Reengenharia de um sistema de controladores domóticos utilizando Redes de Petri

Por:

Mauro António Pereira dos Reis

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Doutor Luís Filipe dos Santos Gomes

Maio de 2011

Reengenharia de um sistema de controladores domóticos utilizando Redes de Petri

Resumo

Nesta dissertação apresenta-se todo um processo de reengenharia de um sistema de controladores domóticos designado por Tiny Domots.

Partindo da análise do sistema já existente, encontraram-se os modelos comportamentais associados utilizando como formalismo de modelação primário os Diagramas de Estados.

Criaram-se regras de tradução manual entre os Diagramas de Estados e Redes de Petri (RdP), sendo que para a edição das Redes de Petri foi utilizada a ferramenta Snoopy IOPT que permite aplicar ao seu resultado, ferramentas de geração automática de código. Neste caso foi utilizada a ferramenta de geração automática de código C (PNML2C), sendo este o tipo de código suportado pela plataforma utilizada (PIC18F4620 da Microchip).

Desta forma obteve-se um sistema de controladores domóticos com as mesmas funcionalidades do sistema existente, no entanto, criado com auxílio de ferramentas de edição de Redes de Petri e geração automática de código. É feita ainda uma análise comparativa entre os dois tipos de código (manual existente – gerado automaticamente).

O modelo encontrado, permite a fácil adição de novas funcionalidades ao sistema actual, servindo este de base a novas gerações de controladores domóticos Tiny Domots.

Através das várias fases deste processo de “reverse engineering” obteve-se uma definição e validação de metodologia que pode ser extensível a outras áreas que recorram a sistemas embutidos, não se limitando apenas à área da Domótica.

Palavras-chave: Domótica, Tiny Domot, Redes de Petri, Diagramas de Estados.

Abstract

This dissertation presents a whole reengineering process of a home automation system controller named Tiny Domots.

Based on the analysis of an existing system, we found the behavioural models associated using the State Diagrams as primary modelling formalism.

Were created rules for manual translation between the State Diagrams and Petri Nets (PN), and for editing of Petri Nets was used the tool Snoopy IOPT which allows to apply its result, tools for automatic code generation. In this case was used the tool for automatic generate C code (PNML2C), therefore this is the kind of code supported by the used platform (PIC18F4620 by Microchip).

Thus, were obtained a system of home automation controllers with the same features of the existing system, however, created with aid of editing tools and automatic code generation. It is also made a comparative analysis between the two types of code (manual created – automatically created).

The model found, allows the easy addition of new features to the current system, serving as basis for the new generations of home automation controllers Tiny Domots.

Through the various phases of this “reverse engineering” process was obtained a definition and validation of methodology that can be extended to other areas which use embedded systems, not limited to the Home Automation area.

Keywords: Home Automation, Tiny Domot, Petri Nets, State Diagrams.

Índice de Matérias

Índice de Figuras

Índice de Quadros

Simbologia e Notações

=	*Igual*
$\neq$	*Diferente*
<	*Menor que*
>	*Maior que*
$\leq$	*Menor ou igual que*
$\geq$	*Maior ou igual que*
$\forall$	*Quantificado universal*
$\subseteq$	*Subconjunto*
N	*Números naturais*
$\cap$	*Intersecção de conjunto*
$\cup$	*União de conjuntos*
$\varnothing$	*Conjunto vazio*
$A \times B$	*Produto Cartesiano*
$f : A \rightarrow B$	*Aplicação f de A em B. Função f definida em A e com valores em B*
ACK	*Acknowledge*
BD	*Base de dados*
CRC	*Cyclic redundancy check*
E/S	*Entrada/Saída*
FORDESIGN	*Formal Methods for Embedded Systems Co-Design*
IOPT	*Input-Output Place-Transition*
I/O	*Input/Output*
LED	*Light Emitting Diode*
PDC	*Platform Dependent Code*
PIC	*Programmable Intelligent Computer*
PNML	*Petri Net Markup Language*
RAM	*Random Acess Memory*
RdP	*Rede de Petri*
ROM	*Read Only Memory*
TD	*Tiny Domot*
UART	*Universal Asynchronous Receiver/Transmiter*
UML	*Unified Modeling Language*
VHDL	*VHDL (VHSIC hardware description language)*

Introdução

Neste capítulo pretende-se apresentar a motivação que levou à realização desta dissertação, tal como os seus objectivos, metodologia utilizada e a sua estrutura.

Motivação

O conceito de Edifício Inteligente apareceu na década de 80 associado sobretudo ao sector dos serviços. A principal motivação era a de realizar economias na gestão da energia e de fornecer novas facilidades aos seus utilizadores, principalmente nas áreas do conforto, da segurança e das comunicações.

Cedo se constatou que havia lugar para a aplicação dessas mesmas ideias à habitação. Isso ocorreu com particular relevância em países como os EUA, o Japão e a França. Em Portugal, o conceito passou a ser designado por Domótica sob influência do termo francês *Domotique*. Em termos da língua inglesa, designações comuns são *Smart House* e *Intelligent House*, sendo esta última a mais usada nos nossos dias.

Funções típicas tais como controlo de luminosidade, persianas, fugas de gás, inundação, intrusão, electrodomésticos fazem parte da domótica habitacional. Esta está normalmente associada às habitações de luxo, devido principalmente ao custo elevado das soluções existentes no mercado. No entanto, pode oferecer também meios poderosos de auxílio e suporte a pessoas com necessidades especiais com é o caso das pessoas com deficiências e pessoas idosas.

Na opinião do autor, a fraca divulgação da domótica deve-se, sobretudo, ao seu elevado custo face ao grau de utilidade/benefício oferecido e, por outro lado, a aspectos relacionados com dificuldades de instalação e de utilização. Para além disto, existem também razões tecnológicas que dificultam a expansão da domótica, de que se destaca a grande diversidade de soluções existentes, normalmente propriedade dos respectivos fabricantes, as quais são incompatíveis entre si.

Nesta dissertação, têm-se como objectivos adicionais a reengenharia de um sistema que seja simples de implementar e de baixo custo, mas ao mesmo tempo, que ofereça uma grande flexibilidade e capacidade de expansão.

O constante aumento da complexidade dos sistemas faz com que as exigências ao nível da sua modelação sejam cada vez maiores. Possuir um modelo que descreva adequadamente o sistema é uma mais-valia quer para o seu desenvolvimento, quer para a sua documentação. Isto porque, contribui para a redução de custos de desenvolvimento bem como os custos de

manutenção dos produtos. Desta forma, a criação de um modelo passa a ser crucial para o desenvolvimento de sistemas.

O principal formalismo a utilizar nesta dissertação para realização do modelo do sistema é o de Redes de Petri (RdP). Tirando partido de algumas das suas características únicas, como por exemplo a simplicidade de compreensão, associada à flexibilidade da sua representação gráfica, torna-se possível a sua reutilização, no todo ou em parte, no desenvolvimento de novos sistemas semelhantes assim como na criação de novas funcionalidades ou alteração de outras já existentes. Desta forma, possibilita o ajuste do sistema a uma determinada solução ou permite aumentar a sua área de abrangência. Por exemplo, os modelos encontrados poderão ainda possibilitar a portabilidade do sistema para outras plataformas que não a utilizada.

O autor considera que esta dissertação poderá constituir uma mais-valia, não só na criação de uma nova geração de controladores domóticos mas também na definição e validação de uma metodologia que pode ser extensível a outras áreas que não a domótica.

Objectivos

Pretende-se analisar o sistema de controladores domóticos distribuídos (Tiny Domots) desenvolvido no âmbito de trabalhos académicos anteriores, de forma a caracterizar as funcionalidades e modelos comportamentais associados (expressos através de diagramas de estados e de redes de Petri), no sentido de permitir adicionar novas funcionalidades e alterar outras funcionalidades existentes.

Como resultado das tarefas de reengenharia (“reverse engineering” seguido de adição e alteração de novas funcionalidades) e da obtenção de modelos comportamentais para o sistema pretendido, utilizaram-se ferramentas de edição assim como de geração automática de código, resultado de dissertações de Mestrado anteriores desenvolvidas no âmbito do projecto Fordesign [Fordesign 07]. Com isto pretende-se a obtenção de protótipos da próxima geração de Tiny Domots.

Como objectivos secundários, temos ainda a validação das ferramentas mencionadas assim como a comparação entre dois tipos de código: código existente, gerado manualmente, e nova versão, com as mesmas funcionalidades, mas gerado com o auxílio de ferramentas de geração automática.

Há que referir que a única componente do equipamento que se pretende desenvolver no âmbito desta dissertação é a componente de controlo do Tiny Domot, excluindo-se assim a parte de hardware.

Metodologia

Optou-se por dividir o trabalho em três principais fases. A primeira fase consiste na análise do sistema existente (criado de forma tradicional/manual), a segunda fase passa pela criação de um novo sistema com as mesmas funcionalidades, utilizando uma determinada metodologia e recorrendo à utilização de ferramentas de edição assim como de geração automática de código. Por último temos a fase a adição de novas funcionalidades ao sistema criado na fase anterior.

O conjunto das duas primeiras fases trata-se de um processo de “reverse engineering”, ou seja, em vez de se começar por criar os modelos e a partir dos modelos criar o código final, começa-se por analisar o código já existente, com base nesse código é criado de forma manual um Diagrama de Estados do comportamento do sistema. Em paralelo são recolhidas algumas funções específicas existentes no código fornecido, com vista a estas serem reutilizadas no novo sistema modelado, constituindo estas a base operacional do sistema.

Em seguida, o Diagrama de Estados é traduzido manualmente para uma Rede de Petri, segundo um método encontrado. Após este ponto são aplicadas ferramentas de geração automática de código para a obtenção do código referente ao modelo comportamental do sistema.

A geração automática de código tem início com a utilização de uma ferramenta desenvolvida no âmbito do projecto FORDESIGN [Fordesign, 07], o Snoopy IOPT [Nunes, 08]. O Snoopy IOPT é um editor gráfico de RdP, que gera ficheiros PNML. É com esse ficheiro PNML e com outra ferramenta desenvolvida no âmbito do mesmo projecto, a ferramenta PNML2C [Rebelo, 2010], que é gerado o novo código C.

Esse código C gerado corresponde ao modelo comportamental do sistema, sendo ainda necessário incluir as funções específicas, extraídas do código existente. Após a junção destes dois códigos obtêm-se o código C final do sistema.

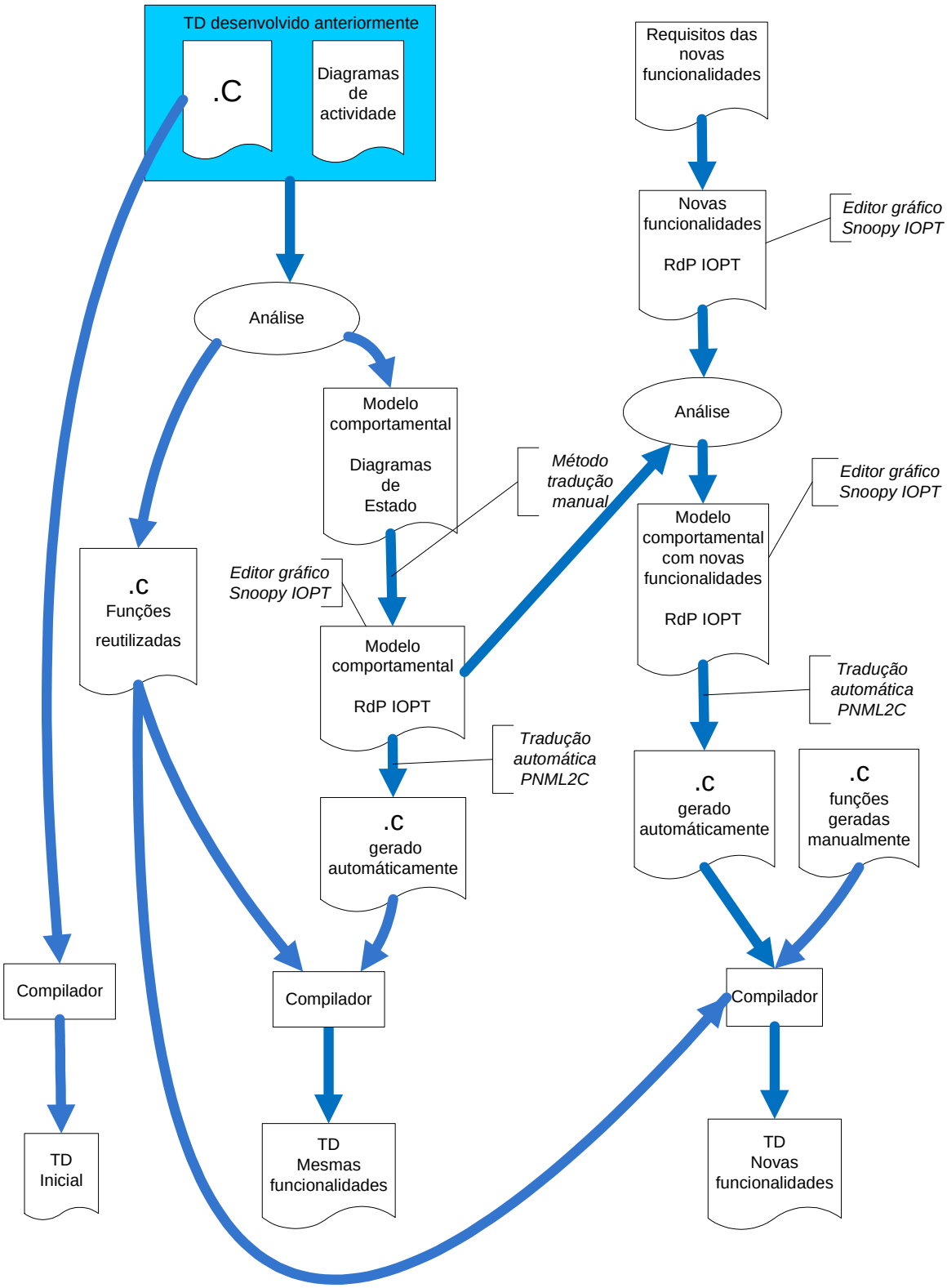


Figura 0.1 – Representação da Metodologia

O modelo encontrado serve de base à implementação de novas funcionalidades com vista à criação do modelo da nova versão de Tiny Domots. Para tal são introduzidos no modelo todos os requisitos específicos da nova versão que se pretende desenvolver. Essa junção dá origem a um modelo comportamental completo do equipamento pretendido. Posteriormente o modelo é automaticamente convertido em código C e são criadas as funções necessárias, dando origem a um novo código C para o novo equipamento.

Dessa forma, sempre que se pretender desenvolver um novo equipamento será apenas necessário juntar ao modelo genérico em RdP os requisitos específicos para o novo equipamento e voltar a gerar o código C.

A Figura 0.1 representa as várias fases do processo descrito.

Estrutura da dissertação

Esta dissertação foi estruturada em seis capítulos distintos, incluindo a introdução. É apresentada de seguida uma breve descrição de cada um deles.

A dissertação começa por apresentar os formalismos de modelação utilizados para o desenvolvimento do modelo comportamental dos controladores domóticos Tiny Domots, quer na versão existente, quer para a nova versão. Assim, este capítulo apresenta vários formalismos de modelação divididos em dois subcapítulos (UML e Redes de Petri), decompostos em: Casos de Uso, Diagramas de Actividade, Diagramas de Estados e Redes de Petri. No subcapítulo dedicado às redes de Petri, além de abordados os temas gerais relacionados com este formalismo é ainda abordada a classe de Redes de Petri utilizada para a modelação do sistema, as Redes de Petri IOPT. No subcapítulo seguinte é apresentada uma forma de representação dessa mesma classe de Redes de Petri, a representação em PNML.

Depois de estudados os vários formalismos utilizados na modelação do sistema, é necessário expor o trabalho realizado anteriormente através, principalmente, da descrição do sistema e das suas funcionalidades. Desta forma são identificadas as necessidades e as características que o equipamento terá igualmente de ter após aplicado o processo de “reverse engineering”. Este segundo capítulo encontra-se dividido em quatro subcapítulos. No primeiro apresenta-se uma descrição do sistema. No segundo subcapítulo são identificadas as funcionalidades do sistema sendo as mesmas expressas em Casos de Usos. No terceiro subcapítulo é apresentado o modelo comportamental do sistema expresso através de Diagramas de Actividade e respectiva análise. Por último, no quarto subcapítulo é feita uma breve discrição da aplicação criada para interagir com o sistema.

Após exposição e análise do trabalho realizado anteriormente, são apresentadas no capítulo três, as contribuições desta dissertação. Começa-se por apresentar o modelo

comportamental expresso em Diagramas de Estados obtido através da análise do código C existente, assim como são indicados os comentários às decisões e melhoramentos encontrados aquando dessa análise. No subcapítulo seguinte é apresentado o modelo comportamental obtido através da tradução dos Diagramas de Estados em Redes de Petri IOPT, para tal são indicadas as respectivas regras utilizadas para a tradução. No mesmo subcapítulo é ainda efectuada a descrição do modelo comportamental encontrado, ou seja, é explicada de forma pormenorizada a Rede de Petri criada para constituir o modelo comportamental.

De seguida é apresentado e analisado o código obtido através da ferramenta de geração automática de código C a partir da Rede de Petri encontrada. Logo depois é identificado o código manual utilizado. Para terminar este capítulo é feita uma análise comparativa entre o código manual (versão existente) e o código da versão criada com recurso a ferramentas de geração automática.

Após encontrado e analisado o modelo final com as mesmas características do sistema existente, passa-se ao desenvolvimento de uma nova versão do sistema de controladores domóticos. Neste capítulo começa-se por descrever as novas funcionalidades pretendidas para associar ao sistema. Depois disto são apresentados os modelos das novas funcionalidades e a forma como estes se podem associar ao modelo encontrado, descrito no capítulo anterior. Após a geração de código de forma automática e das funções criadas manualmente, necessárias à implementação das novas funcionalidades, é feita uma análise ao resultado obtido.

No último capítulo, procede-se à apresentação das conclusões da dissertação.

1. Formalismos de Modelação

Neste capítulo serão apresentados alguns formalismos de modelação importantes para o desenvolvimento desta dissertação.

1.1. UML

A *Unified Modeling Language* (UML) é uma linguagem padrão para modelação orientada a objectos. Esta é essencialmente gráfica e permite modelar, especificar e documentar elementos de sistemas.

Esta linguagem de modelação não proprietária tem como papel auxiliar a visualização do desenho e a comunicação entre objectos. Ela permite que quem desenvolve, visualize o produto do seu trabalho em diagramas padronizados, e é muito usada para criar modelos de sistemas de software.

Além de fornecer a tecnologia necessária para apoiar a prática de engenharia de software orientada a objectos, a UML poderá ser a linguagem de modelação padrão para modelar sistemas concorrentes e distribuídos.

Utiliza um conjunto de técnicas de notação gráfica para criar modelos visuais de sistemas de software intensivo, combinando as melhores técnicas de modelação de dados, negócios, objectos e componentes. É uma linguagem de modelação única, comum e amplamente utilizável.

A UML possui diagramas (representações gráficas do modelo de um sistema) que são usados em combinação, com a finalidade de obter todas as visões e aspectos do sistema.

A UML define os seguintes diagramas gráficos:

- Perspectiva do utilizador
    - Diagrama de Casos de Uso
- Estrutura
    - Diagrama de Classes
    - Diagrama de Objectos
- Comportamento
    - Diagrama de Sequência
    - Diagrama de Colaboração
    - Diagrama de Estados

    - Diagrama de Actividade
- Implementação
    - Diagrama de Componentes
    - Diagramas de Desenvolvimento

Em seguida abordam-se as técnicas de modelação que foram utilizadas neste trabalho, nomeadamente os Diagramas de Casos de Uso na identificação dos diversos cenários, e os Diagramas de Actividade existentes (projecto anterior) assim como os Diagramas de Estados actuais, ambos na modelação comportamental do sistema.

1.1.1. Casos de Uso

Os casos de uso permitem uma visão alargada da funcionalidade primária do sistema expressa de uma forma simples, facilitando a sua interpretação. Os diagramas de casos de uso podem tornar-se num autêntico mapa centralizado dos cenários de utilização dos sistemas, especificando os seus requisitos. Os próprios casos de uso podem ser decompostos em outros casos de uso e finalmente em cenários que mostrem sequências detalhadas das interacções dos objectos.

Uma vez que têm um nível conceptual bastante alto e utilizam vocabulário do domínio do problema, diversas pessoas encontram nos diagramas de casos de uso a ferramenta mais importante para estabelecer a comunicação com os utilizadores [Douglass, 98].

Aos casos de uso associam-se actores. Isto significa que o caso de uso pode receber mensagens de actores e enviar mensagens para eles. Os actores podem ser dispositivos físicos ou utilizadores que interagem com o sistema [Douglass, 99].

O analista deverá identificar os diversos cenários que mapeiam os diferentes aspectos do sistema e deduzir os respectivos casos de uso. Para identificar os cenários, é possível um variado número de aproximações.

Cada um dos casos de uso identificados deve ser detalhado ou descrito em termos de cenários de utilização. Esses cenários são os possíveis caminhos seguidos dentro do caso de uso, de forma a fornecer uma resposta. Esta descrição pode assumir a forma de descrição estruturada ou texto livre segundo um conjunto de passos numerados, ficando esta decisão ao critério do analista.

Também se introduz os conceitos de pré-condição e pós-condição que indicam, respectivamente, o estado inicial e final do sistema aquando da realização do caso de uso. A pré-condição indica o que deve existir inicialmente para que o cenário descrito seja seguido com sucesso. No caso de pós-condição é demonstrado o que irá acontecer depois do cenário ser concluído.

O número de cenários pode crescer bastante. Todos os cenários que invoquem respostas idênticas de sistemas a cada oportunidade, constituem um cenário único. Os cenários principais elaboram os principais caminhos que os casos de uso realizam na prática. Os cenários secundários são variantes de cenários primários para explorar excepções, aspectos de segurança e outras questões importantes.

Os casos de uso são representados como ovais com linhas sólidas, enquanto que o ícone para o actor é uma figura. As associações são representadas como linhas. Os casos de uso podem ter relações com outros casos de uso. O UML usa a generalização para isso, o que significa que um caso de uso é uma forma mais geral de outro. Dois estereótipos de caso de uso são definidos:

- “extends” significa que um caso de uso é uma versão mais expandida de outro;
- “include” significa que um caso de uso utiliza as funcionalidades fornecidas por outro.

Na Figura 1.1 é apresentado o exemplo de diagrama de casos de uso de um sistema onde constam os actores, casos de uso e a relação entre eles.

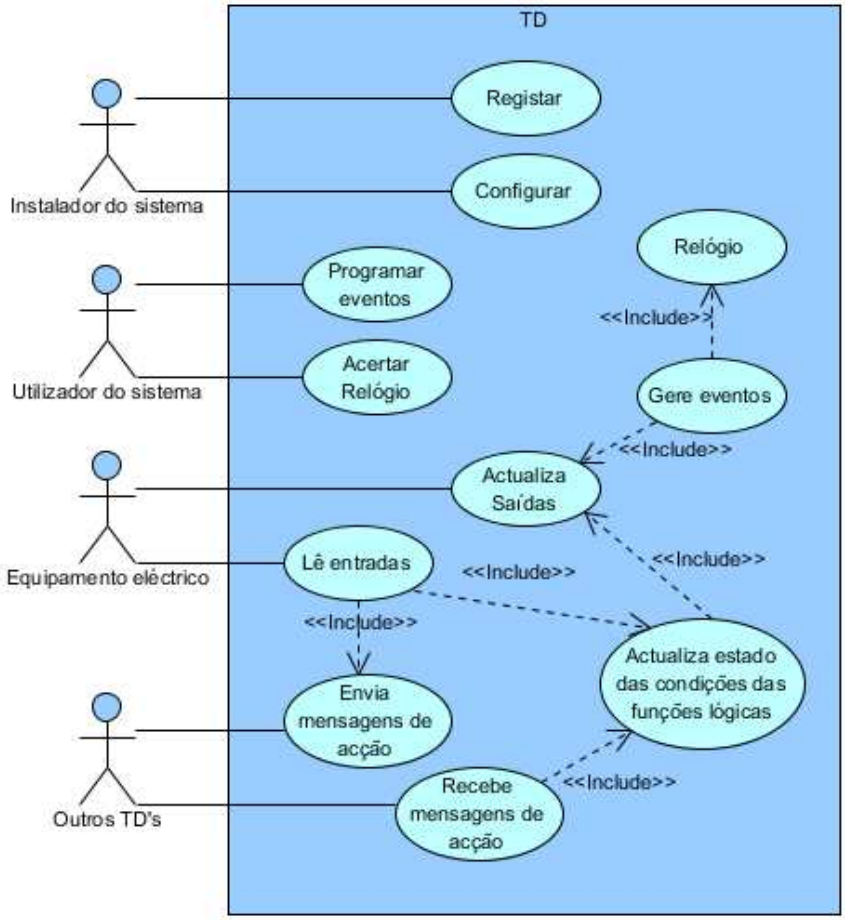


Figura 1.1 - Exemplo de Diagrama de Casos de Uso, adaptado de [Moutinho, 04]

1.1.2. Diagramas de Actividade

Os diagramas de actividade mostram o fluxo sequencial das actividades, são normalmente utilizados para demonstrar as actividades executadas por uma operação específica do sistema. Consistem em estados de acção, que contém a especificação de uma actividade a ser desempenhada por uma operação do sistema. Decisões e condições, como execução paralela, também podem ser mostrados no diagrama de actividade.

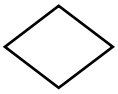
Diagramas de actividade especificam as acções e os seus resultados. Eles focam o trabalho executado na implementação de uma operação, e as suas actividades numa instância de um objecto.

Componentes do Diagrama de Actividade

Estado Inicial: Marca o início da Actividade

Estado Final: Marca o fim da actividade

Estado de acção: estado de execução de uma acção, cuja conclusão determina a saída do estado. As transições de saída não têm eventos, mas podem ter condições e acções.



Decisão: estado de passagem em que são testadas condições. Essas condições aparecem nas transições de saída. Não é um estado verdadeiro, mas uma ramificação numa transição.

Ponto de sincronização: onde os subfluxos de actividade paralelos ou concorrentes se juntam ou se separam.

Transição: representada como uma seta, situa-se entre dois dos outros componentes, por exemplo, entre dois estados de acção.

Num diagrama de actividades, os estados são estados de acção ou subactividade e em que todas ou pelo menos a maioria das transições são disparadas pela conclusão das acções ou subactividades nos estados de origem. Os diagramas de actividades deverão ser utilizados em situações em que todos ou quase todos os eventos representam a conclusão das acções geradas internamente.

Na Figura 1.2 podemos observar um exemplo de um diagrama de actividades.

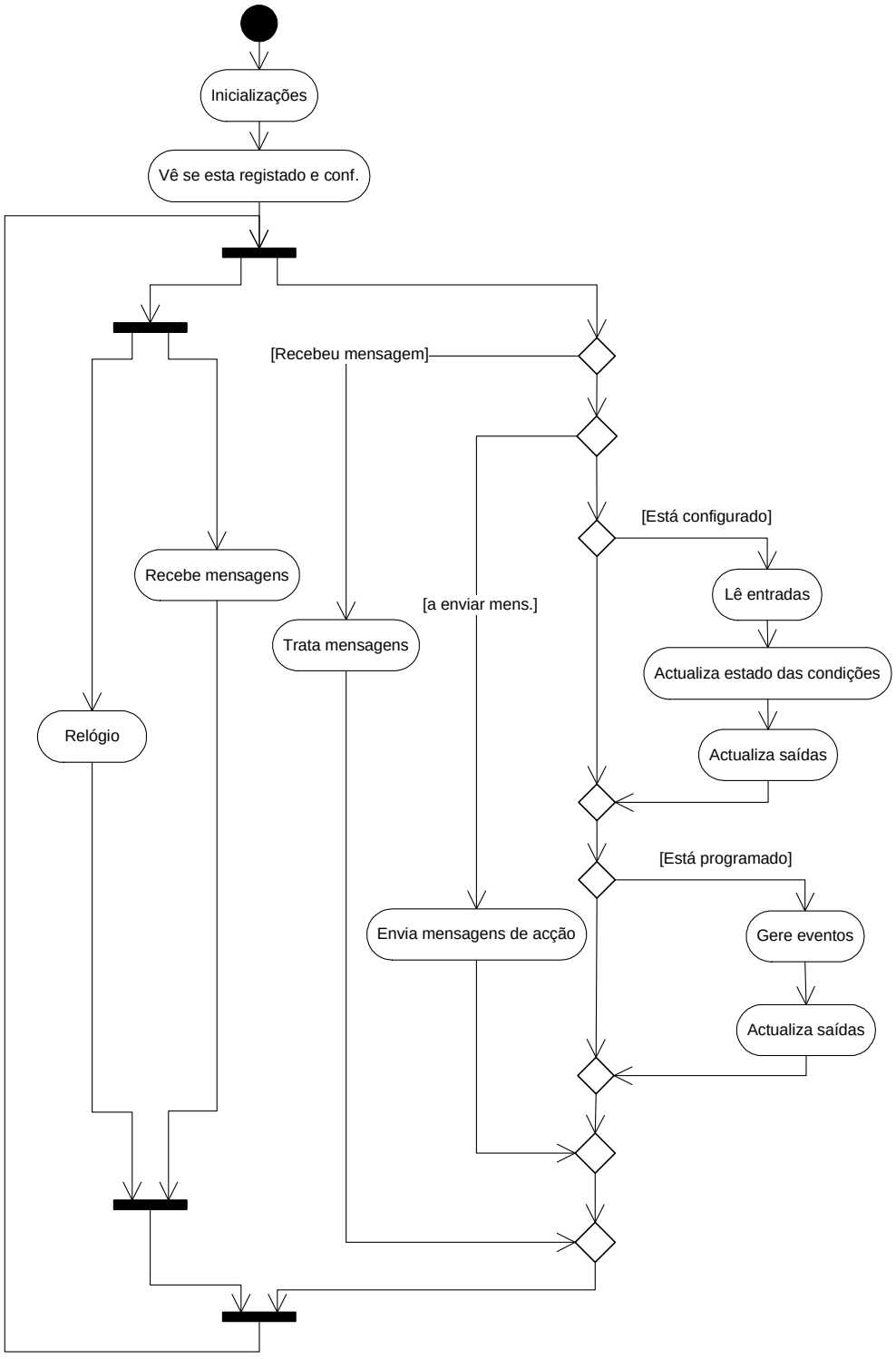


Figura 1.2 - Exemplo de Diagrama de Actividade. Adaptado de [Moutinho, 04]

1.1.3. Diagramas de Estados

Os diagramas de estados são usados para mostrar todos os estados possíveis de um objecto na execução de um processo, assim como os eventos do sistema que geram as diversas mudanças.

Os diagramas de estados não são definidos para todas as classes de um sistema, mas apenas para aquelas que possuem um número definido de estados conhecidos e onde o comportamento das classes é afectado e modificado pelos diferentes estados.

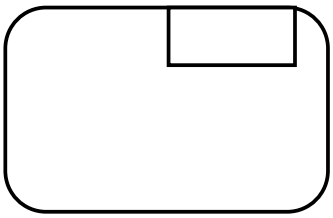
Eles mostram os estados que um objecto pode possuir e como os eventos (mensagens recebidas, erros e condições sendo satisfeitas) afectam estes estados na execução do processo.

Componentes do Diagrama de Estados

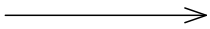
Estado inicial: representa a criação do objecto e início da máquina de estados.

Estado final: representa o fim da máquina de estados e destruição do objecto.

Estado: é mostrado como um rectângulo com cantos arredondados. Um estado é uma condição ou situação na vida de um objecto, durante a qual o objecto satisfaz alguma condição, realiza alguma actividade ou espera por algum evento. A ele pode estar associada uma actividade.



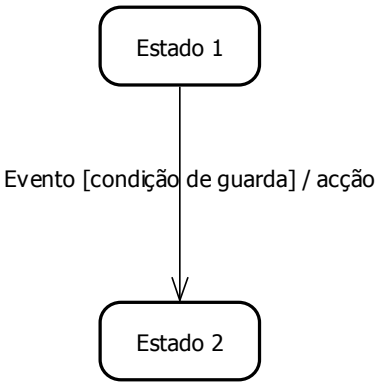
Super-estado: é a representação de um estado múltiplo, ou seja, um conjunto de estados.
Estes são utilizados para melhorar a organização do diagrama de estados e facilitar a sua compreensão. O super-estado é composto por dois ou mais sub-estados.



Transição: representada como uma seta, situa-se entre 2 estados. Uma transição de estado representa a mudança de um estado de origem para um estado destino (que pode ser o mesmo estado origem).

O rótulo de transição de estado tem três componentes, todos opcionais, no formato:

Evento [condição de guarda] / acção



A transição pode ser nomeada com o seu evento causador. Quando o evento acontece, a transição de um estado para outro é executada ou disparada.

A condição de guarda retorna verdadeiro ou falso. A transição é executada quando a condição de guarda retorna verdadeiro.

Acção é associada à transição entre estados, ou seja, é executada durante a transição, após ocorrer o evento associado e a condição de guarda ser verdadeira.

Quando nenhum dos 3 componentes (evento, condição de guarda ou acção) está associado à transição, esta ocorre quando a actividade associada ao estado é terminada.

Na Figura 1.3 podemos observar um exemplo de diagrama de estados onde constam transições com condições de guarda e transições simples.

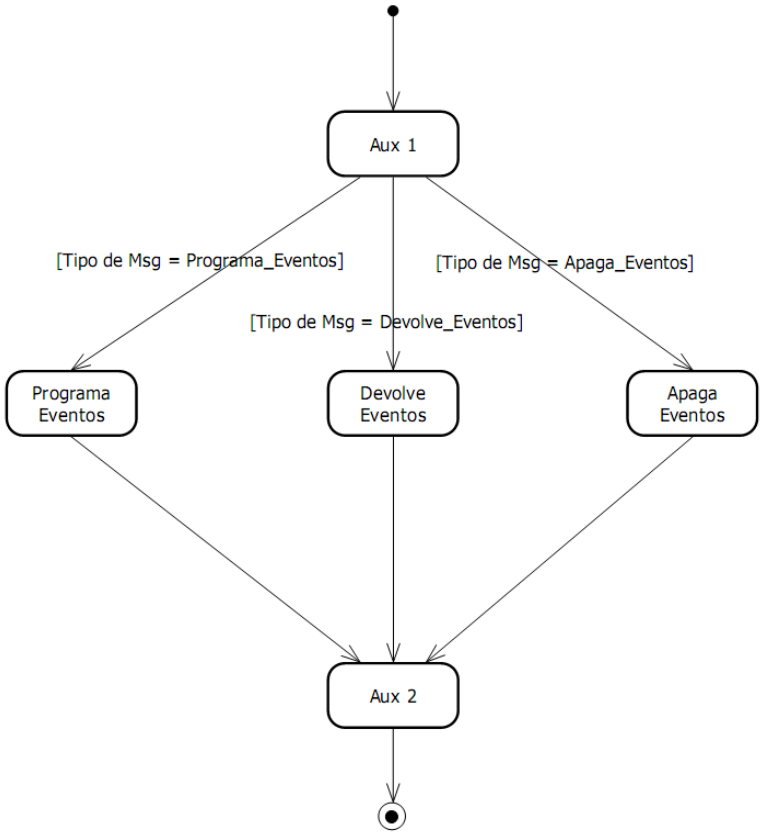


Figura 1.3 - Exemplo de Diagrama de Estados

1.2. Redes de Petri

As redes de Petri (RdP) devem o seu nome ao trabalho de Carl Adam Petri que na sua dissertação de doutoramento, submetida, em 1962, à Faculdade de Matemática e Física da Universidade Técnica de Darmstadt na Alemanha, apresentou um tipo de grafo bipartido com estados associados, com o objectivo de estudar a comunicação entre autómatos [Murata, 89].

O seu desenvolvimento foi continuado devido às suas numerosas potencialidades de modelação, designadamente: sincronização de processos, concorrência, conflitos e partilha de recursos. Até aos dias de hoje têm sido desenvolvidos trabalhos teóricos e aplicações sobre RdP tendo este estudo levado, quer a um desenvolvimento das técnicas de análise das RdP e sua aplicação prática, quer ao desenvolvimento de muitas variantes ao modelo seminal das RdP tendo em vista aplicações específicas.

Como ferramentas matemáticas e gráficas, as RdP oferecem um ambiente uniforme para a modelação, análise formal e simulação de sistemas a eventos discretos, permitindo uma visualização simultânea da sua estrutura e comportamento.

As RdP modelam dois aspectos desses sistemas, eventos e condições, bem como, as relações entre eles. Segundo esta caracterização, em cada estado do sistema verificam-se determinadas condições. Estas podem possibilitar a ocorrência de eventos que por sua vez podem ocasionar a mudança de estado do sistema [Peterson, 77].

1.2.1. Estrutura de uma Rede de Petri

Uma RdP é constituída por dois tipos de nós: *lugares e transições*. Os lugares são normalmente representados por circunferências ou elipses, e as transições por segmentos de recta, rectângulos ou barras. Os lugares encontram-se ligados às transições, e estas aos lugares, através de arcos dirigidos. [Barros, 96]

Na Figura 1.4 representa-se um lugar, uma transição e um arco dirigido.

Figura 1.4 - Representação de um lugar, de uma transição e de um arco dirigido

Na Figura 1.5 temos o exemplo de uma RdP.

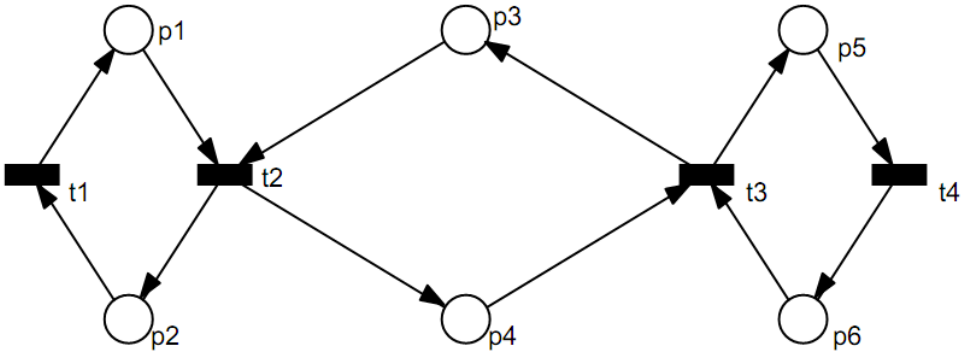


Figura 1.5 – Exemplo de uma rede de Petri

Uma atitude comum de modelação de sistemas, através de RdP, considera que os lugares podem ser vistos como depósitos de recursos e as transições como acções que manipulam esses recursos. Os recursos são representados graficamente por pontos negros dentro dos lugares. A cada um dos pontos negros dá-se o nome de marca [Lino, 03].

A função das transições consiste em destruir/criar as marcas. Uma transição está obrigatoriamente entre lugares, logo, a sua acção (denominada disparo) vai alterar a marcação de um lugar. Os arcos indicam, para cada transição, os lugares sobre os quais estas actuam. Podemos ver na Figura 1.6 um exemplo de uma rede marcada.

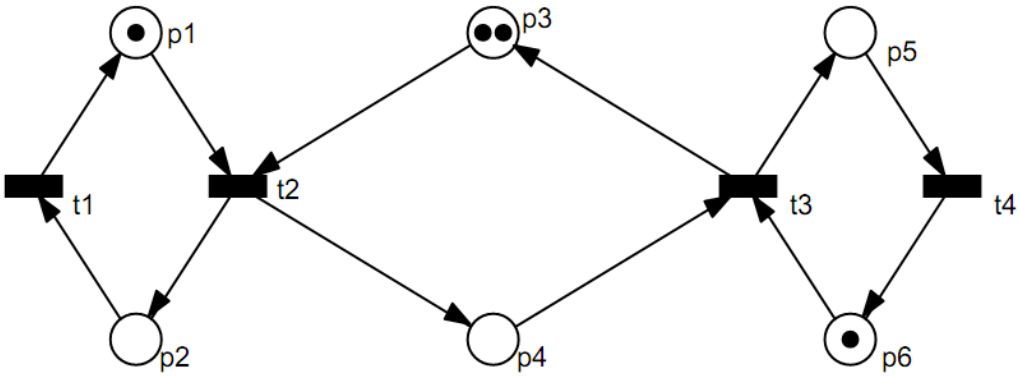


Figura 1.6 – Exemplo de uma rede de Petri marcada

1.2.2. Evolução da rede — disparo das transições

Designa-se por arco de entrada, um arco com origem num lugar e que termina numa transição indicando que essa transição subtrai, aquando do seu disparo, uma marca desse lugar. Da mesma forma, um arco de saída designa um arco com origem numa transição e fim num lugar indicando que essa transição adiciona, aquando do seu disparo, uma marca a esse lugar. Desta forma, podemos pensar nos arcos como indicando o sentido do movimento das marcas de um lugar para outro, atravessando a transição.

Uma transição só pode ser disparada se cada lugar de entrada possuir pelo menos uma marca, de forma a que esta possa ser retirada aquando do disparo da transição. Quando este disparo ocorre, verifica-se a remoção de marcas nos lugares de entrada e a criação de novas marcas nos lugares de saída.

O disparo de uma transição é instantâneo, ou seja, as duas acções citadas são efectuadas ao mesmo tempo.

Apresenta-se na Figura 1.7 o exemplo do disparo de uma transição.

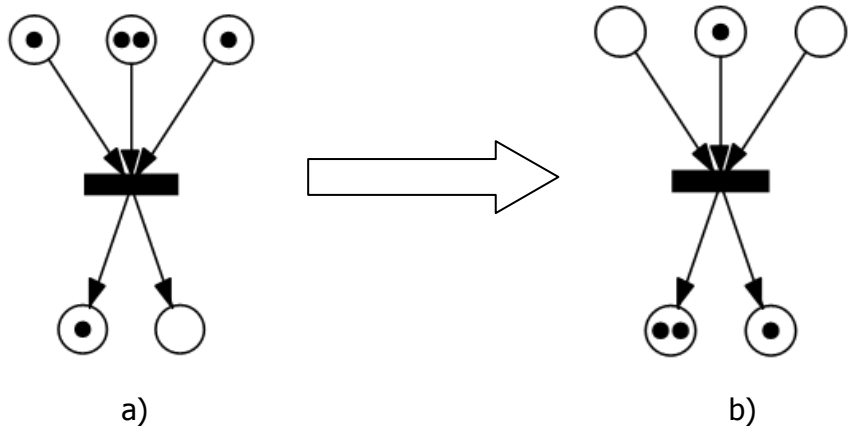


Figura 1.7 – Disparo de uma transição. a) corresponde à situação inicial e b) representa a rede obtida após o disparo

1.2.3. Modelação com Redes de Petri

Ao modelar um sistema através de uma RdP, está necessariamente a ser criada uma interpretação da rede. É essa interpretação ou significação que efectua a ligação do modelo abstracto que qualquer RdP representa, com o sistema concreto que se pretende modelar.

Por exemplo, uma possível interpretação da rede da Figura 1.6, é a que se apresenta na Figura 1.8, onde temos a modelação de um sistema produtor-consumidor onde o armazém tem capacidade igual a dois [Gomes, 97].

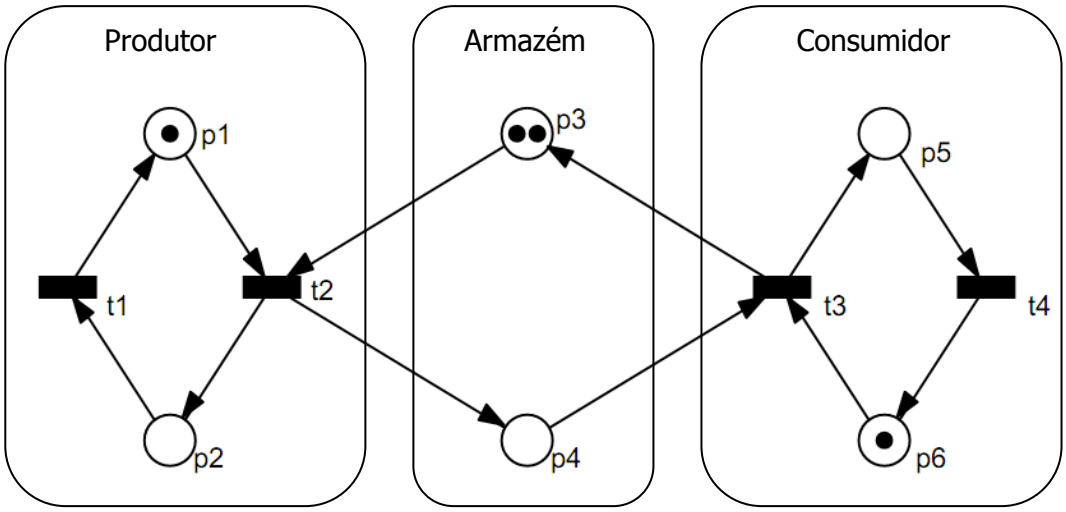


Figura 1.8 - Rede de Petri de sistema produtor-consumidor

O ciclo de produção é modelado pelos lugares p1 e p2 e pelas transições t1 e t2, enquanto o ciclo de consumo agrupa os lugares p5 e p6 e as transições t3 e t4. Os lugares p3 e p4 modelam um armazém, sendo a sua capacidade representada pelo lugar p3 e os produtos armazenados por p4.

1.2.4. Redes de Petri não autónomas

As RdP podem-se dividir em dois grupos: RdP autónomas e RdP não autónomas. As primeiras incluem a componente abstracta do modelo RdP do sistema (estrutura de grafo e marcação) e as segundas integram características que as ligam ao mundo exterior (por exemplo, temporizações, sinais de entrada e saída) [Gomes, 97].

Como as RdP autónomas não são adequadas para modelar alguns dos aspectos importantes presentes em grande parte de sistemas a eventos discretos, nomeadamente as questões de tempo e sinais externos, encontram-se três tipos de extensões para dar resposta a essas questões, segundo [Gomes, 97]:

- As que conferem uma interpretação específica à rede, permitindo integrar no grafo referências a características do sistema físico (como sinais de controlo);
- As que conferem a capacidade de testar um estado (qualquer) de marcação;
- As que conferem a capacidade de integrar dependências temporais.

No presente trabalho, enfatiza-se a utilização do primeiro grupo, ou seja, utilizam-se RdP que integram a representação de sinais e eventos externos. Veremos de seguida (classe IOPT).

1.2.4.1. Classe IOPT (Input-Output Place-Transition)

As RdP IOPT possuem propriedades que possibilitam a modelação de sistemas com ligação ao meio físico, sendo que a sua evolução depende desse mesmo meio físico, ou seja, é uma rede não-autónoma.

As RdP IOPT constituem uma extensão simples e intuitiva às RdP Lugar-Transição [Reisig, 85] e que se destinam à modelação do comportamento de controladores de sistemas conduzidos por eventos discretos. Para tal, permite a especificação de sinais e eventos externos, quer de entrada, quer de saída.
Os eventos e sinais de entrada ficam associados à evolução da rede, nomeadamente às transições. É possível ainda associar, a cada transição, zero ou vários sinais de entrada que formam uma guarda de transição, estas guardas são condição necessária, mas não suficiente para a habilitação das transições respectivas. Por último, os sinais de saída ficam associados à marcação dos lugares.

Em [Gomes et al., 2007], são apresentadas algumas definições básicas das redes IOPT:

- Definição 1 (interface do sistema): A interface do sistema a ser controlado com a rede IOPT é um tuplo *ICS = (IS, IE, OS, OE)* satisfazendo as seguintes condições:

1) *IS* é um conjunto finito de sinais de entrada.
2) *IE* é um conjunto finito de eventos de entrada.
3) *OS* é um conjunto finito de sinais de saída.
4) *OE* é um conjunto finito de eventos de saída.
5) $IS \cap IE \cap OS \cap OE = \varnothing$

- Definição 2 (estado de entrada do sistema): Considerando a interface *ICS = (IS, IE, OS, OE)* com um sistema controlado (Definição 1), estado de entrada do sistema é definido pelo par *SIS = (ISB, IEB)* , satisfazendo as seguintes condições:

1) *ISB* é um conjunto finito de sinais de entrada: $ISB \subseteq IS \times \mathrm{N}_0$
2) *IEB* é um conjunto finito de eventos de entrada: $IEB \subseteq IE \times \mathrm{B}$

A definição das redes IOPT pressupõe o uso de uma linguagem da inscrição, com uma sintaxe concreta, permitindo a especificação de expressões algébricas, variáveis e funções para a especificação de guardas e de condições da transição, bem como acções de saída associadas aos lugares.

O conjunto de expressões booleanas tem o nome de *BE* e a função *Var(E)* retorna um conjunto de variáveis utilizados na expressão *E* [Gomes et al., 2007].

- Definição 3 (rede IOPT): Dado o controlador com a interface *ICS = (IS, IE, OS, OE)*, a rede IOPT é um tuplo *N=(P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)* satisfazendo as seguintes condições:

1) *P* é um conjunto finito de lugares.

2) *T* é um conjunto finito de transições (separados de *P*).

3) *A* é um conjunto de arcos, tal que $A \subseteq \left((P \times T) \cup (T \times P)\right)$.

4) *TA* é um conjunto de arcos de teste, tal que $TA \subseteq (P \times T)$.

5) *M* é a função de marcação $M : P \to \mathrm{N}_0$.

6) *weight* : $A \to \mathrm{N}_0$, representa a função de peso dos arcos.

7) *weighTest* : $TA \to \mathrm{N}_0$, representa a função peso dos arcos de teste.

8) *priority* é uma função parcial que aplica transições a inteiros não negativos: $priority : T \to N_0$

9) *isg* é uma função parcial de guarda de sinal de entrada que aplica transições a expressões booleanas, onde as variáveis são sinais de entrada: $isg : T \to BE$ onde $\forall eb \in$ *isg(T), Var(eb)* $\subseteq$ *IS.*

10) *ie* é uma função parcial de evento de entrada que aplica transições a eventos de entrada: $ee : T \to IE$.

11) *oe* é uma função parcial de evento de saída que aplica transições a eventos de saída: $ee : T \to IE$.

12) *osc* é uma função de sinal de saída de lugares para conjuntos de regras: $osc : P \to \mathrm{P}(RULES)$, onde $RULES \subseteq (BES \times OS \times \mathrm{N}_0), BES \subseteq BE$ e $\forall e \in BES, Var(e) \subseteq ML$ com *ML* sendo o conjunto de identificadores para cada marcação de lugar depois de um estado de execução.

Quando disparadas, as transições podem emitir eventos de saída (*oe*), que por sua vez irão alterar os sinais a que estão associados. Os sinais de saída também podem ser alterados pelas marcações de lugar (*osc*). Para cada transição, é também possível associar nível de prioridade (*priority*), guardas que usam sinais externos (*isg*), tal como eventos de entrada (*ie*) e de saída (*oe*).

As guardas são funções dos sinais externos de entrada. Sempre que uma transição está habilitada e as condições externas associadas são avaliadas como verdadeiras (eventos de entrada e sinais de entrada) então a transição dispara.

1.2.4.2. Representação em PNML

O PNML (Petri Net Markup Language) é um formato de representação de RdP baseado na acção XML. A característica específica do PNML é a sua distinção entre características gerais de todos os tipos de RdP e características específicas de um determinado tipo de RdP.

As características específicas são definidas em separado, numa norma: Petri Net Type Definition (PNTD) para cada tipo de RdP.

O uso do PNML facilita a troca de representações de RdP entre diferentes ferramentas existentes utilizando RdP, mesmo que essas ferramentas suportem classes de RdP ligeiramente diferentes. Características específicas a uma classe de redes que outras ferramentas desconheçam, ou simplesmente não utilizem, podem ser simplesmente ignoradas.

De acordo com [Barros & Gomes, 04], existem duas formas de compor modelos RdP utilizando PNML:

- Utilizando PNML estrutural: cada rede torna-se numa página e cada página pode conter referências para outras páginas através do uso de lugares e/ou transições de referência.

- Utilizando PNML modular: cada rede torna-se um módulo. Cada módulo pode especificar uma parte de uma implementação e uma parte da interface com nós de referência, portas de saída. Os módulos são conectados, única e exclusivamente através da importação e exportação de nós.

O PNML estrutural oferece uma forma simples de partir um modelo em vários submodelos, contendo referências uns aos outros. Cada nó pode ser uma referência para outro nó noutra página, existindo assim, nós e nós de referência.

O PNML modular é um pouco diferente, permite a criação de várias instâncias independentemente do modelo inicial. É muito útil uma vez que cada módulo pode ser utilizado em contextos diferentes, possivelmente ao mesmo tempo. Contudo, o projectista do modelo não tem que saber em que contexto o mesmo será utilizado, apenas a sua funcionalidade.

O PNML foi utilizado em trabalhos associados a esta dissertação como meio de ligação entre o editor de Redes de Petri Snoopy IOPT [Nunes, 08] e a ferramenta de geração autónoma de código C [Rebelo, 10], desenvolvidos no âmbito do projecto FORDESIGN [Fordesign, 07] [Gomes, Barros, Costa, 07].

Para informação mais detalhada sobre PNML assim como a forma de o obter através do editor de Redes de Petri, consultar [Nunes, 08].

1.2.5. Projecto FORDESIGN

O PNML foi utilizado no trabalho desenvolvido nesta dissertação como elo de ligação entre o editor de Redes de Petri Snoopy IOPT e o conversor de código para a linguagem C (PNML2C). Estas ferramentas foram desenvolvidas no âmbito do projecto FORDESIGN que tem como objectivo a criação de diversas ferramentas de apoio à modelação de sistemas por RdP, fazendo uso do PNML como linguagem neutra intermédia. É ilustrada a relação entre estas ferramentas na Figura 1.9.

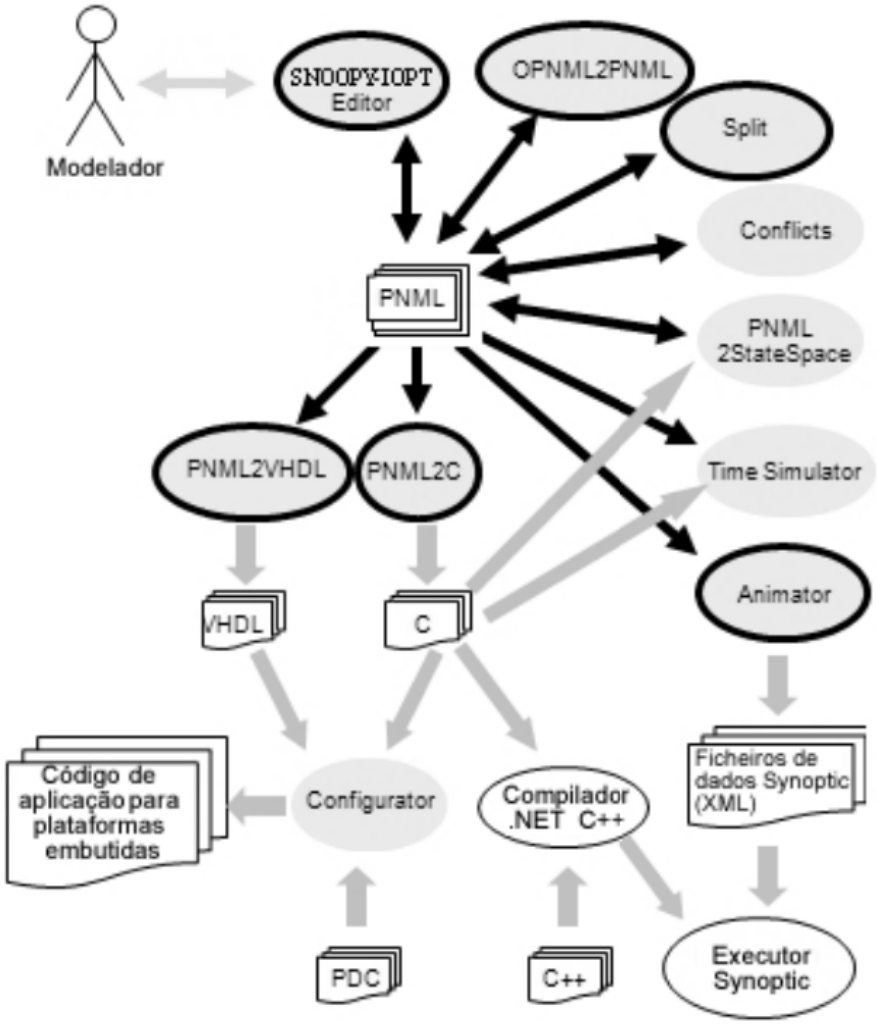


Figura 1.9 – Ferramentas Fordesign , adaptado de [Fordesign, 07]

Para uma análise mais pormenorizada das ferramentas utilizadas para o desenvolvimento desta aplicação, sugere-se a consulta de [Fordesign, 07] e [Gomes, Barros, Costa, 07].

2. Sistema de controladores domóticos – Tiny Domots

Neste capítulo são apresentadas as características do sistema de controladores domóticos existente, realizado no âmbito de um projecto de final de curso [Moutinho, 04]. Tal como já mencionado, o sistema a desenvolver com recurso a ferramentas de edição de RdP e geração automática de código deverá possuir as mesmas funcionalidades da versão exposta neste capítulo.

O trabalho realizado anteriormente, consistiu em desenvolver um sistema de domótica que permitisse controlar e interligar os equipamentos eléctricos existentes numa casa, tornando-a assim inteligente.

Para tal, foi criada uma unidade genérica básica a que se chamou Tiny Domot.

Junto a cada equipamento eléctrico pode ser colocado um Tiny Domot que o controla, ou seja, lê informação e actua sobre o aparelho. Por sua vez os vários Tiny Domots são ligados entre si através de uma rede.

Foi desenvolvido um Tiny Domot genérico, que é configurado/ programado de acordo com equipamento que controla e o modo como o quer controlar. Este é configurado/programado a partir de uma aplicação desenvolvida em C++ para funcionar no sistema operativo Windows e que permite ao utilizador ter controlo sobre os vários Tiny Domots, sendo através dela que o utilizador regista, configura, programa eventos, reconfigura ou retira os Tiny Domots da rede.

2.1. Descrição do sistema

Trata-se de uma arquitectura mista, onde a inteligência está distribuída pelos equipamentos e onde estes conseguem comunicar entre si. O objectivo é interligar / integrar equipamentos diferentes, com tecnologias distintas como exemplificado na Figura 2.1.

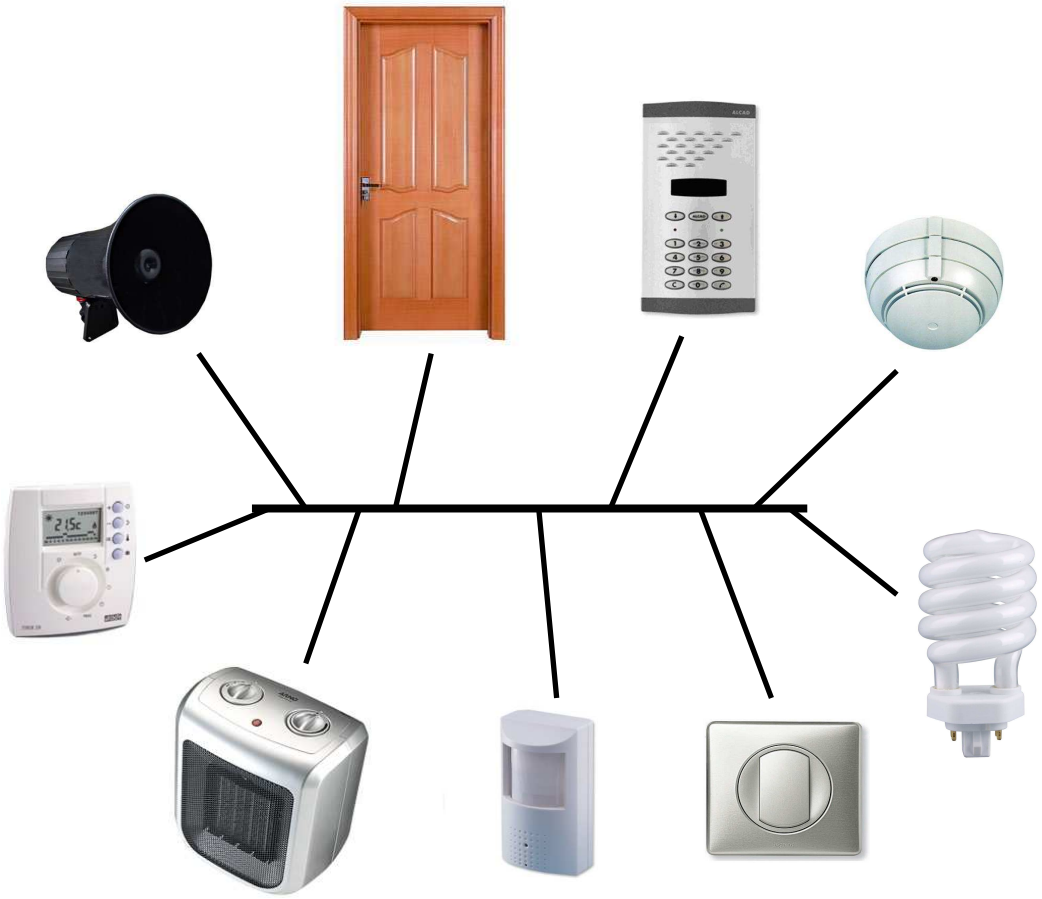


Figura 2.1 – Equipamentos domésticos, adaptado de [Moutinho, 04]

Para tal, foi desenvolvida uma unidade inteligente, que faz a ligação entre os equipamentos e a rede de comunicação. A rede de comunicação (também designada por *BUS* de comunicação) interliga todas as unidades entre si.

Ao longo deste documento, chamar-se-á Tiny Domot – TD (o termo já vem de um projecto anterior) à unidade inteligente.

"desenvolveu-se um sistema de unidades microcontroladas vocacionadas para controlo domótico. Estas unidades, a que se dá o nome de Tiny Domot (Tiny = pequeno + Domot = domótica – pequeno dispositivo domótico), " [Santana, 2002]

A título de exemplo, na Figura 2.2, atribui-se um equipamento a cada TD, no entanto, cada TD possui 12 entradas digitais e 11 saídas digitais. Desta forma, cada TD tem a possibilidade de interagir directamente com 1 ou mais equipamentos ligados a si sendo o seu limite o número de entradas e saídas disponíveis.

Os TD's têm a capacidade de enviar mensagens entre eles (arquitectura mista), por exemplo, se o interruptor do quarto for pressionado, o seu TD enviará uma mensagem para o TD que controla a lâmpada e esta acender-se-á.

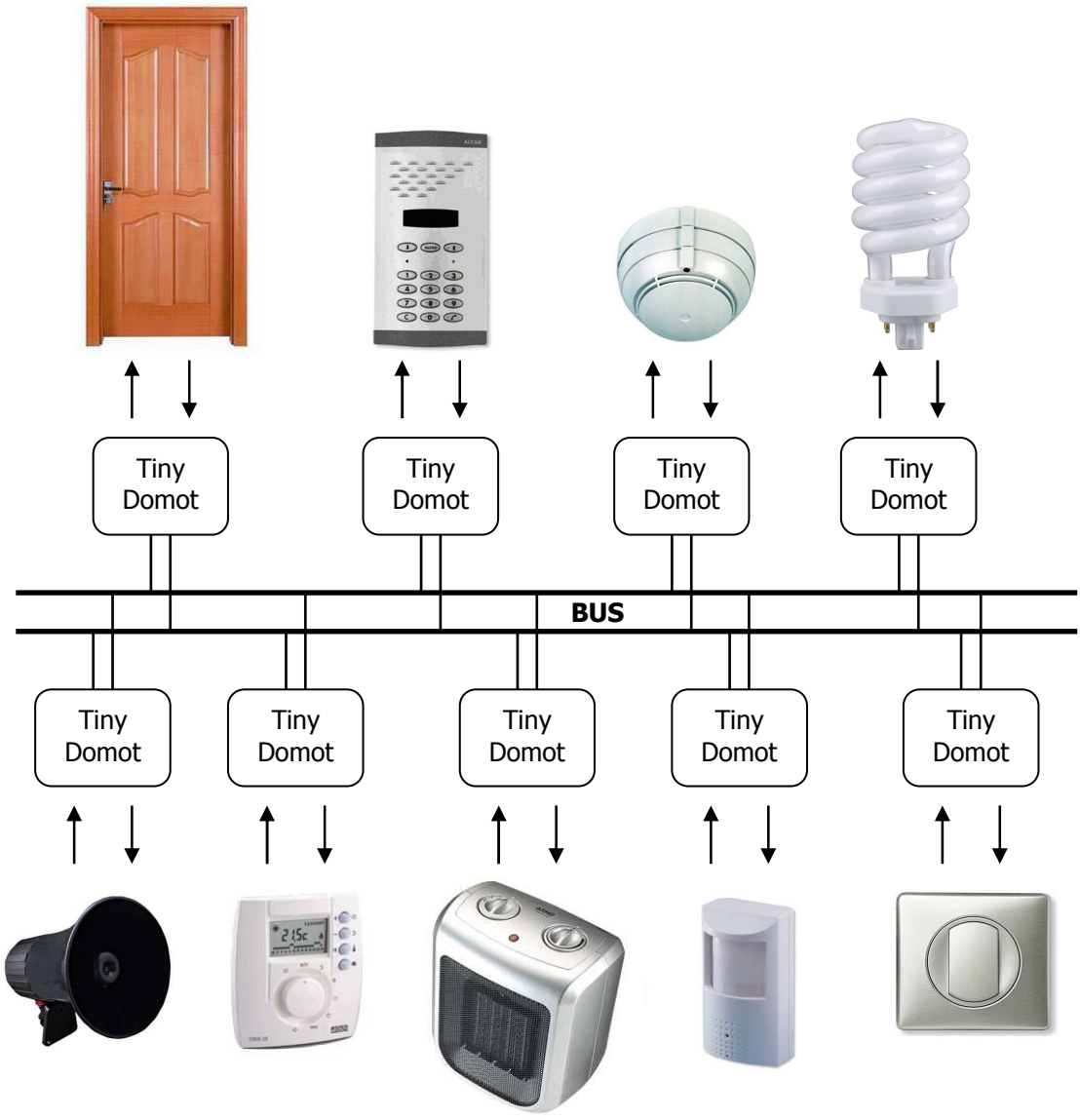


Figura 2.2 – Equipamentos domésticos interligados através de TDs, adaptado de [Moutinho, 04]

Cada TD tem um comportamento específico, dependendo do tipo de equipamentos que controla e das funcionalidades que se pretende que estes tenham. Para tal foi desenvolvido um TD configurável, que se pode adaptar aos equipamentos e às funcionalidades que se pretende dar aos mesmos. Por exemplo, enquanto que a lâmpada de um quarto só se liga quando se carrega no interruptor, a lâmpada do hall de entrada pode ligar-se quando se carrega no interruptor, quando se detecta presença ou quando existe baixa luminosidade no hall de entrada.

Outra das potencialidades do sistema, é a execução de determinados eventos a determinadas horas. Por exemplo, o despertador pode ser programado para tocar todos os dias da semana às 7h.

Resumidamente, as principais características do TD desenvolvido anteriormente são:

- a realização de determinadas acções (actuar uma saída ou enviar uma mensagem) quando se verifiquem determinados acontecimentos no equipamento que está a controlar ou nos outros equipamentos (realizar funções lógicas);

- ser configurável de acordo com o modo de funcionamento desejado, para que possa realizar a função pretendida (de acordo com o ponto anterior);

- ser reconfigurável, para que o sistema se possa adaptar a futuras alterações;

- ter capacidade de executar eventos na data e hora programada.

Antes do sistema iniciar o seu normal funcionamento, é necessário registar, configurar e programar os TD's. Para tal, foi desenvolvida uma aplicação para ser executada num computador, o qual está ligado ao *BUS* de comunicação, para assim poder comunicar com todos os TD's. Na Figura 2.3 apresenta-se esta ligação.

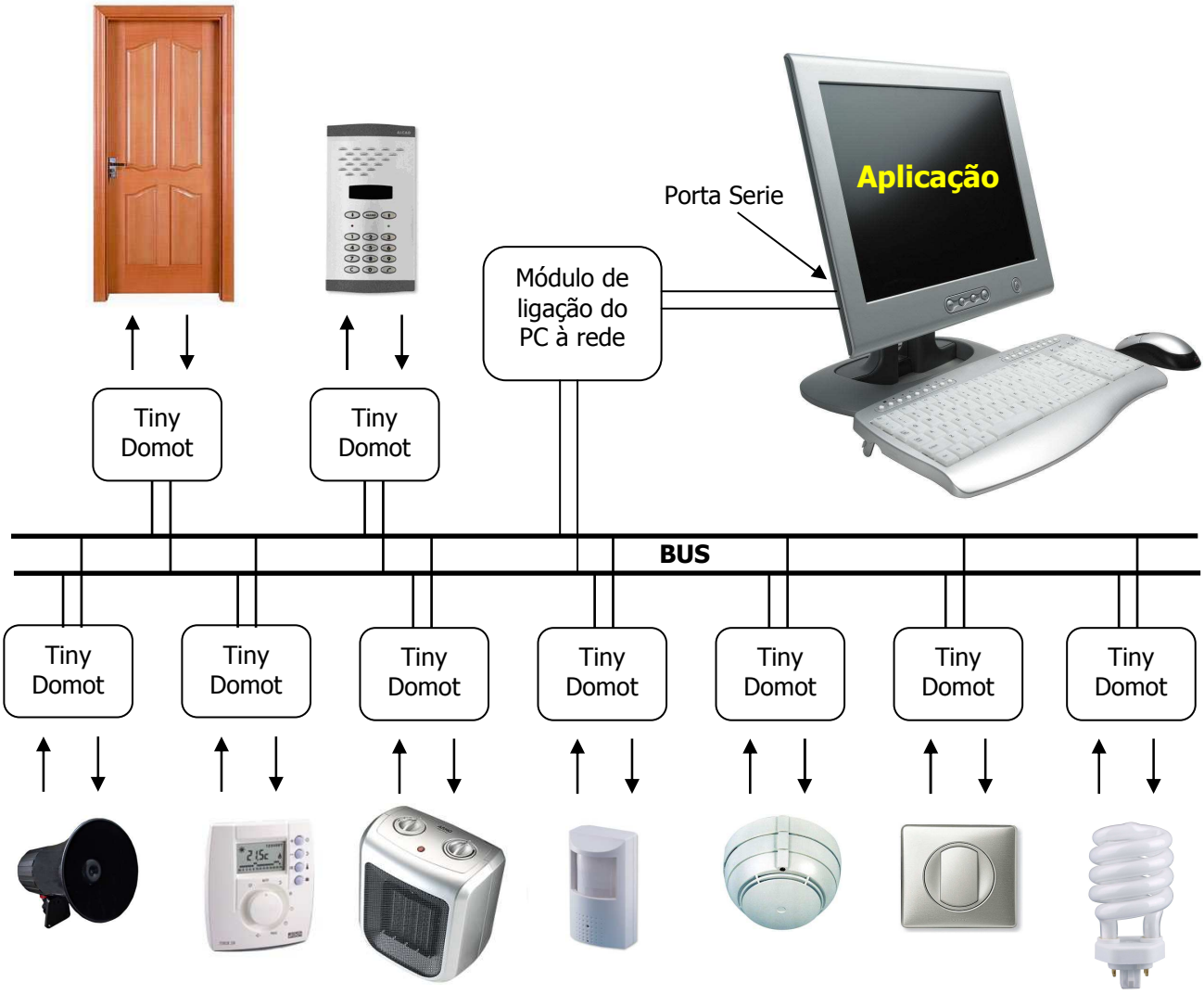


Figura 2.3 – Ligação da aplicação ao sistema, adaptado de [Moutinho, 04]

2.2. Funcionalidades

Todos os TD's têm de ter uma identificação. Como tal, quando se liga um TD na rede, é necessário registá-lo, ou seja, atribuir-lhe um nome e um endereço).

Em seguida, para definir o seu comportamento é necessário configurá-lo. Configurar é associar funções lógicas às saídas e associar acções às entradas.

As acções que o TD realiza são: enviar mensagens para outros TD's ou actuar nas suas saídas.

Para que o TD execute determinados eventos em determinadas datas, é necessário programá-lo adequadamente (inserir data e hora, qual a saída a actuar e definir se será um evento único ou semanal).

Depois de registado, configurado e programado, o TD entra no seu normal funcionamento. Assim, a aplicação deixa de ser necessária e o computador pode ser desligado. Ficam apenas os TD's ligados na rede, a enviar mensagens entre eles com a informação do estado das suas entradas, ou seja, do equipamento, sensor, etc. que estão a observar, e a actuar nos equipamentos a que estão ligados.

Os TD's executam funções lógicas, isto é, actuam nas suas saídas (nos equipamentos) quando se verificam determinadas condições, previamente estabelecidas.

As condições podem ser:

- o estado das suas entradas
- e/ou determinadas mensagens recebidas

De seguida apresenta-se o Diagrama de Casos de Uso, onde se podem observar os requisitos e as funcionalidades do TD.

2.2.1. Diagrama de Casos de Uso

Na Figura 2.4 apresenta-se o diagrama de casos de uso do TD, sendo que no próximo subcapítulo é feita a descrição dos vários casos de uso.

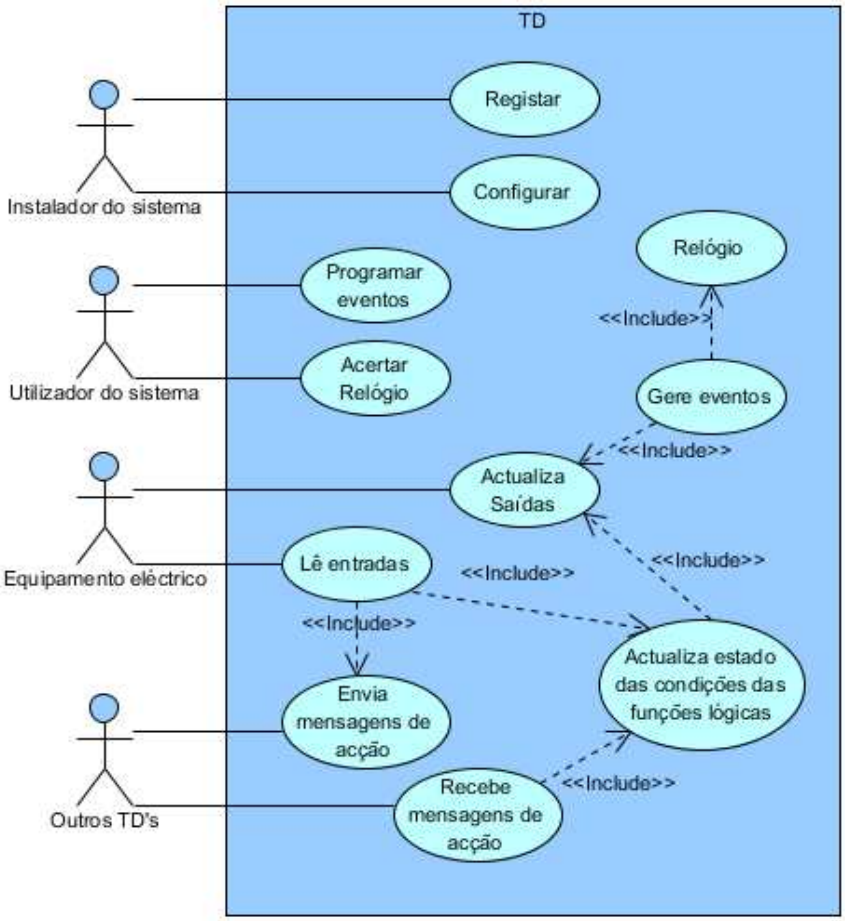


Figura 2.4 – Diagrama de casos de uso do TD, adaptado de [Moutinho, 04]

Os actores do TD são:

- Instalador do sistema – Pessoa que instala o sistema de domótica na casa, e que faz a sua gestão.

- Utilizadores do sistema – Todas as pessoas que estiverem dentro de casa.

- Equipamento eléctrico – São todos os equipamentos da casa que vão ser controlados pelo sistema de domótica.

- Outros TD's – São os TD's que se encontram ligados ao sistema

2.2.2. Descrição dos Casos de Uso

Registar	
Pré-condição	O novo TD tem de estar ligado na rede.
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de registo vinda da aplicação (instalador do sistema). **2.** Se o TD não estiver registado, regista-se, isto é, fica com o endereço que vem na mensagem. **3.** Envia a mensagem de resposta à aplicação.
Pós-condição	

Quadro 2.1 – Descrição do Caso de Uso *Registar*

Configurar	
Pré-condição	O TD tem de estar registado.
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de configuração vinda da aplicação (instalador do sistema). **2.** O TD guarda a informação de configuração na base de dados. **3.** Envia a mensagem de resposta à aplicação.
Pós-condição	

Quadro 2.2 – Descrição do Caso de Uso *Configurar*

Programar eventos	
Pré-condição	O TD tem de estar registado.
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de programação da aplicação. **2.** O TD guarda os eventos na base de dados. **3.** Envia a mensagem de resposta.
Pós-condição	A determinadas horas o TD vai realizar os eventos.

Quadro 2.3 – Descrição do Caso de Uso *Eventos*

Relógio	
Pré-condição	O relógio necessita de ter a data e a hora correcta.
Descrição:	Como o nome indica, é o relógio do TD.
Pós-condição	

Quadro 2.4 – Descrição do Caso de Uso *Relógio*

Gere eventos	
Pré-condição	O TD tem de ter eventos programados.
Descrição:	**1.** O caso de uso está constantemente a ver o relógio e os eventos programados. **2.** Quando encontra um evento para ser realizado, realiza-o, isto é, actualiza a saída correspondente. **3.** Caso se trate de um evento único, apaga-o.
Pós-condição	

Quadro 2.5 – Descrição do Caso de Uso *Gere Eventos*

Lê entradas	
Pré-condição	O TD tem de estar configurado.
Descrição:	O caso de uso está constantemente a analisar as entradas e a configuração, se alguma for alterada vai guardar essa informação na base de dados, para mais tarde desencadear acções.
Pós-condição	Vai haver uma acção de saída nesse aparelho, ou o envio de uma mensagem para outro aparelho.

Quadro 2.6 – Descrição do Caso de Uso *Lê Entrada*

Envio de mensagens de acção	
Pré-condição	O TD tem de estar configurado.
Descrição:	Envia mensagens para outros TDs.
Pós-condição	O TD irá receber as mensagens de resposta. Caso não receba terá de voltar a enviar as mensagens.

Quadro 2.7 - Descrição do Caso de Uso *Envia Mensagem*

Recepção de mensagens de acção	
Pré-condição	O TD tem de estar configurado.
Descrição:	**1.** O caso de uso começa quando recebe uma mensagem de outro TD. **2.** Confirma se a mensagem é para si. **3.** Consulta a base de dados para ver se a mensagem lhe interessa. **4.** Guarda na base de dados informação em como recebeu a mensagem. **5.** Envia mensagem de resposta.
Pós-condição	Esta mensagem recebida poderá alterar as saídas dos TDs.

Quadro 2.8 – Descrição do Caso de Uso Recebe Mensagem de Acção

Actualiza estado das condições das funções lógicas	
Pré-condição	Estar configurado e receber uma mensagem de outro TD ou uma das entradas ser alterada.
Descrição:	Actualiza estado das condições das funções lógicas.
Pós-condição	Estas alterações nos estados das condições, poderão alterar os valores das saídas.

Quadro 2.9 – Descrição do Caso de Uso *Actualiza estado das condições das funções lógicas*

Actualiza saídas	
Pré-condição	O TD tem de estar configurado ou ter eventos programados.
Descrição:	O TD analisa a configuração (funções lógicas) das suas saídas, o estado das condições e actualiza as saídas, isto é, actua nos equipamentos electricos.
Pós-condição	

Quadro 2.10 – Descrição do Caso de Uso *Actualiza Saídas*

2.3. Modelo comportamental – Diagramas de Actividade

Para se perceber o funcionamento do TD apresenta-se de seguida vários diagramas de actividade do mesmo. Apresentam-se apenas os diagramas principais e algumas breves explicações.

Quando o TD é ligado na rede, a função que regista fica à espera de receber uma mensagem de registo. Ao receber esta mensagem, a função guarda na base de dados o endereço do TD, e envia uma mensagem de resposta à aplicação a dizer que está registado.

Depois de registado a função de configuração fica à espera das mensagens de configuração e a função de programação de eventos fica à espera das mensagens de programação de eventos, ou da mensagem para acertar a data e a hora.

Depois do TD estar registado, configurado, programado e com a data e a hora certas, entra no seu normal funcionamento, onde recebe e envia mensagens, lê as entradas, actualiza as saídas e gere eventos.

Na Figura 2.5 pode ver-se o diagrama de actividade principal do TD. Em relação a este diagrama há algumas notas a fazer:

- Quando o TD é ligado, este vê se está registado e configurado. Se sim, vai manter as configurações anteriores. Esta função é importante para o caso de lhe faltar a alimentação, assim, quando voltar a ser alimentado não necessita de voltar a ser registado e configurado.

- Outra nota a fazer é o facto de o TD só receber uma nova mensagem depois de ter analisado a mensagem anterior.

- Só envia mensagens se não estiver a receber mensagem, pois se enviasse esta ia chocar com a outra.

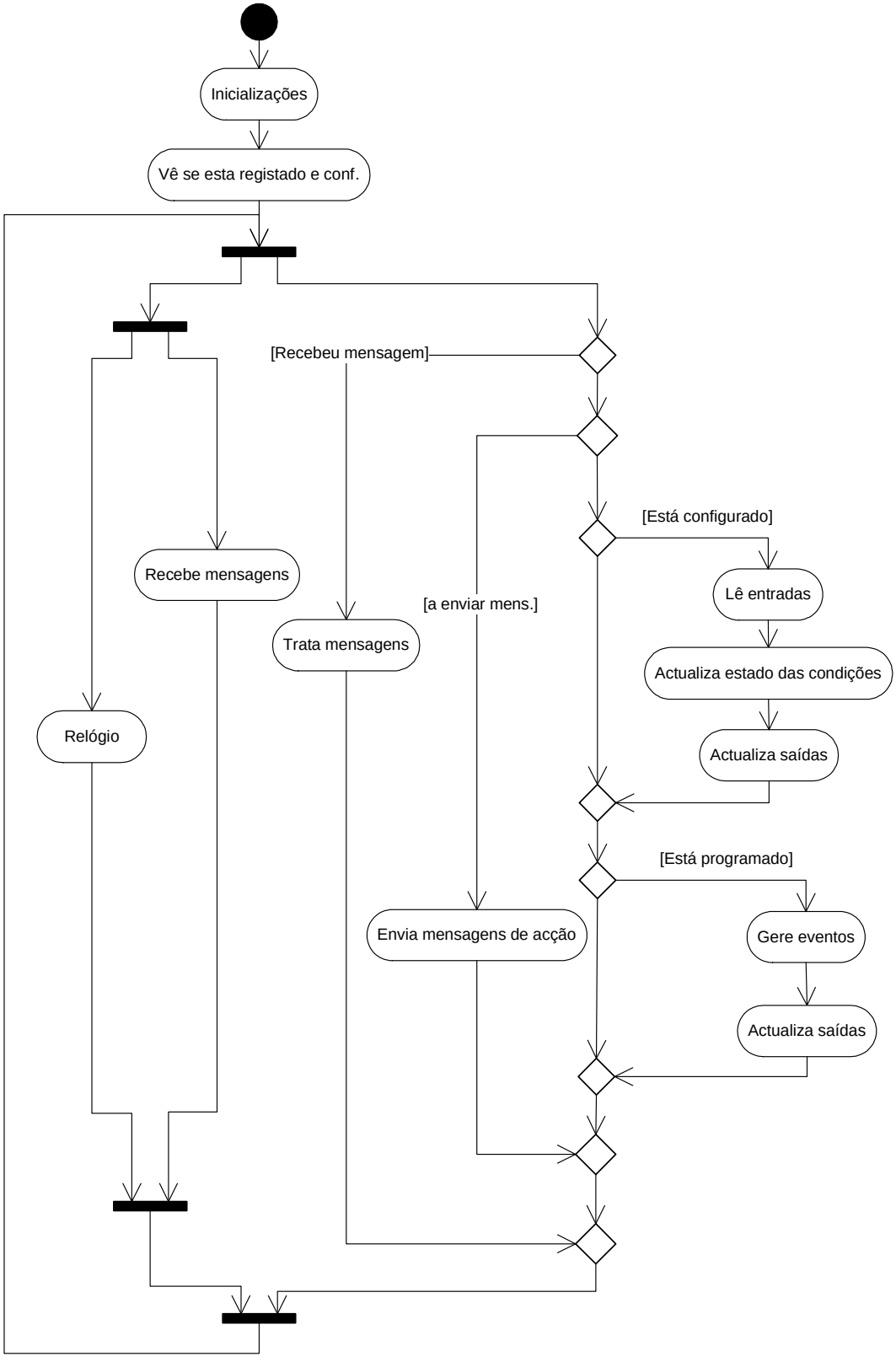


Figura 2.5 – Diagrama de actividades da função *Main*

Trata mensagens

Quando recebe uma mensagem, o TD começa por verificar se a mensagem foi bem recebida e de seguida verifica que tipo de mensagem é:

- mensagem de resposta (também designada por *acknowlegde*),
- mensagem de registo,
- mensagem de configuração,
- mensagem de programação de eventos,
- mensagem de actualização de data e hora,
- mensagem de acção.

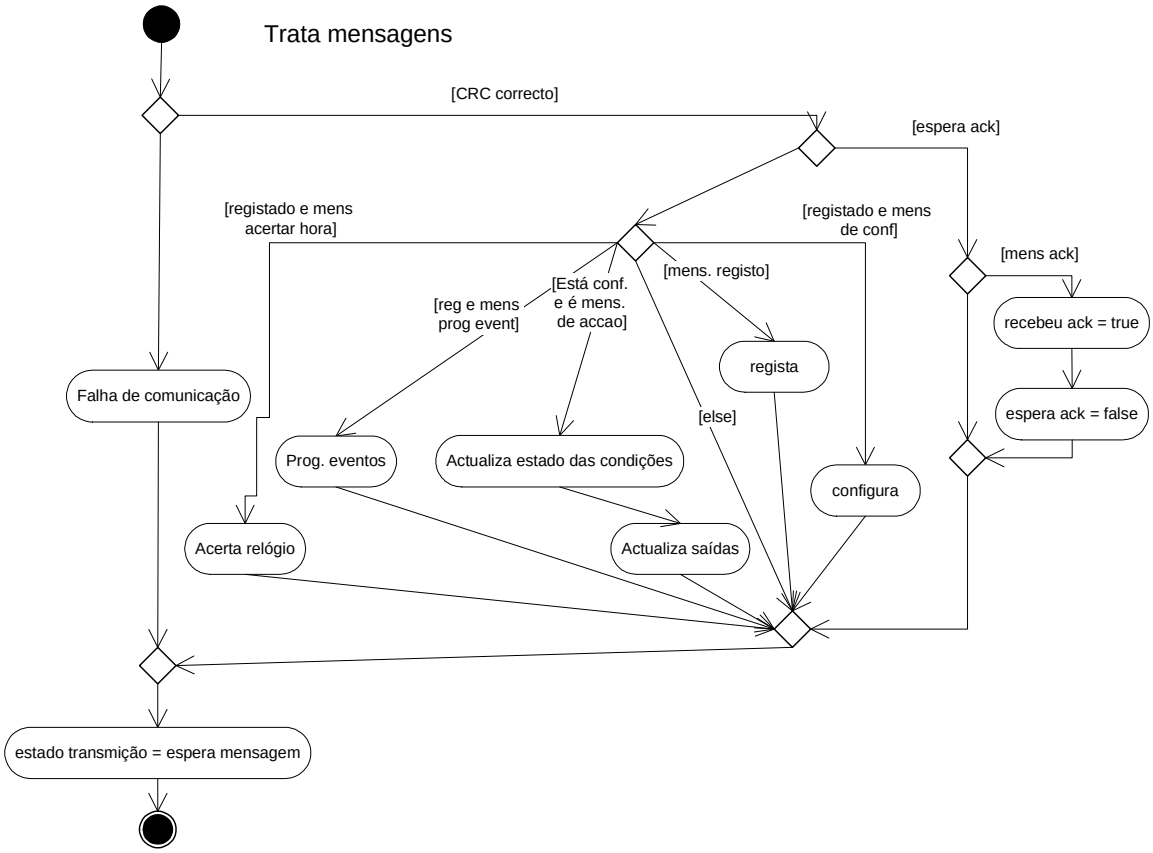


Figura 2.6 – Diagrama de actividades da função *Trata mensagens*

Regista

Cada TD tem de ter um endereço, como os TDs não têm endereço físico, não há nada que diferencie um TD de outro antes de este ser registado. Assim na altura do registo, é necessário ter em atenção que apenas deve existir um TD não registado na rede.

É através da aplicação que o instalador do sistema envia para a rede uma mensagem de registo, como só existe um TD à escuta da mensagem de registo, só um a vai receber. Nessa mensagem, o TD recebe o seu endereço, e vai enviar uma mensagem de resposta à aplicação, a dizer que recebeu o endereço e que tipo de TD é. A partir deste momento o TD está registado e como tal, tem uma identificação diferente de todos os outros na rede.

Para se perceber melhor o funcionamento da função de registo ver Figura 2.7.

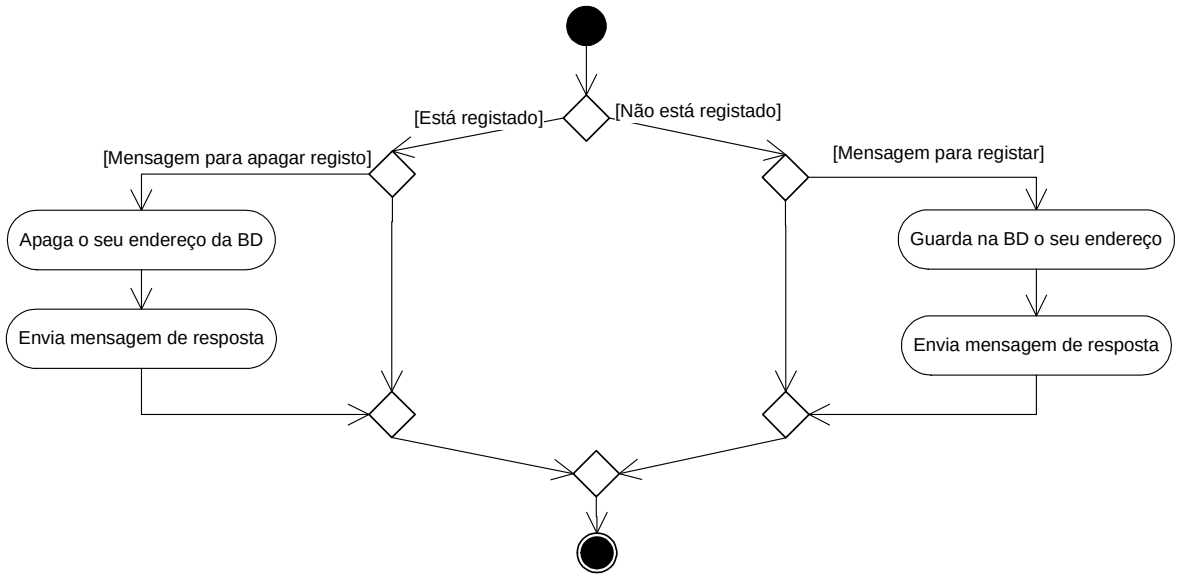


Figura 2.7 - Diagrama de actividades da função *Regista*

Configura

Depois de registado, o TD tem de ser configurado, isto é, é necessário dizer ao TD quais as entradas a que tem de prestar atenção, as saídas em que tem de actuar e em que situações (funções lógicas), para que TDs vai enviar mensagens, de quais vai receber mensagens e quais são as mensagens importantes.

Mais uma vez se usa a aplicação para configurar os TDs.

Esta funcionalidade é ilustrada na Figura 2.8.

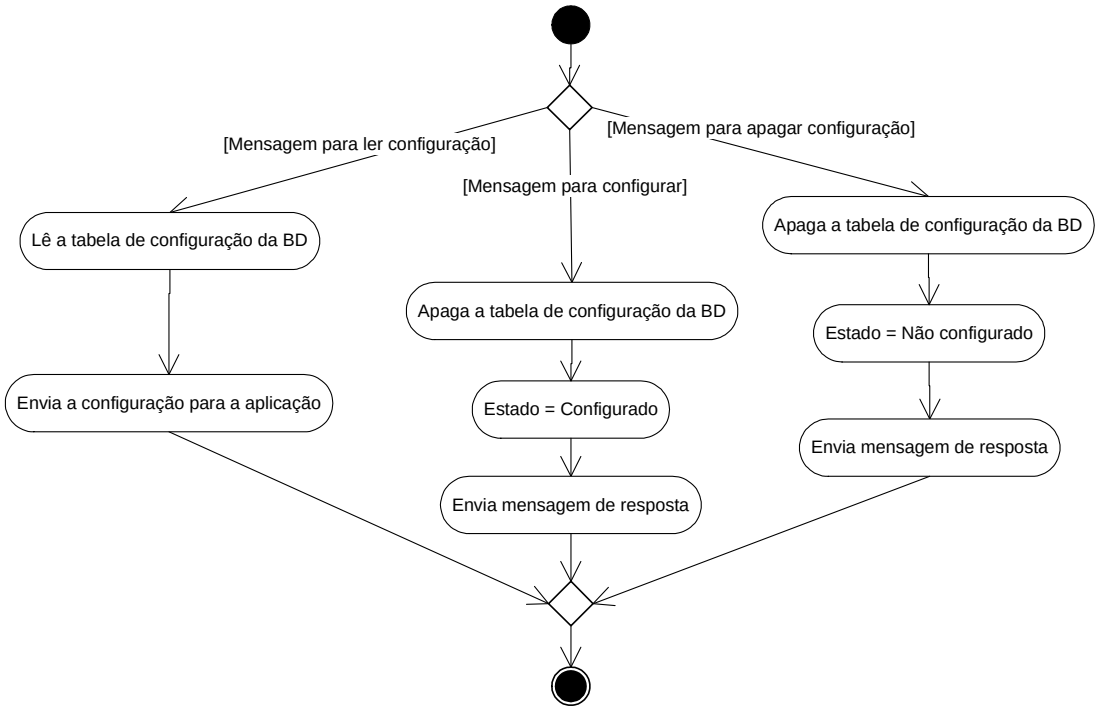


Figura 2.8 – Diagrama de actividades da função *Configura*

A função de configuração recebe a mensagem de configuração vinda da aplicação e guarda a informação correspondente na sua base de dados (BD), nas tabelas de configuração das entradas e das saídas.

Programação de eventos

No TD podem ser programados eventos que vão ocorrer num determinado dia a uma determinada hora. Os eventos podem ser únicos ou semanais. Se for um evento único, este só é executado uma vez, se for semanal, é executado todas as semanas no mesmo dia da semana e à mesma hora.

Esta função recebe as mensagens vindas do computador e guarda na base de dados a informação sobre os eventos que terá de realizar. A mesma é ilustrada na Figura 2.9.

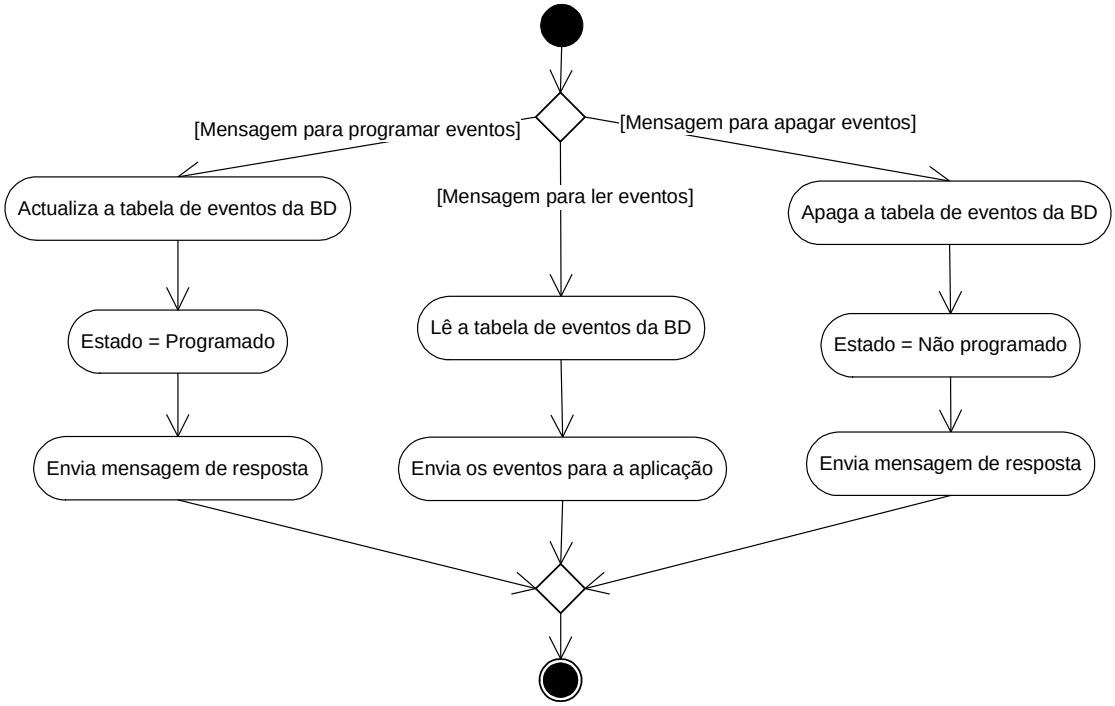


Figura 2.9 – Diagrama de actividades da função *Programa Eventos*

Lê entradas

Os TDs têm entradas, que estão constantemente a ler, para obterem informação sobre o exterior.

O Diagrama de actividades desta função é ilustrado na Figura 2.10.

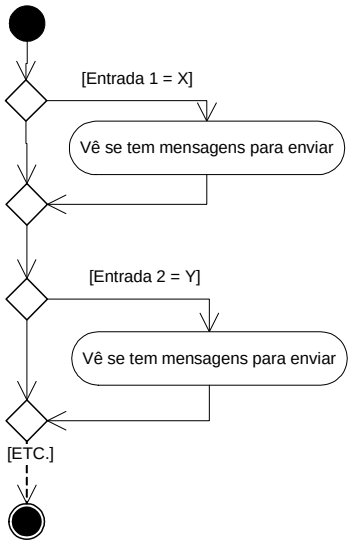


Figura 2.10 – Diagrama de actividades da função *Lê Entradas*

Vê se tem mensagens para enviar

Esta função começa por verificar se a entrada está configurada. Se estiver, vai comparar o seu estado actual com o seu estado anterior, se forem diferentes, o TD terá de actualizar o estado das condições das funções lógicas das suas saídas ou activar o envio mensagens. Ver Figura 2.11.

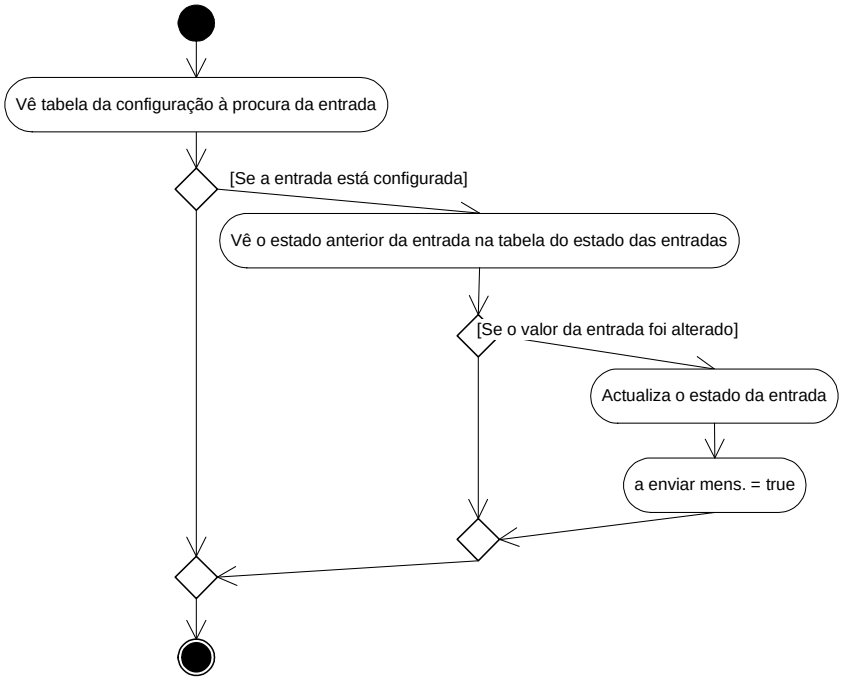


Figura 2.11 – Diagrama de actividades da função *Vê se tem mensagens para enviar*

Envia mensagens

Esta função envia as mensagens de acção para os outros TDs. Para se perceber o funcionamento desta, ver os diagramas seguintes.

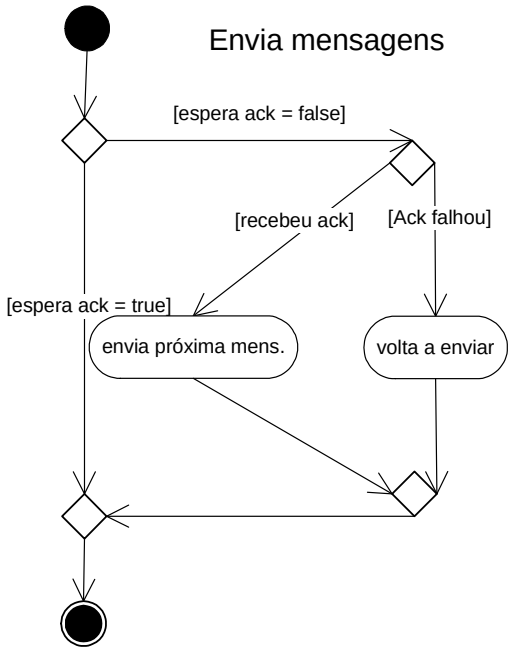


Figura 2.12 – Diagrama de actividades da função *Envia mensagens*

Envia próxima mensagem

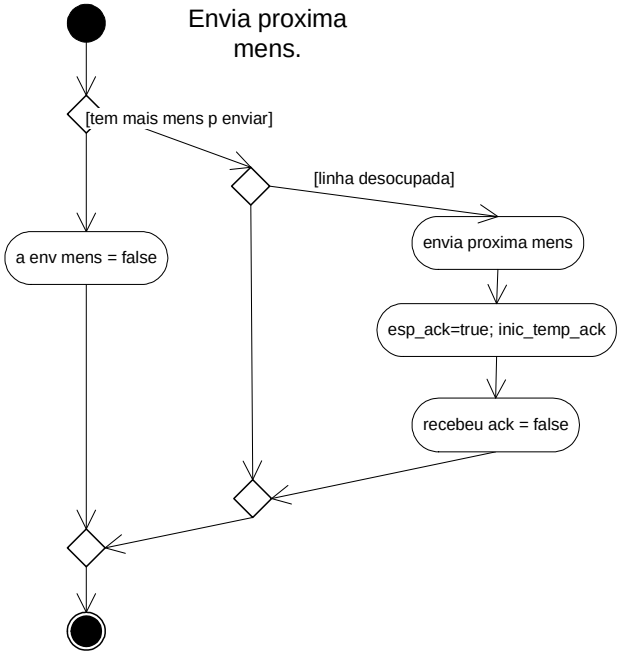


Figura 2.13 – Diagrama de actividades da função *Envia próxima mensagem*

Volta a enviar

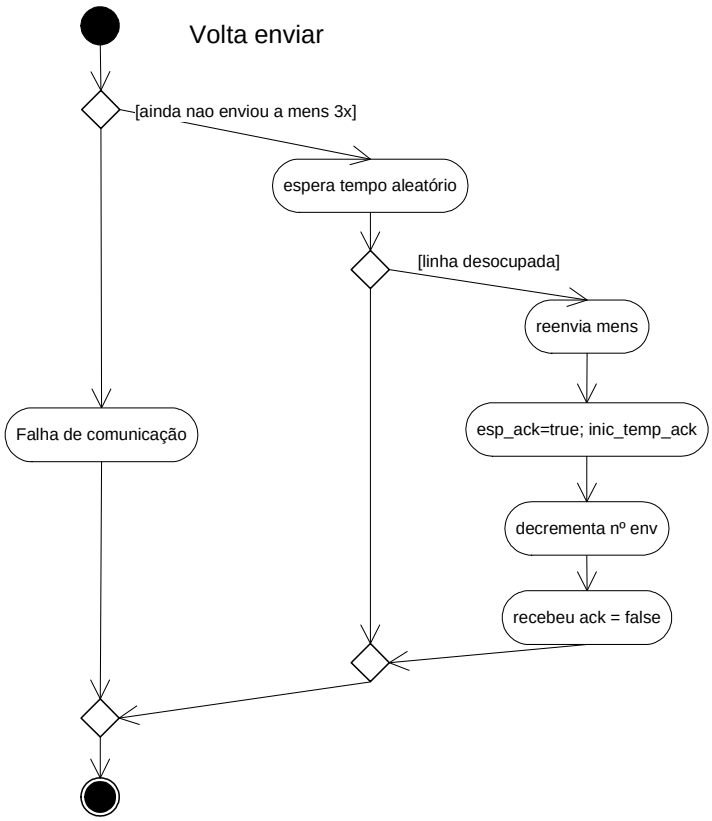


Figura 2.14 – Diagrama de actividades da função *Volta a enviar*

Gere Eventos

Os TDs têm a capacidade de executar eventos, para isto é necessário que estes sejam programados antecipadamente e que a data e a hora do TD estejam correctas.

Os eventos são guardados na base de dados do TD (nas tabelas dos eventos únicos e dos eventos semanais). As tabelas dos eventos únicos e dos eventos semanais estão constantemente a ser lidas, e as datas e as horas dos eventos a serem comparadas com a data e a hora actual, se coincidirem, actualiza-se a saída.

Se for um evento único, este é apagado do TD.

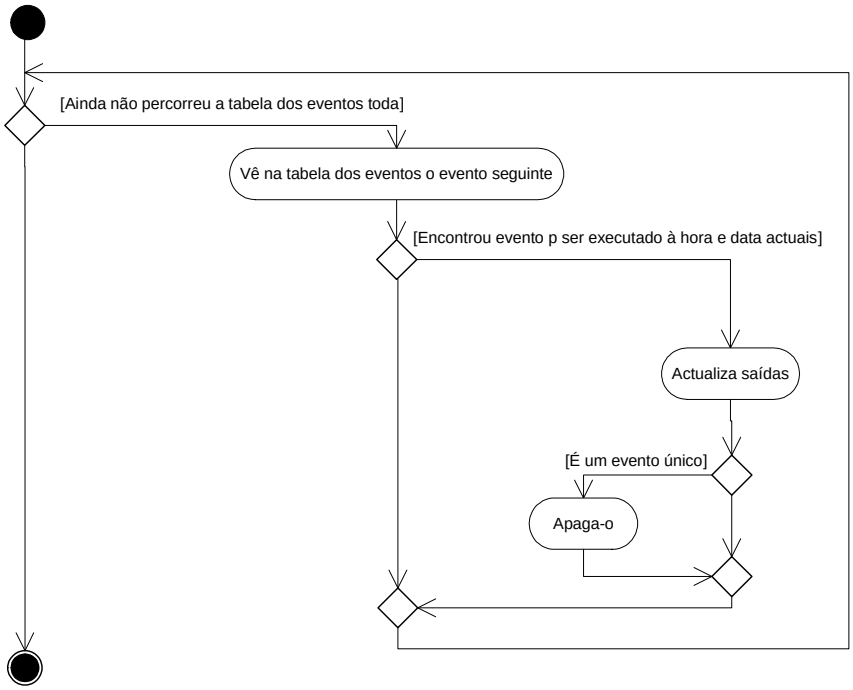


Figura 2.15 – Diagrama de actividades da função *Gere eventos*

Actualiza estado das condições

Esta função é executada quando existe uma alteração numa entrada, ou quando é recebida uma mensagem de acção, e a sua função é actualizar o estado das condições das funções lógicas associadas às saídas.

As funções lógicas e o estado das condições estão armazenados na base de dados, na tabela de configuração das saídas e na tabela dos estados das condições respectivamente.

Exemplo de uma função lógica:

Z = A.B + C.D.E

A saída Z é activada se as condições A e B estiverem activas, ou se as condições C, D e E estiverem activas.

Actualiza saídas

Esta função actualiza o estado das saídas, com base nas funções lógicas a elas associadas e no estado das suas condições.

Relógio

Esta função como o nome indica é o relógio do TD, tem a data e hora actuais. Inicialmente tem de receber informação sobre a data e hora actual e depois disso vai sempre actualizando a hora e o dia (através de um *Timer* do *PIC* que a função actualiza a hora e a data).

O TD possui um relógio para que o gestor de eventos possa saber qual o momento em que o evento deve ocorrer.

2.4. Descrição da aplicação

Os Tiny Domots são registados, configurados e programados a partir de uma aplicação desenvolvida em C++ para funcionar no sistema operativo Windows. Esta aplicação permite ao instalador/utilizador ter controlo sobre os Tiny Domots, sendo através dela que este regista, configura, programa eventos, reconfigura ou retira os Tiny Domots da rede.

De seguida apresenta-se um diagrama de Casos de Uso, onde se podem observar os requisitos da aplicação.

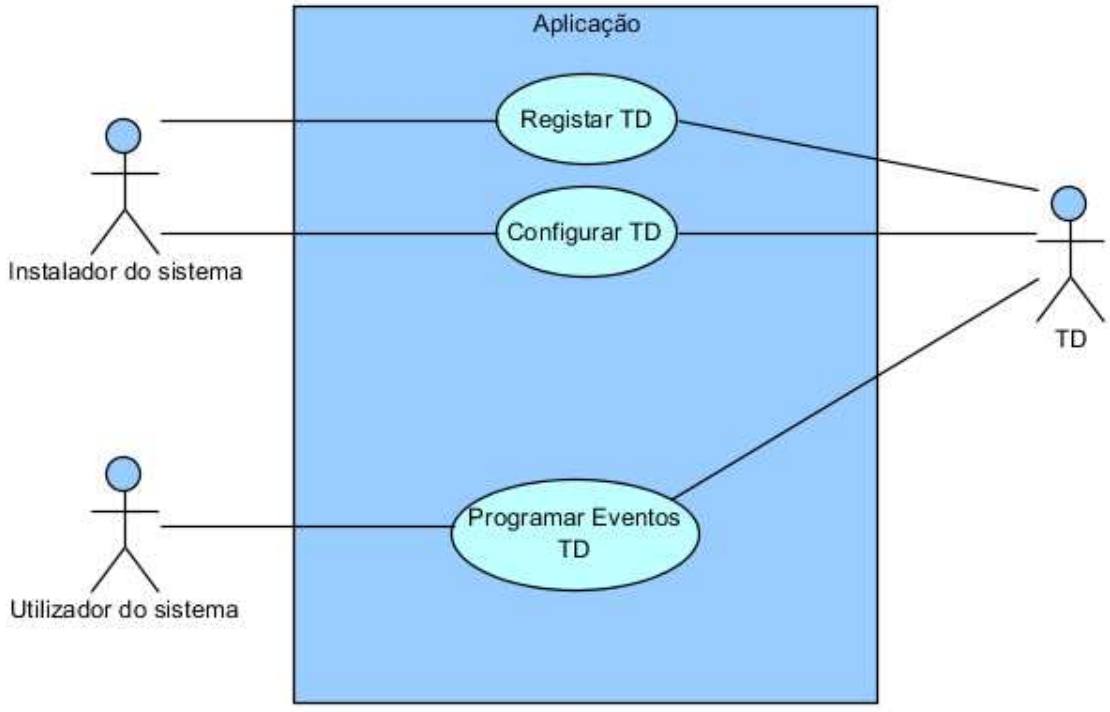


Figura 2.16 – Diagrama de Casos de Uso da Aplicação

Actores:

- Instalador do sistema – Pessoa que instala o sistema de domótica na casa, e que faz a sua gestão.

- Utilizadores do sistema – Todas as pessoas que estiverem dentro de casa.

- TD – Unidade de controlo inteligente colocada junto a cada equipamento eléctrico.

Descrição dos Casos de Uso

Registar TD	
Pré-condição	A aplicação e o novo TD têm de estar ligados na rede.
Descrição:	**1.** O caso de uso começa quando o instalador do sistema selecciona a opção Registar TD. **2.** Em seguida aparecem todos os TDs que já estão registados, com os seus endereços. **3.** O instalador dá um nome ao novo TD e um endereço (nome e endereço diferentes dos existentes). **4.** Constroi a mensagem de registo. **5.** Envia a mensagem de registo para TD. **6.** Fica à espera da resposta. **7.** Quando recebe a resposta actualiza na sua base de dados a informação do novo TD. O instalador tem a confirmação que o TD foi registado.
Pós-condição	É necessário guardar as alterações no ficheiro.

Quadro 2.11 – Descrição do Caso de Uso *Registar TD*

Configurar TD	
Pré-condição	O TD tem de estar registado.
Descrição:	**1.** O caso de uso começa quando o instalador do sistema selecciona a opção Configurar TD. **2.** O instalador escolhe qual o TD que pretende configurar. **3.** Selecciona as entradas que quer ler e quais são as condições que elas desencadeiam. **4.** Selecciona as saídas que quer controlar e quais são as condições que as desencadeiam. **5.** Constroi a mensagem de configuração. **6.** Envia a mensagem de configuração para TD. **7.** Fica à espera da resposta. **8.** Quando recebe a resposta o instalador tem a confirmação que o TD foi configurado.
Pós-condição	

Quadro 2.12 – Descrição do Caso de Uso *Configurar TD*

Programar eventos	
Pré-condição	O TD tem de estar registado.
Descrição:	**1.** O caso de uso começa quando o utilizador do sistema selecciona a opção Programar eventos. **2.** O instalador escolhe qual o TD que pretende programar. **3.** Selecciona as saídas que quer controlar e qual é a data e a hora em que as saídas vão tomar o valor escolhido. **4.** Constroi a mensagem de programação. **5.** Envia a mensagem de programação para TD. **6.** Fica à espera da resposta. **7.** Quando recebe a resposta o instalador tem a confirmação que o TD tem os eventos programados.
Pós-condição	

Quadro 2.13 – Descrição do Caso de Uso *Programar Eventos*

Descrição do funcionamento da aplicação

Apresenta-se de seguida uma breve descrição do funcionamento da aplicação. Para uma informação mais pormenorizada, sugere-se a consulta de [Moutinho, 04].

Tal como já mencionado, a aplicação permite gerir os registos, configurações e programação de eventos.

Só o instalador do sistema poderá registar ou configurar TDs, para tal é necessário introduzir a palavra pass (Figura 2.17). No caso dos eventos, estes podem ser geridos pelo utilizador, não necessitando a introdução de palavra pass.

Figura 2.17 – Janela de password da aplicação

Registar

É na página *Registar* que se enviam as mensagens de registo para os TDs. Para registar um TD é necessário introduzir o nome e o endereço, e depois carrega-se no botão *Regista TD.* Para retirar o TD da rede escolhe-se o TD da lista e carrega-se no botão *Retira o TD da rede*.

Quando se carrega num dos botões, é construída a mensagem e enviada para o módulo de ligação do computador à rede. Este envia a mensagem para o TD assim que a rede se encontre desocupada.

Na figura 2.18 mostra-se a janela responsável pelo registo.

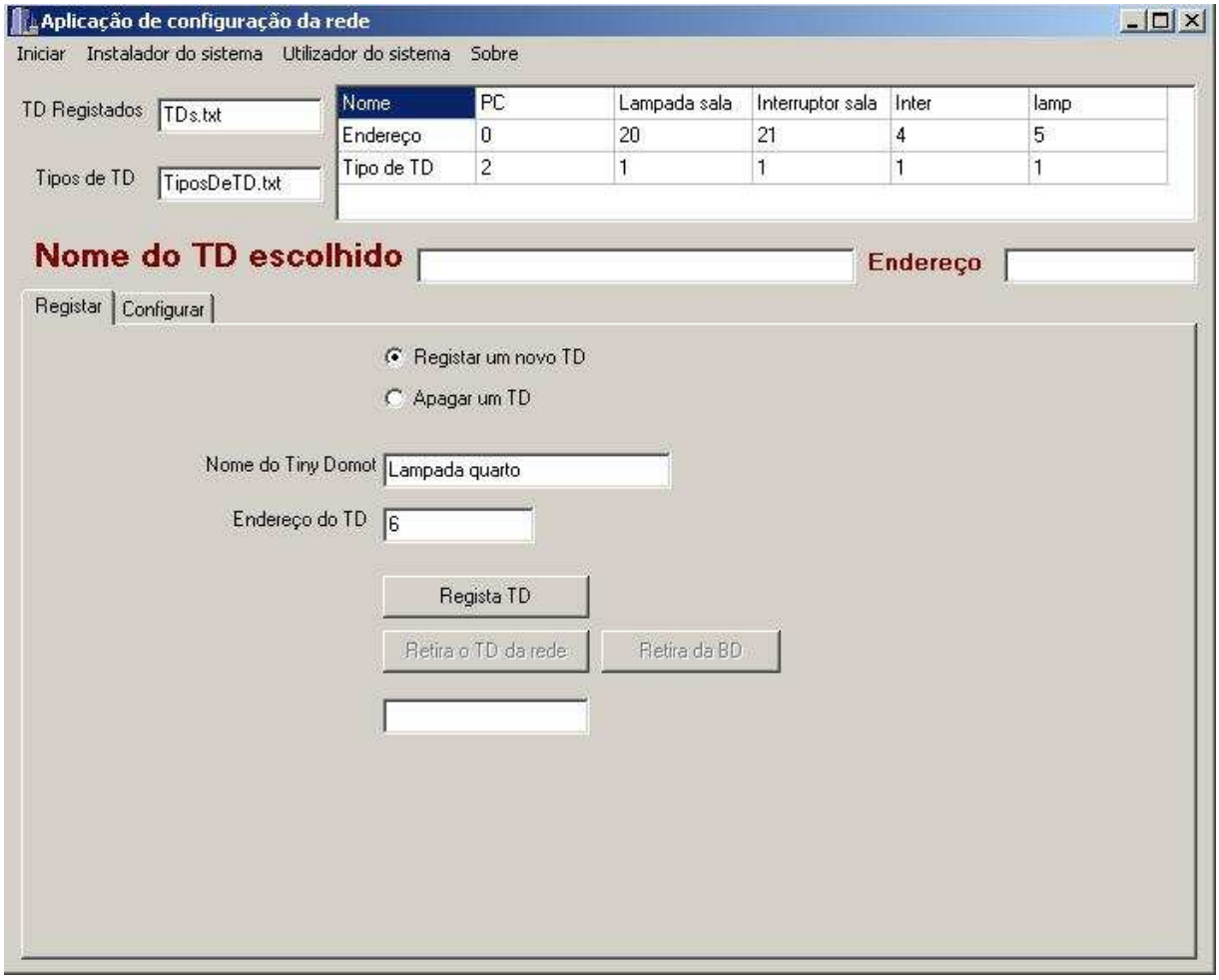


Figura 2.18 – Janela de registo da aplicação

Configurar

Para configurar um TD é necessário escolhe-lo da lista que se encontra no topo da aplicação.

Depois na página C*onfigurar* (Figura 2.19), selecciona-se, qual a entrada ou saída a utilizar.

No caso de ser uma entrada, é necessário escolher os destinatários da mensagem. No caso de ser uma saída, é necessário escolher os emissores das mensagens, e os acontecimentos associados.

É possível também fazer um pedido ao TD para visualizar a sua configuração actual assim como apagá-la.

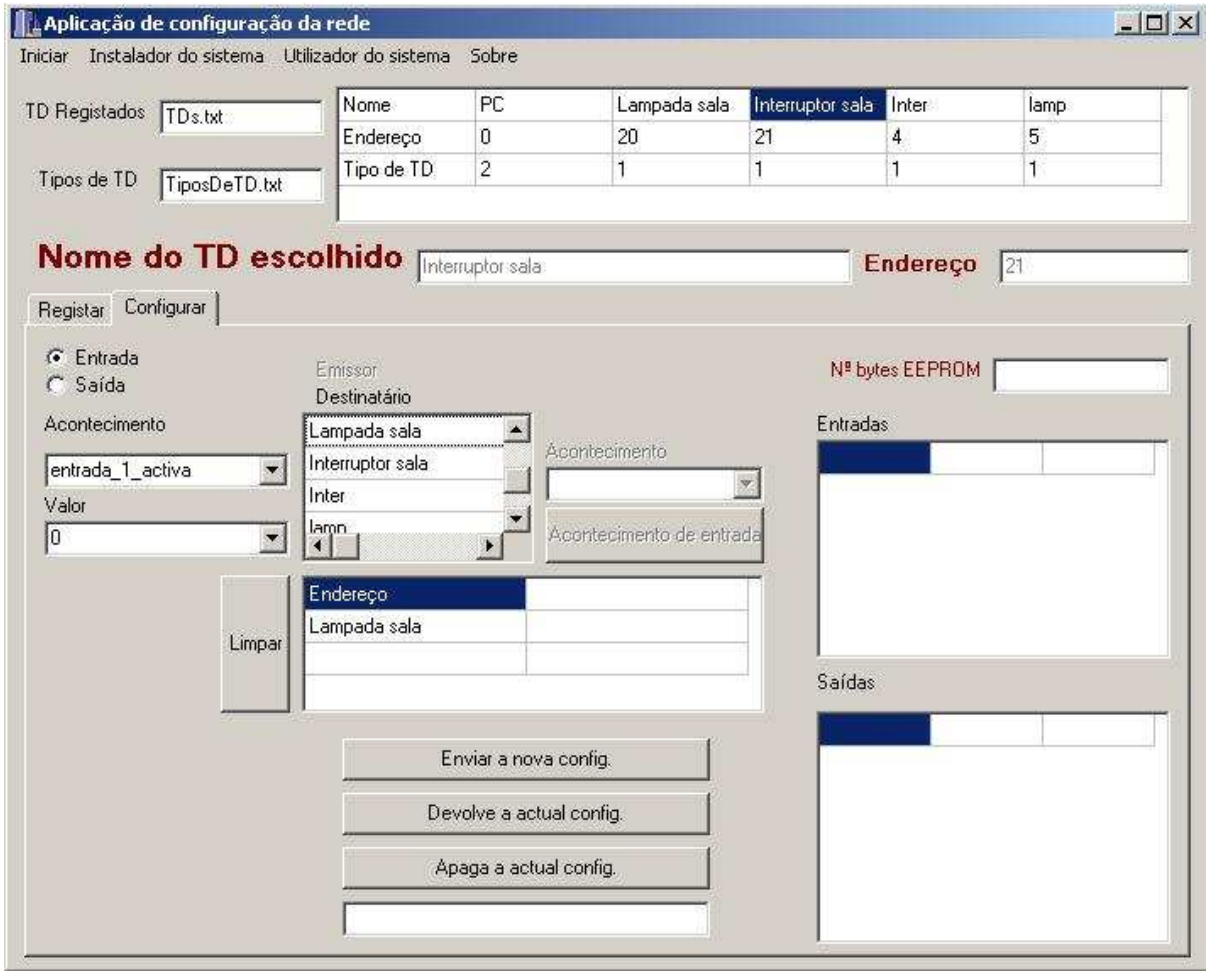


Figura 2.19 – Janela de configuração da aplicação

Programar eventos

Tal como para configurar um TD, para programar eventos também é necessário escolher o TD da lista.

Na página *Configurar eventos* (Figura 2.20) começa-se por escolher o tipo de evento, se é único ou se é semanal. Se for semanal escolhe-se os dias da semana em que ocorre.
Depois escolhe-se a data e a hora do evento e por fim envia-se o evento para o TD.

Seguindo a mesma lógica como em *Configurar,* aqui também é possível fazer um pedido ao TD para visualizar a lista de eventos programados assim como apagá-los.

É também na página *Configurar eventos* que se pode actualizar e ler a hora dos TD's.

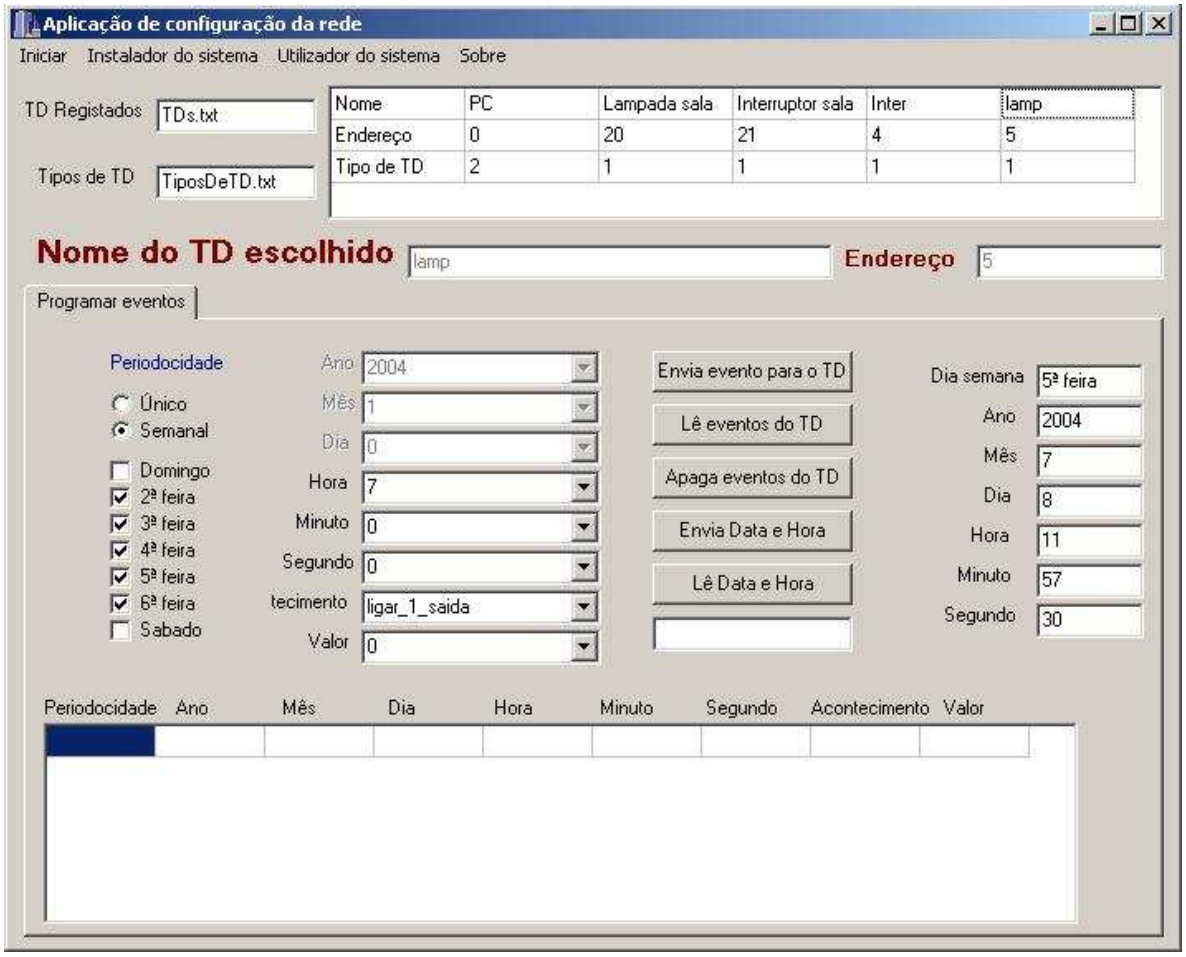


Figura 2.20 – Janela de programação de eventos da aplicação

3. Contribuições

Através de uma primeira análise aos elementos existentes, foi detectado uma ligeira incoerência entre o código C e os diagramas de actividade. Desta forma procurou-se, através de um processo de "reverse engineering" [Chikofsy & Cross II, 90], obter uma representação do código fonte existente expresso em C num nível de abstracção mais elevado, nomeadamente através de Diagramas de Estados e posteriormente através de redes de Petri IOPT.

Durante este processo, procurou-se ainda melhorar o comportamento do sistema e simplificar determinadas operações / comportamentos que pareceram pertinentes. Ao mesmo tempo procurou-se manter um nível de abstracção constante em toda a modelação do sistema com vista a encontrar um modelo comportamental que seja de fácil compreensão e que possibilite a sua reutilização.

Uma outra parte deste processo com bastante influência na definição do nível de abstracção consiste na selecção de quais as funções a ser transportadas para a nova implementação. Pretendeu-se efectuar apenas uma modelação comportamental do sistema, sendo que esta base comportamental reutilizará funções já existentes (eventualmente algumas delas após ligeiras adaptações).

Desta forma, o modelo invocará as referidas funções através de sinais de saída associados às mesmas. Por exemplo, quando é activado o sinal de saída "APAGA_REGISTO" é invocada a função "Apaga_Registo()", que por sua vez acede à memória do microprocessador, apaga o registo, actualiza várias variáveis necessárias à caracterização do TD como *não registado* e por fim envia a mensagem de *acknowledge* para a aplicação.

3.1. Modelo comportamental - Diagramas de Estados

Apresentam-se de seguida os modelos comportamentais encontrados, expressos em Diagramas de Estados.

Todos os estados identificados como "Aux" são considerados estados auxiliares, ou seja, não contemplam nenhuma actividade.

Diagrama de Estados principal (Main) do TD

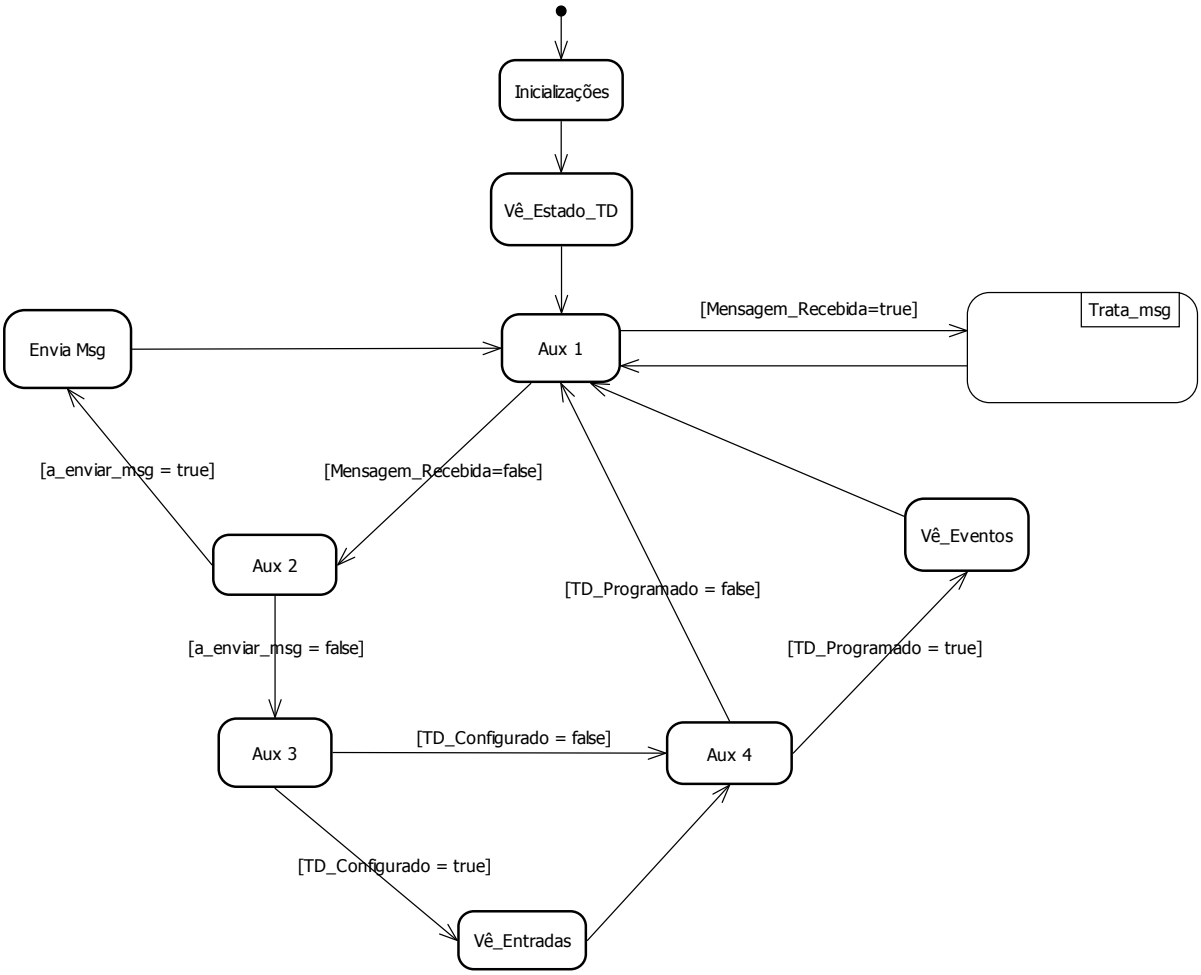


Figura 3.1 – Diagrama de Estados principal

A título de exemplo, um dos aspectos melhorados, está relacionado com a verificação de eventos. No código fonte existente, em cada ciclo, seria sempre analisada a existência de eventos programados para aquele instante e desencadeá-lo em caso afirmativo. Desta forma, foi adicionada uma variável com a informação se o TD possui ou não algum evento programado, caso não tenha, não existe a necessidade de análise da lista em cada ciclo.

Diagrama de Estados Trata Mensagens

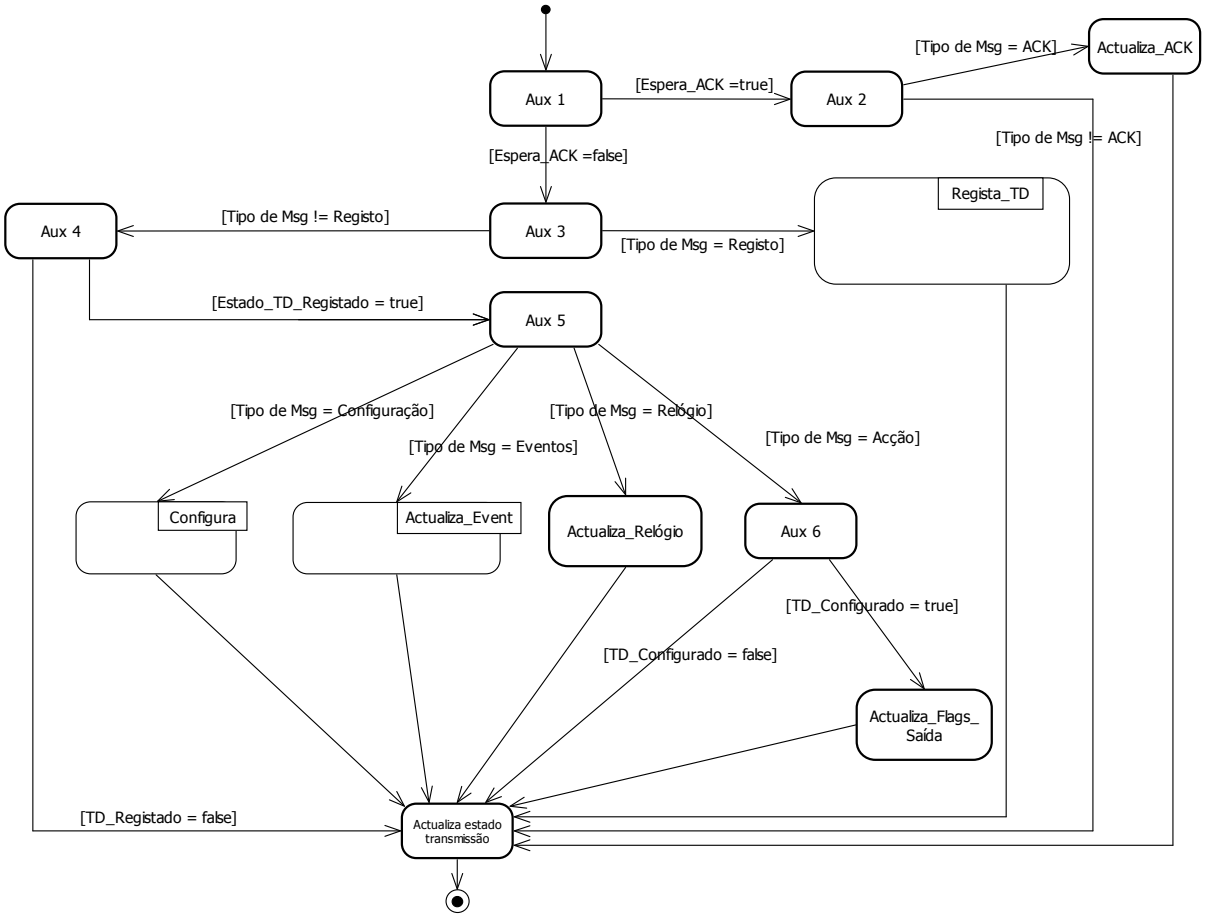


Figura 3.2 – Diagrama de Estados *Trata Mensagens*

Neste caso foram introduzidos alguns melhoramentos significativos, nomeadamente:

- a verificação do CRC deixa de ser feita neste módulo e passa a ser feita imediatamente após a recepção da mensagem (ainda no módulo de recepção), caso o CRC não esteja correcto, a mensagem é simplesmente ignorada não havendo a necessidade de esta ser processadas várias vezes.

- a verificação se o TD está ou não registado é feita num ponto comum antes da verificação de qual o tipo de mensagem recebido em vez de ser sempre confirmado em todos os pontos desta verificação.

Diagrama de Estados Regista

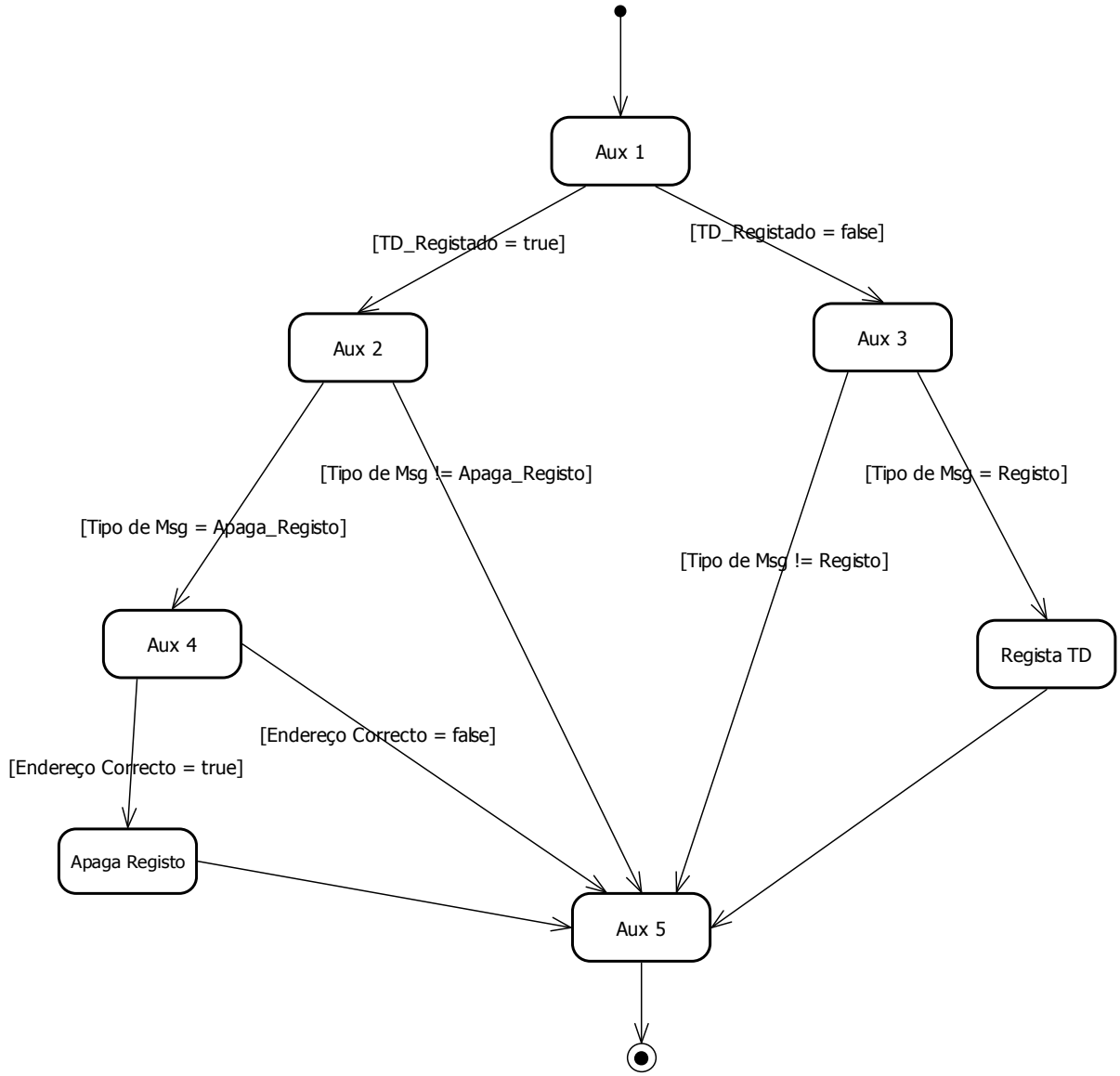


Figura 3.3 – Diagrama de Estados *Regista*

No módulo *Regista* , representado na Figura 3.3, caso a mensagem recebida seja a de apagar registo vai apenas verificar a variável(Input Signal) *Endereço_Correcto* para decidir se apaga ou não o registo. Esta variável é afectada logo após a recepção da mensagem, ou seja, se a mensagem não é para este TD, esta é automaticamente ignorada não existindo a necessidade da mensagem ser verificada mais à frente no módulo *Regista* poupando assim tempo de processamento.

Diagrama de Estados Configura

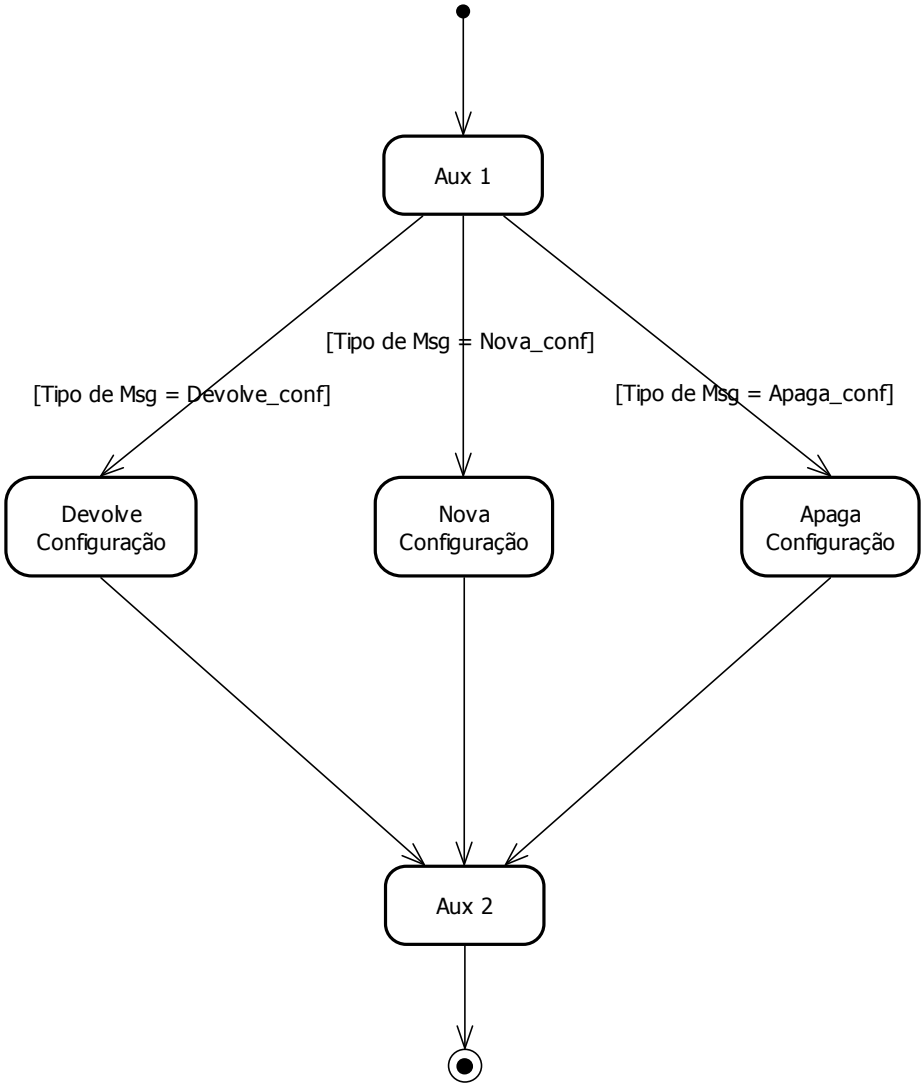


Figura 3.4 – Diagrama de Estados *Configura*

Tal como o nome indica, este módulo processa as mensagens de configuração, quer estas sejam de nova configuração, apagar configuração ou devolver a actual configuração.

Diagrama de Estados Actualiza Eventos

O seu comportamento é idêntico ao módulo *Configura,* no entanto, aplicado a eventos (programação de novos eventos, apagar eventos existentes e devolver lista de eventos).

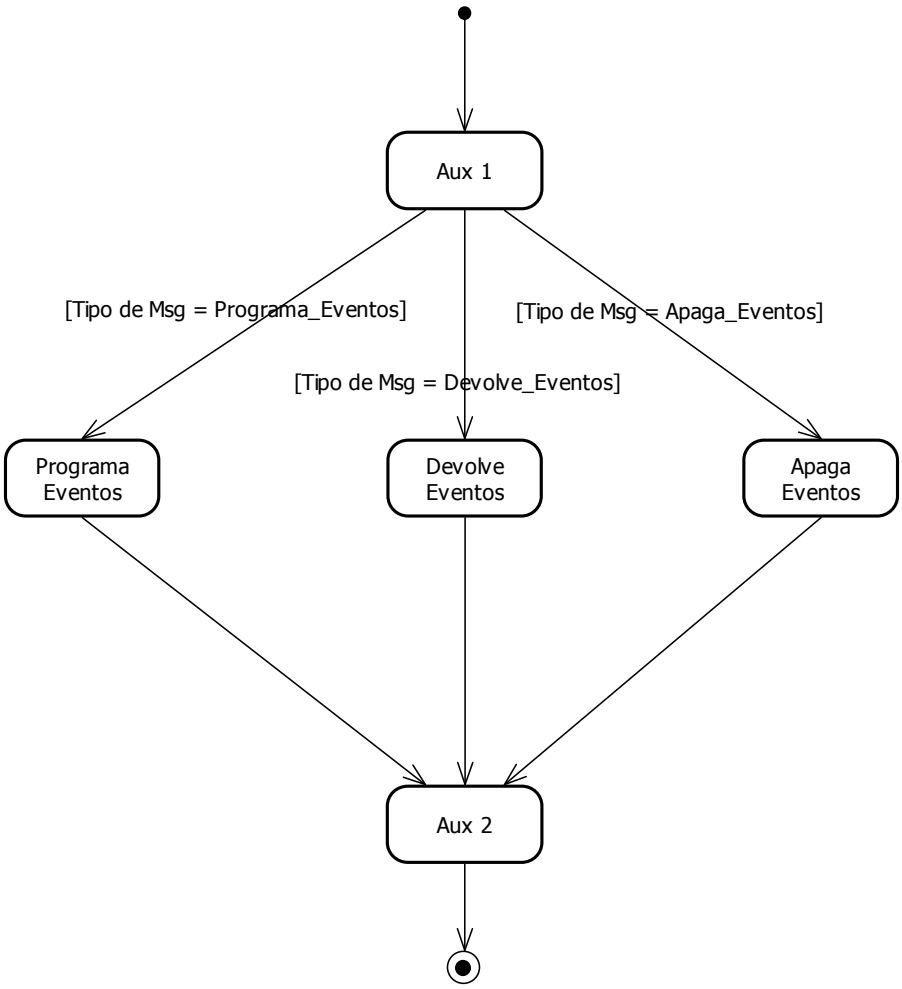


Figura 3.5 – Diagrama de Estados *Actualiza Eventos*

3.2. Modelo comportamental - Rede IOPT

Tradução entre Diagramas de Estados e RdP IOPT

Uma das classes de RdP conhecidas é a Máquina de Estados [Peterson, 77], onde cada transição tem exactamente um arco de entrada e um arco de saída. Uma máquina de estados permite a representação de decisões (conflitos) mas não permite representar a sincronização de actividades paralelas.

Uma vez que no modelo encontrado expresso em Diagramas de Estados não constam actividades paralelas sendo a sua evolução sequencial, baseada em decisões impostas por condições de guarda, a equivalência entre os diagramas de estados e redes de Petri IOPT (neste caso assumindo características de máquina de estados) é quase directa, o que permite facilmente converter o sistema expresso num formalismo de modelação para outro, neste caso, converter de diagramas de estados para redes de Petri IOPT.

Como tal, foi adoptada uma regra, indicada na Figura 3.6 para efectuar a esta tradução.

Do ponto de vista deste trabalho, é de interesse identificar um conjunto de procedimentos de tradução que permitem a obtenção de modelos expressos em RdP IOPT a partir de Diagramas de Estados. Desta forma, torna-se possível partir de representações do código fonte, chegar a uma representação através de Diagramas de Estados e convertê-los em modelos expressos em RdP utilizáveis directamente pela ferramenta de geração automática de código que se pretende utilizar.

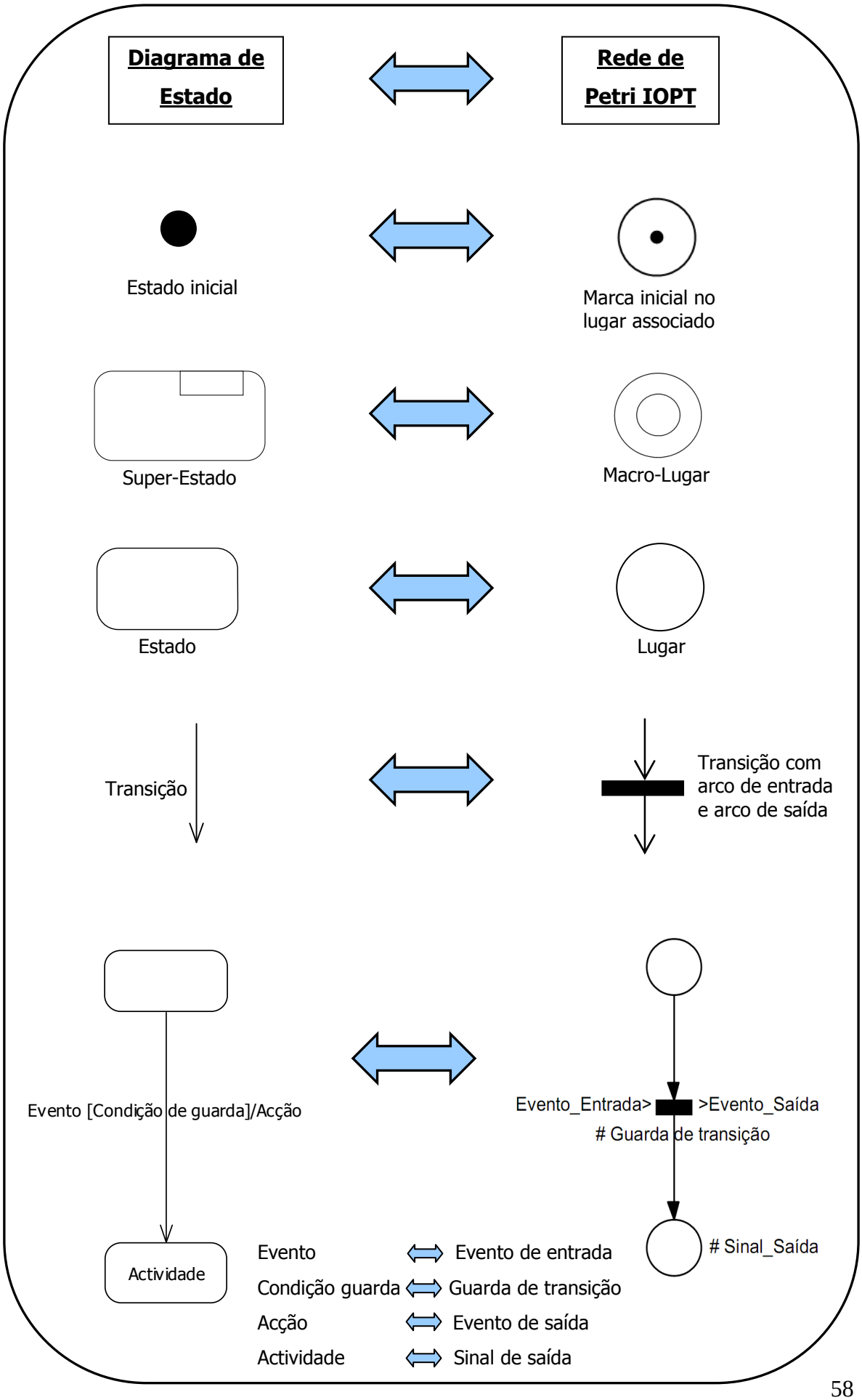


Figura 3.6 - Equivalência entre Diagramas de Estados e RdP IOPT

Na Figura 3.7, temos um exemplo de equivalência entre Diagramas de Estados e redes de Petri IOPT

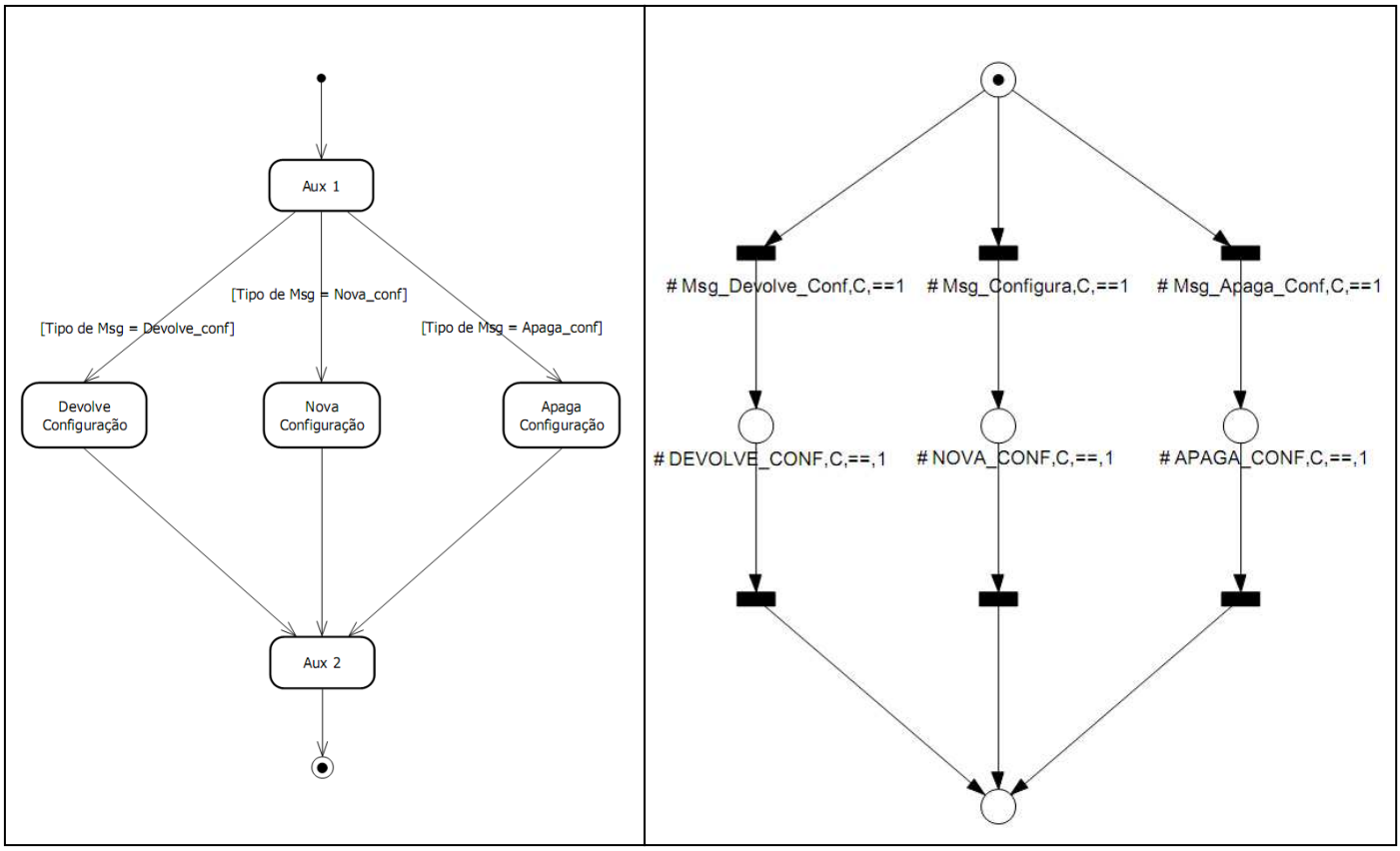


Figura 3.7 – Exemplo de equivalência entre Diagramas de Estados e RdP IOPT

Com a não modelação total do sistema, criou-se, para além das saídas que necessitam de existir no sistema descrito anteriormente, umas saídas adicionais que invocarão determinadas funções.

RdP IOPT principal(Main) do TD

O primeiro modelo apresentado, na Figura 3.8, é o modelo do topo da hierarquia, isto é, o modelo principal. Este refere o sub-modelo *TRATA_MSG* representado por um macrolugar que será analisado de forma mais detalhada, no ponto seguinte.

O modelo obtido resulta da tradução do diagrama de estados da Figura 3.1.

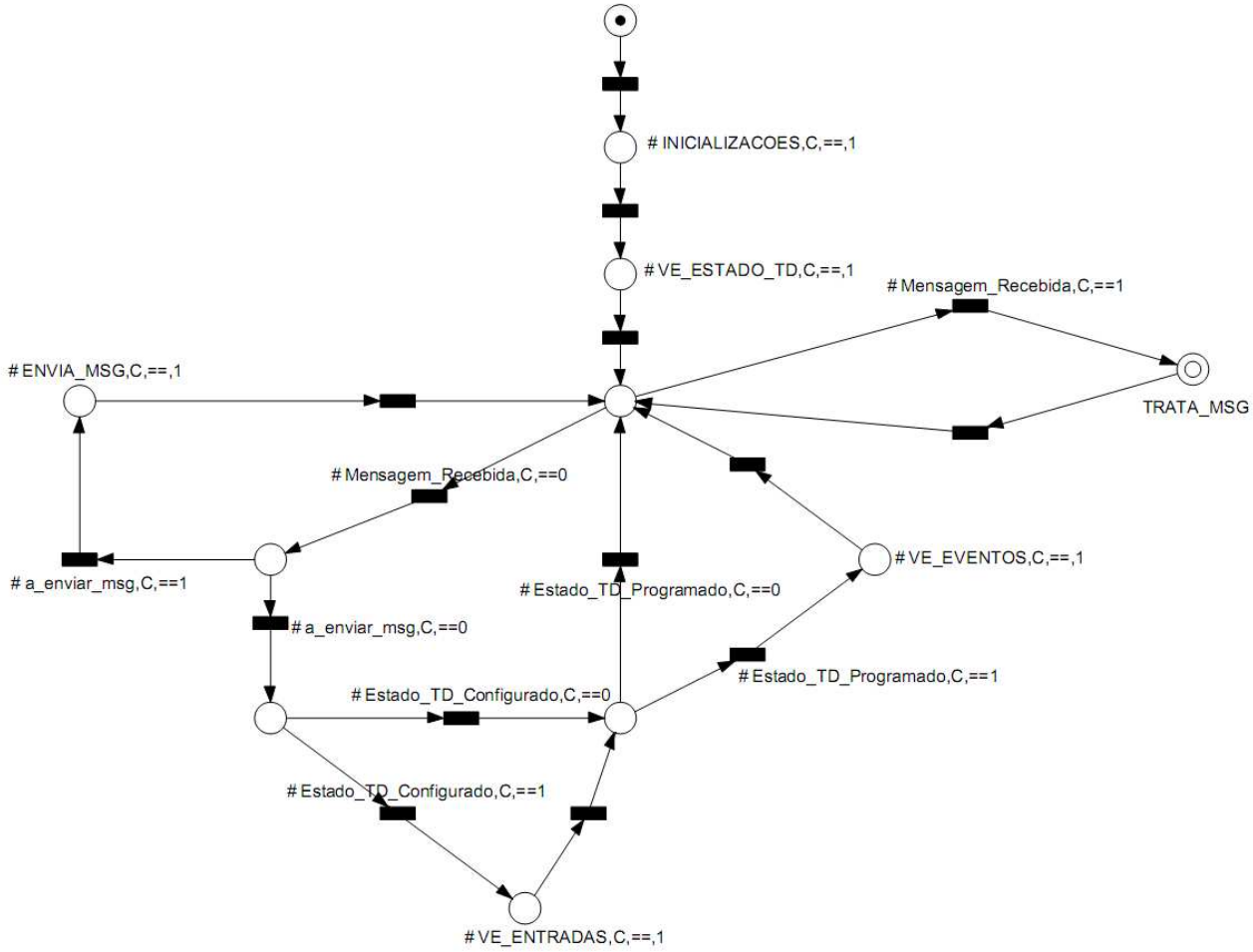


Figura 3.8 – Rede de Petri IOPT *Principal*

RdP IOPT Trata Mensagens

No modelo apresentado na Figura 3.9, de acordo com o tipo de mensagem recebida e o estado do TD, vai desencadear as respectivas funções ou sub-modelos associados. Este modelo resulta da tradução do diagrama de estados da Figura 3.2.

Para tratar o tipo de mensagem foi criado um sinal de entrada designado *Tipo_Msg* do tipo *Range*, desta forma, este sinal tem a capacidade de assumir vários valores. De acordo com o tipo de mensagem recebido este sinal vai assumir um valor distinto para cada um deles. Apresenta-se no Anexo 1 a listagem dos sinais utilizados na construção do modelo geral, assim como um quadro com a correspondência entre o valor do sinal *Tipo_Msg* e a tipo de mensagem recebido pelo TD.

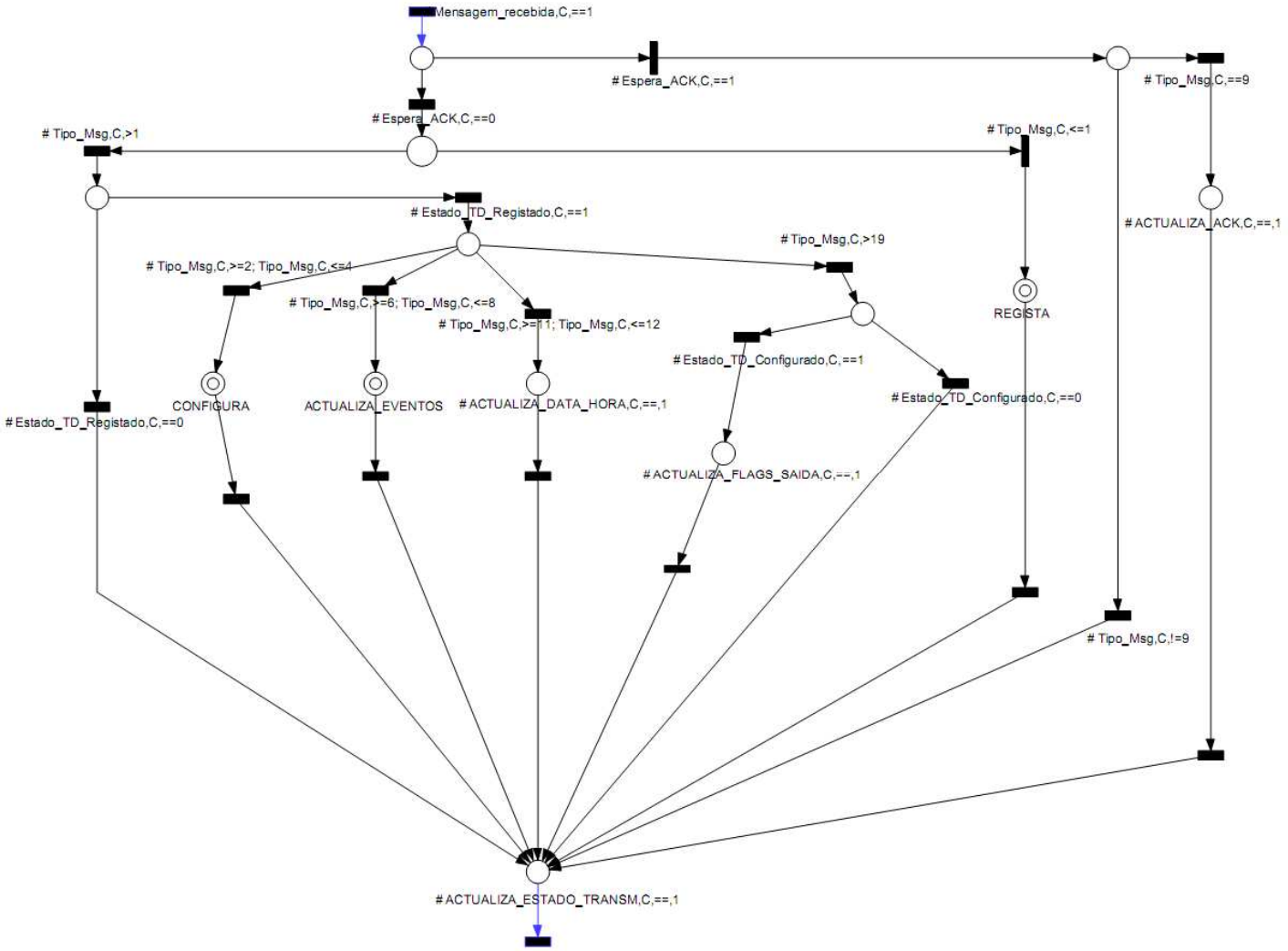


Figura 3.9 – Rede de Petri IOPT *Trata Mensagens*

RdP IOPT Regista

O modelo apresentado na Figura 3.10 é responsável pela parte de registo do TD. É aqui que é verificado se existem condições para se efectuar o registo ou mesmo apagá-lo, de acordo com o pretendido. Caso existam condições, são executados os pedidos.

Este modelo foi obtido através da tradução do diagrama de estados da Figura 3.3.

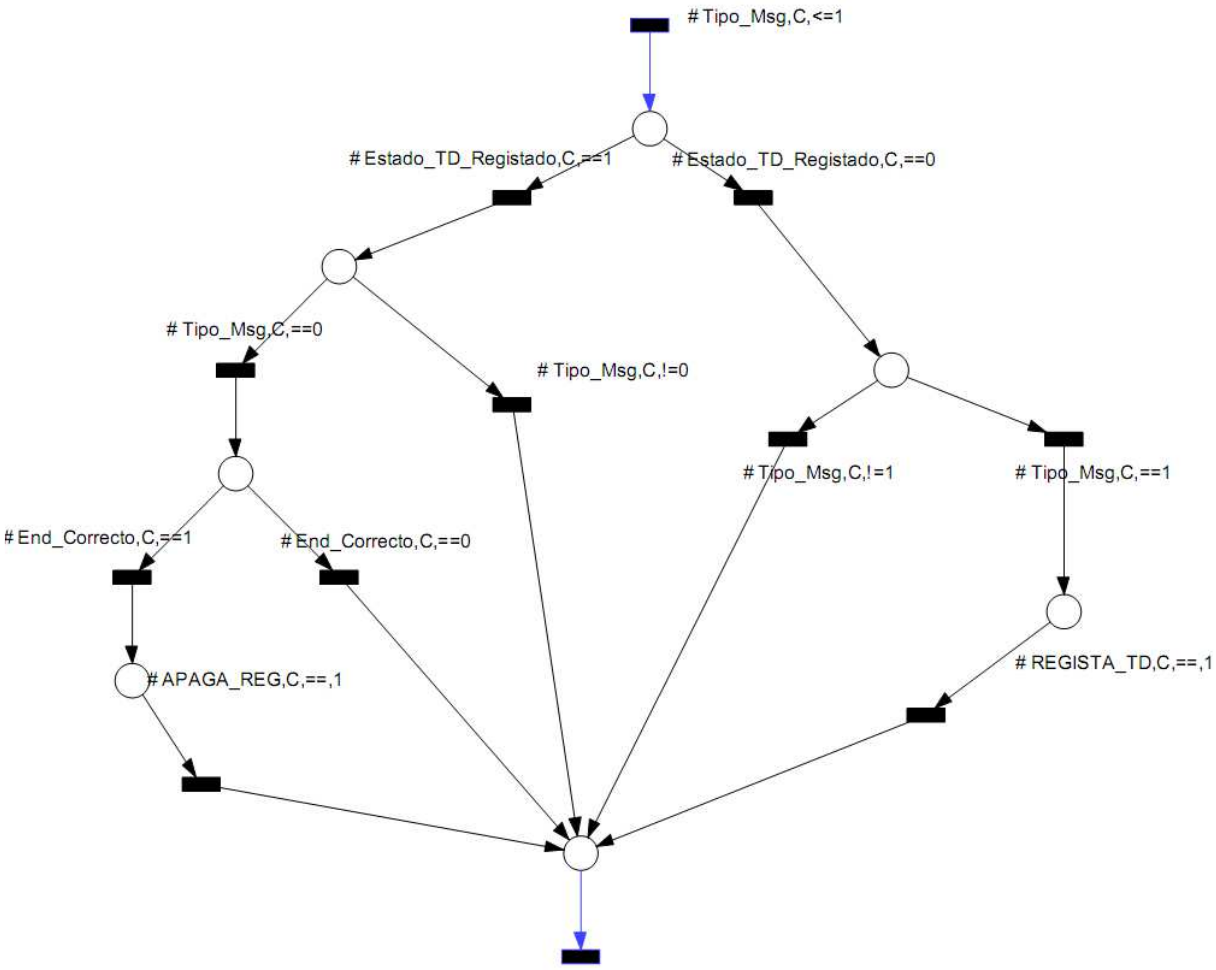


Figura 3.10 – Rede de Petri IOPT *Regista*

RdP IOPT Configura

Na Figura 3.11 encontra-se representado o modelo responsável pela configuração do TD, quer seja para processar uma nova configuração, devolver a configuração existente ou apagar a configuração actual.

Este modelo foi obtido através da tradução do diagrama de estados da Figura 3.4.

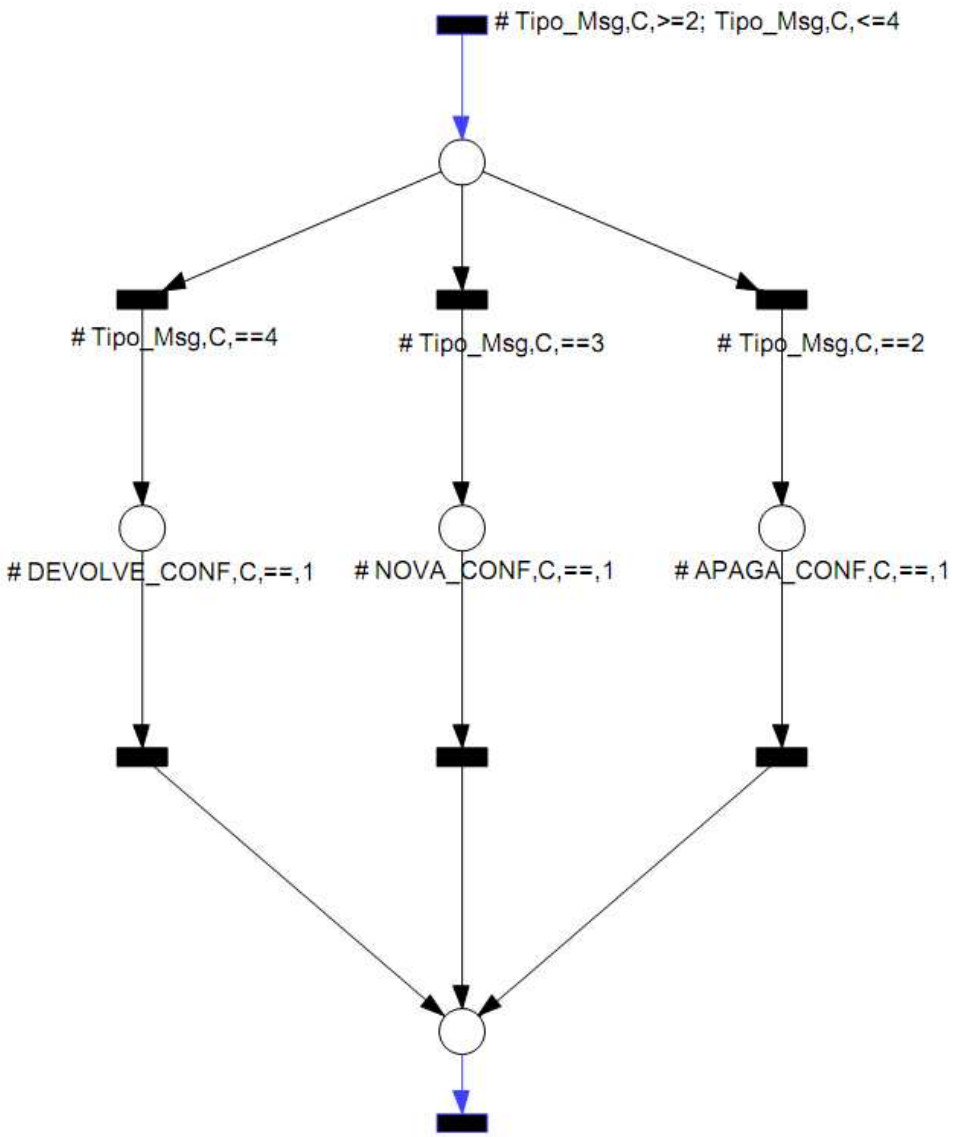


Figura 3.11 – Rede de Petri IOPT *Configura*

RdP IOPT Actualiza Eventos

Mostra-se na Figura 3.12, o modelo responsável pela gestão de eventos do TD, o seu comportamento é idêntico ao modelo *Configura,* no entanto, aplicado a eventos (programação de novos eventos, apagar eventos existentes e devolver lista de eventos).

Este modelo foi obtido pela tradução do diagrama de estados da Figura 3.5.

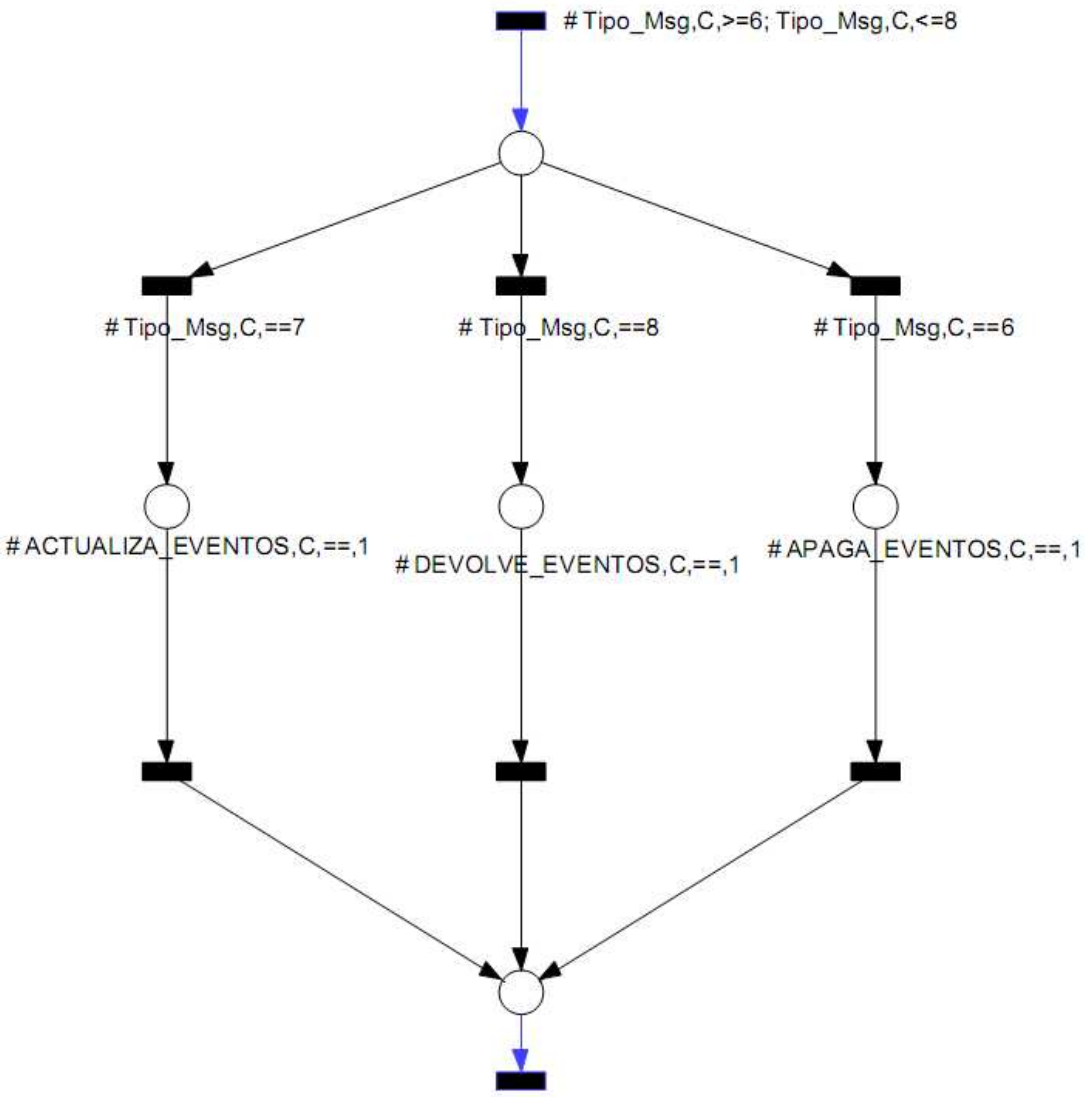


Figura 3.12 – Rede de Petri IOPT *Actualiza Eventos*

3.3. Código gerado automaticamente

O editor de redes de Petri Snoopy IOPT gera, para além do ficheiro IOPT, um ficheiro PNML. Este último pode ser carregado pela ferramenta de geração automática PNML2C que gera cinco ficheiros em linguagem C: *main.c, functionsc.c, netc.c, functionsc.h e netc.h*.

O ficheiro *netc* é responsável pela gestão/evolução da RdP.

Neste caso, a título de exemplo, apresenta-se na Figura 3.13 apenas três sinais de entrada, três sinais de saída e quatro lugares uma vez que a estrutura se mantém para os restantes sinais.

```
#include "netc.h"

char start(void)
{
//----------------Init Input Signals Values------------
      Mensagem_Recebida = 0;
      Estado_TD_Registado = 0;
      Estado_TD_Configurado = 0;
//-----------------------------------------------------

//----------------Init Output Signals Values-----------
      ENVIA_MSG = 0;
      VE_ENTRADAS = 0;
      VE_EVENTOS = 0;
//-----------------------------------------------------

//----------------Init Input Events Values-------------
//-----------------------------------------------------

//----------------Init Output Events Values------------
//-----------------------------------------------------

//----------------Init Place Marks---------------------
      p_184_Mark = 0; // Place
      p_212_Mark = 0; // Place
      p_226_Mark = 0; // Place
      p_240_Mark = 0; // Place
//-----------------------------------------------------

      return 1;
}

char run(void)
{

      new_p_184_Mark = 0; // Place
      new_p_212_Mark = 0; // Place
      new_p_226_Mark = 0; // Place
      new_p_240_Mark = 0; // Place

      aux_p_184_Mark = p_184_Mark; // Place
      aux_p_212_Mark = p_212_Mark; // Place
      aux_p_226_Mark = p_226_Mark; // Place
      aux_p_240_Mark = p_240_Mark; // Place

      //Save Output Signals
      last_ENVIA_MSG = ENVIA_MSG;
      last_VE_ENTRADAS = VE_ENTRADAS;
      last_VE_EVENTOS = VE_EVENTOS;

      // Reset Output Events

       Analyse_Input_Events();
```

Figura 3.13. a) – Ficheiro *netc.c* gerado de forma automática. Completado pela Figura 3.13.b)

```
        // Transition 318 (  )
        if(aux_p_184_Mark >= 1 && (Mensagem_Recebida==0))
         {
            aux_p_184_Mark = aux_p_184_Mark - 1;
            new_p_212_Mark = new_p_212_Mark + 1;
        }

        // Transition 336 (  )
        if(aux_p_212_Mark >= 1 && (a_enviar_msg==1))
         {
            aux_p_212_Mark = aux_p_212_Mark - 1;
            new_p_226_Mark = new_p_226_Mark + 1;
        }

        // Transition 354 (  )
        if(aux_p_226_Mark >= 1)
         {
            aux_p_226_Mark = aux_p_226_Mark - 1;
            new_p_184_Mark = new_p_184_Mark + 1;
        }

        // Transition 372 (  )
        if(aux_p_212_Mark >= 1 && (a_enviar_msg==0))
         {
            aux_p_212_Mark = aux_p_212_Mark - 1;
            new_p_240_Mark = new_p_240_Mark + 1;
        }


        // Actualize Marks
        p_184_Mark = aux_p_184_Mark + new_p_184_Mark;
        p_212_Mark = aux_p_212_Mark + new_p_212_Mark;
        p_226_Mark = aux_p_226_Mark + new_p_226_Mark;
        p_240_Mark = aux_p_240_Mark + new_p_240_Mark;


        //Save Input Signals
        last_Mensagem_Recebida = Mensagem_Recebida;
        last_Estado_TD_Registado = Estado_TD_Registado;
        last_Estado_TD_Configurado = Estado_TD_Configurado;

        // Reset Input Events

        ENVIA_MSG = 0;
        VE_ENTRADAS = 0;
        VE_EVENTOS = 0;

            if(p_226_Mark >= 1)
              {
                  if(ENVIA_MSG < 1)
                  ENVIA_MSG = 1;
              }


        Analyse_Output_Events();

       return 1;
}
```

Figura 3.14. b) – Continuação do ficheiro *netc.c* da Figura 3.13.a)

No ficheiro *funtionsc,* apresentado na Figura 3.15, encontram-se as funções disponíveis para alterar os sinais de entrada e as funções que permitem obter o valor actual dos sinais de saída. Neste caso apresenta-se também apenas três sinais de entrada, três sinais de saída visto que a estrutura se mantém para os restantes sinais.

```
#include "functionsc.h"

//Input Signal
void Set_Mensagem_Recebida (char val)
{
   last_Mensagem_Recebida = Mensagem_Recebida;
   Mensagem_Recebida = val;
}

//Input Signal
void Set_Estado_TD_Registado (char val)
{
   last_Estado_TD_Registado = Estado_TD_Registado;
   Estado_TD_Registado = val;
}

//Input Signal
void Set_Estado_TD_Configurado (char val)
{
   last_Estado_TD_Configurado = Estado_TD_Configurado;
   Estado_TD_Configurado = val;
}



//Output Signal
char Get_ENVIA_MSG (void)
{
   return ENVIA_MSG;
}

//Output Signal
char Get_VE_ENTRADAS (void)
{
   return VE_ENTRADAS;
}

//Output Signal
char Get_VE_EVENTOS (void)
{
   return VE_EVENTOS;
}



void Analyse_Input_Events(void)
{
}

void Analyse_Output_Events(void)
{
}
```

Figura 3.15 – Ficheiro *functionsc.c* gerado de forma automática

De salientar que no caso de serem utilizados eventos, quer de entrada, quer de saída, a declaração apareceria neste mesmo ficheiro, na zona reservada para tal. No entanto, no sistema modelado não se identificou a necessidade da utilização de eventos.

Por último, no ficheiro *main* consta a declaração de todas as variáveis necessárias ao funcionamento da RdP, tais como as variáveis associadas aos sinais e lugares assim como variáveis auxiliares. É neste ficheiro que deverá ser inserido o código produzido manualmente necessário ao funcionamento do sistema, tais como código correspondente à plataforma utilizada (PIC18F4620), funções e variáveis do código gerado manualmente.

Apresenta-se na Figura 3.16 o ficheiro *main* gerado automaticamente. A função *start* é responsável pela inicialização da RdP e a função *run* trata da evolução da RdP. Mais uma vez, considera-se o exemplo com apenas três sinais de entrada, três sinais de saída e quatro lugares.

```
#include <string.h>
#include "netc.h"

//----------------Input Signals Values---------------------
char Mensagem_Recebida;char last_Mensagem_Recebida;
char Estado_TD_Registado;char last_Estado_TD_Registado;
char Estado_TD_Configurado;char last_Estado_TD_Configurado;

//---------------------------------------------------------

//----------------Output Signals Values--------------------
char ENVIA_MSG;char last_ENVIA_MSG;
char VE_ENTRADAS;char last_VE_ENTRADAS;
char VE_EVENTOS;char last_VE_EVENTOS;

//---------------------------------------------------------

//----------------Input Events Values----------------------
//---------------------------------------------------------

//----------------Output Events Values---------------------
//---------------------------------------------------------

//---------------------Place Marks-------------------------
char p_184_Mark;char aux_p_184_Mark;char new_p_184_Mark;
char p_212_Mark;char aux_p_212_Mark;char new_p_212_Mark;
char p_226_Mark;char aux_p_226_Mark;char new_p_226_Mark;
char p_240_Mark;char aux_p_240_Mark;char new_p_240_Mark;

//---------------------------------------------------------


int main()
{
  start();

  //insert your code here...

  //Set Input Signals
  //Set Input Events

  run();

  //Get Output Signals
  //Get Output Events

  //insert your code here...

  return 1;
}
```

Figura 3.16 – Ficheiro *main.c* gerado de forma automática

3.4. Código Manual Utilizado

No ficheiro *main.c* gerado automaticamente, tal como indicado em comentário na função *main* (Figura 3.16), é aqui se serão alterados os valores dos sinais de entrada assim como a obtenção dos sinais de saída. Como tal, é inserido um ciclo infinito (while) para que em cada ciclo (em cada estado da RdP) seja feita a actualização e obtenção dos sinais indicados. Para tal, foram criadas as funções *InputData* e *OutputData*, respectivamente.

```
int main()
{
    start();

    PlatformDependentCode();

    while(1)
    {
        InputData();

        run();

        OutputData();
    }

return 1;
}
```

Figura 3.17 – Função *main* actualizada

Para além destas duas funções inseridas no ciclo infinito, foi ainda inserida a função *PlatformDependentCode* fora deste, pois apenas é necessário ocorrer uma vez. Nesta função é feita a configuração da plataforma utilizada (definição de entradas e saídas, configuração de *timers*, de *interrupts*, etc…), neste caso, o PIC18F4620 da Microchip. Desta forma, caso se pretenda aplicar o modelo numa outra plataforma, basta adaptar o conteúdo desta função. A função *PlatformDependentCode* é apresentada no Anexo 2.

Um exemplo das funções *InputData* e *OutputData*, apenas com três sinais de entrada e três sinais de saída, é apresentado nas Figura 3.18.

Utilizando este exemplo da função *InputData*, no caso da variável *a_enviar_mens* estar activa é consequentemente activada a entrada digital *a_enviar_msg* através da função *Set_a_enviar_msg,* disponibilizada no ficheiro *functionsc*.

```
void InputData (void)
{

    if (estado_TD == registado){
        Set_Estado_TD_Registado(1);
        Set_Estado_TD_Configurado(0);
    }

    if (estado_TD == configurado){
        Set_Estado_TD_Registado(1);
        Set_Estado_TD_Configurado(1);
    }

    if (a_enviar_mens == 1)
        Set_a_enviar_msg(1);
    else
        Set_a_enviar_msg(0);

}
```

```
void OutputData (void)
{

    if (ENVIA_MSG == 1) {
        envia_mensagens();
    }

    if (VE_ENTRADAS == 1) {
        ve_entradas();
    }

    if (VE_EVENTOS == 1) {
        ve_eventos();
    }

}
```

Figura 3.18 – Exemplo de função *InputData* e *OutputData* criadas manualmente

Quando um sinal de saída está activo, este irá desencadear uma determinada função. É isso que é possível observar na função *OutputData*, por exemplo: se o sinal de saída *ENVIA_MSG* estiver activo, este invocará a função *envia_mensagens*.

É igualmente isto que se passa com os restantes sinais de saída. Apresenta-se na Figura 3.19 a função *OutputData* completa onde é possível visualizar as funções associadas aos vários sinais.

Estas são as funções recuperadas da versão existente, algumas delas na sua integra, outras com ligeiras adaptações, como é o caso da *actualiza_eventos* existente (ver Figura 3.20). Esta função seria genérica, sempre que surgisse alguma situação relacionada com eventos (novo evento, apaga eventos, devolve evento) esta função seria invocada e já dentro dessa mesma função era feita a decisão sobre qual a operação a efectuar, como podemos ver na Figura 3.20. Neste caso, esta função deu origem a três funções distintas: *actualiza_eventos*, *apaga_eventos* e *devolve_eventos*. Desta forma, estas serão invocadas independentemente de acordo com a evolução da RdP. Podemos ver estas três funções na Figura 3.21.

```
void OutputData (void)
{
    if (INICIALIZACAO == 1)
        inicializacao();

    if (VE_ESTADO_TD == 1)
        vai_ver_se_estado_TD();

    if (ENVIA_MSG == 1)
        envia_mensagens();

    if (VE_ENTRADAS == 1)
        ve_entradas();

    if (VE_EVENTOS == 1)
        ve_eventos();

    if (REGISTA_TD == 1)
        Regista_TD();

    if (APAGA_REG == 1)
        Apaga_Registo();

    if (NOVA_CONF == 1)
        configura_TD();

    if (DEVOLVE_CONF == 1)
        devolve_configuracao();

    if (APAGA_CONF == 1)
        apaga_config();

    if (ACTUALIZA_EVENTOS == 1)
        actualiza_eventos();

    if (DEVOLVE_EVENTOS == 1)
        devolve_eventos();

    if (APAGA_EVENTOS == 1)
        apaga_eventos();

    if (ACTUALIZADA_DATA_HORA == 1)
        actualiza_data_hora();

    if (ACTUALIZA_FLAGS_SAIDA == 1)
        actualiza_flags_saida();

    if (ACTUALIZA_ACK == 1)
        set_espera_ack_0();

    if (ACTUALIZA_ESTADO_TRANSM == 1)
        set_estado_transm_esp_pre();
}
```

Figura 3.19 - Função *OutputData* criada manualmente

```
void actualiza_eventos() {
  unsigned char j, n_b_EEPROM_old, n_b_EEPROM_new;
  n_b_EEPROM_old = ler_eeprom(pos_inicial_eventos);

  if (bytesrec[6] == 6) {//Apaga eventos
    colocaEEprom(pos_inicial_eventos, 0);
    envia_acknowledge_pc();
  }

  else if (bytesrec[6] == 8) {//Devolve eventos
    unsigned char j, numbytesx;
    numbytesx = 96;
    bytesenv[0]=bytesrec[0];
    bytesenv[1]=numbytesx;
    bytesenv[2]=bytesrec[4];
    bytesenv[3]=bytesrec[5];
    bytesenv[4]=bytesrec[2];
    bytesenv[5]=bytesrec[3];
    bytesenv[6]=bytesrec[6];
    for (j=0;j<(numbytesx-7);j++) {
      bytesenv[7+j] = ler_eeprom(j+pos_inicial_eventos);
    }
    bytesenv[numbytesx-1]=calcula_crc(numbytesx, 1);
    envia_mensagem_pc();

  } else if (bytesrec[6] == 7) { //guarda novos eventos
    n_b_EEPROM_new = n_b_EEPROM_old + bytesrec[1]-(num_bytes_cab_mens+1);
    if (n_b_EEPROM_old < n_b_EEPROM_new) {
      colocaEEprom(pos_inicial_eventos, n_b_EEPROM_new);
      for (j=n_b_EEPROM_old; j < n_b_EEPROM_new; j++) {
        colocaEEprom(pos_inicial_eventos+j+1,
        bytesrec[j - n_b_EEPROM_old + num_bytes_cab_mens]);
      }
      envia_acknowledge_pc();
    }
  }
}
```

Figura 3.20 - Função *actualiza eventos* da versão existente

```
void apaga_eventos(void){
    colocaEEprom(pos_inicial_eventos, 0);
    envia_acknowledge_pc();

    estado_td_programado=0;
}
```

```
void devolve_eventos(void){
    unsigned char j, numbytesx;
    numbytesx = 96;
    bytesenv[0]=bytesrec[0];
    bytesenv[1]=numbytesx;
    bytesenv[2]=bytesrec[4];
    bytesenv[3]=bytesrec[5];
    bytesenv[4]=bytesrec[2];
    bytesenv[5]=bytesrec[3];
    bytesenv[6]=bytesrec[6];
    for (j=0;j<(numbytesx-7);j++) {
      bytesenv[7+j] = ler_eeprom(j+pos_inicial_eventos);
    }
    bytesenv[numbytesx-1]=calcula_crc(numbytesx, 1);
    envia_mensagem_pc();
}
```

```
void actualiza_eventos(void){
    unsigned char j, n_b_EEPROM_old, n_b_EEPROM_new;
    n_b_EEPROM_old = ler_eeprom(pos_inicial_eventos);
    n_b_EEPROM_new = n_b_EEPROM_old + bytesrec[1]-(num_bytes_cab_mens+1);
    if (n_b_EEPROM_old < n_b_EEPROM_new) {
      colocaEEprom(pos_inicial_eventos, n_b_EEPROM_new);
      for (j=n_b_EEPROM_old; j < n_b_EEPROM_new; j++) {
        colocaEEprom(pos_inicial_eventos+j+1,
        bytesrec[j - n_b_EEPROM_old + num_bytes_cab_mens]);
      }
      envia_acknowledge_pc();
      estado_td_programado=1;
    }
}
```

Figura 3.21 – Funções *apaga_eventos*, *devolve_eventos* e *actualiza_eventos* adaptadas

Através do código gerado automaticamente e com a inserção das funções necessárias obtém-se o modelo pronto a ser compilado e executado no TD.

3.5. Comparação código Manual – Gerado automaticamente

Tal como já mencionado, um dos objectivos desta dissertação consiste em criar um TD com as mesmas características do TD existente, no entanto, recorrendo a ferramentas de geração automática de código. Como tal, considera-se de extrema importância/interesse, a comparação de desempenho e recursos utilizados pelas duas versões.

A capacidade de memória de programa e memória de dados do microprocessador utilizado (PIC18F4620 da Microchip) é de 32 *Kbytes* e cerca de 4 *Kbytes* (3968 *bytes*), respectivamente.

No que confere à memória de programa, verificou-se que a versão obtida com recurso a geração automática de código produziu um acréscimo de 47% em relação à versão existente. Este acréscimo deve-se essencialmente à inserção do código referente à gestão/evolução da RdP que acaba por superar a quantidade de código existente que é dispensada.

No que diz respeito à memória de dados, verificou-se um aumento de cerca de 34% em relação à versão existente. Tal, deve-se sobretudo ao aumento do número de variáveis utilizadas, neste caso, com a utilização de variáveis adicionais referentes aos sinais de entrada e saída da RdP assim como referente à própria gestão da RdP.

Apresenta-se nas figuras 3.22 e 3.23, a memória utilizada (marcada a verde) na versão existente e na nova versão, respectivamente. De salientar que embora tenha existido um relativo acréscimo de memória utilizada, esta representa apenas 23% no caso da memória de programa e 17% na memória de dados, relativamente à capacidade máxima do microprocessador. Desta forma, verifica-se que esta plataforma terá ainda capacidade para suportar, do ponto de vista de memória, um acréscimo de funcionalidades.

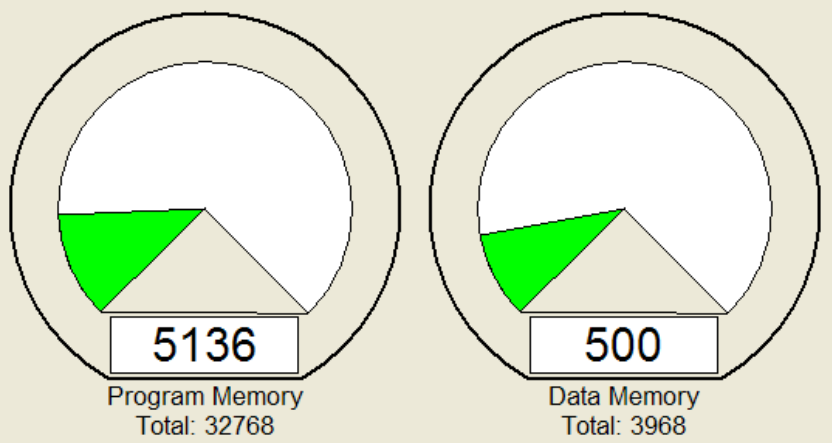


Figura 3.22 – Memória de programa e memória de dados utilizadas pela versão existente (código totalmente manual)

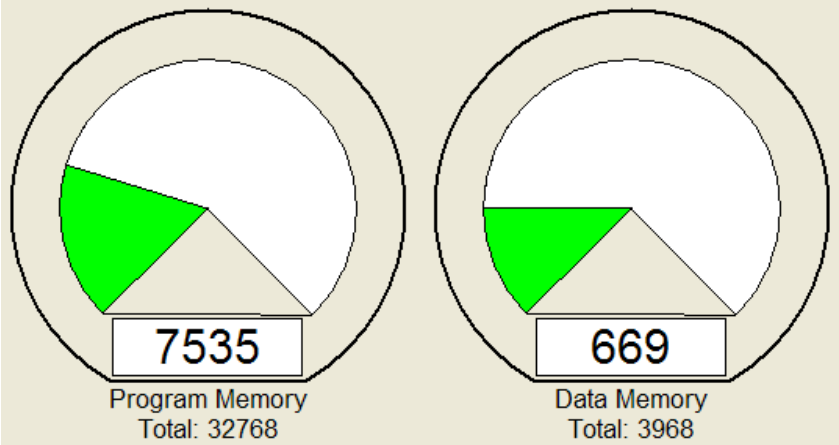


Figura 3.23 – Memória de programa e memória de dados utilizadas pela versão criada com recurso a geração automática de código.

Relativamente ao desempenho, foram analisados vários cenários. Para tal, contabilizou-se o tempo (utilizando a ferramenta de simulação do MPLAB) decorrente entre o momento em que o TD recebe a mensagem enviada pela aplicação ou por outro TD (dependendo do caso a ser testado) e o momento em que essa mensagem é completamente processada pelo TD. Dos vários cenários comparados (registo, programação de eventos, configuração, actualização da data e hora, activação de uma saída) concluiu-se que existe um acréscimo de tempo de resposta entre os 9% e os 12% na versão com recurso a geração automática de código.

No que diz respeito a simplicidade de compreensão, ainda que o código produzido manualmente esteja bem estruturado, torna-se mais simples perceber o funcionamento do sistema a partir do modelo gráfico.

Através do modelo gerado, simplifica-se também a alteração de funcionalidades assim como a adição de novas funcionalidades ao sistema.

4. Nova versão Sistema de Controladores Domóticos - Tiny Domots

4.1. Novas funcionalidades

Com certeza existe um sem número de possíveis novas funcionalidades a adicionar ao sistema.

Procurou-se implementar novas funcionalidades que permitissem aumentar o nível de controlo e liberdade por parte do instalador/utilizador assim como alargar o tipo de utilização.

A maioria das novas funcionalidades implementadas necessitou, não só da adição das mesmas ao modelo, como da respectiva adaptação da aplicação uma vez que em alguns casos, as funcionalidades do TD estão relacionadas com o tipo de mensagem que este recebe da aplicação. Como tal, apresenta-se no Subcapítulo 4.5 a adaptação da aplicação às novas funcionalidades.

Descrição das novas funcionalidades

1) **Testar comunicação com o TD seleccionado na aplicação.**

   Permite detectar a existência de problemas de comunicação com o TD, por exemplo, pode ter sido simplesmente desligado da rede de forma incorrecta.

2) **Testar a comunicação com todos os TD registados na aplicação.**

   Idêntica à anterior, no entanto efectua esses testes a todos os TD's registados na aplicação.

3) **Auxilio na identificação física do TD.**

   O TD pode receber uma mensagem da aplicação que desencadeia uma sequência de flashes dos LED's de informação do TD. Imaginando um cenário onde temos vários TD's instalados numa habitação, caso exista a necessidade de identificar um TD, esta será uma boa ferramenta visto que os TD's não possuem nenhuma identificação física.

4) Função “discovery”.

Permite verificar se existe algum TD ligado à rede que não esteja registado. Se existir, esse TD ao receber esse mesma mensagem irá enviar uma resposta à aplicação indicando que se encontra nessa condição. Todos os TD's que estão registados vão ignorar essa mensagem.

5) Visualizar estado de todas as entradas e saídas.

O TD pode fornecer à aplicação o estado de todas as suas entradas e saídas. Na aplicação, é possível atribuir designações a cada uma das entradas e saídas, tornando a sua identificação mais intuitiva

6) Monitorizar estado de todas as entradas e saídas.

O TD pode fornecer repetidamente à aplicação, em intervalos de tempo definidos, o estado das suas entradas e saídas. Como trabalho futuro, seria interessante a possibilidade da aplicação construir um gráfico, ao longo do tempo, com esta informação

7) Actuar saídas manualmente.

O instalador tem a possibilidade de forçar o estado das saídas do TD.

8) Função E/S analógica

O instalador do sistema pode, aquando do registo do TD, escolher entre 2 tipos, sendo que cada um deles terá uma configuração de E/S diferente.

Tipo 1 - TD terá:

- 12 entradas digitais
- 11 saídas digitais

Tipo 2 : TD terá:

- 1 entrada analógica
- 11 entradas digitais
- 1 saída analógica(a 8bits)
- 3 saídas digitais

9) Contador

Permite contar o número de vezes que a Entrada digital 1 é activada e guarda a data e hora em que esta foi activada pela última vez. Para desencadear esta acção não é necessária intervenção directa da aplicação, basta que a entrada digital 12 do TD esteja activa, ou seja, esta entrada funciona como que uma permissão de contagem. Na aplicação, é possível a qualquer momento pedir ao TD o valor desta contagem assim como a informação da última ocorrência.

4.1.1. Diagrama de Casos de Uso

Na Figura 4.1 apresenta-se o diagrama de casos de uso das novas funcionalidades do TD, sendo que no próximo subcapítulo é feita a descrição dos vários casos de uso.

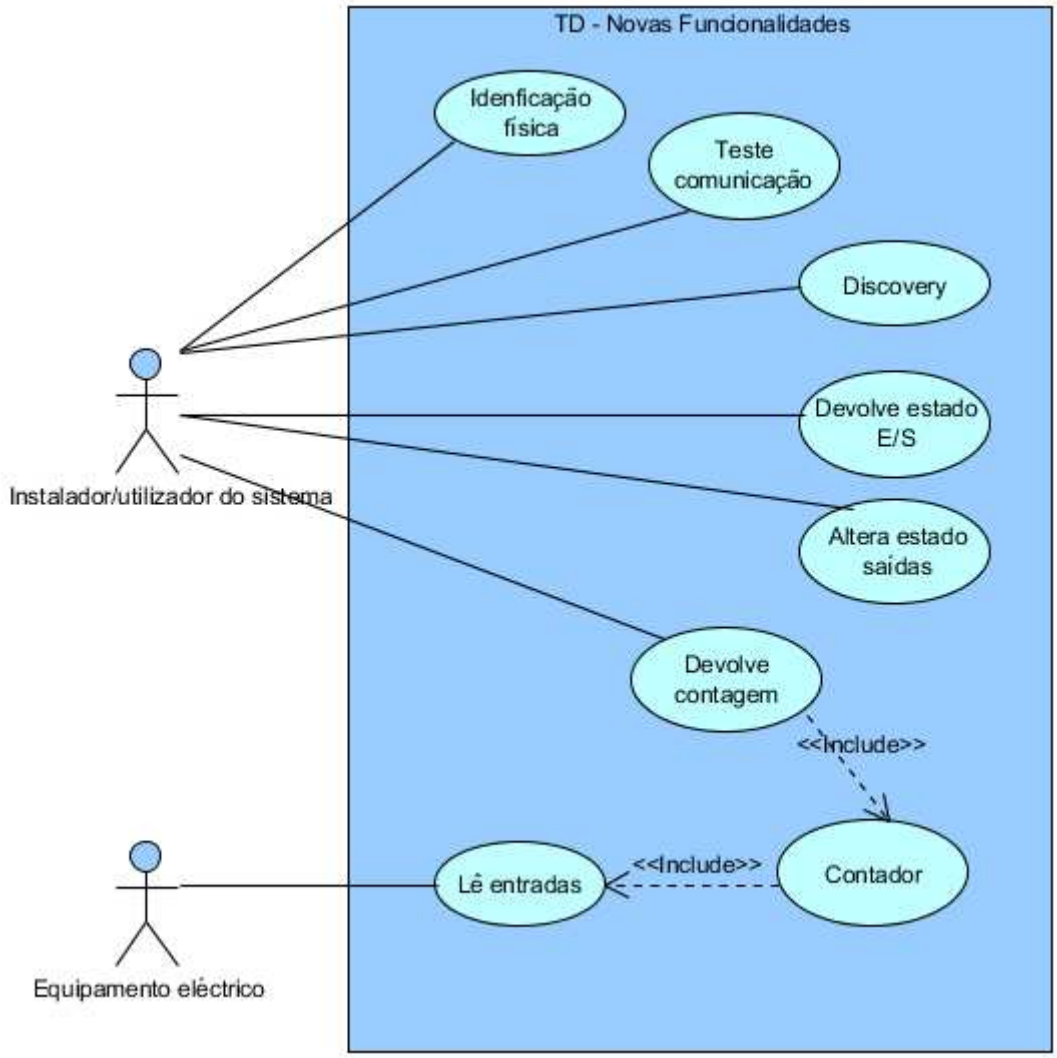


Figura 4.1 – Diagrama de Casos de Uso das novas funcionalidades do TD

4.1.2. Descrição dos Casos de Uso

Teste de Comunicação	
Pré-condição	O TD tem de estar registado
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de teste de comunicação vinda da aplicação **2.** Envia uma mensagem de resposta à aplicação.
Pós-condição	

Quadro 4.1 – Descrição do caso de uso *Teste de Comunicação*

Identificação física	
Pré-condição	O TD tem de estar registado
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de identificação vinda da aplicação **2.** O TD executa a sequência de intermitência dos seus LED's de informação **3.** Envia uma mensagem de resposta à aplicação.
Pós-condição	

Quadro 4.2 – Descrição do caso de uso *Identificação física*

Discovery	
Pré-condição	O TD tem de estar ligado à rede
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de "discoberta" vinda da aplicação **2.** Se o TD não estiver registado, envia mensagem de resposta à aplicação **3.** Se o TD estiver registado, ignora a mensagem
Pós-condição	

Quadro 4.3 - Descrição do caso de uso *Discovery*

Devolve estado das entradas e saídas	
Pré-condição	O TD tem de estar registado
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de pedido de estado das E/S vinda da aplicação **2.** Envia uma mensagem com o estado das E/S à aplicação
Pós-condição	

Quadro 4.4 – Descrição do caso de uso *Devolve estado E/S*

Altera estado da saídas	
Pré-condição	O TD tem de estar registado
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de alteração do estado das saídas vinda da aplicação **2.** Altera o estado das saídas **3.** Envia uma mensagem de resposta à aplicação
Pós-condição	

Quadro 4.5 – Descrição do caso de uso *Altera estado das saídas*

Contador	
Pré-condição	O TD tem de estar registado
Descrição:	**1.** O caso de uso começa quando a entrada 12 do TD é activada **2.** Sempre que a entrada 1 passa para o estado activado o contador incrementa e é guardada a hora desse evento **3.** Envia uma mensagem de resposta à aplicação
Pós-condição	

Quadro 4.6 – Descrição do caso de uso *Contador*

Devolve contagem	
Pré-condição	O TD tem de estar registado
Descrição:	**1.** O caso de uso começa quando o TD recebe uma mensagem de solicitação do valor de contagem **2.** Envia uma mensagem com a informação solicitada pela aplicação **3.** Reinicia o contador
Pós-condição	

Quadro 4.7 – Descrição do caso de uso Devolve contagem

4.2. Modelos comportamentais

Na Figura 4.2 é apresentado o modelo *Trata Mensagens* com a adição da funcionalidade 1) apresentada no subcapítulo anterior.

Comparativamente ao modelo anterior (Figura 3.8), é apenas adicionado um bloco, destacado na figura, referente ao tratamento deste novo tipo de mensagem enviada pela aplicação. O sinal de saída *RESPONDE_MSG_TESTE* irá desencadear uma função simples que trata de enviar uma mensagem de resposta para a aplicação.

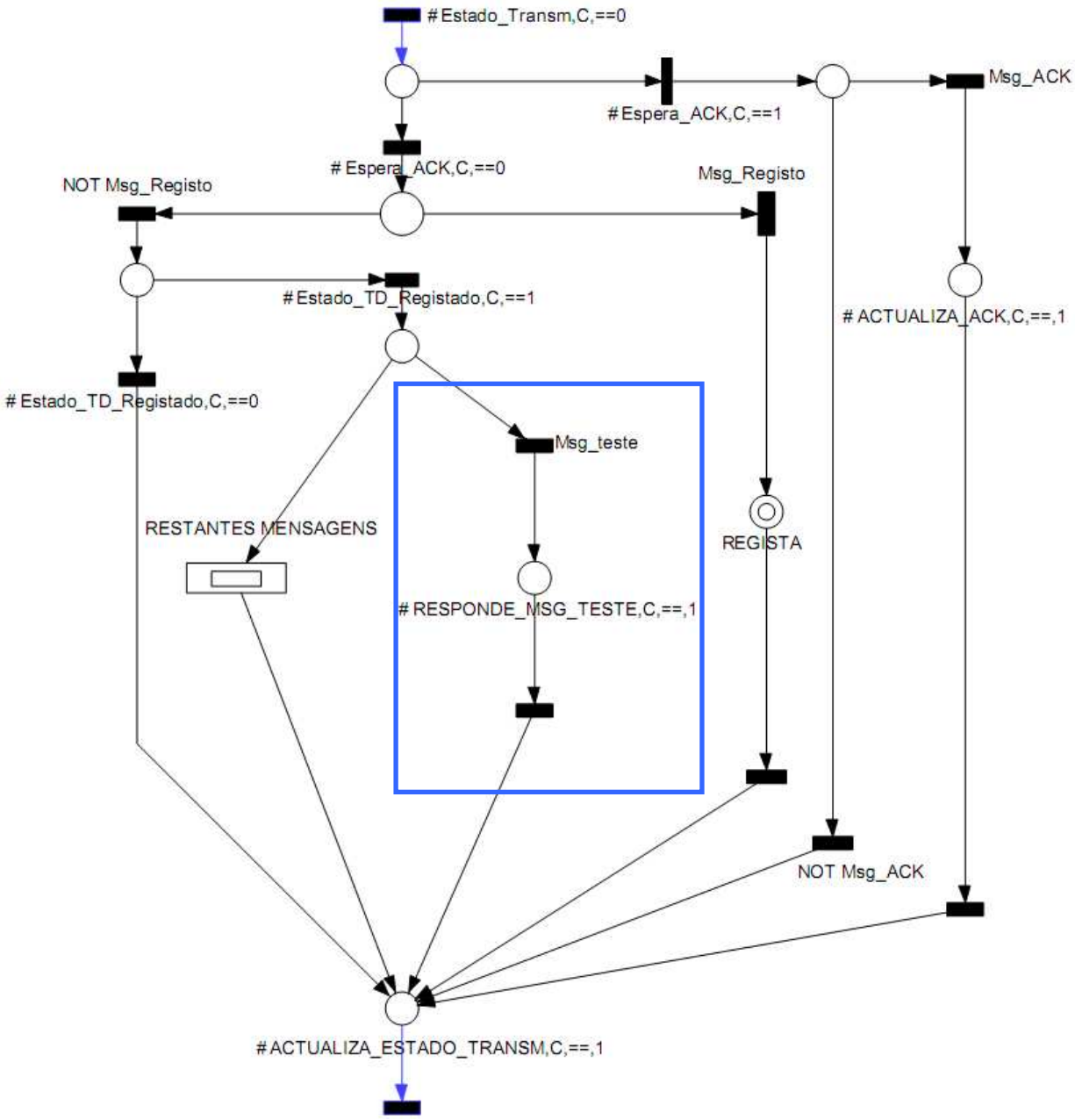


Figura 4.2 – RdP IOPT *Trata Mensagens* com adição da funcionalidade 1)

De forma a simplificar a visualização, os restantes processos referentes aos tipos de mensagens já utilizadas nos modelos anteriores, foram inseridos numa macro-transição (esta informação aplica-se às Figuras 4.2, 4.3 e 4.5).

Na Figura 4.3 é apresentado o modelo *Trata Mensagens* com a adição da funcionalidade 3) apresentada no subcapítulo anterior. O processo é idêntico ao modelo da Figura 4.2, no entanto, referente a outro tipo de mensagem.

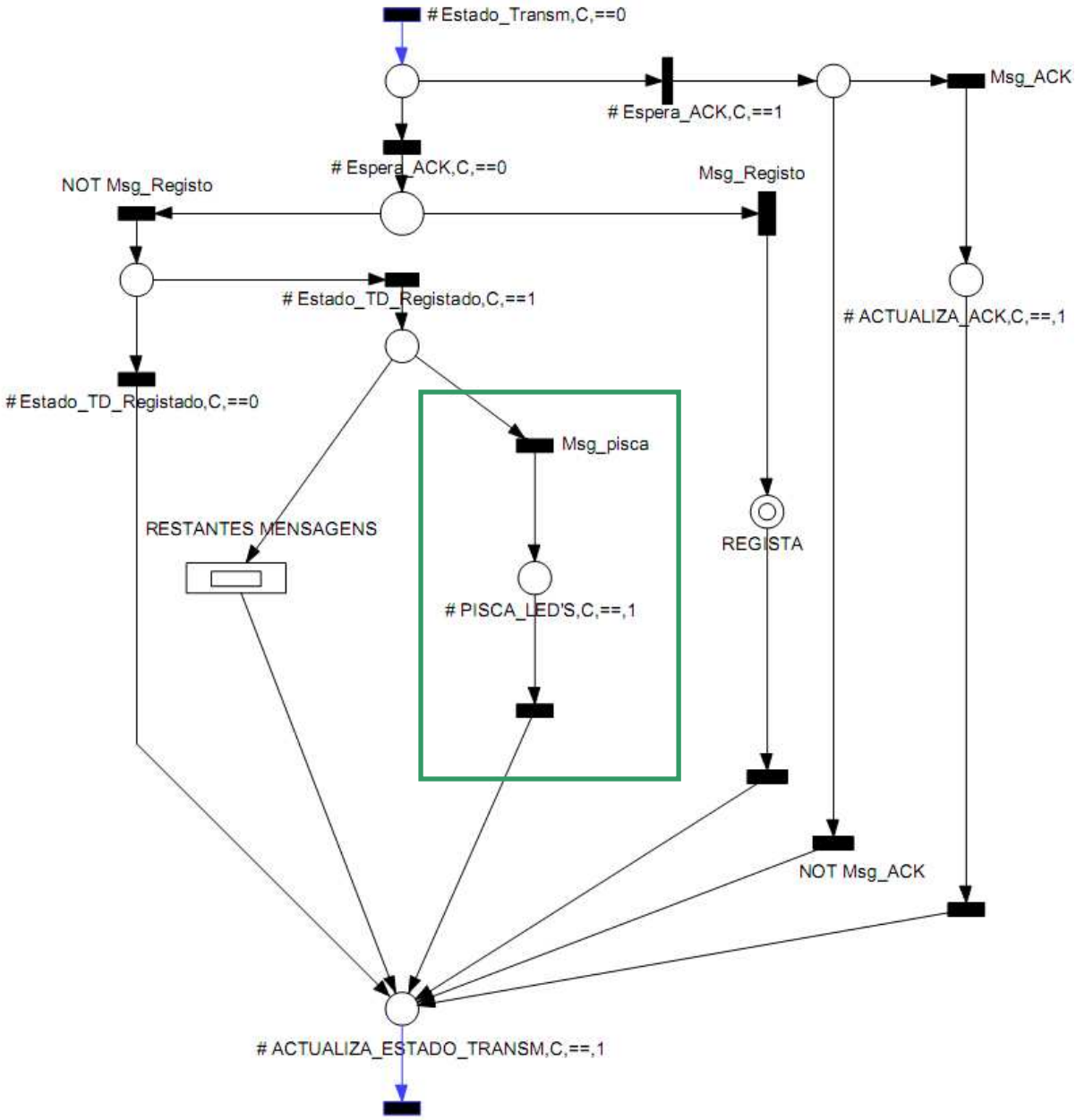


Figura 4.3 – RdP IOPT *Trata Mensagens* com adição de funcionalidade 3)

Na Figura 4.4 mostra-se o sub-modelo *Regista* , com a adição da funcionalidade 4), sendo que a parte do modelo referente a esta funcionalidade se encontra destacada. Neste caso, comparativamente ao modelo da Figura 3.10, o novo bloco do modelo veio substituir um arco.

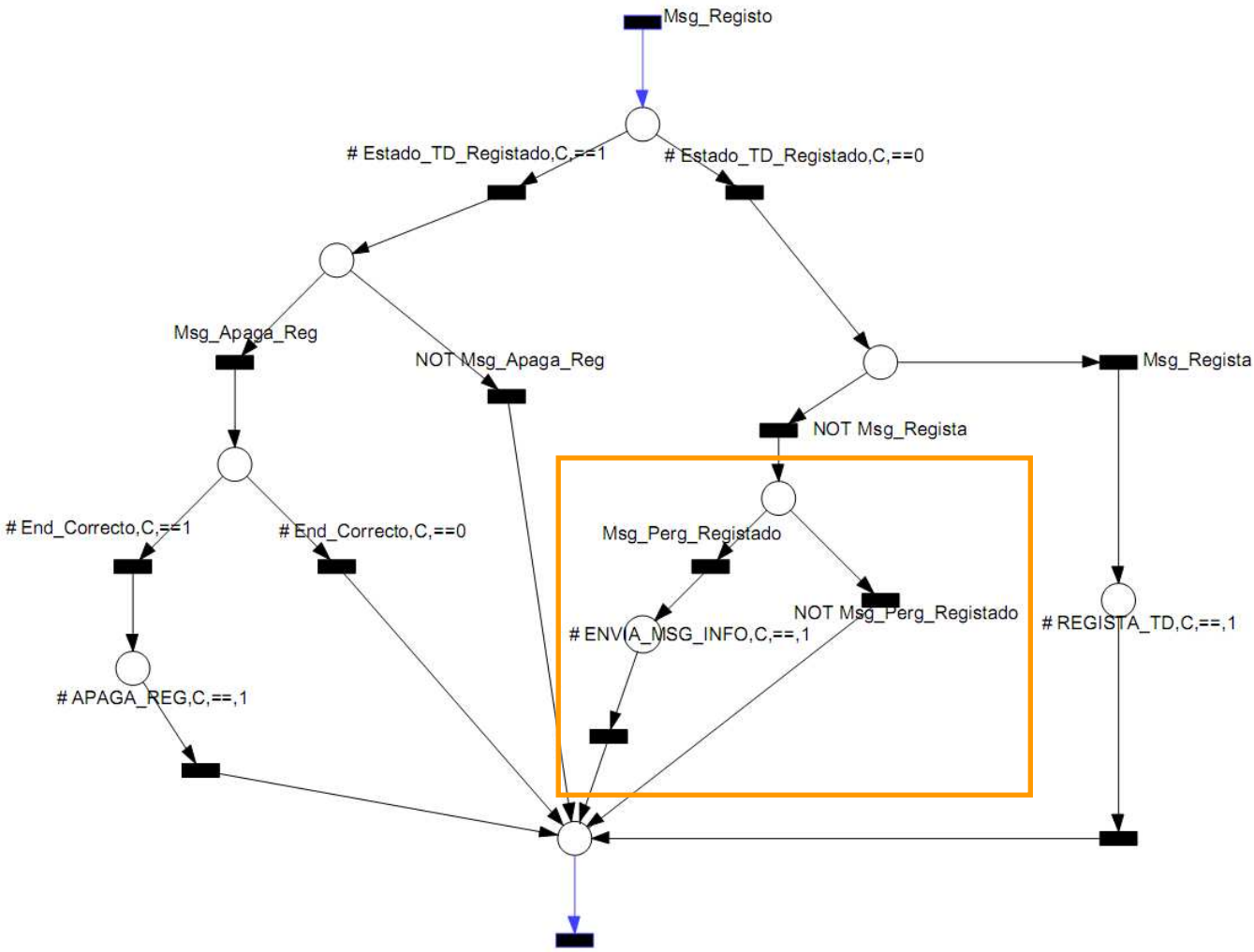


Figura 4.4 – RdP IOPT *Regista* com adição da funcionalidade 4)

Na Figura 4.5 é apresentado o modelo *Trata Mensagens* com a inclusão das funcionalidades 5) e 7). São ambas apresentadas na mesma figura pois estão directamente ligadas. Comparativamente à Figura 3.9 podemos ver que surgiu mais um troço referente as estas duas funcionalidades.

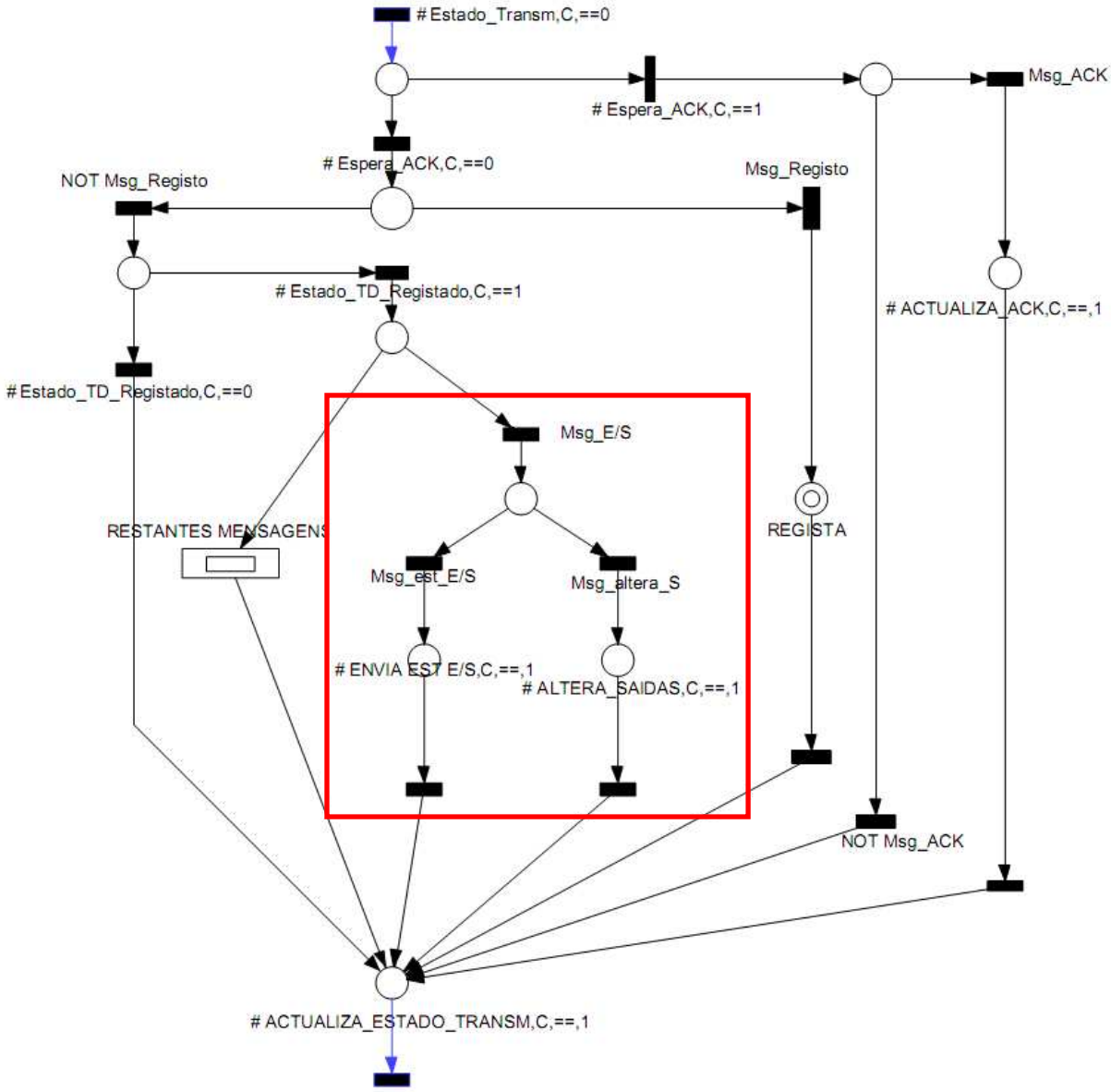


Figura 4.5 - RdP IOPT *Trata Mensagens* com adição das funcionalidades 5) e 7)

Relativamente à funcionalidade 8), mostra-se na Figura 4.6 a forma como esta foi inserida no modelo original da Figura 3.10. Esta aparece no lugar de o arco, no troço referente ao registo, uma vez que está relacionada com as características de registo do TD.

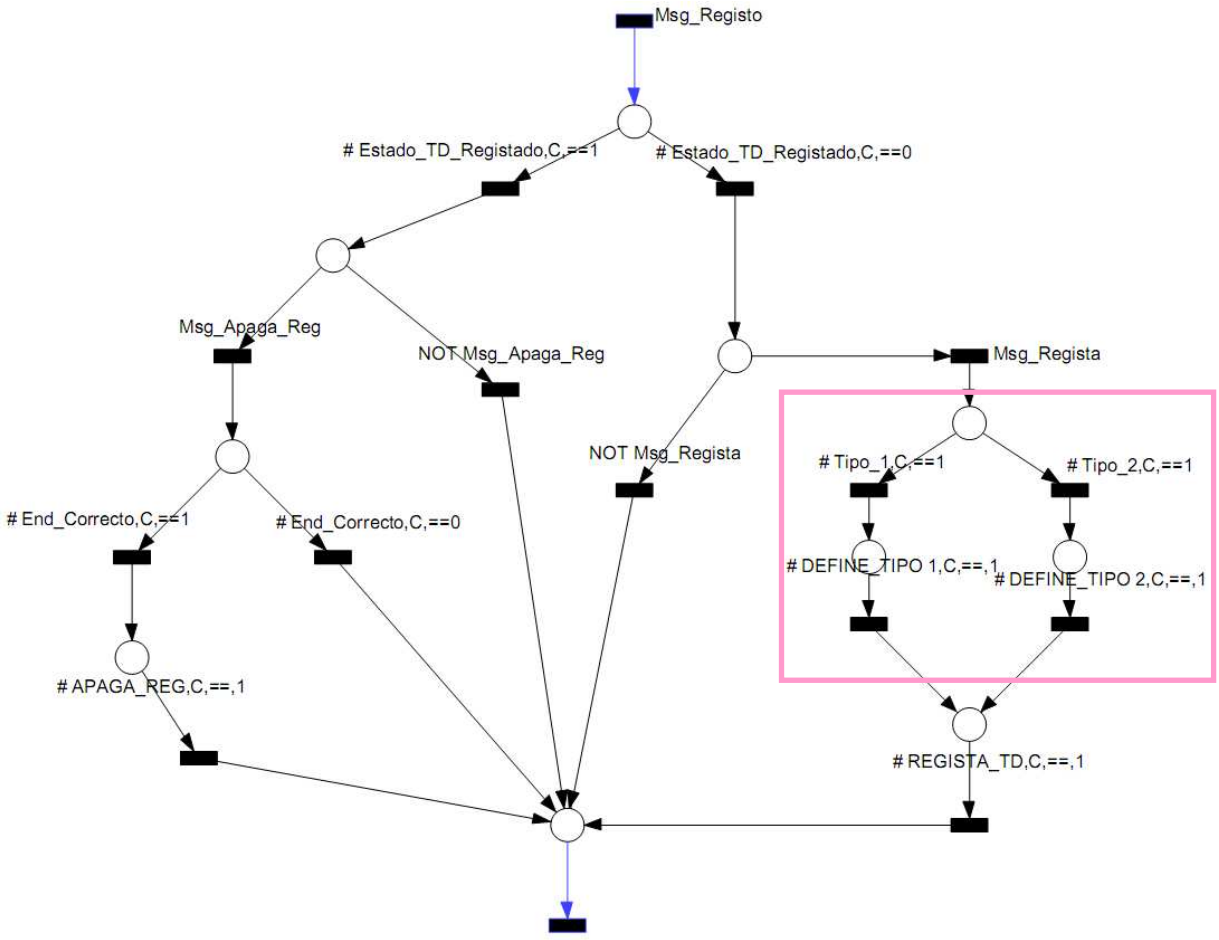


Figura 4.6 – RdP IOPT *Trata Mensagens* com adição da funcionalidade 8)

Por último, temos o modelo que diz respeito à funcionalidade 9) destacado a vermelho na Figura 4.7. Neste caso não existe qualquer ligação através de arcos ao modelo principal.

O lugar assinalado com a letra A, é o lugar onde serão adicionadas marcações ao longo da contagem. O número de marcações corresponderá ao número de vezes que o *sinal de entrada 1* foi activado (representado pelo evento de entrada *incrementa).*

Neste modelo, temos ainda o evento de saída *grava_hora* responsável por desencadear a função que memoriza a data e hora do momento em que o *sinal de entrada 1* foi activado pela última vez.

À parte deste modelo, existiu ainda a necessidade de inserir no modelo *Trata Mensagens* um módulo responsável por enviar estes dados para a aplicação sempre que solicitado. Para tal, foi implementado um modelo idêntico ao das Figuras 4.2 e 4.3 no entanto, referente a este tipo de mensagem.

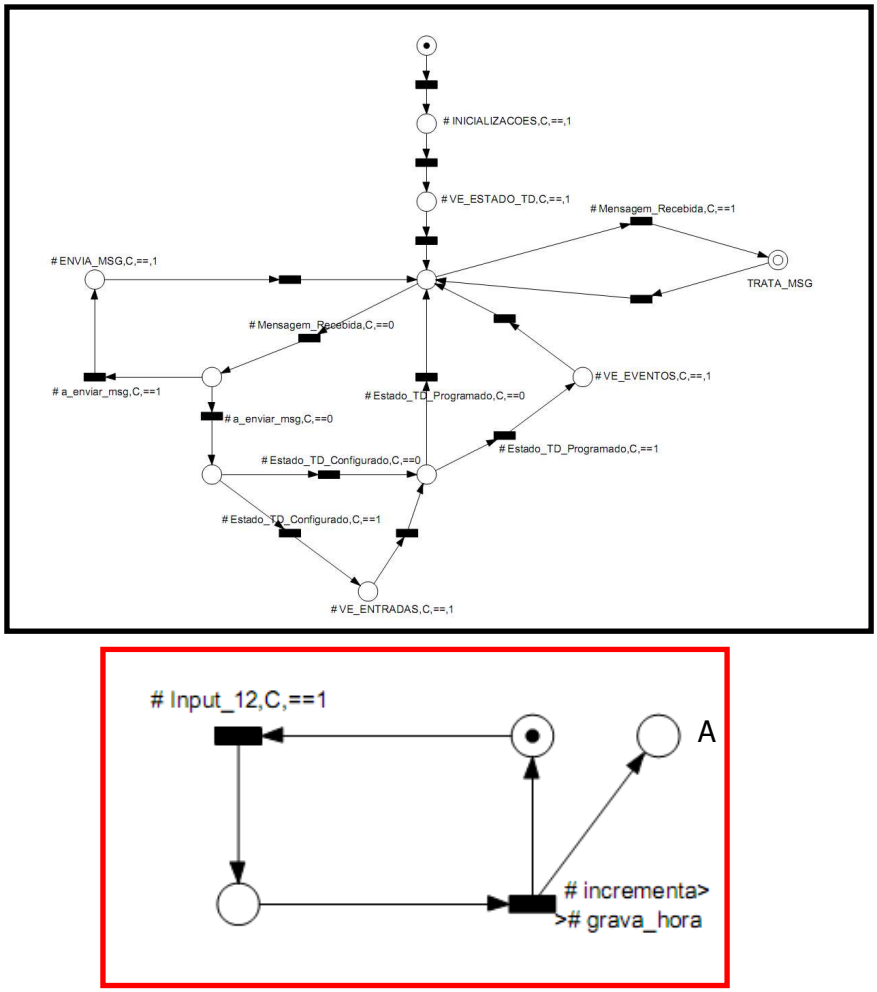


Figura 4.7 – RdP IOPT *Principal* com adição da funcionalidade 9)

4.3. Código gerado automaticamente

À imagem do apresentado no capítulo 3, aplicou-se o resultado do editor de Redes de Petri(Snoopy IOPT) ao gerador de código C (PNML2C), gerando este, o mesmo tipo de ficheiros em linguagem C: *main.c, functionsc.c, netc.c, functionsc.h e netc.h.*

Relativamente aos ficheiros existentes da versão base, os novos ficheiros criados contemplam, para além do código já conhecido, o código referente à adição gráfica de novos nós, novos sinais e eventos assim como da própria evolução da rede responsáveis pelas novas funcionalidades, no entanto os ficheiros mantêm a mesma estrutura.

Para a criação do modelo operacional adicionaram-se aos ficheiros gerados de forma automática, as funções base existentes da versão anterior assim como as novos funções criadas de forma manual referentes à novas funcionalidades.

Apresenta-se na Figura 4.8 o código manual inserido na função *OutputData* relativa às novas funcionalidades, ou seja, adicionado ao código apresentado na Figura 3.19. Desta forma, sempre que um sinal de saída é activado é desencadeada a respectiva função.

```
//####NOVAS FUNCIONALIDADES##############

//-----SINAIS--------
if (RESPONDE_MSG_TESTE == 1)
    envia_acknowledge_pc();

if (PISCA_LEDS == 1)
    pisca_leds();

if (ENVIA_MSG_INFO == 1)
    envia_msg_info();

if (ENVIA_ESTADO_E_S ==1)
     devolve_IO();

if (ALTERA_SAIDAS == 1)
    activa_saidas();

if (DEFINE_TIPO_1 == 1)
    define_tipo_1();

if (DEFINE_TIPO_2 == 1)
    define_tipo_2();

if (ENVIA_DADOS_CONTADOR == 1)
    envia_dados_contador();

//-----EVENTOS--------
if (grava_hora == 1)
    grava_hora_();
```

Figura 4.8 – Excerto da nova função *OutputData* referente à adição de novas funcionalidades

4.4. Análise

Para analisar as características do código gerado, interessa fazer-se a comparação entre os vários códigos gerados com auxílio de ferramentas de geração automática.

Relativamente à nova versão procurou-se também perceber qual a influência que o número de nós (lugares e transições), sinais e número de funcionalidades adicionais têm nos recursos utilizados.

Apresenta-se no Quadro 4.8 o resumo comparativo das características das várias versões, sendo (A) referente à versão base e as restantes linhas (B, C, D, E e F) referentes à versão criada que possui funcionalidades adicionais, sendo que foram registados valores à medida que se adicionou progressivamente novas funcionalidades. Neste quadro consideram-se o número de nós e sinais adicionais associados às novas funcionalidades, o número total de nós e sinais assim como recursos de memória utilizados em valores absolutos e em percentagem em relação à capacidade total (32 *Kbytes* de memória de programa e cerca de 4 *Kbytes* de memoria de dados). Apresenta-se ainda nas Figura 4.9 e 4.10, os gráficos comparativos da relação entre o número de nós e os recurso de memória em valores absolutos e em percentagem respectivamente.

	Funcion. adicionais	Nº nós adicionais	Nº sinais adicionais	Nº total nós	Nº total sinais	Mem. Programa		Mem. Dados	
						Valor (KB)	% capacid.	Valor (KB)	% capacid.
Versão Base (A)	0	0	0	78	37	7,54	23,0	0,669	16,9
Nova versão (B)	2	8	2	86	39	7,87	24,0	0,688	17,3
Nova versão (c)	3	11	3	89	40	7,96	24,3	0,693	17,5
Nova versão (D)	4	14	4	92	41	8,23	25,1	0,698	17,6
Nova versão (E)	5	24	10	102	47	8,59	26,2	0,720	18,1
Nova versão (F)	7	36	14	114	51	9,02	27,5	0,750	18,9

Quadro 4.8 – Características do código das várias versões

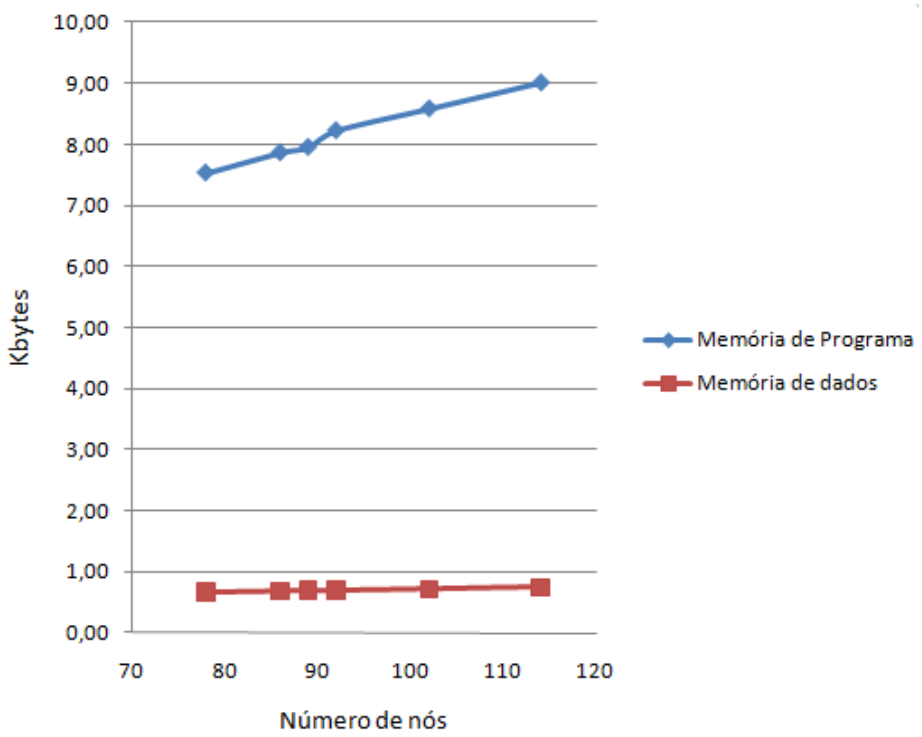


Figura 4.9 - Gráfico com relação entre o número de nós e memória utilizada em valores absolutos

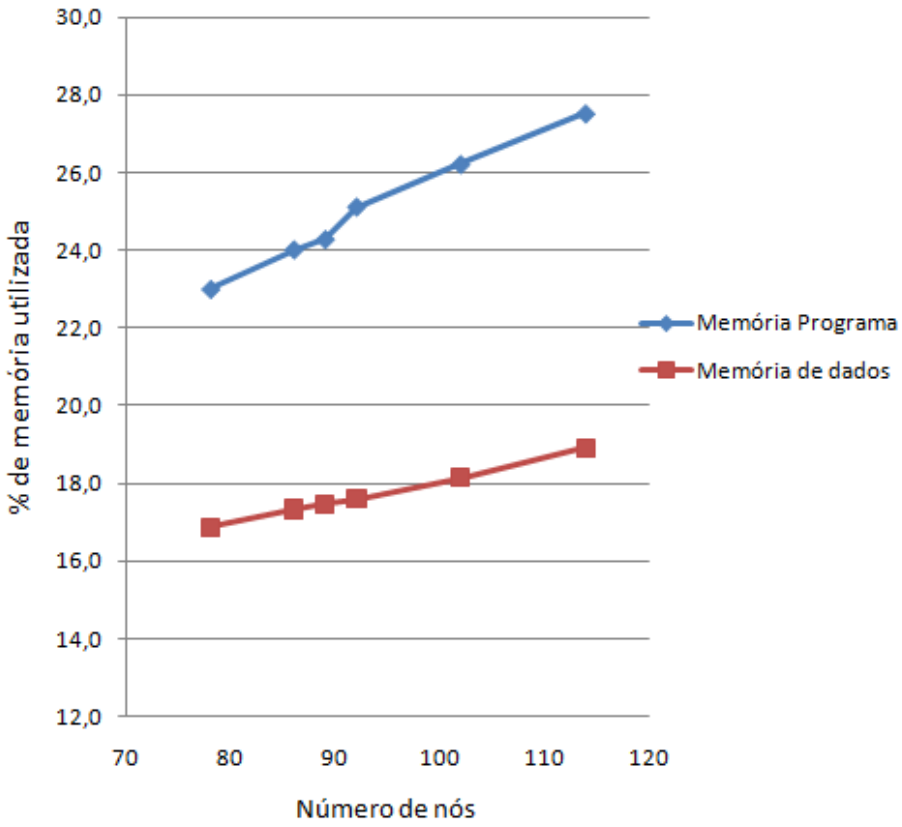


Figura 4.10 - Gráfico com relação entre o número de nós e memória utilizada em percentagem

Analisando o Quadro 4.8 e as Figuras 4.9 e 4.10 vemos que, tal como seria de esperar, à medida que aumentamos o número de nós, este reflecte-se no aumento de recurso utilizados.

No entanto, um dado interessante é o facto de ao comparar-se, por exemplo, a versão base com a versão com o maior número de novas funcionalidades, vemos um aumentado de 46% de número de nós (78 passámos a ter 114) mas apenas um aumento de 20% da memória de programa e 12% na memória de dados), desta forma vemos que a progressão da utilização de memória, tanto de dados como de programa, é relativamente inferior à progressão do número de nós e consequentemente de funcionalidades.

4.5. Adaptação da aplicação

A modificação da aplicação não está contemplada nos objectivos desta dissertação, no entanto, verificou-se a necessidade de esta se adaptar às novas funcionalidades criadas.

A maioria das novas funcionalidades implementadas necessitou, não só da adição das mesmas ao modelo, como da respectiva adaptação da aplicação uma vez que em alguns casos, as

funcionalidades do TD estão relacionadas com o tipo de mensagem que este recebe da aplicação e são estas que vão desencadear determinado funcionamento por parte do TD.

Para a adaptação da aplicação foi utilizada a ferramenta *Borland C++ Builder.*

Na Figura 4.11 mostra-se a forma como a aplicação foi adaptada de forma permitir a visualização a qualquer instante do estado de todas as entradas e saídas do TD seleccionado, permite alterar manualmente o estado das saídas, monitorizar as entradas e saídas em intervalos de tempo seleccionados e atribuir designação a cada uma das entradas e saídas.

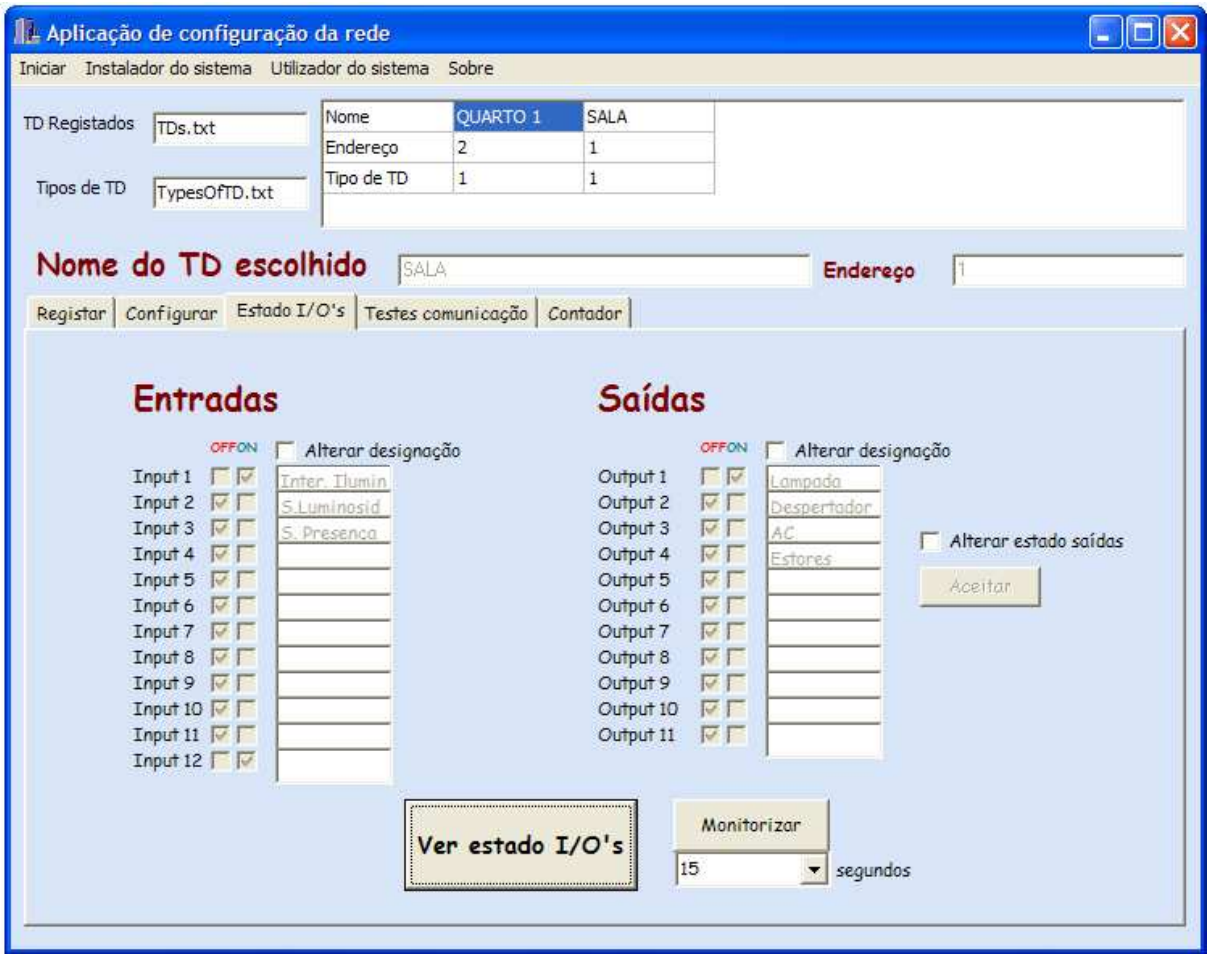


Figura 4.11 – Aplicação adaptada às novas funcionalidades relacionadas com as entradas e saídas

Para visualizar o estado das E/S basta carregar no botão “Ver estado I/O’s” que faz com que a aplicação envie uma mensagem de pedido ao TD e este responda com a informação solicitada. As “checkboxs” de cada E/S são actualizadas de acordo com a mensagem recebida.

Para alterar as saídas basta seleccionar a opção “Alterar estado saídas” que faz com que as “checkboxs” das saídas fiquem editáveis, escolher qual o estado que se pretende e finalmente carregar no botão “Aceitar” que entretanto também foi activado.

A aplicação foi também adaptada para possibilitar efectuar alguns testes de comunicação aos TD's. Na Figura 4.12 apresenta-se a forma como estes estão implementados.

Tal como indicado no Subcapítulo 4.1, é possível testar a comunicação com o TD seleccionado assim como com todos os TD registados na aplicação. Permite ainda fazer com que os led's de informação do TD (o TD possuí 4 Leds de informação) pisquem de forma sucessiva para auxiliar na identificação física do TD. Neste campo foi ainda implementada a funcionalidade "Discovery" que permite verificar se existe algum TD ligado à rede mas que no entanto no se encontra registado.

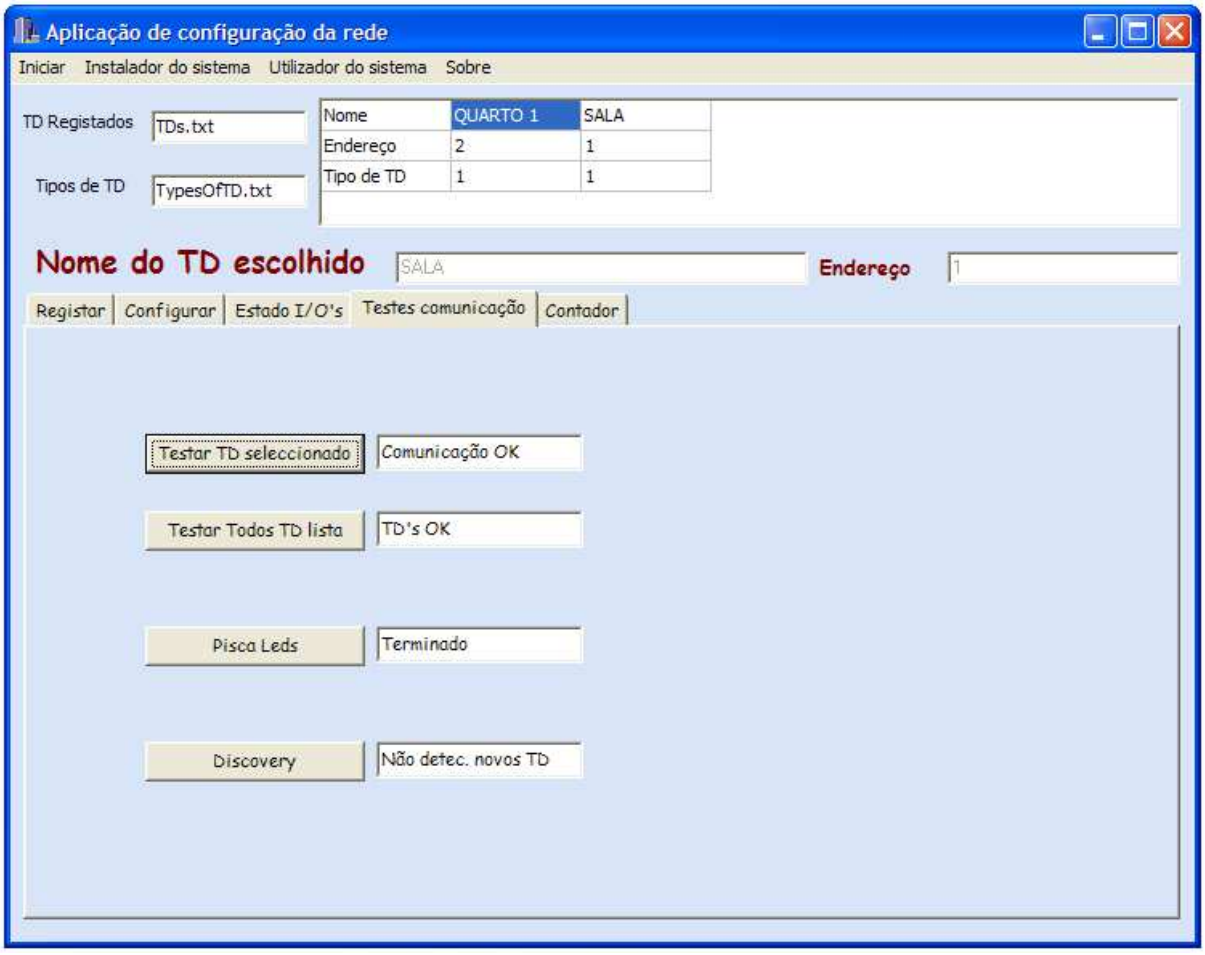


Figura 4.12 – Aplicação adaptada às novas funcionalidades relacionadas com testes de comunicação

Relativamente à funcionalidade que permite atribuir um tipo (tipo 1 ou 2) a cada TD, correspondendo cada um deles a uma configuração diferente do número e tipo de entradas e saídas, mostra-se na Figura 4.13 a forma como esta é conseguida. Aquando do registo do TD, existe mais este parâmetro que tem de ser escolhido, desta forma, a mensagem de registo enviada ao TD possui esta informação que fará com que o TD assuma o seu tipo e efectue as configurações necessárias ao funcionamento correspondente.

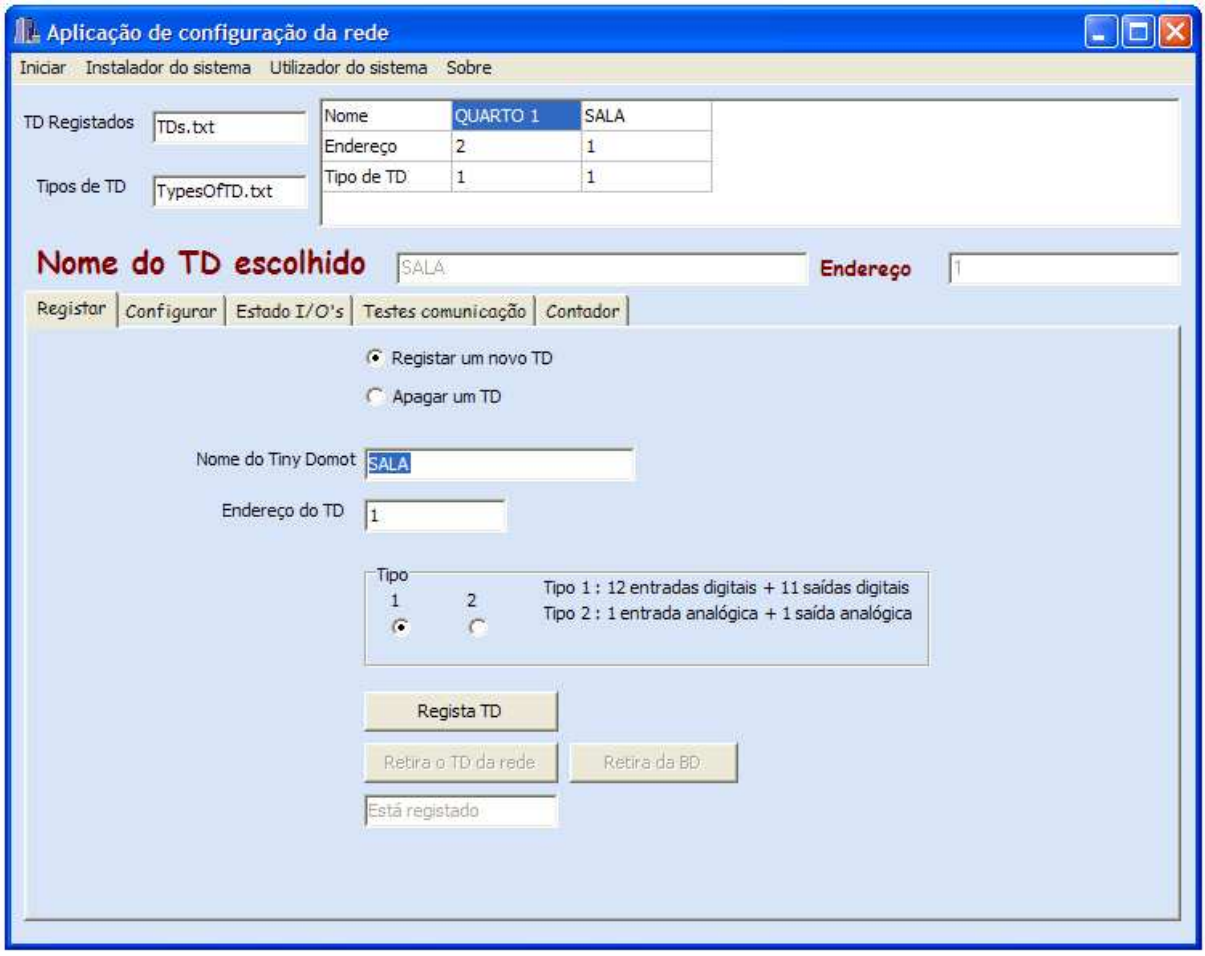


Figura 4.13 – Aplicação adaptada às novas funcionalidades relacionadas com o tipo de TD

Por último mostra-se na Figura 4.14, a funcionalidade que permite solicitar ao TD a contagem do número de vezes que a *entrada 1* foi activada, assim com a data e hora deste último evento.

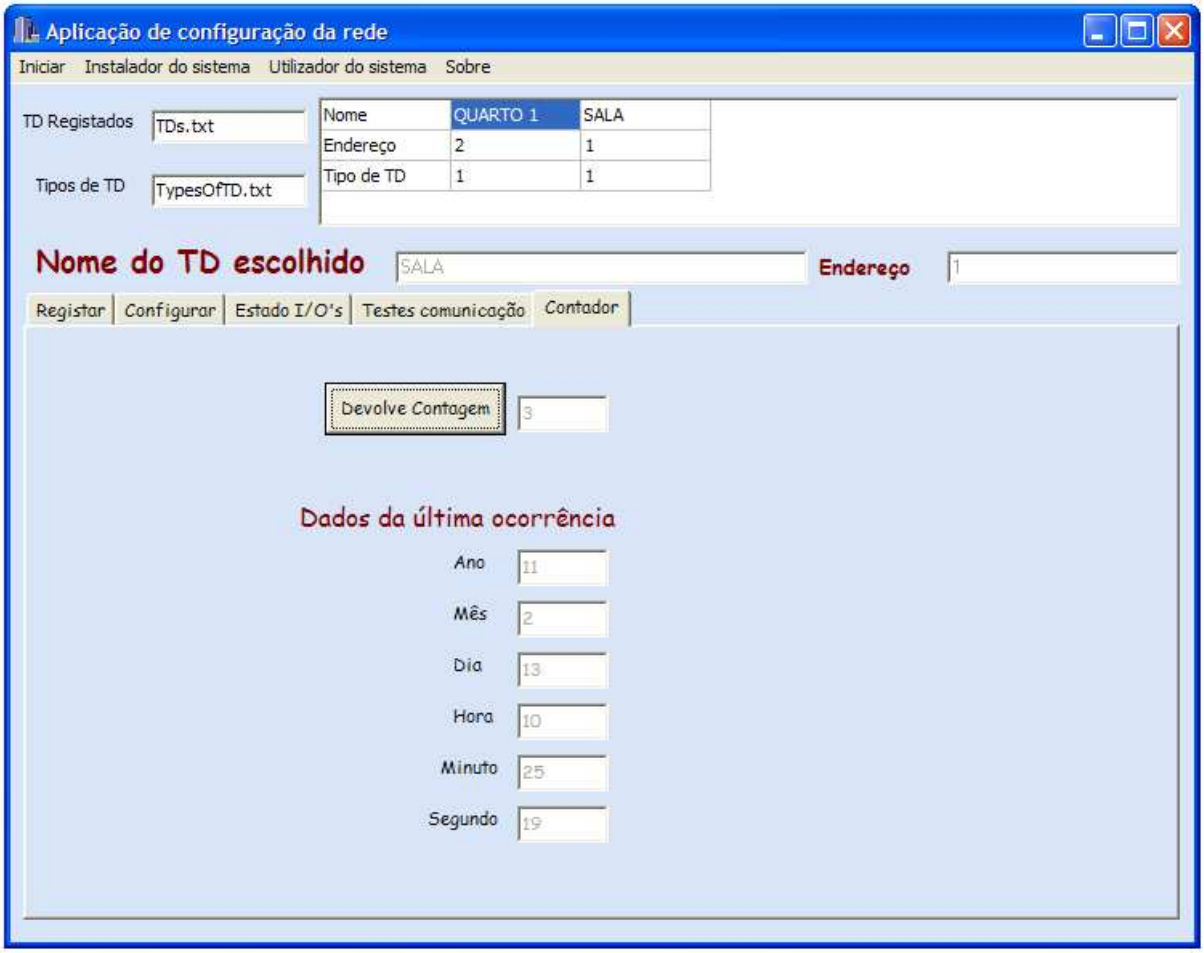


Figura 4.14 - Aplicação adaptada às novas funcionalidades relacionadas contador de entrada

5. Conclusões

Para além da criação de uma nova versão de controladores Domóticos Tiny Domots, esta dissertação possibilitou encontrar e validar regras de tradução manual entre os Diagramas de Estados e Redes de Petri (RdP), de forma a permitir tirar partido de ferramentas de edição de RdP e geração automática de código C criadas no âmbito do projecto Fordesign [Fordesign 07]. O uso dessas ferramentas (Editor de RdP e gerador de código C automático) nesta aplicação serviu também para validar a sua funcionalidade, verificando-se que são ferramentas com elevadas potencialidades, no entanto, ainda com a possibilidade de evolução futura.

O código final pronto a ser executado na plataforma, foi produzido maioritariamente com recurso a ferramentas de geração automáticas de código. No entanto, existiu também a necessidade de integrar funções específicas já existentes (algumas delas após ligeiras adaptações) uma vez que estas ferramentas não nos permitem, em alguns casos, atingir o mesmo nível de abstracção que o código manual.

Outro ponto desta dissertação consistiu em comparar os dois tipos de código (manual existente – gerado automaticamente) com as mesmas funcionalidades. Quanto ao tempo de resposta, verifica-se que o código gerado a partir do modelo, revela-se ligeiramente superior ao criado manualmente, no entanto, neste caso esse incremento não é significativo, não comprometendo o correcto funcionamento do sistema.

Verificou-se ainda que o código produzido a partir do modelo necessita de mais recursos ao nível de memória de programa e de dados. Este acréscimo deve-se essencialmente à inserção do código referente à gestão/evolução da RdP que acaba por superar a quantidade de código existente que é dispensada. Para além disso existe ainda um aumento do número de variáveis, referentes aos sinais de entrada e saída da RdP assim como referente à própria gestão da RdP que acabam por aumentar a necessidade de memória de dados.

Tal como demonstrado, o modelo encontrado, permite a fácil adição de novas funcionalidades ao sistema actual. No entanto, essa adição, pode implicar a necessidade de adaptação da aplicação, de forma a tornar possível a utilização dessas novas funcionalidades.

A nova versão (com novas funcionalidades) criada permite alargar o tipo de utilização assim como permite um maior grau de liberdade por parte do instalador e/ou utilizador. Através do resultado desta nova versão analisou-se ainda a relação entre o aumento do número de nós e a memória necessária de forma a encontra-se um padrão aproximado entre inclusão de novas funcionalidades e a necessidade de recursos. Através desta análise verificou-se que o aumento do número de nós não é proporcional ao aumento de necessidade de recursos, ou seja, a progressão da necessidade de recursos é menos acentuada que a progressão do número de nós.

Desta forma, verifica-se que o modelo encontrado, assim como nova versão de Tiny Domots criada, poderão servir de base a novas gerações de controladores domóticos Tiny Domots através da adição de novas funcionalidades e/ou alteração de funcionalidades existentes suportando a sua constante adaptação às necessidades do momento.

Um aspecto não menos importante diz respeito à possibilidade de portabilidade do sistema para outras plataformas, por exemplo, se tivermos uma plataforma diferente da utilizada mas que possibilite igualmente a utilização de código C, basta garantir que esta possui os recursos necessários e que seja alterada a função referente ao código dependente da plataforma. Para além disto, através do modelo criado, pode ser aplicada outra ferramenta de geração automática de outro tipo de código, por exemplo, código VHDL (também desenvolvida no âmbito do projecto FORDESIGN) permitindo alargar a aplicação do modelo a um vasto número de plataformas

Por último, verifica-se que a metodologia encontrada pode ser extensível a outras áreas que recorram a sistemas embutidos, não se limitando apenas à área da Domótica.

Referências

Referências Bibliográficas

[Barros, 96] Barros, João Paulo M. P. Ramos, "CpPNeTS: Uma Classe de Redes de Petri de Alto-nível - Implementação de um sistema de suporte à sua aplicação e análise", Dissertação de Mestrado, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 1996.

[Barros & Gomes, 04] Barros, João Paulo e Gomes, Luís, "Operational PNML: Towards a PNML Support for Model Construction and Modification", in Workshop on the Definition, Implementation and Application of Standard Interchange Format for Petri Nets 2004, Junho 2004.

[Chekofsky & Cross II, 90] Chikofsky, E. J. e Cross II, J. H. "Reverse Engineering and Design Recovery: A Taxonomy." IEEE Software, v. 7, n. 1, 1990.

[David, Alla, 92] David, R. and Alla, H., "Petri Nets and Grafcet Tools for modelling discrete event systems, Prentice Hall, 1992.

[Douglass, 98] Douglass, Bruce Powel, "Real-Time UML: Developing Efficient Objects For Embedded Systems", Addison Wesley Longman Inc., 1998.

[Douglass, 99] Douglass, Bruce Powel, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns", Addison-Wesley, 1999.

[Gomes, 97] Gomes, Luís Filipe dos Santos, "Redes de Petri Reactivas e Hierárquicas – integração de formalismos no projecto de sistemas reactivos de tempo-real", Tese de Doutoramento, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 1997.

[Gomes et al., 2007] Gomes, Luís; Barros, João Paulo; Costa, Anikó e Nunes, Ricardo, "The Input-Output Place-Transition Petri Net Class and Associated Tools", 5th IEEE International Conference on Industrial Informatics (INDIN 2007), Viena, Austria, 2007

[Gomes, Barros, Costa, 07] Gomes, Luís; Barros, João Paulo; Costa, Anikó, "Petri Nets Tools and Embedded Systems Design"; PNSE'07- International Workshop on Petri Nets and Software Engineering, Siedlce, Polónia, 25-26 de Junho, 2007

[Lino, 03] Lino, Rui Manuel Gonçalves, "Detecção de Falhas em Sistemas de Automação utilizando Redes de Petri", Dissertação de Mestrado, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2003.

[Mealy, 55] Mealy, George H., "Method for Synthesizing Sequential Circuits" , Bell System Technical Journal 34: 1045–1079, 1955

[Moore, 56] Moore, Edward F., "Gedanken-Experiments on Sequential Machines", Automata Studies, New Jersey, Princeton, 1956

[Moutinho, 04] Moutinho, Filipe de Carvalho, "Controladores Domóticos em Rede", Relatório de Projecto de final de curso, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2004.

[Murata, 89] Murata, Tadao, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, No. 4, April 1989.

[Nunes, 08] Nunes, Ricardo Filipe Pires, "Editor de redes de Petri suportando especificações hierárquicas e modulares", Dissertação de Mestrado, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2008.

[Peterson, 77] Peterson, James L., "Petri Nets", Computing Surveys, Vol. 9, No.3, September 1977.

[Rebelo, 2010] Rebelo, Rogério, "Geração automática de Código ANSI C a partir de Redes de Petri IOPT – PNML2C, Dissertação de Mestrado, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2010.

[Reisig, 85] Reisig, Wolfgang, “Petri nets: an introduction”, Springer-Verlag New York, Inc., 1985

[Santana, 02] Santana, Simão Faro, “TINY DOMOT - Sistema de Controlo Domótico Distribuído”, Projecto Final de Curso, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2002.

Referências Electrónicas

[Fordesign, 07] FORDESIGN- Formal Methods for Embedded Systems Co-Design, 2007, R&D Project POSC/EIA/61364/2004 sponsored by FCT – Fundação para a Ciência e a Tecnologia, ultima visualização: 19 Outubro de 2009, disponível em: http://www.uninova.pt/fordesign/

Anexo 1 – Sinais utilizados

Sinais de entrada	**Tipo**
Mensagem_Recebida	boolean
Estado_TD_Registado	boolean
Estado_TD_Configurado	boolean
a_enviar_msg	boolean
Espera_ACK	boolean
End_Correcto	boolean
Tipo_Msg	**range**
Estado_TD_Programado	boolean

Quadro A1.1 – Sinais de entrada utilizados na versão base

Sinais de Saída	**Tipo**
ENVIA_MSG	boolean
VE_ENTRADAS	boolean
VE_EVENTOS	boolean
CONFIGURA	boolean
ACTUALIZA_EVENTOS	boolean
ACTUALIZADA_DATA_HORA	boolean
ACTUALIZA_FLAGS_SAIDA	boolean
REGISTA_TD	boolean
VE_ESTADO_TD	boolean
ACTUALIZA_ACK	boolean
ACTUALIZA_ESTADO_TRANSM	boolean
APAGA_REG	boolean
DEVOLVE_CONF	boolean
NOVA_CONF	boolean
APAGA_CONF	boolean
DEVOLVE_EVENTOS	boolean
APAGA_EVENTOS	boolean
INICIALIZACAO	boolean

Quadro A1.2 – Sinais de saída utilizados na versão base

Tipo de mensagem	Valor
Apaga Registo	0
Novo Registo	1
Apaga configuração	2
Nova configuração	3
Devolve configuração	4
Apaga Eventos	6
Novo evento	7
Devolve eventos	8
ACK	9
Actualiza data e hora	11
Devolver data e hora	12
Mensagem de acção	>19

Quadro A1.3 – Valor associado a cada tipo de mensagem

Anexo 2 - PDC

```
void PlatformDependentCode(void)
{

  //#################PORTOS####################
  TRISA=0xFF;//input
  TRISB=0x37;//inputs exepto os bits de programação, RB3, RB6 e RB7.
  TRISC=0x90;//outputs exepto, RX e RC4 que são inputs.
  TRISD=0x00;//outputs.
  TRISE=0x00;//outputs.

  PORTB=0xff;
  PORTC=0xff;
  PORTD=0xff;
  PORTE=0xff;


  //################TRANSMISSÃO#################
  //inicializa transmissão - 8bits/assincrono
  TXSTAbits.SYNC=0;//0-assincrono
  BAUDCONbits.BRG16=0;
  TXSTAbits.BRGH=1;//alta velocidade
  TXSTAbits.TX9=0;//8bits

  //inicializa a recepção
  RCSTAbits.SPEN=1;//serial port enable
  RCSTAbits.RX9=0;//8bits
  SPBRG=31;// Baud Rate=FOSC/[16(SPBRG+1)] ->9600
  //Inicialização do timer 2 com frequência de 2ms
  //Input clock= Fosc/4 = 4,9152MHz/4 = 1,2288 MHz
  //Com prescaler de 1:16 para um incremento do clock de entrada obtêm-se
  //a frequência de contagem de 1,2288MHz/16 = 76,8KHz
  //2ms corresponde a uma frequência de 500Hz
  //Valor do registo PR2: 76800Hz/500Hz=153.6
  PR2=154;      //frequência de interrupts de 2ms para cristal de 4MHz

  XSTAbits.TXEN=0;//transmit disable
  PIE1bits.TXIE=0;//disable dos int de transmissao
  RCSTAbits.CREN=1 ;//continuous receive on
  PIE1bits.RCIE=1;//enable dos int de recepcao


  //################TIMER#########################
  PIE1bits.TMR2IE=1;//enable dos int do timer 2
  T2CON=0b00000111; //com prescaler de 16 e switch on
  TMR2=0;


   //################INTERRUPTS###################
  //Inicialização dos interrupts
  INTCONbits.GIE=1;//enable dos int globais
  INTCONbits.PEIE=1;//enable dos int dos perifericos
  PIE2bits.EEIE=1; //Data EEPROM/Flash Write Operation Interrupt Enable
  INTCONbits.INT0IE = 1; //1 = Enables the INT0 external interrupt
}
```

Figura A2.1 – Código dependente da plataforma