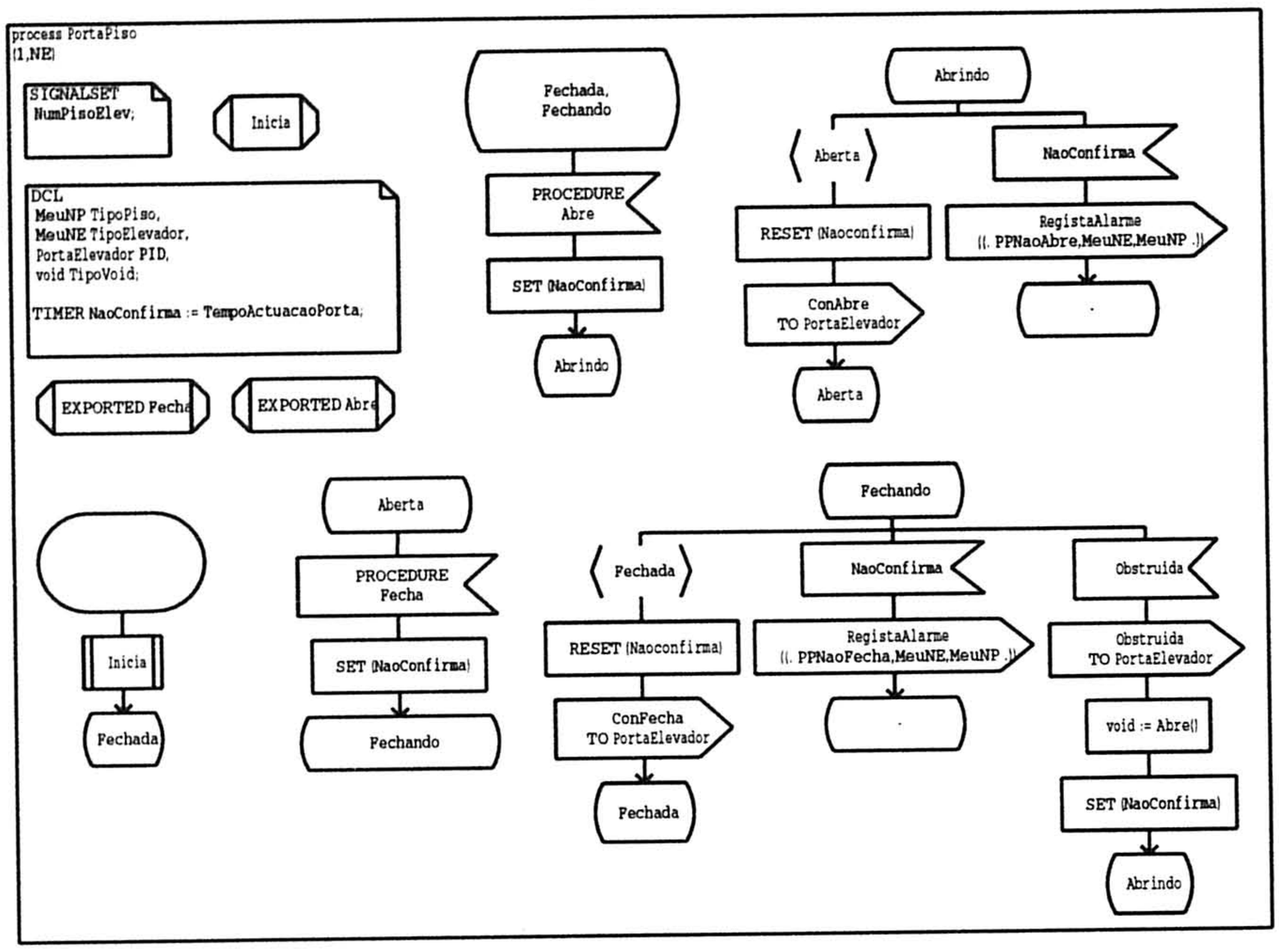


Figura A.17: Diagrama de Processo SDL da Porta do Piso.

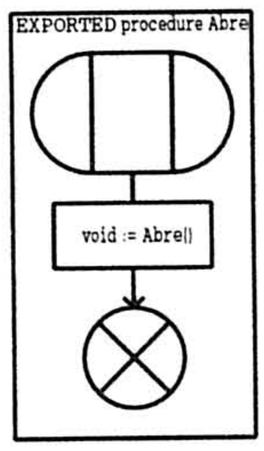


Figura A.18: Diagrama de Processo SDL do Procedimento Abre da Porta do Piso.

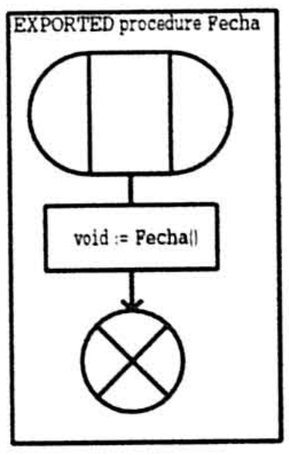


Figura A.19: Diagrama de Processo SDL do Procedimento Fecha da Porta do Piso.

A.9 Processo Elevador

As figuras seguintes constituem, em conjunto, a especificação SDL do processo `Elevador`.

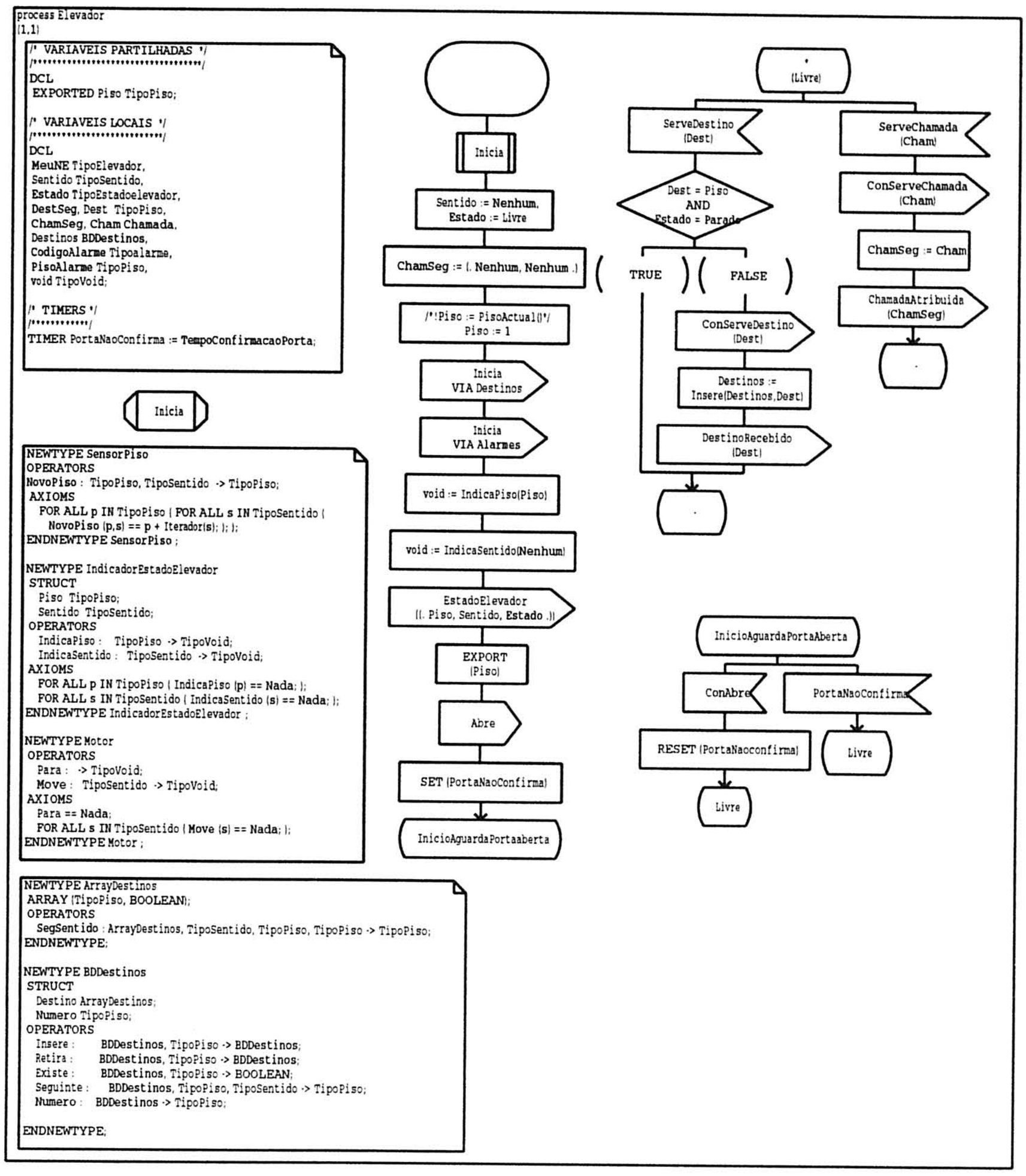


Figura A.20: Diagrama de Processo SDL do Elevador (1/5).

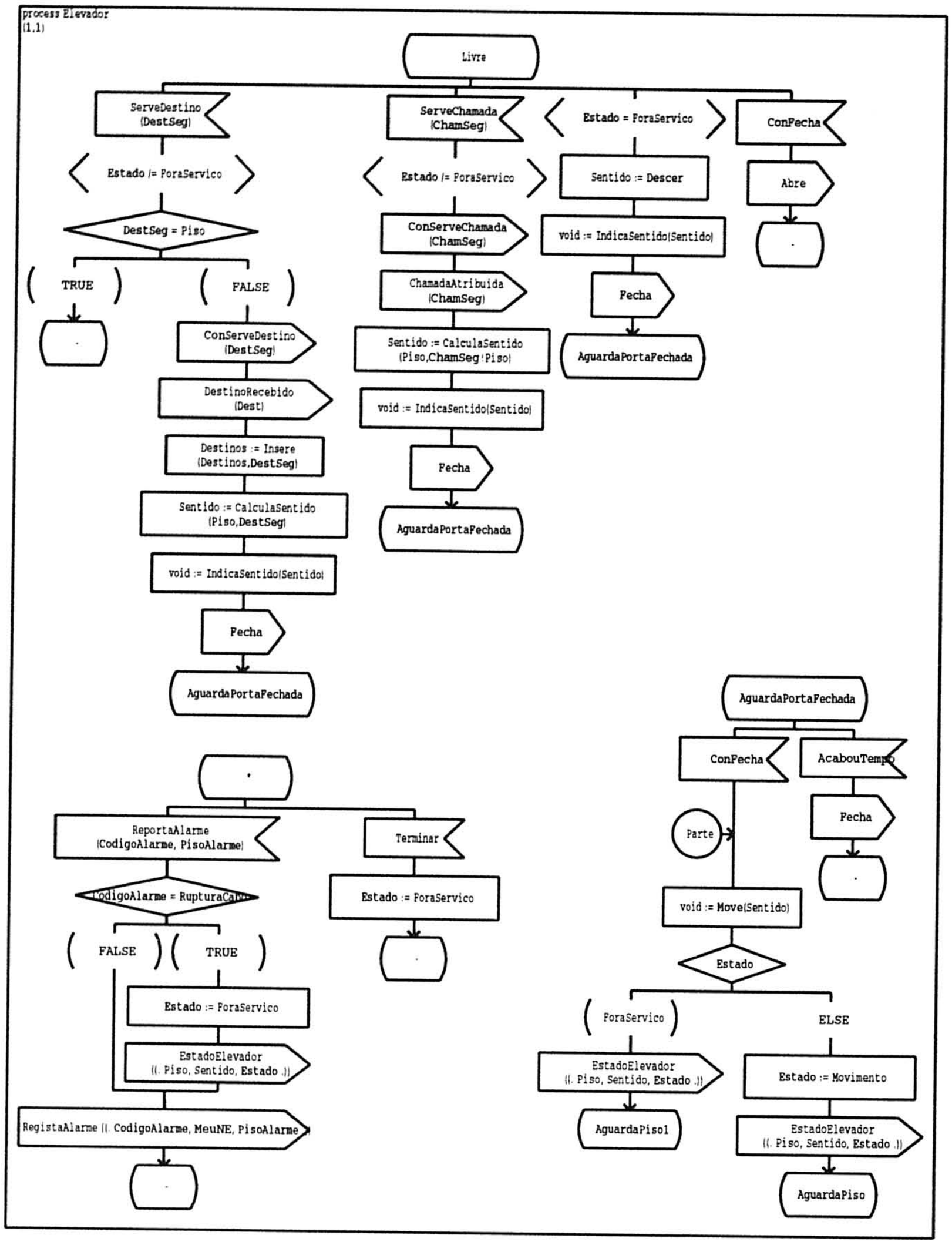


Figura A.21: Diagrama de Processo SDL do Elevador (2/5).

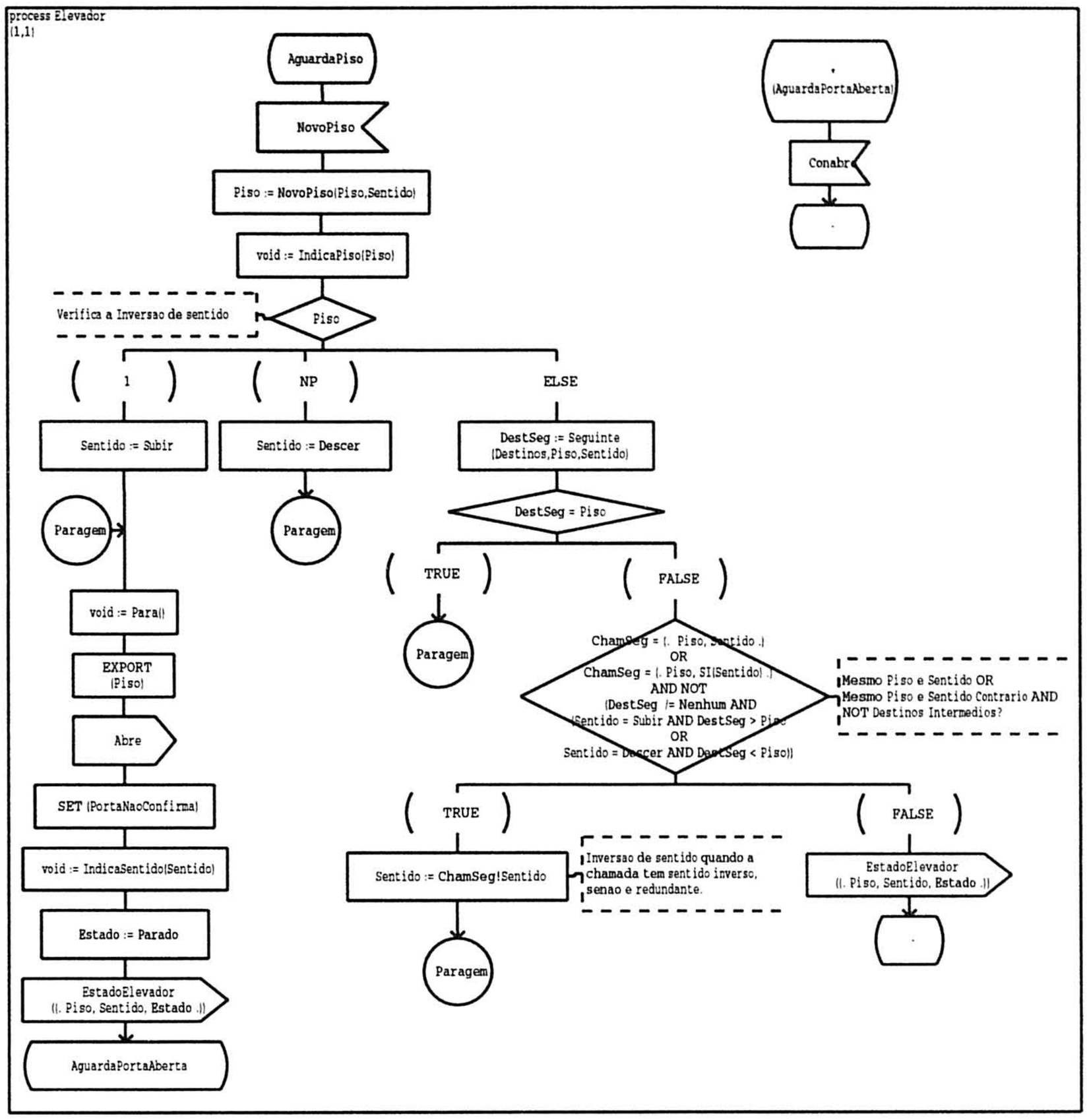


Figura A.22: Diagrama de Processo SDL do Elevador (3/5).

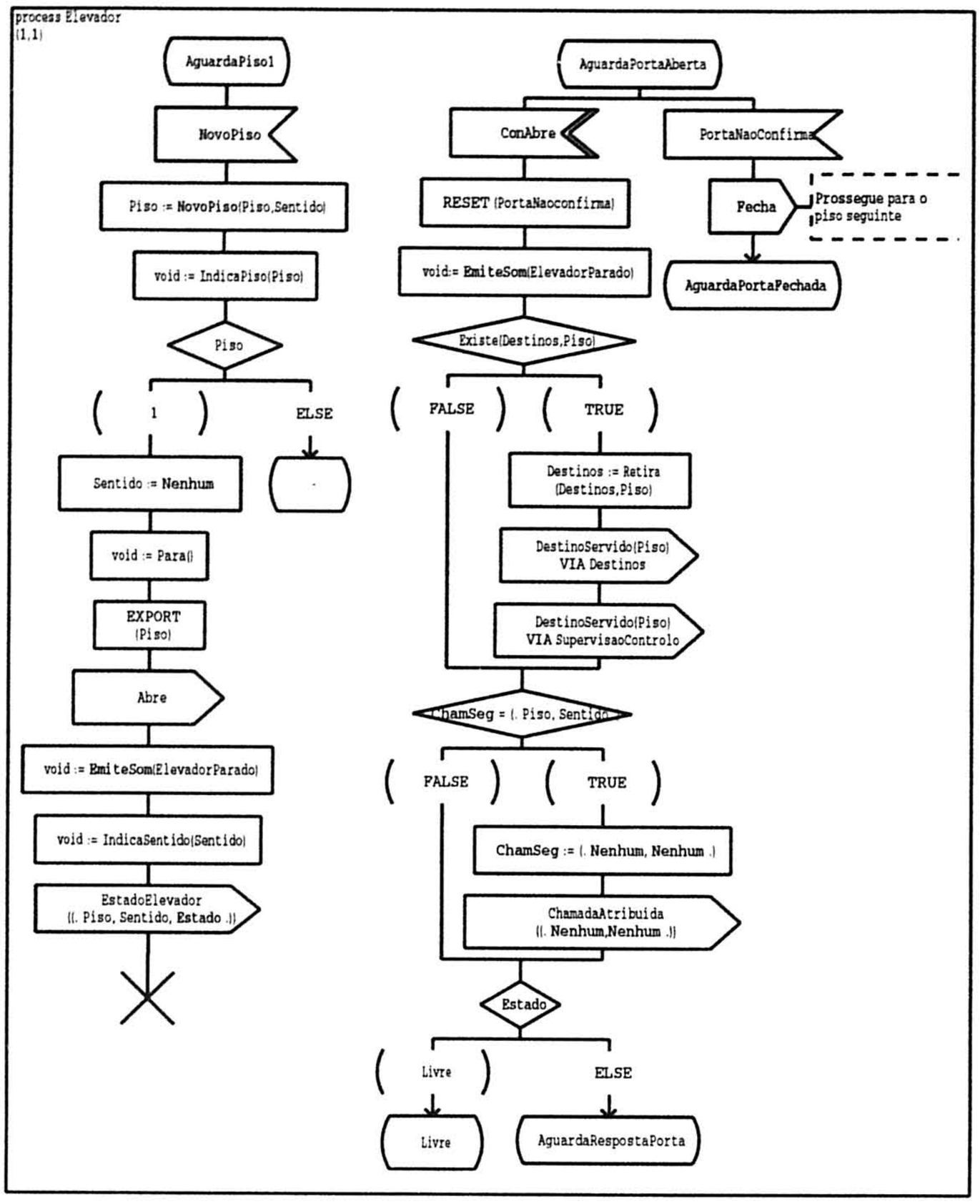


Figura A.23: Diagrama de Processo SDL do Elevador (4/5).

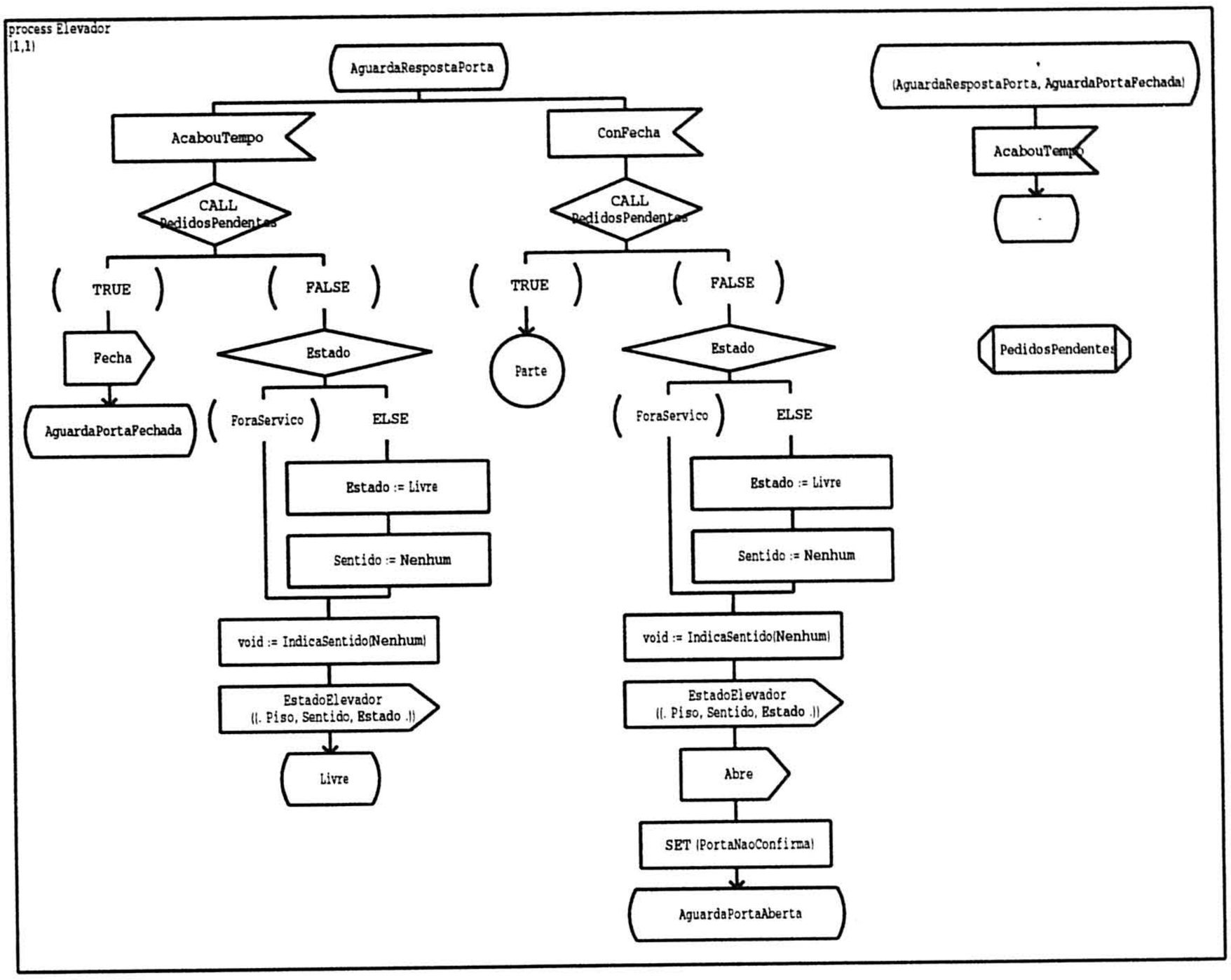


Figura A.24: Diagrama de Processo SDL do Elevador (5/5).

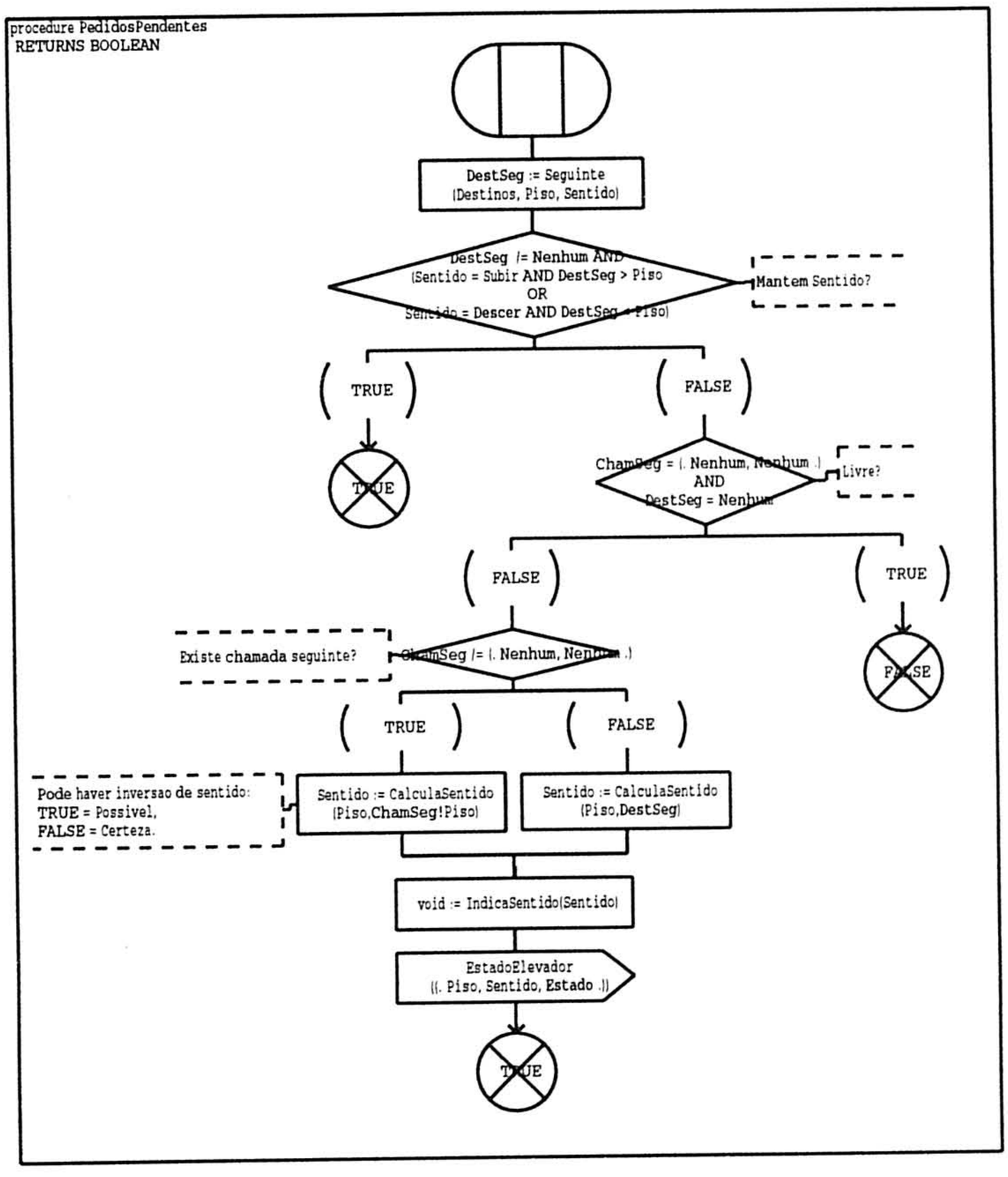


Figura A.25: Diagrama de Processo do Procedimento Pedidos Pendentes do Elevador.

A.10 Processo Painel de Destinos

As figuras seguintes constituem, em conjunto, a especificação SDL do processo `PainelDestinos`.

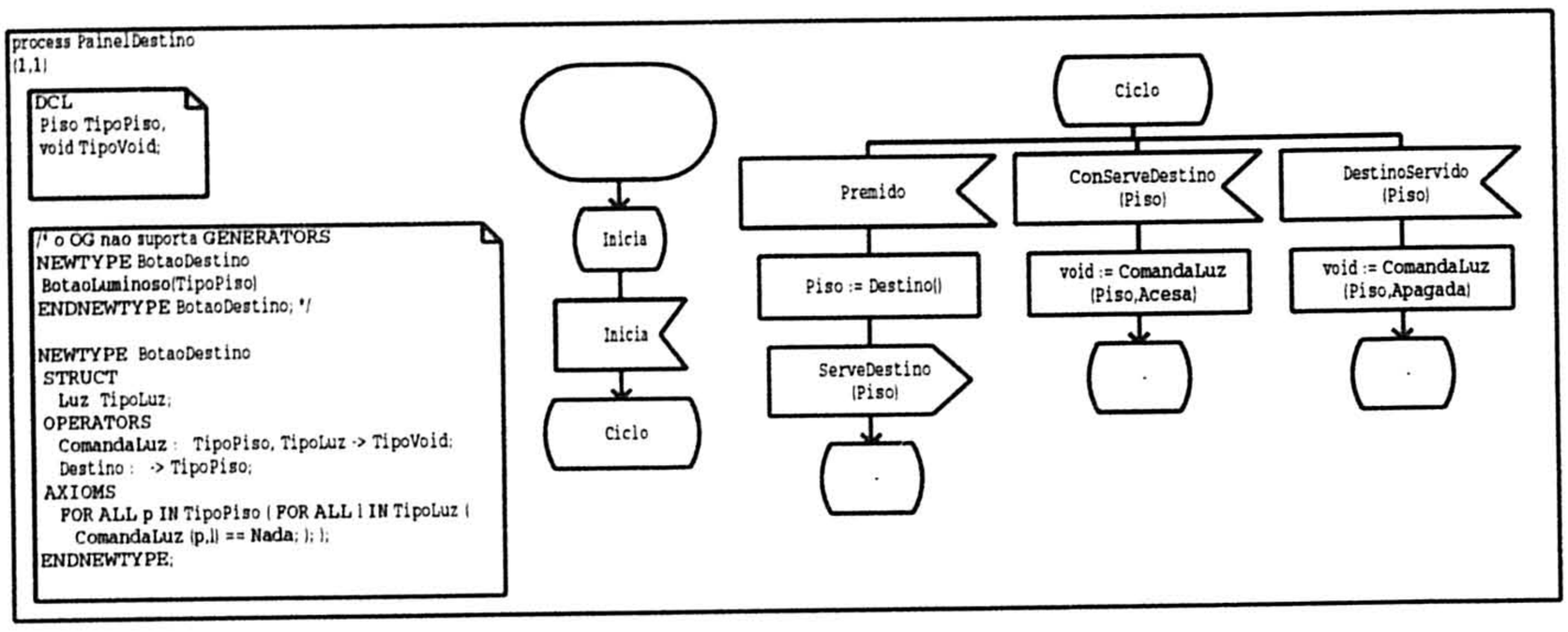


Figura A.26: Diagrama de Processo SDL do Painel de Destinos.

A.11 Processo Porta do Elevador

As figuras seguintes constituem, em conjunto, a especificação SDL do processo `PortaElevador`.

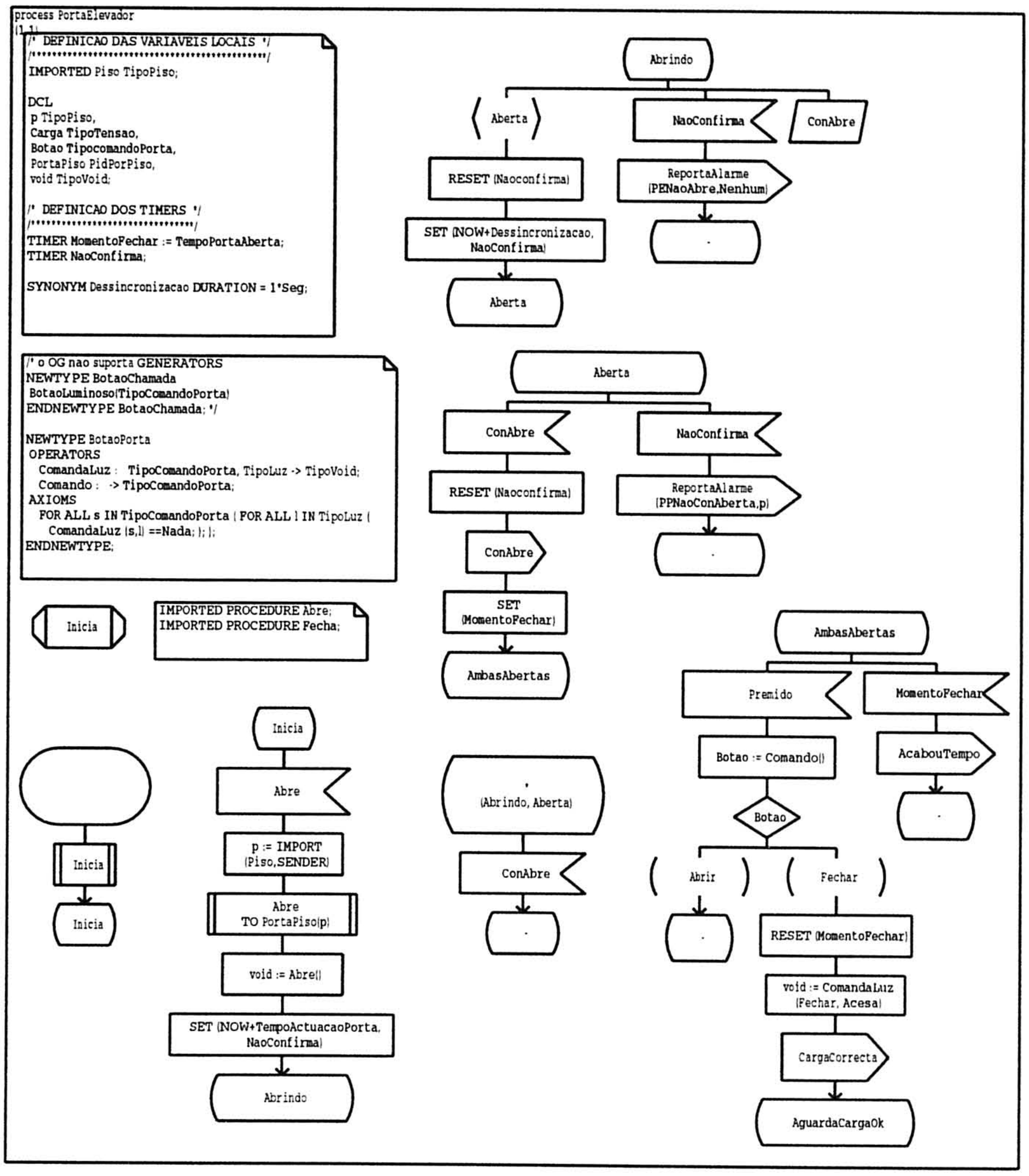


Figura A.27: Diagrama de Processo SDL da Porta do Elevador (1/2).

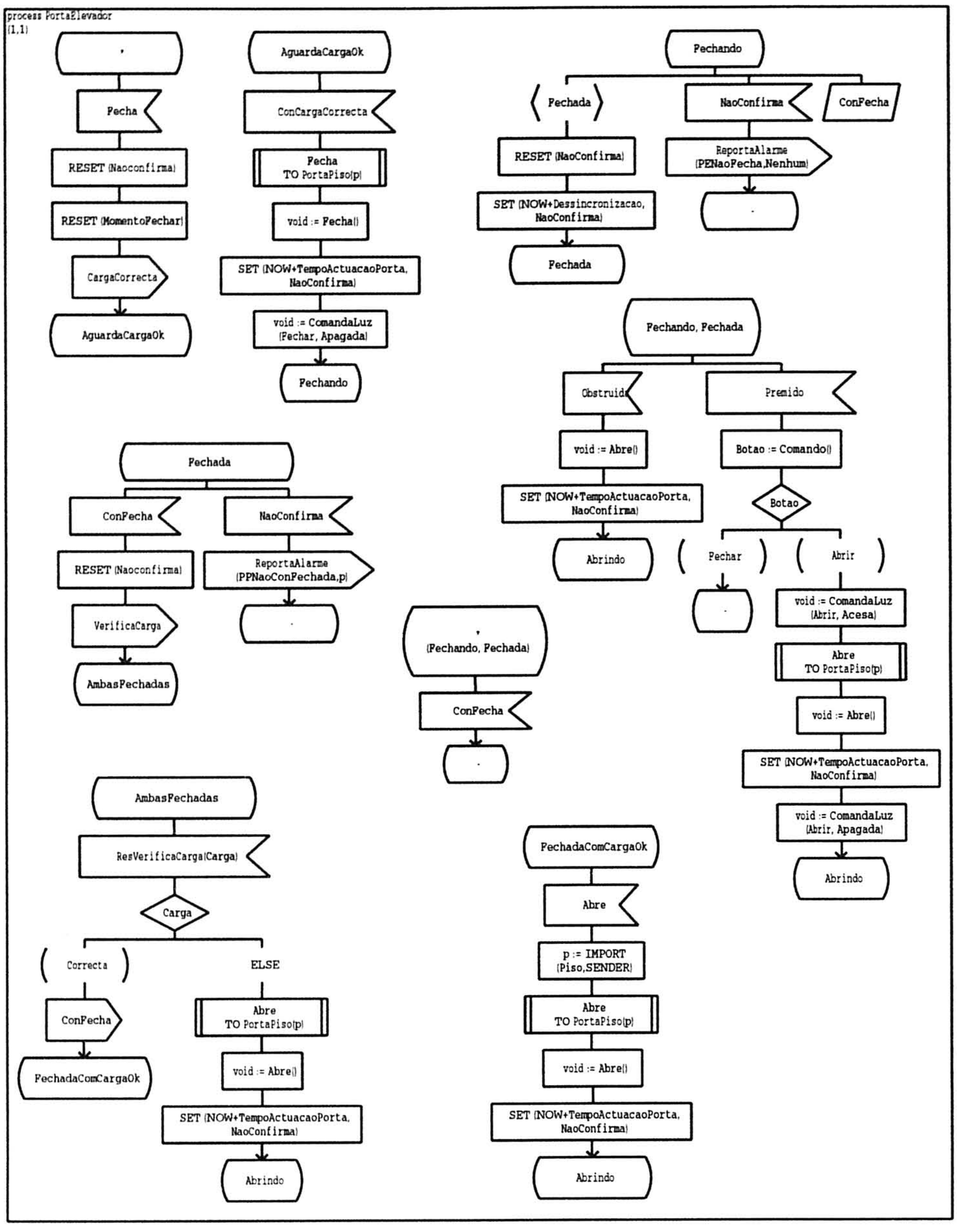


Figura A.28: Diagrama de Processo SDL da Porta do Elevador (2/2).

A.12 Processo Monitor do Cabo

A figura seguinte constitui a especificação SDL do processo `MonitorCabo`.

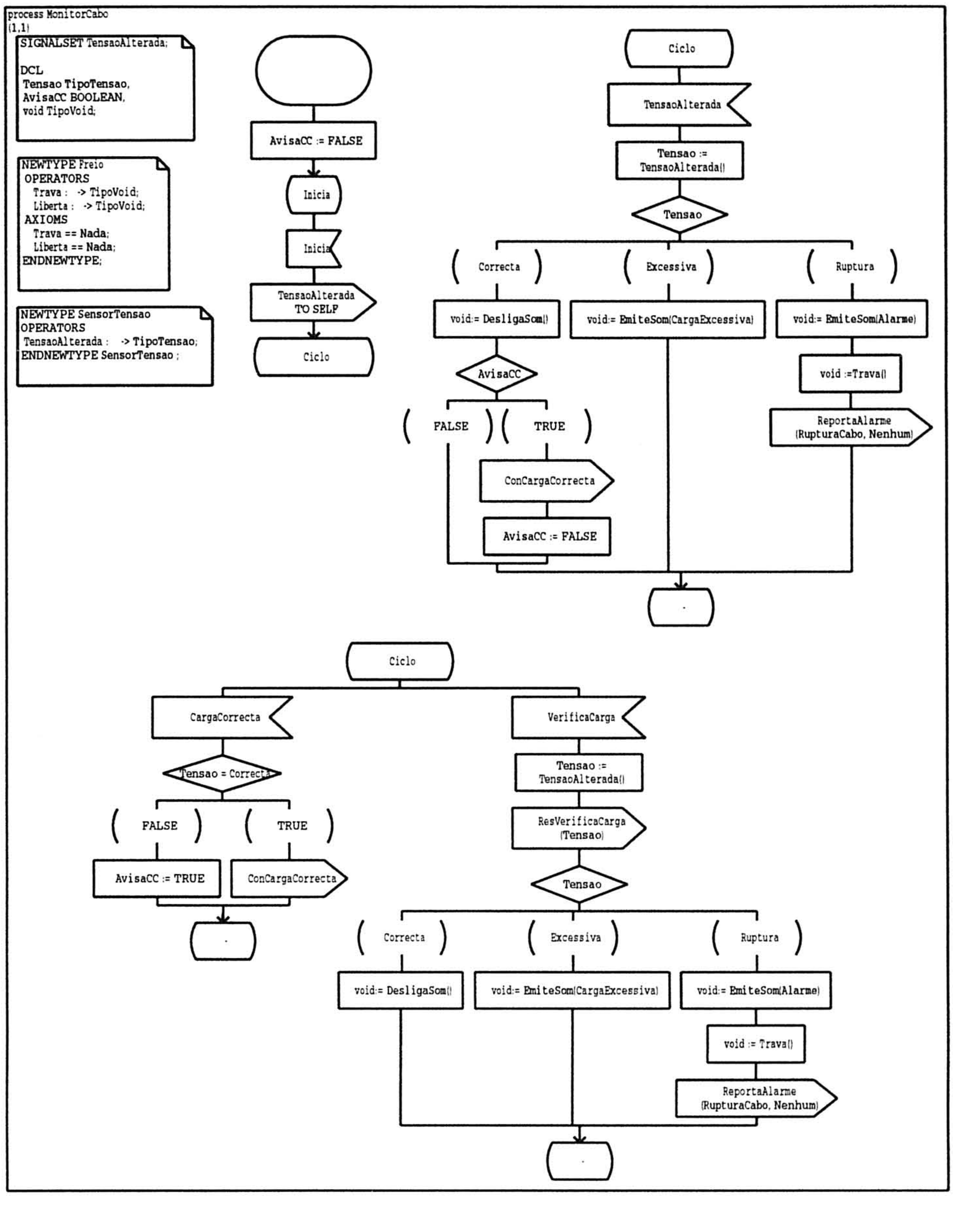
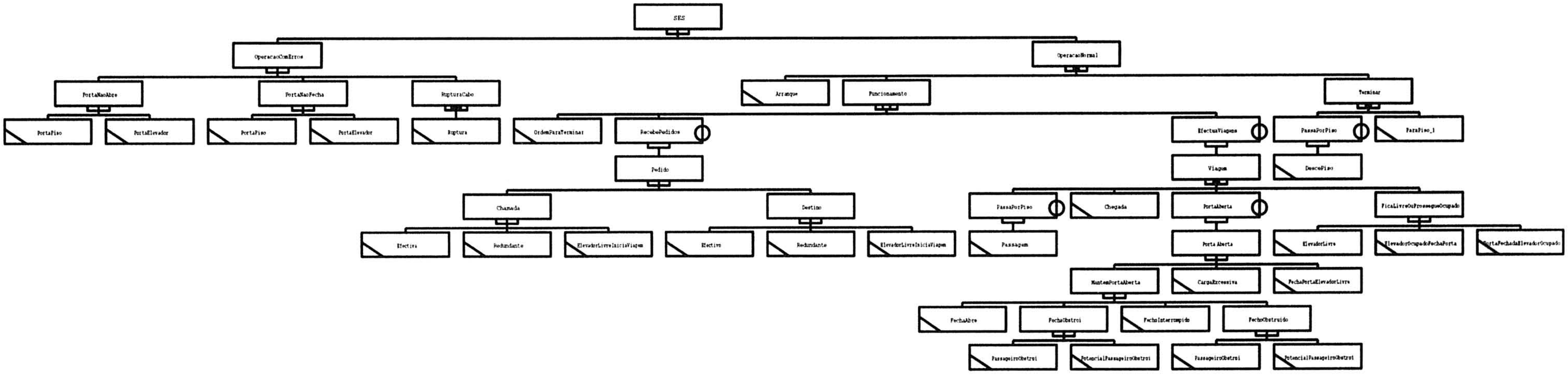
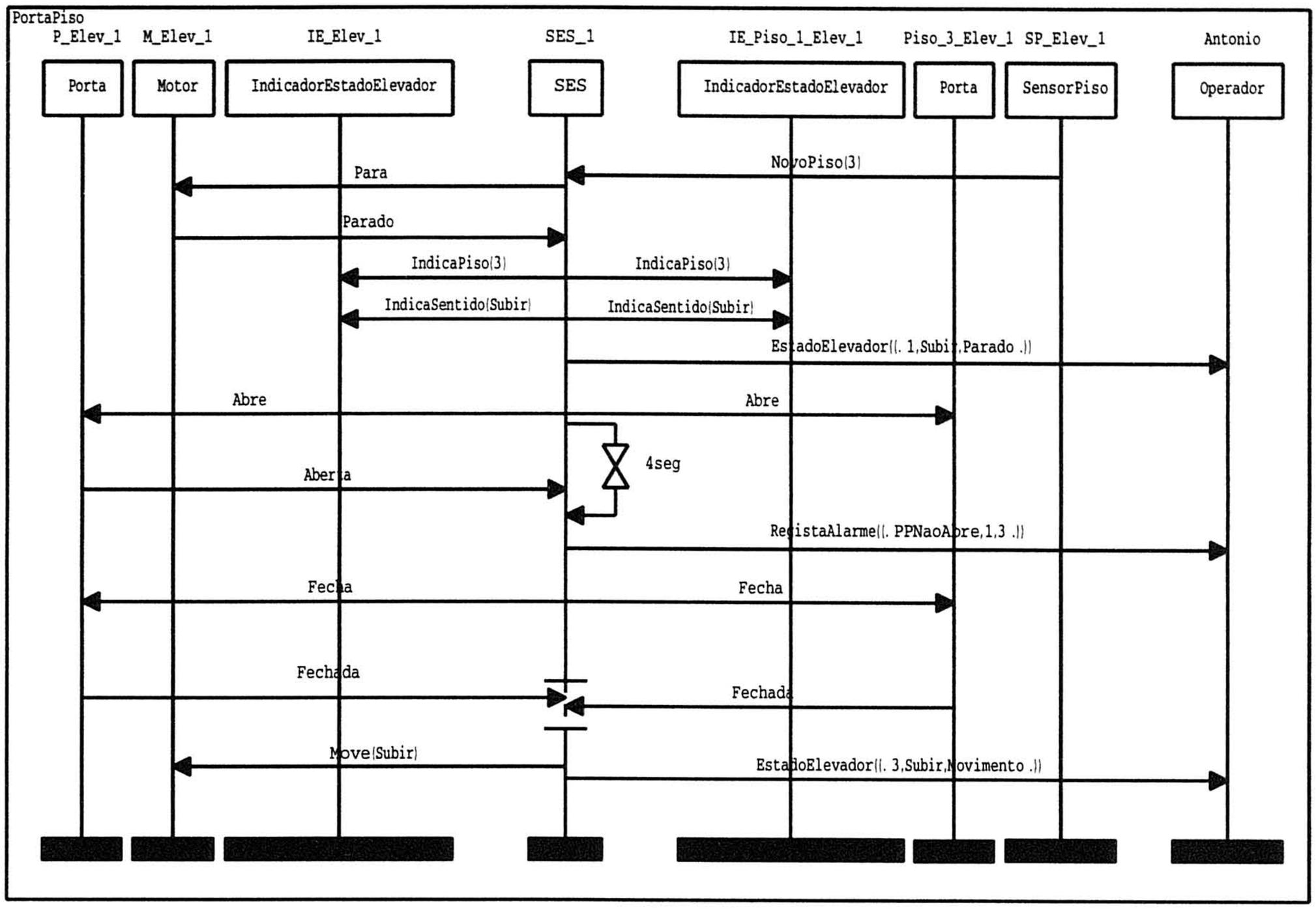


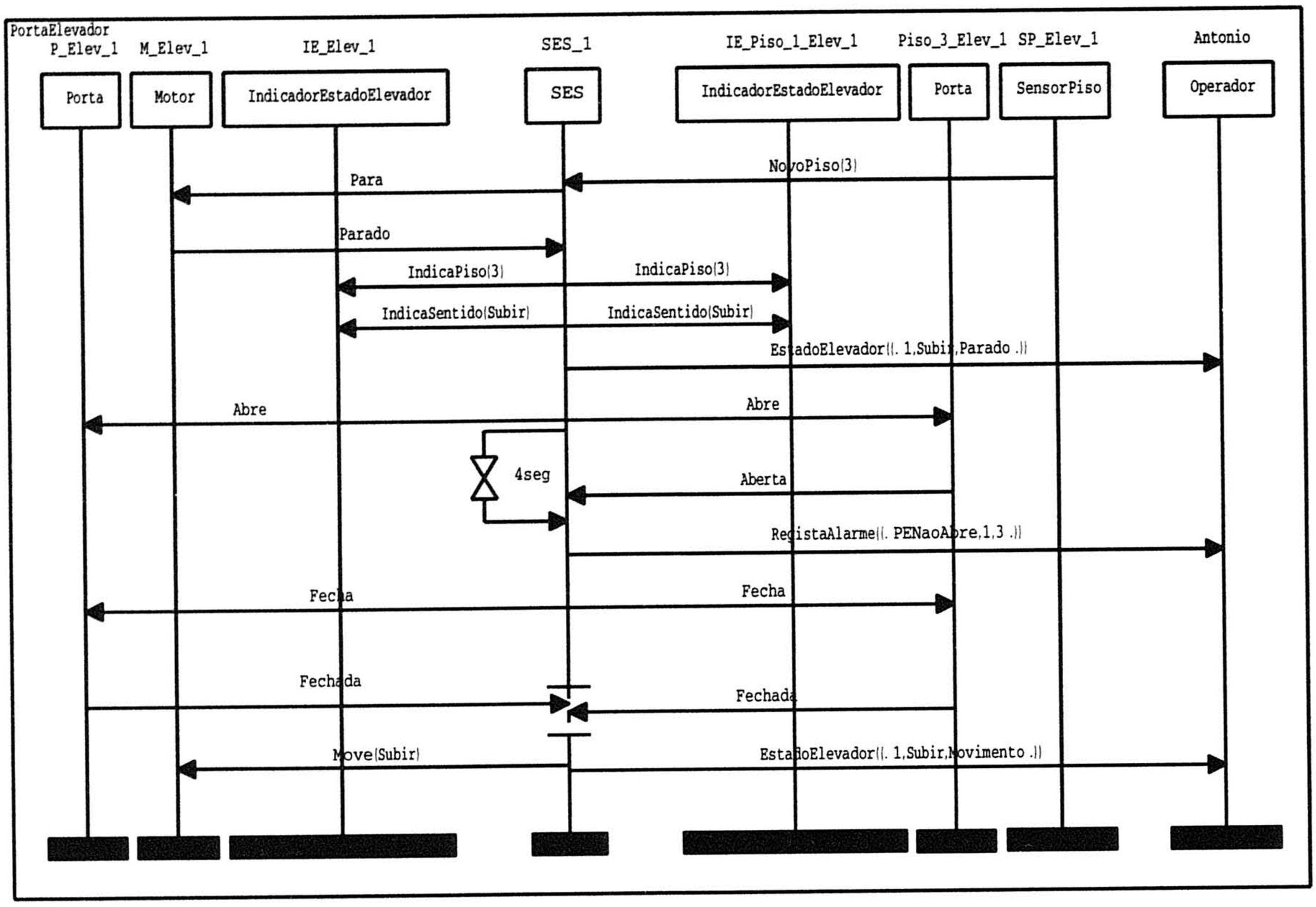
Figura A.29: Diagrama de Processo SDL do Monitor do Cabo.

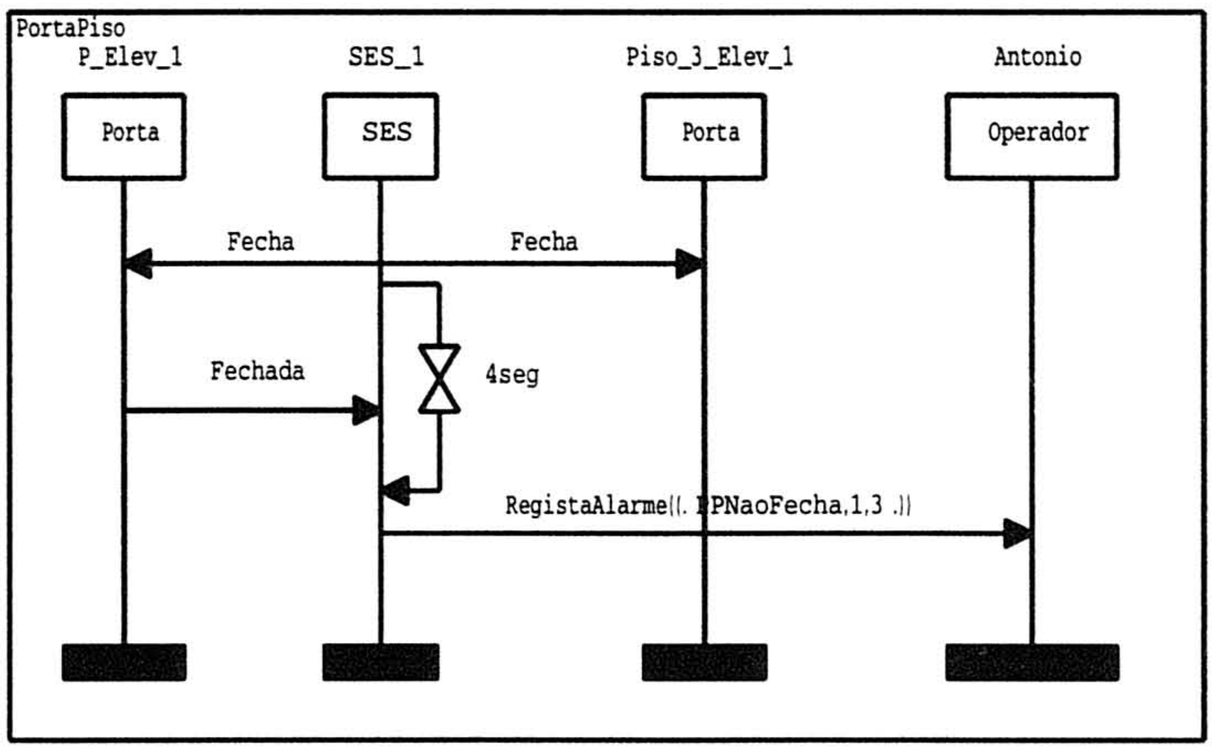
Anexo B MSCs da Modelação Dinâmica do SES

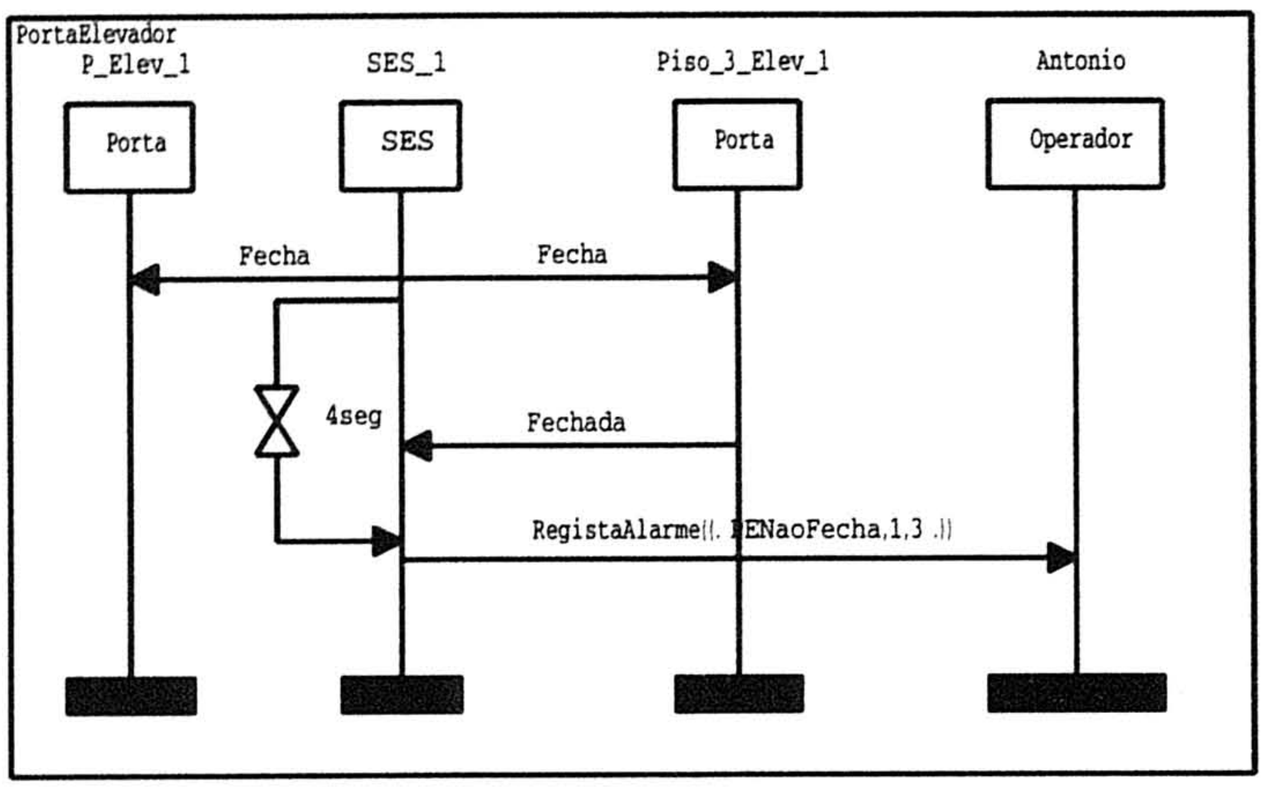
Listagem dos MSCs referentes à modelação dinâmica do SES durante a fase de análise. Os diagramas são apresentados tal como a ferramenta *Object*GEODE os fornece.

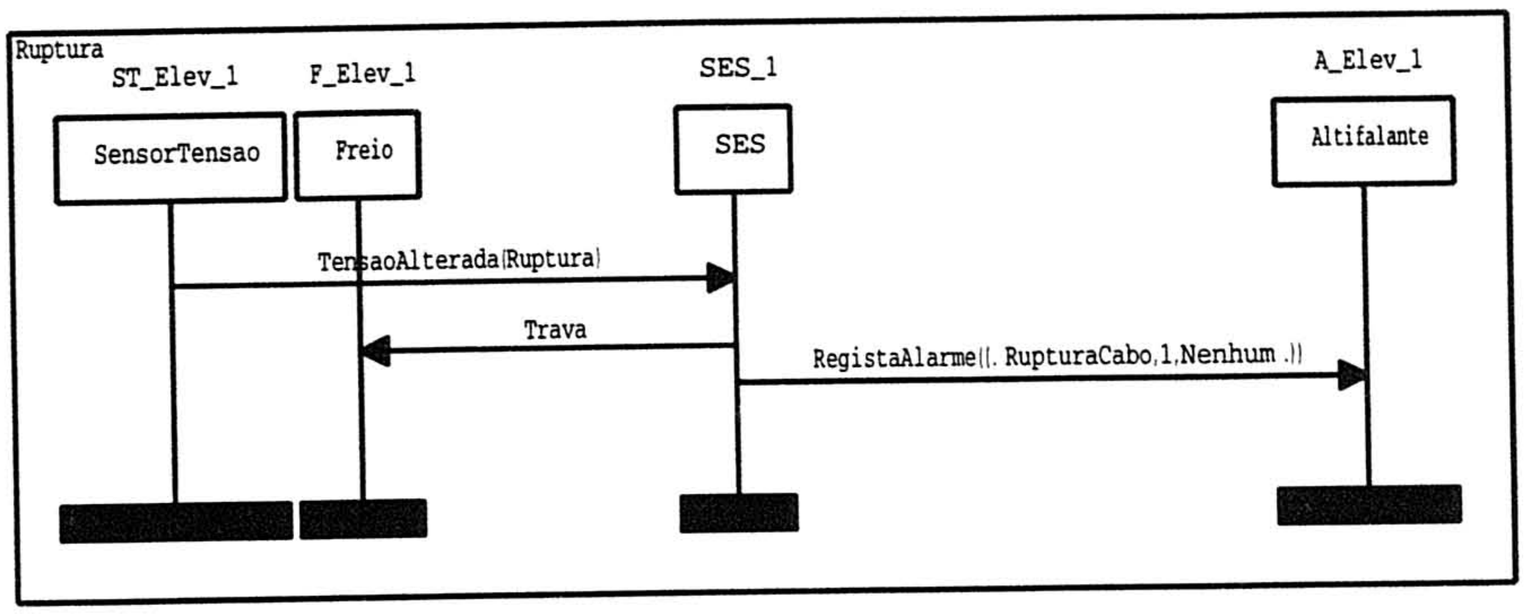

SES
OperacaoComErros
OperacaoNormal
PortaNaoAbre
PortaNaoFecha
RupturaCabo
Arranque
Funcionamento
Terminar
PortaPiso
PortaElevador
PortaPiso
PortaElevador
Ruptura
OrdemParaTerminar
RecebePedidos
EfectuaViagem
PassaPorPiso
ParaPiso_1
Pedido
Viagem
DesocePiso
Chamada
Destino
PassaPorPiso
Chegada
PortaAberta
FicaLivreOuProssegueOcupado
Efectiva
Redundante
ElevadorLivreIniciaViagem
Efectivo
Redundante
ElevadorLivreIniciaViagem
Passagem
Porta Aberta
ElevadorLivre
ElevadorOcupadoFechaPorta
PortaFechadaElevadorOcupado
MantemPortaAberta
CargaExcessiva
FechaPortaElevadorLivre
FechaAbre
FechaObstroi
FechoInterrompido
FechoObstruido
PassageiroObstroi
PotencialPassageiroObstroi
PassageiroObstroi
PotencialPassageiroObstroi

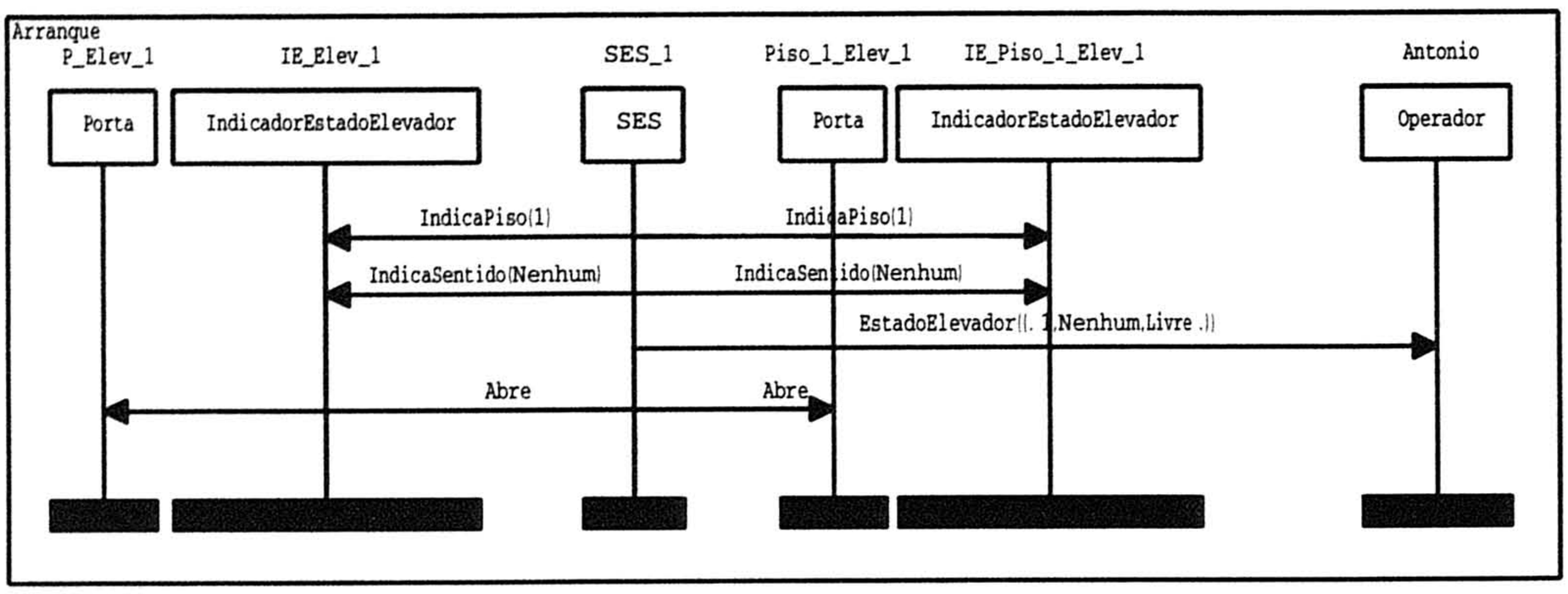

PortaPiso
P_Elev_1
M_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_3_Elev_1
SP_Elev_1
Antonio
Porta
Motor
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
SensorPiso
Operador
NovoPiso(3)
Para
Parado
IndicaPiso(3)
IndicaPiso(3)
IndicaSentido(Subir)
IndicaSentido(Subir)
EstadoElevador(|. 1,Subir,Parado .|)
Abre
Abre
4seg
Aberta
RegistaAlarme(|. PPNaoAbre,1,3 .|)
Fecha
Fecha
Fechada
Fechada
Move(Subir)
EstadoElevador(|. 3,Subir,Movimento .|)

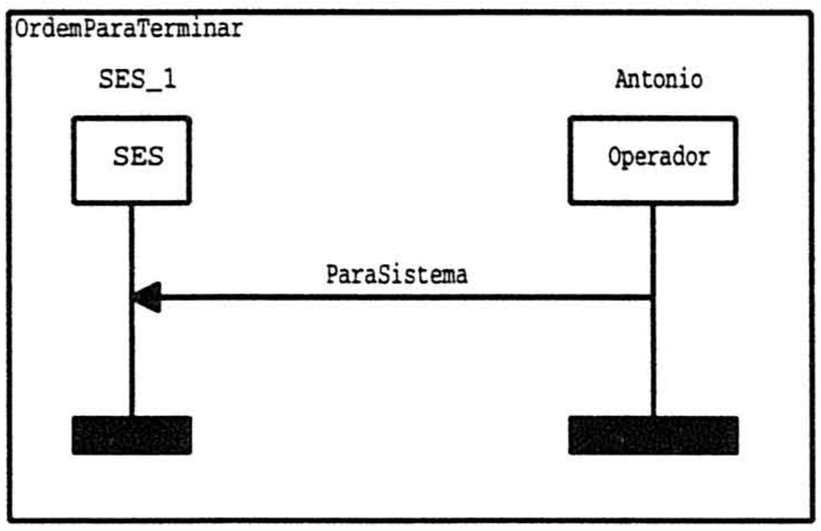

PortaElevador
P_Elev_1
M_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_3_Elev_1
SP_Elev_1
Antonio
Porta
Motor
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
SensorPiso
Operador
NovoPiso(3)
Para
Parado
IndicaPiso(3)
IndicaPiso(3)
IndicaSentido(Subir)
IndicaSentido(Subir)
EstadoElevador(|. 1,Subir,Parado .|)
Abre
Abre
4seg
Aberta
RegistaAlarme(|. PENaoAbre,1,3 .|)
Fecha
Fecha
Fechada
Fechada
Move(Subir)
EstadoElevador(|. 1,Subir,Movimento .|)

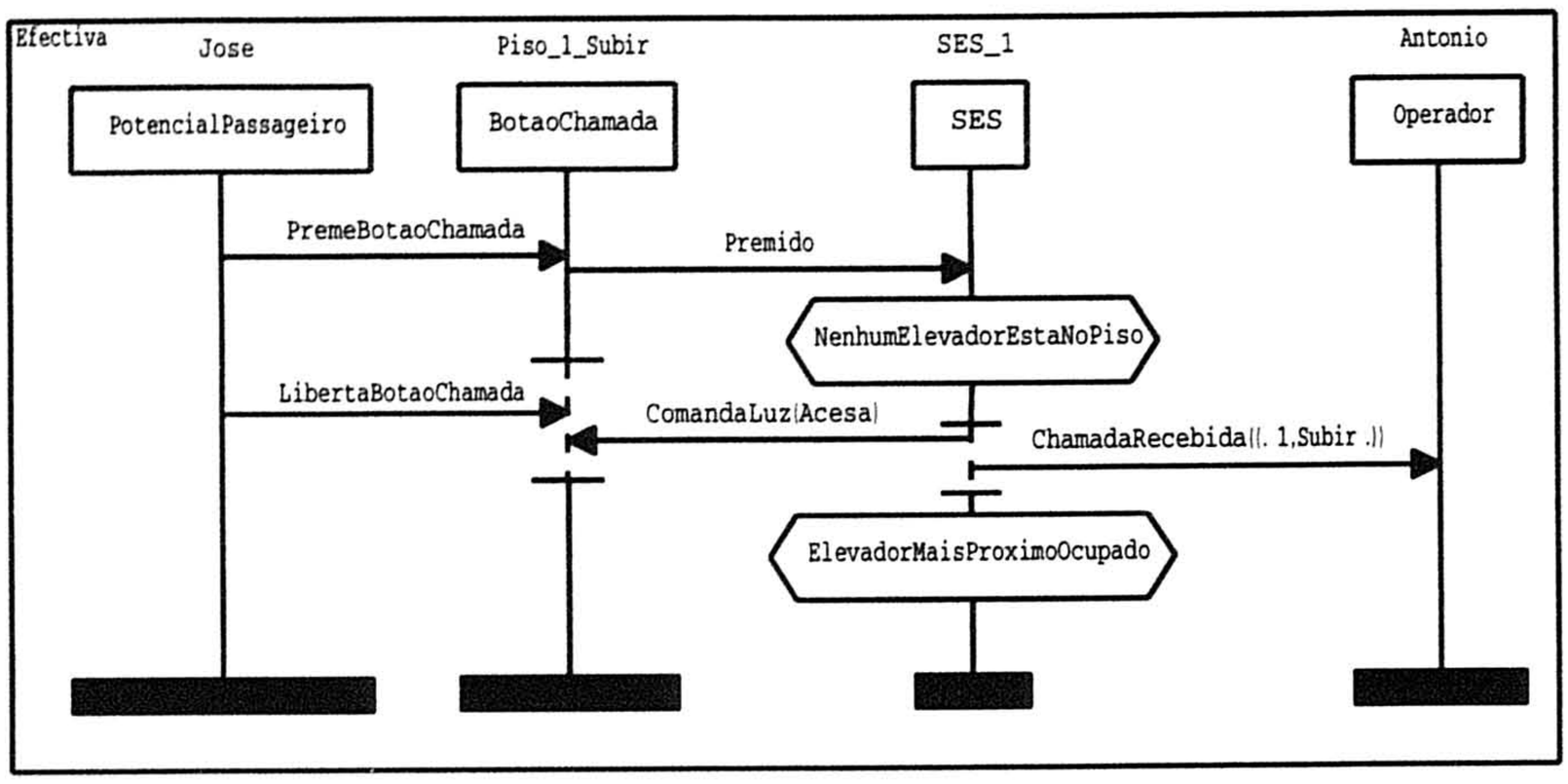

PortaPiso
P_Elev_1
SES_1
Piso_3_Elev_1
Antonio
Porta
SES
Porta
Operador
Fecha
Fecha
Fechada
4seg
RegistaAlarme(I. PNaoFecha,1,3 .)

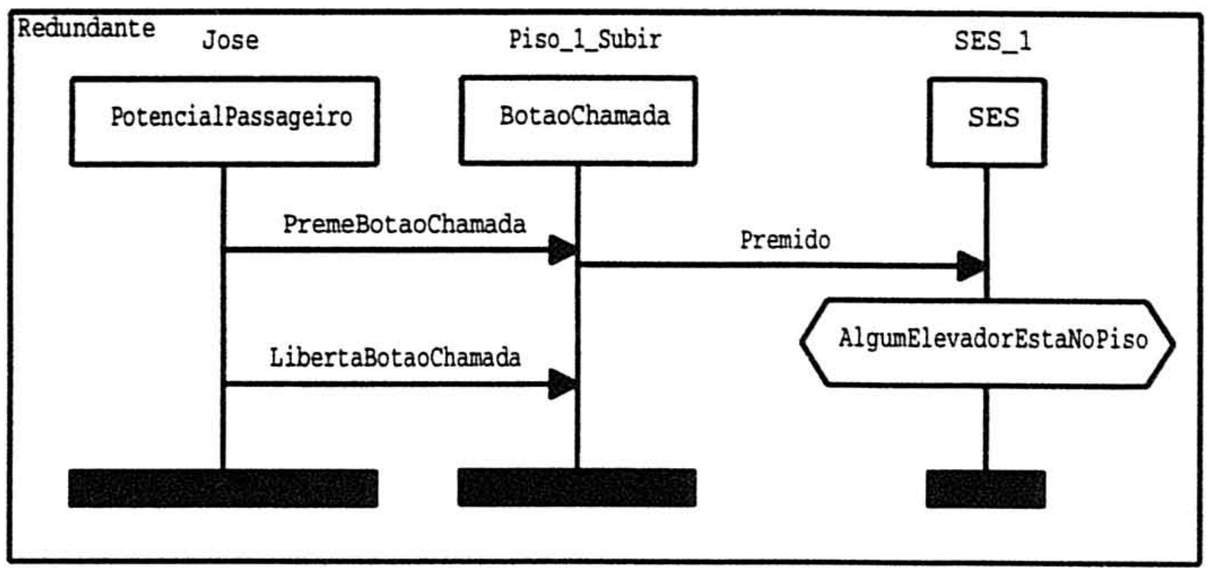

PortaElevador
P_Elev_1
SES_1
Piso_3_Elev_1
Antonio
Porta
SES
Porta
Operador
Fecha
Fecha
4seg
Fechada
RegistaAlarme(I. IENaoFecha,1,3 .I)

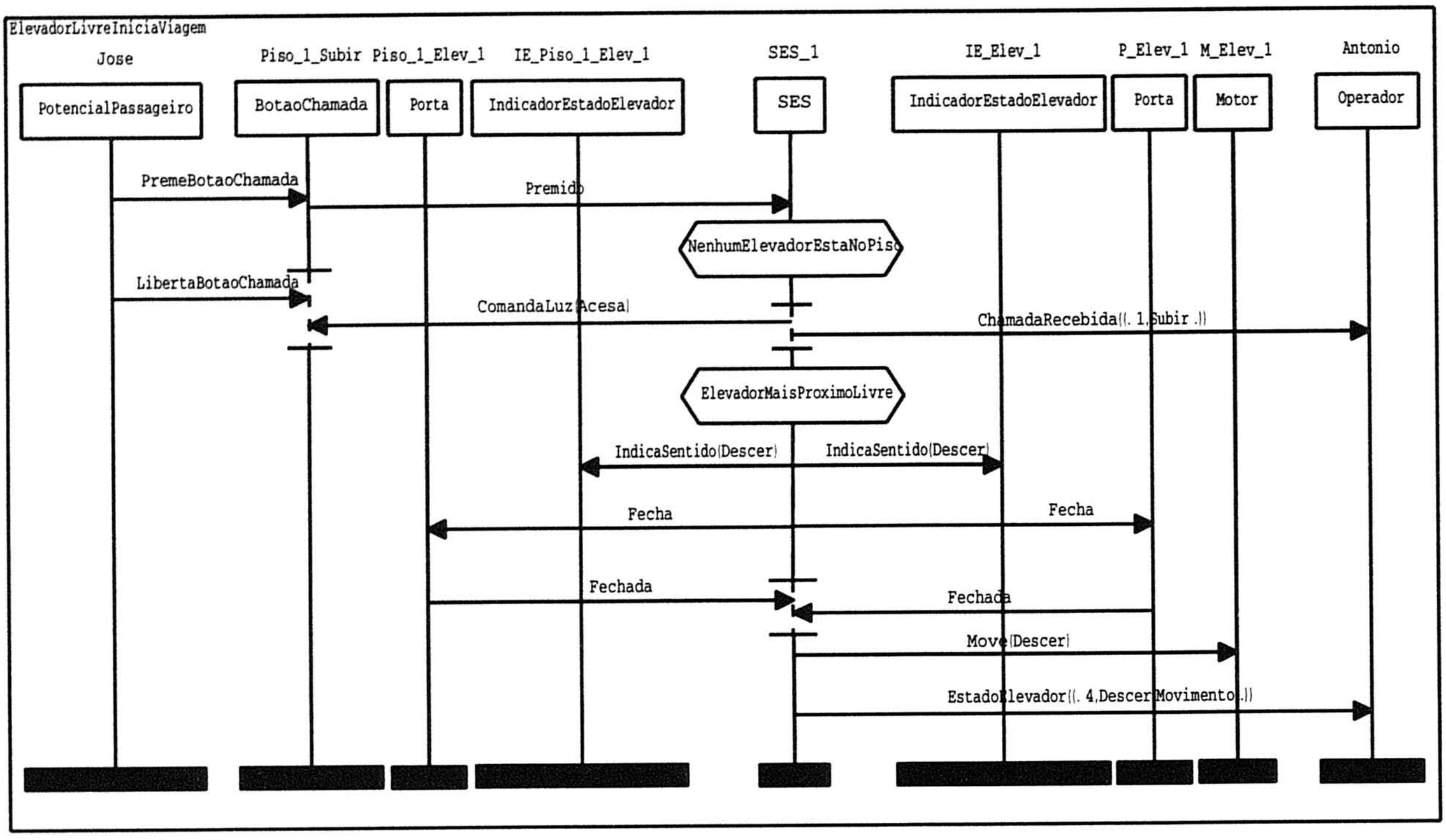

Ruptura
ST_Elev_1
F_Elev_1
SES_1
A_Elev_1
SensorTensao
Freio
SES
Altifalante
TensaoAlterada(Ruptura)
Trava
RegistaAlarme(|. RupturaCabo,1,Nenhum .|)

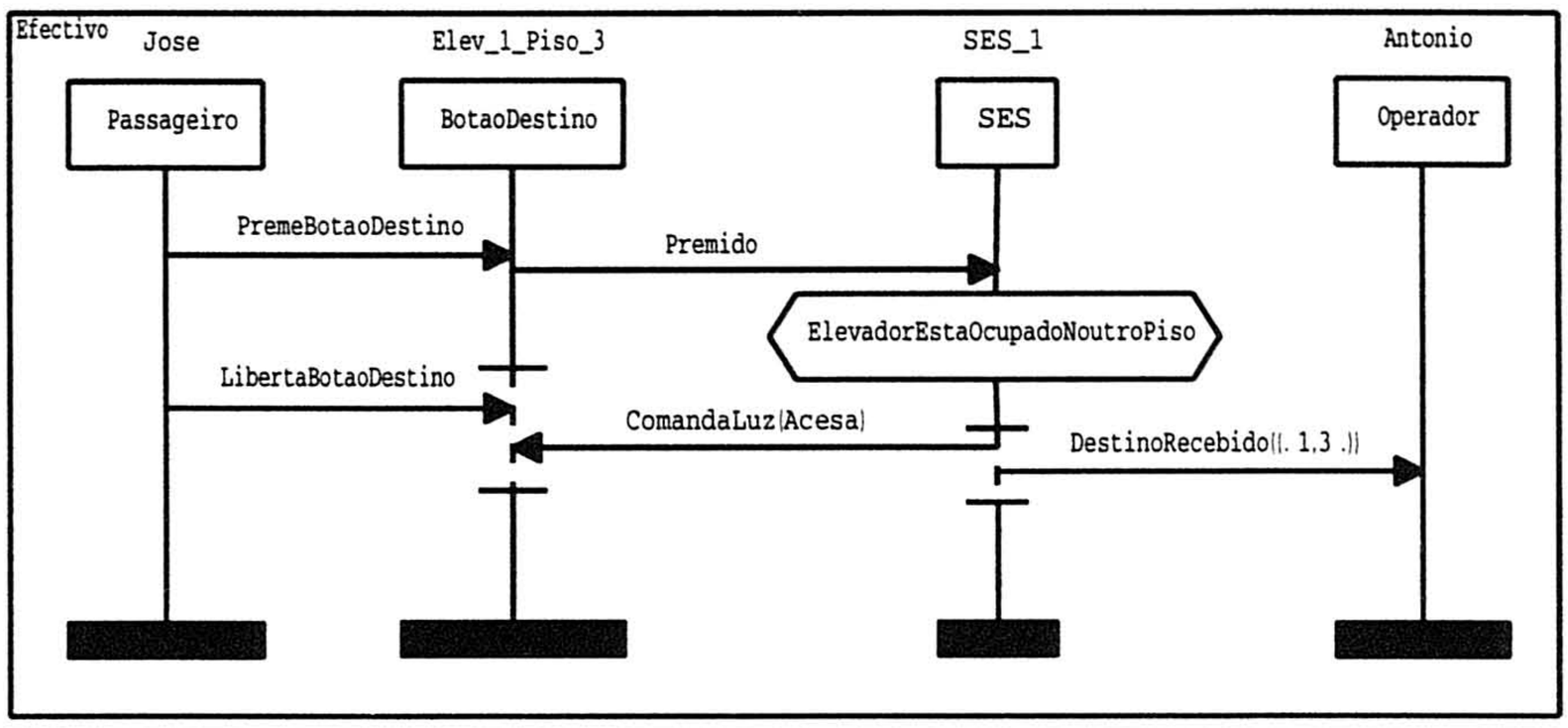

Arranque
P_Elev_1
IE_Elev_1
SES_1
Piso_1_Elev_1
IE_Piso_1_Elev_1
Antonio
Porta
IndicadorEstadoElevador
SES
Porta
IndicadorEstadoElevador
Operador
IndicaPiso(1)
IndicaPiso(1)
IndicaSentido(Nenhum)
IndicaSentido(Nenhum)
EstadoElevador({. 1,Nenhum,Livre .})
Abre
Abre

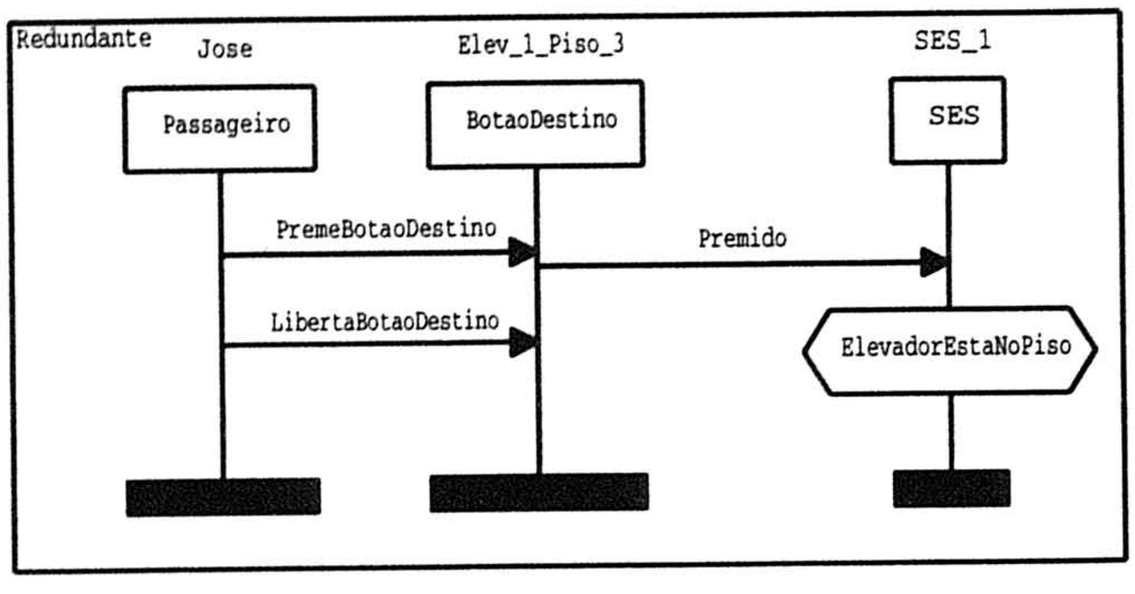

OrdemParaTerminar
SES_1
Antonio
SES
Operador
ParaSistema

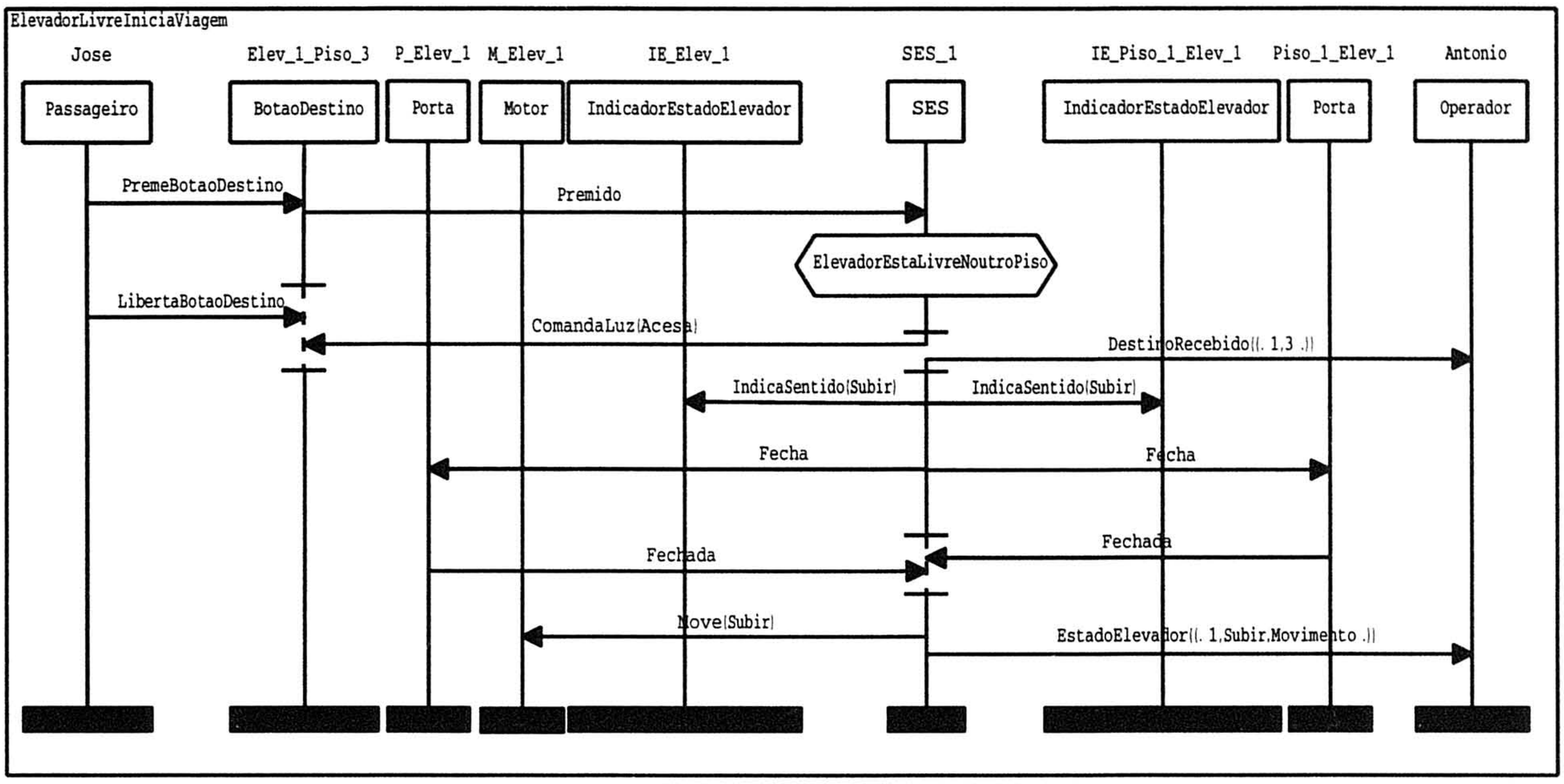

Efectiva
Jose
Piso_1_Subir
SES_1
Antonio
PotencialPassageiro
BotaoChamada
SES
Operador
PremeBotaoChamada
Premido
NenhumElevadorEstaNoPiso
LibertaBotaoChamada
ComandaLuz(Acesa)
ChamadaRecebida((. 1,Subir .))
ElevadorMaisProximoOcupado

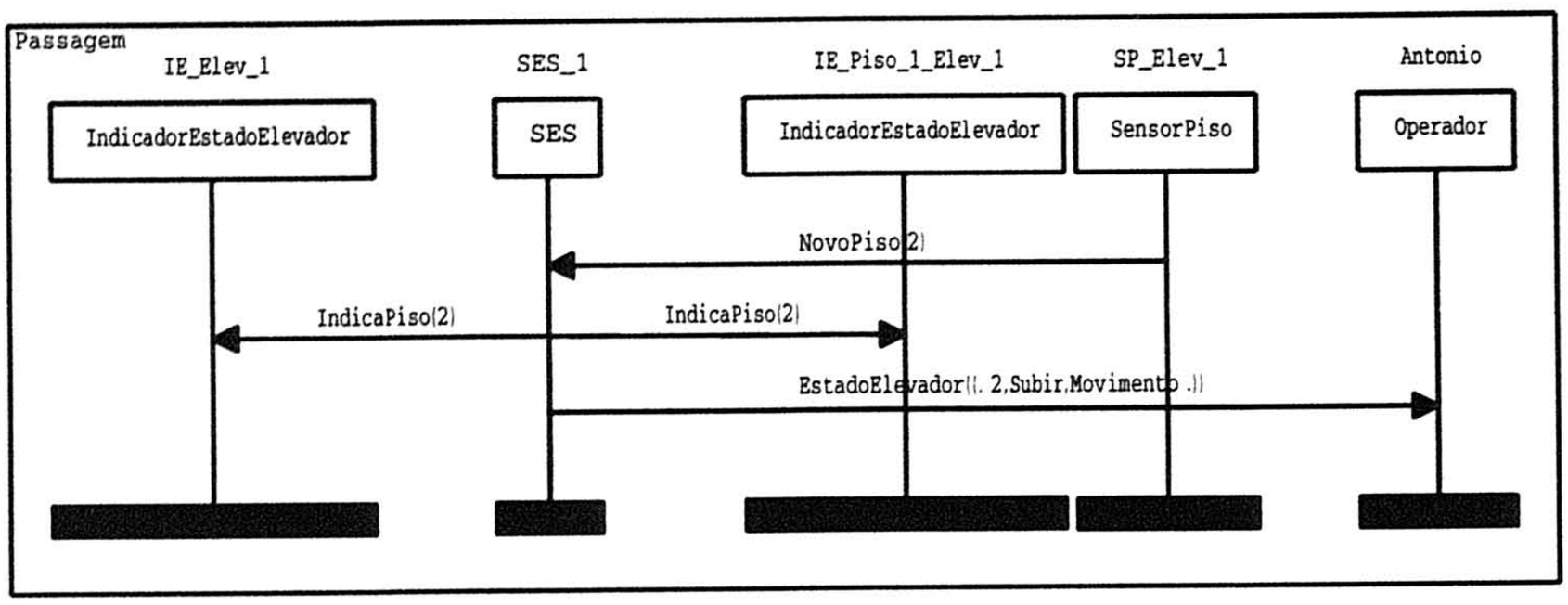

Redundante
Jose
Piso_1_Subir
SES_1
PotencialPassageiro
BotaoChamada
SES
PremeBotaoChamada
Premido
LibertaBotaoChamada
AlgumElevadorEstaNoPiso

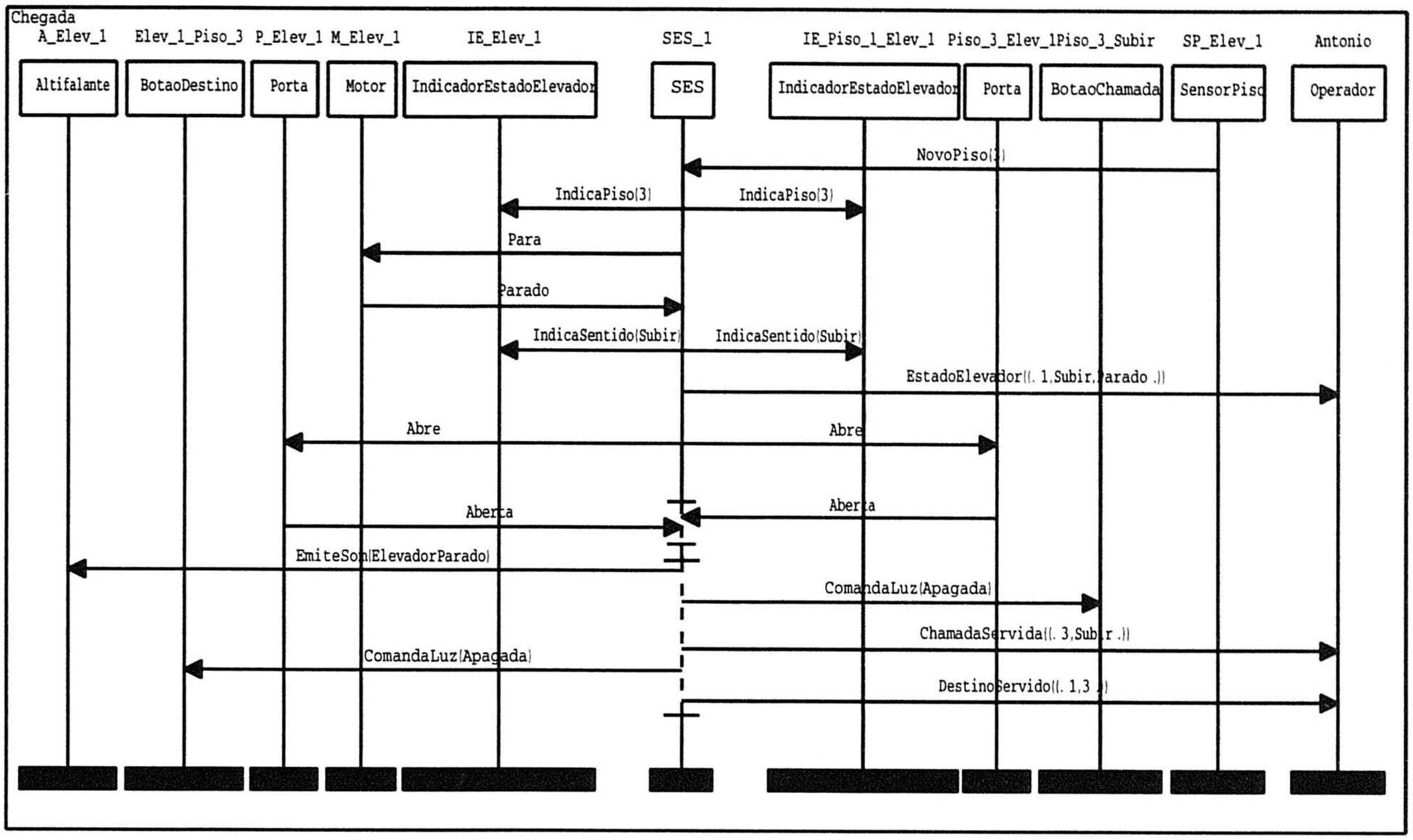

ElevadorLivreIniciaViagem
Jose
Piso_1_Subir
Piso_1_Elev_1
IE_Piso_1_Elev_1
SES_1
IE_Elev_1
P_Elev_1
M_Elev_1
Antonio
PotencialPassageiro
BotaoChamada
Porta
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
Motor
Operador
PremeBotaoChamada
Premido
NenhumElevadorEstaNoPiso
LibertaBotaoChamada
ComandaLuzAcesa()
ChamadaRecebida(1,Subir)
ElevadorMaisProximoLivre
IndicaSentido(Descer)
IndicaSentido(Descer)
Fecha
Fecha
Fechada
Fechada
Move(Descer)
EstadoElevador(4,Descer,Movimento)

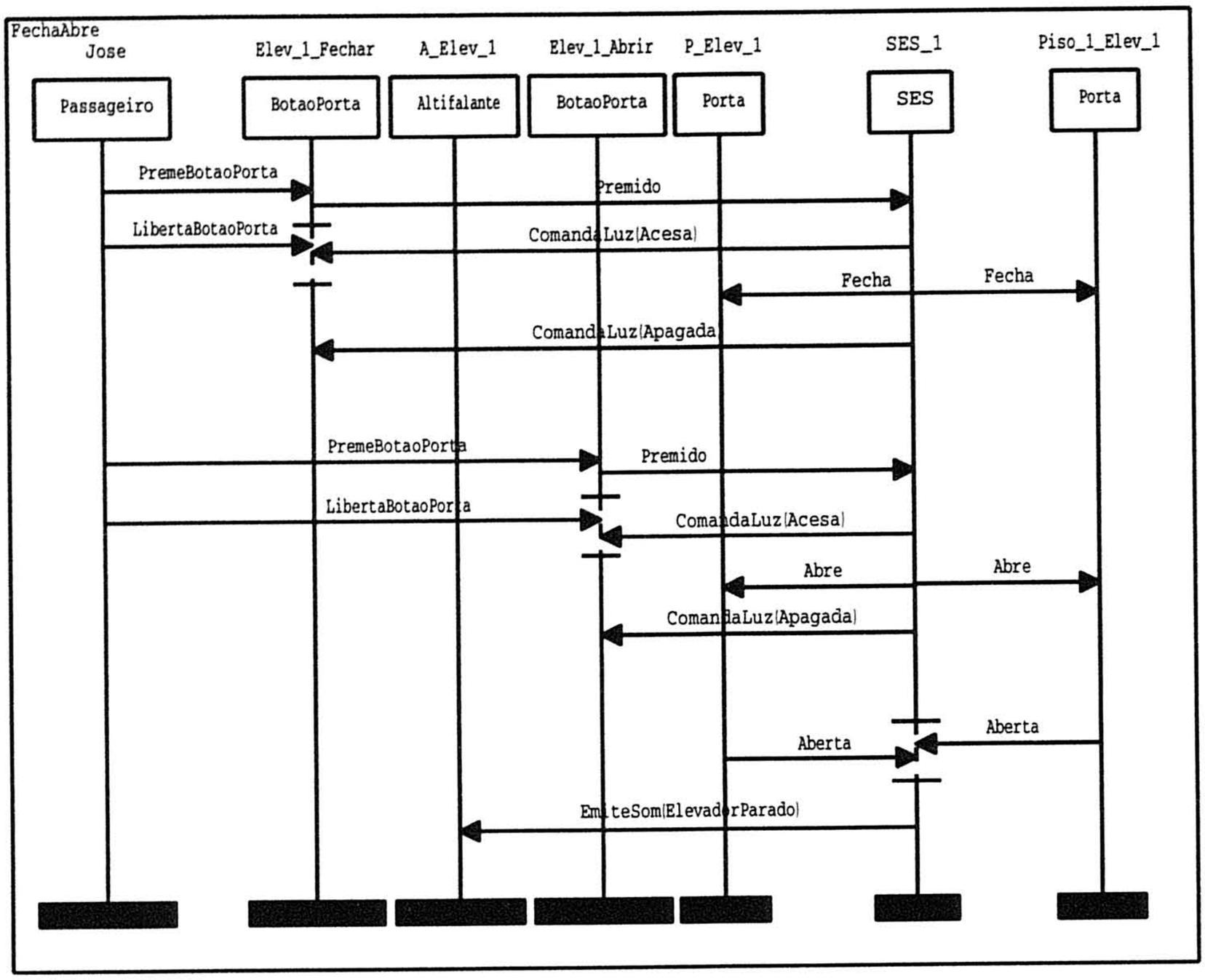

Efectivo
Jose
Elev_1_Piso_3
SES_1
Antonio
Passageiro
BotaoDestino
SES
Operador
PremeBotaoDestino
Premido
ElevadorEstaOcupadoNoutroPiso
LibertaBotaoDestino
ComandaLuz(Acesa)
DestinoRecebido({. 1,3 .})

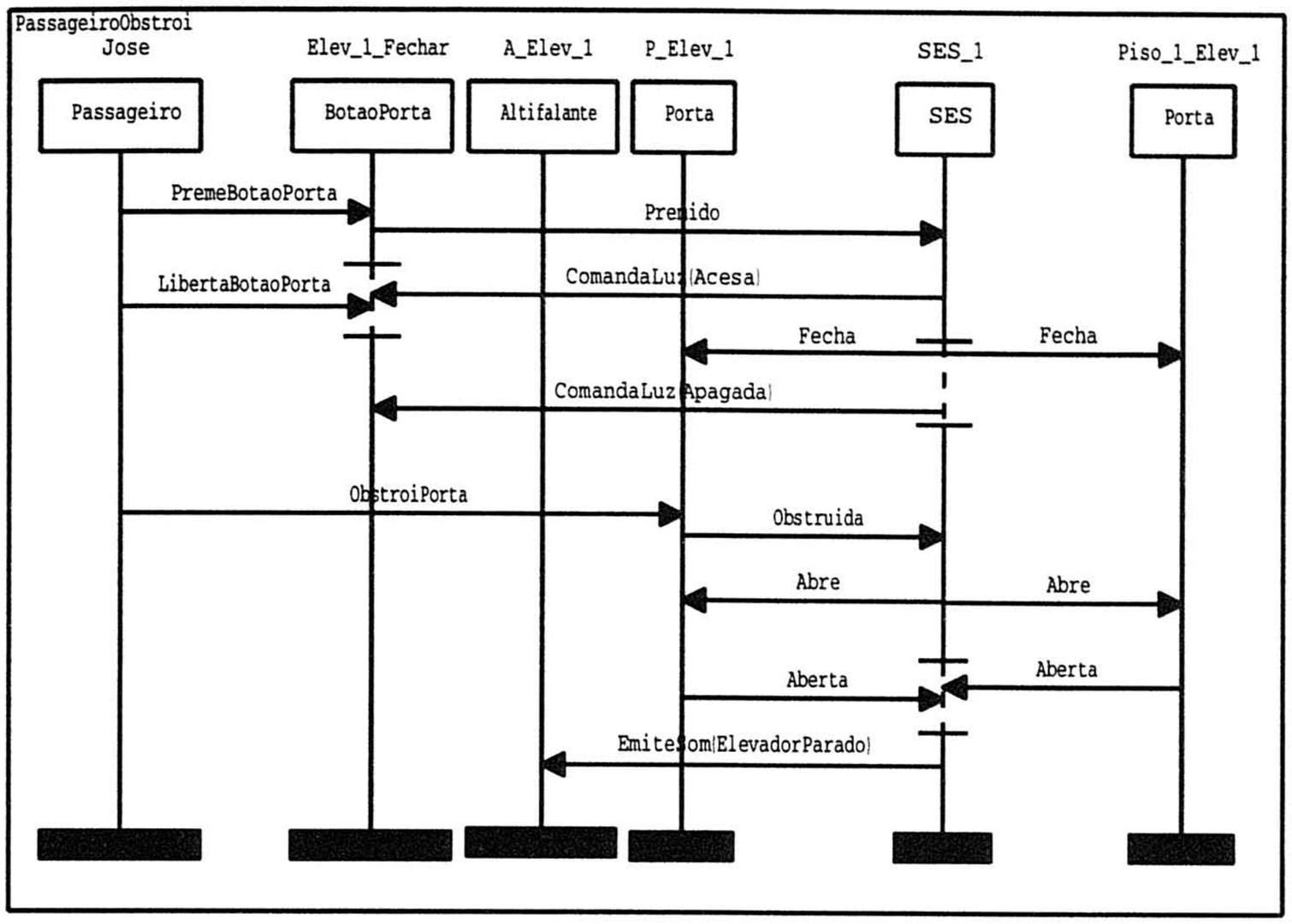

Redundante
Jose
Elev_1_Piso_3
SES_1
Passageiro
BotaoDestino
SES
PremeBotaoDestino
Premido
LibertaBotaoDestino
ElevadorEstaNoPiso

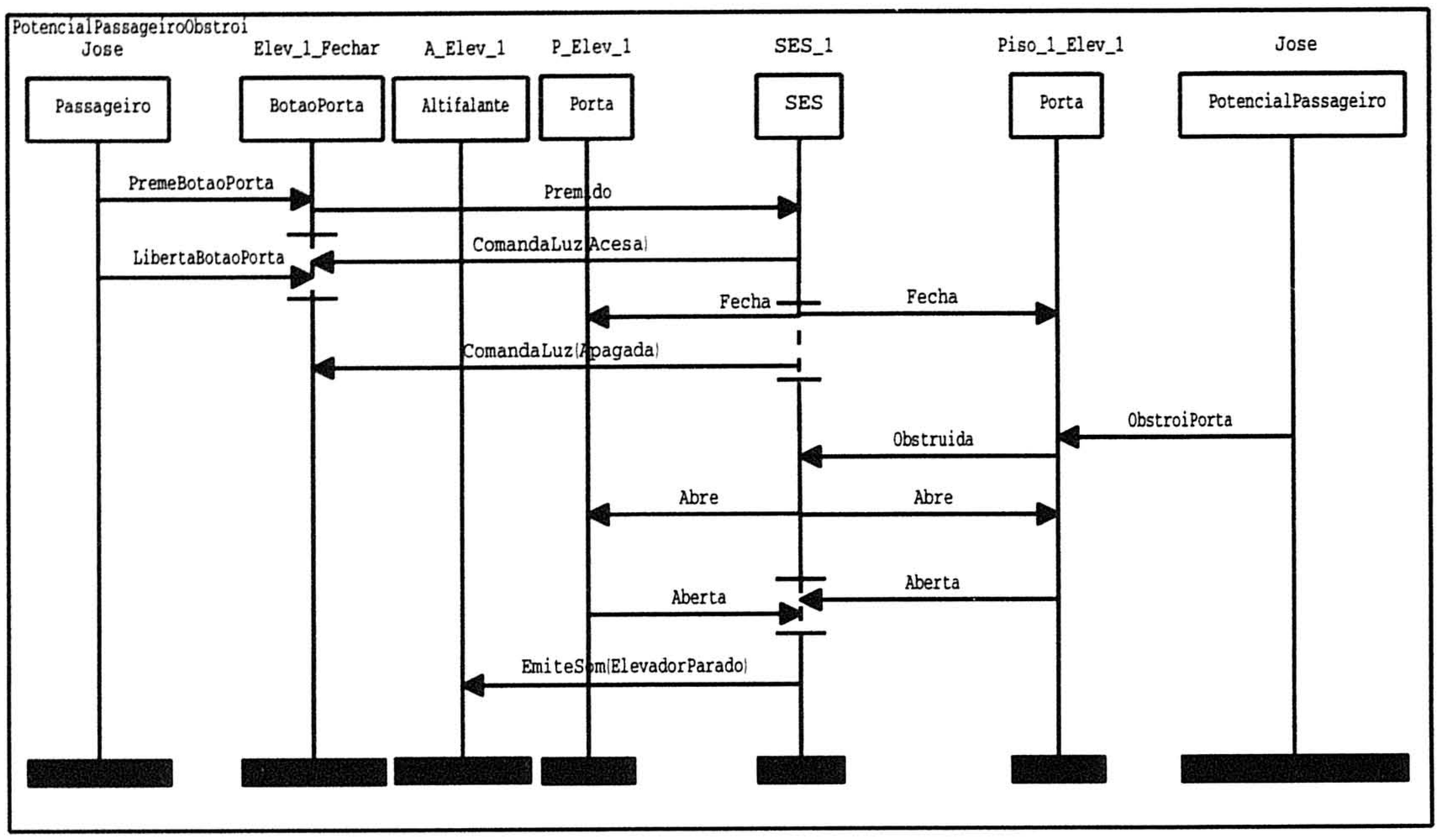

ElevadorLivreIniciaViagem
Jose
Elev_1_Piso_3
P_Elev_1
M_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_1_Elev_1
Antonio
Passageiro
BotaoDestino
Porta
Motor
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
Operador
PremeBotaoDestino
Premido
ElevadorEstaLivreNoutroPiso
LibertaBotaoDestino
ComandaLuz(Acesa)
DestinoRecebido{(. 1,3 .)}
IndicaSentido(Subir)
IndicaSentido(Subir)
Fecha
Fecha
Fechada
Fechada
Move(Subir)
EstadoElevador{(. 1,Subir,Movimento .)}

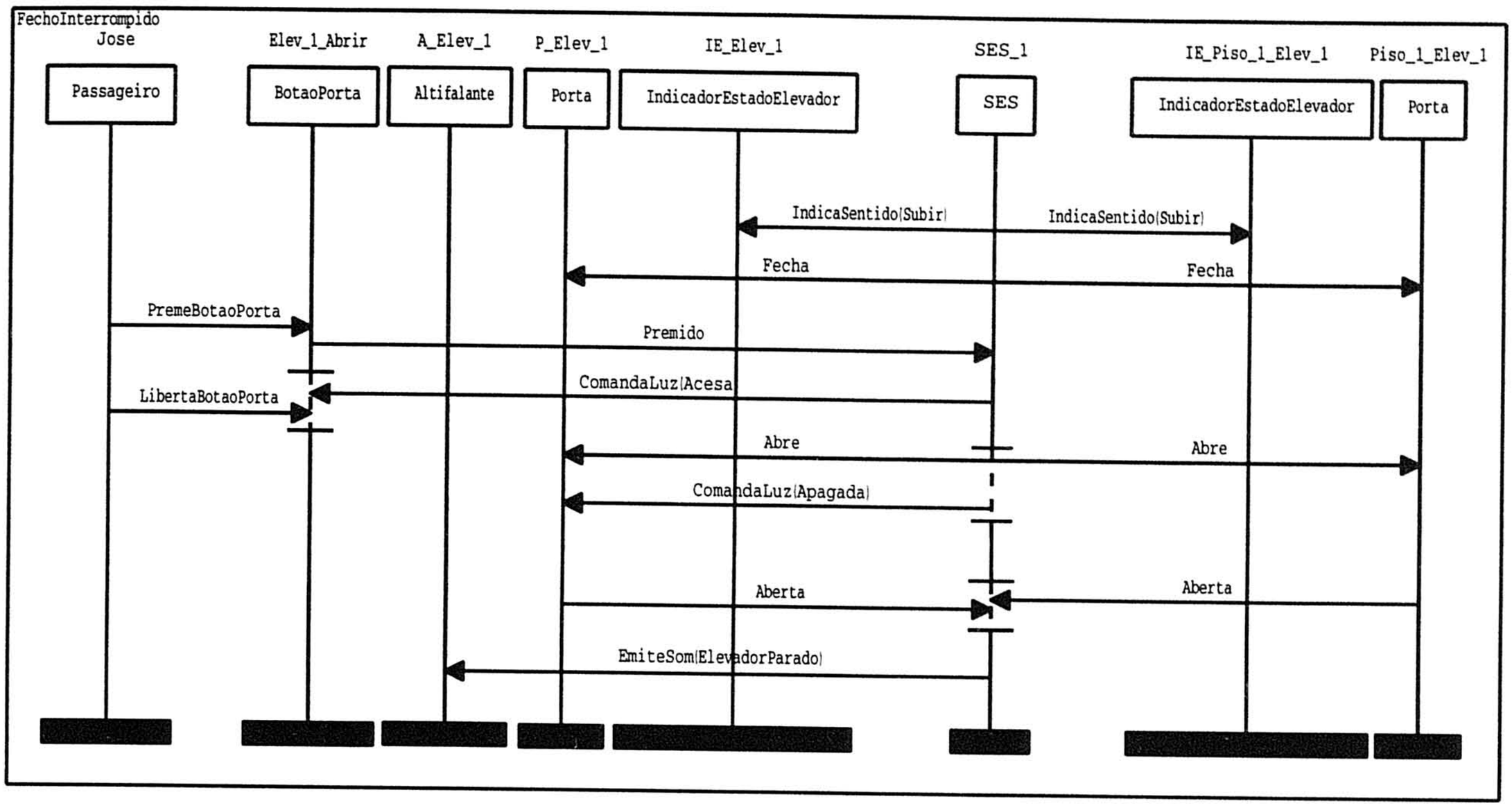

Passagem
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
SP_Elev_1
Antonio
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
SensorPiso
Operador
NovoPiso(2)
IndicaPiso(2)
IndicaPiso(2)
EstadoElevador(1, 2,Subir,Movimento ,))

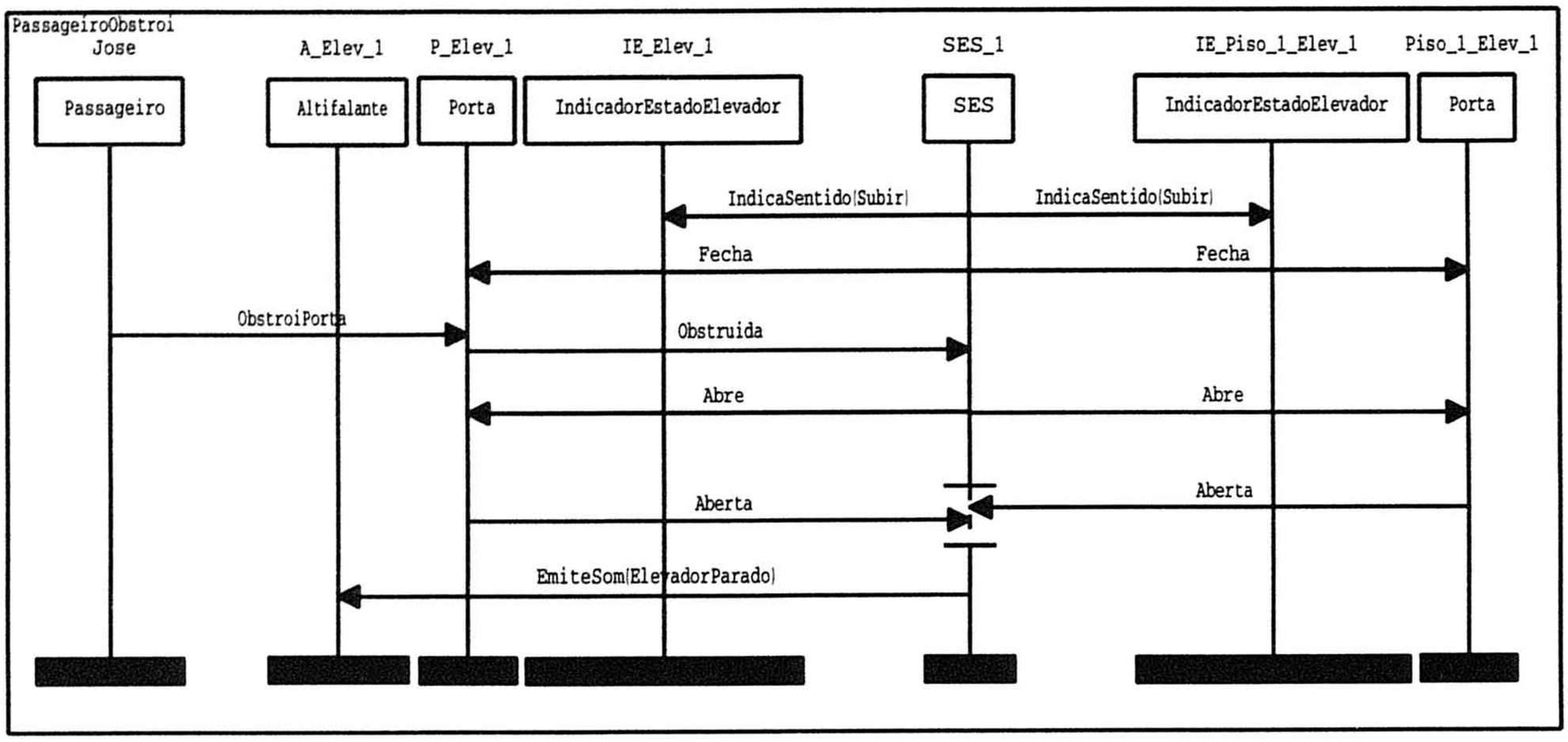

Chegada
A_Elev_1
Elev_1_Piso_3
P_Elev_1
M_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_3_Elev_1
Piso_3_Subir
SP_Elev_1
Antonio
Altifalante
BotaoDestino
Porta
Motor
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
BotaoChamada
SensorPiso
Operador
NovoPiso(3)
IndicaPiso(3)
IndicaPiso(3)
Para
Parado
IndicaSentido(Subir)
IndicaSentido(Subir)
EstadoElevador((. 1,Subir,Parado .))
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)
ComandaLuz(Apagada)
ChamadaServida((. 3,Subir .))
ComandaLuz(Apagada)
DestinoServido((. 1,3 .))

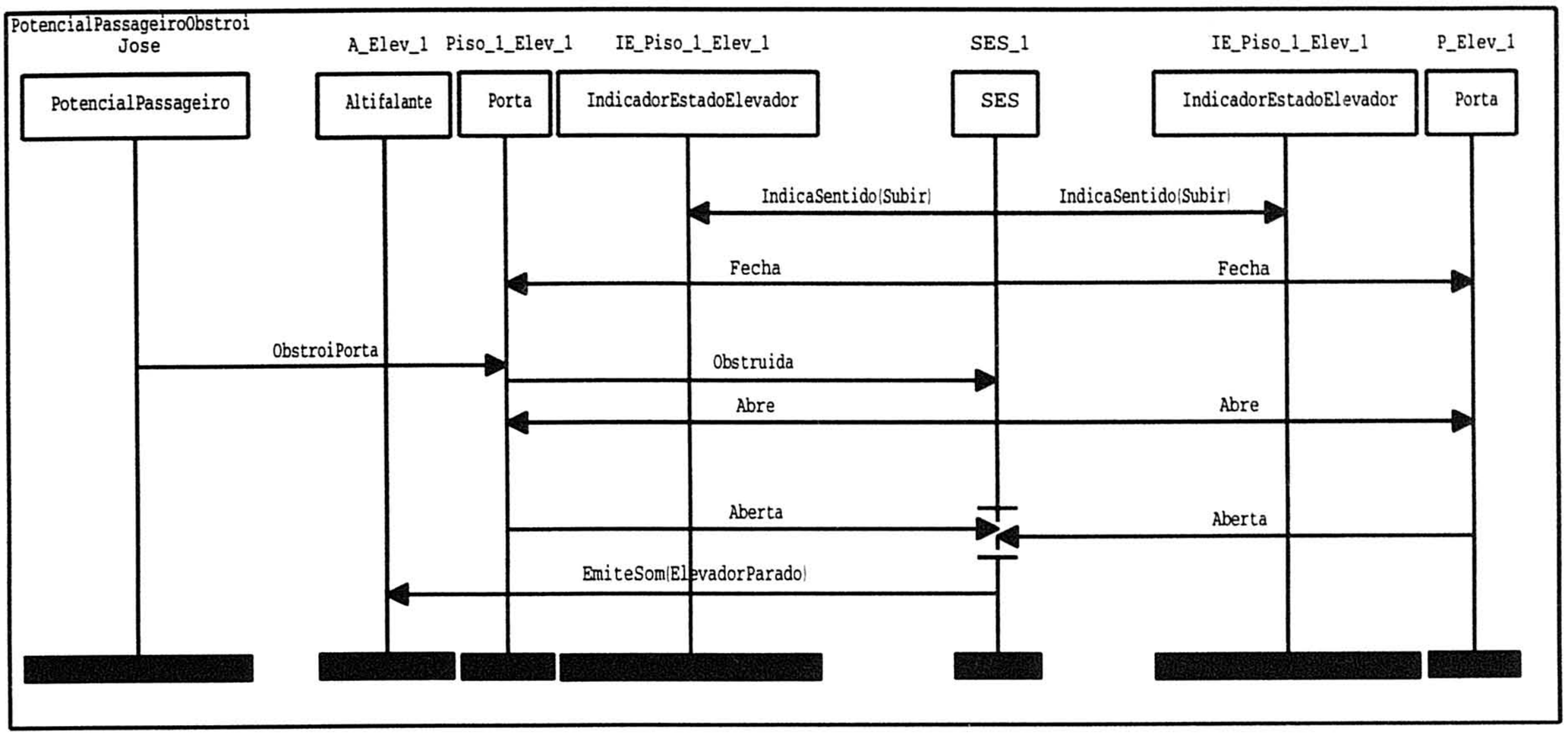

FechaAbre
Jose
Elev_1_Fechar
A_Elev_1
Elev_1_Abrir
P_Elev_1
SES_1
Piso_1_Elev_1
Passageiro
BotaoPorta
Altifalante
BotaoPorta
Porta
SES
Porta
PremeBotaoPorta
Premido
LibertaBotaoPorta
ComandaLuz(Acesa)
Fecha
Fecha
ComandaLuz(Apagada)
PremeBotaoPorta
Premido
LibertaBotaoPorta
ComandaLuz(Acesa)
Abre
Abre
ComandaLuz(Apagada)
Aberta
Aberta
EmiteSom(ElevadorParado)

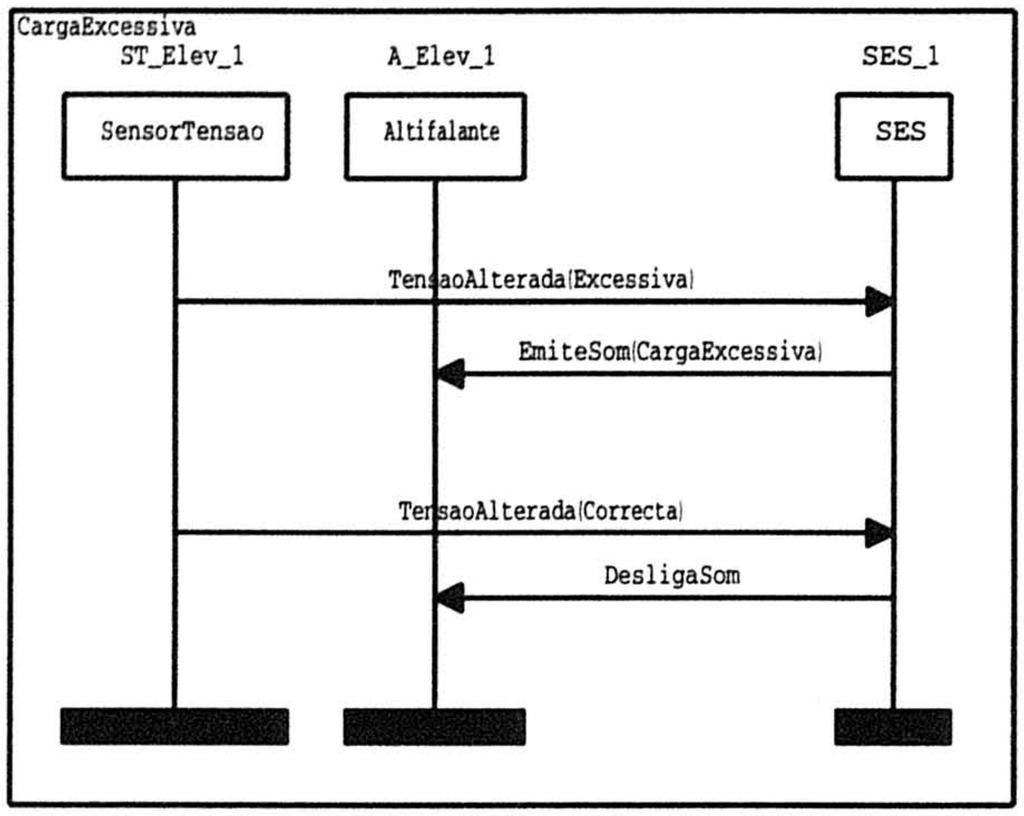

PassageiroObstroi
Jose
Elev_1_Fechar
A_Elev_1
P_Elev_1
SES_1
Piso_1_Elev_1
Passageiro
BotaoPorta
Altifalante
Porta
SES
Porta
PremeBotaoPorta
Premido
LibertaBotaoPorta
ComandaLuz(Acesa)
Fecha
Fecha
ComandaLuz(Apagada)
ObstroiPorta
Obstruida
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)

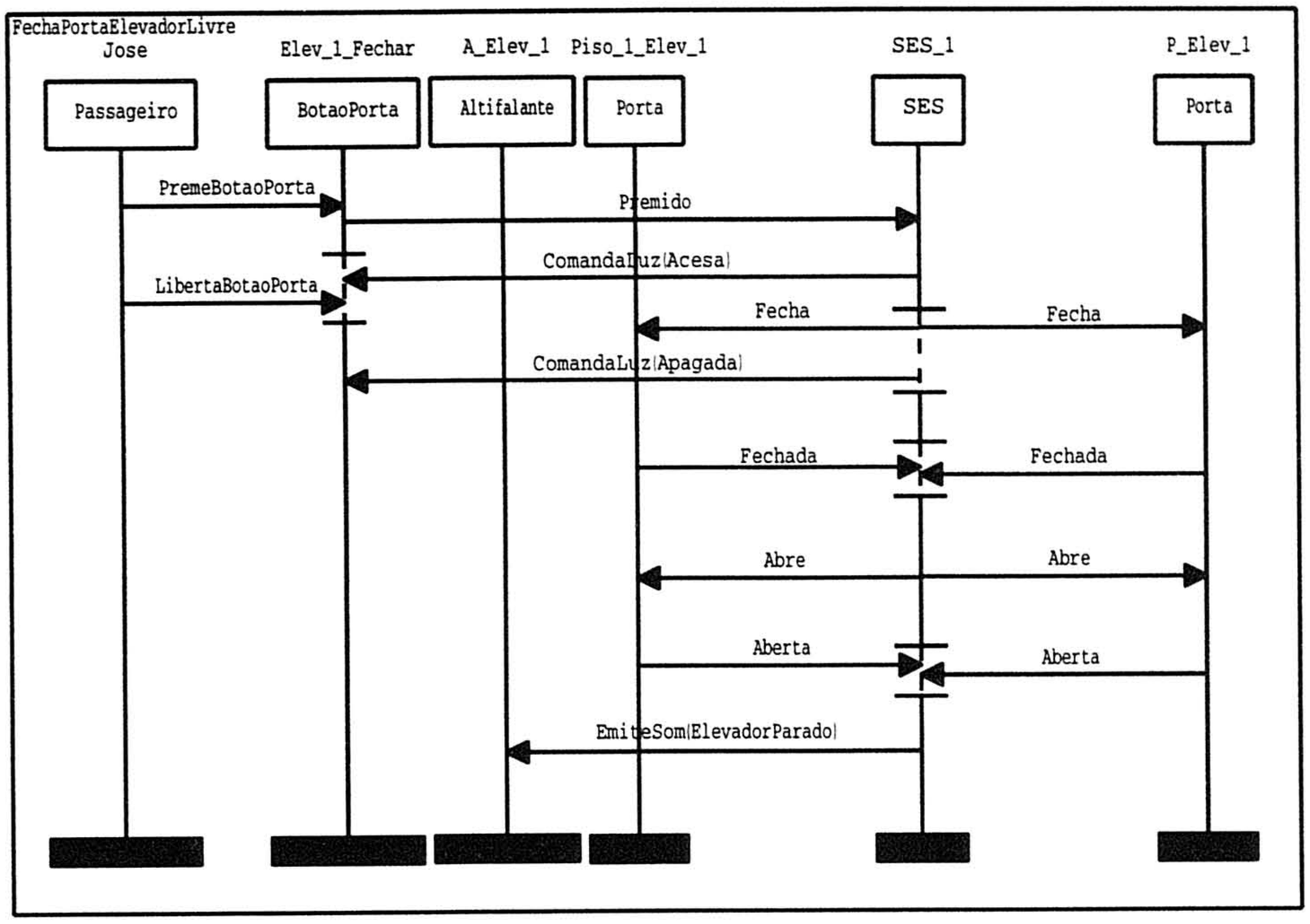

PotencialPassageiroObstroi
Jose
Elev_1_Fechar
A_Elev_1
P_Elev_1
SES_1
Piso_1_Elev_1
Jose
Passageiro
BotaoPorta
Altifalante
Porta
SES
Porta
PotencialPassageiro
PremeBotaoPorta
Premido
ComandaLuz(Acesa)
LibertaBotaoPorta
Fecha
Fecha
ComandaLuz(Apagada)
ObstroiPorta
Obstruida
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)

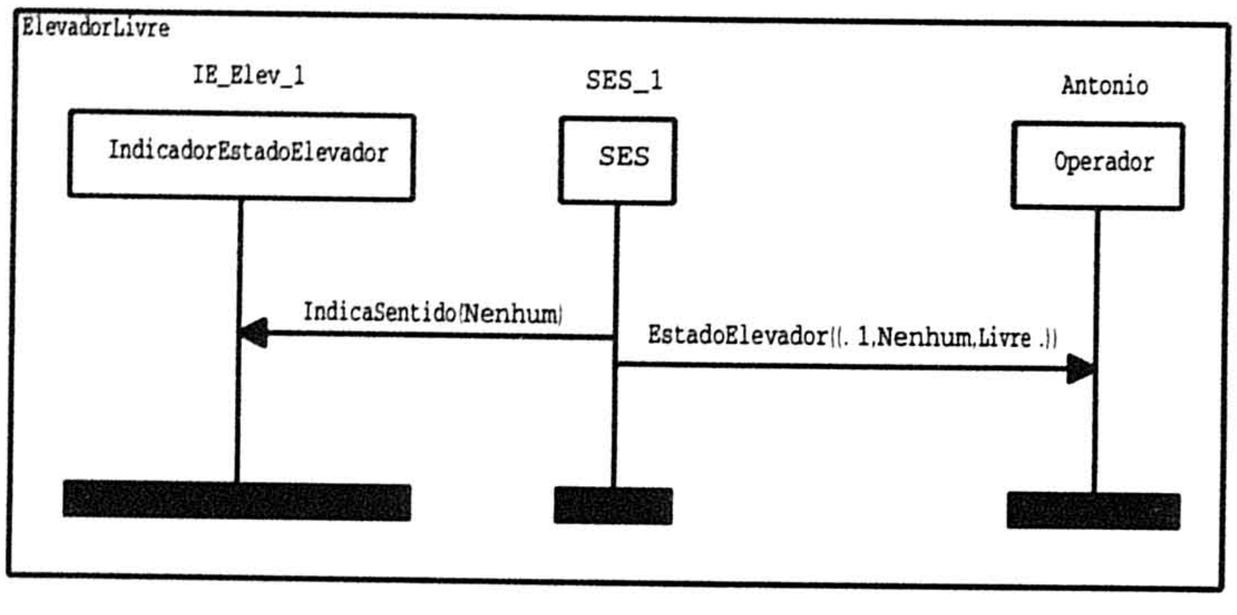

FechoInterrompido
Jose
Elev_1_Abrir
A_Elev_1
P_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_1_Elev_1
Passageiro
BotaoPorta
Altifalante
Porta
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
IndicaSentido(Subir)
IndicaSentido(Subir)
Fecha
Fecha
PremeBotaoPorta
Premido
ComandaLuz(Acesa)
LibertaBotaoPorta
Abre
Abre
ComandaLuz(Apagada)
Aberta
Aberta
EmiteSom(ElevadorParado)

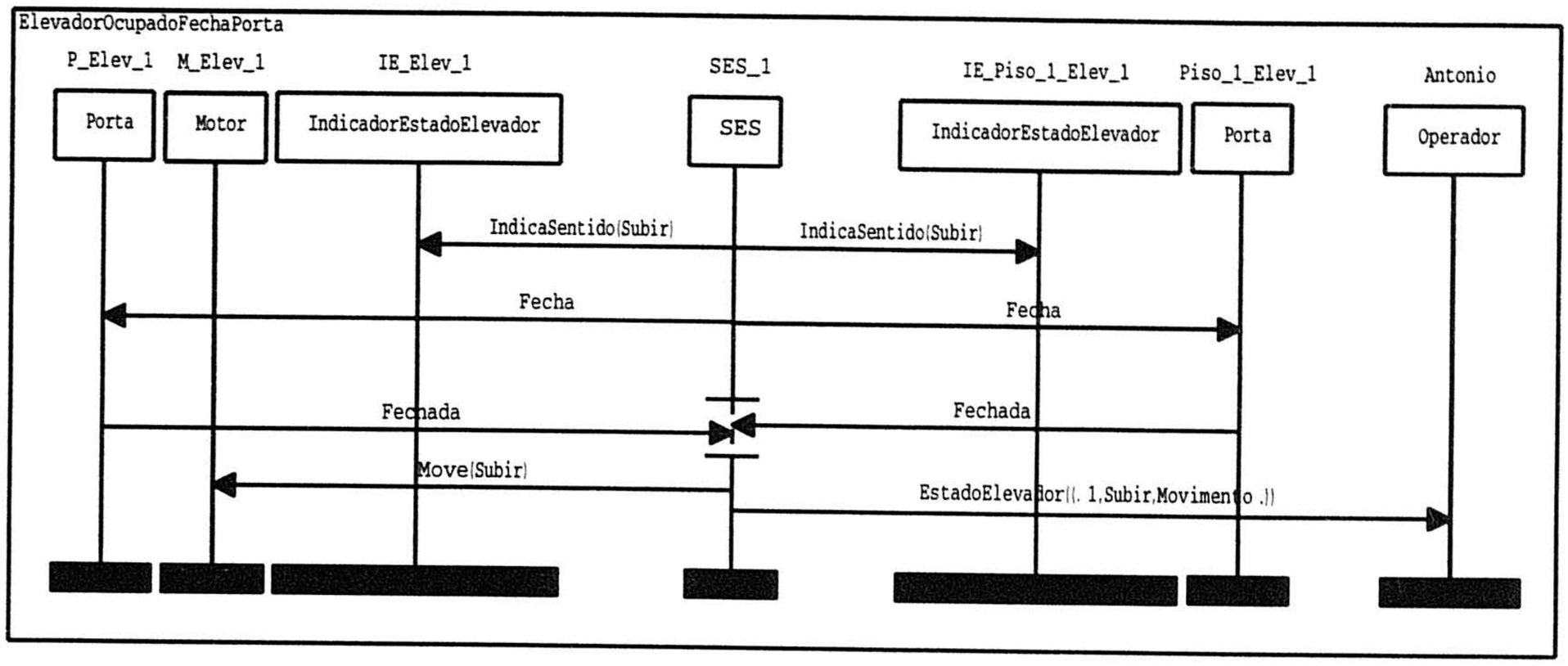

PassageiroObstroi
Jose
A_Elev_1
P_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_1_Elev_1
Passageiro
Altifalante
Porta
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
IndicaSentido(Subir)
IndicaSentido(Subir)
Fecha
Fecha
ObstroiPorta
Obstruida
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)

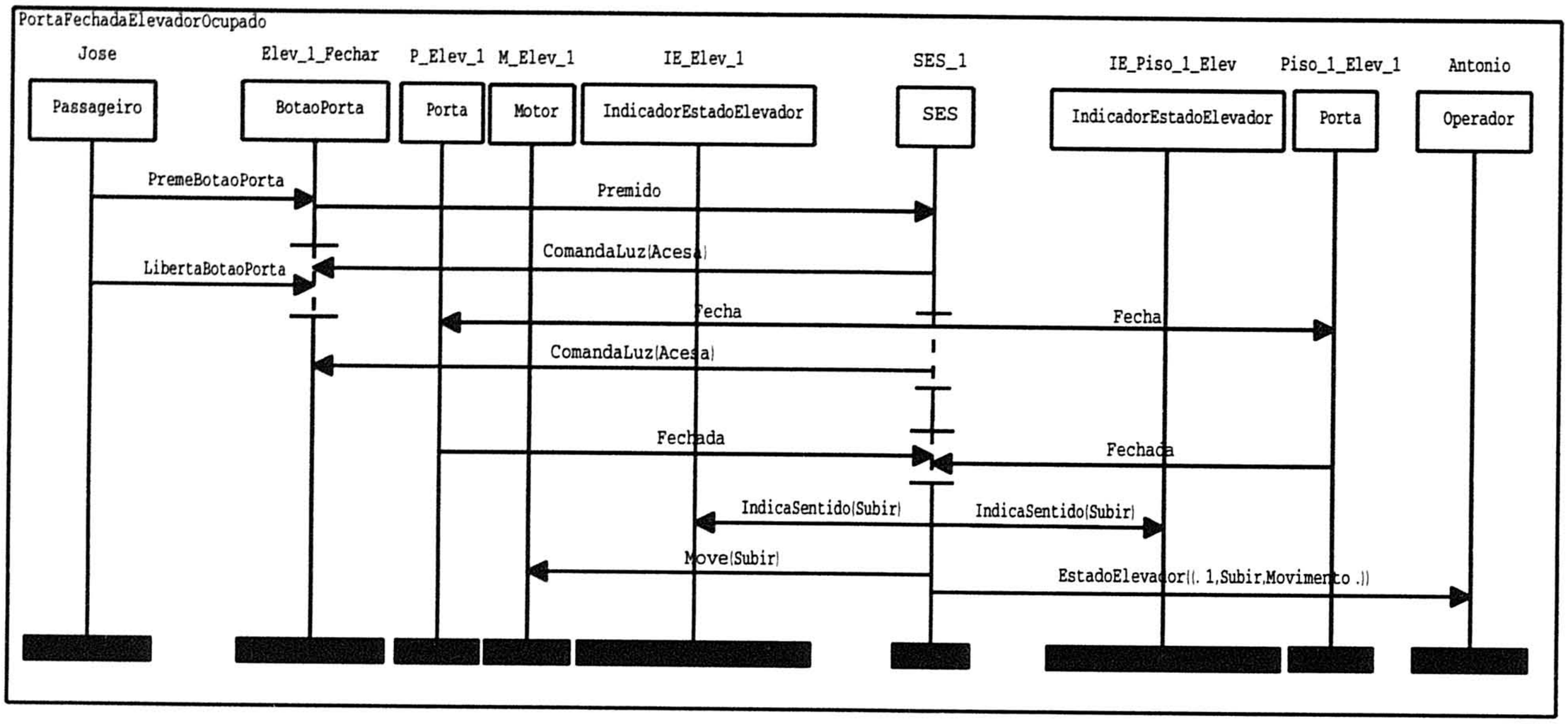

PotencialPassageiroObstroi
Jose
A_Elev_1
Piso_1_Elev_1
IE_Piso_1_Elev_1
SES_1
IE_Piso_1_Elev_1
P_Elev_1
PotencialPassageiro
Altifalante
Porta
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
IndicaSentido(Subir)
IndicaSentido(Subir)
Fecha
Fecha
ObstroiPorta
Obstruida
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)

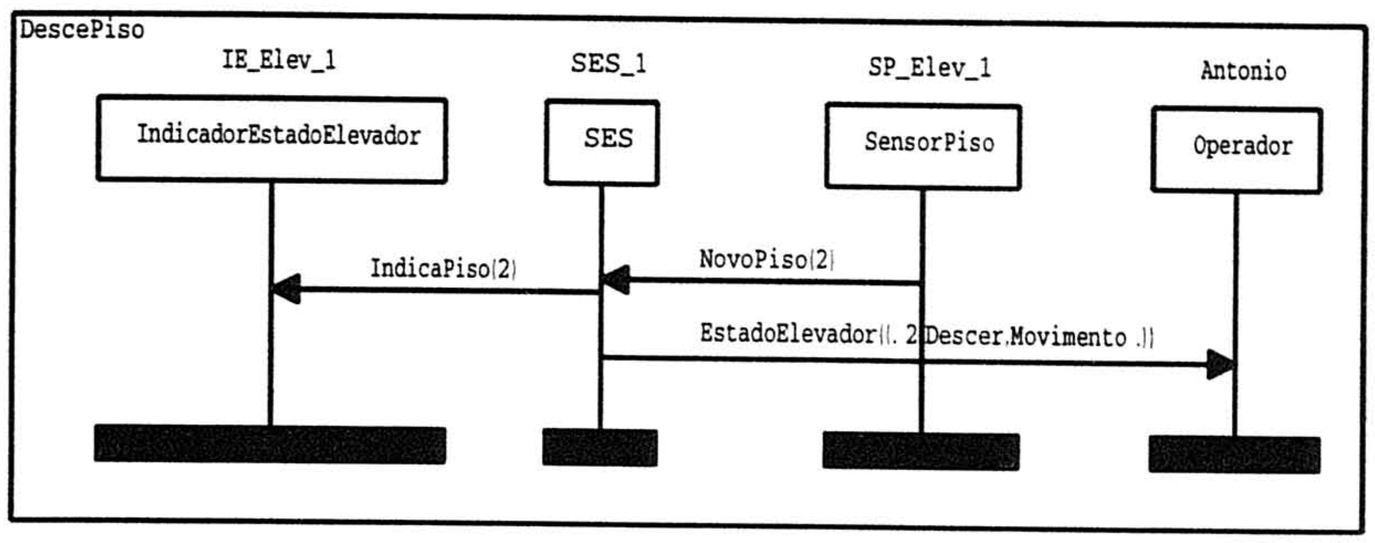

CargaExcessiva
ST_Elev_1
A_Elev_1
SES_1
SensorTensao
Altifalante
SES
TensaoAlterada(Excessiva)
EmiteSom(CargaExcessiva)
TensaoAlterada(Correcta)
DesligaSom

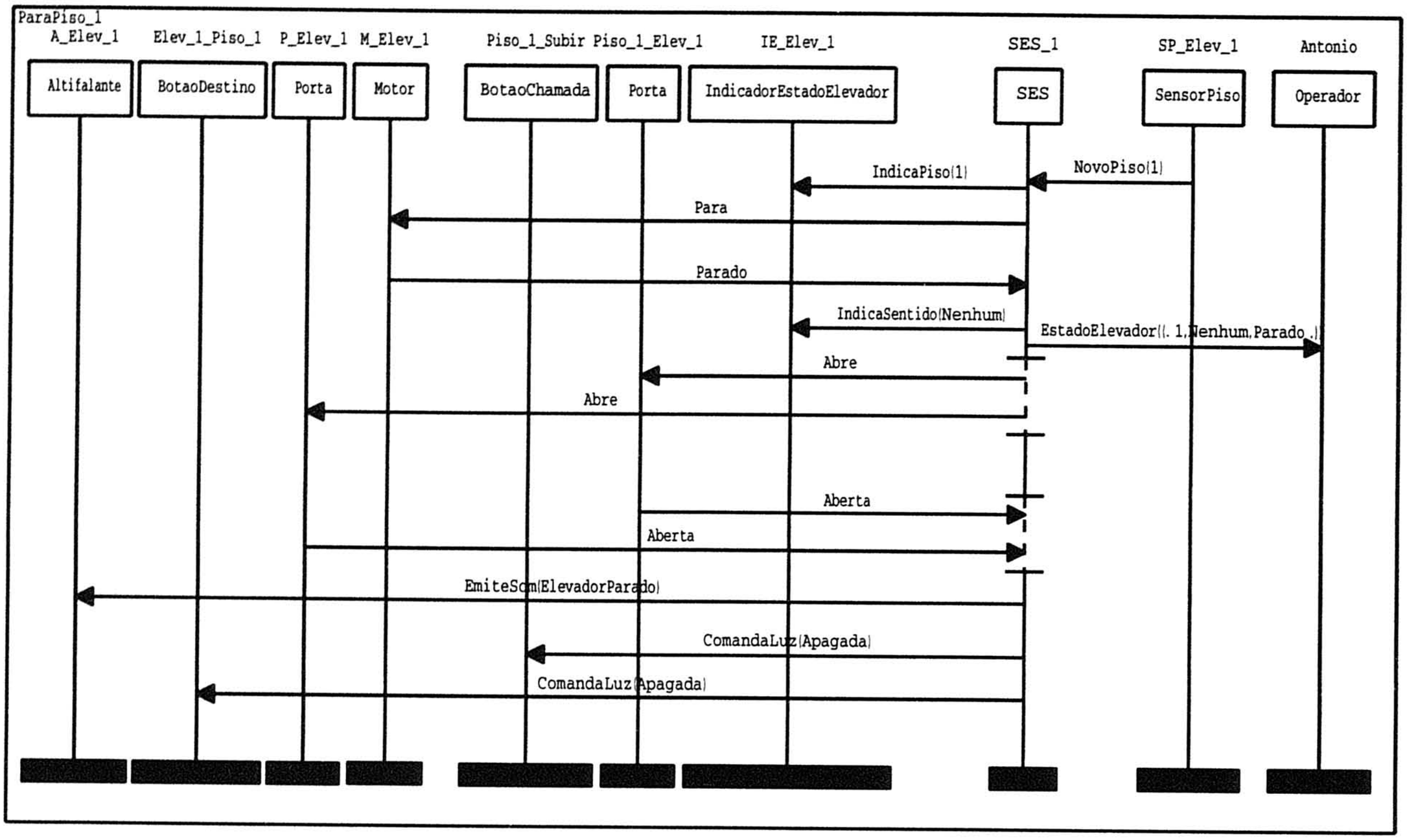

FechaPortaElevadorLivre
Jose
Elev_1_Fechar
A_Elev_1
Piso_1_Elev_1
SES_1
P_Elev_1
Passageiro
BotaoPorta
Altifalante
Porta
SES
Porta
PremeBotaoPorta
Premido
ComandaLuz(Acesa)
LibertaBotaoPorta
Fecha
Fecha
ComandaLuz(Apagada)
Fechada
Fechada
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)


ElevadorLivre
IE_Elev_1
SES_1
Antonio
IndicadorEstadoElevador
SES
Operador
IndicaSentido(Nenhum)
EstadoElevador((. 1,Nenhum,Livre .))


ElevadorOcupadoFechaPorta
P_Elev_1
M_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev_1
Piso_1_Elev_1
Antonio
Porta
Motor
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
Operador
IndicaSentido(Subir)
IndicaSentido(Subir)
Fecha
Fecha
Fechada
Fechada
Move(Subir)
EstadoElevador(1,Subir,Movimento)

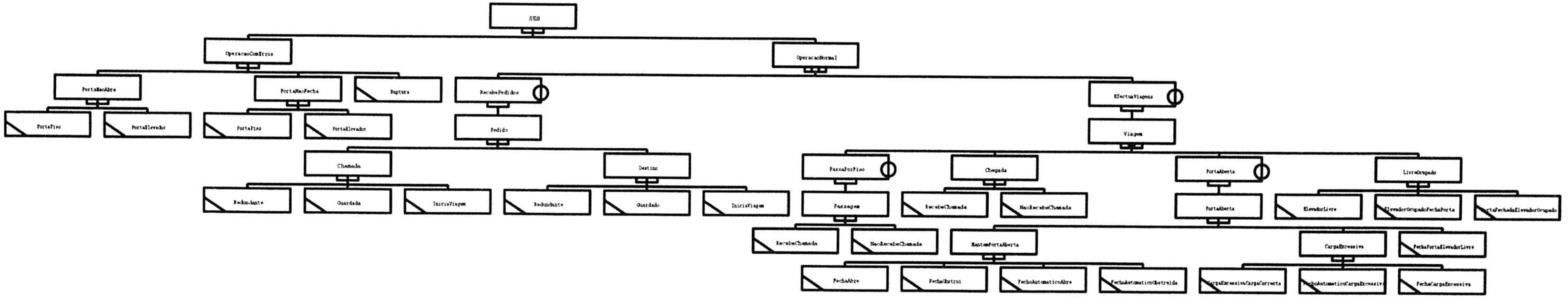

PortaFechadaElevadorOcupado
Jose
Elev_1_Fechar
P_Elev_1
M_Elev_1
IE_Elev_1
SES_1
IE_Piso_1_Elev
Piso_1_Elev_1
Antonio
Passageiro
BotaoPorta
Porta
Motor
IndicadorEstadoElevador
SES
IndicadorEstadoElevador
Porta
Operador
PremeBotaoPorta
Premido
ComandaLuz(Acesa)
LibertaBotaoPorta
Fecha
Fecha
ComandaLuz(Acesa)
Fechada
Fechada
IndicaSentido(Subir)
IndicaSentido(Subir)
Move(Subir)
EstadoElevador(. 1,Subir,Movimento .)

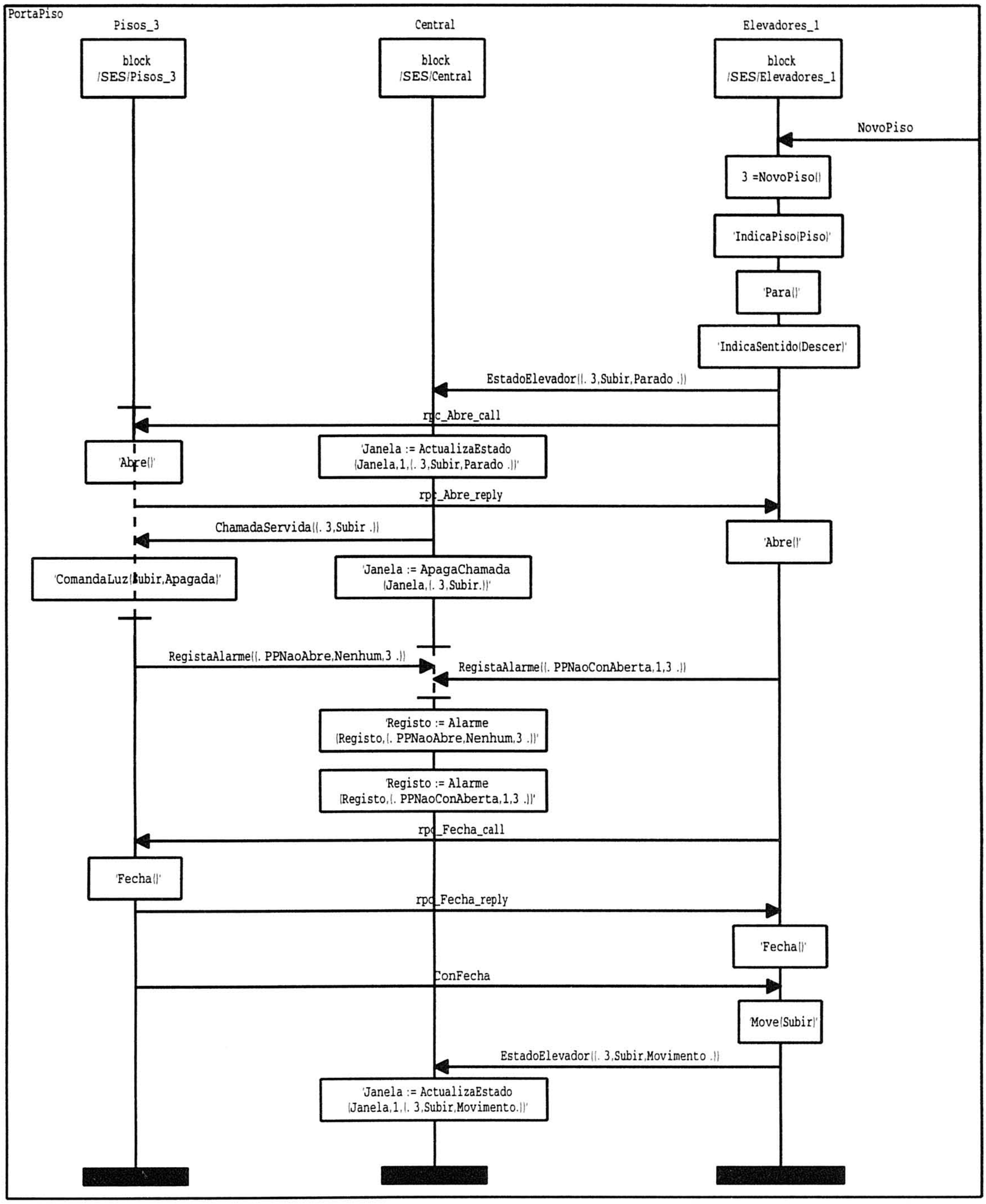

DescePiso
IE_Elev_1
SES_1
SP_Elev_1
Antonio
IndicadorEstadoElevador
SES
SensorPiso
Operador
IndicaPiso(2)
NovoPiso(2)
EstadoElevador(. 2 Descer,Movimento .)

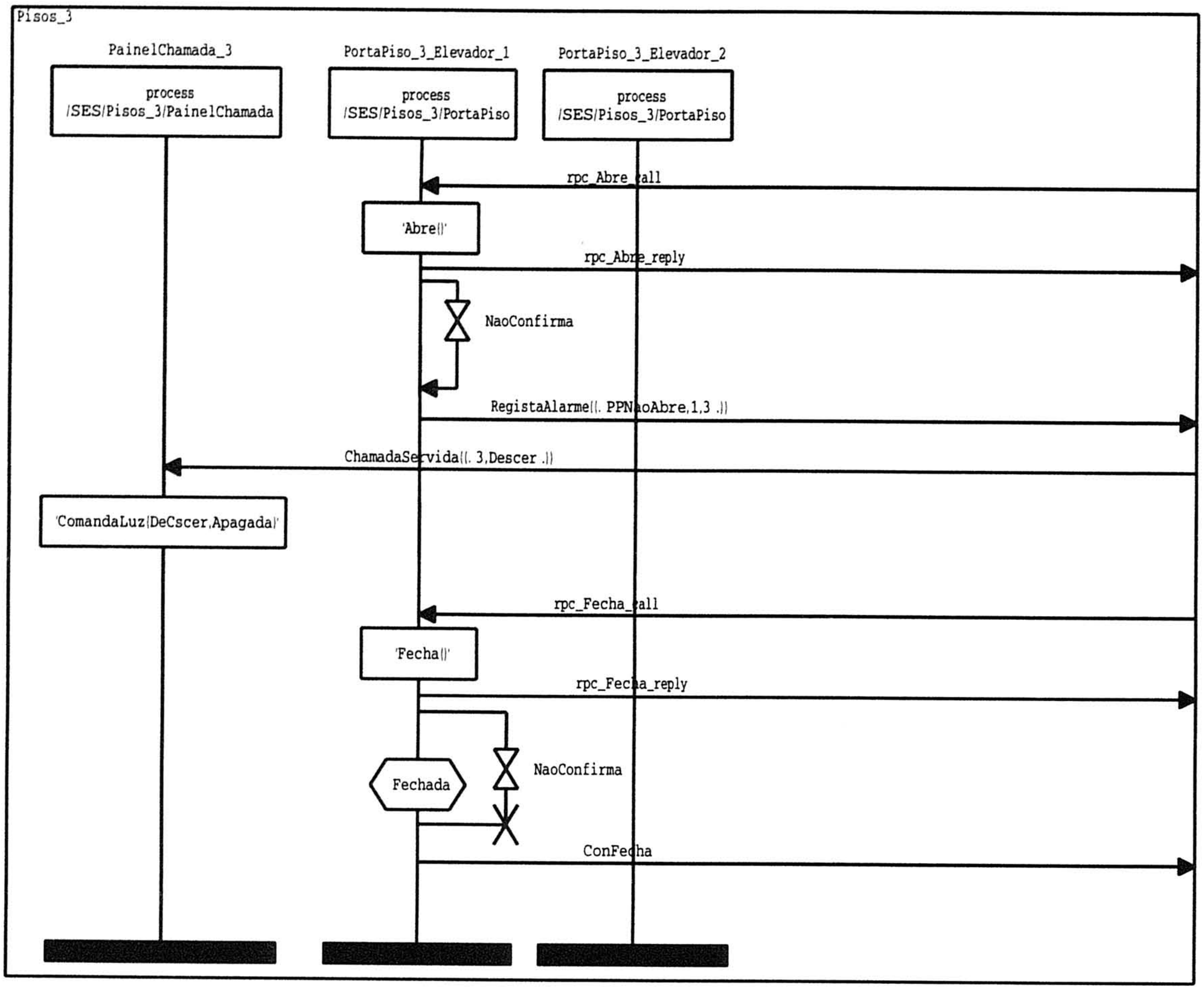

ParaPiso_1
A_Elev_1
Elev_1_Piso_1
P_Elev_1
M_Elev_1
Piso_1_Subir
Piso_1_Elev_1
IE_Elev_1
SES_1
SP_Elev_1
Antonio
Altifalante
BotaoDestino
Porta
Motor
BotaoChamada
Porta
IndicadorEstadoElevador
SES
SensorPiso
Operador
IndicaPiso(1)
NovoPiso(1)
Para
Parado
IndicaSentido(Nenhum)
EstadoElevador(1, Nenhum, Parado)
Abre
Abre
Aberta
Aberta
EmiteSom(ElevadorParado)
ComandaLuz(Apagada)
ComandaLuz(Apagada)

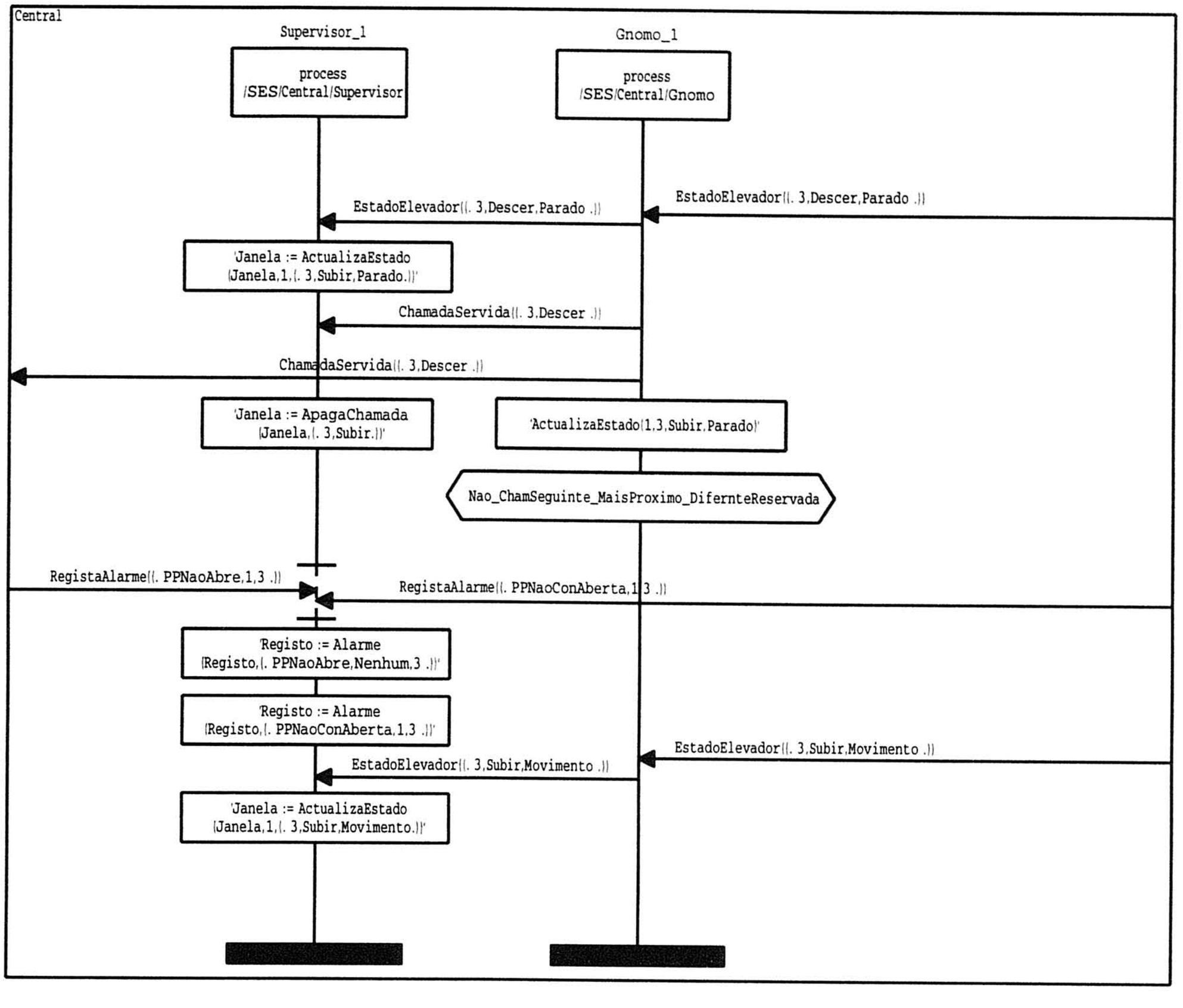
Anexo C MSCs do Projecto de Testes do SES

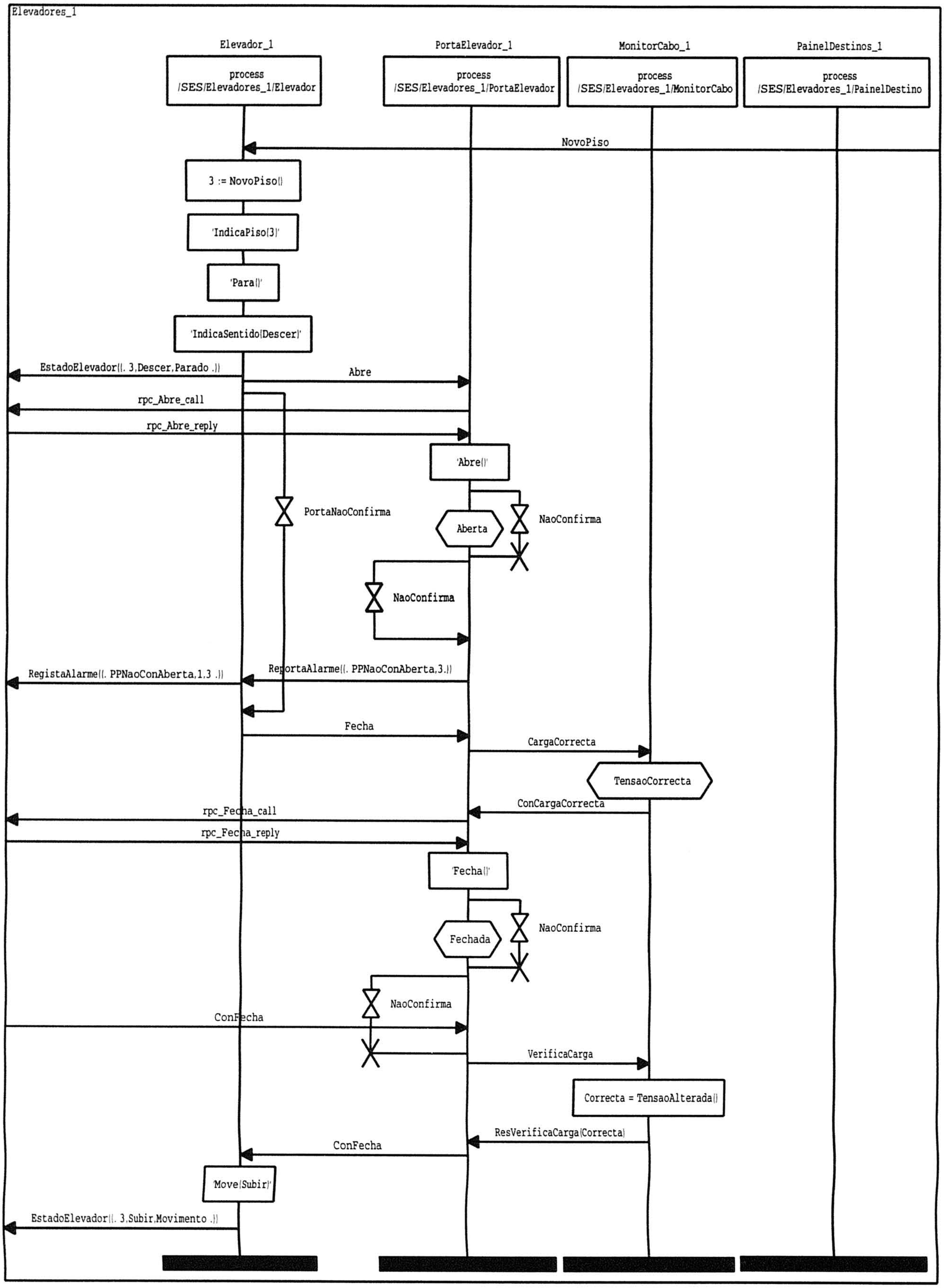
Listagem dos MSCs referentes aos testes do SES, elaborados durante a fase de projecto de testes. Os diagramas são apresentados tal como a ferramenta *Object*GEODE os fornece.

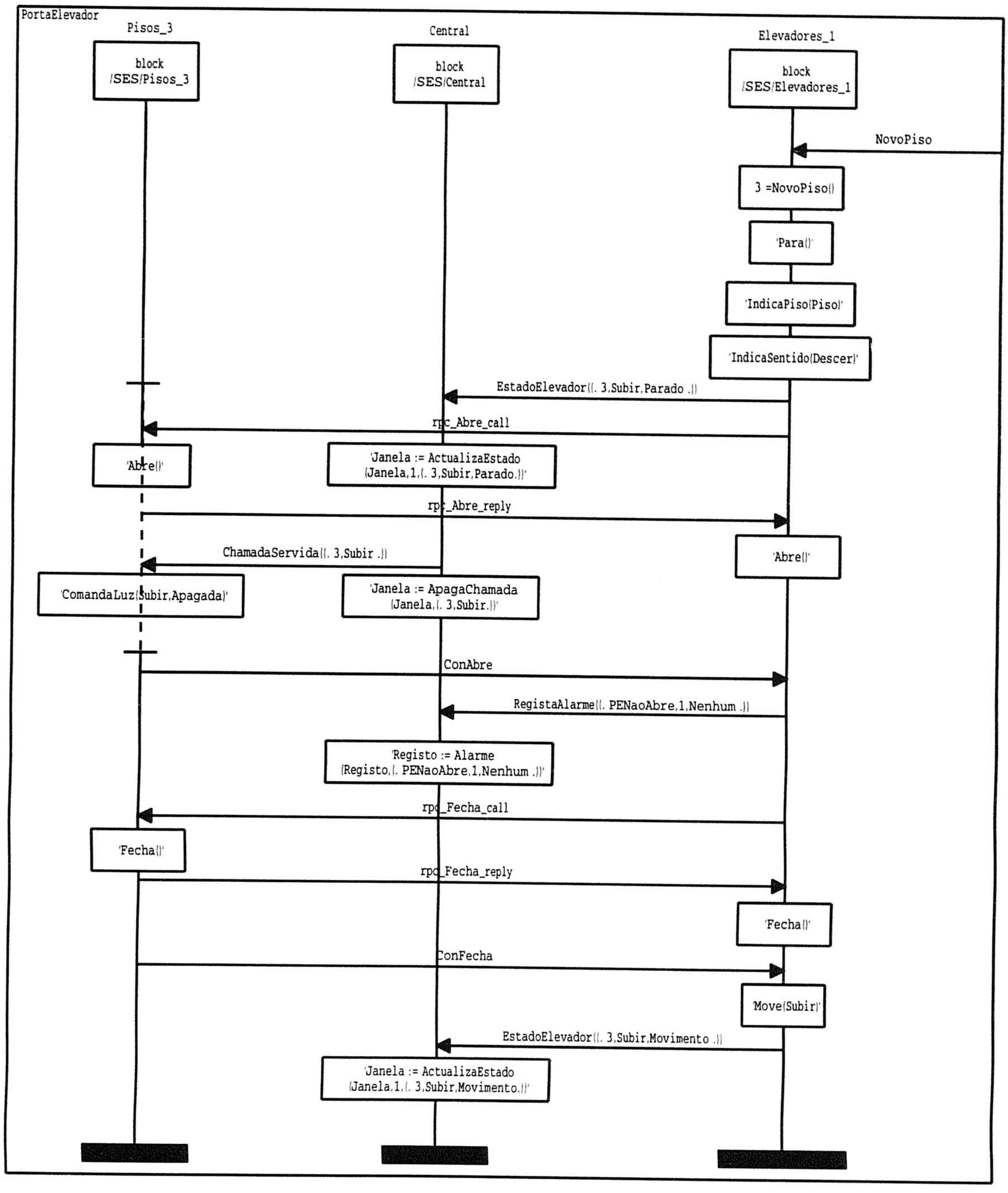

SES
OperacaoComErros
OperacaoNormal
PortaNaoAbre
PortaNaoFecha
Ruptura
RecebePedidos
EfectuaViagens
PortaPiso
PortaElevador
PortaPiso
PortaElevador
Pedido
Viagem
Chamada
Destino
PassaPorPiso
Chegada
PortaAberta
LivreOcupado
Rechamante
Guardada
IniciaViagem
Rechamante
Guardado
IniciaViagem
Passagem
RecebeChamada
NaoRecebeChamada
PortaAberta
ElevadorLivre
ElevadorOcupadoFechaPorta
PortaFechadaElevadorOcupado
RecebeChamada
NaoRecebeChamada
MantemPortaAberta
CargaExcessiva
FechaPortaElevadorLivre
FechaAbre
FechaEntrou
FechaAutomaticoAbre
FechaAutomaticoEntrouida
CargaExcessivaCargaCorrecta
FechaAutomaticoCargaExcessiva
FechaCargaExcessiva

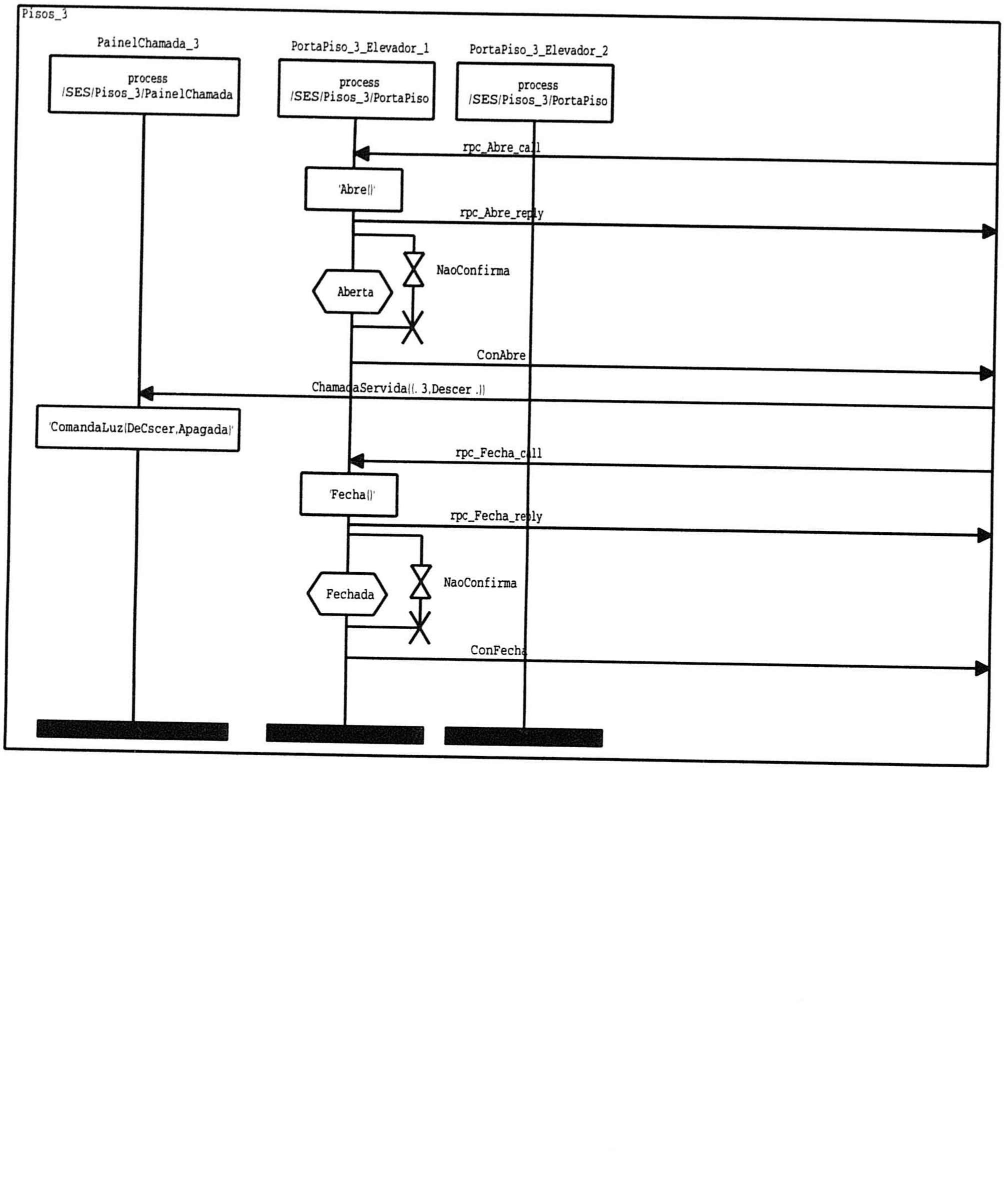

PortaPiso
Pisos_3
block /SES/Pisos_3
Central
block /SES/Central
Elevadores_1
block /SES/Elevadores_1
NovoPiso
3 =NovoPiso()
'IndicaPiso(Piso)'
'Para()'
'IndicaSentido(Descer)'
EstadoElevador({. 3,Subir,Parado .})
rpc_Abre_call
'Abre()'
'Janela := ActualizaEstado (Janela,1,{. 3,Subir,Parado .})'
rpc_Abre_reply
'Abre()'
ChamadaServida({. 3,Subir .})
'ComandaLuz(Subir,Apagada)'
'Janela := ApagaChamada (Janela,{. 3,Subir.})'
RegistaAlarme({. PPNaoAbre,Nenhum,3 .})
RegistaAlarme({. PPNaoConAberta,1,3 .})
'Registo := Alarme (Registo,{. PPNaoAbre,Nenhum,3 .})'
'Registo := Alarme (Registo,{. PPNaoConAberta,1,3 .})'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
'Move(Subir)'
EstadoElevador({. 3,Subir,Movimento .})
'Janela := ActualizaEstado (Janela,1,{. 3,Subir,Movimento.})'

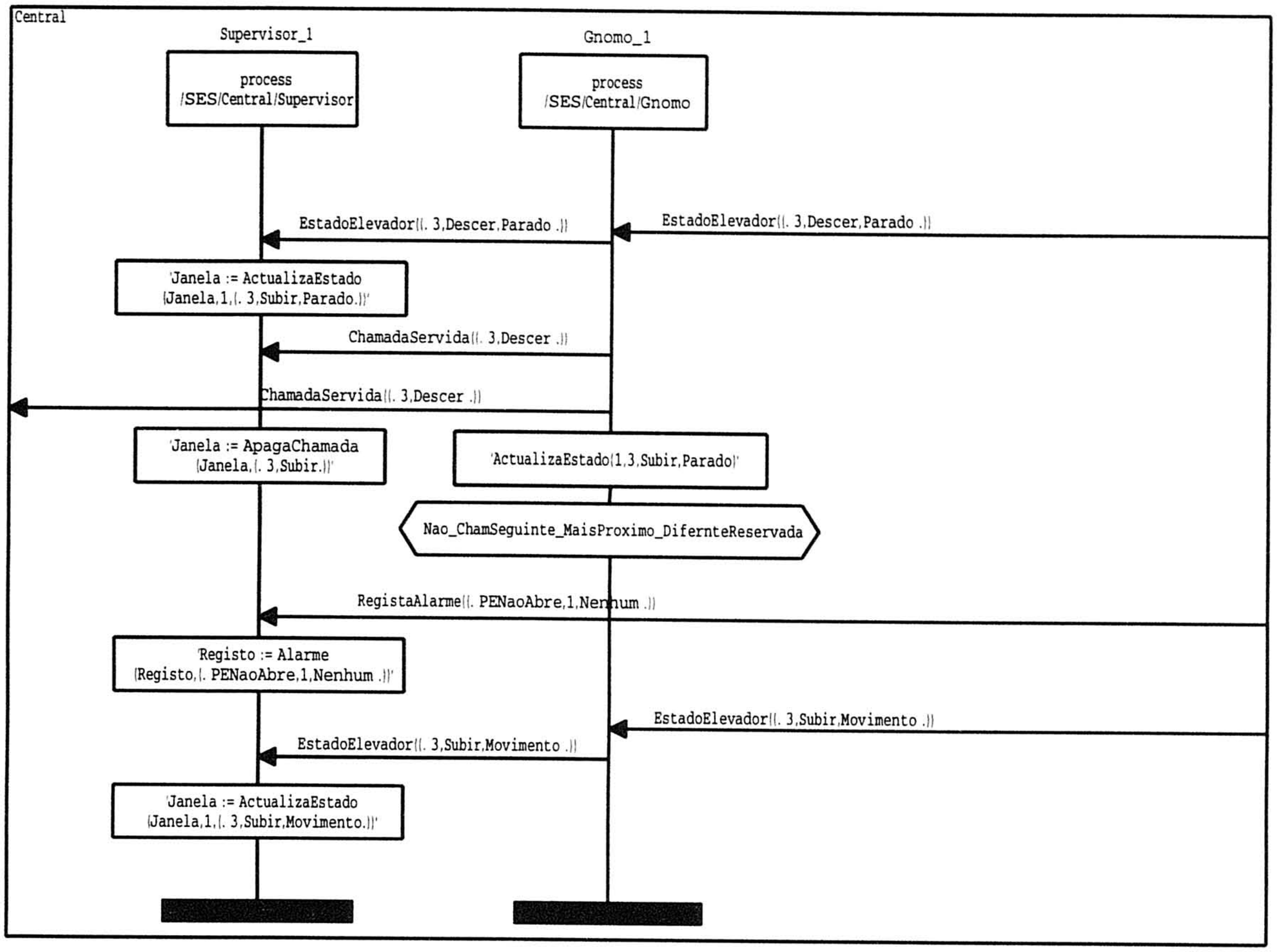

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
RegistaAlarme(. PPNaoAbre,1,3 .)
ChamadaServida(. 3,Descer .)
'ComandaLuz(DeCscer,Apagada)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
Fechada
NaoConfirma
ConFecha

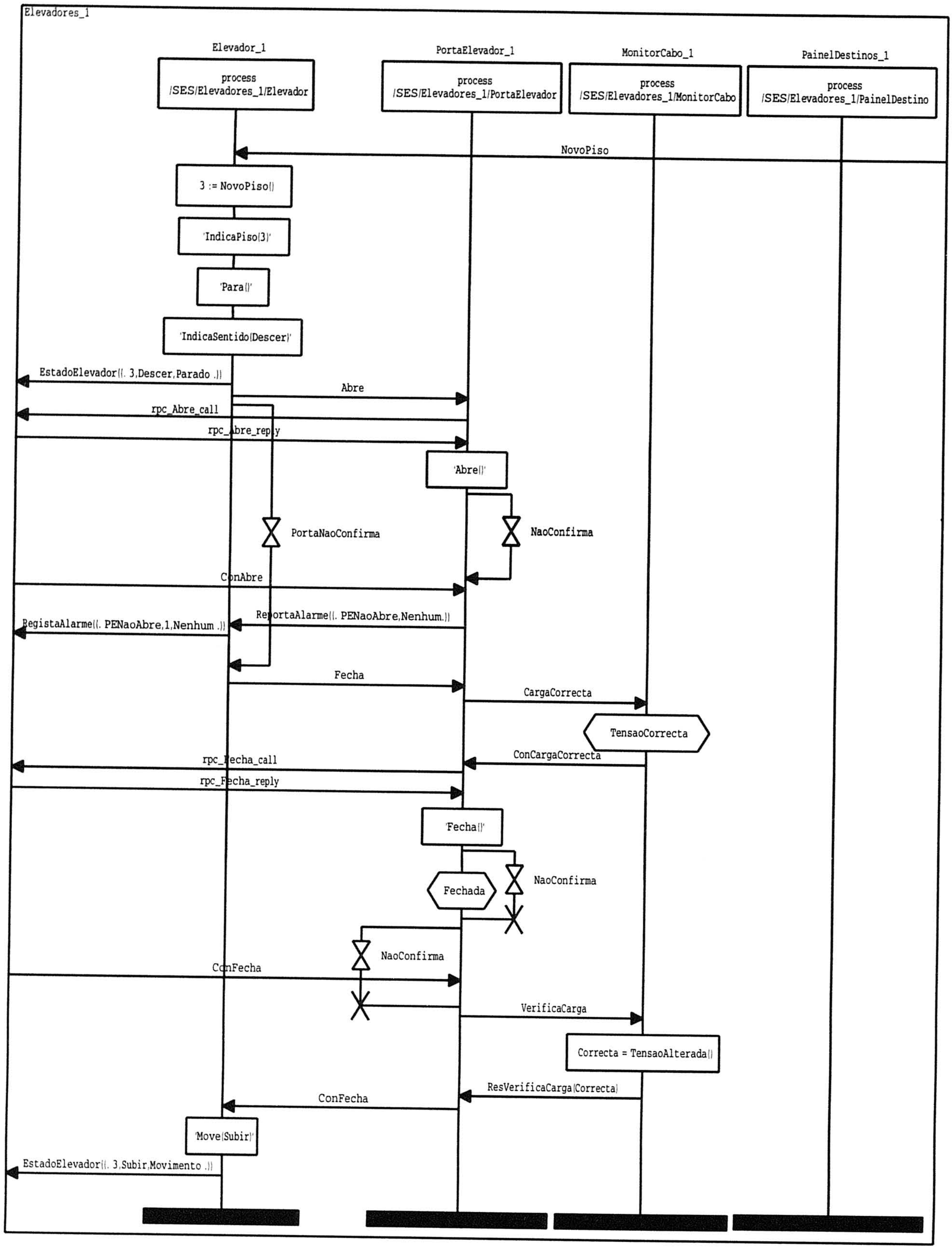

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Parado .|}
EstadoElevador{|. 3,Descer,Parado .|}
'Janela := ActualizaEstado (Janela,1,{. 3,Subir,Parado.|}'
ChamadaServida{|. 3,Descer .|}
ChamadaServida{|. 3,Descer .|}
'Janela := ApagaChamada (Janela,{. 3,Subir.|}'
'ActualizaEstado(1,3,Subir,Parado)'
Nao_ChamSeguinte_MaisProximo_DifernteReservada
RegistaAlarme{|. PPNaoAbre,1,3 .|}
RegistaAlarme{|. PPNaoConAberta,1 3 .|}
'Registo := Alarme (Registo,{. PPNaoAbre,Nenhum,3 .|}'
'Registo := Alarme (Registo,{. PPNaoConAberta,1,3 .|}'
EstadoElevador{|. 3,Subir,Movimento .|}
EstadoElevador{|. 3,Subir,Movimento .|}
'Janela := ActualizaEstado (Janela,1,{. 3,Subir,Movimento.|}'

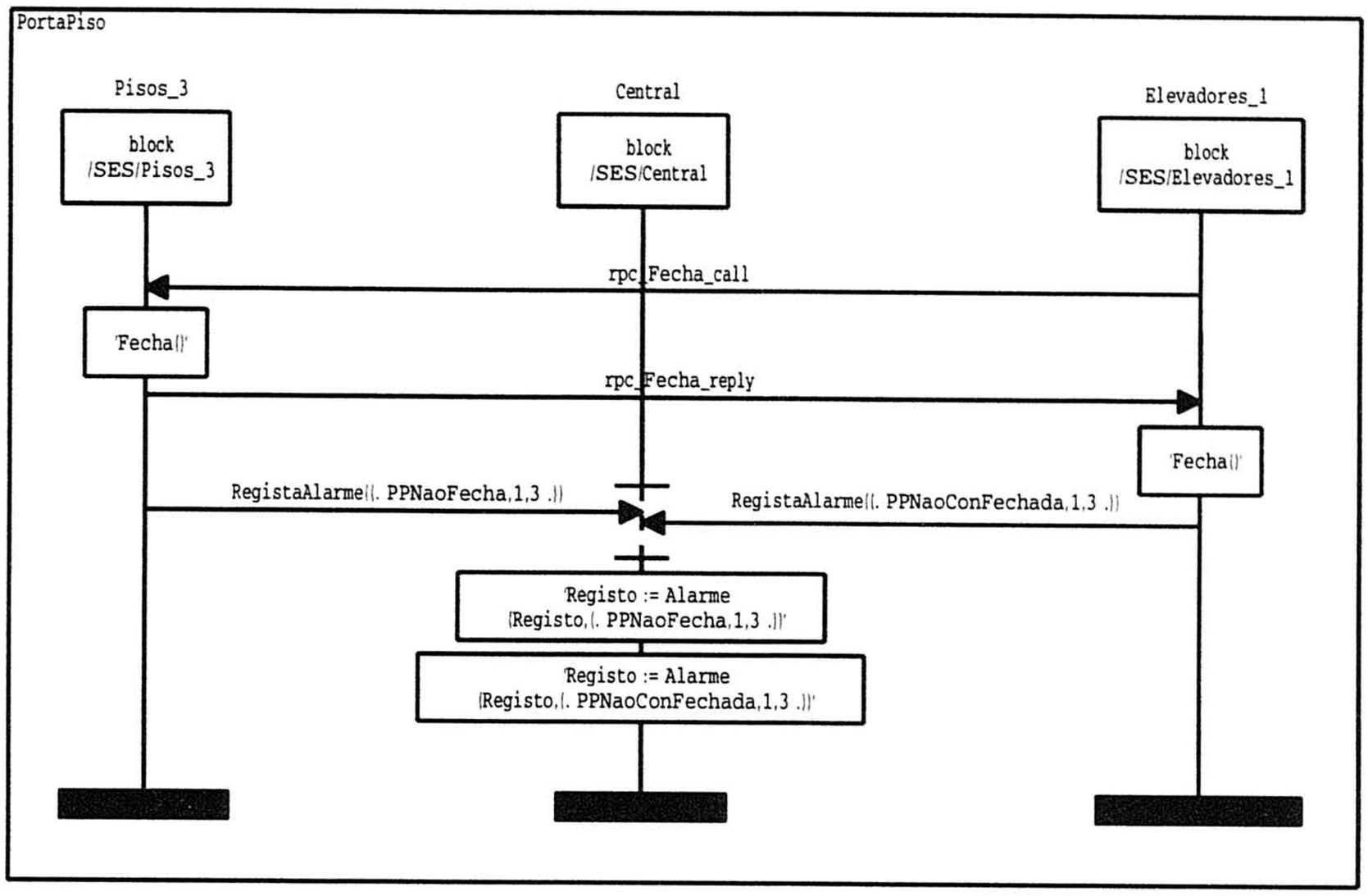

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
NovoPiso
3 := NovoPiso()
'IndicaPiso(3)'
'Para()'
'IndicaSentido(Descer)'
EstadoElevador(. 3,Descer,Parado .)
Abre
rpc_Abre_call
rpc_Abre_reply
'Abre()'
PortaNaoConfirma
Aberta
NaoConfirma
NaoConfirma
RegistaAlarme(. PPNaoConAberta,1,3 .)
ReportaAlarme(. PPNaoConAberta,3.)
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
Fechada
NaoConfirma
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
'Move(Subir)'
EstadoElevador(. 3,Subir,Movimento .)

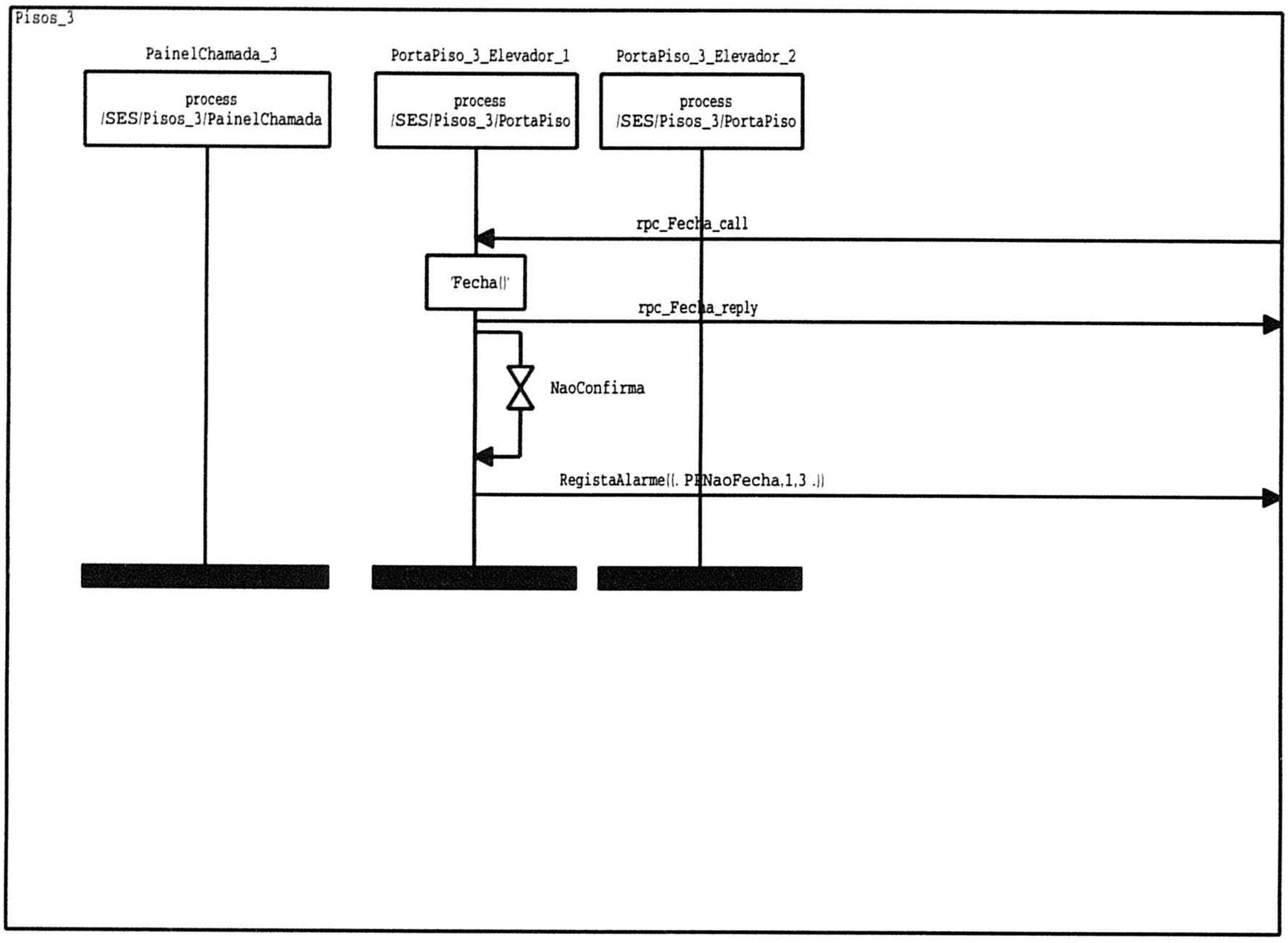

PortaElevador
Pisos_3
block /SES/Pisos_3
Central
block /SES/Central
Elevadores_1
block /SES/Elevadores_1
NovoPiso
3 =NovoPiso()
'Para()'
'IndicaPiso(Piso)'
'IndicaSentido(Descer)'
EstadoElevador({. 3,Subir,Parado .})
rpc_Abre_call
'Abre()'
'Janela := ActualizaEstado (Janela,1,{. 3,Subir,Parado.})'
rpc_Abre_reply
'Abre()'
ChamadaServida({. 3,Subir .})
'ComandaLuz(Subir,Apagada)'
'Janela := ApagaChamada (Janela,{. 3,Subir.})'
ConAbre
RegistaAlarme({. PENaoAbre,1,Nenhum .})
'Registo := Alarme (Registo,{. PENaoAbre,1,Nenhum .})'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
'Move(Subir)'
EstadoElevador({. 3,Subir,Movimento .})
'Janela := ActualizaEstado (Janela,1,{. 3,Subir,Movimento.})'

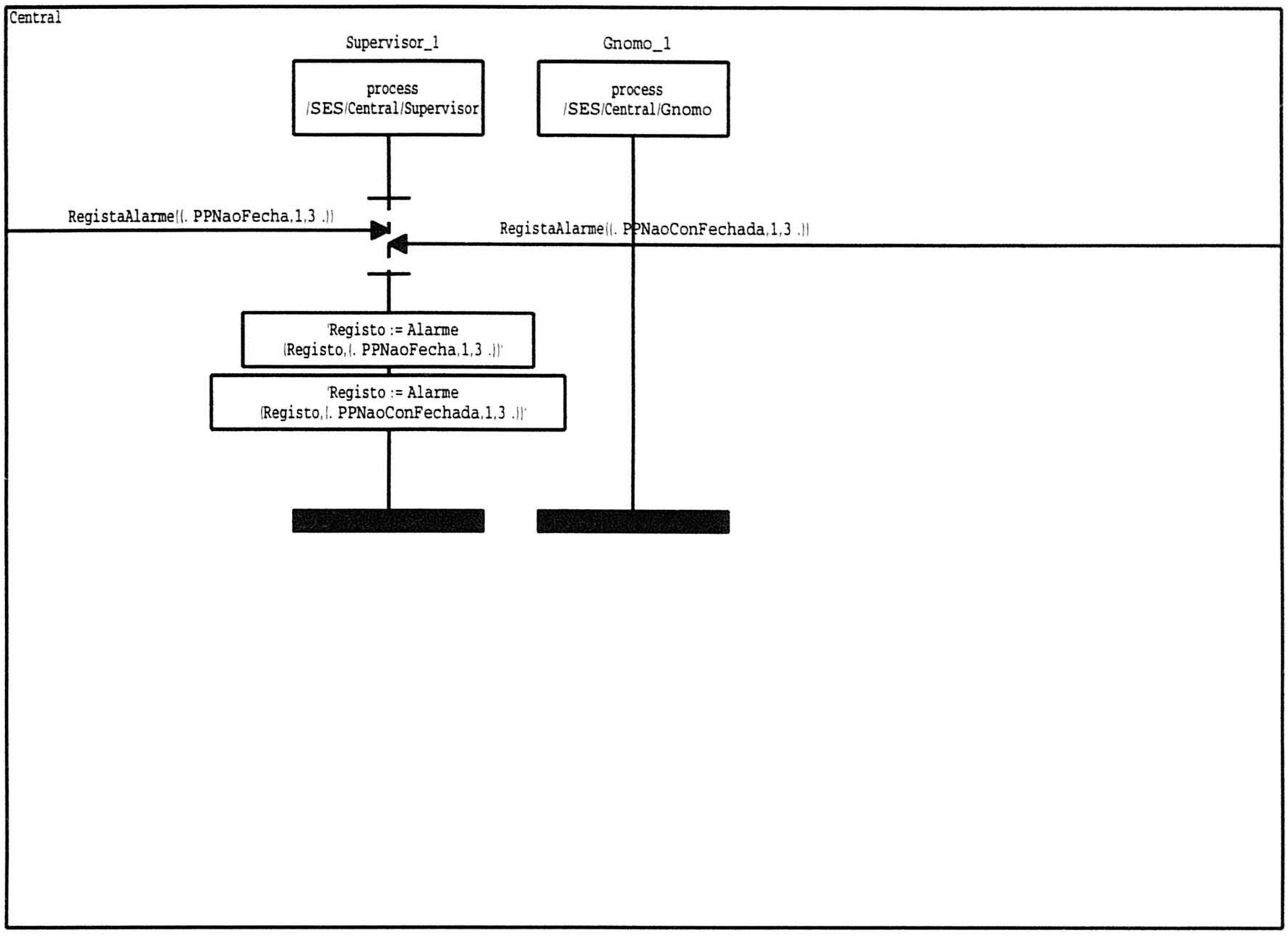

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre
ChamadaServida(. 3,Descer .)
'ComandaLuz(DeCscer,Apagada)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha

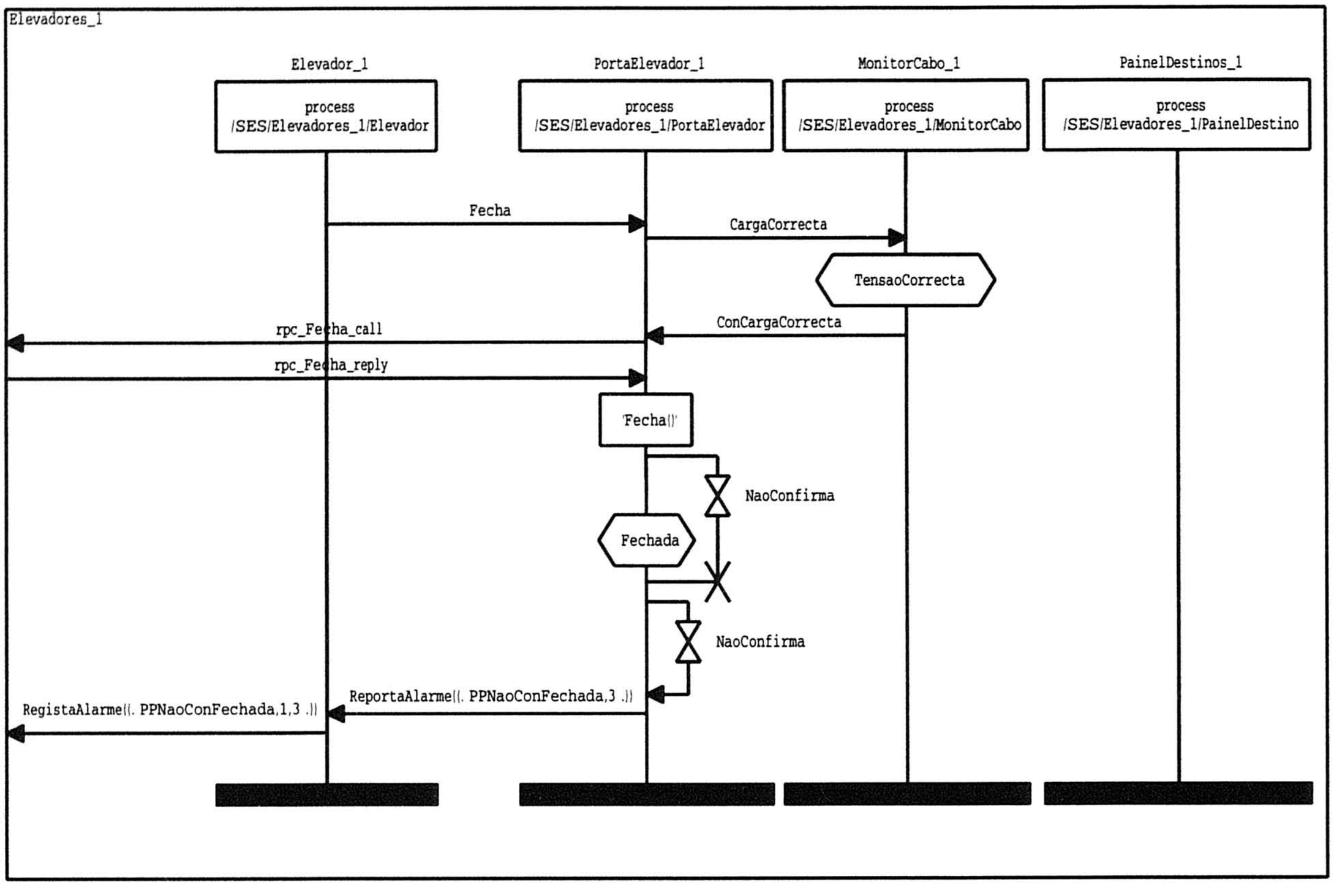

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Parado .|}
EstadoElevador{|. 3,Descer,Parado .|}
'Janela := ActualizaEstado (Janela,1,|. 3,Subir,Parado.|}'
ChamadaServida{|. 3,Descer .|}
ChamadaServida{|. 3,Descer .|}
'Janela := ApagaChamada (Janela,|. 3,Subir.|}'
'ActualizaEstado(1,3,Subir,Parado)'
Nao_ChamSeguinte_MaisProximo_DifernteReservada
RegistaAlarme{|. PENaoAbre,1,Nenhum .|}
'Registo := Alarme (Registo,|. PENaoAbre,1,Nenhum .|}'
EstadoElevador{|. 3,Subir,Movimento .|}
EstadoElevador{|. 3,Subir,Movimento .|}
'Janela := ActualizaEstado (Janela,1,|. 3,Subir,Movimento.|}'

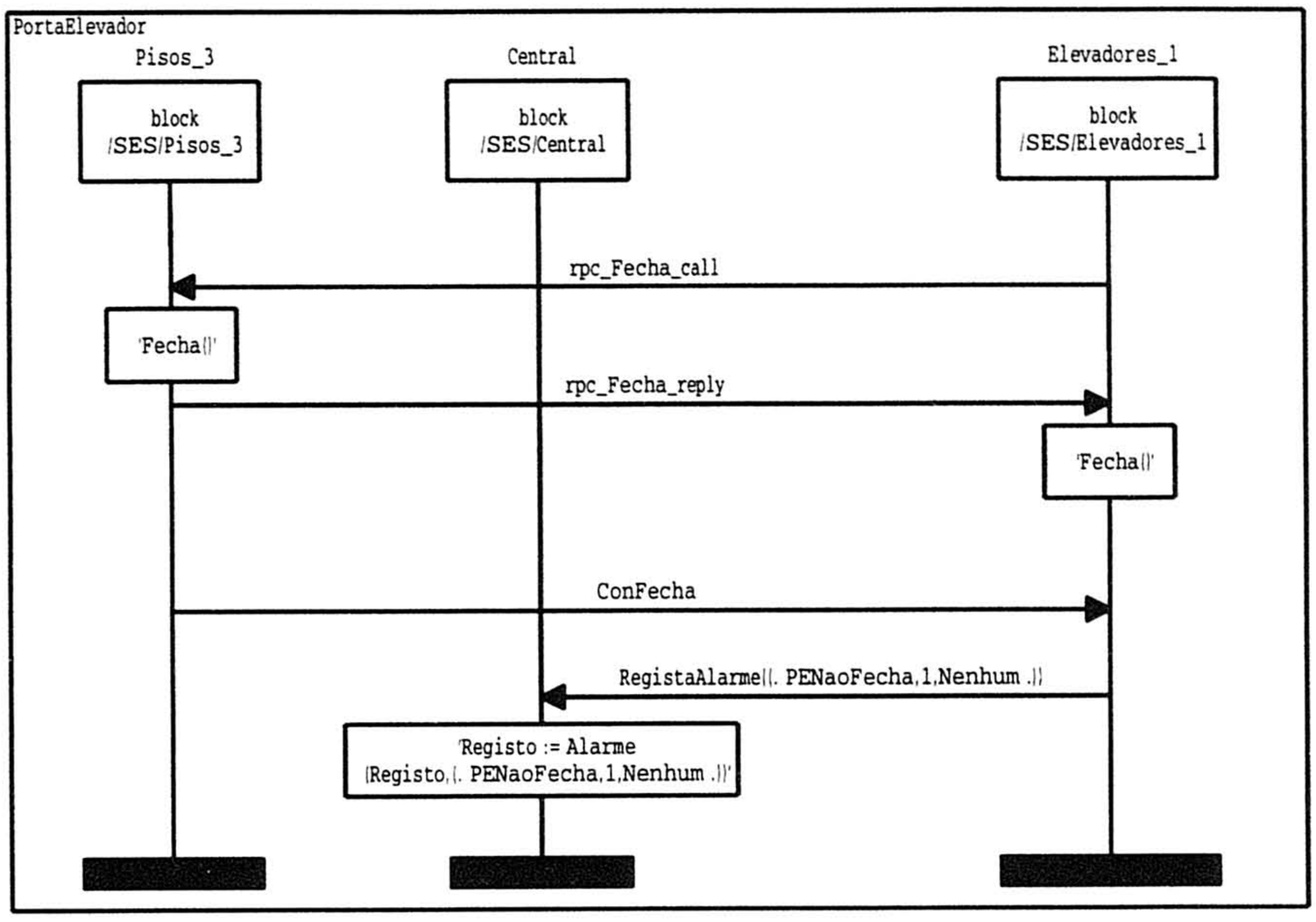

Elevadores_1
Elevador_1
process /SES/Elevadores_1/Elevador
PortaElevador_1
process /SES/Elevadores_1/PortaElevador
MonitorCabo_1
process /SES/Elevadores_1/MonitorCabo
PainelDestinos_1
process /SES/Elevadores_1/PainelDestino
NovoPiso
3 := NovoPiso()
'IndicaPiso(3)'
'Para()'
'IndicaSentido(Descer)'
EstadoElevador(. 3,Descer,Parado .)
Abre
rpc_Abre_call
rpc_Abre_reply
'Abre()'
PortaNaoConfirma
NaoConfirma
ConAbre
RegistaAlarme(. PENaoAbre,1,Nenhum .)
ReportaAlarme(. PENaoAbre,Nenhum.)
Fecha
CargaCorrecta
TensaoCorrecta
rpc_Fecha_call
ConCargaCorrecta
rpc_Fecha_reply
'Fecha()'
Fechada
NaoConfirma
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
'Move(Subir)'
EstadoElevador(. 3,Subir,Movimento .)

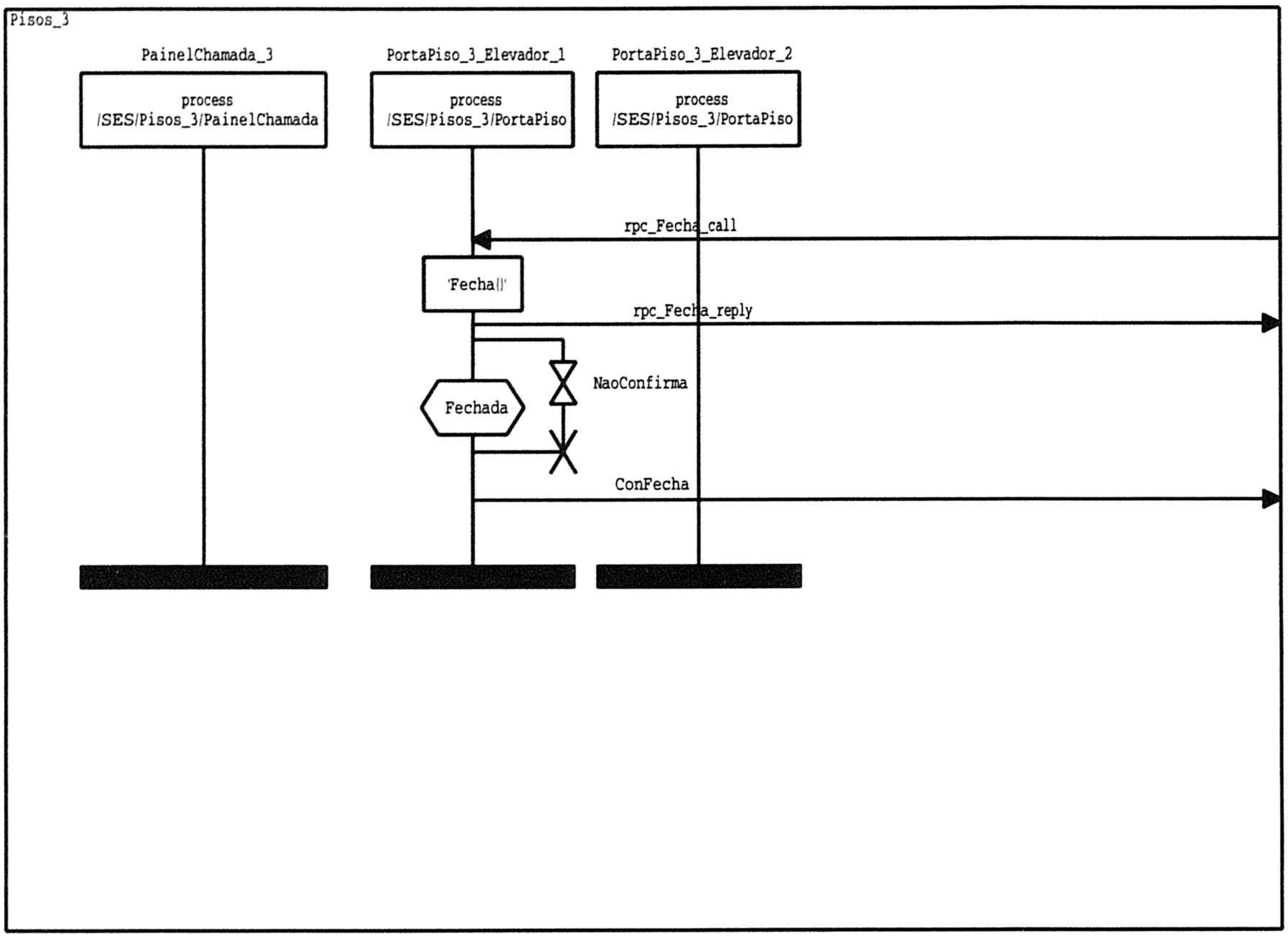

PortaPiso
Pisos_3
block
/SES/Pisos_3
Central
block
/SES/Central
Elevadores_1
block
/SES/Elevadores_1
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
RegistaAlarme(. PPNaoFecha,1,3 .)
RegistaAlarme(. PPNaoConFechada,1,3 .)
'Registo := Alarme
(Registo,(. PPNaoFecha,1,3 .))'
'Registo := Alarme
(Registo,(. PPNaoConFechada,1,3 .))'

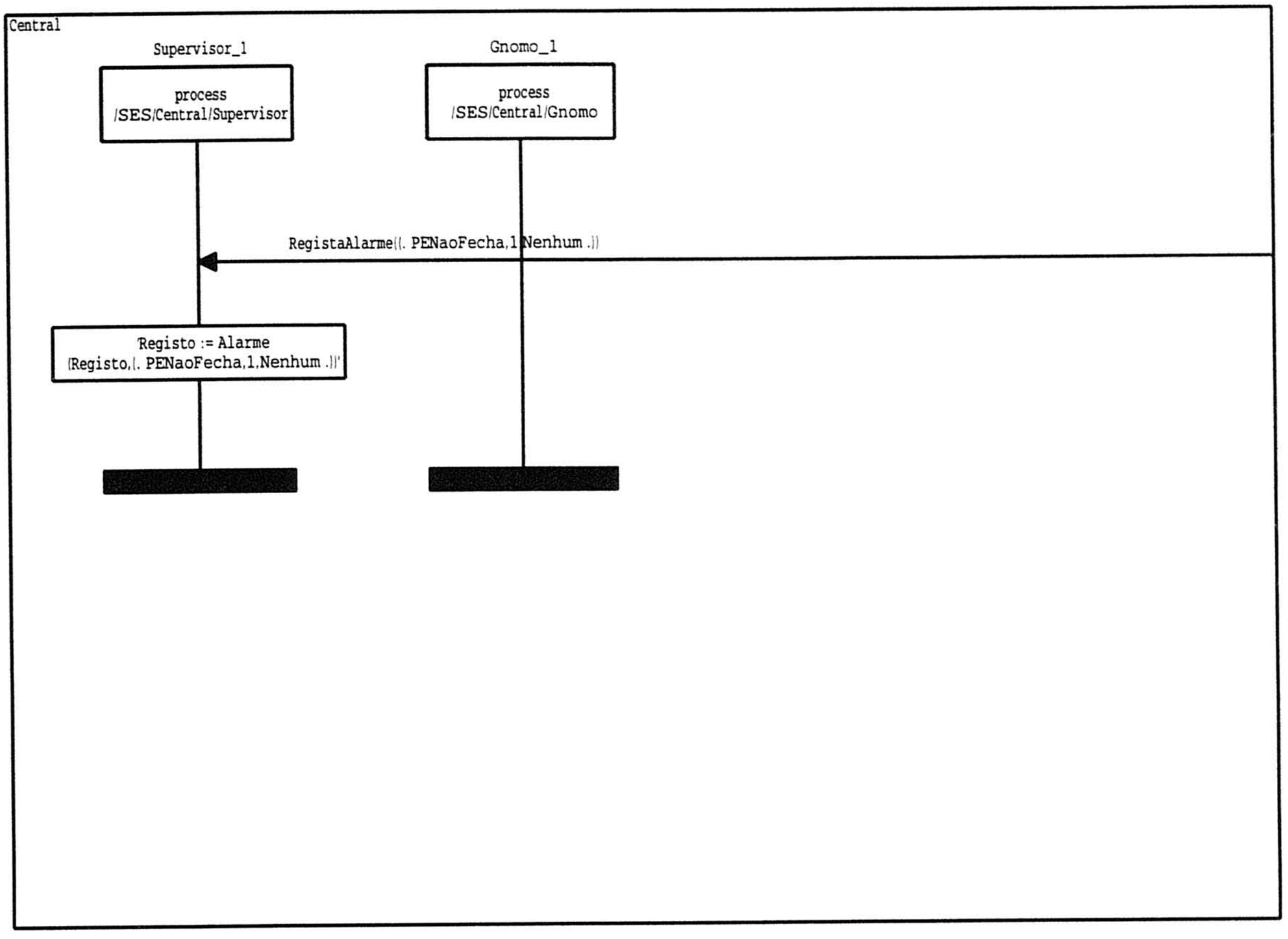

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
RegistaAlarme(|. PENaoFecha,1,3 .|)

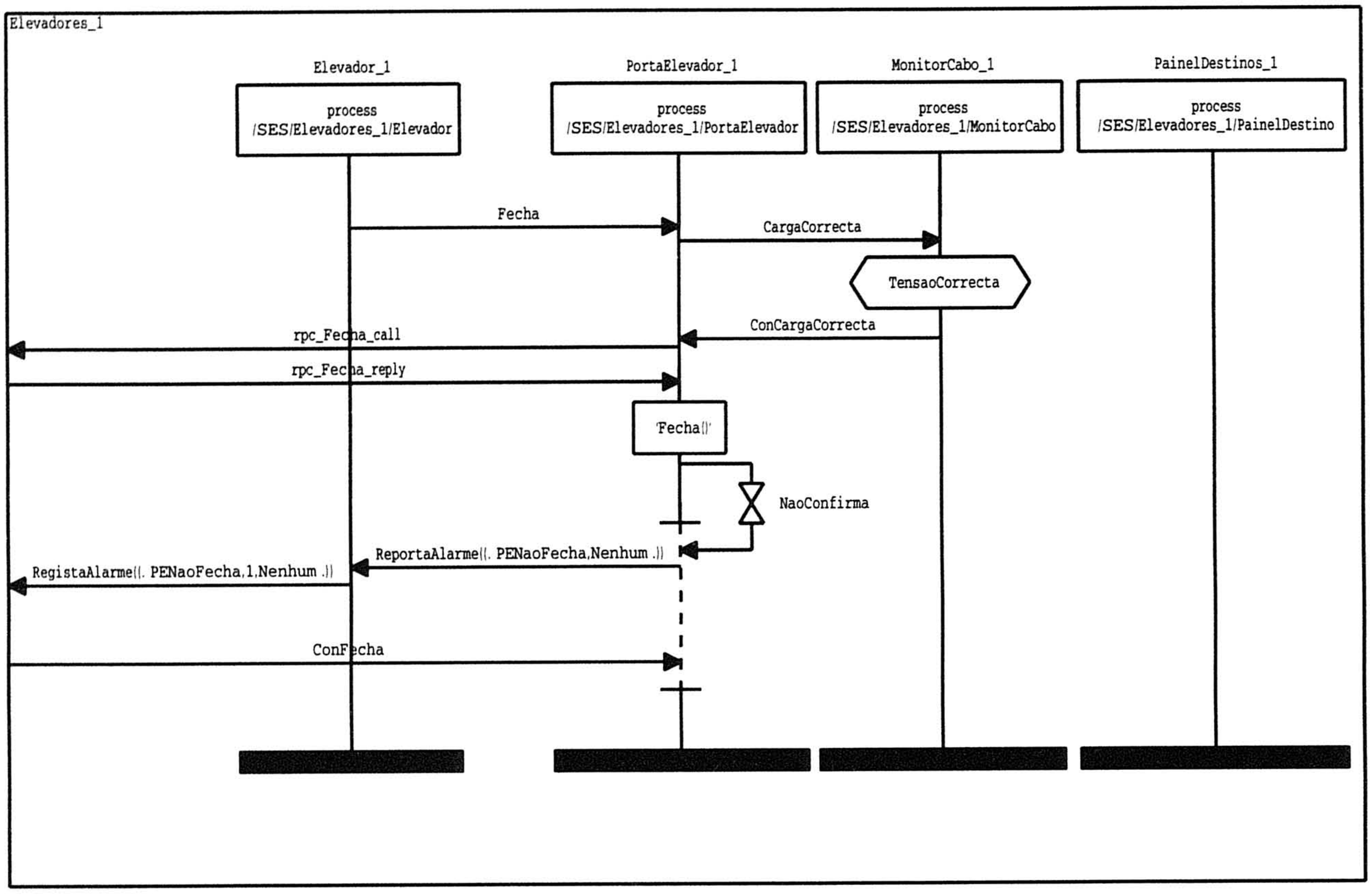

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
RegistaAlarme{|. PPNaoFecha,1,3 .|}
RegistaAlarme{|. PPNaoConFechada,1,3 .|}
'Registo := Alarme
(Registo,{. PPNaoFecha,1,3 .|}'
'Registo := Alarme
(Registo,|. PPNaoConFechada,1,3 .|}'

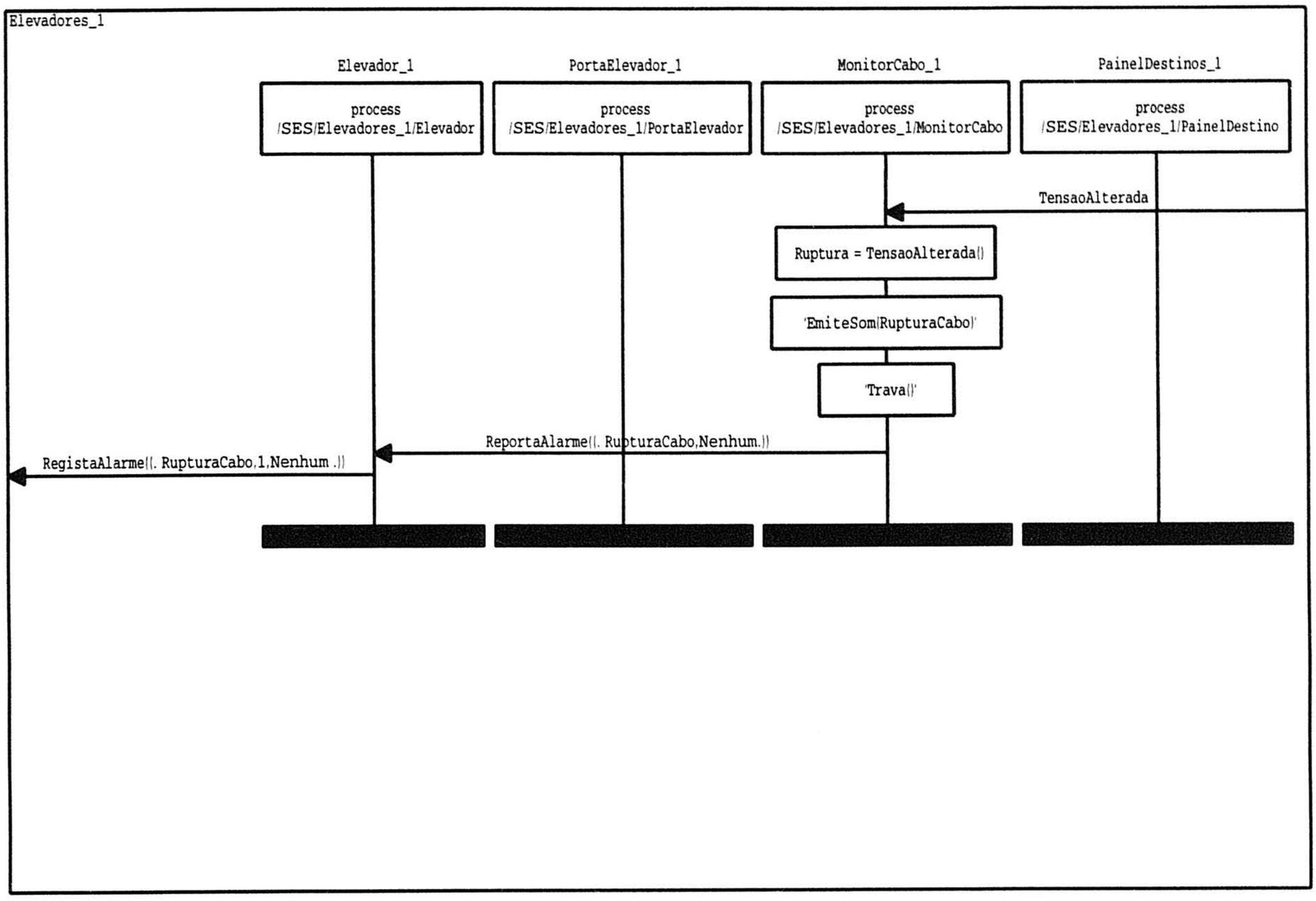

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
Fecha
CargaCorrecta
TensaoCorrecta
rpc_Fecha_call
ConCargaCorrecta
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Fechada
NaoConfirma
RegistaAlarme(. PPNaoConFechada,1,3 .)
ReportaAlarme(. PPNaoConFechada,3 .)

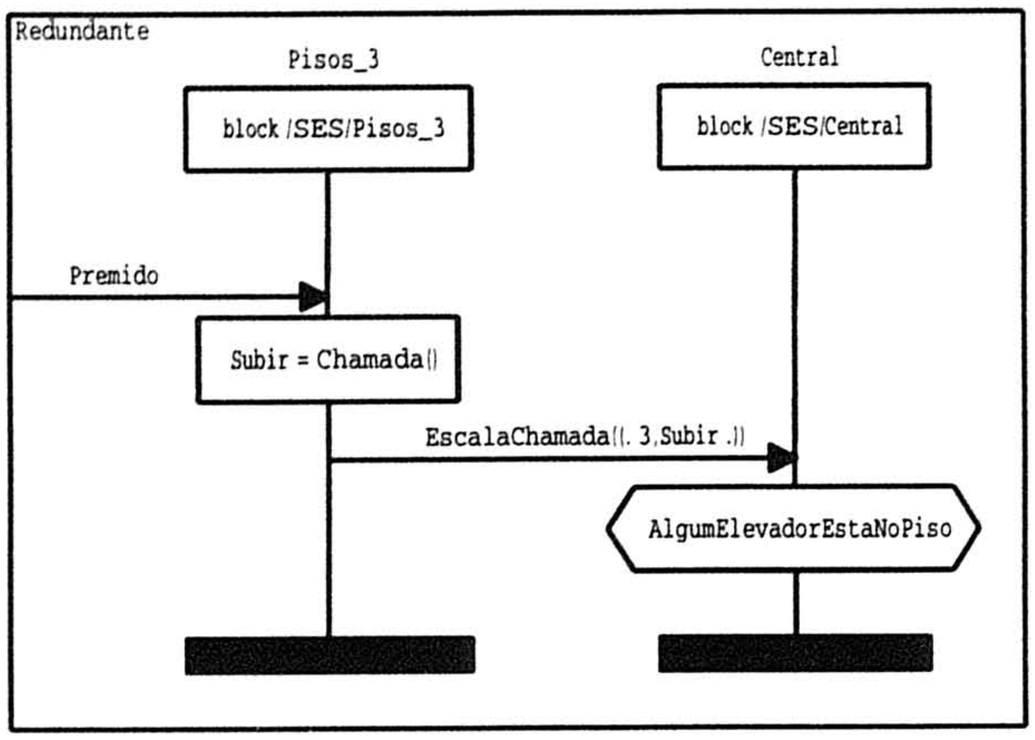

PortaElevador
Pisos_3
Central
Elevadores_1
block /SES/Pisos_3
block /SES/Central
block /SES/Elevadores_1
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
RegistaAlarme(. PENaoFecha,1,Nenhum .)
'Registo := Alarme (Registo,(. PENaoFecha,1,Nenhum .))'

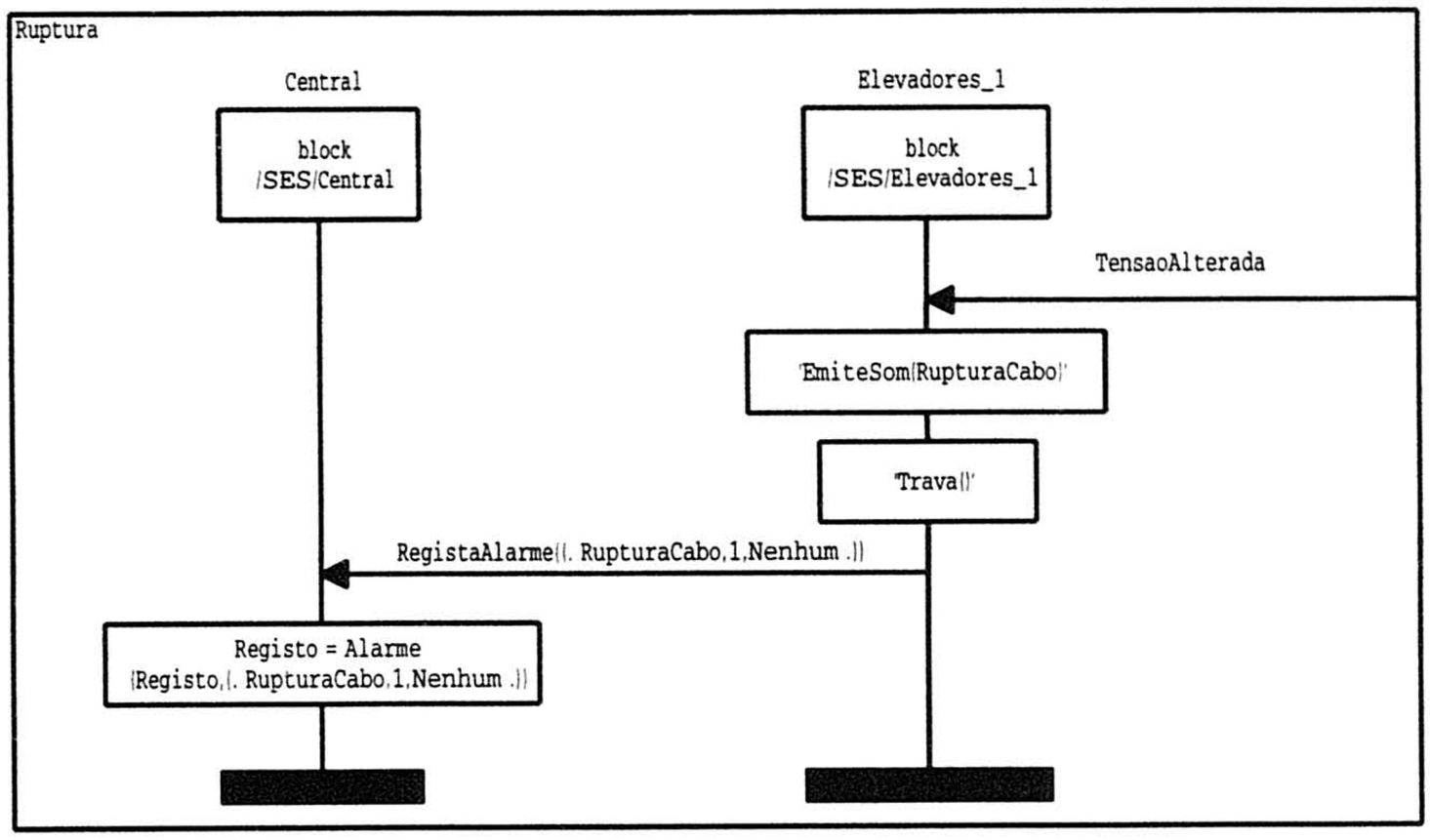

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha

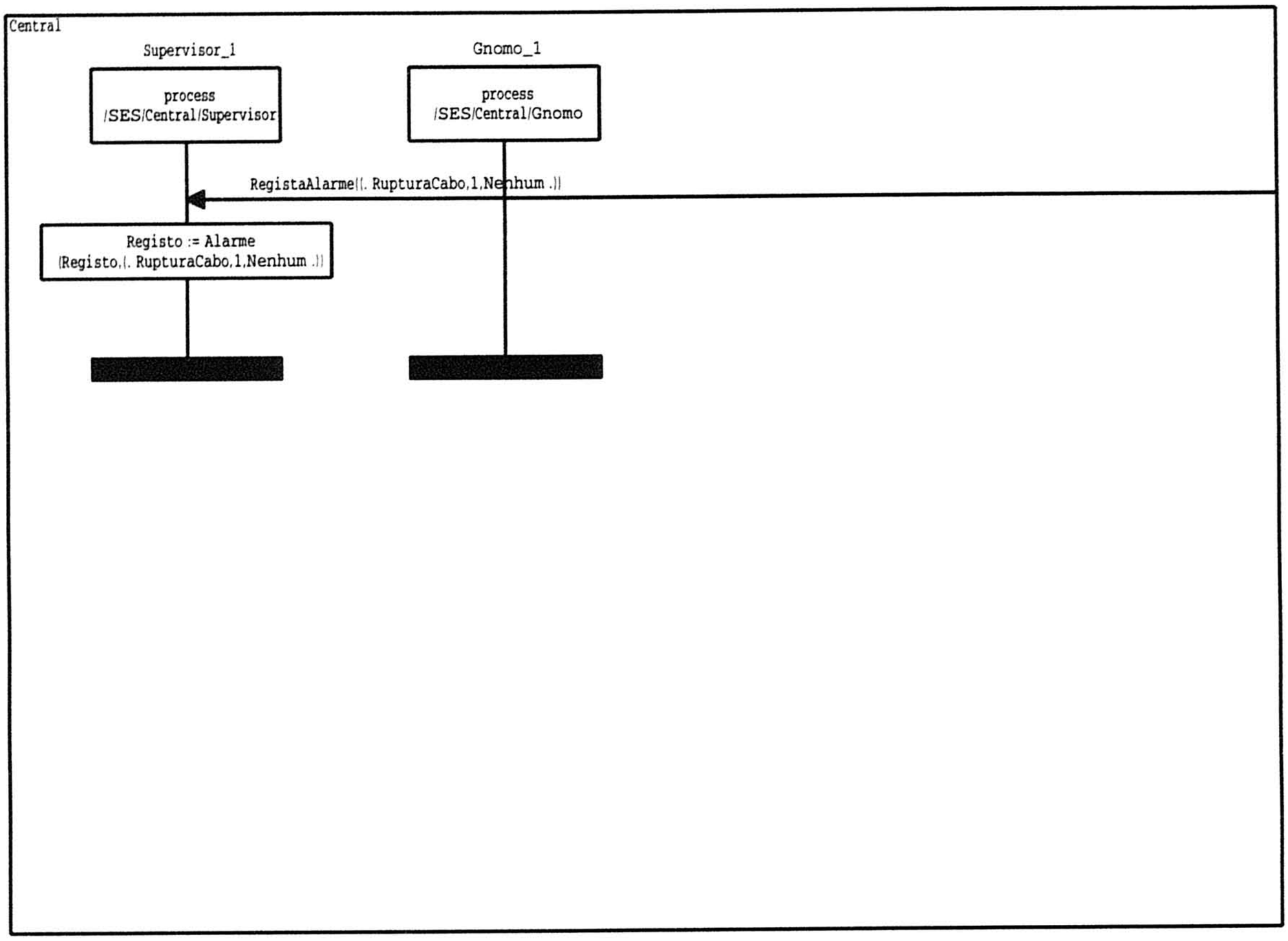

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
RegistaAlarme{(. PENaoFecha,1,Nenhum .)}
Registo := Alarme
(Registo,(. PENaoFecha,1,Nenhum .))'

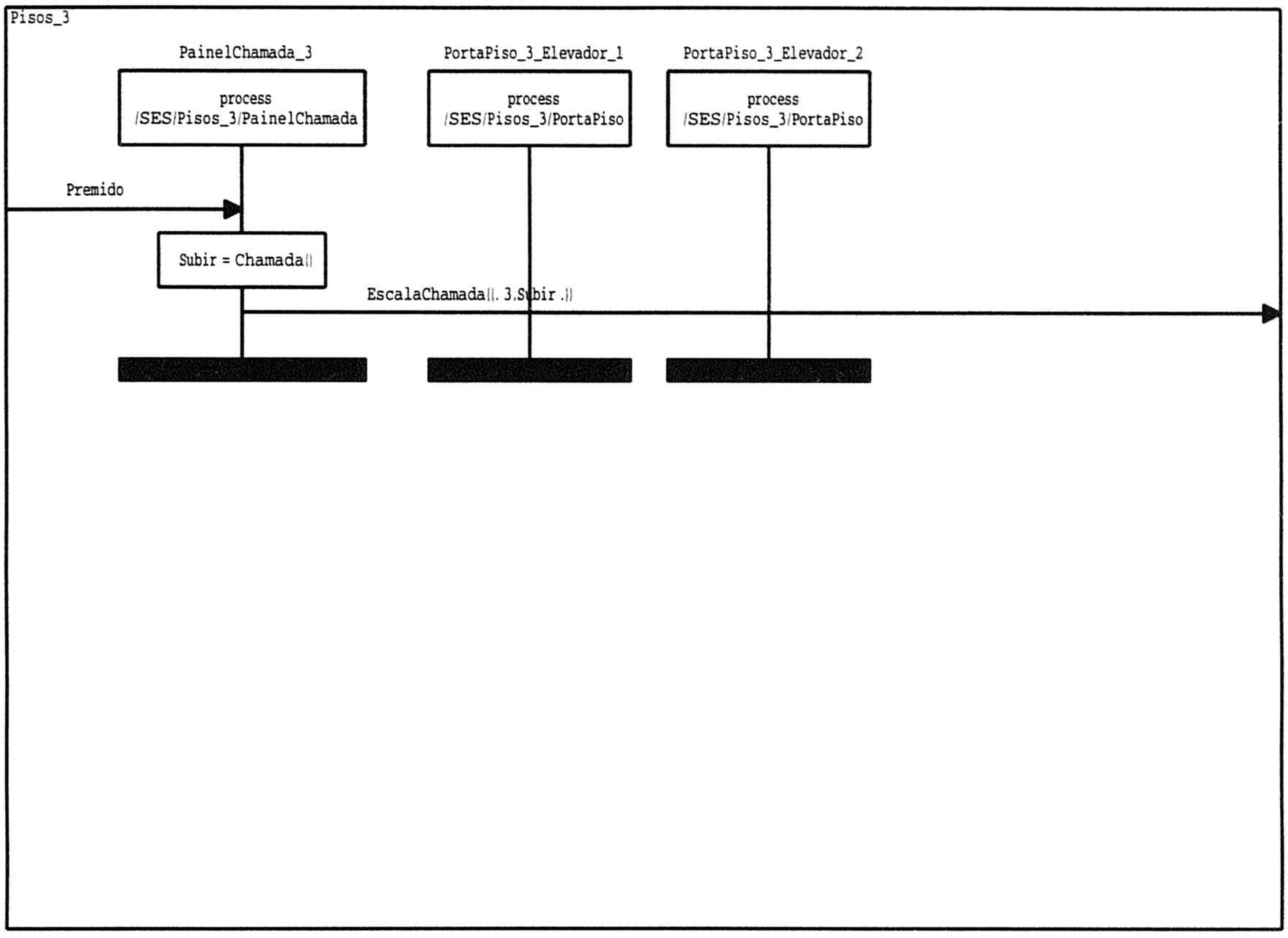

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
ReportaAlarme(|. PENaoFecha,Nenhum .|)
RegistaAlarme(|. PENaoFecha,1,Nenhum .|)
ConFecha

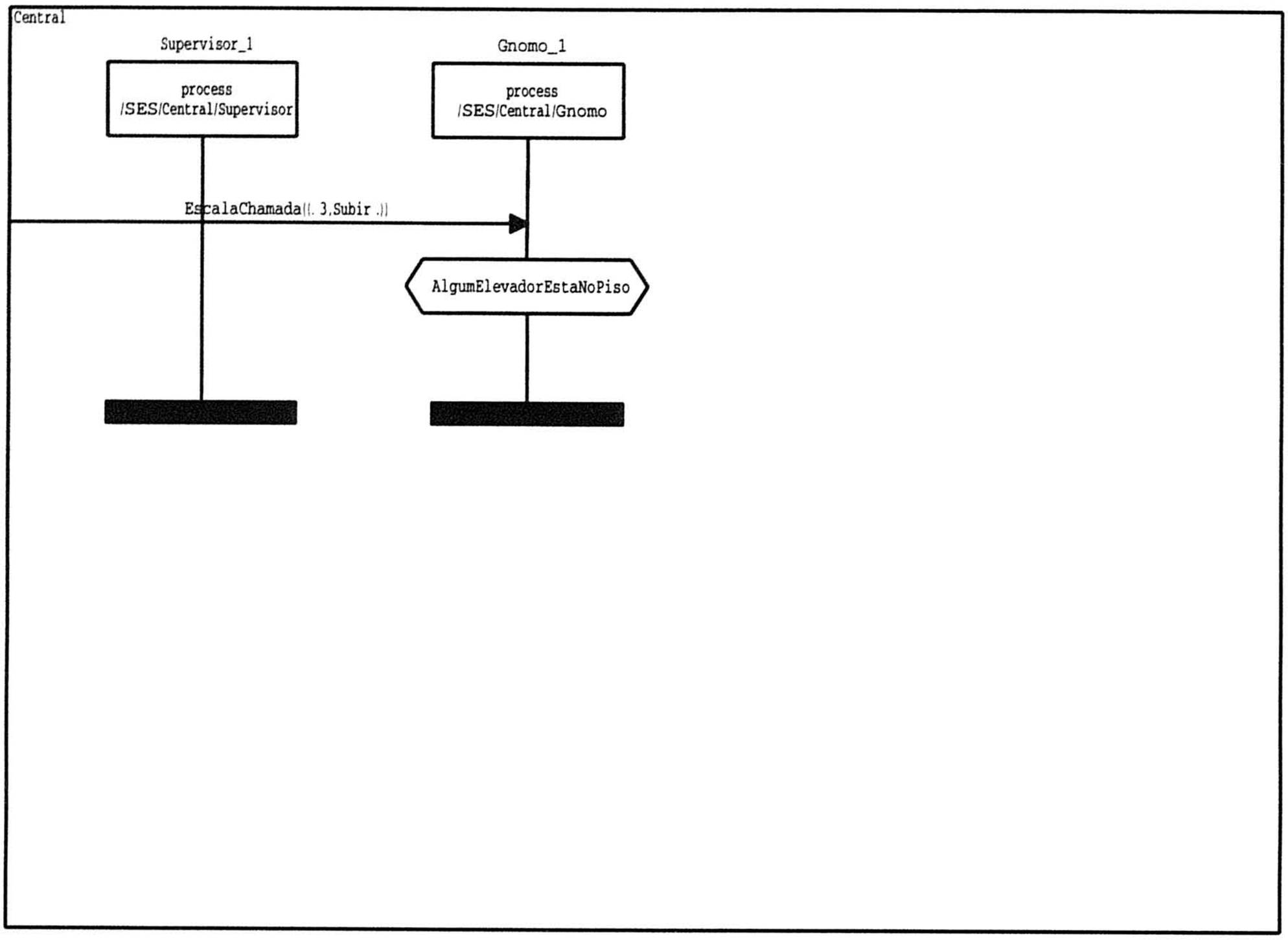

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
TensaoAlterada
Ruptura = TensaoAlterada()
'EmiteSom(RupturaCabo)'
'Trava()'
ReportaAlarme(. RupturaCabo,Nenhum.)
RegistaAlarme(. RupturaCabo,1,Nenhum .)

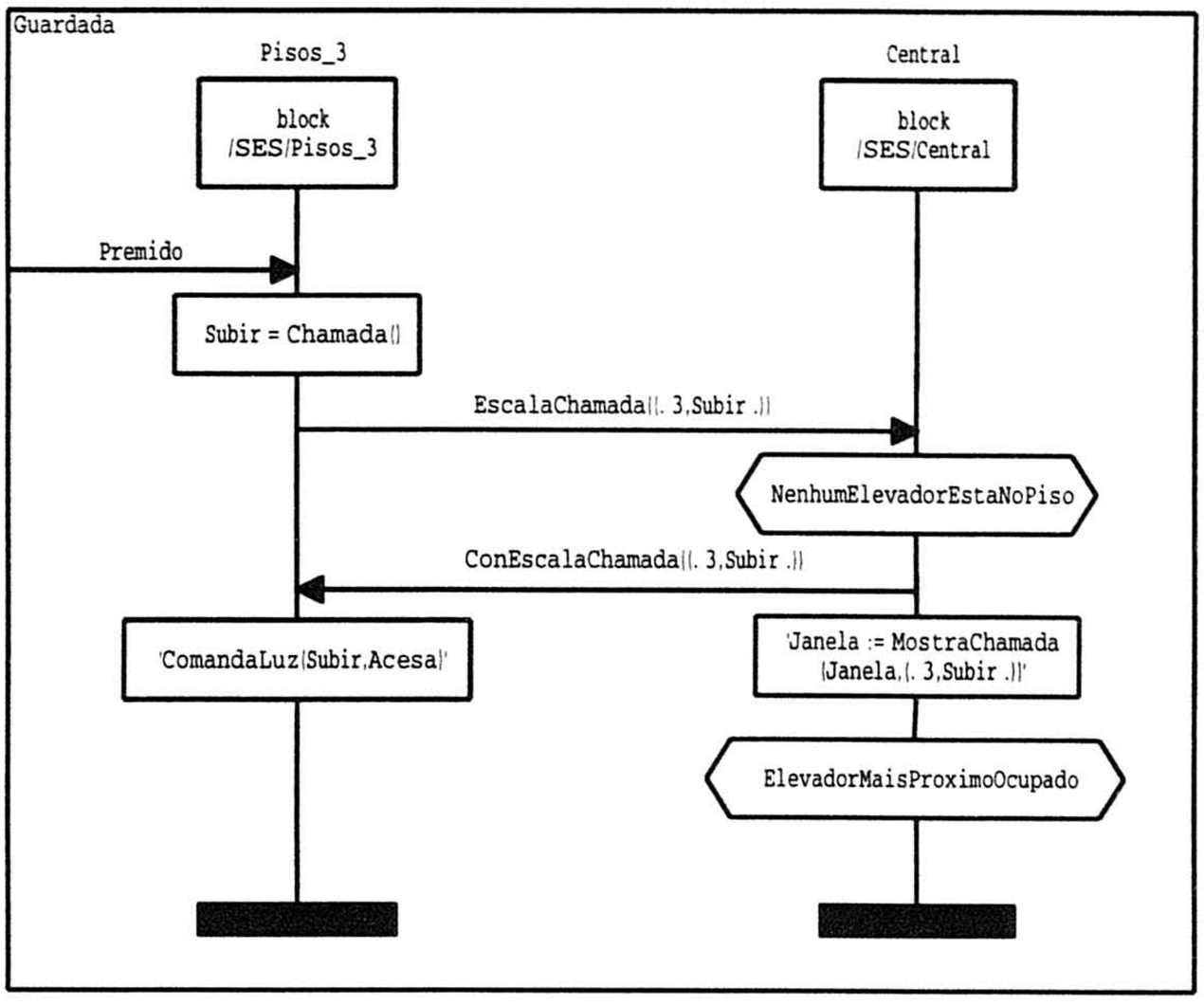

Redundante
Pisos_3
Central
block /SES/Pisos_3
block /SES/Central
Premido
Subir = Chamada()
EscalaChamada(1, 3, Subir .)
AlgumElevadorEstaNoPiso

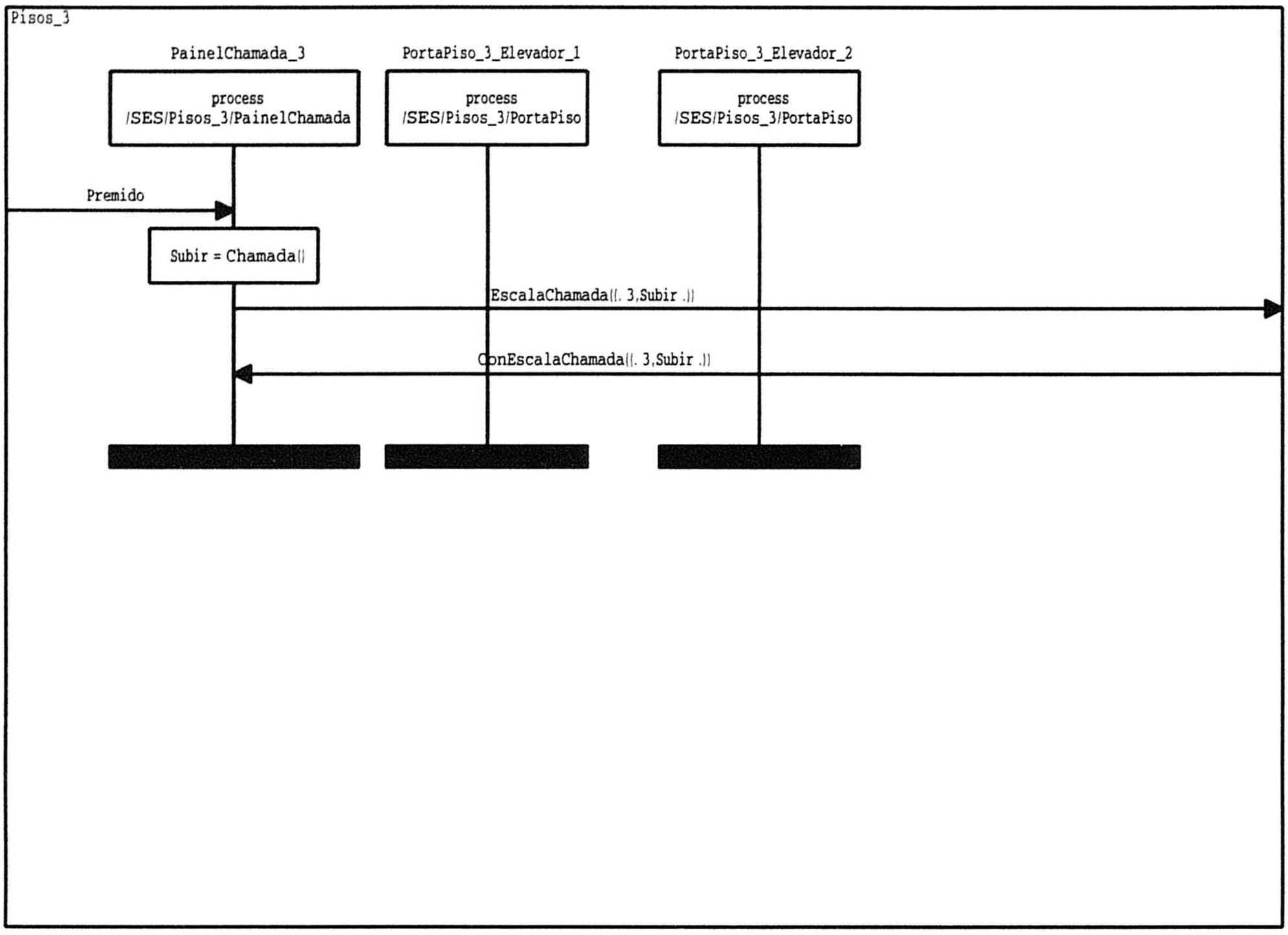

Ruptura
Central
Elevadores_1
block
/SES/Central
block
/SES/Elevadores_1
TensaoAlterada
'EmiteSom(RupturaCabo)'
'Trava()'
RegistaAlarme(|. RupturaCabo,1,Nenhum .|)
Registo = Alarme
(Registo,|. RupturaCabo,1,Nenhum .|)

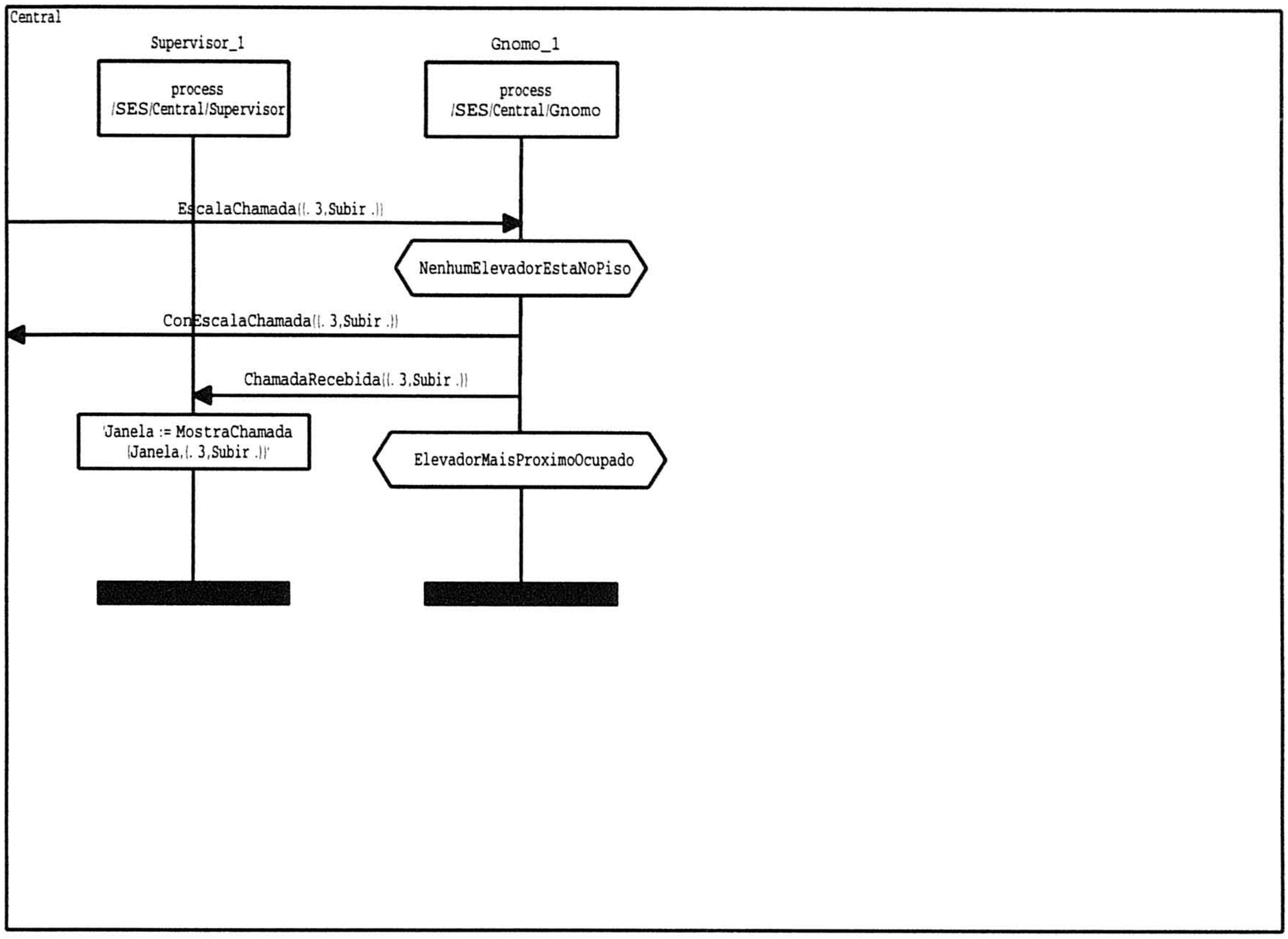

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
RegistaAlarme(|. RupturaCabo,1,Nenhum .|)
Registo := Alarme
(Registo,|. RupturaCabo,1,Nenhum .|)

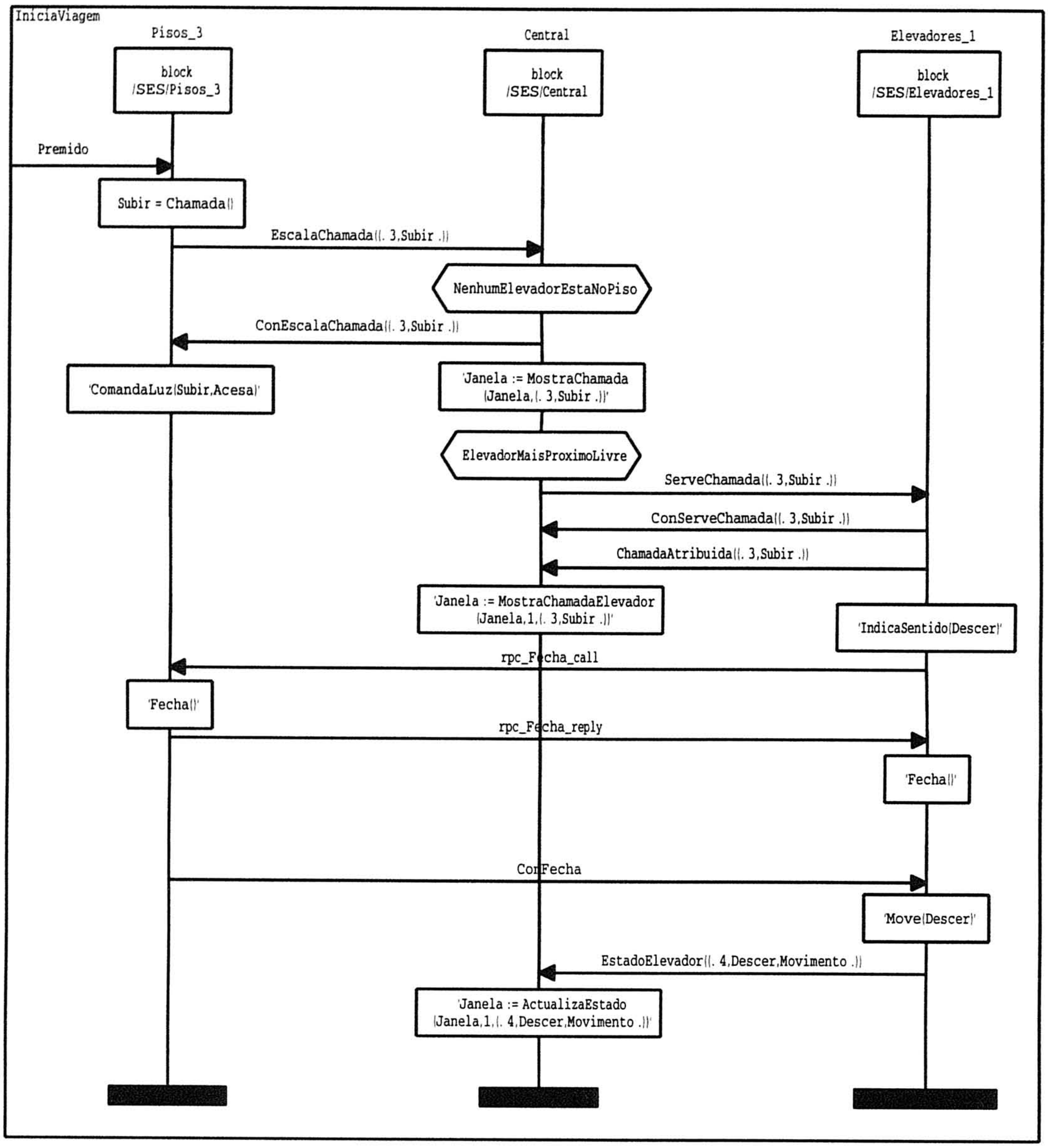

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process
/SES/Pisos_3/PainelChamada
process
/SES/Pisos_3/PortaPiso
process
/SES/Pisos_3/PortaPiso
Premido
Subir = Chamada()
EscalaChamada(. 3,Subir .)

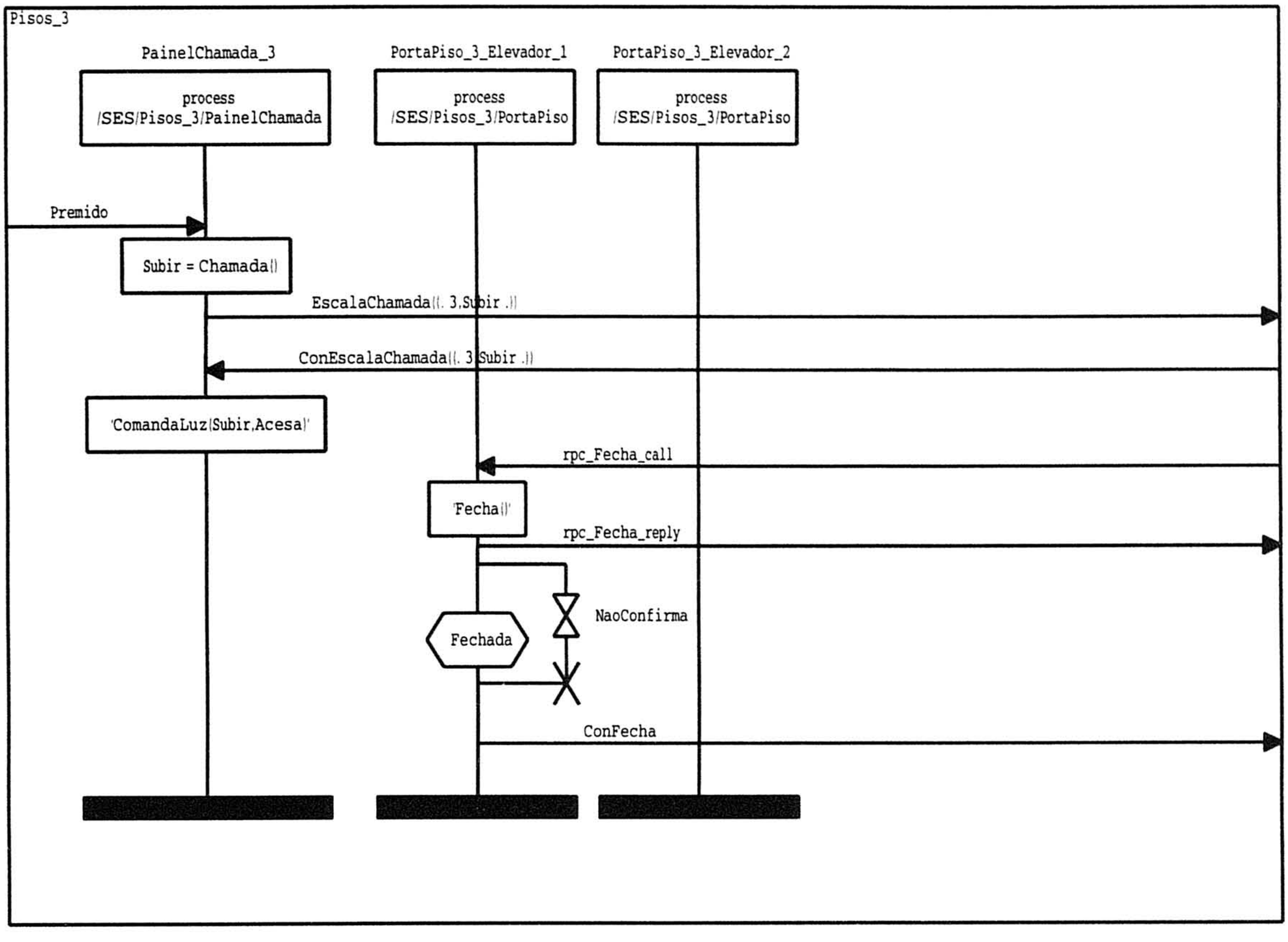

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
EscalaChamada{|. 3,Subir .|}
AlgumElevadorEstaNoPiso

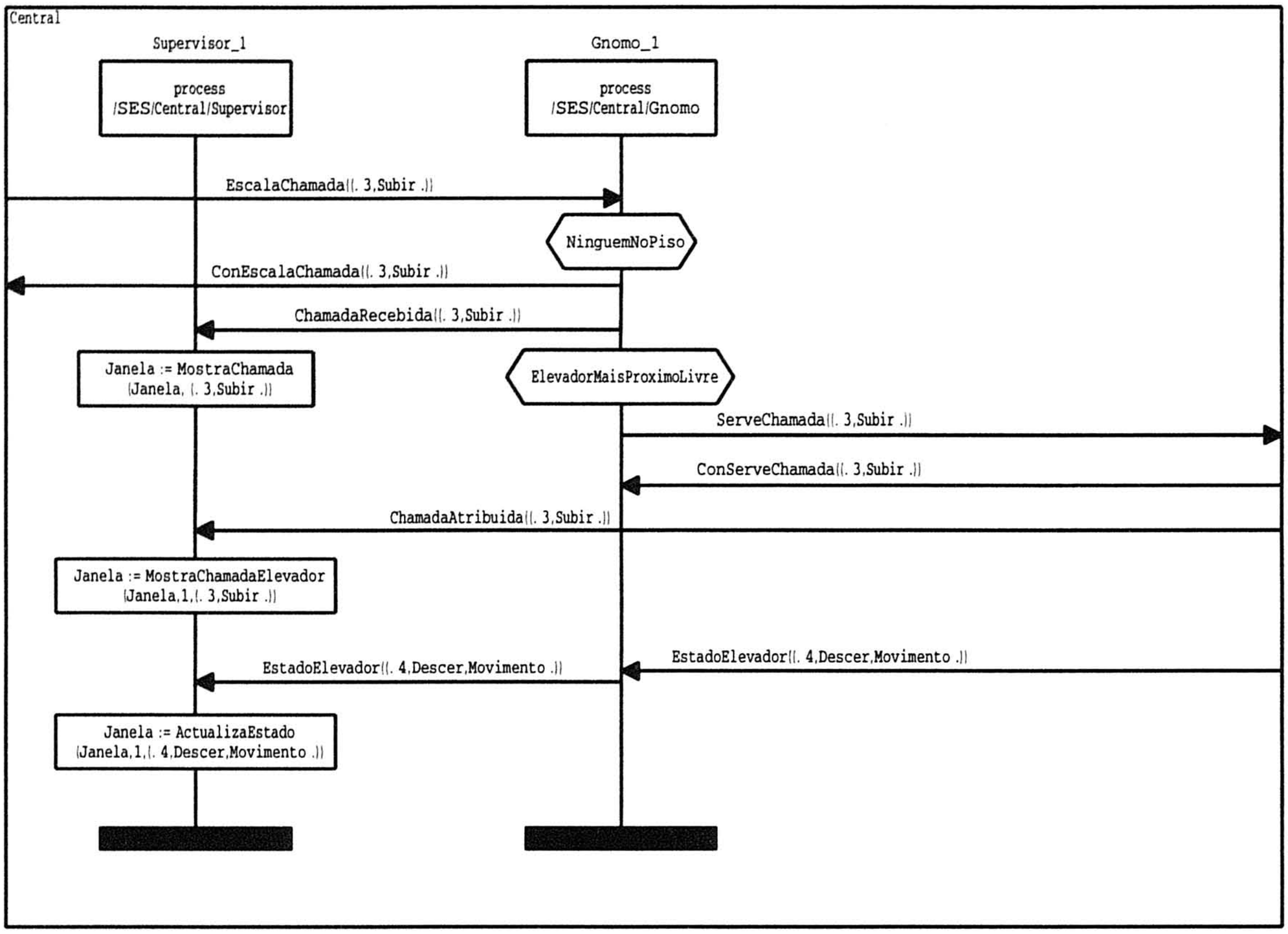

Guardada
Pisos_3
Central
block /SES/Pisos_3
block /SES/Central
Premido
Subir = Chamada()
EscalaChamada(1. 3,Subir .)1
NenhumElevadorEstaNoPiso
ConEscalaChamada(1. 3,Subir .)1
'ComandaLuz(Subir,Acesa)'
'Janela := MostraChamada (Janela,(. 3,Subir .))'
ElevadorMaisProximoOcupado

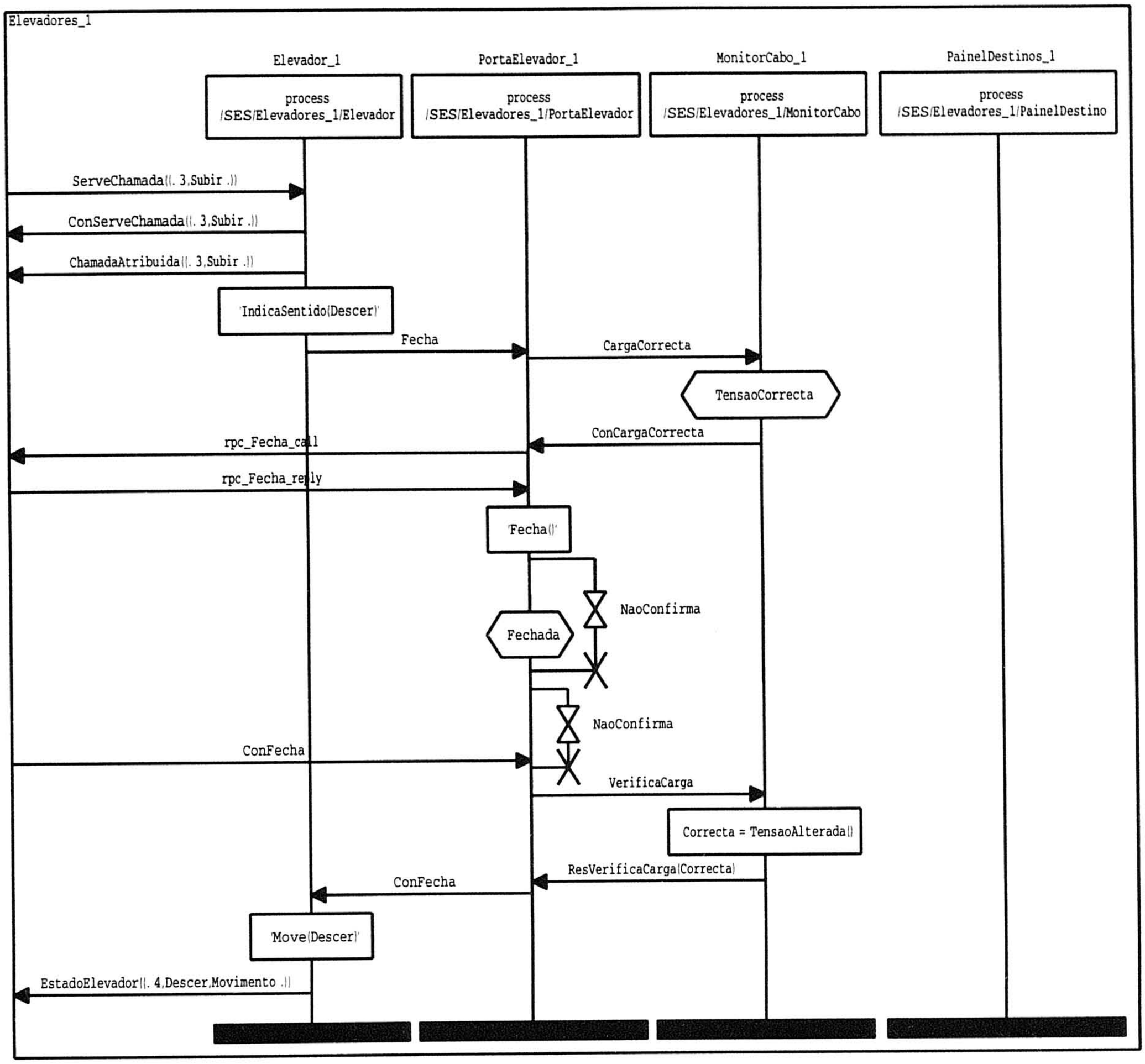

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
Premido
Subir = Chamada()
EscalaChamada({. 3,Subir .})
ConEscalaChamada({. 3,Subir .})

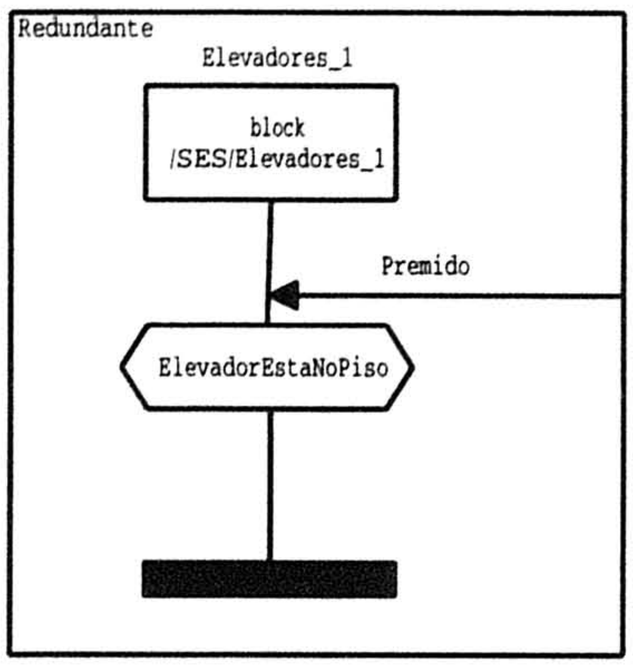

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EscalaChamada{|. 3,Subir .|}
NenhumElevadorEstaNoPiso
ConEscalaChamada{|. 3,Subir .|}
ChamadaRecebida{|. 3,Subir .|}
'Janela := MostraChamada {Janela,{|. 3,Subir .|}'
ElevadorMaisProximoOcupado

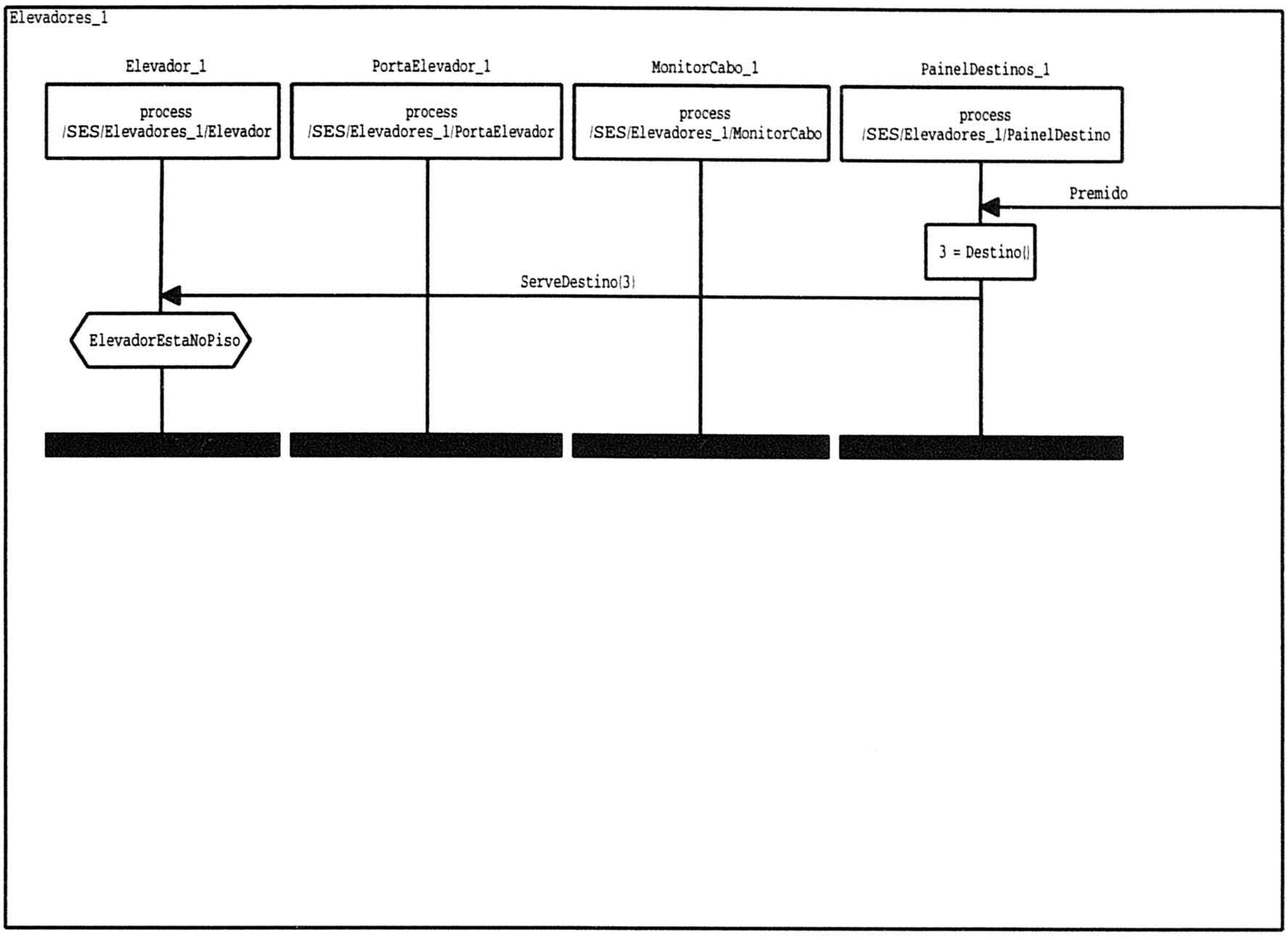

IniciaViagem
Pisos_3
Central
Elevadores_1
block /SES/Pisos_3
block /SES/Central
block /SES/Elevadores_1
Premido
Subir = Chamada()
EscalaChamada({. 3,Subir .})
NenhumElevadorEstaNoPiso
ConEscalaChamada({. 3,Subir .})
'ComandaLuz(Subir,Acesa)'
'Janela := MostraChamada (Janela,{. 3,Subir .})'
ElevadorMaisProximoLivre
ServeChamada({. 3,Subir .})
ConServeChamada({. 3,Subir .})
ChamadaAtribuida({. 3,Subir .})
'Janela := MostraChamadaElevador (Janela,1,{. 3,Subir .})'
'IndicaSentido(Descer)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
'Move(Descer)'
EstadoElevador({. 4,Descer,Movimento .})
'Janela := ActualizaEstado (Janela,1,{. 4,Descer,Movimento .})'

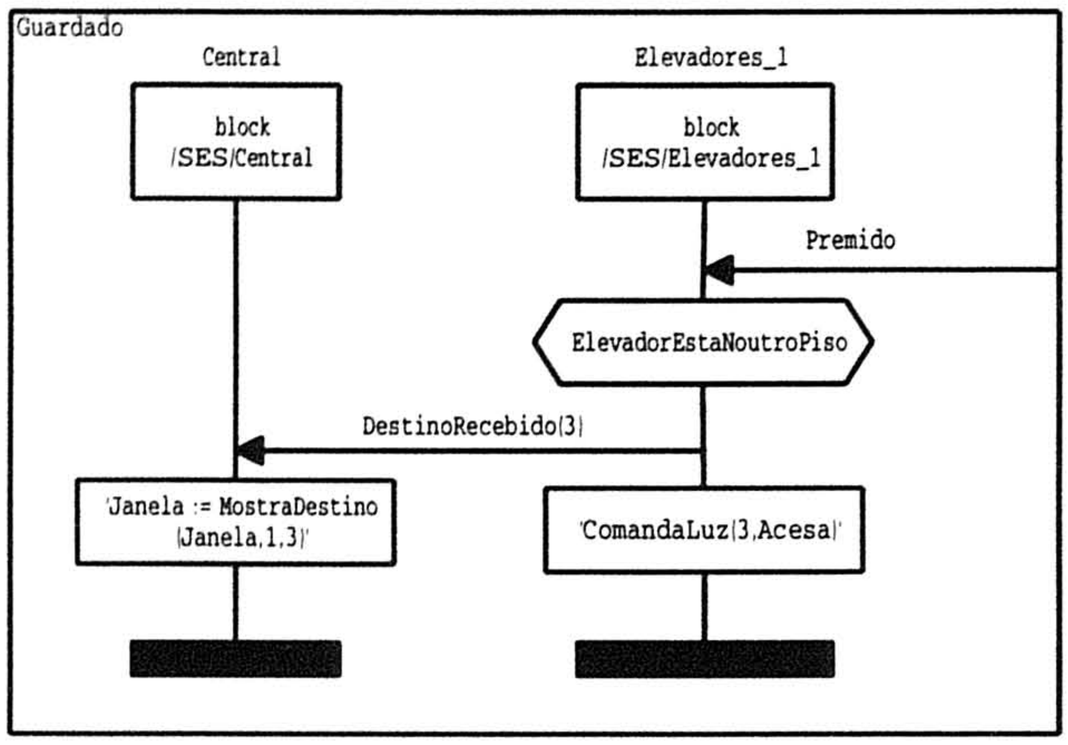

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
Premido
Subir = Chamada()
EscalaChamada{(. 3,Subir .)}
ConEscalaChamada{(. 3,Subir .)}
'ComandaLuz(Subir,Acesa)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha

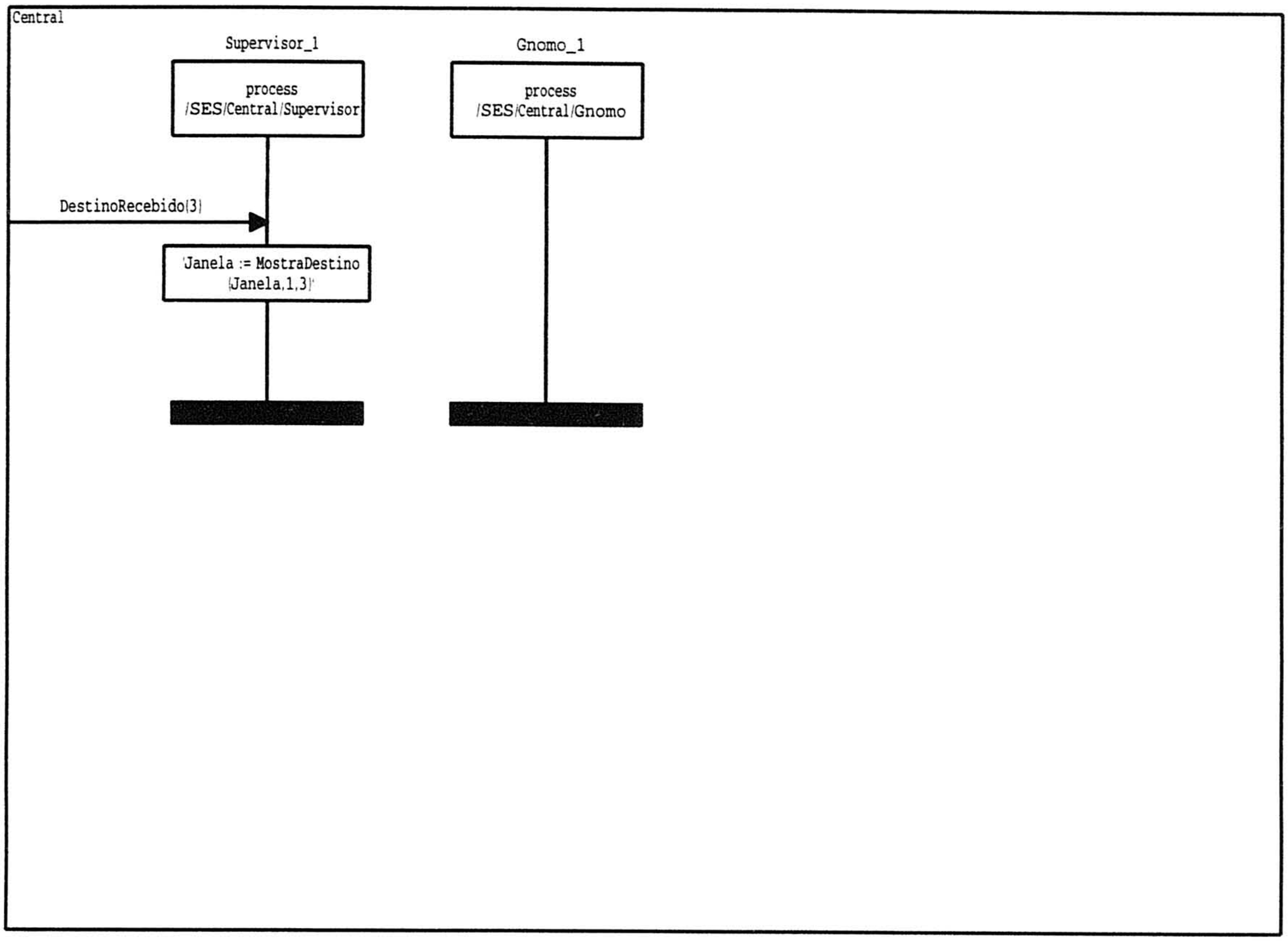

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EscalaChamada{|. 3,Subir .|}
NinguemNoPiso
ConEscalaChamada{|. 3,Subir .|}
ChamadaRecebida{|. 3,Subir .|}
Janela := MostraChamada {Janela, |. 3,Subir .|}
ElevadorMaisProximoLivre
ServeChamada{|. 3,Subir .|}
ConServeChamada{|. 3,Subir .|}
ChamadaAtribuida{|. 3,Subir .|}
Janela := MostraChamadaElevador {Janela,1,|. 3,Subir .|}
EstadoElevador{|. 4,Descer,Movimento .|}
EstadoElevador{|. 4,Descer,Movimento .|}
Janela := ActualizaEstado {Janela,1,|. 4,Descer,Movimento .|}

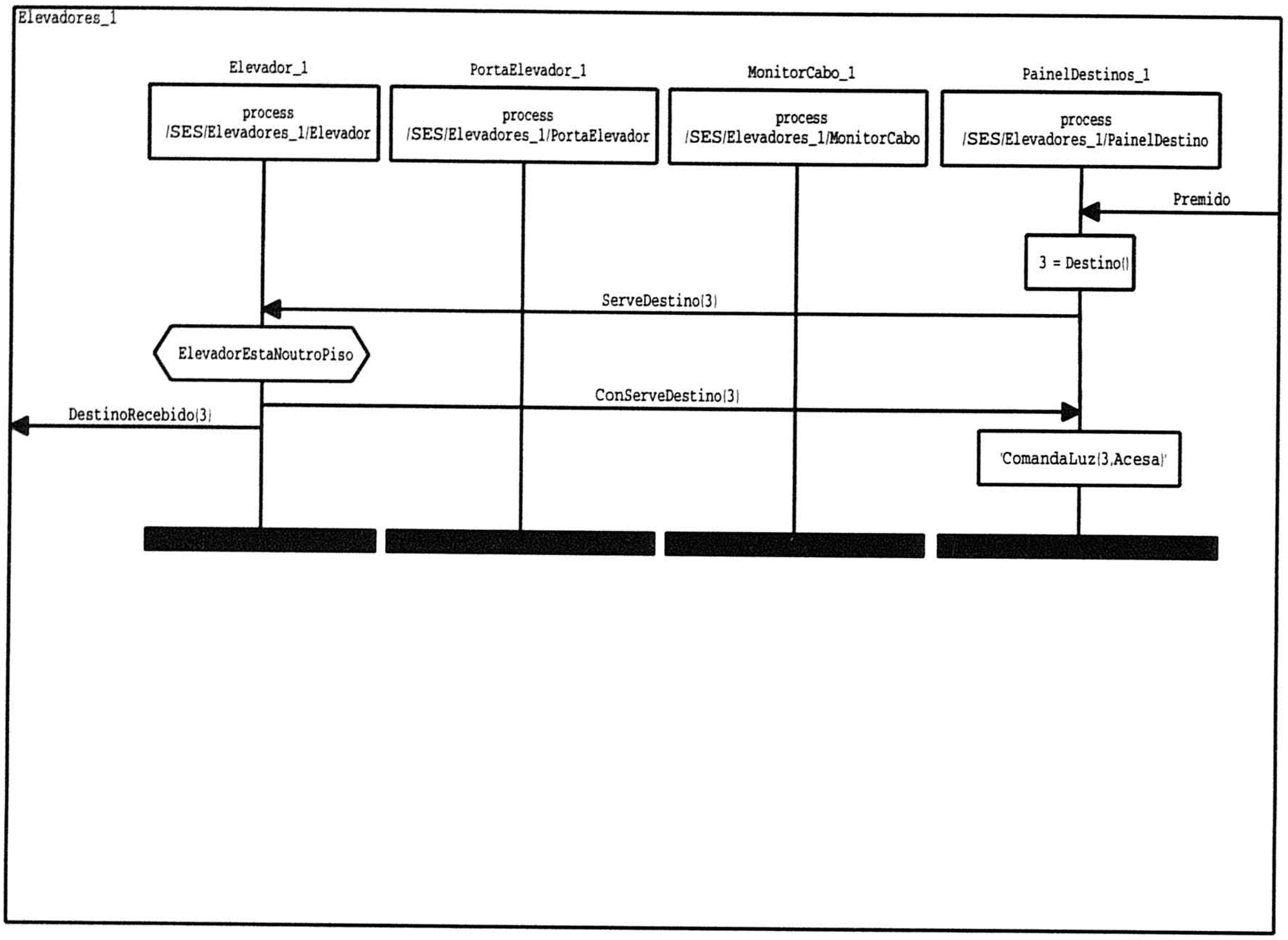

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
ServeChamada{|. 3,Subir .|}
ConServeChamada{|. 3,Subir .|}
ChamadaAtribuida{|. 3,Subir .|}
'IndicaSentido(Descer)'
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
'Move(Descer)'
EstadoElevador{|. 4,Descer,Movimento .|}

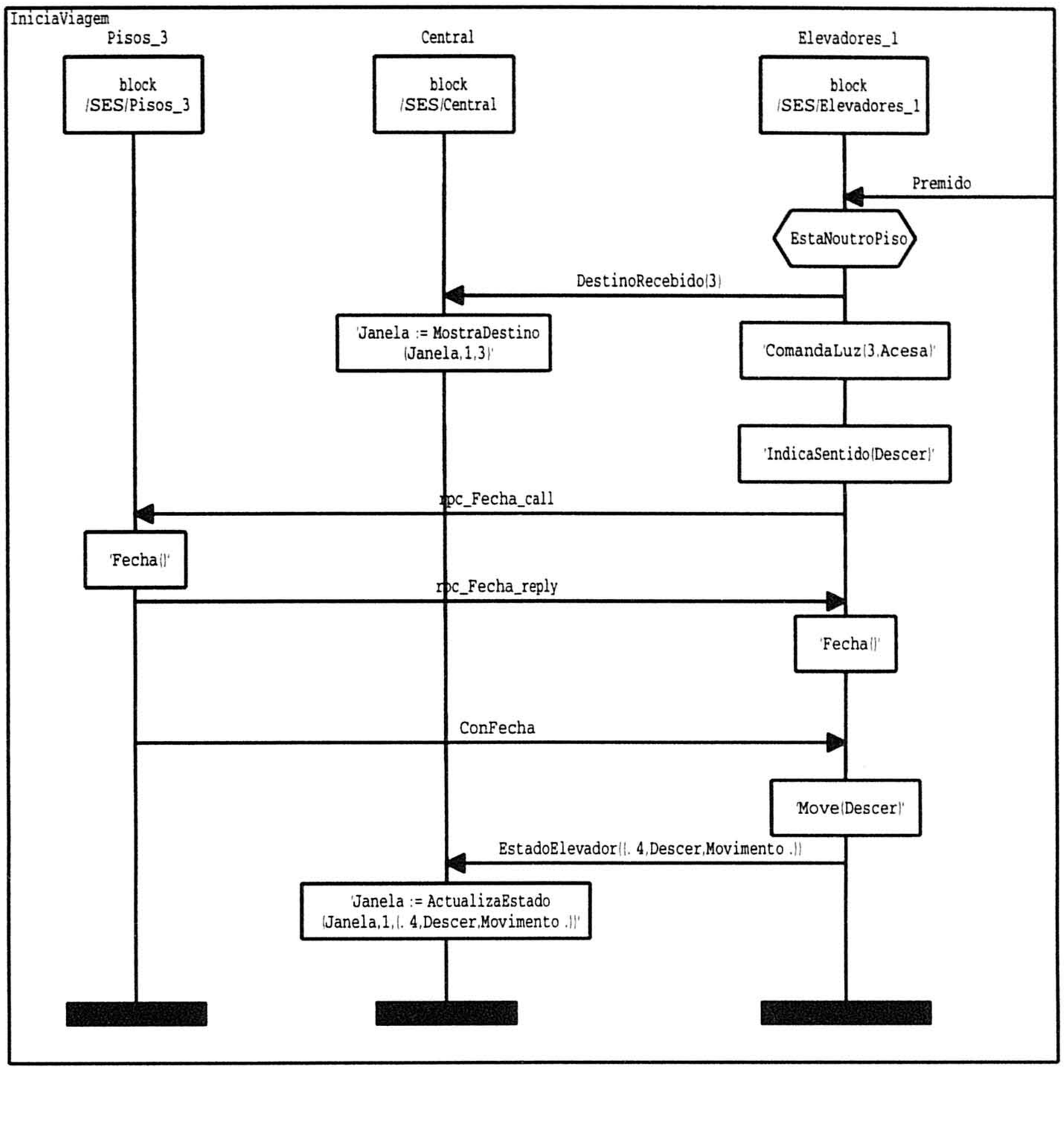

Redundante
Elevadores_1
block
/SES/Elevadores_1
Premido
ElevadorEstaNoPiso

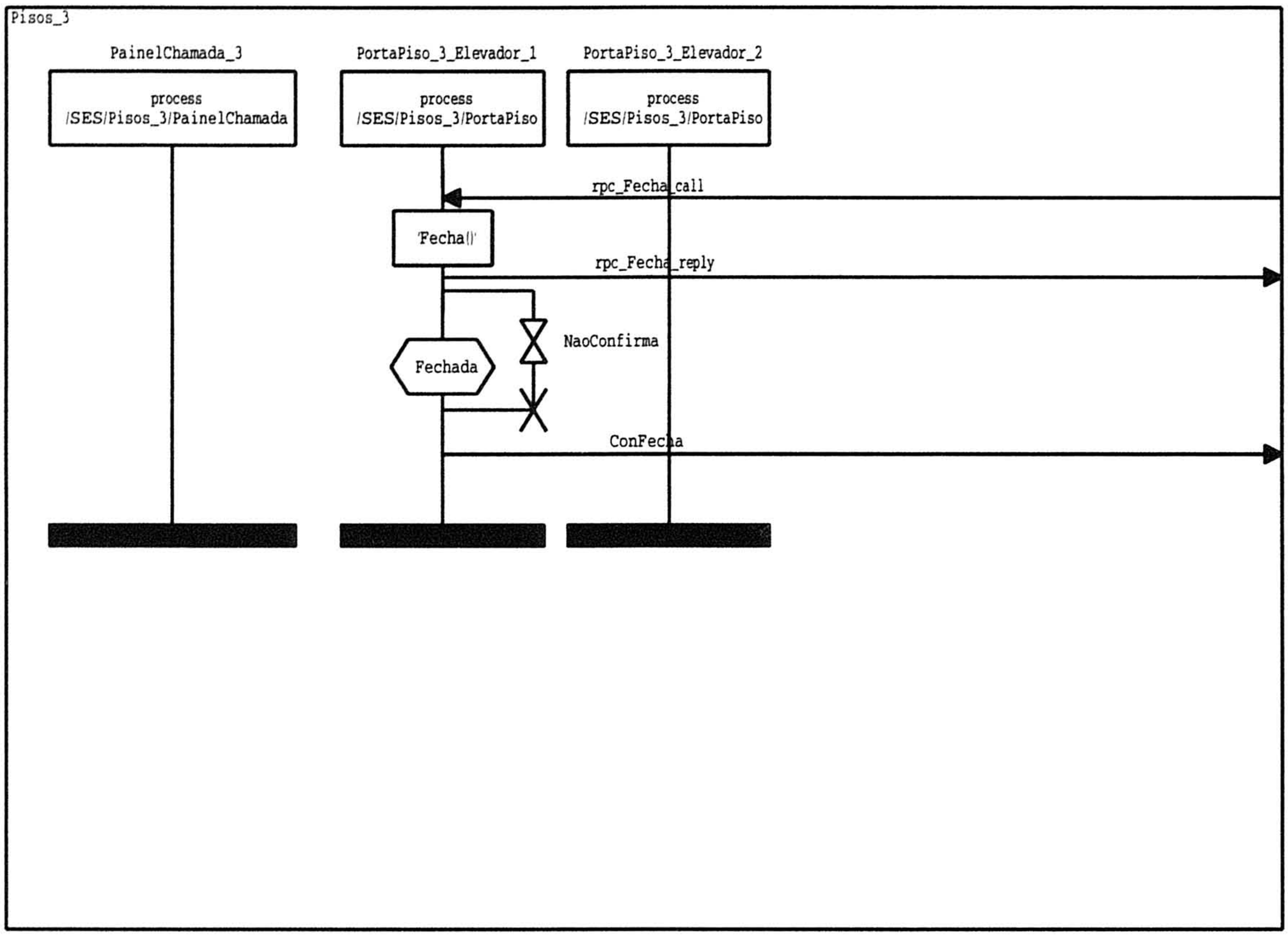

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
Premido
3 = Destino()
ServeDestino(3)
ElevadorEstaNoPiso

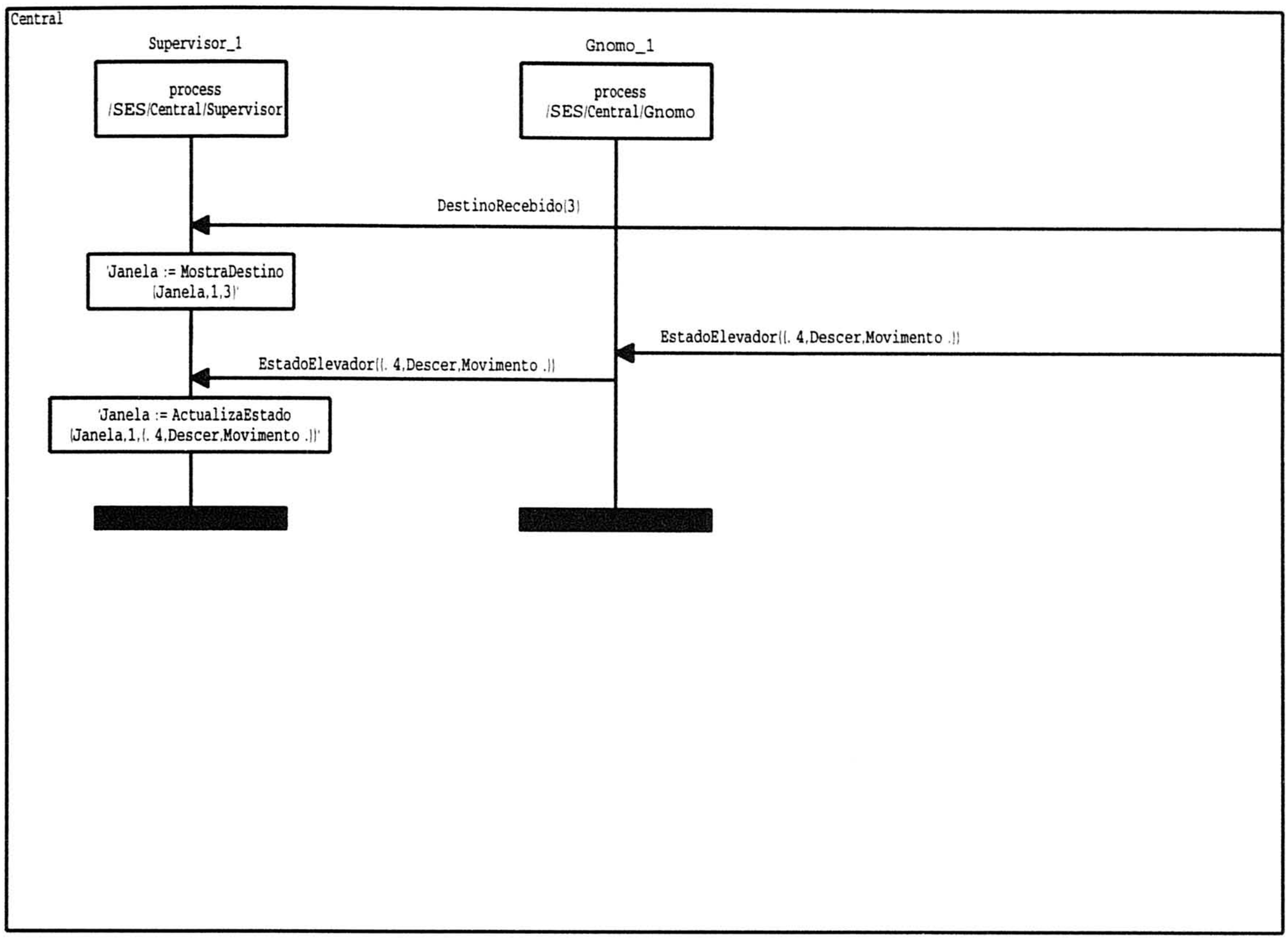

Guardado
Central
Elevadores_1
block
/SES/Central
block
/SES/Elevadores_1
Premido
ElevadorEstaNoutroPiso
DestinoRecebido(3)
'Janela := MostraDestino
(Janela,1,3)'
'ComandaLuz(3,Acesa)'

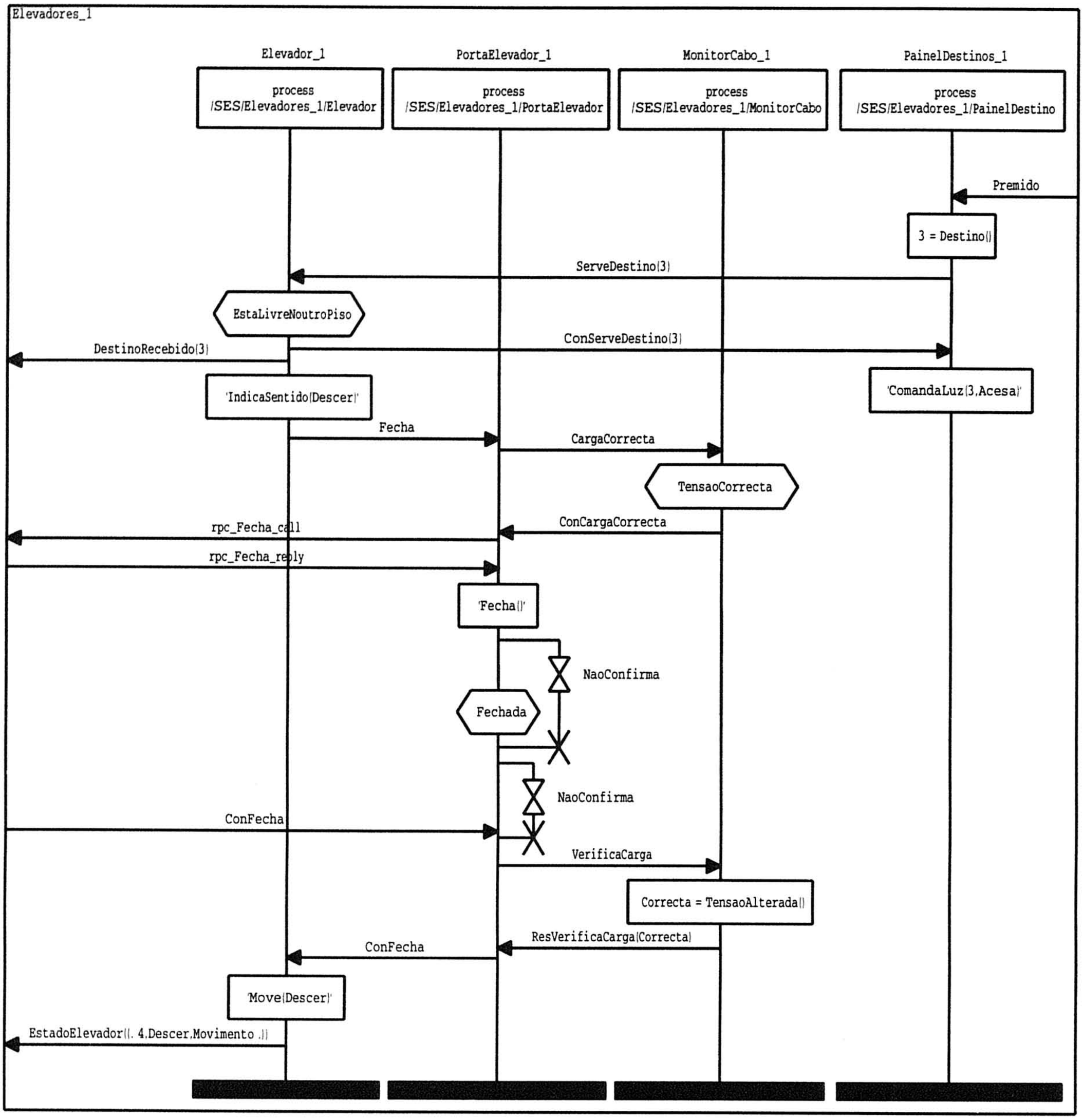

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
DestinoRecebido(3)
'Janela := MostraDestino
(Janela,1,3)'

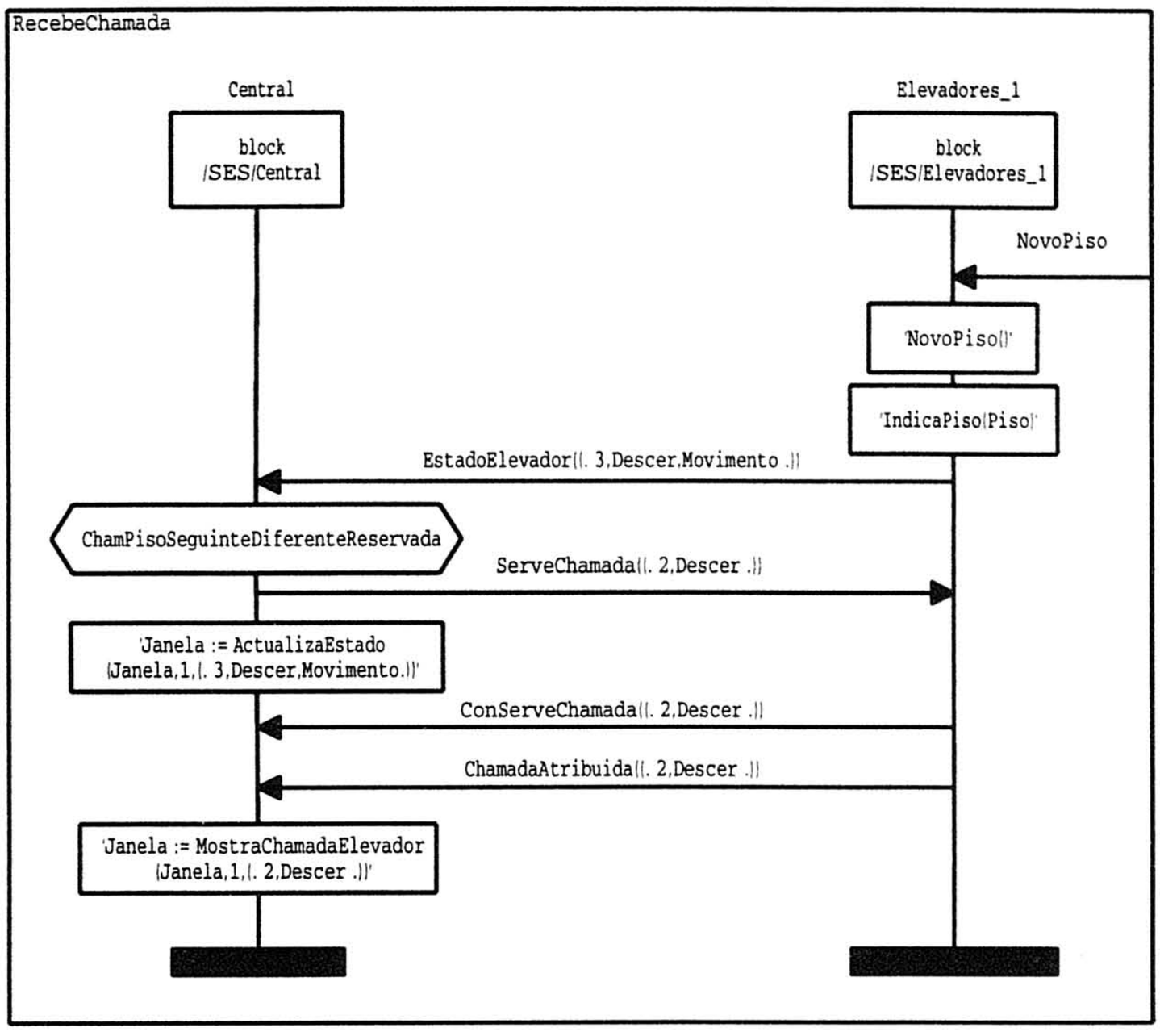

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
Premido
3 = Destino()
ServeDestino(3)
ElevadorEstaNoutroPiso
ConServeDestino(3)
DestinoRecebido(3)
'ComandaLuz(3,Acesa)'

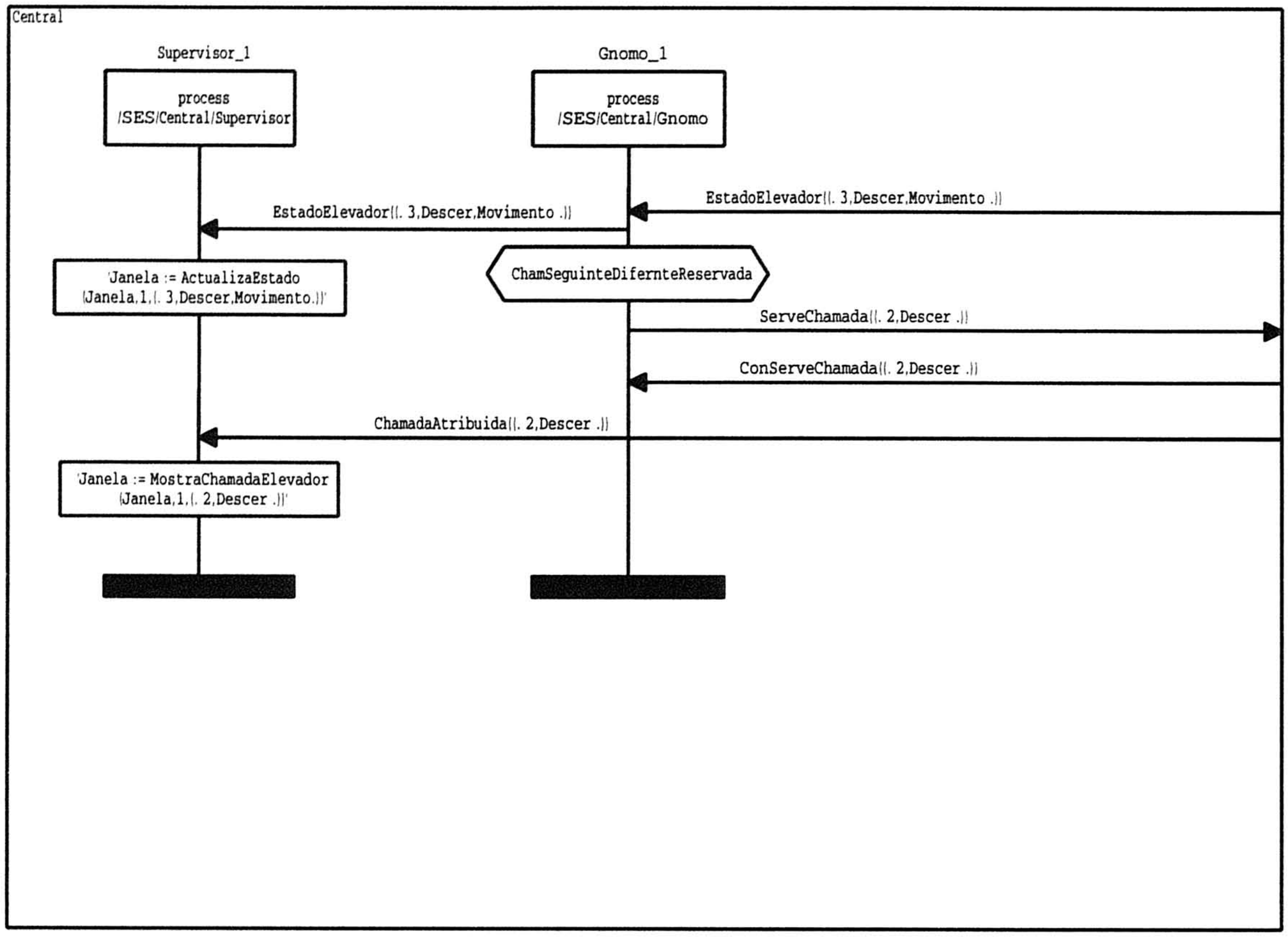

IniciaViagem
Pisos_3
Central
Elevadores_1
block /SES/Pisos_3
block /SES/Central
block /SES/Elevadores_1
Premido
EstaNoutroPiso
DestinoRecebido(3)
'Janela := MostraDestino (Janela,1,3)'
'ComandaLuz(3,Acesa)'
'IndicaSentido(Descer)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
'Move(Descer)'
EstadoElevador(|. 4,Descer,Movimento .|)
'Janela := ActualizaEstado (Janela,1,|. 4,Descer,Movimento .|)'

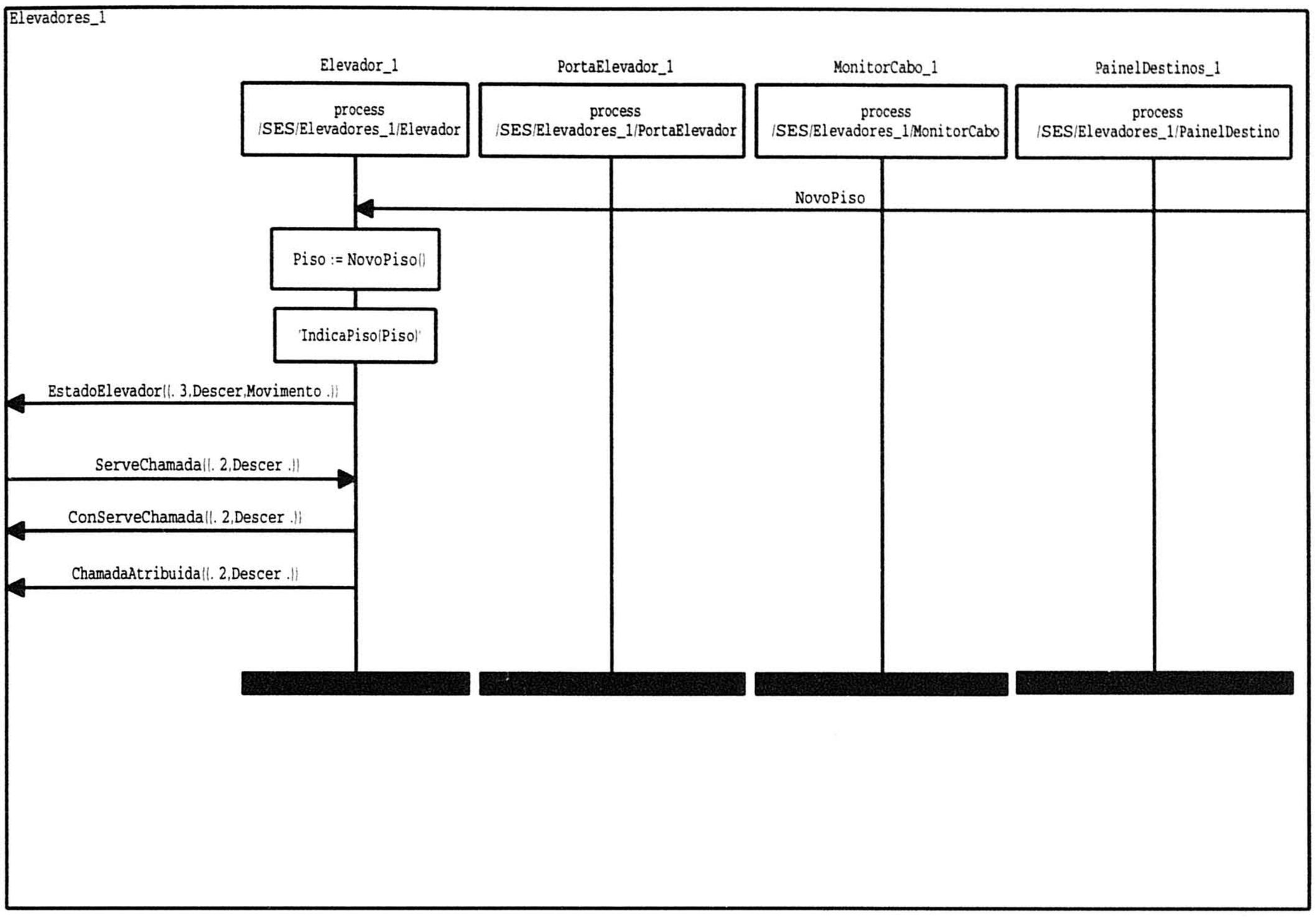

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha

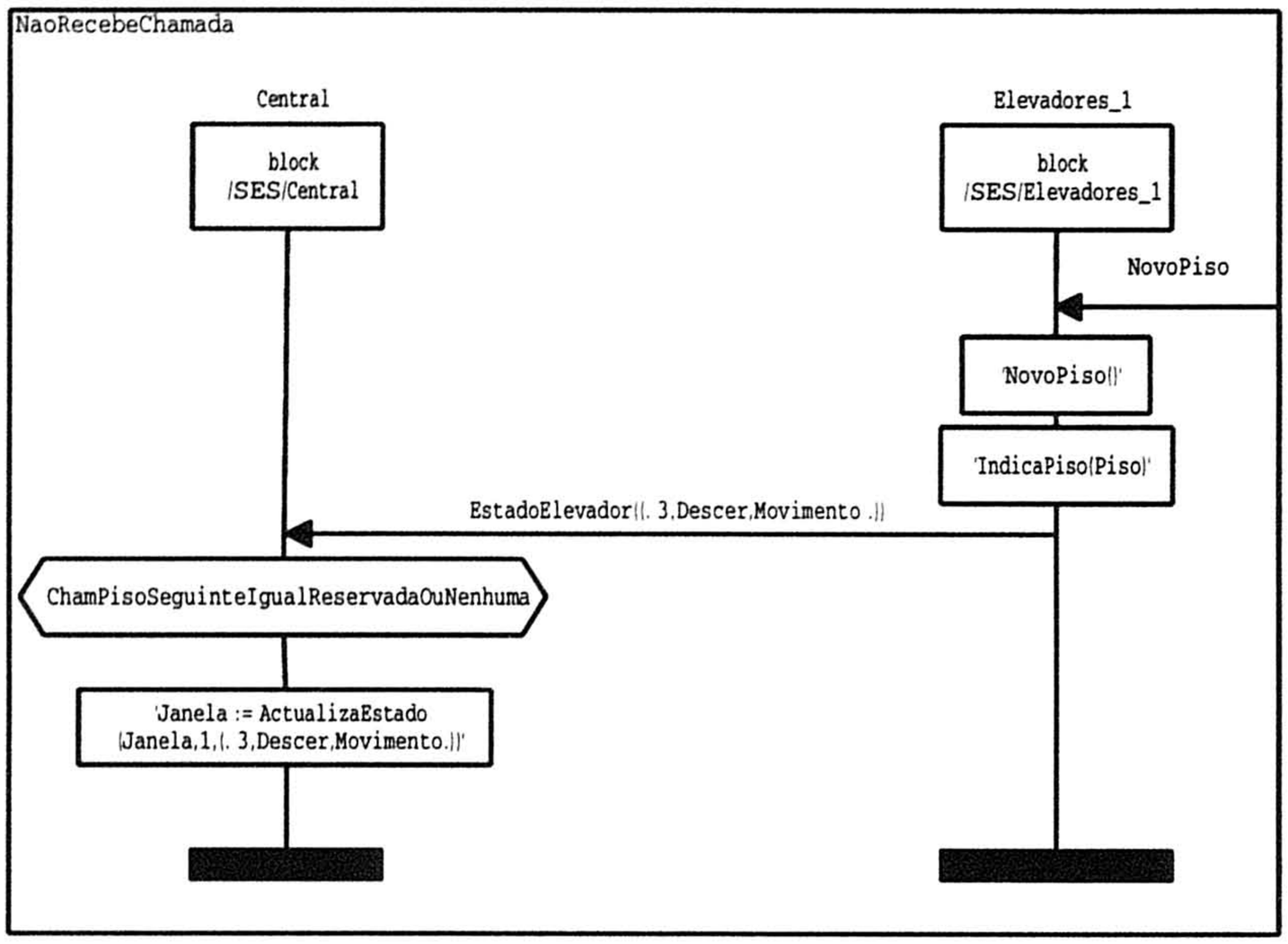

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
DestinoRecebido(3)
'Janela := MostraDestino
(Janela,1,3)'
EstadoElevador({. 4,Descer,Movimento .})
EstadoElevador({. 4,Descer,Movimento .})
'Janela := ActualizaEstado
(Janela,1,{. 4,Descer,Movimento .})'

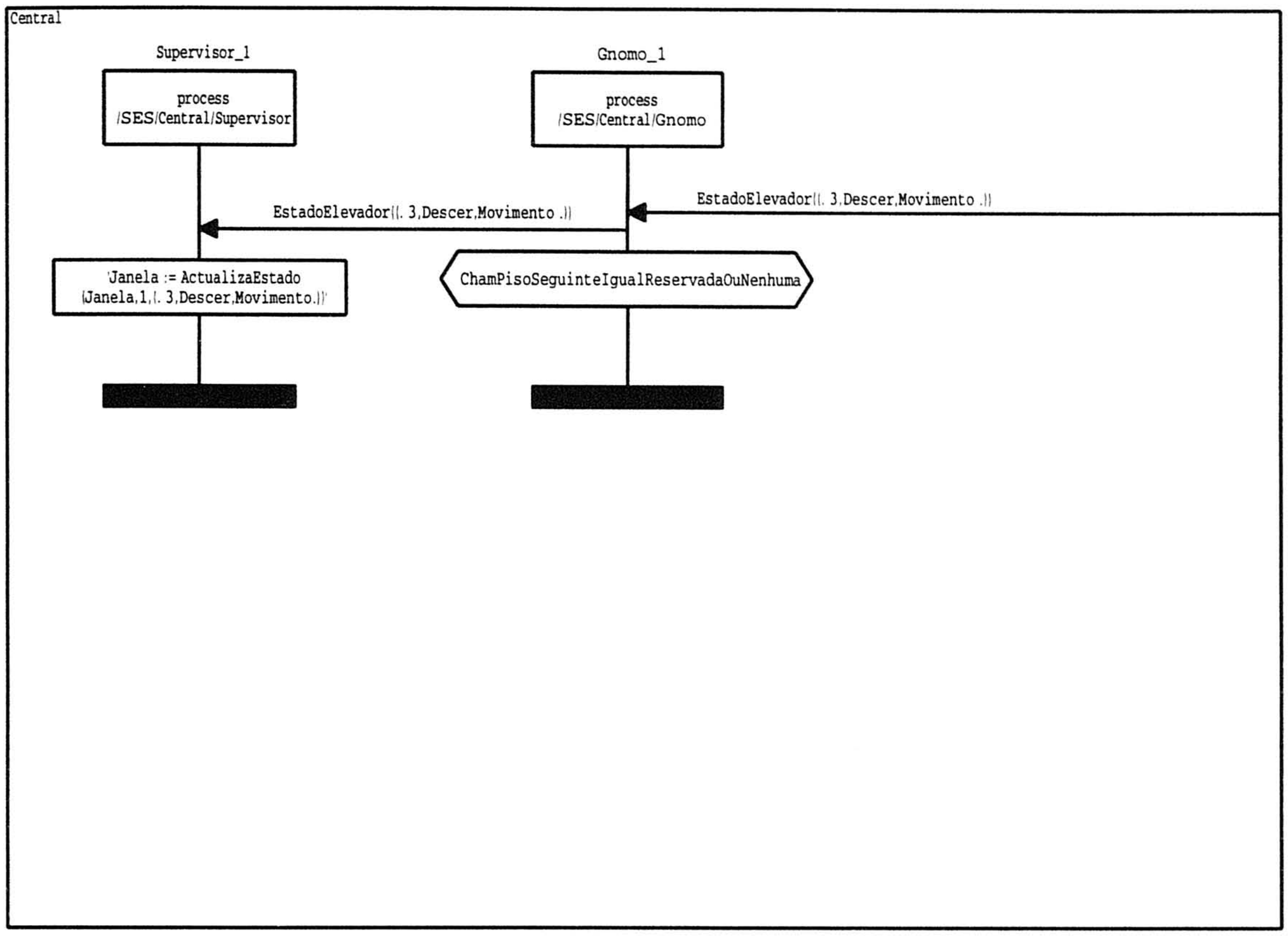

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
Premido
3 = Destino()
ServeDestino(3)
EstaLivreNoutroPiso
ConServeDestino(3)
DestinoRecebido(3)
'IndicaSentido(Descer)'
'ComandaLuz(3,Acesa)'
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
'Move(Descer)'
EstadoElevador(1, 4,Descer,Movimento ,1)

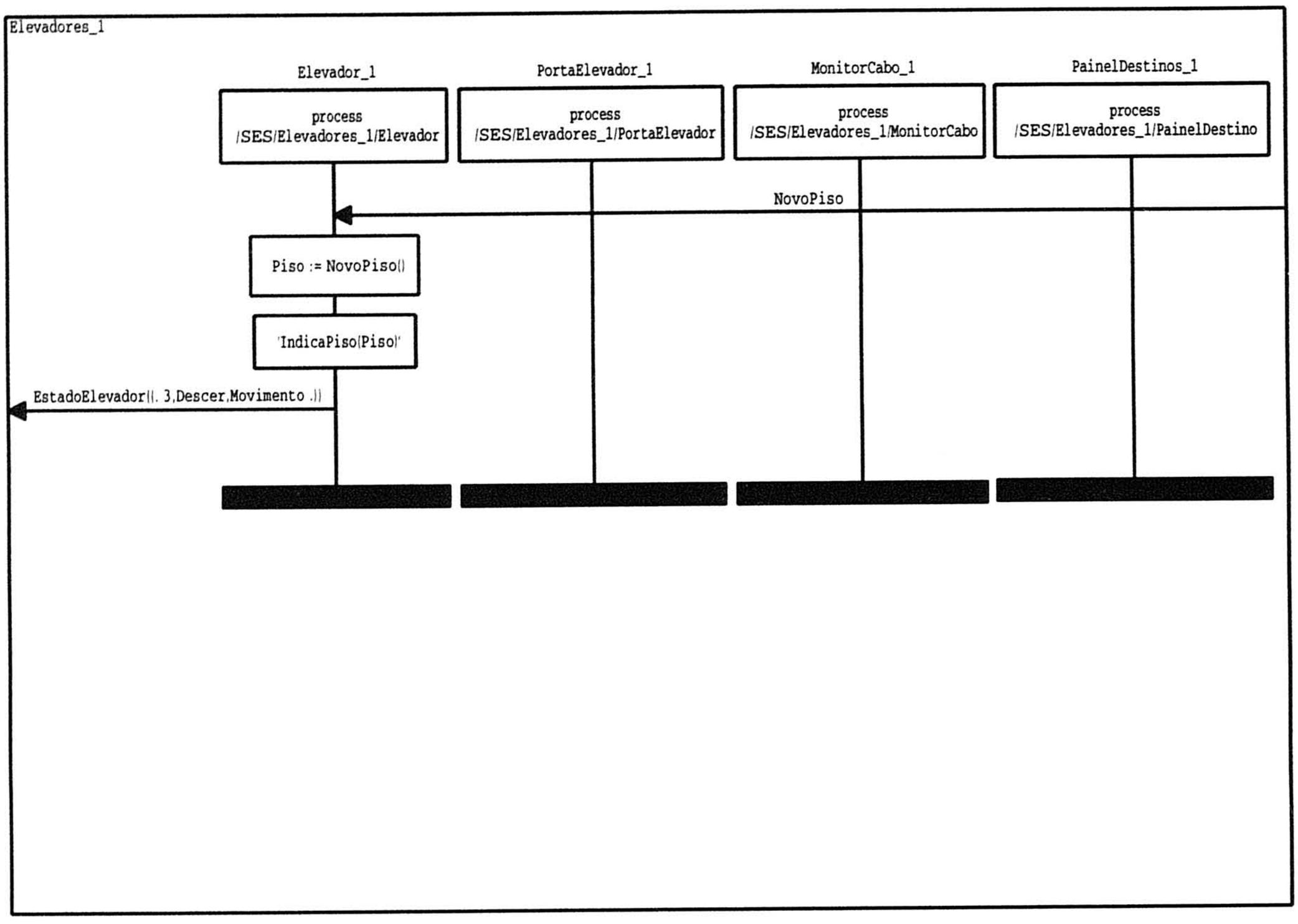

RecebeChamada
Central
block
/SES/Central
Elevadores_1
block
/SES/Elevadores_1
NovoPiso
'NovoPiso()'
'IndicaPiso(Piso)'
EstadoElevador(|. 3,Descer,Movimento .|)
ChamPisoSeguinteDiferenteReservada
ServeChamada(|. 2,Descer .|)
'Janela := ActualizaEstado
(Janela,1,|. 3,Descer,Movimento.|)'
ConServeChamada(|. 2,Descer .|)
ChamadaAtribuida(|. 2,Descer .|)
'Janela := MostraChamadaElevador
(Janela,1,|. 2,Descer .|)'

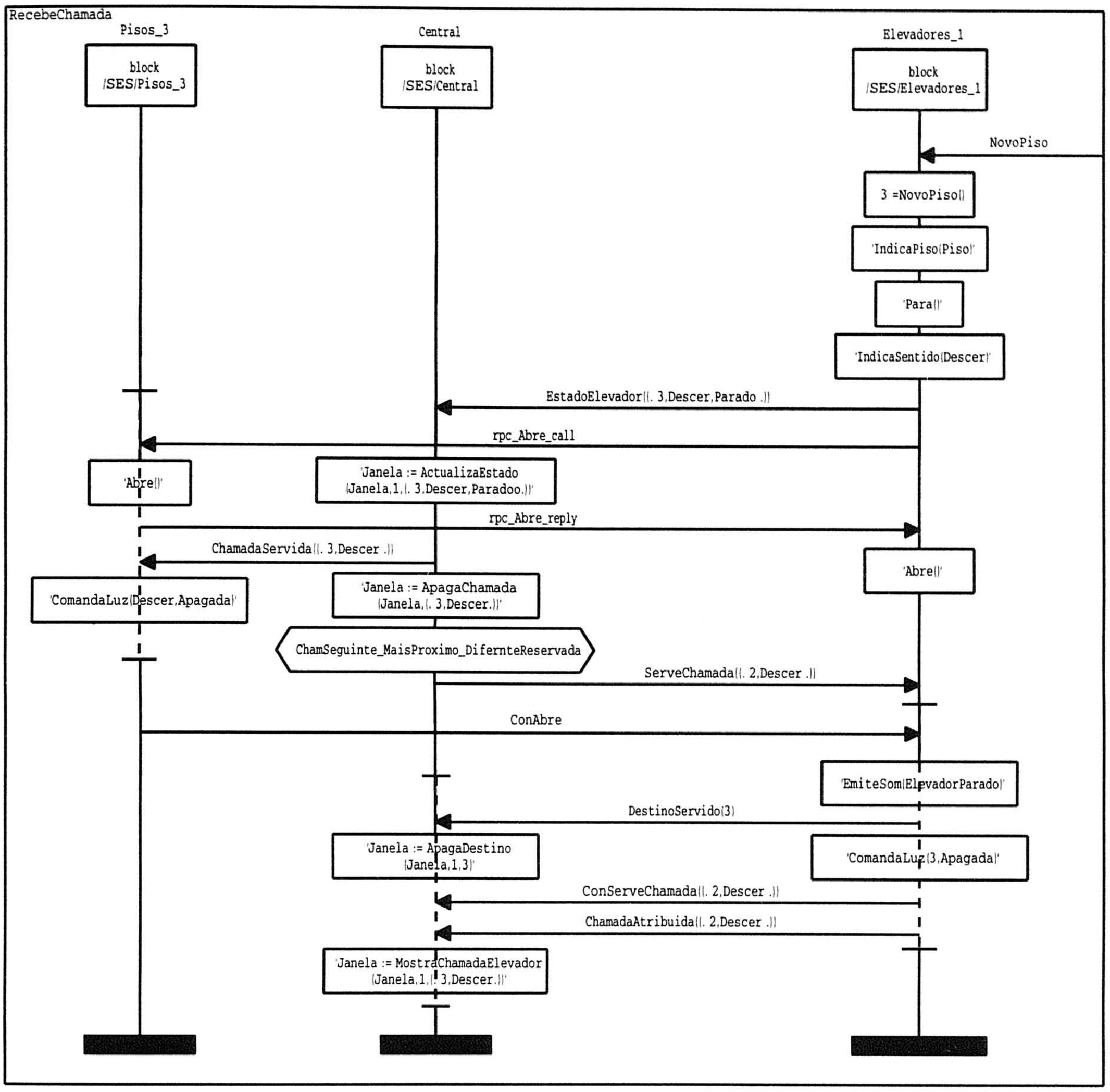

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Movimento .|}
EstadoElevador{|. 3,Descer,Movimento .|}
'Janela := ActualizaEstado (Janela,1,|. 3,Descer,Movimento.|)'
ChamSeguinteDifernteReservada
ServeChamada{|. 2,Descer .|}
ConServeChamada{|. 2,Descer .|}
ChamadaAtribuida{|. 2,Descer .|}
'Janela := MostraChamadaElevador (Janela,1,|. 2,Descer .|)'

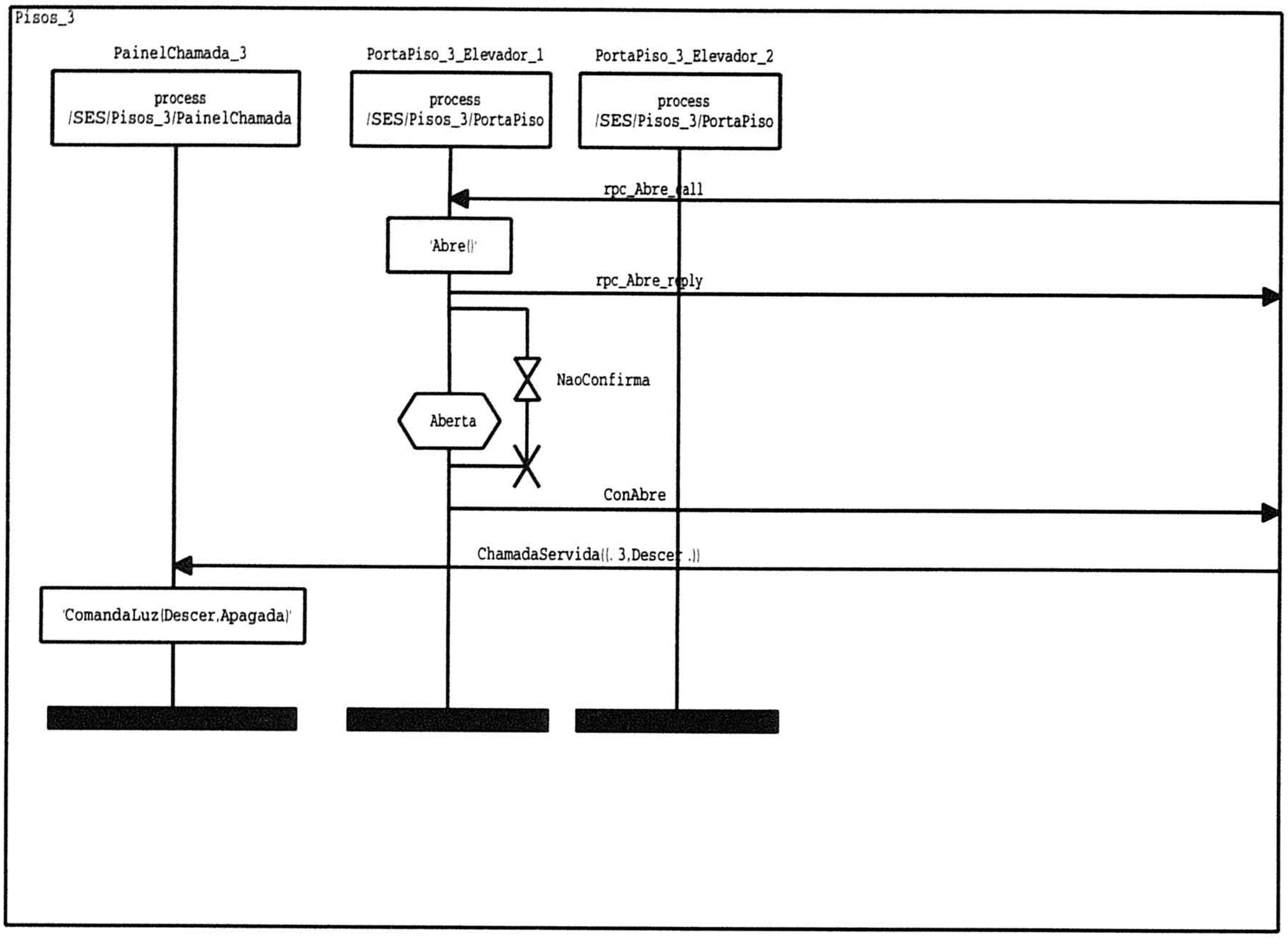

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
NovoPiso
Piso := NovoPiso()
'IndicaPiso(Piso)'
EstadoElevador({. 3,Descer,Movimento .})
ServeChamada({. 2,Descer .})
ConServeChamada({. 2,Descer .})
ChamadaAtribuida({. 2,Descer .})

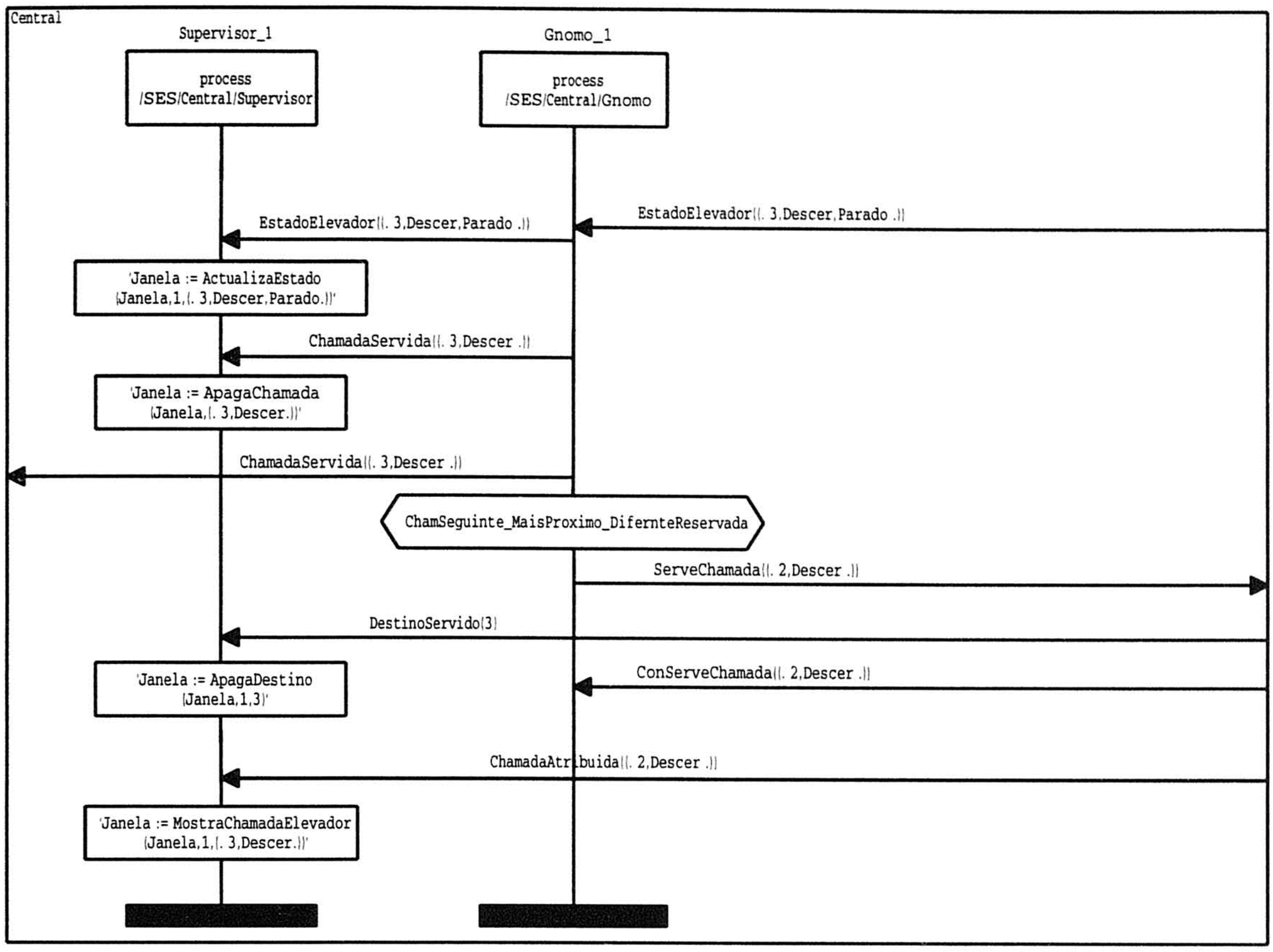

NaoRecebeChamada
Central
block
/SES/Central
Elevadores_1
block
/SES/Elevadores_1
NovoPiso
'NovoPiso{}'
'IndicaPiso(Piso)'
EstadoElevador{(. 3,Descer,Movimento .)}
ChamPisoSeguinteIgualReservadaOuNenhuma
'Janela := ActualizaEstado
(Janela,1,(. 3,Descer,Movimento.))'

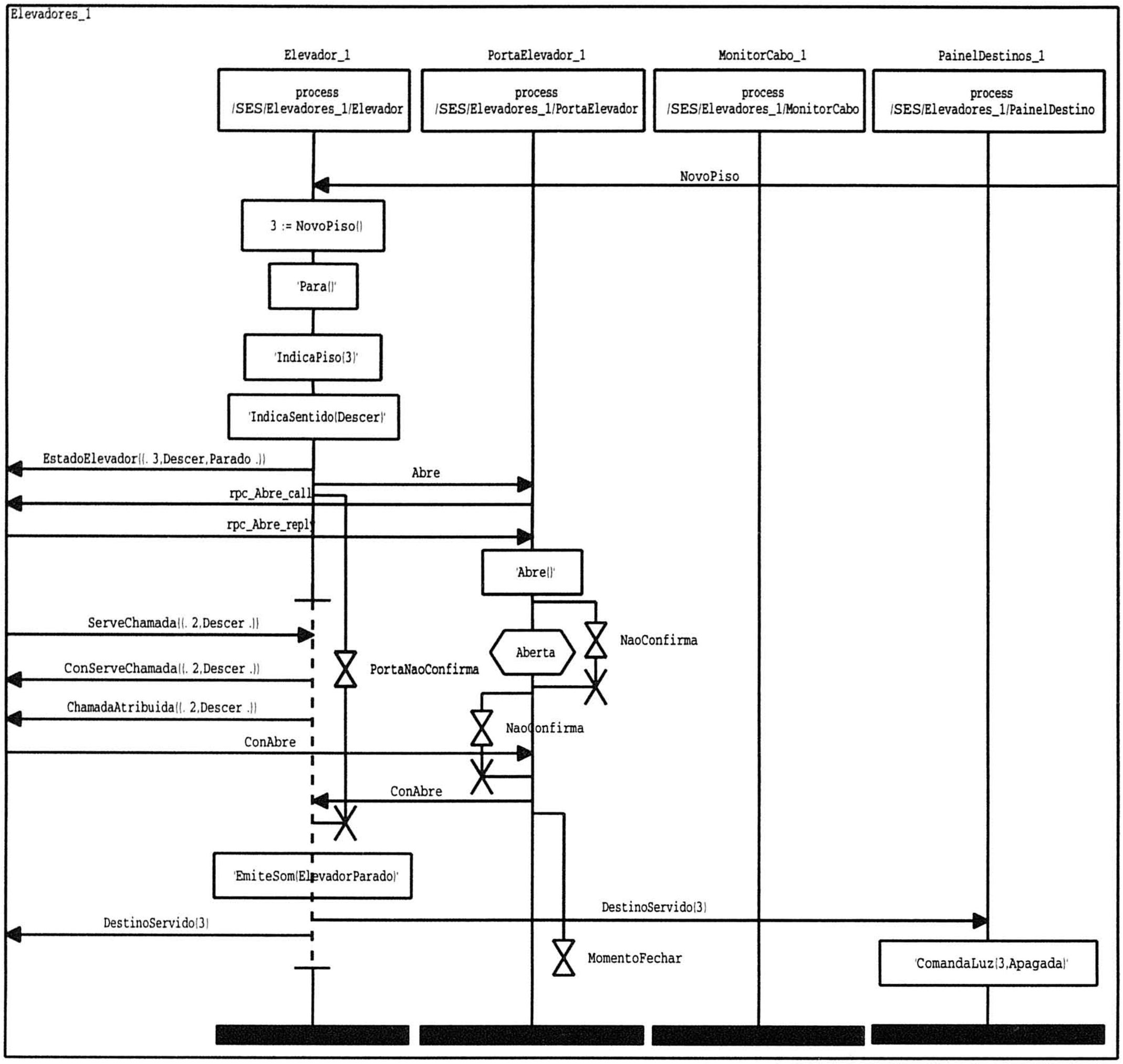

Central
Supervisor_1
process
/SES/Central/Supervisor
Gnomo_1
process
/SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Movimento .|}
EstadoElevador{|. 3,Descer,Movimento .|}
'Janela := ActualizaEstado
(Janela,1,{. 3,Descer,Movimento.|}'
ChamPisoSeguinteIgualReservadaOuNenhuma

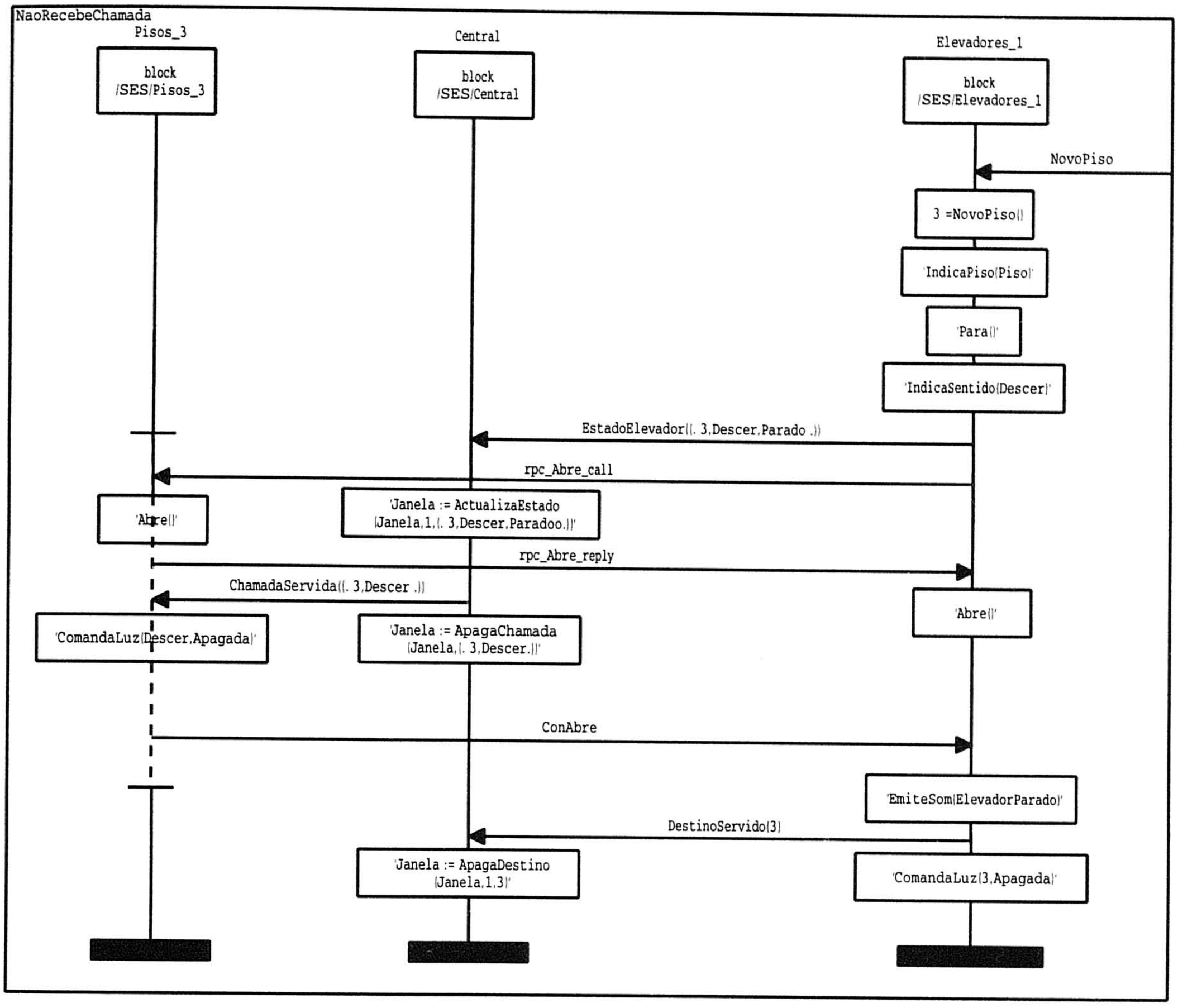

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
NovoPiso
Piso := NovoPiso()
'IndicaPiso(Piso)'
EstadoElevador(!. 3,Descer,Movimento .!)

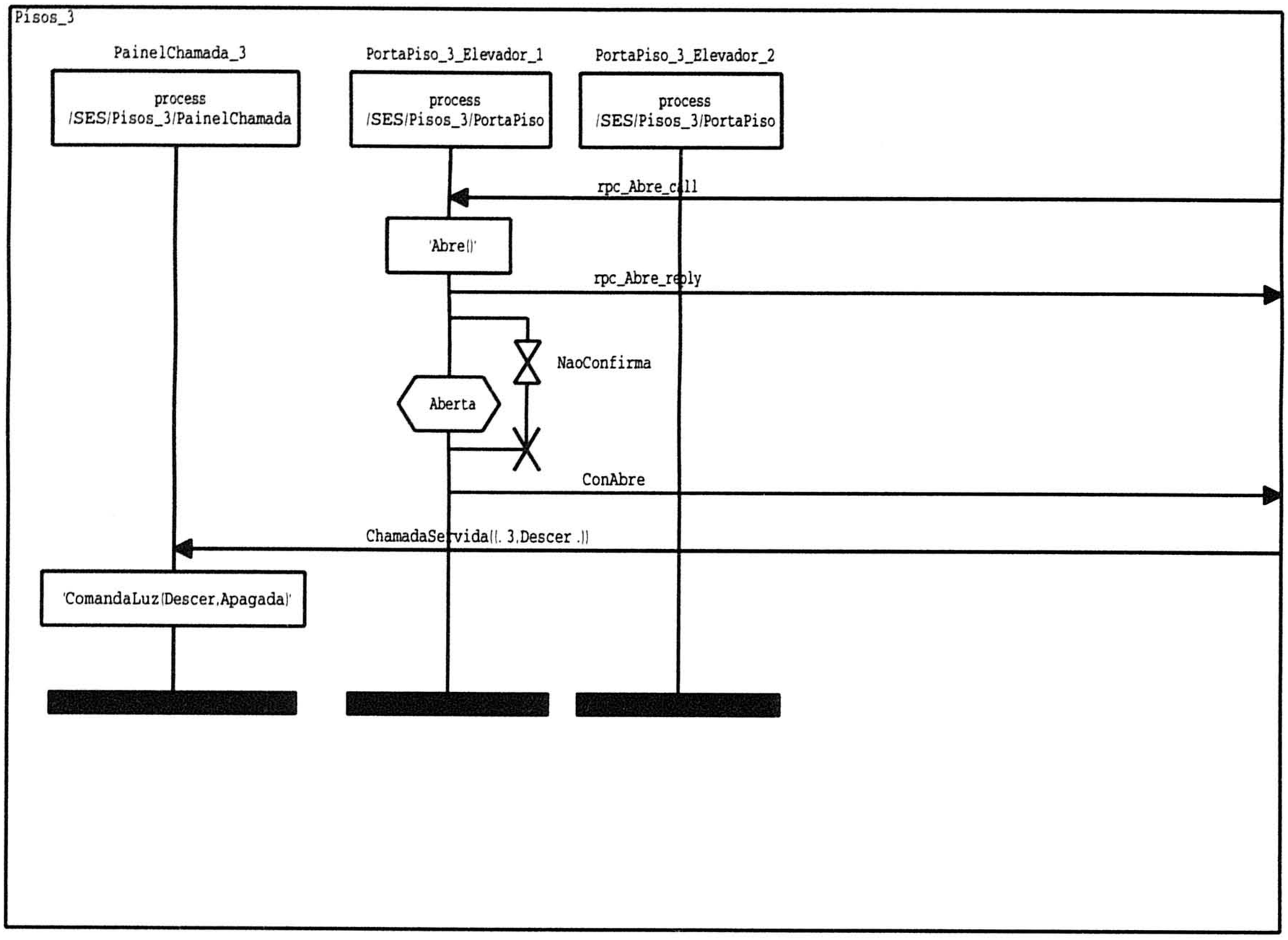

RecebeChamada
Pisos_3
block /SES/Pisos_3
Central
block /SES/Central
Elevadores_1
block /SES/Elevadores_1
NovoPiso
3 =NovoPiso()
'IndicaPiso(Piso)'
'Para()'
'IndicaSentido(Descer)'
EstadoElevador(. 3,Descer,Parado .)
rpc_Abre_call
'Abre()'
'Janela := ActualizaEstado (Janela,1,(. 3,Descer,Paradoo.))'
rpc_Abre_reply
ChamadaServida(. 3,Descer .)
'Abre()'
'ComandaLuz(Descer,Apagada)'
'Janela := ApagaChamada (Janela,(. 3,Descer.))'
ChamSeguinte_MaisProximo_DifernteReservada
ServeChamada(. 2,Descer .)
ConAbre
'EmiteSom(ElevadorParado)'
DestinoServido(3)
'Janela := ApagaDestino (Janela,1,3)'
'ComandaLuz(3,Apagada)'
ConServeChamada(. 2,Descer .)
ChamadaAtribuida(. 2,Descer .)
'Janela := MostraChamadaElevador (Janela,1,(. 3,Descer.))'

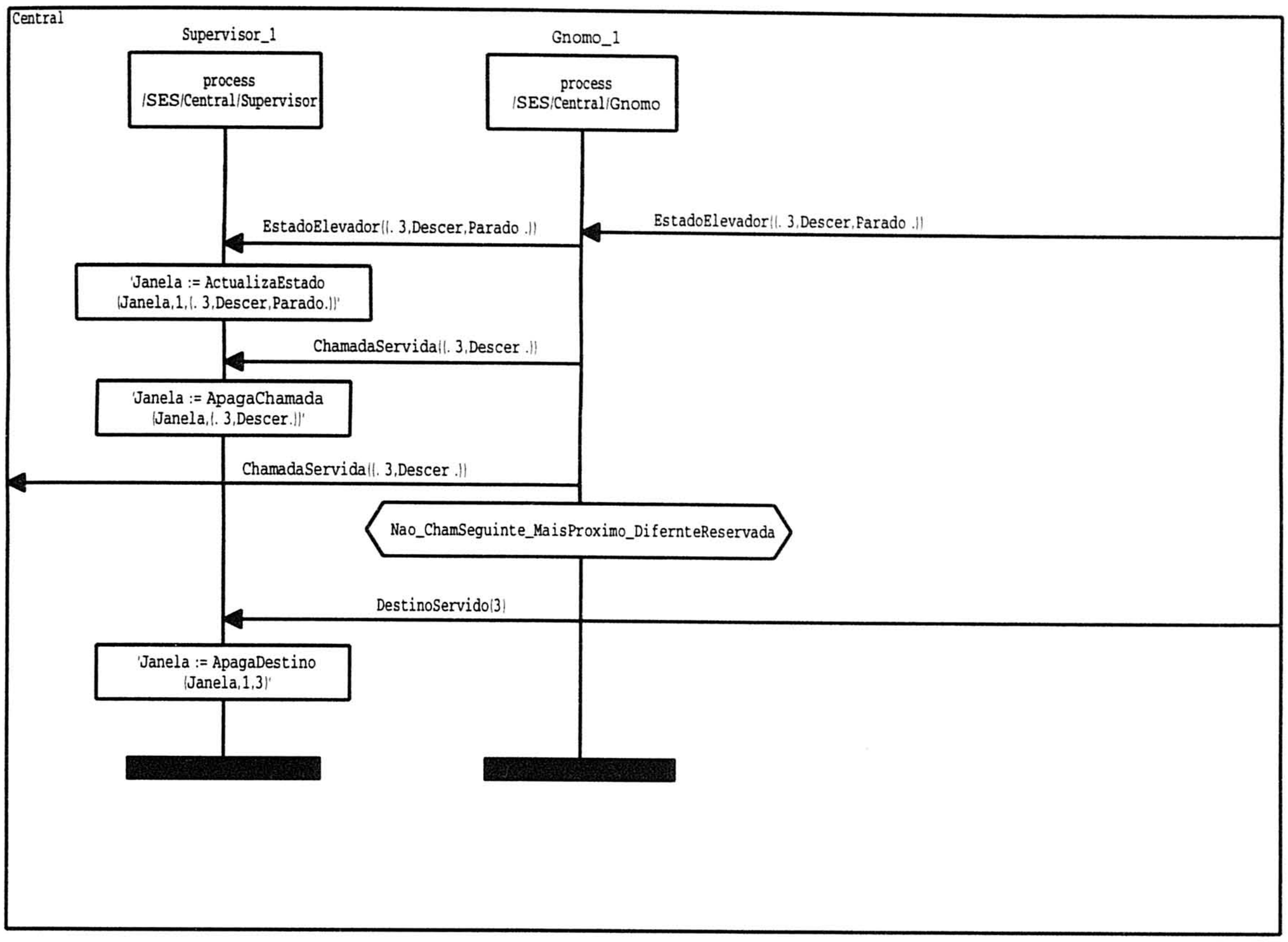

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre
ChamadaServida({. 3,Descer .})
'ComandaLuz(Descer,Apagada)'

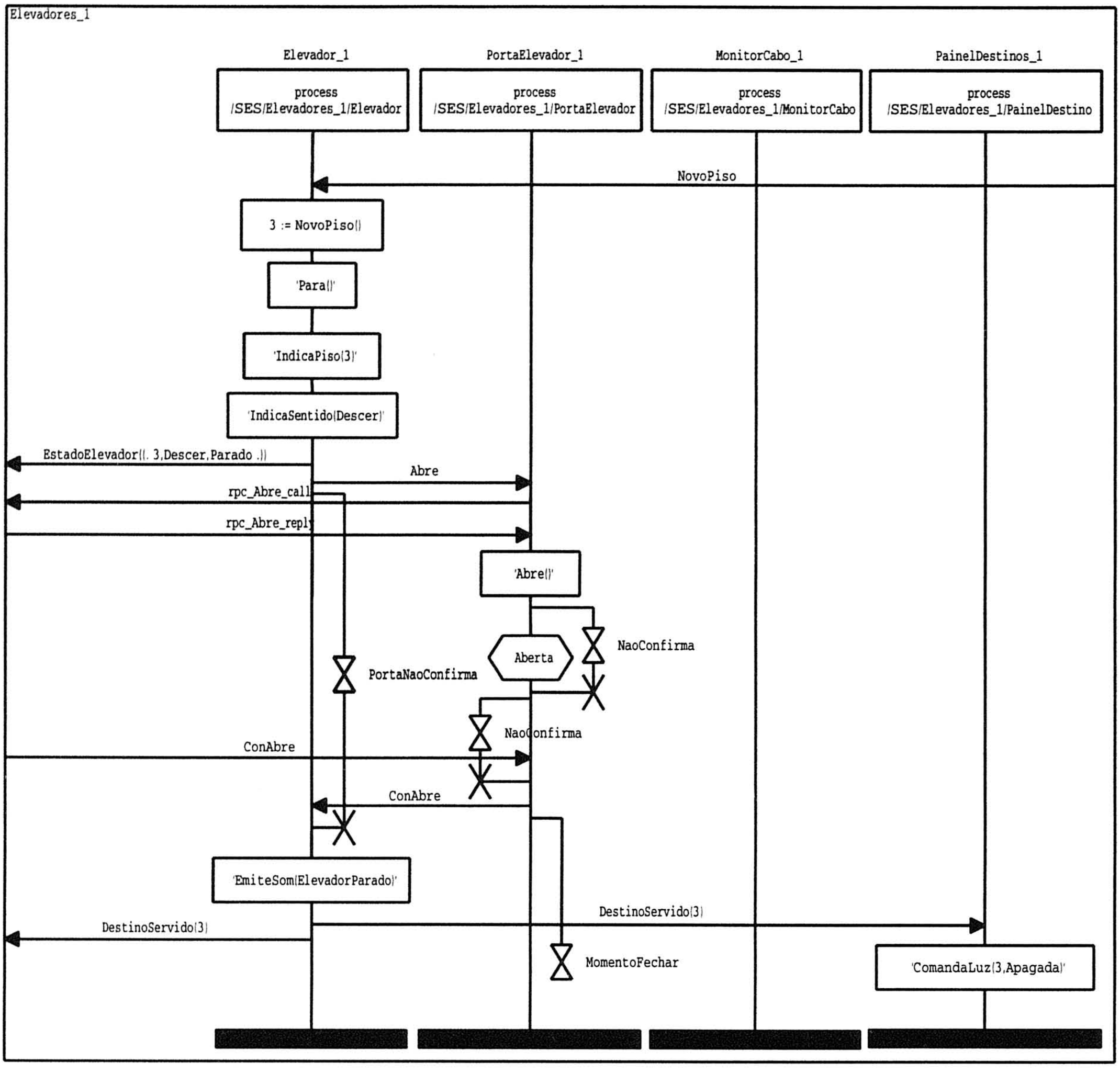

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Parado .|}
EstadoElevador{|. 3,Descer,Parado .|}
'Janela := ActualizaEstado (Janela,1,{. 3,Descer,Parado.|}'
ChamadaServida{|. 3,Descer .|}
'Janela := ApagaChamada (Janela,{. 3,Descer.|}'
ChamadaServida{|. 3,Descer .|}
ChamSeguinte_MaisProximo_DiferenteReservada
ServeChamada{|. 2,Descer .|}
DestinoServido{3|
'Janela := ApagaDestino (Janela,1,3)'
ConServeChamada{|. 2,Descer .|}
ChamadaAtribuida{|. 2,Descer .|}
'Janela := MostraChamadaElevador (Janela,1,{. 3,Descer.|}'

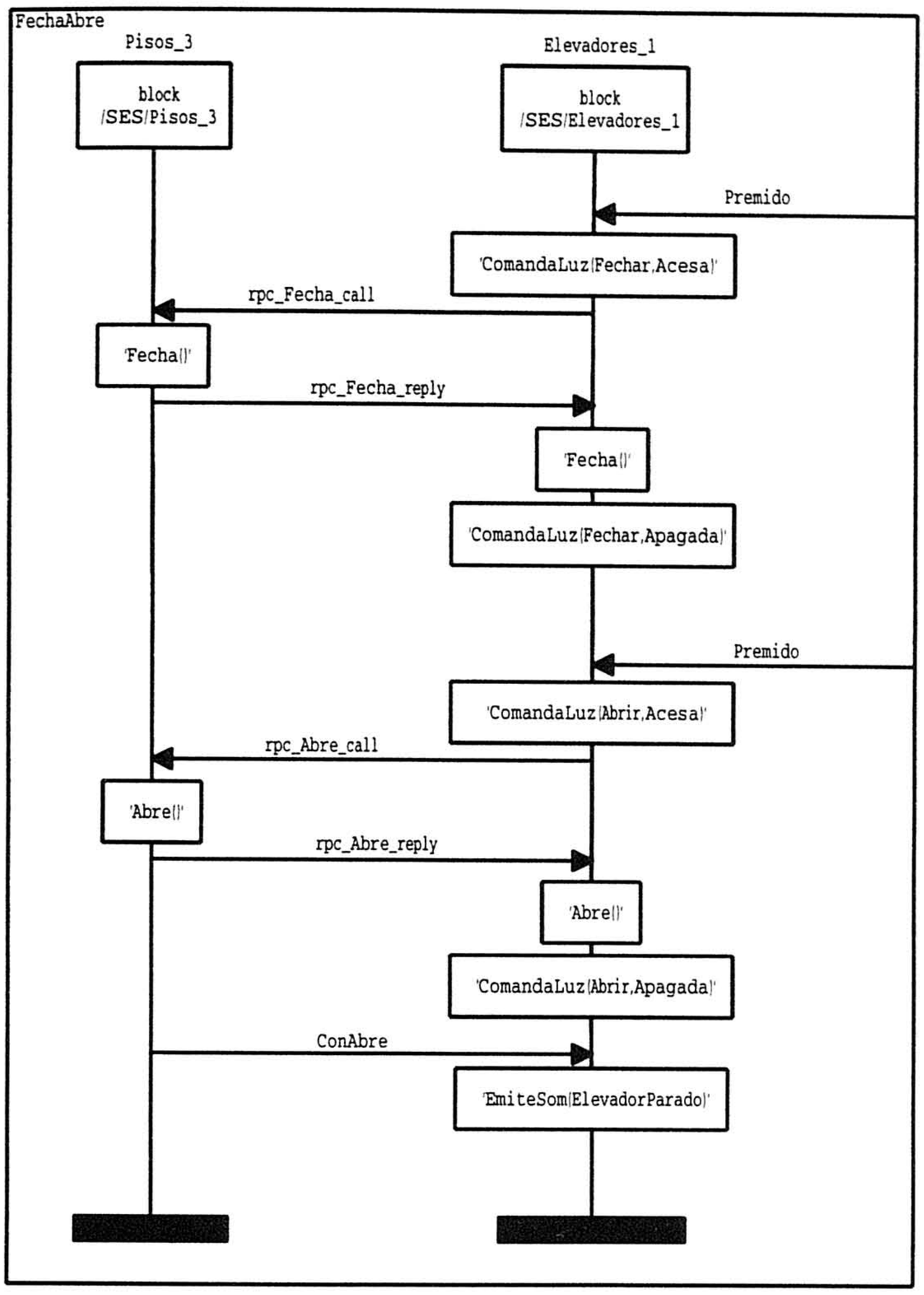

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
NovoPiso
3 := NovoPiso()
'Para()'
'IndicaPiso(3)'
'IndicaSentido(Descer)'
EstadoElevador(. 3,Descer,Parado .)
Abre
rpc_Abre_call
rpc_Abre_reply
'Abre()'
ServeChamada(. 2,Descer .)
NaoConfirma
Aberta
PortaNaoConfirma
ConServeChamada(. 2,Descer .)
ChamadaAtribuida(. 2,Descer .)
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
DestinoServido(3)
DestinoServido(3)
MomentoFechar
'ComandaLuz(3,Apagada)'

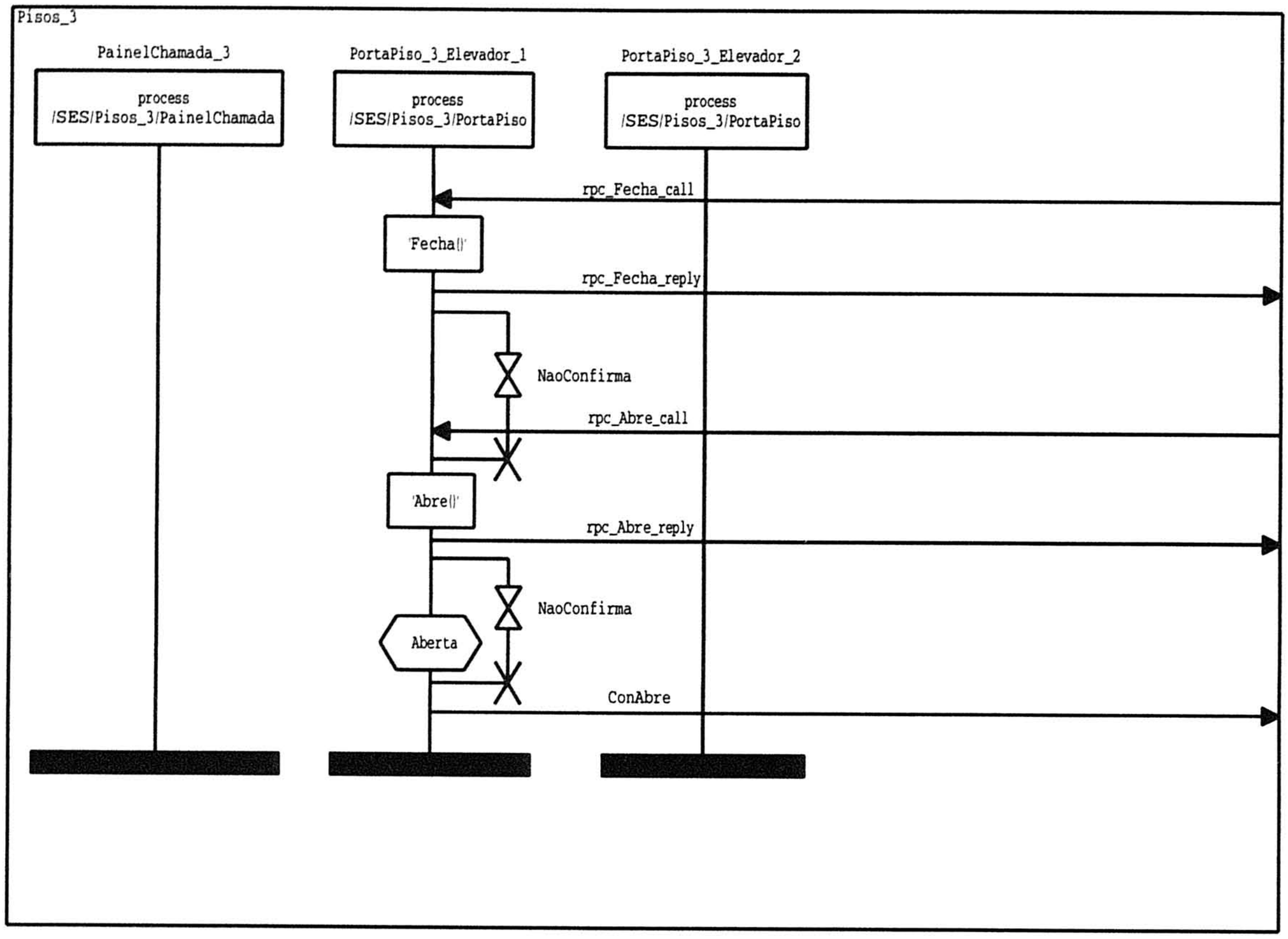

NaoRecebeChamada
Pisos_3
block /SES/Pisos_3
Central
block /SES/Central
Elevadores_1
block /SES/Elevadores_1
NovoPiso
3 =NovoPiso()
'IndicaPiso(Piso)'
'Para()'
'IndicaSentido(Descer)'
EstadoElevador(. 3,Descer,Parado .)
rpc_Abre_call
'Abre()'
'Janela := ActualizaEstado (Janela,1,. 3,Descer,Paradoo.)'
rpc_Abre_reply
ChamadaServida(. 3,Descer .)
'Abre()'
'ComandaLuz(Descer,Apagada)'
'Janela := ApagaChamada (Janela,. 3,Descer.)'
ConAbre
'EmiteSom(ElevadorParado)'
DestinoServido(3)
'Janela := ApagaDestino (Janela,1,3)'
'ComandaLuz(3,Apagada)'

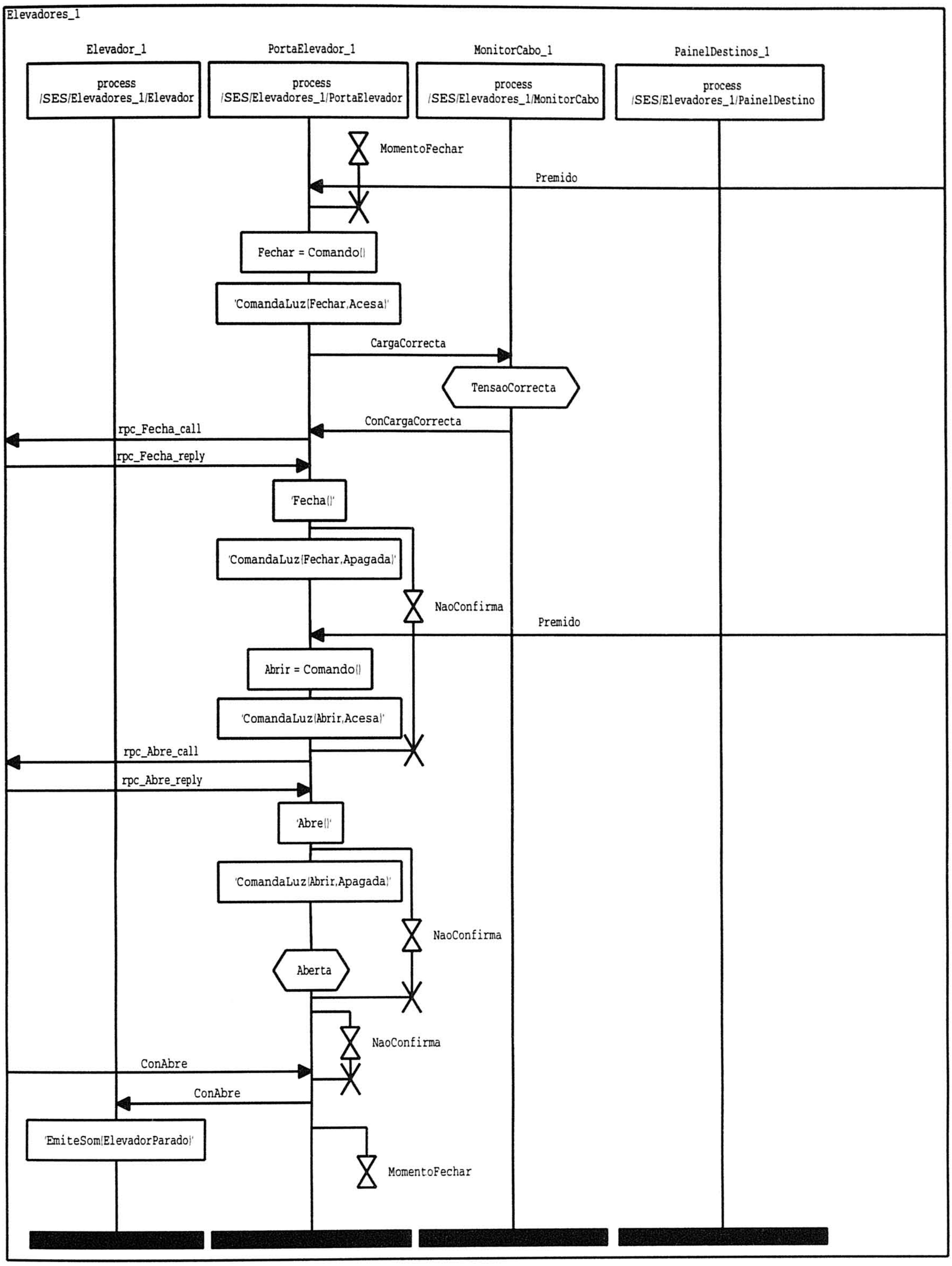

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre
ChamadaServida{|. 3,Descer .|}
'ComandaLuz(Descer,Apagada)'

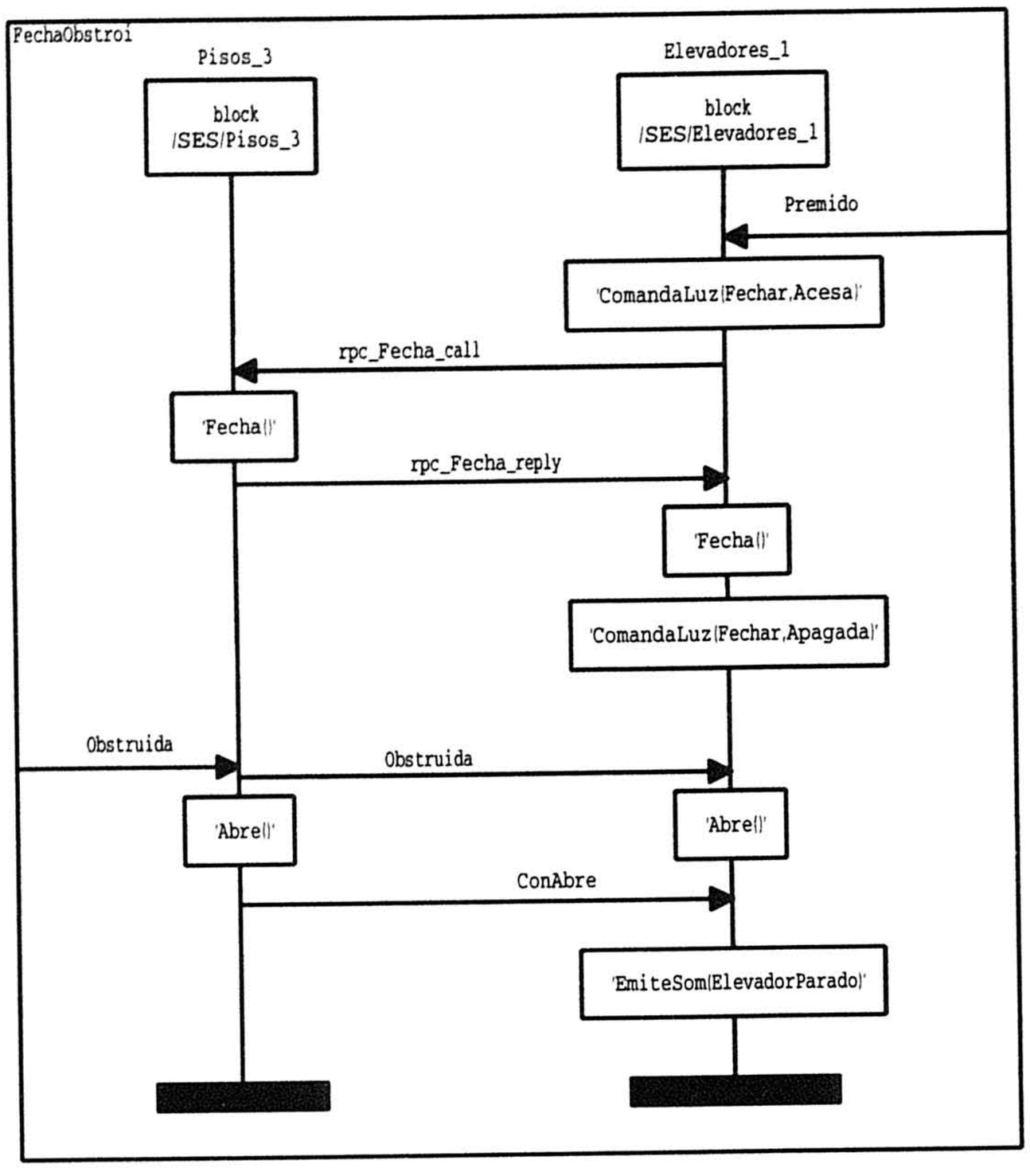

Central
Supervisor_1
Gnomo_1
process /SES/Central/Supervisor
process /SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Parado .|}
EstadoElevador{|. 3,Descer,Parado .|}
'Janela := ActualizaEstado (Janela,1,{. 3,Descer,Parado.|}'
ChamadaServida{|. 3,Descer .|}
'Janela := ApagaChamada (Janela,{. 3,Descer.|}'
ChamadaServida{|. 3,Descer .|}
Nao_ChamSeguinte_MaisProximo_DifernteReservada
DestinoServido{3}
'Janela := ApagaDestino (Janela,1,3}'

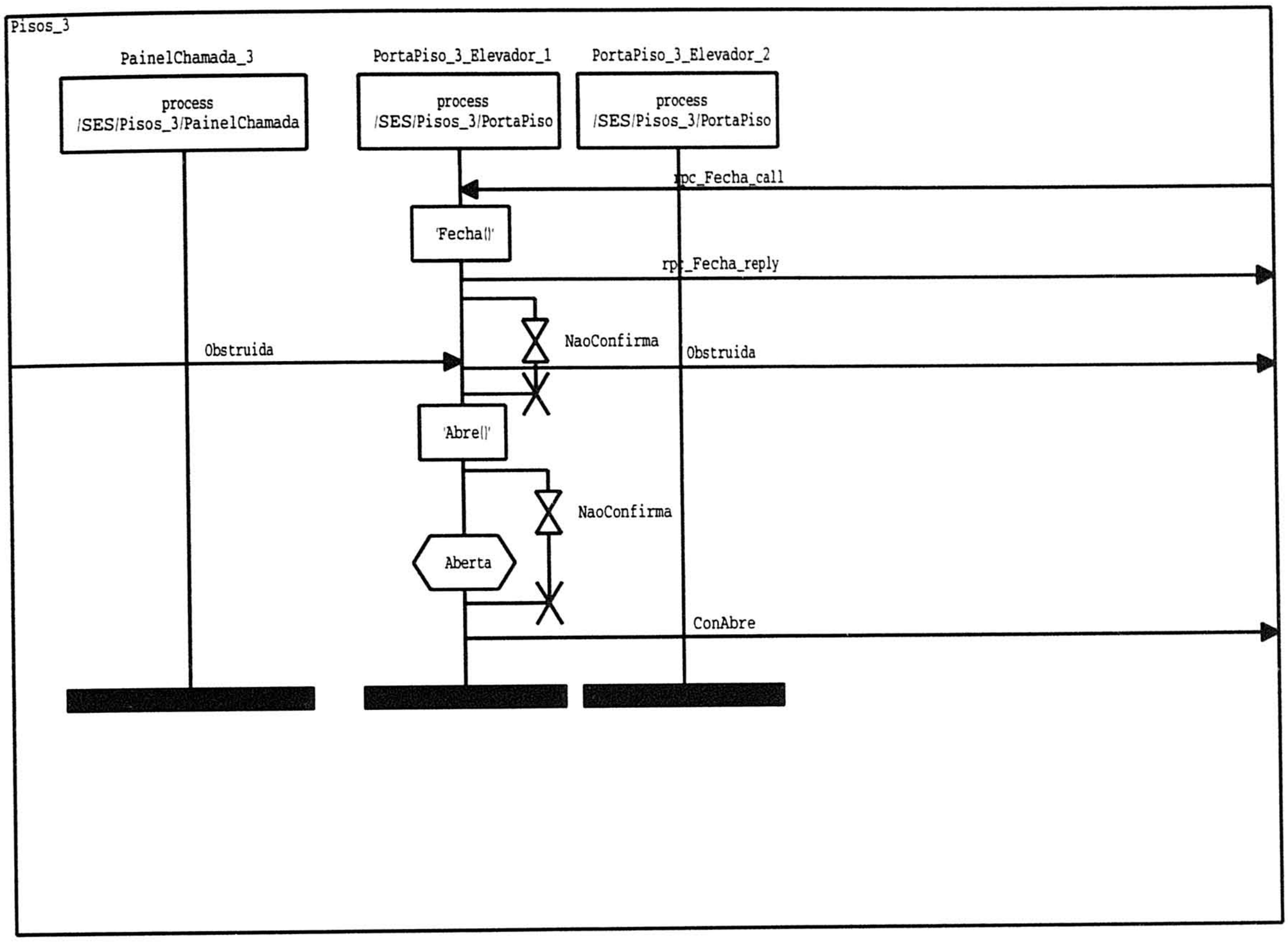

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
NovoPiso
3 := NovoPiso()
'Para()'
'IndicaPiso(3)'
'IndicaSentido(Descer)'
EstadoElevador(. 3,Descer,Parado .)
Abre
rpc_Abre_call
rpc_Abre_reply
'Abre()'
Aberta
NaoConfirma
PortaNaoConfirma
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
DestinoServido(3)
DestinoServido(3)
MomentoFechar
'ComandaLuz(3,Apagada)'

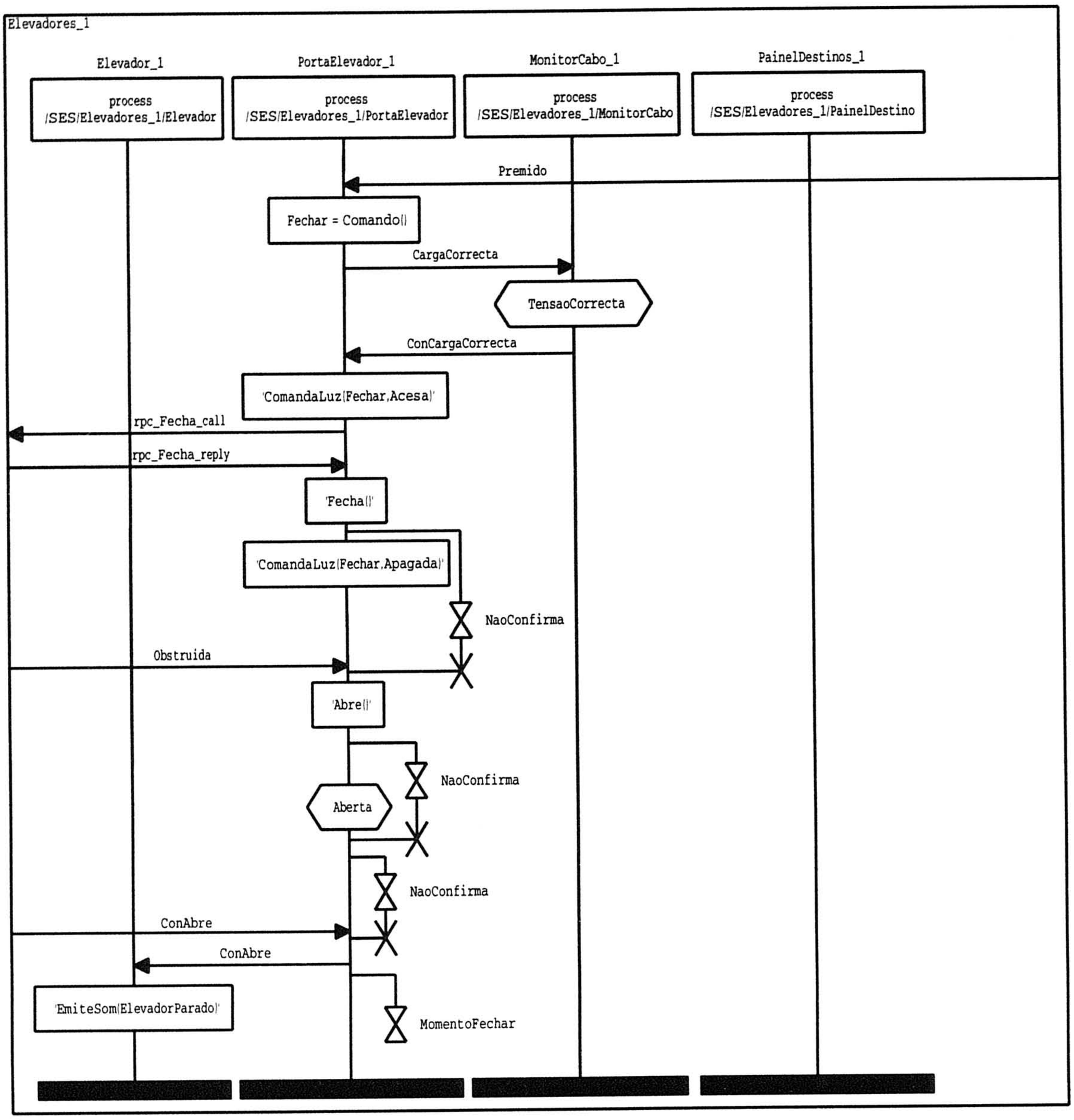

FechaAbre
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
Premido
'ComandaLuz(Fechar,Acesa)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
'ComandaLuz(Fechar,Apagada)'
Premido
'ComandaLuz(Abrir,Acesa)'
rpc_Abre_call
'Abre()'
rpc_Abre_reply
'Abre()'
'ComandaLuz(Abrir,Apagada)'
ConAbre
'EmiteSom(ElevadorParado)'

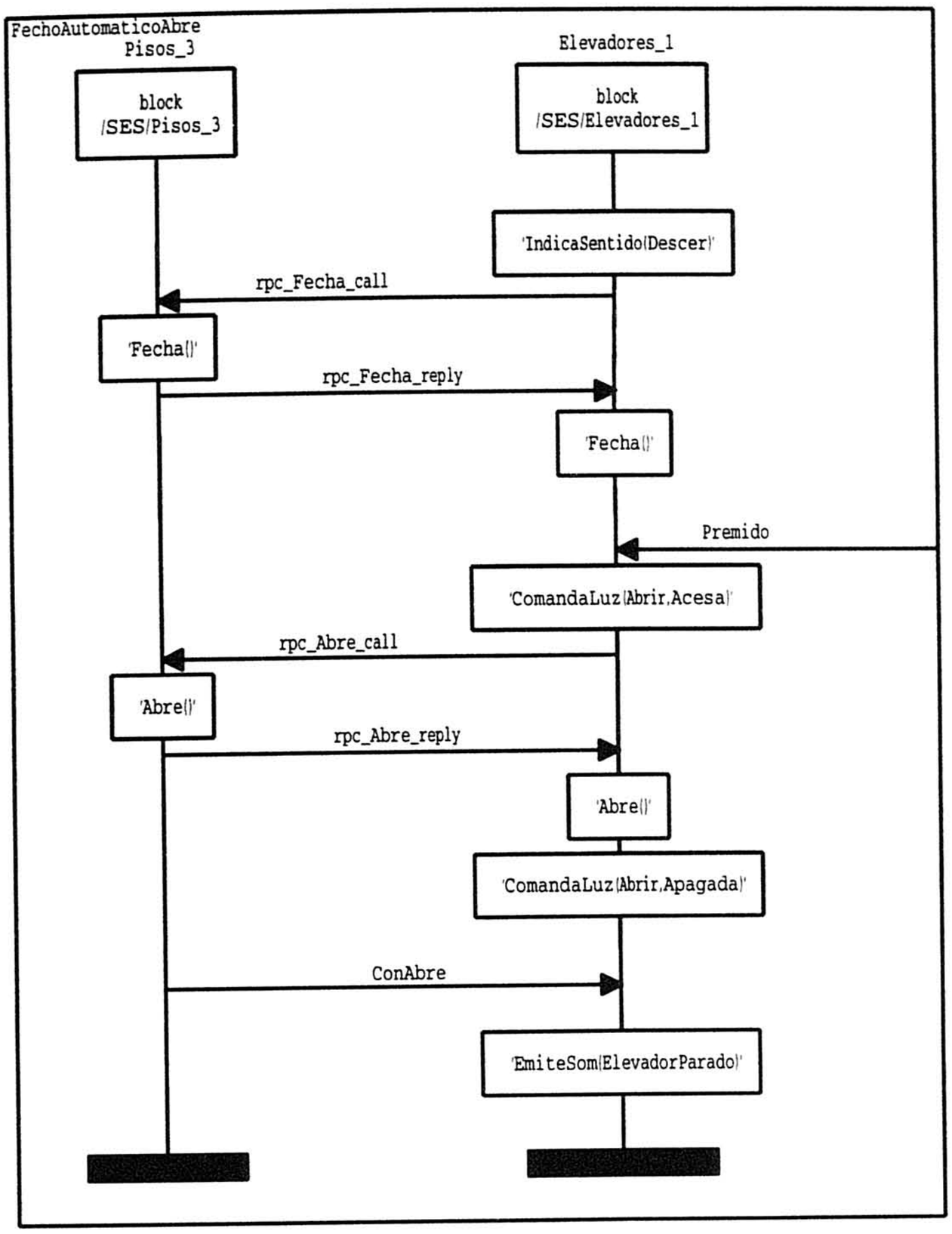

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre

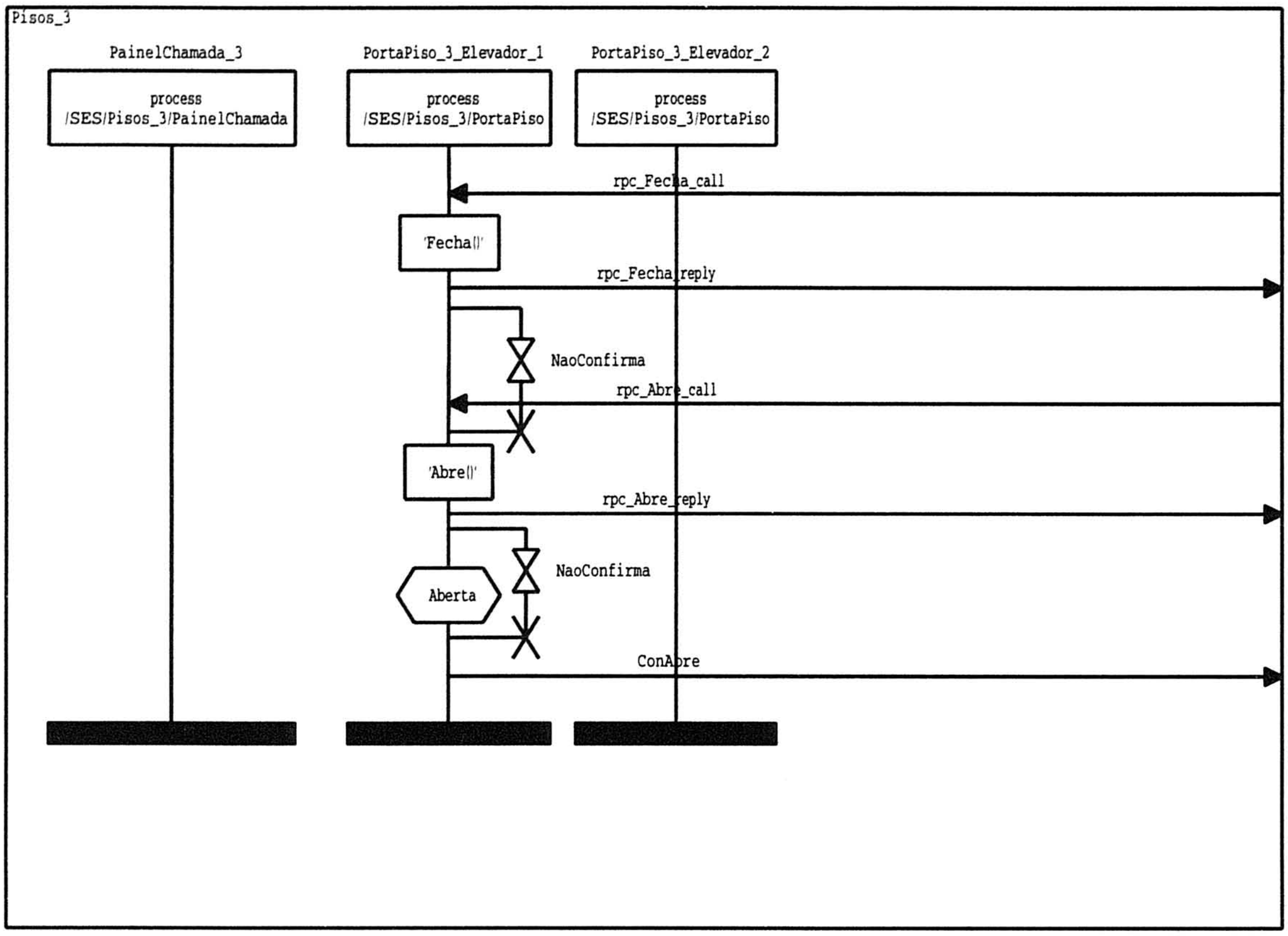

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
Premido
Fechar = Comando()
'ComandaLuz(Fechar,Acesa)'
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
'ComandaLuz(Fechar,Apagada)'
NaoConfirma
Premido
Abrir = Comando()
'ComandaLuz(Abrir,Acesa)'
rpc_Abre_call
rpc_Abre_reply
'Abre()'
'ComandaLuz(Abrir,Apagada)'
NaoConfirma
Aberta
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
MomentoFechar

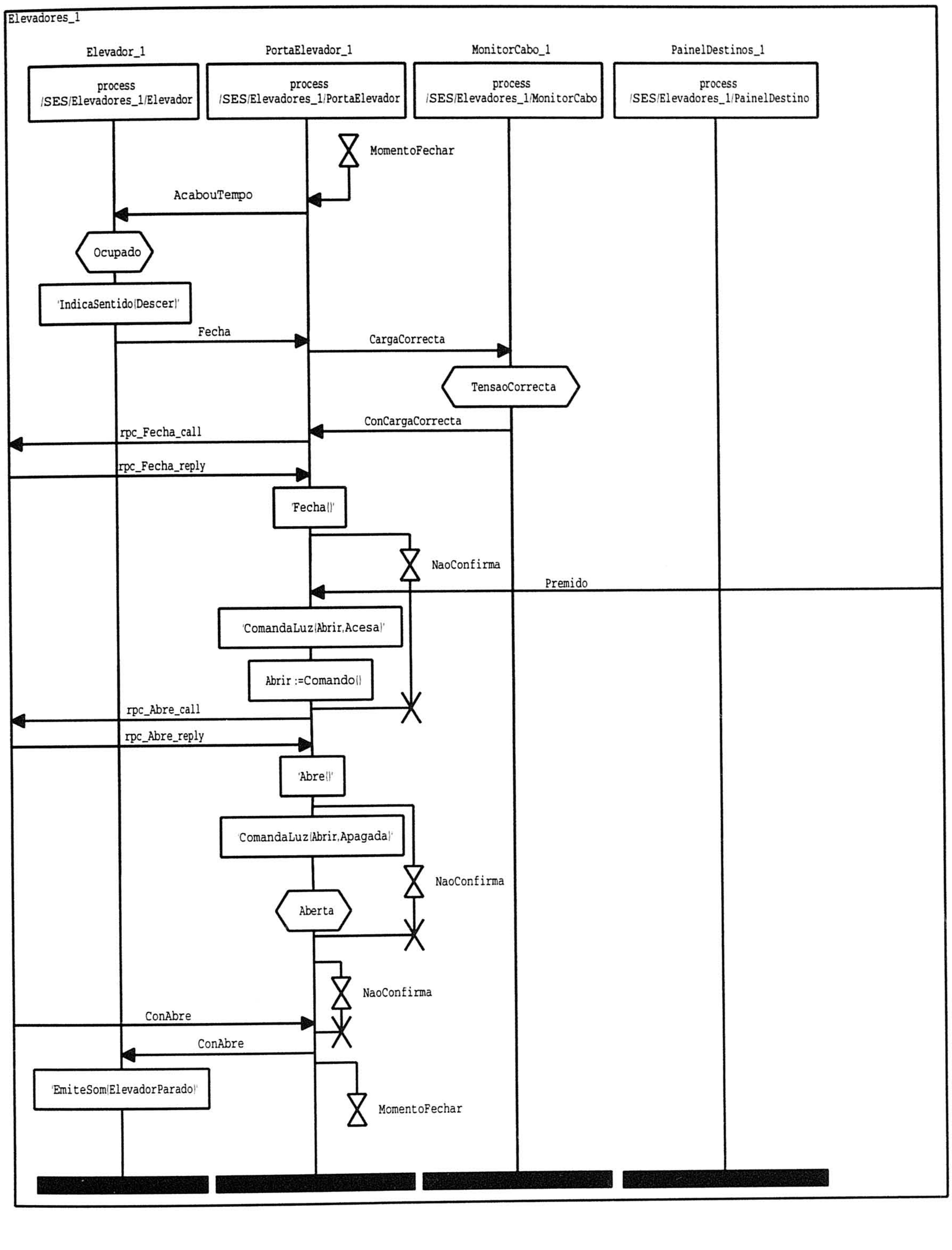

FechaObstroi
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
Premido
'ComandaLuz(Fechar,Acesa)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
'ComandaLuz(Fechar,Apagada)'
Obstruida
Obstruida
'Abre()'
'Abre()'
ConAbre
'EmiteSom(ElevadorParado)'

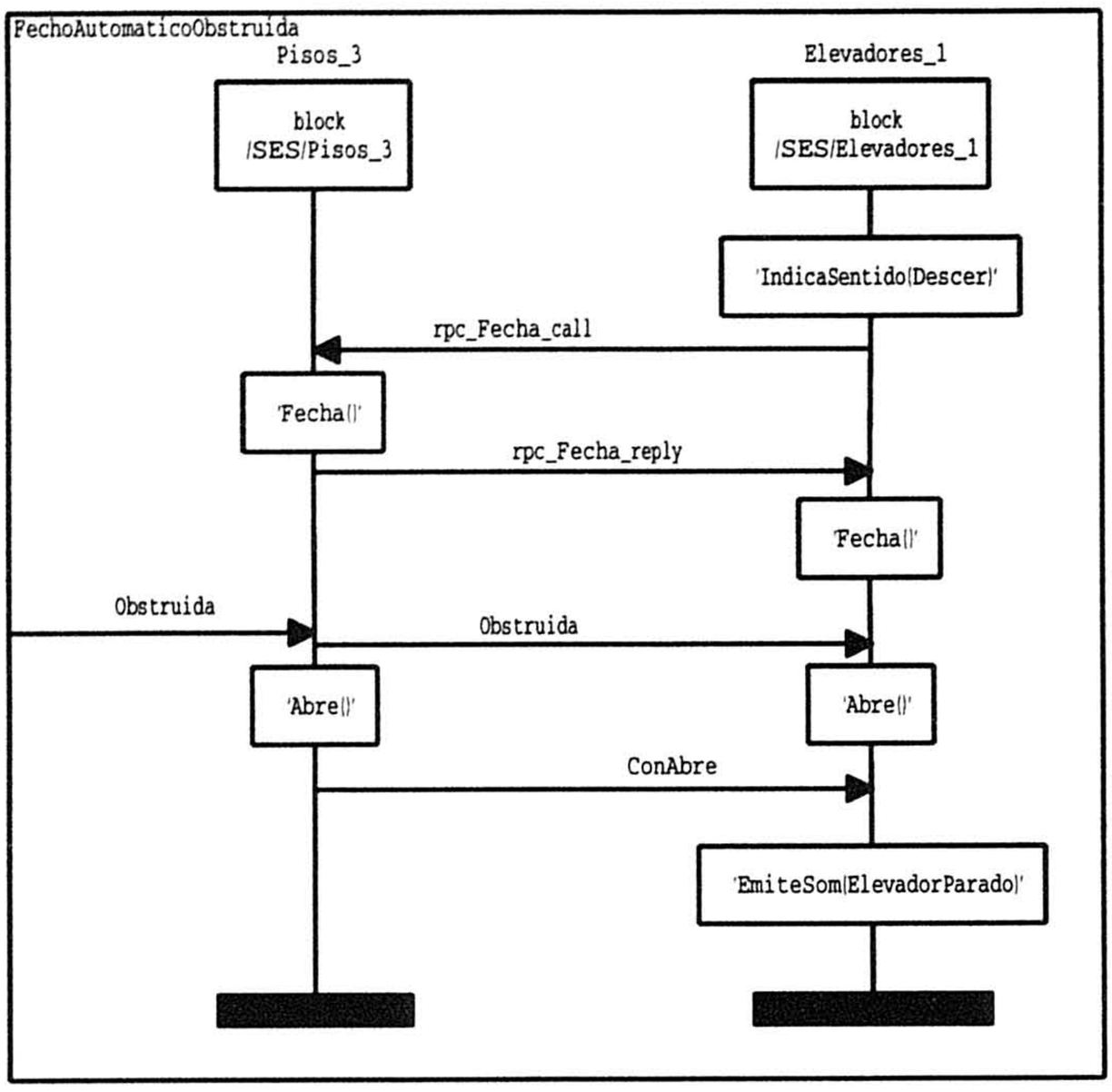

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Obstruida
Obstruida
'Abre()'
NaoConfirma
Aberta
ConAbre

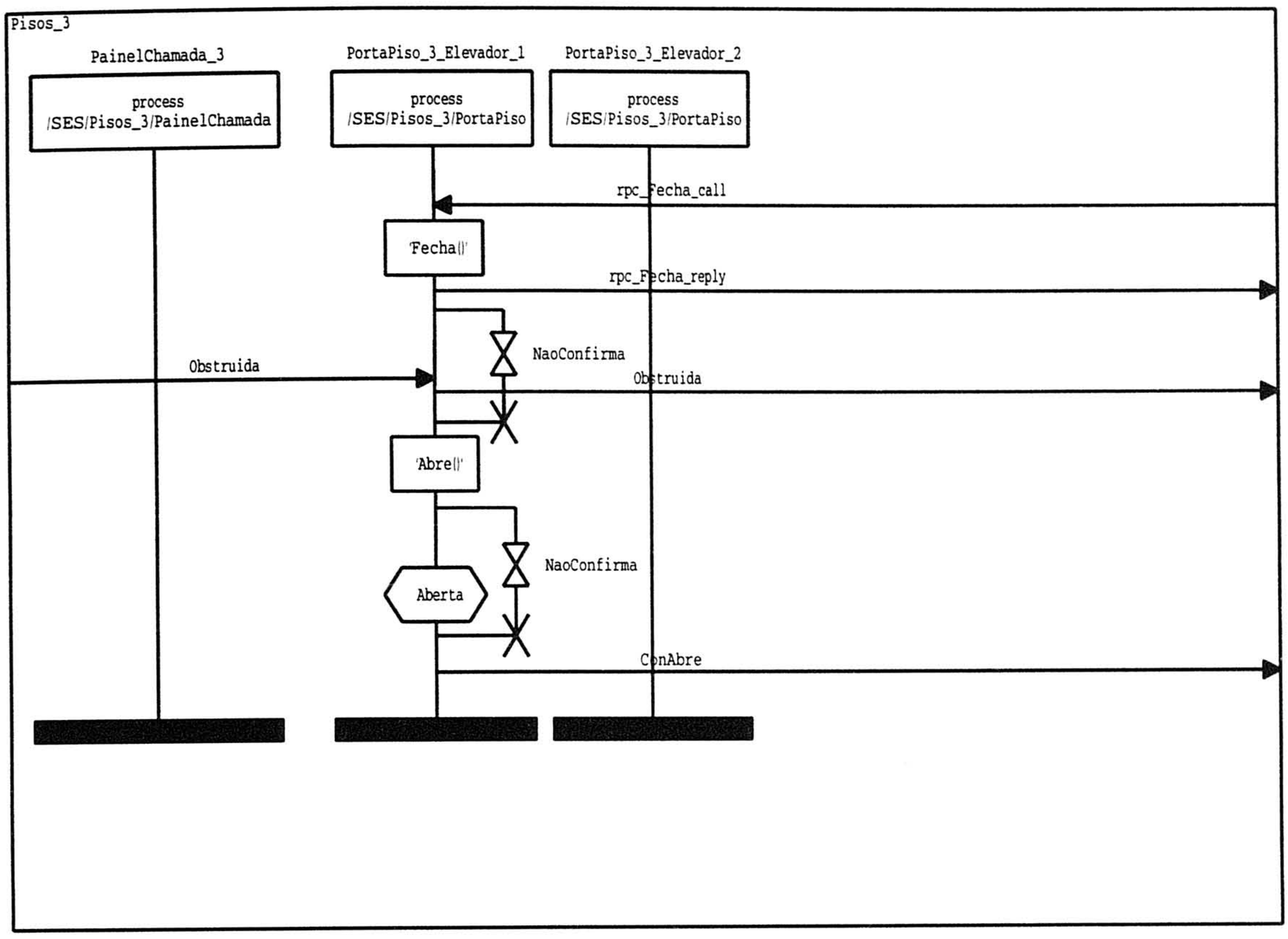

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
Premido
Fechar = Comando()
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
'ComandaLuz(Fechar,Acesa)'
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
'ComandaLuz(Fechar,Apagada)'
NaoConfirma
Obstruida
'Abre()'
NaoConfirma
Aberta
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
MomentoFechar

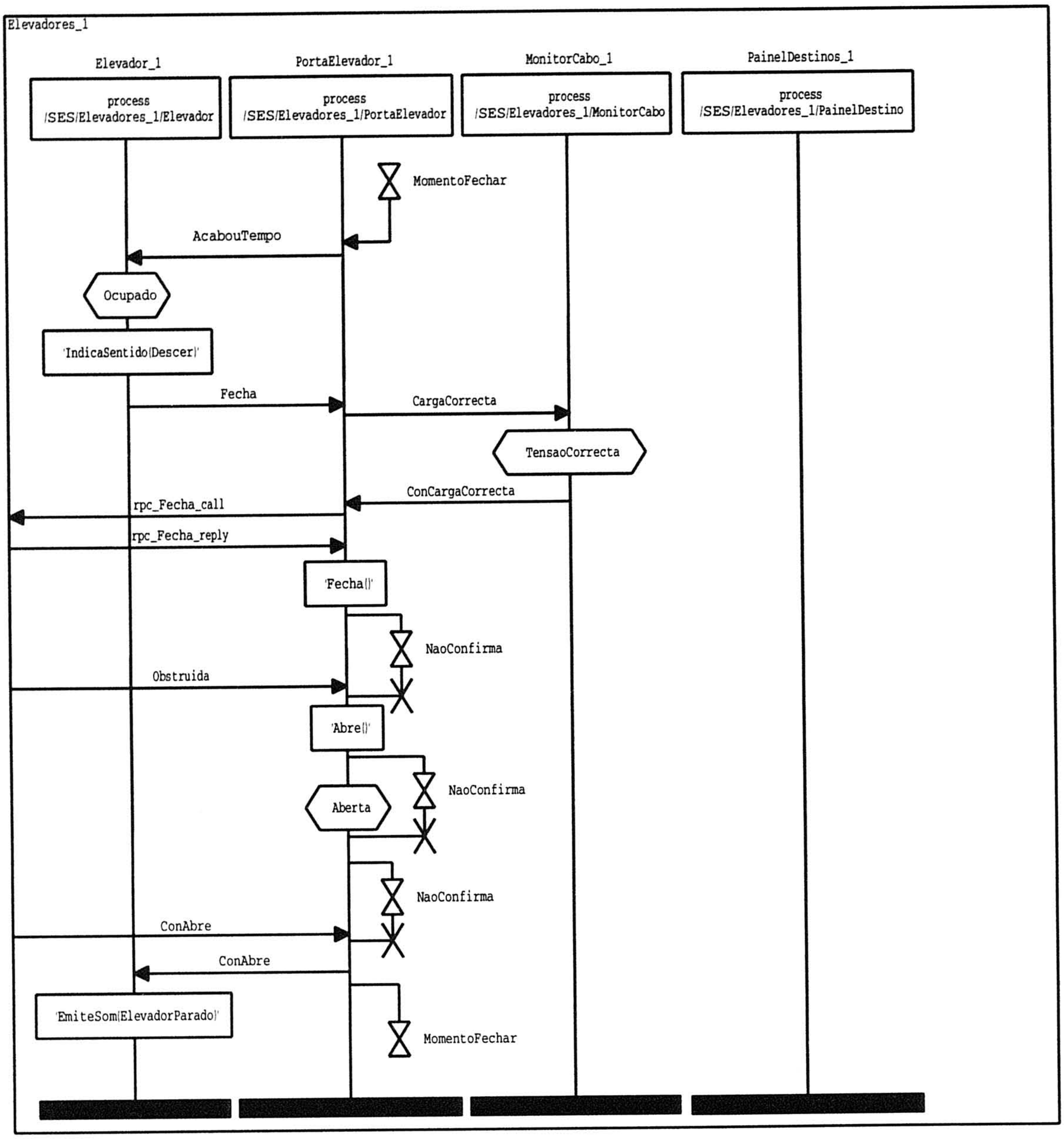

FechoAutomaticoAbre
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
'IndicaSentido(Descer)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
Premido
'ComandaLuz(Abrir,Acesa)'
rpc_Abre_call
'Abre()'
rpc_Abre_reply
'Abre()'
'ComandaLuz(Abrir,Apagada)'
ConAbre
'EmiteSom(ElevadorParado)'

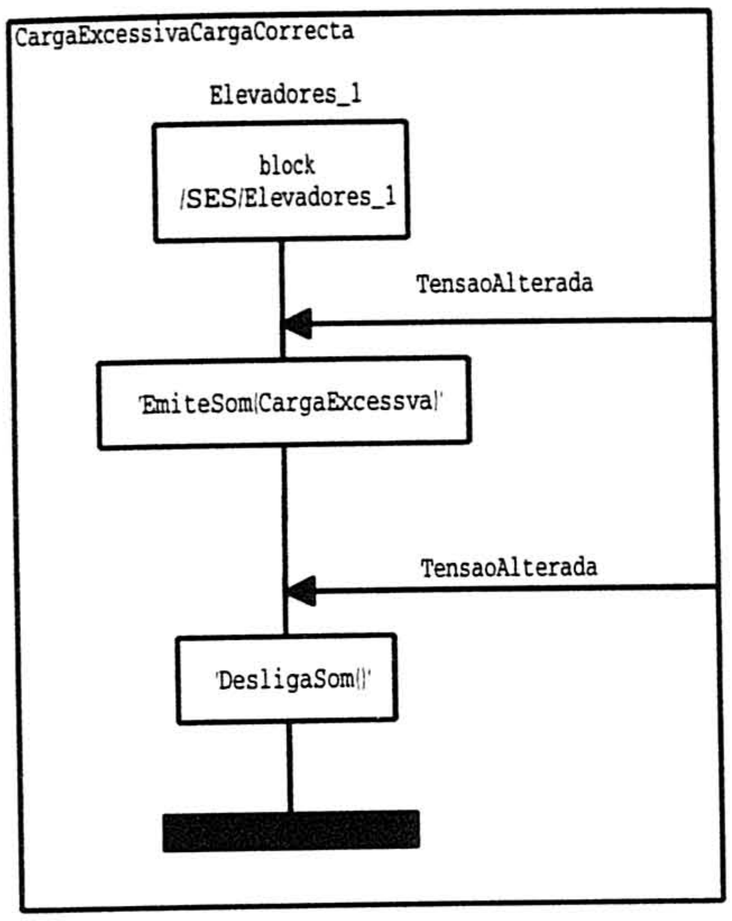

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre

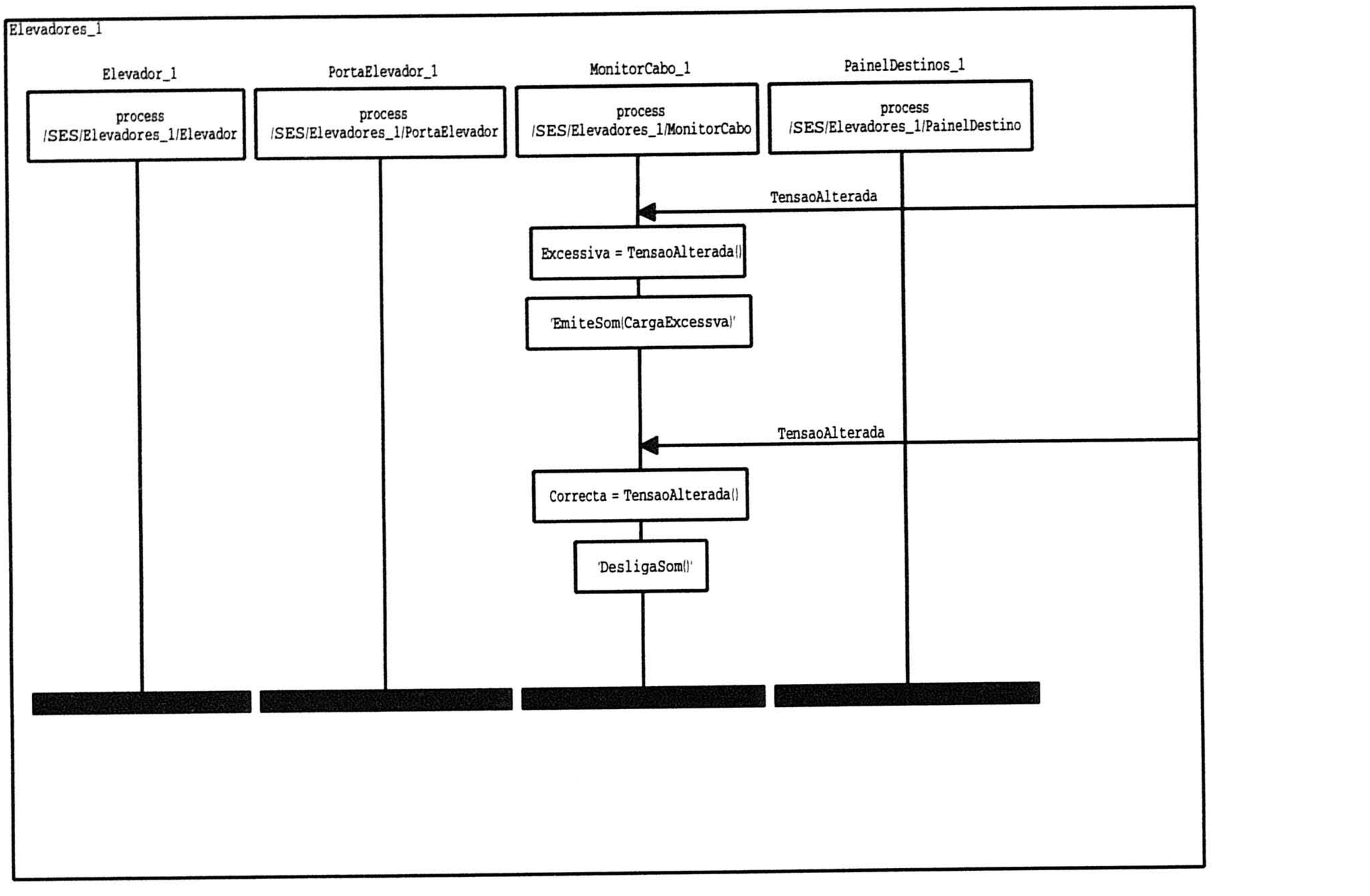

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
AcabouTempo
Ocupado
'IndicaSentido(Descer)'
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Premido
'ComandaLuz(Abrir,Acesa)'
Abrir :=Comando()
rpc_Abre_call
rpc_Abre_reply
'Abre()'
'ComandaLuz(Abrir,Apagada)'
NaoConfirma
Aberta
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
MomentoFechar

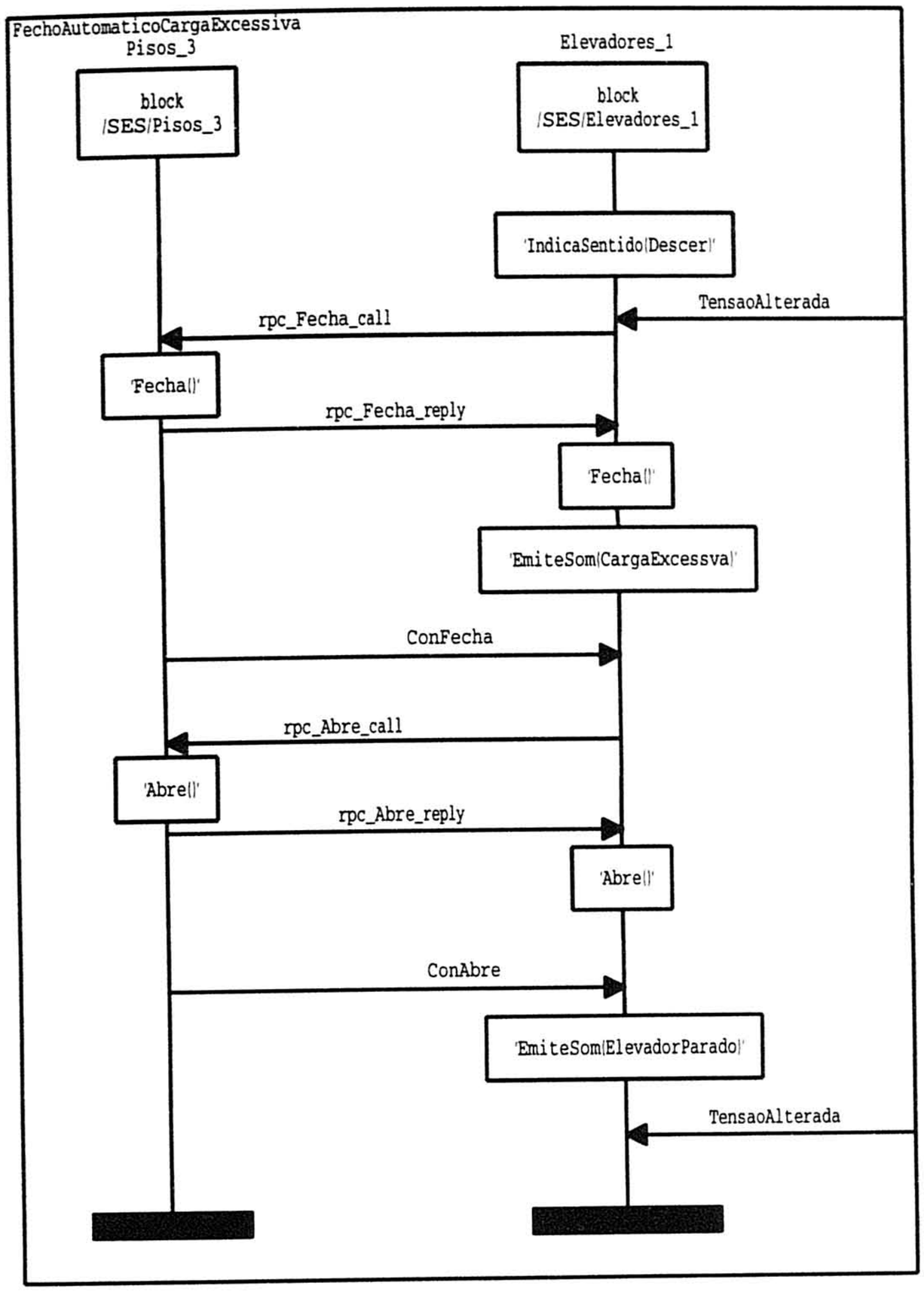

FechoAutomaticoObstruida
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
'IndicaSentido(Descer)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
Obstruida
Obstruida
'Abre()'
'Abre()'
ConAbre
'EmiteSom(ElevadorParado)'

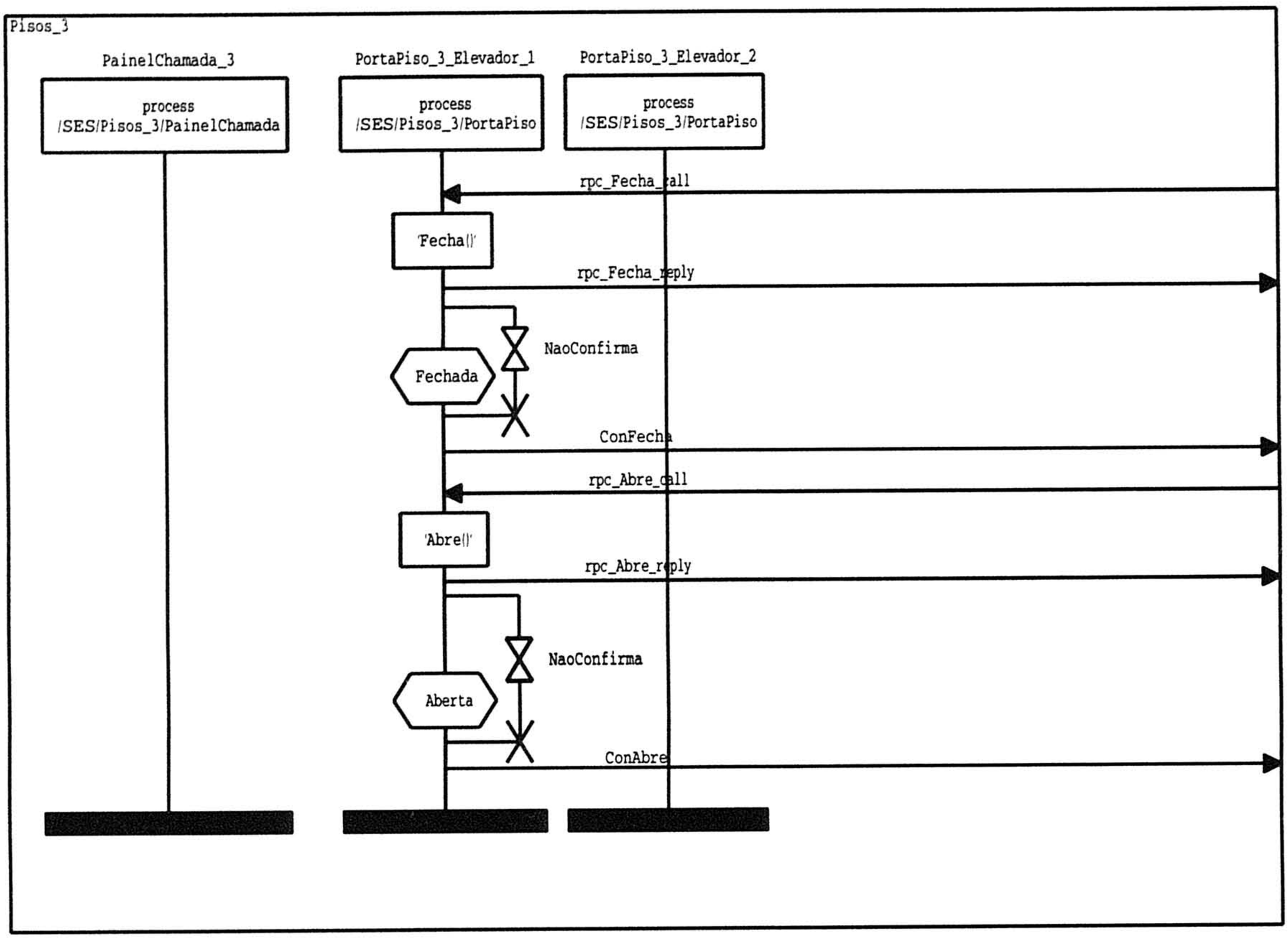

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Obstruida
Obstruida
'Abre()'
NaoConfirma
Aberta
ConAbre

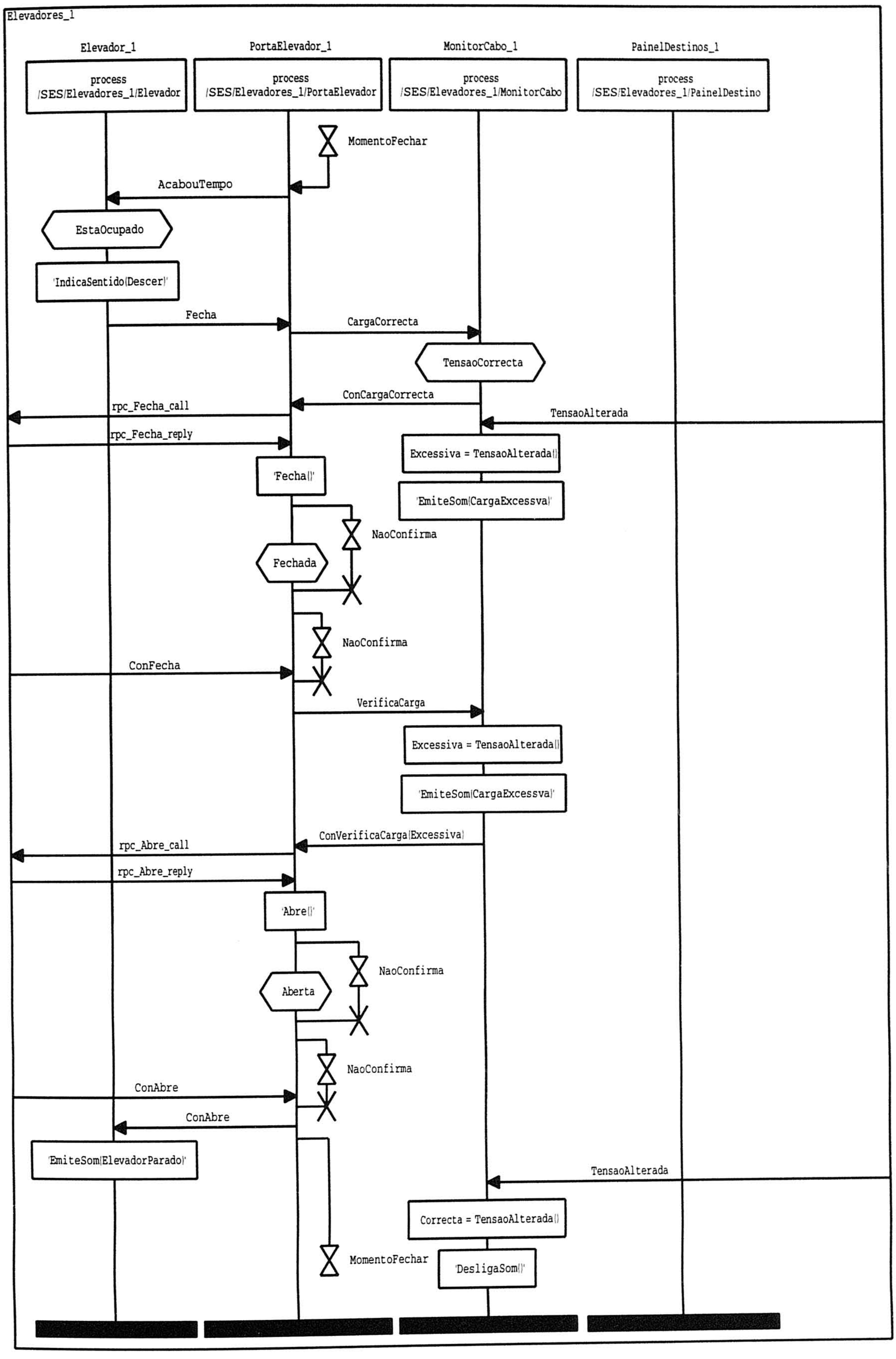

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
AcabouTempo
Ocupado
'IndicaSentido(Descer)'
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Obstruida
'Abre()'
NaoConfirma
Aberta
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
MomentoFechar

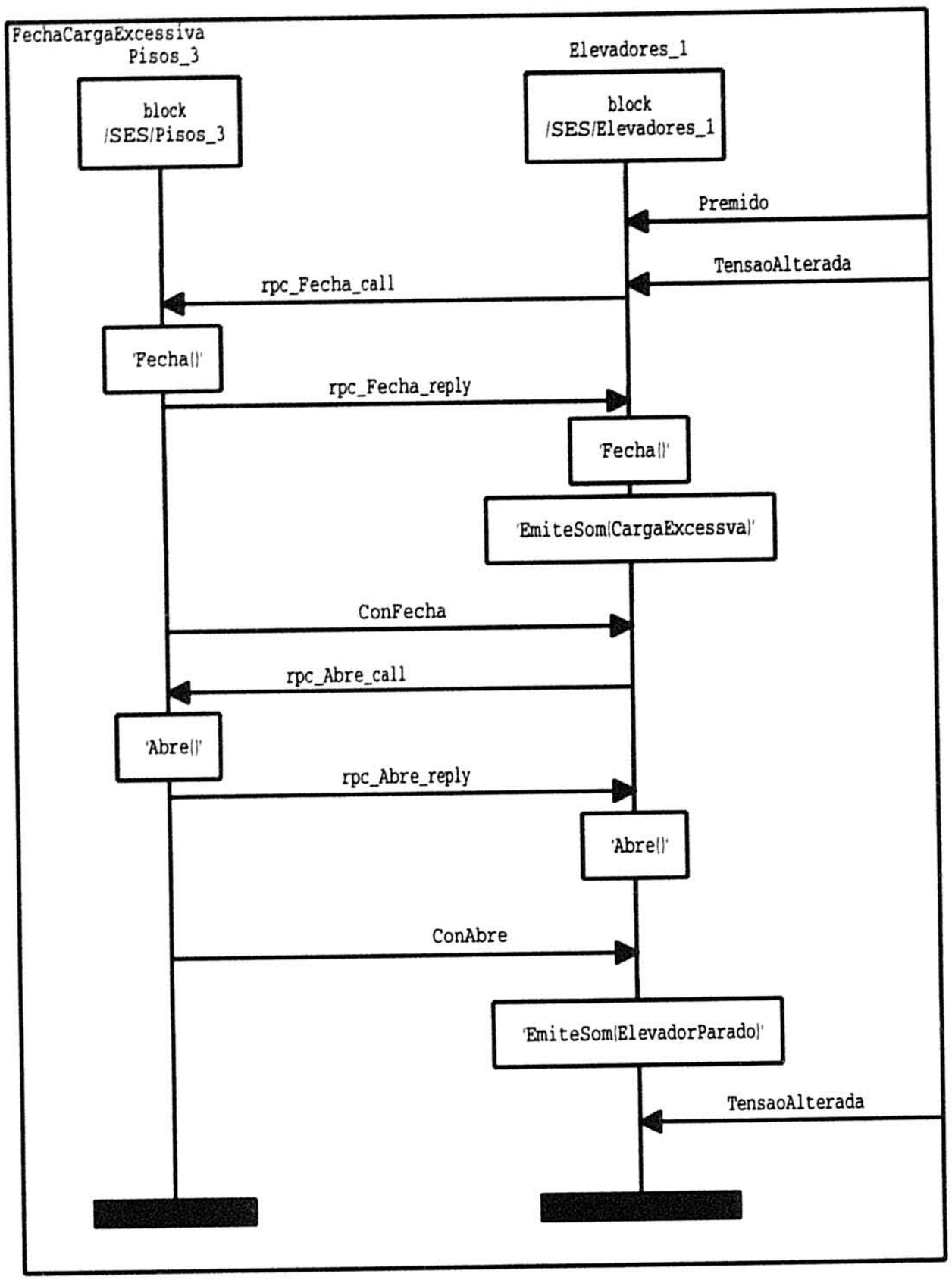

CargaExcessivaCargaCorrecta
Elevadores_1
block
/SES/Elevadores_1
TensaoAlterada
'EmiteSom(CargaExcessva)'
TensaoAlterada
'DesligaSom()'

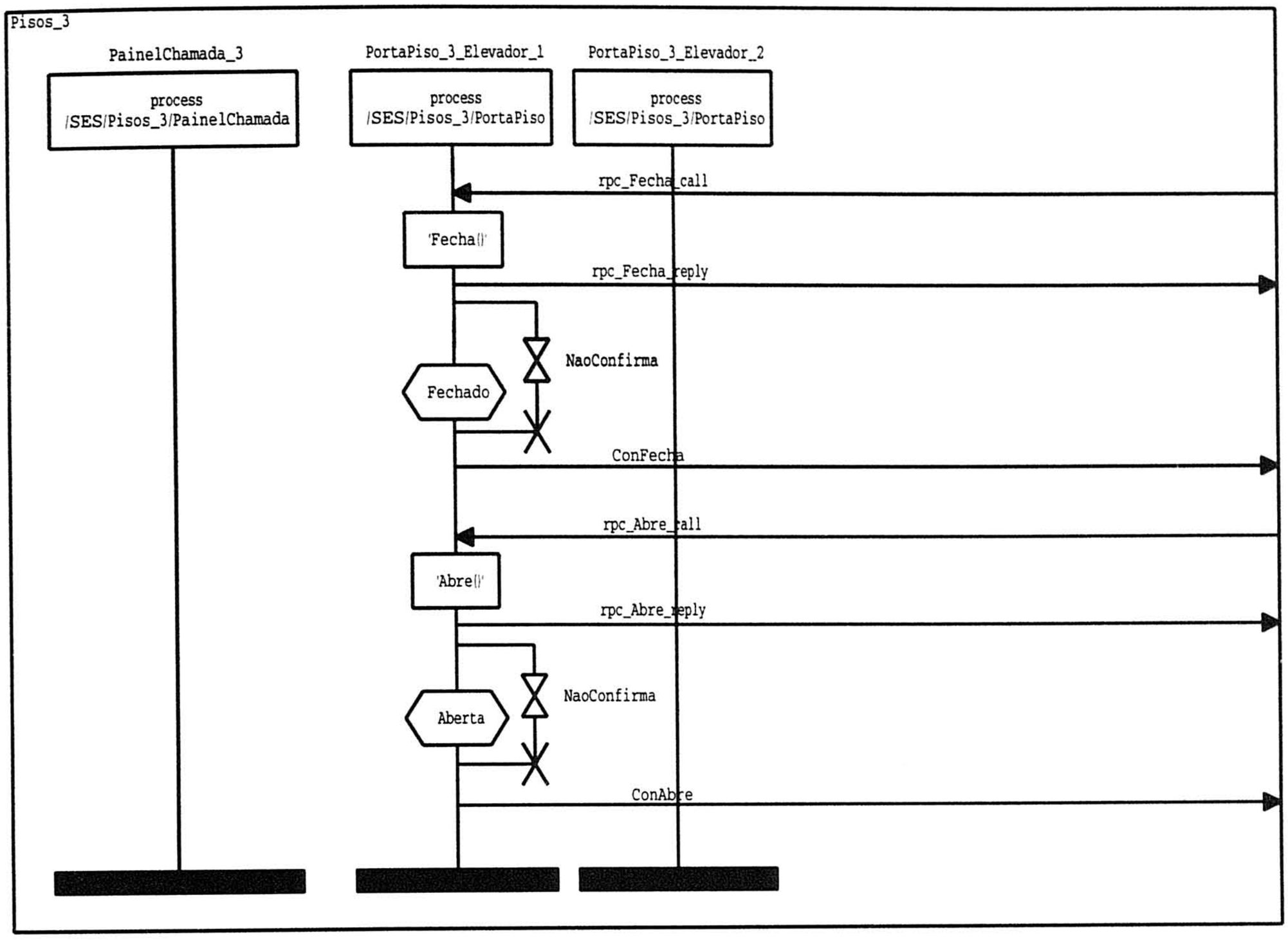

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
TensaoAlterada
Excessiva = TensaoAlterada()
'EmiteSom(CargaExcessva)'
TensaoAlterada
Correcta = TensaoAlterada()
'DesligaSom()'

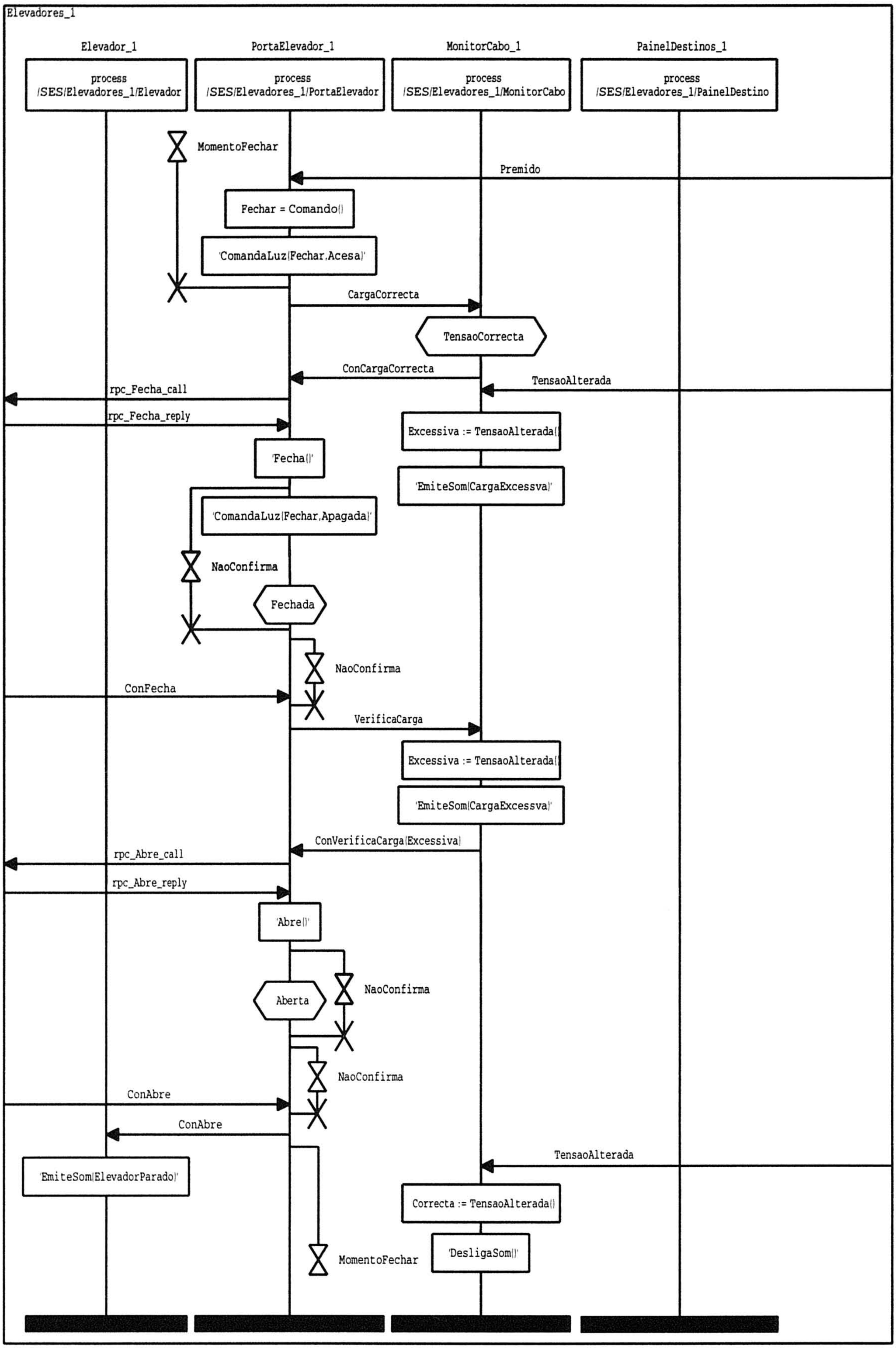

FechoAutomaticoCargaExcessiva
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
'IndicaSentido(Descer)'
TensaoAlterada
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
'EmiteSom(CargaExcessva)'
ConFecha
rpc_Abre_call
'Abre()'
rpc_Abre_reply
'Abre()'
ConAbre
'EmiteSom(ElevadorParado)'
TensaoAlterada

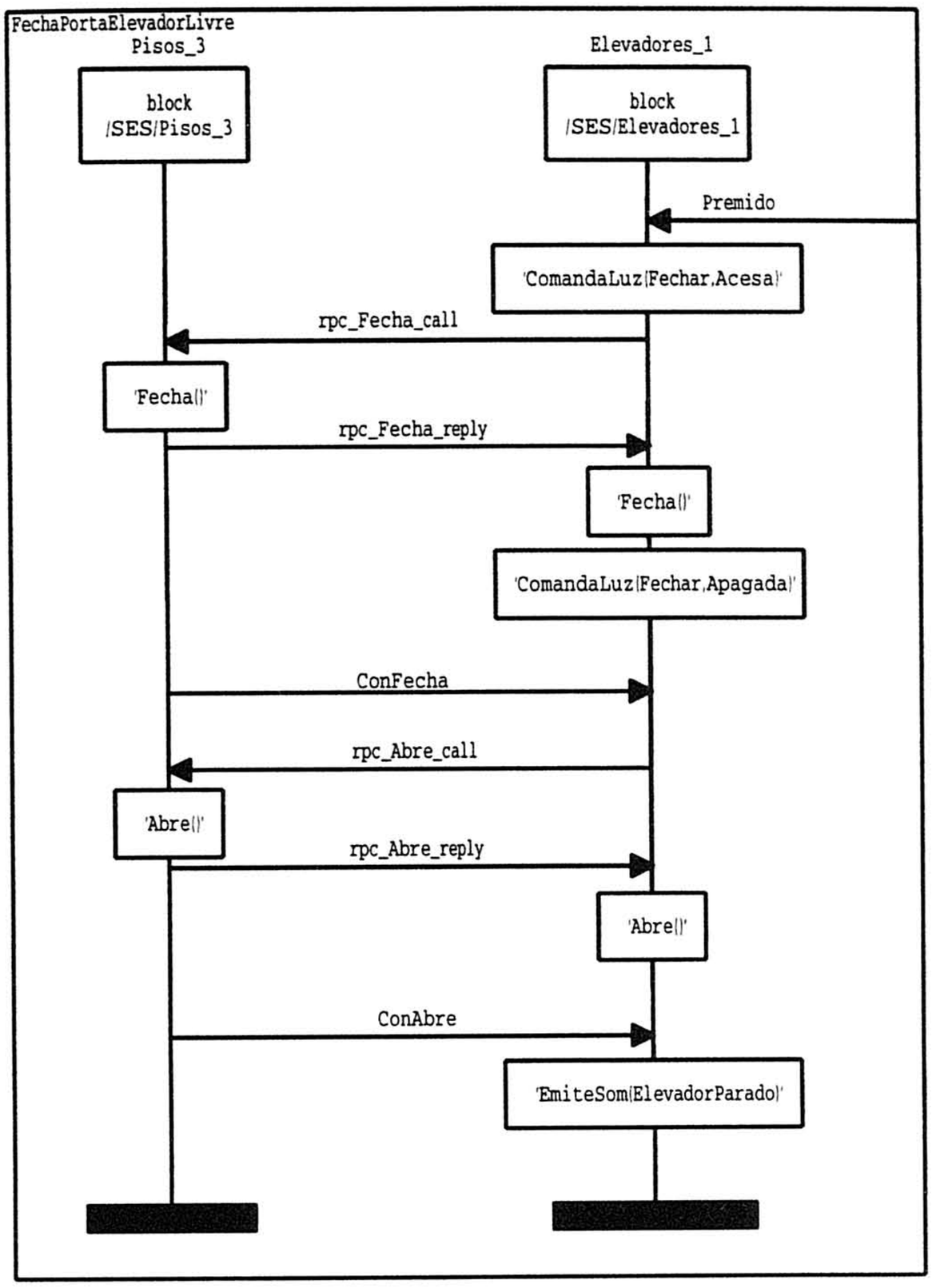

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre

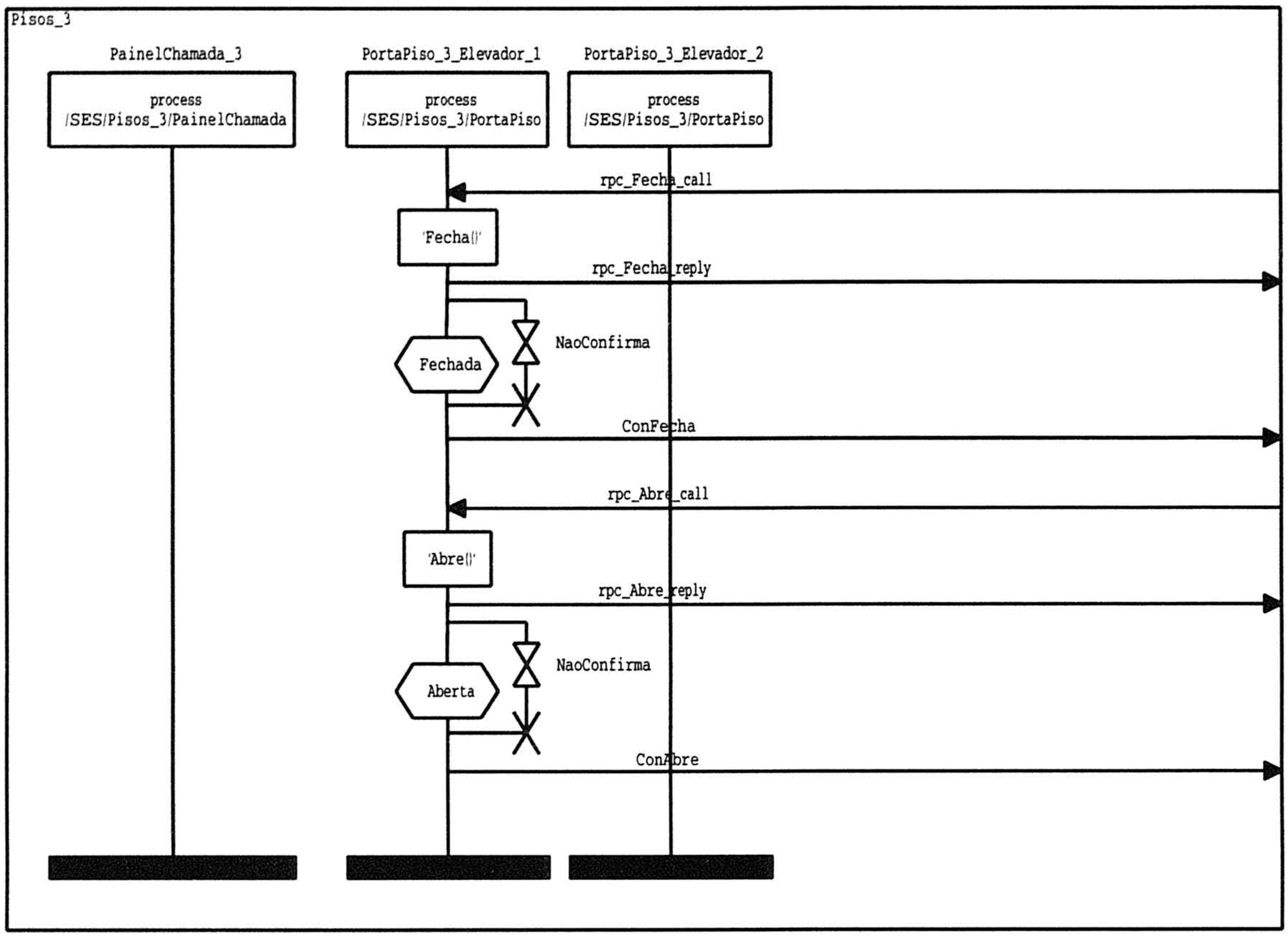

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
AcabouTempo
EstaOcupado
'IndicaSentido(Descer)'
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
TensaoAlterada
rpc_Fecha_reply
Excessiva = TensaoAlterada()
'Fecha()'
'EmiteSom(CargaExcessva)'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Excessiva = TensaoAlterada()
'EmiteSom(CargaExcessva)'
ConVerificaCarga(Excessiva)
rpc_Abre_call
rpc_Abre_reply
'Abre()'
NaoConfirma
Aberta
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
TensaoAlterada
Correcta = TensaoAlterada()
MomentoFechar
'DesligaSom()'

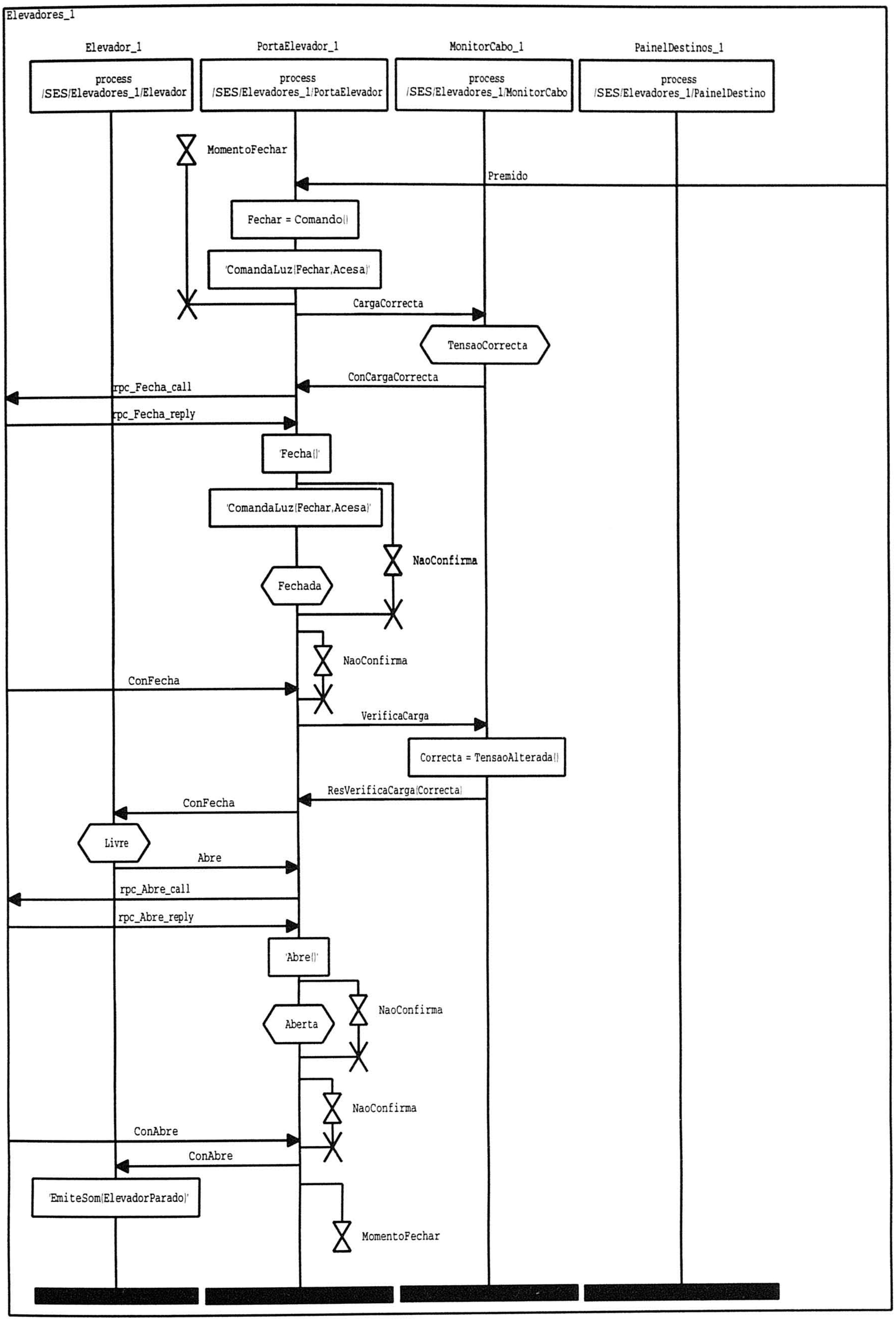

FechaCargaExcessiva
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
Premido
TensaoAlterada
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
'EmiteSom(CargaExcessva)'
ConFecha
rpc_Abre_call
'Abre()'
rpc_Abre_reply
'Abre()'
ConAbre
'EmiteSom(ElevadorParado)'
TensaoAlterada

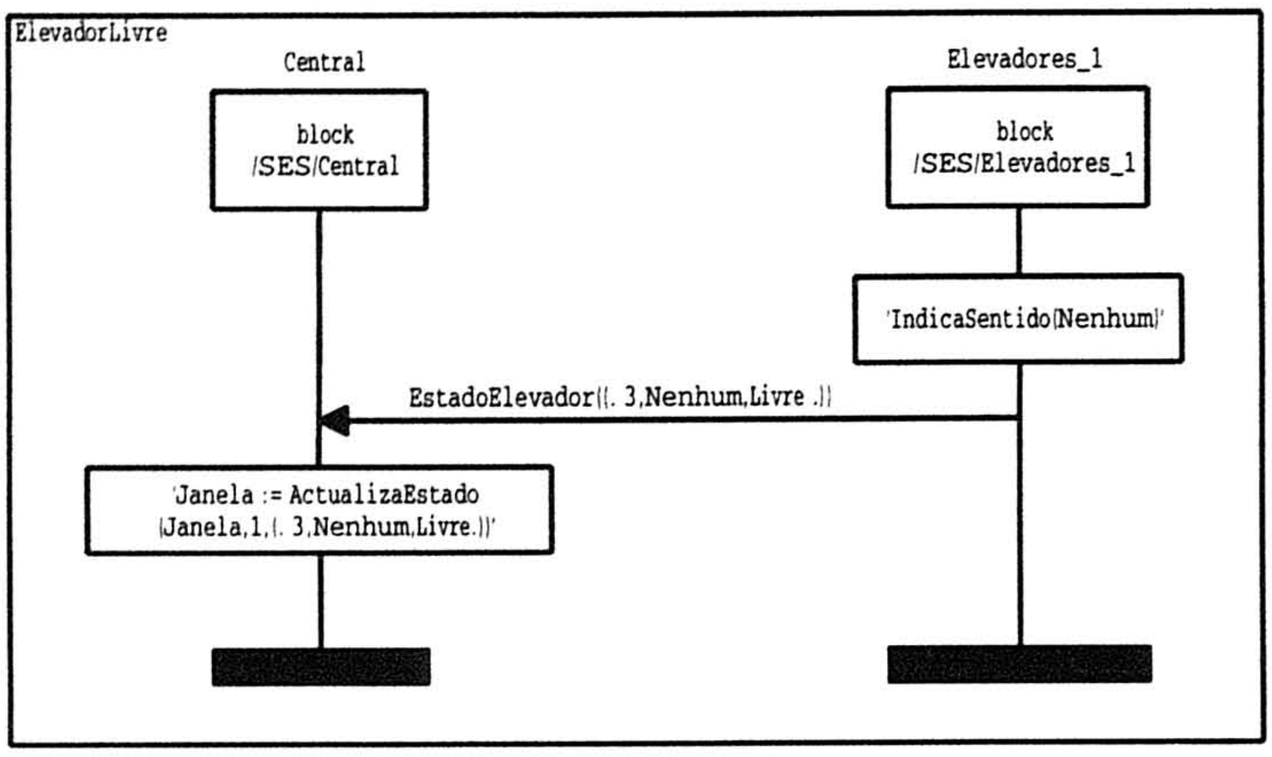

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechado
ConFecha
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre

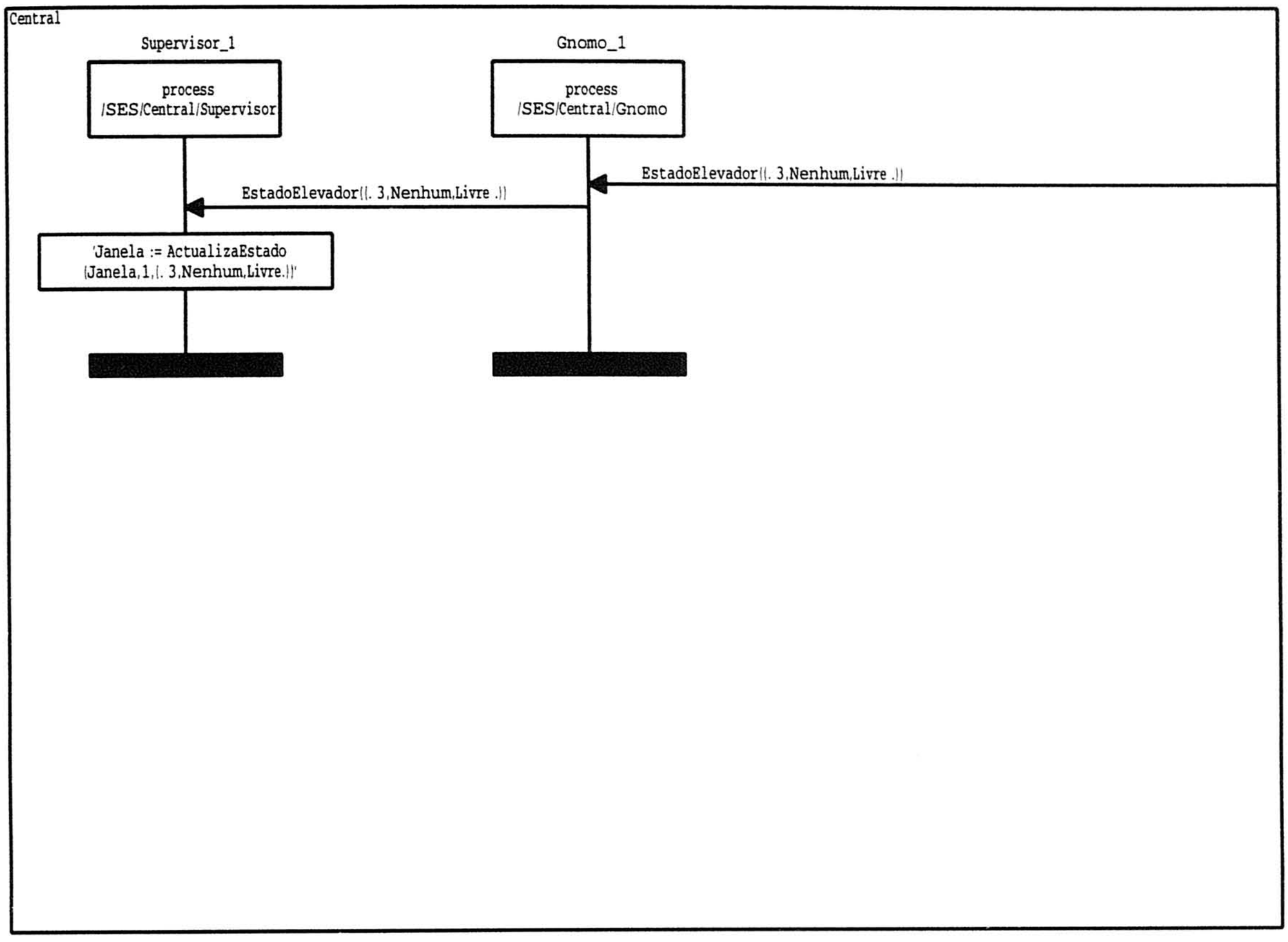

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
Premido
Fechar = Comando()
'ComandaLuz(Fechar,Acesa)'
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
TensaoAlterada
rpc_Fecha_call
rpc_Fecha_reply
Excessiva := TensaoAlterada()
'Fecha()'
'EmiteSom(CargaExcessva)'
'ComandaLuz(Fechar,Apagada)'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Excessiva := TensaoAlterada()
'EmiteSom(CargaExcessva)'
ConVerificaCarga(Excessiva)
rpc_Abre_call
rpc_Abre_reply
'Abre()'
Aberta
NaoConfirma
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
MomentoFechar
TensaoAlterada
Correcta := TensaoAlterada()
'DesligaSom()'

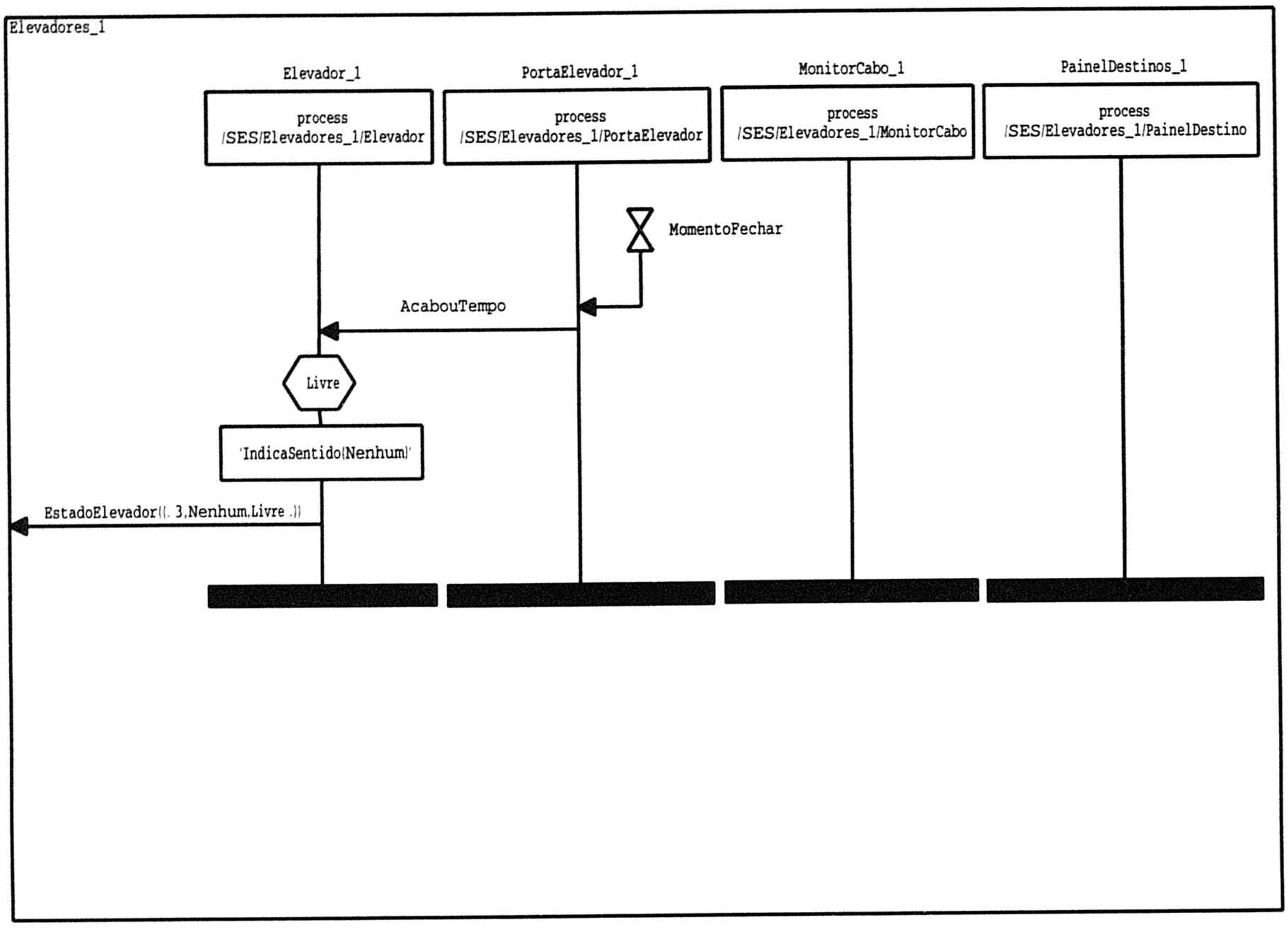

FechaPortaElevadorLivre
Pisos_3
Elevadores_1
block /SES/Pisos_3
block /SES/Elevadores_1
Premido
'ComandaLuz(Fechar,Acesa)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
'ComandaLuz(Fechar,Apagada)'
ConFecha
rpc_Abre_call
'Abre()'
rpc_Abre_reply
'Abre()'
ConAbre
'EmiteSom(ElevadorParado)'

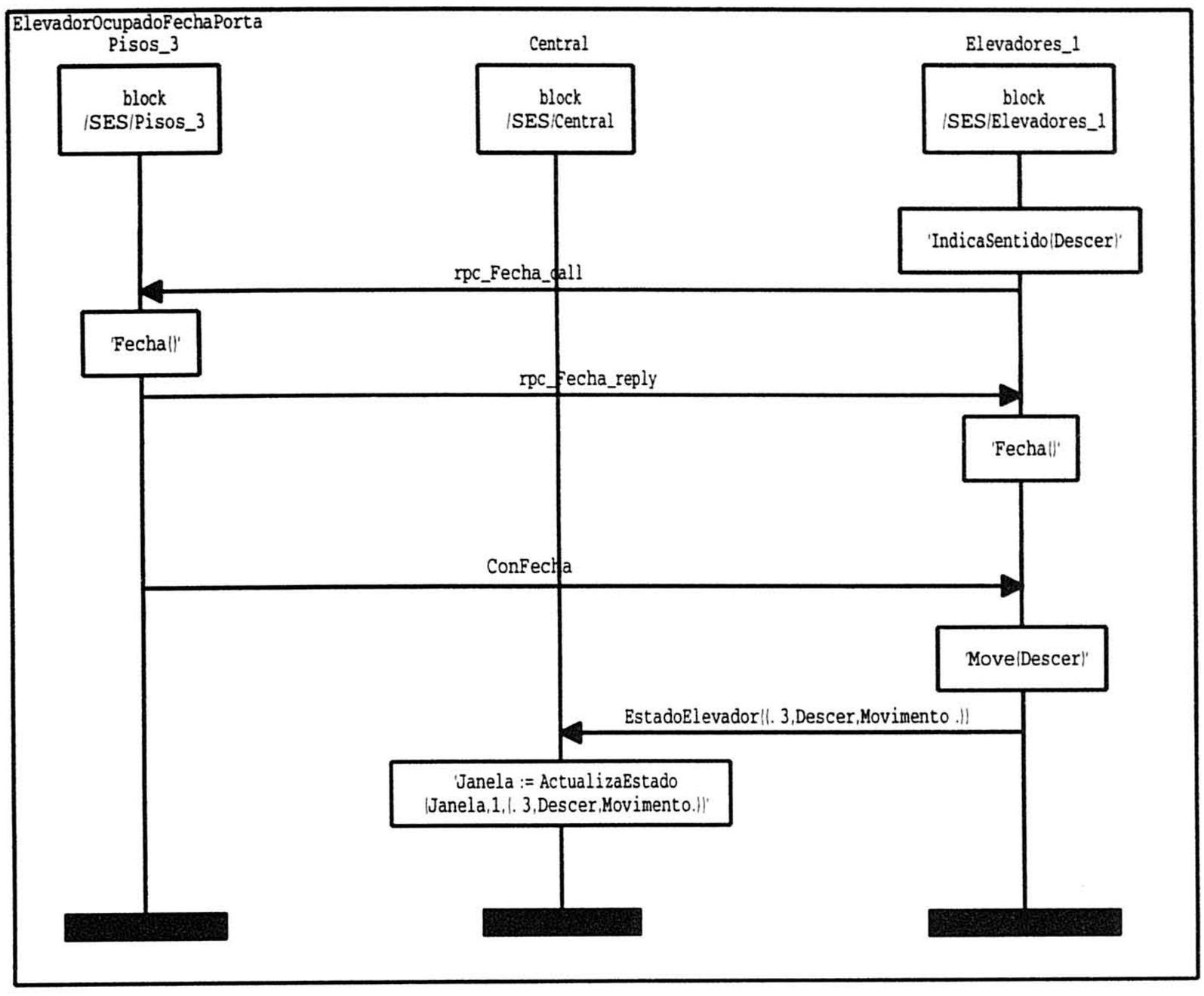

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha
rpc_Abre_call
'Abre()'
rpc_Abre_reply
NaoConfirma
Aberta
ConAbre

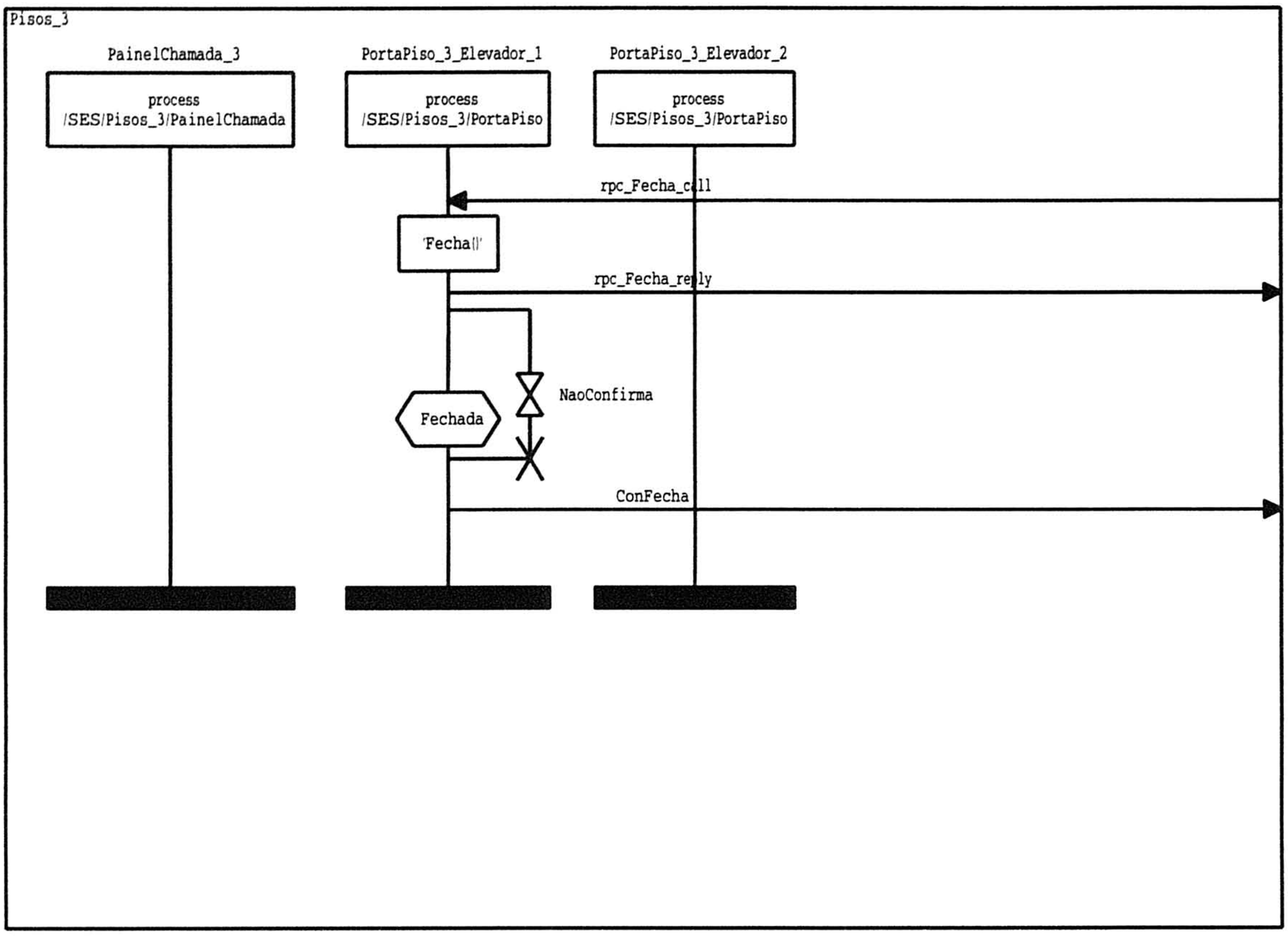

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
Premido
Fechar = Comando()
'ComandaLuz(Fechar,Acesa)'
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
'ComandaLuz(Fechar,Acesa)'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
Livre
Abre
rpc_Abre_call
rpc_Abre_reply
'Abre()'
NaoConfirma
Aberta
NaoConfirma
ConAbre
ConAbre
'EmiteSom(ElevadorParado)'
MomentoFechar

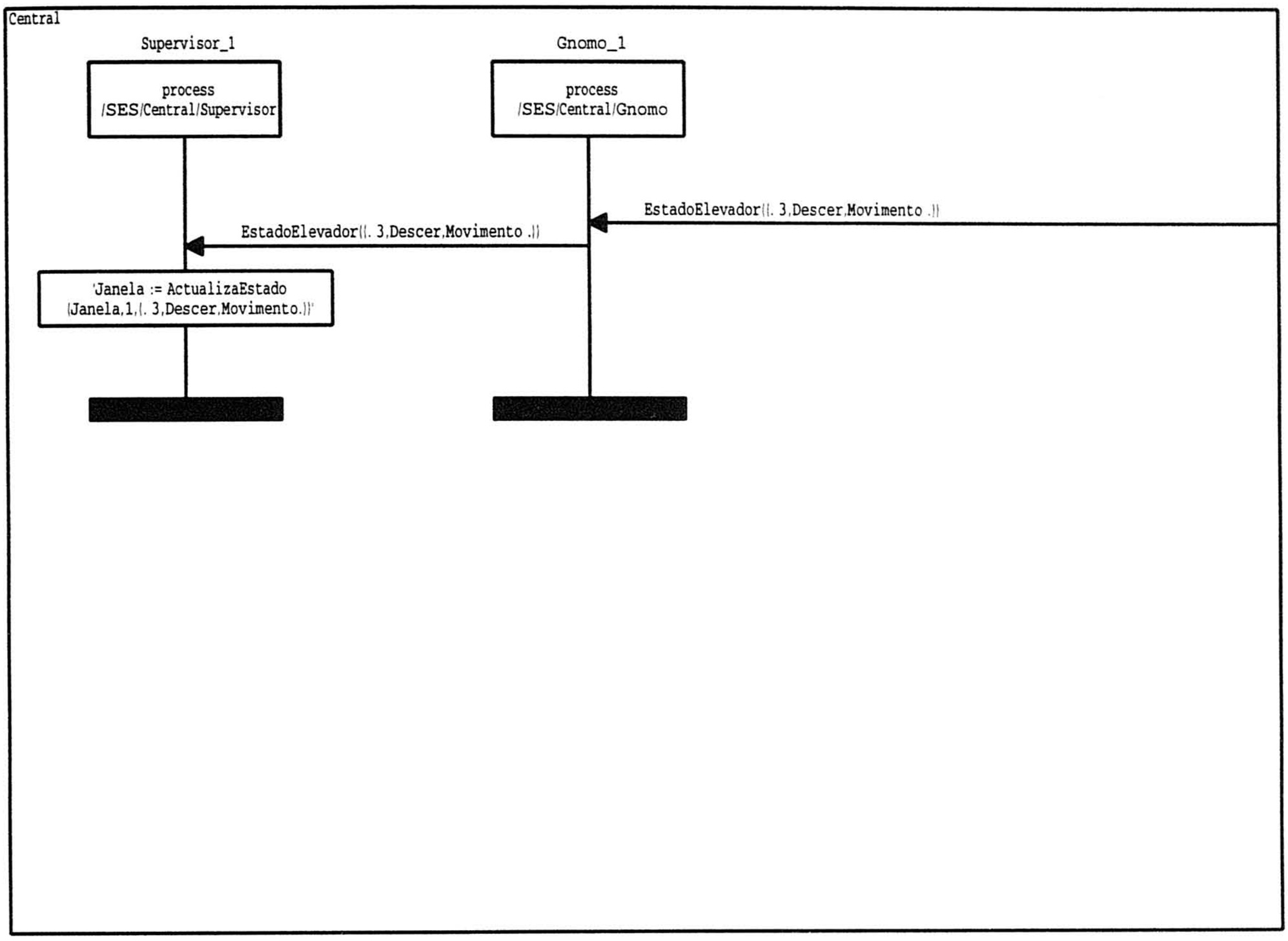

ElevadorLivre
Central
Elevadores_1
block
/SES/Central
block
/SES/Elevadores_1
'IndicaSentido(Nenhum)'
EstadoElevador({. 3,Nenhum,Livre .})
'Janela := ActualizaEstado
(Janela,1,{. 3,Nenhum,Livre.})'

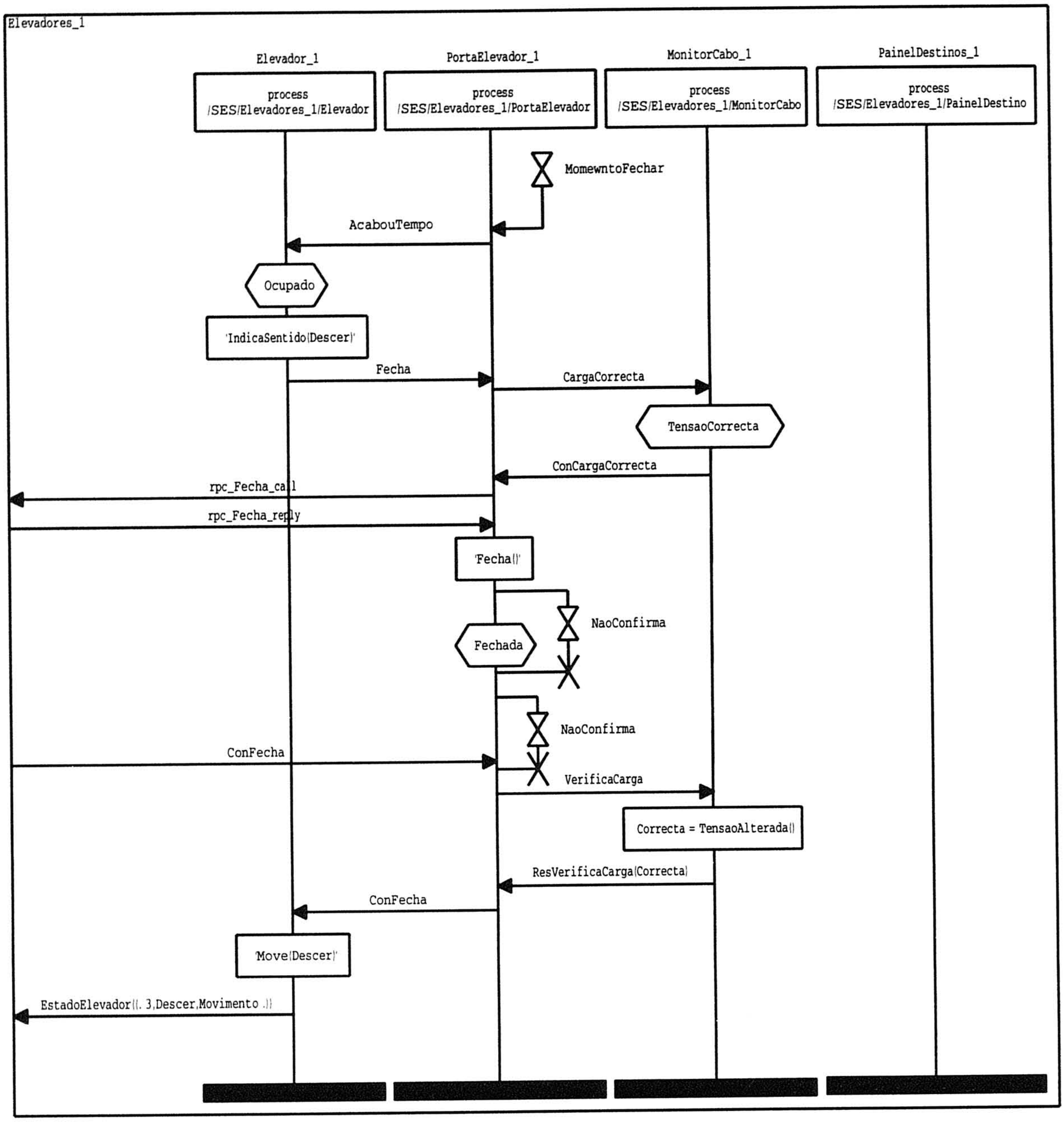

Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
EstadoElevador{|. 3,Nenhum,Livre .|}
EstadoElevador{|. 3,Nenhum,Livre .|}
'Janela := ActualizaEstado
(Janela,1,|. 3,Nenhum,Livre.|}'

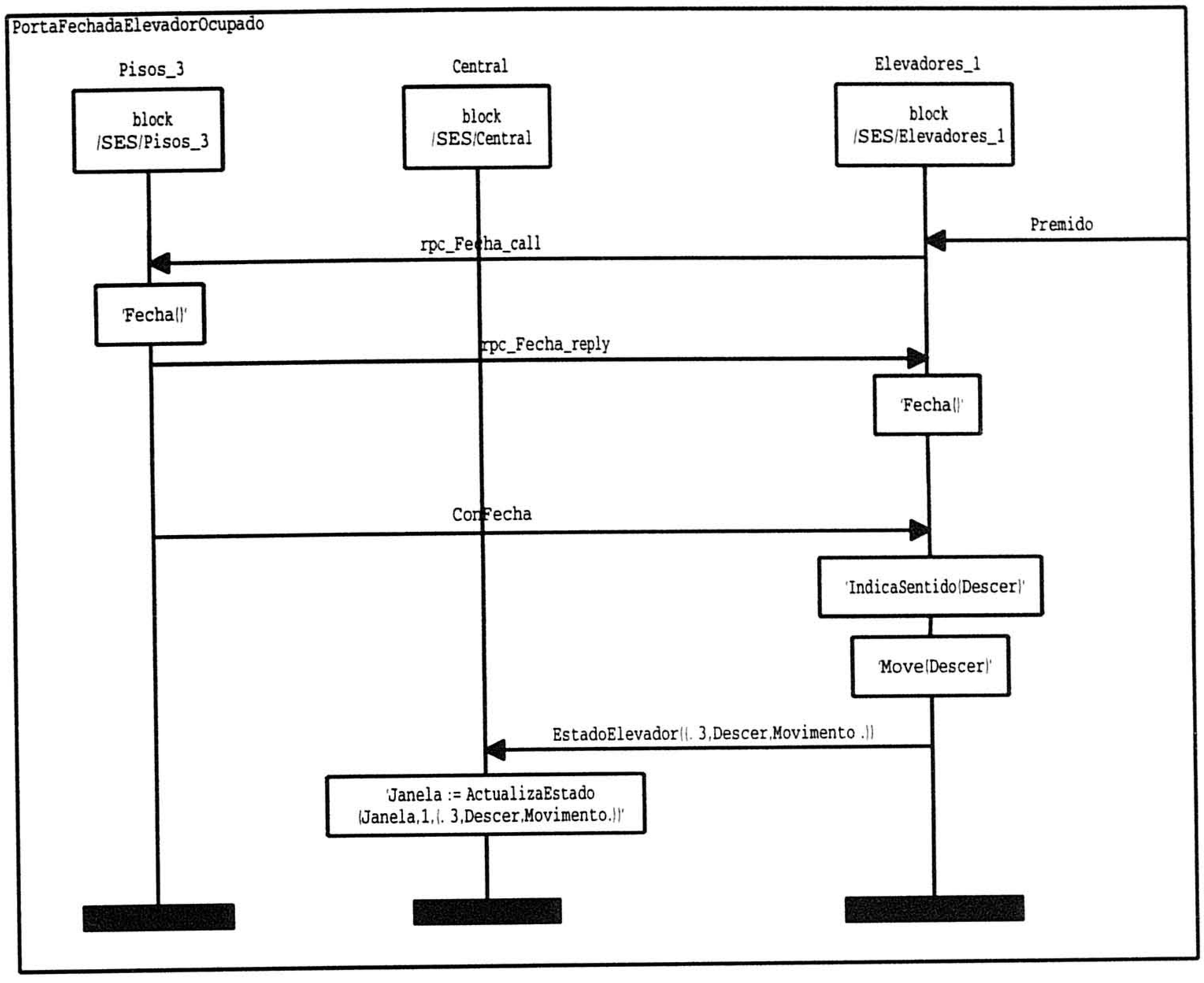

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
AcabouTempo
Livre
'IndicaSentido(Nenhum)'
EstadoElevador((. 3,Nenhum,Livre .))

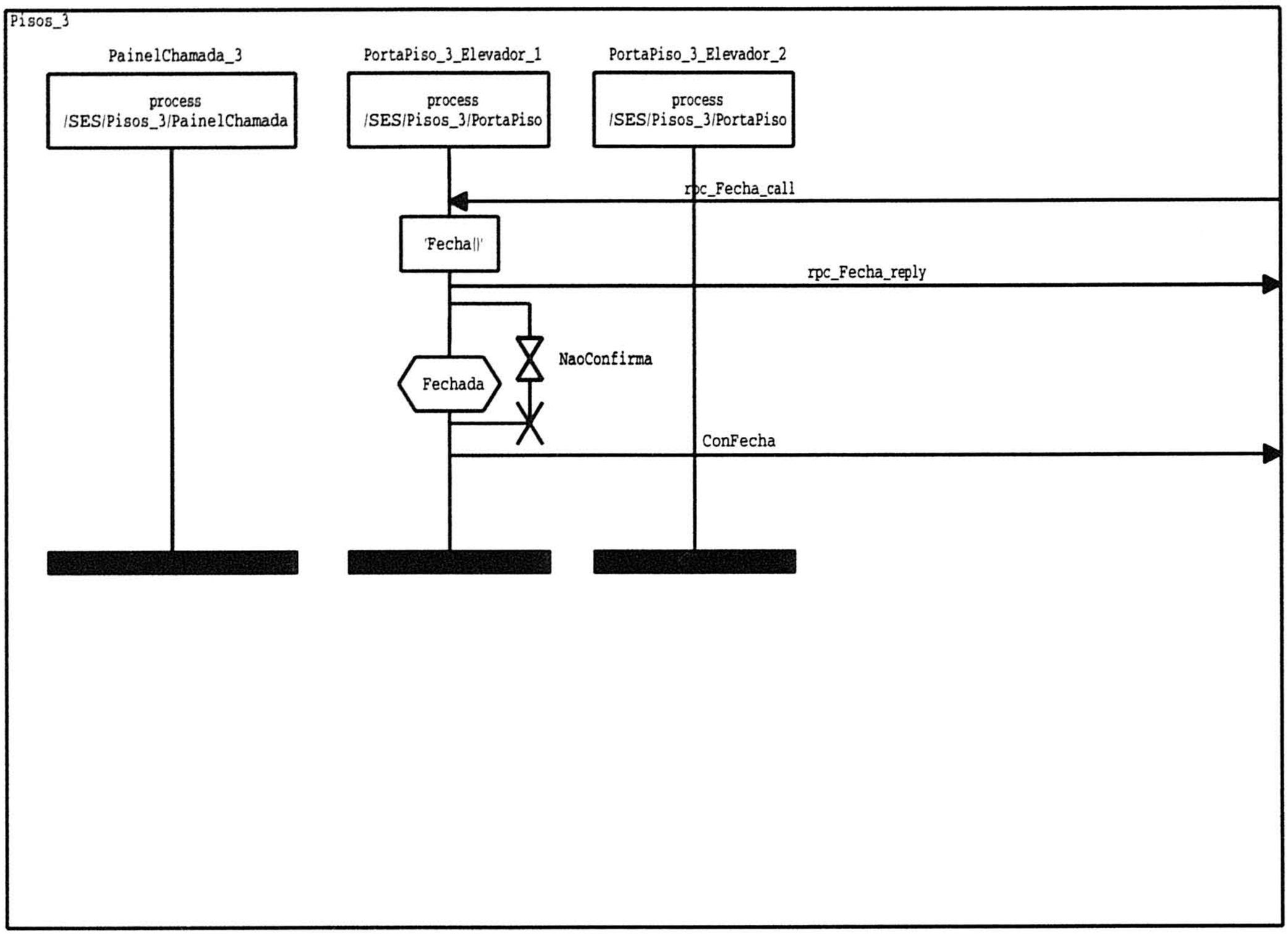

ElevadorOcupadoFechaPorta
Pisos_3
Central
Elevadores_1
block /SES/Pisos_3
block /SES/Central
block /SES/Elevadores_1
'IndicaSentido(Descer)'
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
'Move(Descer)'
EstadoElevador({. 3,Descer,Movimento .})
'Janela := ActualizaEstado (Janela,1,{. 3,Descer,Movimento.})'

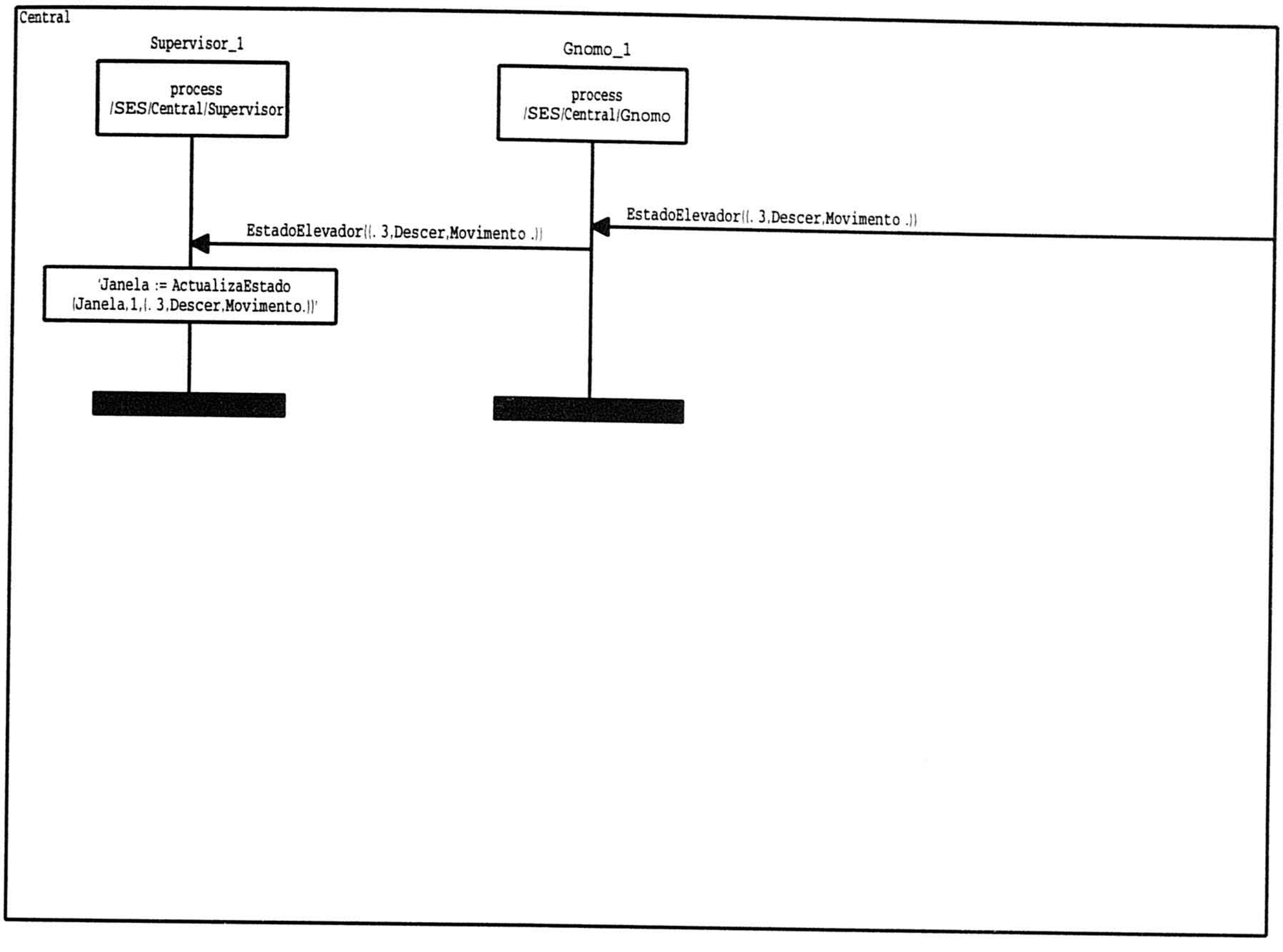

Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha


Central
Supervisor_1
Gnomo_1
process
/SES/Central/Supervisor
process
/SES/Central/Gnomo
EstadoElevador{|. 3,Descer,Movimento .|}
EstadoElevador{|. 3,Descer,Movimento .|}
'Janela := ActualizaEstado
(Janela,1,{. 3,Descer,Movimento.|}'

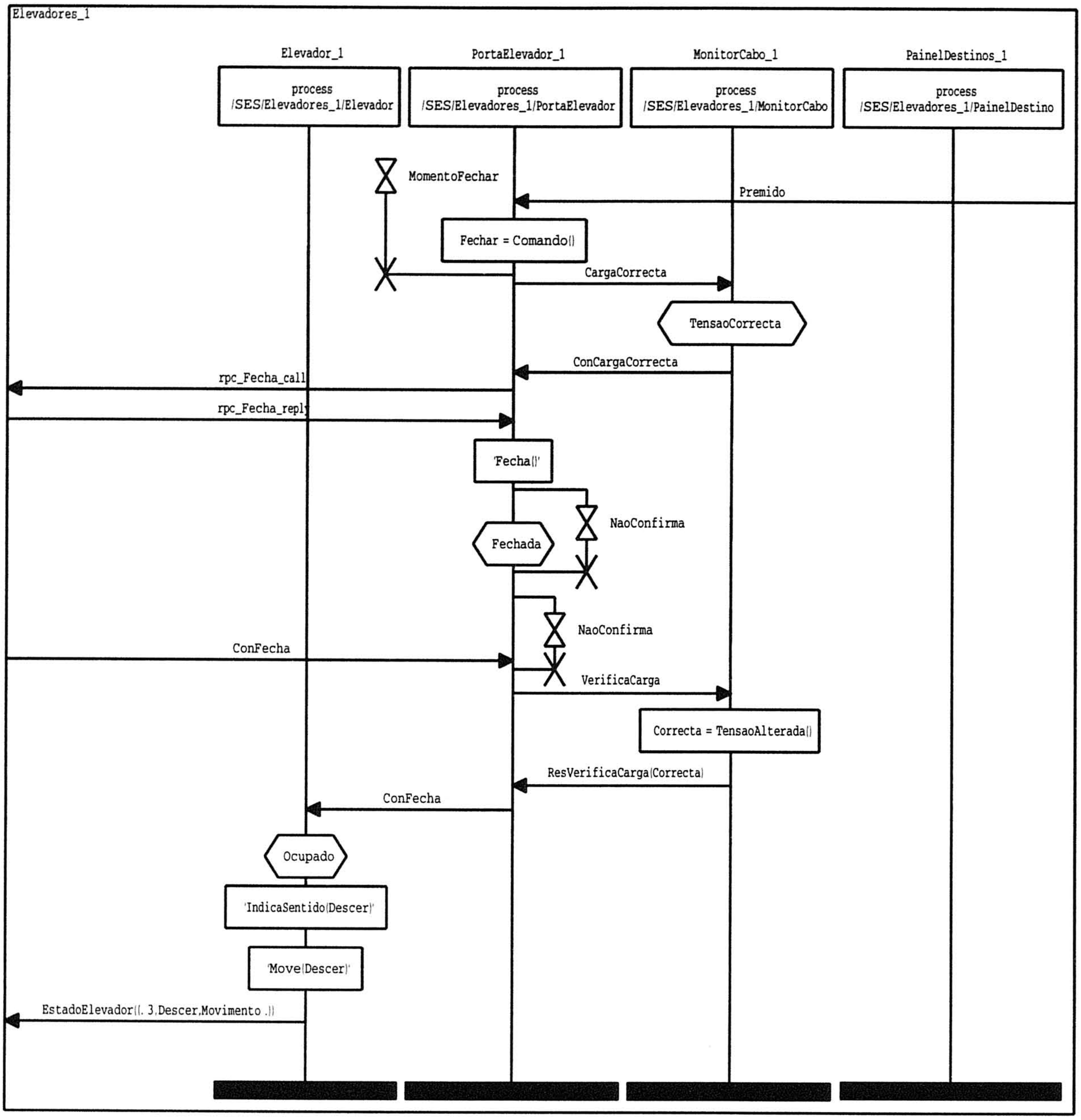

Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomewntoFechar
AcabouTempo
Ocupado
'IndicaSentido(Descer)'
Fecha
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
'Move(Descer)'
EstadoElevador(1, 3,Descer,Movimento ,1)


PortaFechadaElevadorOcupado
Pisos_3
Central
Elevadores_1
block /SES/Pisos_3
block /SES/Central
block /SES/Elevadores_1
Premido
rpc_Fecha_call
'Fecha()'
rpc_Fecha_reply
'Fecha()'
ConFecha
'IndicaSentido(Descer)'
'Move(Descer)'
EstadoElevador(3,Descer,Movimento)
'Janela := ActualizaEstado (Janela,1,(3,Descer,Movimento))'


Pisos_3
PainelChamada_3
PortaPiso_3_Elevador_1
PortaPiso_3_Elevador_2
process /SES/Pisos_3/PainelChamada
process /SES/Pisos_3/PortaPiso
process /SES/Pisos_3/PortaPiso
rpc_Fecha_call
'Fecha||'
rpc_Fecha_reply
NaoConfirma
Fechada
ConFecha



BIBLIOTECA F.E. PORTO


Elevadores_1
Elevador_1
PortaElevador_1
MonitorCabo_1
PainelDestinos_1
process /SES/Elevadores_1/Elevador
process /SES/Elevadores_1/PortaElevador
process /SES/Elevadores_1/MonitorCabo
process /SES/Elevadores_1/PainelDestino
MomentoFechar
Premido
Fechar = Comando()
CargaCorrecta
TensaoCorrecta
ConCargaCorrecta
rpc_Fecha_call
rpc_Fecha_reply
'Fecha()'
NaoConfirma
Fechada
NaoConfirma
ConFecha
VerificaCarga
Correcta = TensaoAlterada()
ResVerificaCarga(Correcta)
ConFecha
Ocupado
'IndicaSentido(Descer)'
'Move(Descer)'
EstadoElevador(. 3,Descer,Movimento .)

FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO
BIBLIOTECA
0000042450