UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
PROGRAMA DE PÓS-GRADUAÇÃO STRICTO SENSU EM COMPUTAÇÃO APLICADA

DISSERTAÇÃO DE MESTRADO

ICARU-FB: UMA INFRAESTRUTURA DE SOFTWARE ADERENTE À NORMA IEC 61499

LEANDRO ISRAEL PINTO

JOINVILLE, 2014

LEANDRO ISRAEL PINTO

ICARU-FB: UMA INFRAESTRUTURA DE SOFTWARE ADERENTE À NORMA IEC 61499

Dissertação submetida ao Curso de Pós-Graduação Stricto Sensu em Computação Aplicada, no Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Mestre em Computação Aplicada.

Orientador: Dr. Cristiano Damiani Vasconcellos
Coorientador: Dr. Roberto Silvio Ubertino Rosso Jr.

JOINVILLE
2014

P659i Pinto, Leandro Israel
ICARU-FB: Uma Infraestrutura de Software Aderente à Norma IEC 61499/ Leandro Israel Pinto. – 2014.
126 p. : il. ; 21 cm

Orientador: Cristiano Damiani Vasconcellos
Bibliografia: p. 103-106
Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de pós-graduação em Computação Aplicada, Joinville, 2014.

1. Automação e Controle. 2. IEC 61499. I. Pinto, Leandro Israel. II. Vasconcellos, Cristiano Damiani. III. Rosso, Roberto Silvio Ubertino Jr. IV. Universidade do Estado de Santa Catarina. Programa de pós-graduação em Computação Aplicada.

CDD: 005.1 – 20.ed.

LEANDRO ISRAEL PINTO

ICARU-FB: UMA INFRAESTRUTURA DE SOFTWARE

ADERENTE À NORMA IEC 61499

Dissertação apresentada ao Curso de Mestrado Acadêmico Computação Aplicada como requisito parcial para obtenção do título de Mestre em Computação Aplicada na área de concentração "Ciência da Computação".

Banca Examinadora

Orientador:

Prof. Dr. Cristiano Damiani Vasconcellos
CCT/UDESC

Coorientador:

Prof. Dr. Roberto Silvio Ubertino Rosso Junior
CCT/UDESC

Membros

Prof. Dr. Marco Aurélio Wehrmeister
UTFPR

Prof. Dr. André Rauber Du Bois
UFPEL

Joinville,SC, 25 de junho de 2014.

AGRADECIMENTOS

Ao meu orientador, professor Cristiano Damiani Vasconcellos, pela sua colaboração no desenvolvimento desse trabalho e incentivo. Ao professor Roberto Ubertino Rosso Jr., coorientador, pelo apoio e incentivo.

Ao bolsista Gabriel Negri pelo trabalho no desenvolvimento da ferramenta visual aderente a norma IEC 61499 e apoio no decorrer desse trabalho.

Aos meus pais pelo apoio e incentivo.

E a minha noiva pelo grande amor e incentivo dados.

RESUMO

PINTO, Leandro. **ICARU-FB: Uma Infraestrutura de Software Aderente à Norma IEC 61499**. 2014. 127f. Dissertação (Mestrado Acadêmico em Computação Aplicada - Área: Engenharia de Software) - Universidade do Estado de Santa Catarina. Programa de Pós-Graduação em Computação Aplicada, Joinville, 2014.

A norma IEC 61499 define um modelo de desenvolvimento para automação e controle industrial, ela estabelece uma linguagem visual que pode facilitar a implementação de sistemas de controle distribuídos. Esse trabalho apresenta a proposta e implementação do ICARU-FB, um ambiente *Open Source* multiplataforma, que é capaz de executar a linguagem definida na norma IEC 61499 em arquiteturas com poucos recursos computacionais. Uma máquina virtual foi projetada e implementada para executar redes blocos de funções em plataformas de 8 bits com o mínimo de recursos. Ela também foi portada para executar em um computador de 64 bits. Dois estudos de caso foram realizados a fim de verificar a conformidade com a norma IEC 61499. Através desses estudos de caso, foi verificado que é possível atender aos requisitos da norma IEC 61499, como configurabilidade, interoperabilidade e portabilidade. Os estudos de caso também demonstraram a habilidade do ambiente de reconfigurar o software em tempo de execução.

Palavras-chave: IEC 61499, Blocos de Funções, Ambiente de Execução, CLP, Automação e Controle.

ABSTRACT

The IEC 61499 standard defines a development model for industrial automation and control, it defines a visual language that can facilitate the implementation of distributed control system. This work presents the proposal and implementation of ICARU-FB, an Open Source multi-platform environment, capable of running the function blocks defined in IEC 61499 on architectures with few computational resources. A virtual machine was designed and implemented to perform networks of function blocks on 8-bit platforms with minimal resources. It was also ported to run on a 64-bit computer. Two case studies were performed in order to verify compliance with IEC 61499. Through the case studies, it was verified that it is possible to meet the requirements of the standard, such as configurability, interoperability and portability. The case studies also demonstrated the ability of the environment to reconfigure the software at runtime.

Key-words: IEC 61499, Runtime Environment, PLC, Automation and control.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A forma com que se desenvolve sistemas de automação está em constante evolução. A possibilidade de reconfiguração dos sistemas sem a necessidade de seu completo desligamento e reinicialização tem se tornado um requisito cada vez mais importante. Sistemas de automação residencial ou predial, fornecimento de energia e alarmes são exemplos de sistemas onde o completo desligamento deve ser minimizado. Em casos como linhas de produção, alterações do sistema podem ser necessárias em função de customizações no produto. Um mecanismo que permita uma adaptação eficiente a novos produtos, pode garantir que a linha de produção fabrique produtos customizados com velocidade semelhante a fabricação de produtos padronizados. Essa adaptação na linha pode incluir desde alterações físicas, como a inclusão ou remoção de novos dispositivos, até alterações de software. Um exemplo é mostrado em Vyatkin (2011), no qual uma linha de produção automatizada é capaz de produzir sapatos customizados com velocidade de produção em massa.

Normalmente, os sistemas de automação são projetados utilizando controladores lógicos programáveis (CLPs). Sistemas de grande porte possuem um considerável número de CLPs, podendo ser caracterizados como sistemas distribuídos. Esses CLPs se comunicam via uma ou mais redes e controlam vários sensores e atuadores, o que normalmente ocasiona dificuldades para futuras modificações e extensões. A dificuldade inerente à implementação de sistemas distribuídos, assim como a incompatibilidade entre dispositivos e ferramentas de diferentes fabricantes, podem dificultar uma ágil adaptação da linha de produção a novos produtos, geralmente tornando a empresa totalmente dependente de um único fabricante.

A norma IEC 61499 (VYATKIN, 2012) fornece uma alternativa de implementação para esse tipo de sistema, propondo um mecanismo capaz de diminuir a complexidade da implementação de sistemas distribuídos e garantir a compatibilidade entre ferramentas de software e dispositivos de diferentes fabricantes. A norma também propõe mecanismos que facilitam a alteração em tempo de execução, e foi definida com a finalidade de padronizar o desenvolvimento de sistemas para automação industrial distribuída, utilizando uma abordagem orientada a componentes (blocos de funções), com a qual é possível descrever um sistema inteiro, independente da plataforma de execução (YOONG; ROOP; SALCIC, 2009). A IEC 61499 estende o conceito de bloco de função, definido na norma IEC 61131-3, tornando-o mais adequado ao desenvolvimento de sistemas distribuídos (CHOUINARD; BRENNAN, 2006). Reduzir a complexidade de programação para um

sistema de controle distribuído (DCS - *Distributed Control System*) é um dos principais objetivos da IEC 61499 (VYATKIN, 2009; ZOITL et al., 2005).

Diferentes ambientes de execução foram propostos para executar os blocos de funções da norma IEC 61499, destacam-se o FBRT (HOLOBLOCK Inc., 2014) e o FORTE (4DIAC-CONSORTIUM, 2014) que são iniciativas abertas. Entretanto, para a execução desses ambientes é necessário um sistema operacional. Atualmente, a menor plataforma para execução do FORTE é composta por um processador ARM7 (ARM Ltd., 2014) e o sistema operacional eCos (eCosCentric Ltd., 2014). Não parece viável, considerando os custos, utilizar essa plataforma para operar alguns poucos sensores e atuadores, um problema que pode ser resolvido com microcontroladores mais simples de 8-bits. A descentralização de um sistema para automação implica também em distribuição das tarefas entre os dispositivos, consequentemente, esses dispositivos podem ser bastante simples e com pouca capacidade de processamento e recursos. Isso não é possível utilizando os ambientes aderentes a norma IEC 61499 atualmente. Muitos desses ambientes utilizam vários processos ou *threads* para executar as especificações definidas pela norma, porém, é necessária uma considerável capacidade de processamento e memória para executar um sistema operacional que gerencie esses processos.

Muitos desses ambientes de execução são implementados em linguagem Java e executados pela máquina virtual Java (LINDHOLM et al., 2014). Implementar a reconfiguração dinâmica, que é a capacidade de alterar o software em tempo de execução, em linguagem Java não é uma tarefa simples, não sendo suportado pelos ambientes que adotam essa solução.

Nesse trabalho é apresentado o projeto e implementação de um ambiente para execução aderente a norma IEC 61499 que pode ser executado em hardwares como o Arduino ATmega2560 (ARDUINO, 2014), que utiliza um microcontrolador de 8-bits. O principal objetivo é demonstrar a viabilidade da adoção da norma para o desenvolvimento em ambientes de baixo custo. O ambiente proposto suporta a reconfiguração dinâmica, conforme previsto na norma. O código fonte e exemplos de aplicações desse trabalho estão disponíveis em Pinto (2014).

1.1 OBJETIVOS

O objetivo desse trabalho é propor e implementar um ambiente de execução aderente a norma IEC 61499 que execute em plataformas de hardware com baixo custo, e menor capacidade de processamento e recursos.

Para isso foi desenvolvido um ambiente composto de um pré-processador, um compilador, uma máquina virtual e uma ferramenta de atualização dinâmica. Sendo a máquina virtual projetada para executar em dispositivos com poucos recursos como o Arduino ATmega2560. As principais contribuições desse trabalho são:

- O desenvolvimento de um pré-processador e um compilador para os blocos funções, conforme definidos na norma IEC 61499;
- O desenvolvimento de uma máquina virtual capaz de executar essas descrições (blocos funções);
- O desenvolvimento de uma ferramenta de atualização dinâmica para alterar, em tempo de execução, os blocos de funções na máquina virtual;
- A implementação de um CLP com a infraestrutura proposta.

Para validar o funcionamento do ambiente foram implementados dois estudos de casos usando editores de blocos de funções, esses editores geram especificações em *xml* conforme definido na norma IEC 61499.

1.2 ORGANIZAÇÃO DO TRABALHO

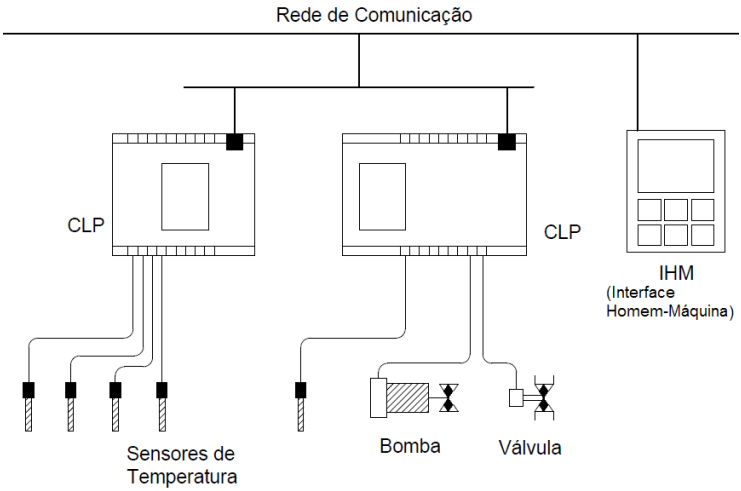
Os demais capítulos estão organizados na seguinte forma: No Capítulo 2 é feita uma introdução a norma IEC 61499, são descritas, por meio de exemplos, suas principais características e são apresentados alguns ambientes de execução que satisfazem essa norma, também são discutidas as vantagens e limitações de cada um desses ambientes. O Capítulo 3 apresenta a proposta para uma infraestrutura de software de um CLP aderente a norma IEC 61499 e alguns detalhes de sua implementação. No Capítulo 4 são apresentados dois experimentos realizados e os resultados verificados nos testes de execução. Por fim, no Capítulo 5 tem-se a conclusão, contribuições e trabalhos futuros.

2 REVISÃO DA LITERATURA

Inicialmente, os CLPs eram programados através de linguagens proprietárias oferecidas pelos fornecedores do equipamento, porém, com os usuários demandando uma solução de software mais aberta, uma nova geração de controladores está surgindo. A tendência é que esses novos sistemas adotem uma solução distribuída, onde pequenos dispositivos possam ser programados e conectados diretamente a rede industrial. Podendo, dessa forma, se comunicar com dispositivos de maior capacidade de processamento, como dispositivos de interface homem-máquina ou computadores.

A norma IEC 61131 (TIEGELKAMP, 2010), que é anterior a norma IEC 61499, padroniza um conjunto de linguagens de programação para os CLPs. Esse padrão é implementado pela maioria dos fabricantes de CLPs e tem grande aceitação na automação. Entretanto, essa norma não garante a compatibilidade entre diferentes fabricantes, pois apesar de utilizarem a mesma linguagem de programação, os ambientes de desenvolvimentos geralmente não são compatíveis. Assim, uma solução implementada através das ferramentas de um determinado fabricante, geralmente não pode ser alterada por ferramentas de outros e nem executada através de dispositivos que não sejam do mesmo fabricante. Um sistema de automação normalmente é constituído por um ou mais CLPs como apresenta a Figura 1.

Figura 1 – Sistema de controle com CLPs.



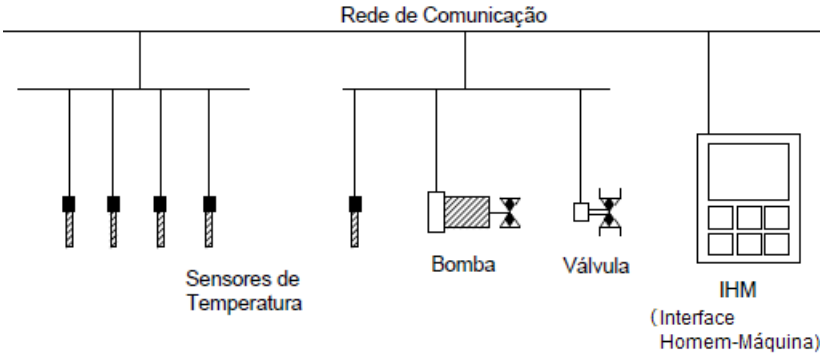
Fonte: Adaptado de Lewis (2008).

Nesse exemplo, tem-se dispositivos (CLPs) capazes de controlar vários processos simultaneamente. Vários sensores e atuadores são conectados ao mesmo dispositivo, o qual irá executar diversos algoritmos para atender aos requisitos da aplicação. A alteração de um único algoritmo nesse sistema, implica em desligar pelo menos um dos dispositivos, o que pode parar vários outros algoritmos sem necessidade. Além disso, desligá-lo pode não ter sido previsto pelo sistema, portanto, isso pode causar danos ao sistema. Então, opta-se por desligar o sistema por completo para garantir que nada será danificado.

Outro fator que fará com que o sistema seja desligado é a alteração do hardware, que pode ser causada por defeitos no equipamento, atualização dos dispositivos, alteração no hardware do sistema para atender a novos requisitos, etc. Quanto mais centralizado é o sistema, maior o impacto dessas alterações, pois a troca de um dispositivo, por exemplo, implica em desligar no mínimo toda a parte do sistema que o mesmo controla. Assim, é necessária uma configuração mais eficiente para esse tipo de requisito.

Um exemplo dessa configuração é apresentado na Figura 2. Nesse exemplo, o controle do sistema acontece de forma distribuída, sendo que cada dispositivo possui seu software de controle. Cada atuador ou sensor pode processar o seu próprio sinal através de algoritmos criados pelo usuário. Segundo Lewis (2008), para atender a esses requisitos, de integração e flexibilidade dos sistema de produção, será necessário uma nova abordagem no projeto de software.

Figura 2 – Funcionalidade distribuída avançada.



Fonte: Adaptado de Lewis (2008).

A norma IEC 61499 define o bloco de funções, que é um bloco capaz de encapsular código, sua execução é orientada a eventos, o que torna a interligação de componentes de software mais simples, pois as interfaces entre esses blocos são eventos e não métodos. Os blocos de funções também podem abstrair várias camadas de software. Assim, a adoção desse tipo de abstração apresenta vantagens, especialmente para os usuários finais. O que inclui, por exemplo, maior produtividade no desenvolvimento de software, proporcionado pelo reuso de soluções padronizadas, e maior flexibilidade através capacidade de adicionar componentes de forma *plug-and-play*.

2.1 A NORMA IEC 61131

A norma IEC 61131 (IEC, 2003) foi criada para uniformizar a programação de controladores lógicos programáveis (CLP). Vários esforços anteriores foram feitos para tentar padronizar a programação de CLPs, a norma IEC 848 que foi publicada em 1977, por exemplo, teve várias definições importadas para a norma IEC 61131, como a linguagem *Sequential Function Chart*. A primeira publicação da IEC 61131 ocorreu em seguida em 1979 (TIEGELKAMP, 2010).

A organização de um programa, escrito em conformidade com a norma IEC 61131, é feita através de POUs (*Program Organization Unit*) que são as menores unidades do programa. Uma POU pode ser escrita utilizando qualquer uma das cinco linguagens definidas na norma IEC 61131. Existem três tipos de POUs:

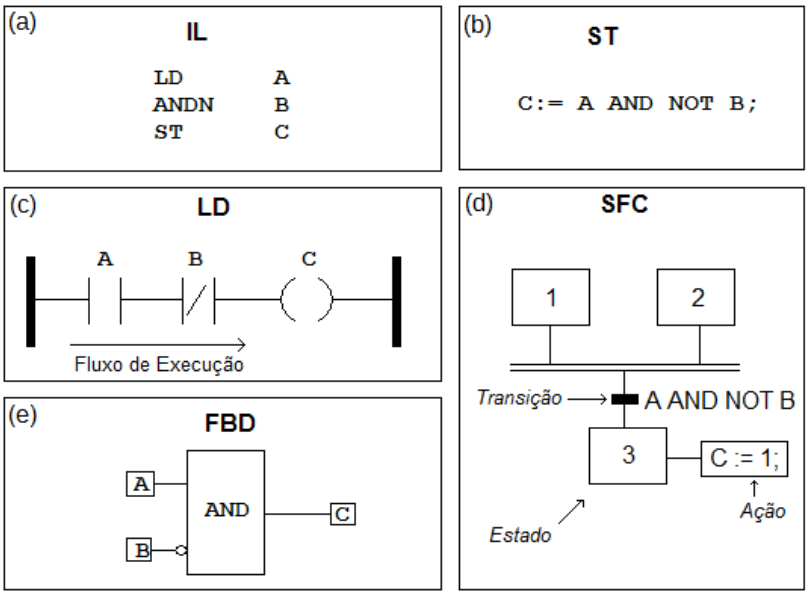
***Function* (FUN):** Uma FUN é uma função que sempre produz o mesmo resultado quando chamada com a mesma entrada (parâmetros), ela não tem nenhum mecanismo que recorde a execução anterior. A FUN pode fazer chamadas somente a outras FUNs;

***Function Block* (FB):** A principal diferença de FB para FUN é que o FB não perde os dados da execução anterior, ele pode produzir saídas diferentes mesmo recebendo os mesmos parâmetros. O FB pode fazer chamadas a outros FBs e também a FUNs;

***Program* (PROG):** Representa o maior nível na organização do programa. Além de ser capaz de chamar FBs e FUNs, o PROG tem acesso as entradas e saídas do dispositivo e pode conceder esse acesso as demais POUs.

As cinco linguagens definidas pela norma IEC 61131 são: *Ladder Diagram* (LD), *Function Block Diagram* (FBD), *Sequential Function Chart* (SFC), *Structured Text* (ST) e *Instruction List* (IL). As linguagens *Ladder Diagram*, *Function Block Diagram* e *Sequential Function Chart* são linguagens visuais, enquanto que as demais são textuais. Para demonstrar as características de cada linguagem, tem-se a Figura 3.

Figura 3 – As cinco linguagens da norma IEC 61131.



A linguagem LD se assemelha a um diagrama elétrico no qual uma entrada é representada por um símbolo de contato e uma saída é representada por um símbolo de bobina. Nessa linguagem, considera-se um fluxo de execução da esquerda para a direita, se não houver nenhuma interrupção do fluxo, então a saída será ativada. Uma interrupção é um contato que está aberto, ou seja, tem valor booleano falso.

Na linguagem FBD, a construção do algoritmo é semelhante a construção de um diagrama elétrico utilizando componentes. Os componentes são na verdade FBs ou FUNs que são representados em blocos, a esquerda do bloco são colocadas as entradas e a direita as saídas.

A linguagem SFC é semelhante a uma rede de Petri, nela tem-se estados e transições. Em cada estado pode-se ter uma ou mais ações a serem

executadas, como uma atribuição ou chamadas de FUNs ou FBs. A mudança de estado se dá por transições que ocorrem mediante a uma condição que é uma expressão booleana.

A linguagem IL se assemelha a uma linguagem de montagem, ela pode operar somente sobre as variáveis definidas dentro da POU, assim como todas as demais linguagens. A linguagem ST é uma linguagem de texto estruturado, ela permite, por exemplo, a separação do código em funções. A Figura 4 mostra uma aplicação escrita totalmente em linguagem ST, as POUs também poderiam ter sido escritas em outras linguagens da norma IEC 61131.

Figura 4 – Bloco de função da norma IEC 61131 em linguagem ST.

```
(*Conta os pulsos em COUNT*)
 FUNCTION_BLOCK COUNTER
   VAR_INPUT
     COUNT: BOOL;
     RESET: BOOL;
   END_VAR
   VAR_OUTPUT
     OUT : INT := 0;
   END_VAR
   VAR
     counted: BOOL;
   END_VAR
   IF(COUNT AND NOT counted)
   THEN
     OUT := OUT + 1;
     counted := TRUE;
   END_IF;
   IF(NOT COUNT) THEN
     counted := FALSE;
   END_IF;
   IF(RESET) THEN
     OUT:=0;
   END_IF;
 END_FUNCTION_BLOCK

PROGRAM ProgramCounter
  VAR
    HW_INPUT AT %IX2.3: BOOL;
    HW_OUTPUT AT %IY2.3: BOOL;
    COUNTER1: COUNTER;
  END_VAR
  (* Inverte o estado de uma
     saida sempre que ocorrem
     10 pulsos numa entrada.
  *)
  COUNTER1(COUNT:=HW_INPUT,
           RESET:=0);
  IF(COUNTER1.OUT > 10) THEN
    HW_OUTPUT := NOT HW_OUTPUT;
    COUNTER1(RESET:=1);
  END_IF;
END_PROGRAM

CONFIGURATION
  RESOURCE Res_1 ON CPU001
    TASK T_1 (
      INTERVAL := T#80ms,
      PRIORITY := 4);
    PROGRAM PCounter1
      WITH T_1: ProgramCounter();
  END_RESOURCE
END_CONFIGURATION
```

Cada CLP pode ser constituído por múltiplas unidades de processamento que são chamadas de *resources* pela norma IEC 61131. Vários programas (PROG) podem executar no mesmo *resource*, eles diferem em

prioridade e tipo de execução que pode ser periódica ou por interrupção. Esse tipo de configuração é feita de forma textual em linguagem ST.

Na linha 44 da Figura 4, é definido o campo de configuração do sistema através da palavra chave CONFIGURATION. Nesse exemplo tem-se uma CPU como um *resource* que executa uma tarefa T_1 periodicamente em intervalos de 80 millisegundos. O programa *Pcounter1* é uma instância do programa *ProgramCounter* que está associado a tarefa T_1.

Um programa para um CLP, de acordo com a norma IEC 61131, consiste em POUs fornecidas pelo fabricante ou criadas pelo usuário. Elas podem ser reutilizadas no mesmo projeto, em projetos diferentes e em CLPs diferentes. As FUNs e os FBs não possuem nenhuma relação com o hardware, pois declarações como a da linha 28 e 29 da Figura 4, que declaram variáveis endereçadas ao hardware, só podem ser feitas na POU PROGRAM. Por fim, as POUs do tipo PROGRAM são atribuídas a tarefas que executam nos *resources* dos CLPs.

2.2 A NORMA IEC 61499

A norma IEC 61499 é definida de forma a satisfazer três requisitos principais:

- A portabilidade é a capacidade de executar o mesmo código em diferentes dispositivos e de interpretar os elementos do sistemas em diferentes ferramentas de software.
- A configurabilidade é a capacidade de alterar dinamicamente os dispositivos e os componentes de software por múltiplas ferramentas de software.
- A interoperabilidade é a capacidade dos dispositivos de operarem juntos para executar as funções de um sistema distribuído.

Esses requisitos são alcançados através de blocos de funções, um bloco de funções é uma abstração de código que pode ser executado em qualquer plataforma compatível com a IEC 61499. Ele é o elemento central dessa norma.

Os blocos de funções da norma IEC 61499 não são os mesmos blocos de função da norma IEC 61131. A seguir, é mostrado que o bloco de funções da norma IEC 61499 é diferente do bloco de função da norma IEC 61131. Existem três tipos de blocos de funções na norma IEC 61499:

***Basic Function Block*:** Esse tipo de bloco pode encapsular algoritmos e variáveis.

***Service Interface Function Block*:** Esse tipo de bloco fornece acesso ao ambiente externo. Através desse tipo de bloco é possível, por exemplo, ler sensores e estabelecer comunicação através de uma rede.

***Composite Function Block*:** O bloco de funções composto pode encapsular vários blocos de funções de qualquer tipo.

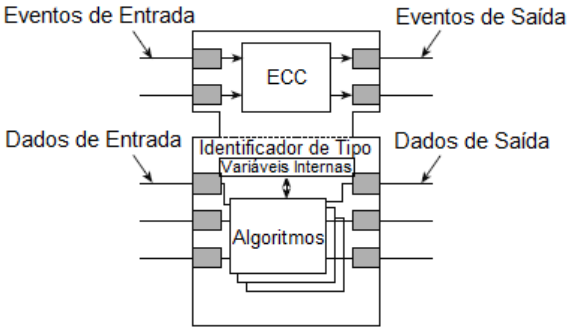
Além desses níveis de abstração existem também o *Resource*, *Device* e *System* que são descritos adiante.

Todas as definições do sistema são armazenados no formato *xml*, desde a implementação dos blocos até a configuração do sistema. A abstração em blocos de funções tem como objetivo facilitar a integração de vários componentes de hardware e software que fazem parte de um projeto. Permitindo o uso desses componentes sem a necessidade do conhecimento de sua implementação.

2.2.1 O Modelo de Bloco de Funções Básico

A interface de um bloco de funções é definida por um conjunto de eventos e dados, que podem ser de entrada ou saída. A representação de um bloco de funções é dividida em cabeçalho e corpo, como mostrado na Figura 5 (VYATKIN, 2009). No cabeçalho são definidos os eventos e um *ECC* (*Execution Control Chart*), o corpo contém as declarações de variáveis e a implementação dos algoritmos. A norma IEC 61499 não define uma linguagem específica para implementação dos algoritmos, a princípio, pode-se utilizar qualquer linguagem de programação, incluindo as linguagens descritas na norma IEC 61131-3 (TIEGELKAMP, 2010).

Figura 5 – Um bloco de funções básico.



Fonte: Adaptado de Lewis (2008).

Uma variável de evento pode assumir dois estados, verdadeiro ou falso. Essas variáveis controlam a execução do bloco de funções. Quando uma variável desse tipo é acionada (ou seja, tem seu valor atribuído como verdade) um dos algoritmos internos é executado. Geralmente o término desse algoritmo aciona um evento de saída para informar o fim da execução. O *ECC* é o elemento responsável por modelar esse fluxo de execução dentro dos blocos de funções.

O *ECC* é uma máquina de estados que gerencia a execução do bloco de funções. Os estados do *ECC* representam os possíveis estados do bloco de funções e as transições ocorrem quando um evento de entrada do bloco de funções é acionado. Quando o *ECC* muda de estado, podem ser executados algoritmos do corpo do bloco de funções que estão associados a esse estado do *ECC*. O término da execução de um algoritmo pode disparar eventos de saída.

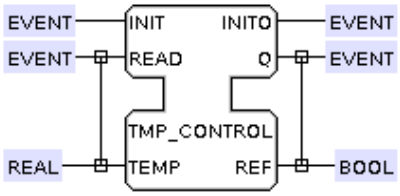
As variáveis de dados transportam informações de um bloco para outro. A execução de um algoritmo pode, por exemplo, requerer o valor de uma variável de entrada e atribuir algum valor a uma variável de saída.

Os algoritmos definidos no corpo do bloco de funções têm acesso a todas as variáveis de entrada e saída, com exceção das variáveis de evento que devem ser utilizadas somente para controle de execução. As variáveis de evento, declaradas no cabeçalho não devem ser conectadas as variáveis de dados do corpo.

A implementação de uma aplicação é feita através de vários blocos de funções conectados entre si. Para exemplificar o funcionamento de um bloco de funções, será utilizado como exemplo uma aplicação simples de controle de temperatura, essa aplicação consiste em tornar verdadeira uma

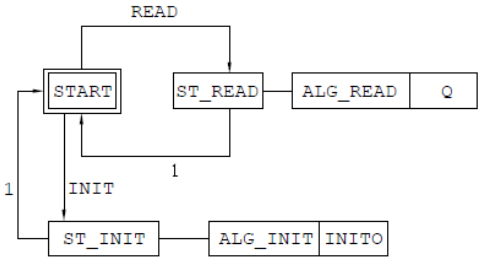
saída booleana (REF) se a temperatura de entrada (TEMP) for maior que 30º Celsius, e torná-la falsa se a entrada (TEMP) for menor que 20º. O objetivo desse bloco é acionar a refrigeração caso a temperatura atinja mais que 30º e desativar quando a temperatura estiver abaixo de 20º. A Figura 6 mostra a interface desse bloco de funções.

Figura 6 – Bloco de funções TMP_CONTROL.



A variável de evento INIT é utilizada para disparar a inicialização do bloco de funções. A variável de evento READ é utilizada para disparar o processamento da temperatura, essa variável está associada a variável TEMP, o que significa que quando esse evento ocorre, a variável TEMP deve ser transferida do bloco de funções de origem para o bloco atual. Para controlar esses eventos é necessário o *ECC* mostrado na Figura 7. Cada estado pode conter um ou mais algoritmos associados, os quais possuem uma ou mais variáveis de eventos de saída associadas. Quando uma transição ocorre, os algoritmos associados ao estado de destino são executados, ao término da execução de cada algoritmo, a variável de evento de saída associada é acionada.

Figura 7 – Um ECC com estados e transições.



Na Figura 7, sendo START o estado inicial do *ECC*, se um evento ocorrer na variável de evento de entrada INIT, fará com que o estado do *ECC*

mude para ST_INIT, o algoritmo ALG_INIT será executado e, após o término desse, a variável de evento INITO será disparada. INITO pode ser conectada ao INIT de outro bloco de funções, causando uma inicialização em cadeia. Isso é útil para definir uma ordem de inicialização quando a inicialização de um bloco de funções depende de outro.

Ao receber um estímulo na variável READ, o bloco de funções executa o algoritmo ALG_READ que lê a temperatura TEMP e calcula a saída REF adequada. Logo após, o evento Q é acionado indicando que REF pode ser lido. O retorno ao evento inicial START é automático, dada a condição de transição 1 de ST_READ para START e ST_INIT para START.

A representação em *xml*, segundo a norma IEC 61499, desse bloco de funções é mostrada na Figura 8

Figura 8 – Representação em *xml* de um bloco de funções.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBType >
<FBType Name="TMP_CONTROL" Comment= "Controle de
   temperatura básico" >
    <Identification Standard="61499-2"/>
    <VersionInfo Author="AUTOR"
       Date="16-09-2013"/>
    <CompilerInfo >
    </CompilerInfo>
    <InterfaceList>
        <EventInputs>
            <Event Name="INIT" >
            </Event>
            <Event Name="READ" >
                <With Var="TMP" />
            </Event>
        </EventInputs>
        <EventOutputs>
            <Event Name="INIT0" >
            </Event>
            <Event Name="Q" >
                <With Var="REF" />
            </Event>
        </EventOutputs>
        <InputVars>
            <VarDeclaration Name="TMP"
              Type="REAL" ArraySize="" />
        </InputVars>
        <OutputVars>
            <VarDeclaration Name="REF"
              Type="BOOL" ArraySize="" />
        </OutputVars>
    </InterfaceList>
    <BasicFB>
        <ECC>
            <ECState Name="START"
               x="320" y="200">
            </ECState>
            <ECState Name="ST_READ" x="323"
              y="276">
                <ECAction Algorithm="ALG_READ"
                  Output="Q"/>
            </ECState>
            <ECState Name="ST_INIT" x="439"
               y="200">
                <ECAction Algorithm="ALG_INIT"
                  Output="INIT0" />
            </ECState>
            <ECTransition Source="START"
              Destination="ST_INIT"
              Condition="INIT" />
            <ECTransition Source="START"
              Destination="ST_READ"
              Condition="READ" />
            <ECTransition Source="ST_READ"
              Destination="START" Condition="1"/>
            <ECTransition Source="ST_INIT"
              Destination="START" Condition="1"/>
        </ECC>
        <Algorithm Name="ALG_READ"  >
            <St Text="
              IF TMP > 30.0 THEN
                REF := TRUE;
              ELSIF TMP < 20.0 THEN
                REF := FALSE;
              END_IF"
            />
        </Algorithm>
        <Algorithm Name="ALG_INIT"  >
            <St Text="" />
        </Algorithm>
    </BasicFB>
</FBType>
```

A norma IEC 61499 define o formato *xml* para representação dos blocos de funções, isso permite que a especificação seja independente de plataforma e que o bloco possa ser alterado em qualquer ferramenta de edição. A linguagem utilizada para escrever os algoritmos na Figura 8 foi a ST (TIEGELKAMP, 2010), que é definida pela norma IEC 61131-3.

2.2.2 Service Interface Function Block (SIFB)

O *Service Interface Function Block* (SIFB) é um tipo de bloco de funções responsável por prover comunicação entre o programa aplicativo e o ambiente. Ele abstrai as dependências de hardware e permite ao desenvolvedor da aplicação focar na lógica da aplicação. A implementação de um SIFB requer conhecimento de baixo nível do hardware. Assim, os SIFBs não devem ser desenvolvidos pelos mesmos desenvolvedores da aplicação, ao invés disso, eles devem ser fornecidos pelos próprios fabricantes do equipamento (VYATKIN, 2012). A IEC 61499 sugere, mas não obriga, as seguintes entradas e saídas para os SIFBs:

Eventos de Entrada

INIT : Inicializa o serviço fornecido pelo SIFB;

REQ : Solicita o serviço fornecido pelo SIFB. Numa operação de escrita, a aplicação precisa ativar REQ para que a operação seja efetuada.

RSP : Solicita o serviço fornecido pelo SIFB. Numa operação de leitura, a aplicação ativa RSP a fim de receber uma resposta.

Eventos de Saída

INITO : Indica que a inicialização foi completada;

CNF : Esse evento confirma que uma operação de escrita foi concluída.

IND : Esse evento indica que uma operação de leitura foi concluída.

Dados de Entrada

QI : BOOL :Se essa entrada é verdadeira durante a ocorrência do evento INIT, a inicialização do serviço é requisitada. Caso contrário, a finalização do serviço é requisitada.

PARAMS : ANY : Parâmetros para inicialização do SIFB. Numa operação de escrita, por exemplo, PARAMS pode conter o endereço de escrita.

SD_1, ..., SD_m : ANY : Dados de entrada que serão transmitidos.

Dados de Saída

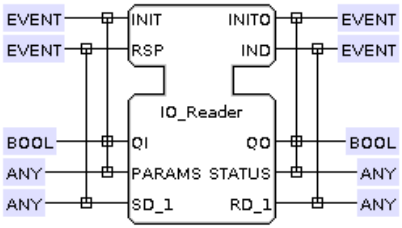
QO : BOOL : Quando INITO dispara e QO é verdadeira, então o serviço foi inicializado com sucesso. Se falso então o serviço não foi inicializado com sucesso.

STATUS : ANY : Essa saída será de um tipo apropriado para expressar o estado do serviço sobre a ocorrência de um evento de saída.

RD_1, ..., RD_n : ANY : Essas saídas são valores recebidos através da rede.

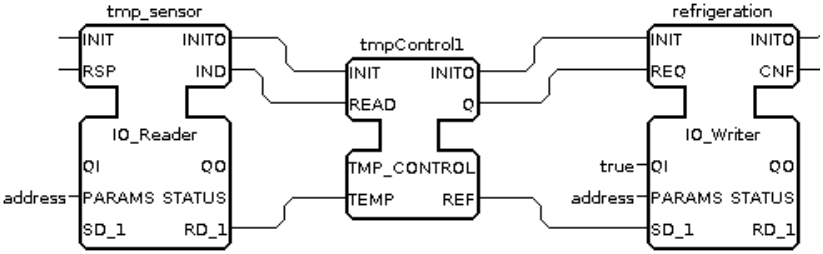
O IO_Reader é um tipo de SIFB capaz de executar uma leitura no hardware como, por exemplo, entradas digitais e sensores. A Figura 9 mostra um IO_Reader.

Figura 9 – Definição do SIFB IO_Reader.



Continuando o exemplo da Figura 6, pode-se acrescentar o IO_Reader para ler a entrada do sensor localizada no hardware do dispositivo. O mesmo pode ser feito para escrever em uma saída do hardware, nesse caso utiliza-se o SIFB IO_Writer. A nova configuração do exemplo é mostrada na Figura 10.

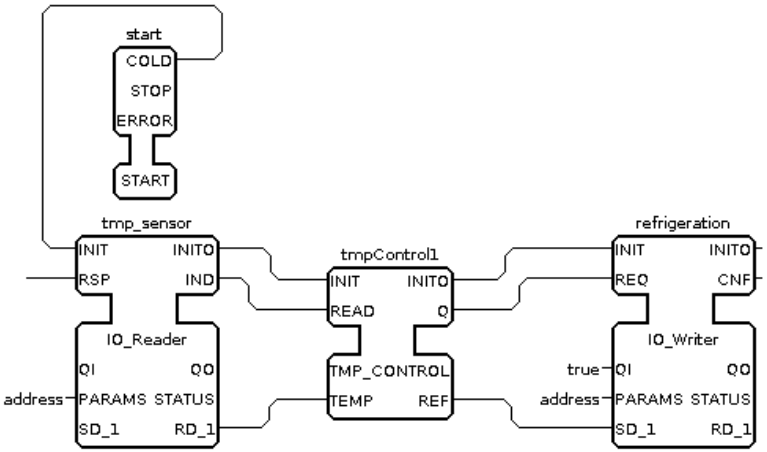
Figura 10 – SIFB IO_Reader e IO_Writer acrescentados a aplicação.



O bloco *tmp_sensor*, que é uma instância de IO_Reader, faz a leitura do sensor localizado no hardware. O bloco *tmpControl1* processa os dados obtidos por *tmp_sensor*, e o bloco *refrigeration* escreve numa saída do hardware, a qual está conectada a um sistema de refrigeração.

Como a norma IEC 61499 é baseado em eventos, faz-se necessário o disparo de um evento inicial no sistema. Isso é obtido através do bloco de funções chamado START. O ambiente de execução irá acionar esse bloco de funções durante sua inicialização, causando o início da aplicação. Acrescentado o bloco START a aplicação, tem-se a nova configuração na Figura 11.

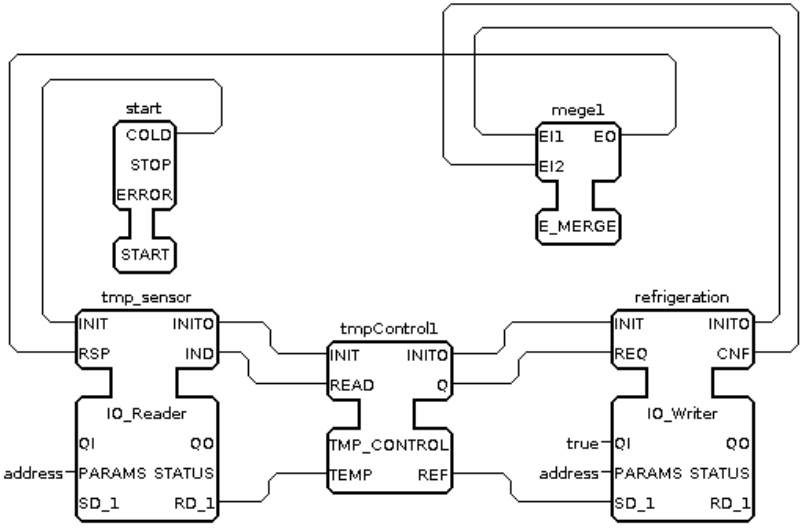
Figura 11 – O bloco de funções START acrescentado a aplicação.



O bloco de funções START fará com que todos os demais blocos de função sejam inicializados. Entretanto, para que a aplicação funcione, é necessário que ocorra um evento em RSP do bloco de funções *tmp_sensor*. Pode-se simplesmente conectar *refrigeration.INITO* em *tmp_sensor.RSP*, fazendo com que a aplicação executasse somente uma vez logo após a inicialização do bloco de funções *refrigeration*.

Portanto, além de conectar *refrigeration.INITO* em *tmp_sensor.RSP*, também é necessário conectar *refrigeration.CNF* em *tmp_sensor.RSP*. Isso faz com que o sistema execute um novo ciclo, logo após o término de um. A IEC 61499 fornece um bloco de funções chamado E_MERGE que deve ser utilizado quando se deseja unir dois eventos em um, no caso da Figura 11, os eventos *refrigeration.INITO* e *refrigeration.CNF* serão ambos conectados em *tmp_sensor.RSP*. A Figura 12 mostra a nova configuração do sistema com o uso de E_MERGE.

Figura 12 – Configuração completa da aplicação de controle de temperatura.

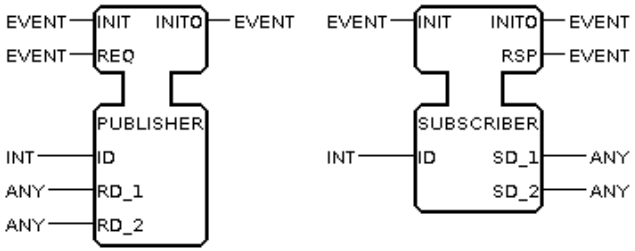


2.2.3 Comunicação

A comunicação é uma característica muito importante em sistemas distribuídos, os SIFBs PUBLISHER e SUBSCRIBER (Figura 13) representam um dos modelos possíveis de comunicação na norma IEC 61499.

Esse modelo consiste em um publicador de informações, o PUBLISHER, e os interessados nessa informação devem possuir um SUBSCRIBER cada.

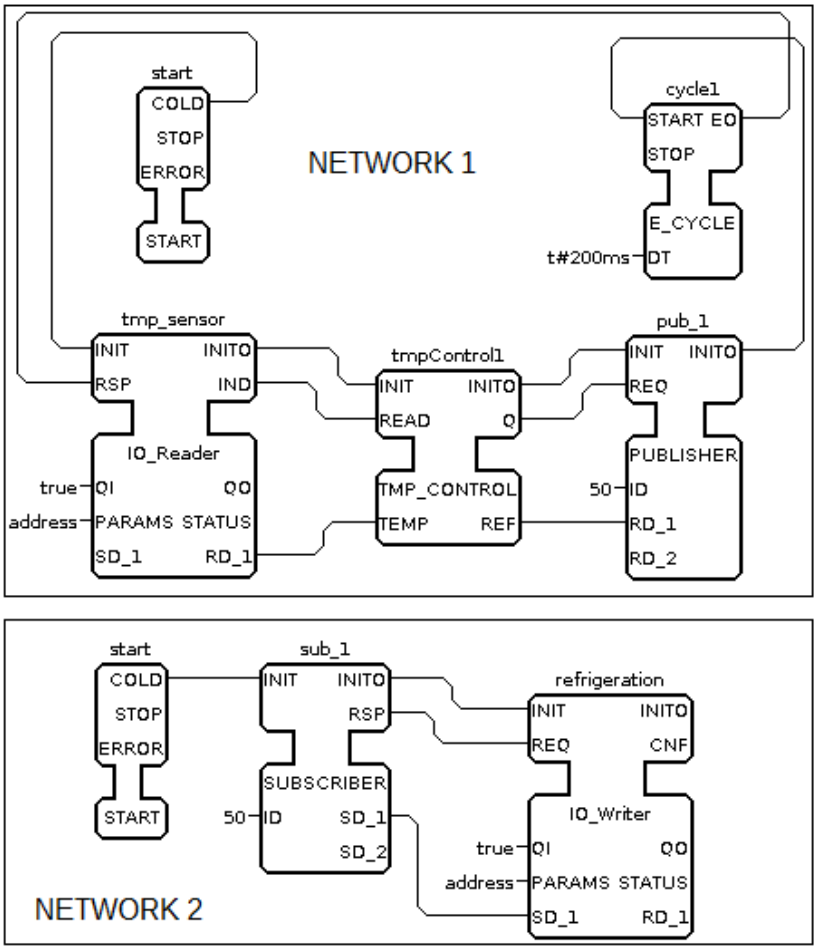
Figura 13 – Um PUBLISHER e um SUBSCRIBER.



Tanto o PUBLISHER quanto o SUBSCRIBER possuem uma variável de entrada chamada ID, SUBSCRIBERs que tenham um ID x, escutam somente informações oriundas de um PUBLISHER com o mesmo ID x.

Pode-se agora distribuir o exemplo da Figura 12 em várias redes de blocos de função. A Figura 14 mostra um exemplo onde a aplicação de controle de temperatura é dividia em duas redes (NETWORK 1 e NETWORK 2). Essa distribuição em rede permite que a aplicação seja executada em diferentes dispositivos, nesse exemplo, está se utilizando dois dispositivos.

Figura 14 – Aplicação distribuída através de PUBLISHERs e SUBSCRIBERs.

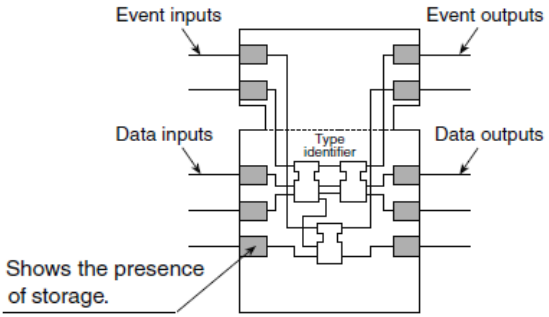


Na Figura 14, o PUBLISHER de ID 50 informa a aplicação que um valor de temperatura está pronto. O SUBSCRIBER correspondente (também de ID 50), que está em outra rede, escuta essa informação e ativa *refrigeration*. O bloco *cycle1* dispara um evento em *tmp_sensor* reiniciando o ciclo da aplicação.

2.2.4 Composite Function Block

O *Composite Function Block* é um tipo de bloco de funções que pode agrupar uma rede de blocos de função. A Figura 15 mostra a estrutura de um Composite function block (CFB). Esse tipo de bloco de funções é utilizado para abstrair uma rede de bloco de funções.

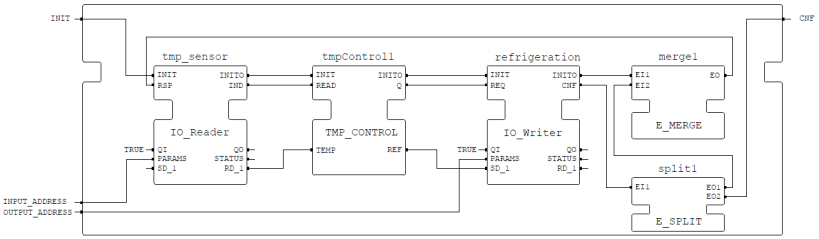
Figura 15 – Um *Composite function block*.



Fonte: (LEWIS, 2008)

O CFB modulariza parte do sistema. A aplicação mostrada na Figura 12 pode ser encapsulada em um CFB como mostra a Figura 16. Assim essa aplicação pode ser portada para outro sistema e replicada várias vezes.

Figura 16 – Aplicação dentro de um CFB.



A Figura 16 também mostra o bloco de funções E_SPLIT da norma IEC 61499. A função desse bloco é gerar dois eventos a partir de um. No exemplo, *split1* duplica o evento CNF de IO_Writer para que o mesmo seja

utilizando para reativar o ciclo e também para emitir uma saída do CFB indicando que um ciclo foi completado.

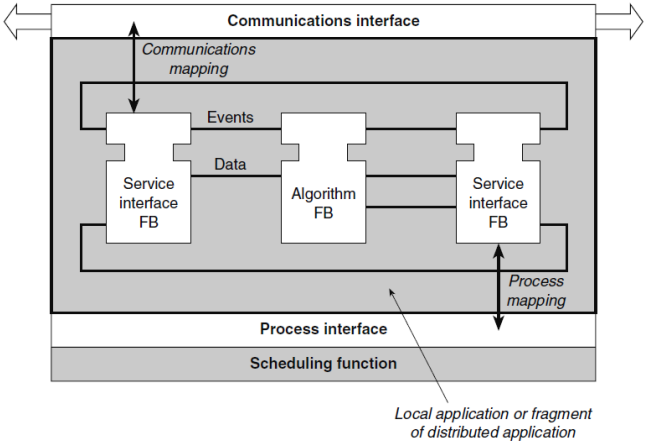
2.2.5 Modelo Resource

Para executar o programa aplicativo, mostrado na Figura 14, é necessário atribuí-lo a um ou mais RESOURCE.

Um RESOURCE é um contêiner para uma rede de blocos de função. Cada RESOURCE tem controle independente de sua operação, a função de um RESOURCE é receber e enviar dados ou eventos do ambiente e da interface de comunicação. Além disso, um RESOURCE fornece meios para execução dos blocos de função, como memória para dados e algoritmos (VYATKIN, 2012).

O modelo de um RESOURCE é mostrado na Figura 17.

Figura 17 – Modelo de um Resource.



Fonte: (LEWIS, 2008).

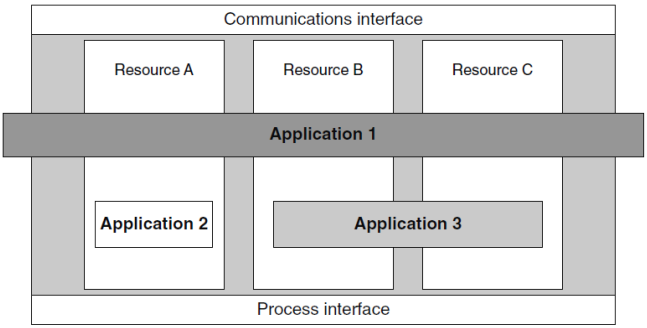
Através da interface de comunicação (*Communications Interface*) o RESOURCE permite que os SIFBs da aplicação se comuniquem. A interface *Process Interface* permite que a aplicação troque informações com o processo controlado, como por exemplo, ler sensores e acionar atuadores.

O RESOURCE pode ser uma abstração para uma *thread* ou um processador, por exemplo. Ele representa um recurso do dispositivo que é capaz de executar uma rede de blocos de função.

2.2.6 Modelo Device

O DEVICE é uma representação de qualquer equipamento de controle com a habilidade de encapsular os RESOURCES descritos anteriormente. A Figura 18 mostra o modelo de um DEVICE

Figura 18 – Modelo de um Device.



Fonte: Adaptado de Lewis (2008).

O DEVICE consiste numa interface de comunicação, uma interface de processo, um gerenciador de dispositivo e pode conter nenhum, um ou mais RESOURCES. A interface de comunicação fornece serviços de comunicação para o dispositivo e para suas aplicações. A interface de processo fornece serviços para acessar as entradas e saídas do hardware (sensores, atuadores e outros) (ZOITL et al., 2007).

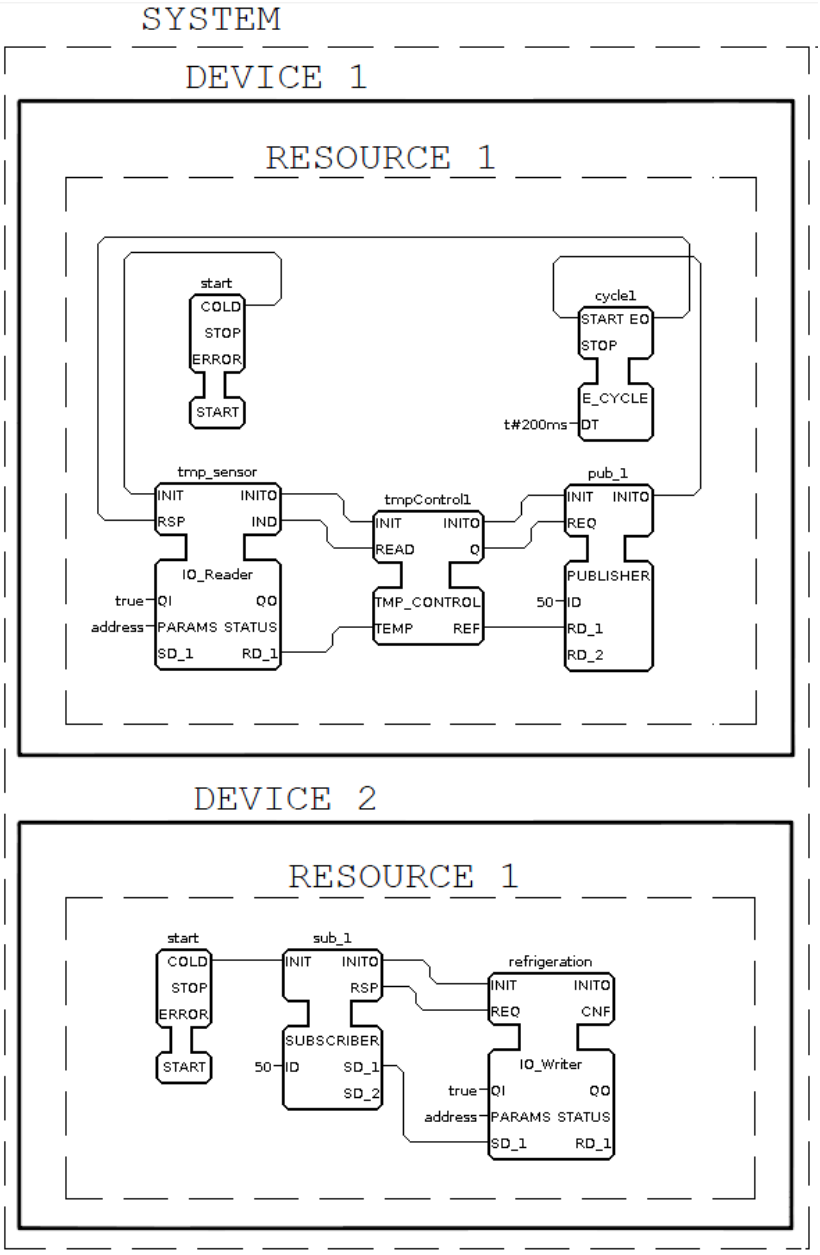
O DEVICE garante o funcionamento dos RESOURCES fornecendo serviços de segundo plano e tempo de execução para cada RESOURCE. Ele também pode criar, excluir e configurar RESOURCES sem interferir nos demais e suas aplicações.

2.2.7 Modelo System

Finalmente, tem-se o mais alto nível de abstração da IEC 61499, a visão do sistema chamada de SYSTEM. Nesse nível, são visíveis todos os dispositivos envolvidos no sistema. A aplicação de controle de temperatura é mostrada na Figura 19.

Na visão de sistema, pode-se verificar que o controle de temperatura está distribuído em dois dispositivos, e utiliza-se um RESOURCE de cada.

Figura 19 – Visão de sistema da aplicação de controle de temperatura.



2.3 AMBIENTES DE EXECUÇÃO ADERENTES A IEC 61499

Nessa seção são apresentados alguns ambientes de execução para a norma IEC 61499 e seus principais detalhes de implementação.

A maioria das soluções IEC 61499 utilizam algum tipo de ambiente de execução como o FORTE (4DIAC-CONSORTIUM, 2014) e o FBRT, que é o ambiente de execução do FBDK (HOLOBLOCK Inc., 2014). Esses ambientes, entretanto, são implementados cada um com um modelo de execução diferente.

Umas das principais dificuldades da implementação de ambientes de execução é com relação as diferentes interpretações da norma IEC 61499. Mesmo que as implementações dos ambientes de execução estejam de acordo com a norma IEC 61499, é possível que eles apresentem comportamentos diferentes durante a execução (VYATKIN, 2009; DUBININ; VYATKIN, 2012; CENGIC; AKESSON, 2010b). A seção 3.3 mostrará alguns desses problemas e as decisões tomadas nesse trabalho.

Ferrarini e Veber (2004) apresentam e comparam algumas abordagens para a implementação de um ambiente de execução. A seguir destacam-se duas possíveis implementações:

Implementação I1: Cada bloco de funções é implementado como um objeto, enquanto que todos os eventos são manipulados por um único objeto. Um bloco de funções contém as variáveis internas como atributos e os algoritmos como métodos e o ECC como o método $run()$. Todos os ECCs também possuem um método $initialize()$. O objeto que manipula eventos contém um método chamado $setEvent(event)$ que sequencialmente chama o método $run()$ dos blocos de função conectados a $event$. A geração de eventos de saída é implementada no método $run()$ do bloco de funções como uma chamada direta de $setEvent(event)$ do objeto que manipula eventos.

Implementação I2: Cada bloco de funções é implementado como um objeto que estende a classe base $Thread(Runnable)$. Esse objeto contém os mesmos atributos e métodos descritos em I1 e uma instância da classe $EventBuffer$. O objeto $EventBuffer$ é instanciado na chamada de $initialize()$ e implementa a interface de eventos de entrada. O método $setEvent(event)$ da classe $EventBuffer$ adiciona $event$ em uma lista de eventos ocorridos e executa a função $NotifyAll()$. A conexão de eventos é implementada como em I1, mas $setEvent(event)$ chama sequencialmente o método $setEvent(event)$ do $eventBuffer$ correspondente. Devido a implementação de $EventBuffer$, essa

operação termina em um intervalo de tempo pré-fixado, diferente do que acontece em I1.

Quando um ECC é "acordado" por $NotifyAll()$, ele lê sequencialmente a lista de eventos ocorridos até que uma transição possa ser apurada. Se nenhuma transição ocorrer, o ECC suspende sua execução.

As implementações I1 e I2 foram realizadas por Ferrarini e Veber (2004) em Java e comparadas entre si. I1 oferece vantagem com relação a velocidade de execução quando a rede de blocos de função é em série, mas apresenta comportamentos inesperados quando há um loop na conexão de eventos e quando mais de um evento precisa ser gerado ao mesmo tempo. A implementação I2 apresenta velocidade comparável a implementação I1 quando a rede é composta por blocos de função em paralelo e é a implementação que apresenta o comportamento mais esperado de acordo com a norma IEC 61499.

O FBRT (HOLOBLOCK Inc., 2014) foi o primeiro ambiente de execução implementado para a norma IEC 61499. É considerado uma implementação de referência e também utilizado para testar a norma, ele segue o modelo de implementação I1. Esse ambiente é implementado em Java (HALL; STARON; ZOITL, 2007).

No FBRT, um bloco de funções é implementado como um objeto. Os eventos de saídas que ativam outros blocos de função, são implementados como chamadas de função ao bloco de funções de destino, isso significa que a execução do bloco de funções atual é parada para que seja executada uma função de outro bloco de funções (YOONG; ROOP; SALCIC, 2009; FERRARINI; VEBER, 2004).

O ambiente de execução FORTE tem uma implementação semelhante à implementação I2. Segundo Strasser et al. (2008), o objetivo desse ambiente é ser uma solução aberta e que possa ser utilizada na indústria atendendo aos requisitos de portabilidade, configurabilidade e interoperabilidade.

O FORTE é implementado em C++. Os blocos de funções são implementados em classes, o disparo de um evento sobre um bloco de funções faz com que este seja colocado em uma fila do tipo *first-in-first-out*. Quando o bloco de funções que originou o evento terminar sua execução, o próximo bloco de funções na fila será executado.

O FBC, proposto em Yoong, Roop e Salcic (2009), é um compilador que utiliza como entrada um arquivo *xml* e gera código em linguagem C para ser compilado e executado sem a utilização de um ambiente de execução. Como mostrado por Yoong, Roop e Salcic (2009),

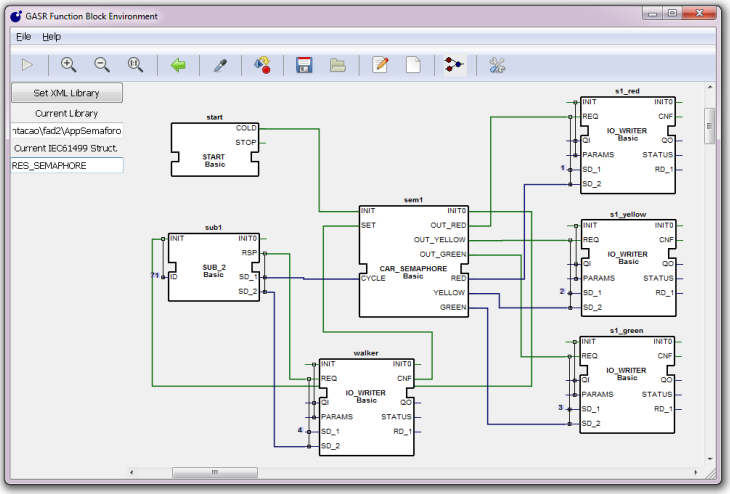
essa abordagem traz vantagens com relação ao desempenho, pois elimina a necessidade de um ambiente de execução.

O FBC implementa os blocos de função como estruturas em linguagem C e alguns métodos de execução. A estrutura de dados guarda o estado do ECC, eventos de entrada e saída e os dados de entrada e saída. As conexões entre os blocos de função são implementadas utilizando memória compartilhada (KUO et al., 2010).

Em Harbs (2012) foi desenvolvido um ambiente de execução para norma IEC 61499, também foram desenvolvidos uma interface gráfica para visualização da rede de blocos de função e um editor de blocos de função. Esse ambiente foi continuado em Negri (2013), o qual fez a junção das ferramentas desenvolvidas em Harbs (2012), essa integração passou a se chamar FBE.

O ambiente de execução e editor FBE (NEGRI, 2013), foi desenvolvido em linguagem Lua (IERUSALIMSCHY, 2006). A implementação consiste em uma classe FB, a qual representa um bloco de funções genérico, que executa a análise sintática, em um arquivo *xml*, identificando os blocos de função, esses blocos são instanciados cada qual por uma classe específica. Por exemplo, se um bloco de funções básico for identificado instancia-se um *basicFB*, as variáveis do bloco de funções tornam-se atributos da classe e os algoritmos tornam-se métodos. A Figura 20 mostra a interface do software desenvolvido em Harbs (2012) para visualização de blocos de função.

Figura 20 – Interface do Software GASR FBE Editor.



Esse editor de blocos de função permite visualizar e editar todos os níveis de um sistema. Ele gera arquivos *xml* de acordo com a norma IEC 61499.

Todos os ambientes de execução citados anteriormente necessitam de uma arquitetura de 32 bit para executarem, com exceção do FORTE que tem como foco executar em arquiteturas de 16 bit. A Tabela 1 faz uma comparação entre os ambientes de execução citados nessa seção.

Tabela 1 – Comparação entre ambientes existentes.

	FBRT	FORTE	FBC	FBE
Linguagem	Java	C++	C	Lua
Implementação dos blocos de função	Classes	Classes	Estruturas	Classes
Reconfiguração dinâmica	Não	Sim	Não	Não
Arquitetura	32 bits	16/32 bits	32 bits	32/64 bits
Plataforma Alvo	JVM	PC, PPC, ARM7	PC	PC

Além dos ambientes citados anteriormente, também existem ISaGRAf (ICS Triplex, 2014) e nxtRT61499F (nxtControl GmbH, 2014)

que são soluções fechadas e não puderam ser avaliadas como os demais ambientes. Entretanto, foi possível verificar que ambos os ambientes necessitam de um sistema operacional para o seu funcionamento.

2.3.1 Reconfiguração Dinâmica

Reconfiguração dinâmica é a habilidade do ambiente de execução de alterar partes do programa aplicativo e do hardware em tempo de execução. O suporte a reconfiguração dinâmica dos sistemas de automação futuros será um fator chave no sucesso de sistemas de produção ágeis . Novos paradigmas como *Produção flexível*, *Customização em massa* e *Produção Contínua* (*Zero Downtime Production*) surgiram como uma forma de reagir as novas exigências de clientes (ZOITL; SUNDER; TERZIC, 2006; STRASSER et al., 2006).

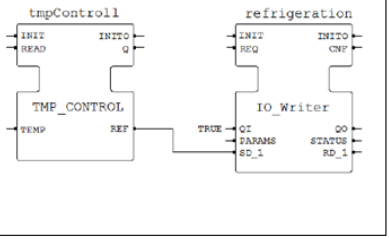
Dentre os ambientes apresentados na Tabela 1, somente o FORTE implementa a reconfiguração dinâmica.

Sistemas reconfiguráveis dinamicamente podem ser modificados sem a necessidade de recompilar todo o software ou reiniciar o sistema. O bloco de funções da IEC 61499 é próprio para esse tipo de aplicação, pois o ECC e os algoritmos fazem uso somente das variáveis do próprio bloco de funções.

A IEC 61499 não define uma lista de comandos para a reconfiguração dinâmica dos dispositivos, mas um perfil de conformidade (*compliance profile*) foi criado para uma tentativa de demonstrar a viabilidade de um conjunto conciso de comandos necessários (VYATKIN, 2012). A Figura 21 mostra um exemplo de comandos de gerenciamento enviados ao dispositivo para criar uma rede de blocos de função. Os comandos são enviados em formato *xml* ao dispositivo, o qual executa as ações solicitadas.

Figura 21 – Um exemplo de comandos de gerenciamento criando uma rede de blocos de função.

#	Comando
1	<Request ID="3" Action="Create"> <FB Name="tmpControl1" Type="TMP_CONTROL"/> </Request>
2	<Request ID="4" Action="CREATE"> <FB Name="refrigeration" Type="IO_Writer"/> </Request>
3	<Request ID="5" Action="CREATE"> <Connection Source="tmpControl1.REF" Destination="refrigeration.SD_1"/> </Request>
4	<Request ID="6" Action="WRITE"> <Connection Source="TRUE" Destination="refrigeration.QI"/> </Request>



Fonte: Adaptado de (VYATKIN, 2012).

O comando 1 cria uma instância do bloco de funções TMP_CONTROL com o nome *tmpControl1*. O comando 2 uma instância do bloco de funções IO_Writer com o nome *refrigeration*. O comando 3 cria uma conexão de dados. E o comando 4 cria uma conexão com uma constante TRUE.

Cada comando de gerenciamento é seguido de uma resposta do dispositivo, informando o resultado da execução do comando.

2.4 CONSIDERAÇÕES

A norma IEC 61499, que é sucessora da IEC 61131, fornece um mecanismo para implementar sistemas de automação distribuídos através do uso de blocos de funções. O bloco de funções da norma IEC 61499 é o componente chave para a reconfiguração dinâmica, as suas características, como interfaceamento baseado em eventos, permitem que o mesmo seja acoplado e desacoplado de uma rede ainda em execução. O fato dos atuais ambientes de execução transformarem o bloco de funções em classes, torna difícil aproveitar tais caraterísticas para implementar a reconfiguração.

3 SOLUÇÃO PROPOSTA

Alguns ambientes que suportam a norma IEC 61499, utilizam a linguagem Java e consequentemente a máquina virtual Java (LINDHOLM et al., 2014) para executar as aplicações. No contexto desse trabalho, essa abordagem não é viável em função da grande quantidade de recursos necessários para sua execução.

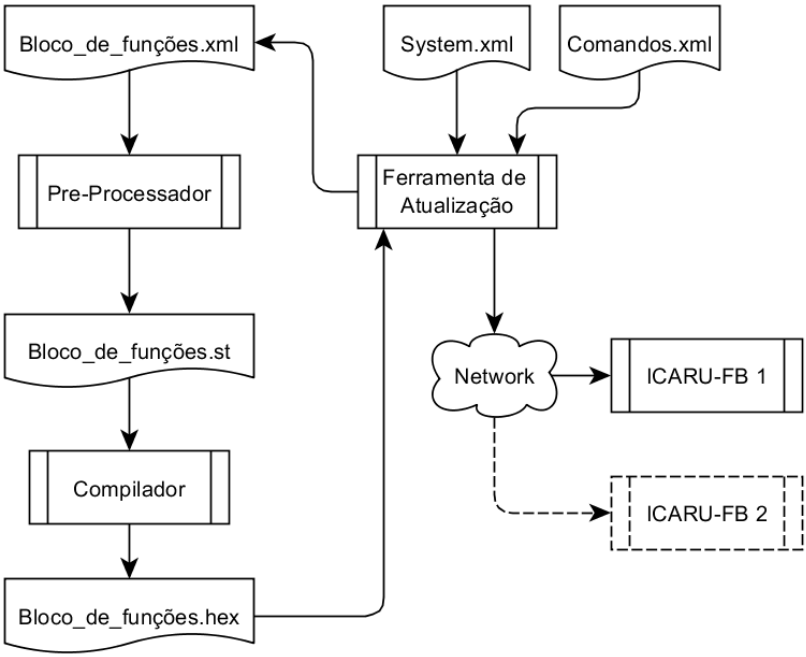
Como alternativa, é proposta a definição de uma máquina virtual mais simples e adaptada para a execução da norma IEC 61499. Que possa ser executada em processadores com menor capacidade de processamento e utilizando uma quantidade reduzida de recursos.

De forma semelhante a linguagem Java, onde o código gerado para cada classe é armazenado em arquivos distintos (*.class*), a máquina virtual proposta executa blocos de funções, também armazenados em arquivos separados, esse arquivos são gerados por um compilador a partir da definição *xml* do bloco de funções.

Além dos blocos de funções, a norma IEC 61499 define as conexões entre blocos, essas conexões são processadas separadamente pela máquina virtual, sendo armazenadas em forma de tabela em um arquivo distinto.

A máquina virtual desenvolvida permite ainda que os blocos e as conexões sejam alterados em tempo de execução. Ou seja, é possível que os blocos e conexões sejam criados ou excluídos durante a execução da aplicação. Esse gerenciamento é feito pela ferramenta responsável por gerenciar o processo de compilação, distribuição e atualização dos blocos de funções, ela se comunica com a máquina virtual através de mensagens e é executada separadamente em um computador. A Figura 22 mostra o diagrama da infraestrutura desenvolvida nesse trabalho. Os principais componentes da infraestrutura são: Ferramenta de Atualização, Pré-processador, Compilador e a máquina virtual chamada ICARU-FB.

Figura 22 – Diagrama da infraestrutura do ambiente de execução desenvolvido.



Os componentes *Function_Block.xml*, *System.xml* e *Commands.xml* são arquivos em formato *xml* de acordo com a norma IEC 61499. Eles são transformados pela *Ferramenta de Atualização* e pelo *Compilador* em uma aplicação que poderá ser executada por uma ou mais máquinas virtuais.

A ferramenta de atualização interpreta todos os arquivos *xml* e atualiza as máquinas virtuais de acordo com as informações desses arquivos. Arquivos que descrevem blocos de função são traduzidos, através de um pré-processador, de *xml* para a linguagem ST (Structured Text) (TIEGELKAMP, 2010) e enviadas ao compilador. Esse é responsável por gerar um arquivo executável pela máquina virtual (*.hex*).

Os arquivos *System.xml* e *Comandos.xml* contêm informações de configuração como, por exemplo, o que deve ser colocado em cada dispositivo ou *resource* e se algum comando de reconfiguração deve ser executado, como excluir ou incluir um bloco.

No contexto desse trabalho, um dispositivo é um hardware que executa a máquina virtual e que possui poucos recursos computacionais. A solução proposta utiliza o hardware livre *Arduino Mega2560* (ARDUINO, 2014), o qual possui um microcontrolador com 8 KB de memória SRAM, 256 KB de memória Flash e velocidade de 16 MHz.

3.1 A MÁQUINA VIRTUAL

Como ocorre na máquina virtual Java (LINDHOLM et al., 2014), a máquina virtual proposta é uma abstração de um processador que possui um conjunto de instruções próprio e fornece interface para acesso ao hardware. Essa máquina virtual foi projetada para executar blocos de funções padronizados pela norma IEC 61499. Um dos principais requisitos atendidos nesse projeto é que a máquina virtual pudesse ser executada em hardware com recursos limitados.

A execução dos blocos de funções difere da execução das classes da linguagem Java. A comunicação entre os objetos é feita através de chamadas de métodos, enquanto que nos blocos de funções os dados são transferidos de uma variável de origem para uma de destino. Para compilar uma classe é necessário que todas as suas dependências estejam presentes para o compilador, isso não acontece ao compilar um bloco de função, pois ele possui uma característica de funcionamento independente.

Uma opção para implementar os blocos de funções na linguagem Java de forma que seja possível a reconfiguração dinâmica, é o uso de recursos como *ClassLoader* e de reflexão (ARNOLD; GOSLING; HOLMES, 2006). Além disso, para manter a característica independente dos blocos de funções, a implementação precisa conter um mecanismo para execução das conexões entre os blocos, de tal forma que permita a inclusão ou remoção de blocos de funções e conexões.

Na literatura consultada, não foram encontradas implementações aderentes a norma IEC 61499 capazes de executar a reconfiguração dinâmica utilizando a linguagem Java.

A execução de um programa aplicativo em uma máquina virtual Java, inicia a partir do método *main*. As classes são instanciadas principalmente em métodos de outras classes através do comando *new* da linguagem Java, e o código da classe instanciada deve ser carregado para a execução. Na máquina virtual desse trabalho, todos os blocos de função são instanciados na inicialização e executados de forma independente, eles também devem permanecer sempre instanciados, pois o estado de cada bloco

de funções não pode ser perdido. Todo bloco possui um método *main*, que faz a verificação dos eventos de entrada, chamadas aos algoritmos internos e ativação dos eventos de saída.

Uma vez que o bloco de funções não faz chamadas a outros blocos e também não instancia os demais blocos, a máquina virtual ICARU-FB é quem precisa transmitir as variáveis de um bloco para outro. Portanto, além de processar os blocos de funções, a máquina virtual faz um processamento extra para as conexões. Essas conexões são armazenadas em um arquivo separado, e executadas de forma independente.

Essas características de execução de uma rede de blocos de função tornam mais fácil a manipulação da mesma em tempo de execução. A execução de um determinado bloco pode ser interrompida sem prejudicar outros blocos, assim como uma conexão pode ser acrescentada ou excluída.

O código objeto, gerado pelo compilador, é o mesmo para todas as implementações da máquina virtual ICARU-FB, independente de plataforma. Isso facilita a distribuição dos blocos e atende os requisitos de portabilidade e configurabilidade.

3.1.1 A Estrutura da Máquina Virtual

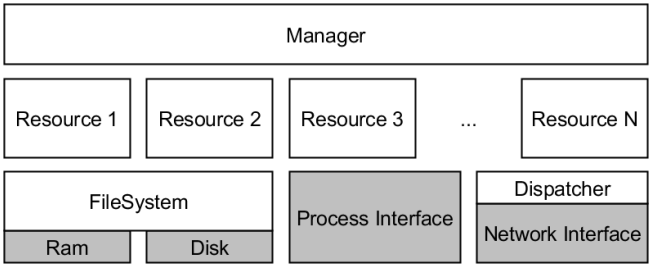
A máquina virtual é implementada em módulos, isso facilita que ela seja portada para diferentes plataformas. A implementação de alguns módulos não serão descritas nesse capítulo, pois variam de uma plataforma para outra. Entretanto, no Capítulo 4, serão mostradas duas possíveis implementações para tais módulos, sendo uma para PC e outra para um microcontrolador ATmega2560 (ATMEL, 2014).

A máquina virtual desse trabalho é projetada para executar diretamente sobre o hardware, sem utilizar nenhum tipo de sistema operacional. Assim, não há abstração da memória em arquivos e nem gerenciamento da memória utilizada por processos, o que significa que essas abstrações devem ser implementadas na máquina virtual.

Entretanto, adotar um sistema de arquivos convencional não é viável devido a quantidade de recursos necessários para execução desse. Por isso, a máquina virtual implementa um sistema de arquivos próprio e simplificado. Esse mesmo sistema de arquivos, também é utilizado pela máquina virtual para gerenciar a memória RAM. Assim os dados de cada instância de bloco de funções são armazenados em arquivos na memória RAM.

Cada instância de um bloco de funções possui o seu arquivo de dados e também uma referência ao arquivo do tipo de bloco, que é o arquivo que contém o ECC e os algoritmos desse bloco, compilados em instruções da máquina virtual. O arquivo do tipo de bloco é compartilhado entre todas as instâncias desse tipo de bloco. A Figura 23 mostra os módulos da máquina virtual.

Figura 23 – Estrutura da máquina virtual ICARU-FB.



Os módulos *Ram*, *Disk*, *Process Interface* e *Network Interface* são os módulos onde a implementação varia de acordo com a plataforma, eles fornecem uma abstração de hardware. E, normalmente, precisam ser alterados cada vez que a máquina virtual é portada para uma nova plataforma.

O módulo *Ram* abstrai o hardware da memória RAM, fornecendo métodos para atualização e consulta sobre essa memória que, do ponto de vista dos outros módulos, é apenas um vetor com *n* bytes. O funcionamento do módulo *Disk* é semelhante ao do módulo *Ram*, ele fornece os mesmos métodos *set* e *get*, mas atua sobre um tipo de memória não volátil.

O acesso ao meio físico é abstraído pelo módulo *Process Interface*, esse módulo permite que a máquina virtual acesse, por exemplo, sensores e atuadores. Quando a máquina executar sobre uma plataforma com interface gráfica, esse módulo pode permitir que a máquina virtual acesse elementos gráficos como botões e caixas de texto. Do ponto de vista dos outros módulos, esse módulo também é visto como um vetor de bytes assim como os módulos *Ram* e *Disk*, mas a alteração de um byte nesse vetor pode implicar por exemplo, no acionamento de um atuador ou de um indicador numa interface gráfica, assim como a leitura de um byte pode corresponder a leitura de um sensor ou de uma caixa de texto.

Os demais módulos da máquina virtual (*FileSystem*, *Dispatcher*, *Resource* e *Manager*) não possuem nenhuma dependência com relação a plataforma de execução.

O sistema de arquivos da máquina virtual é implementado no módulo *FileSystem*. Todo acesso aos módulos *Disk*e *Ram* são efetuados através do módulo *FileSystem*, que abstrai o acesso a memória em arquivos. Esse módulo fornece métodos para acessar arquivos como abrir, ler e escrever. Isso facilita a alteração da aplicação durante sua execução.

O módulo *Manager* executa as operações de configuração da máquina virtual. Através de mensagens de comando provenientes da ferramenta de atualização, esse módulo pode adicionar e remover blocos de função e conexões.

O módulo *Resource* é responsável por executar os blocos de funções. Ele fornece memória para a execução desses blocos e processa as conexões entre eles. Uma máquina virtual pode ter *n Resources*. Detalhes de funcionamento do *Resource* são mostrados na Seção 3.1.5.

O módulo *Network Interface* abstrai o acesso a rede de comunicação. Ele permite que a máquina virtual se comunique com outras máquinas virtuais e também com a ferramenta de atualização. As mensagens recebidas são armazenadas em uma fila e serão processadas pelo *Dispatcher* posteriormente.

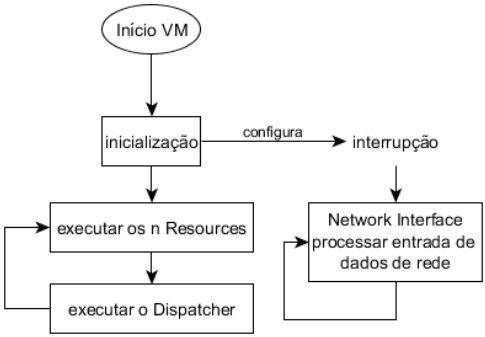
O *Dispatcher* tem a responsabilidade de entregar as mensagens que chegam, através da interface de rede, para o *Manager* ou para os *Resources*, dependendo do endereço de destino contido nessas mensagens. Se o destino da mensagem é o *Manager*, então ela é repassada ao *Manager* que as processa imediatamente. Caso o destino seja um bloco de funções, a mensagem é armazenada numa fila até que possa ser lida pelos blocos de destino.

Os módulos da máquina virtual executam como processos concorrentes cooperativos, ou seja, cada módulo executa parte de seu processamento e retorna o controle ao processo pai da máquina virtual. A Figura 24 mostra a sequência de execução da máquina virtual. Ela possui um laço principal que executa os *Resources* e o *Dispatcher*.

Normalmente os microcontroladores possuem um módulo de comunicação serial com um *buffer* pequeno, com apenas 2 bytes como o Atmega2560. Na maioria desses microcontroladores, é possível utilizar uma interrupção sempre que um byte é recebido. Para garantir que os dados da comunicação não serão perdidos, a função do módulo *Network Interface*, que lê esses dados, é executada quando ocorre essa interrupção. Quando os dados são completamente recebidos e formam uma mensagem válida, ela será colocada numa fila de mensagens recebidas. O módulo *Dispatcher* irá posteriormente ler essa fila de mensagens e enviá-las ao *Manager* ou aos *Resources*.

Quando a máquina virtual é executada sobre sistemas operacionais com suporte a *threads*, o módulo *NetworkInterface* pode ser executado por uma *thread* ao invés de uma interrupção.

Figura 24 – Diagrama da execução da máquina virtual ICARU-FB.



O controle de acesso dos processos a fila de mensagens é a única situação onde é necessário implementar exclusão mútua, o que é feito através do uso de semáforos. Tanto o módulo *Dispatcher* quanto o *NetworkInterface* podem acessar essa estrutura simultaneamente.

3.1.2 Reconfiguração Dinâmica

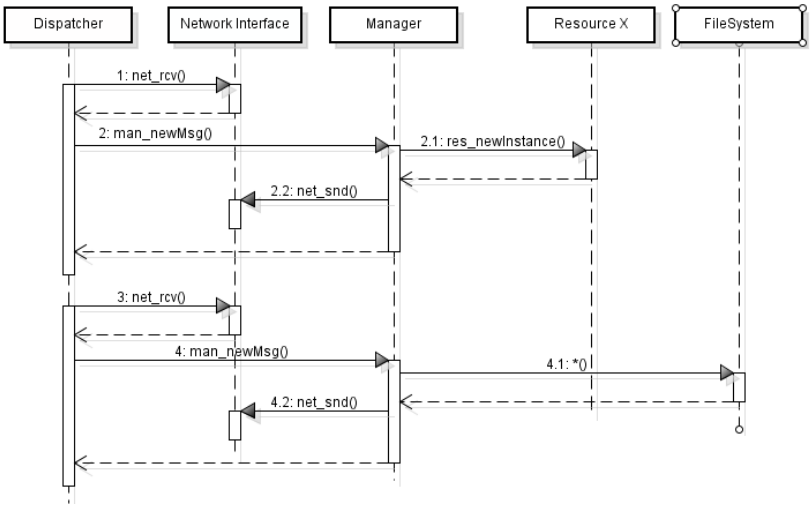
A reconfiguração dinâmica é a capacidade do dispositivo de alterar o programa aplicativo em tempo de execução. Isso permite que uma linha de produção, por exemplo, seja modificada sem que a mesma seja completamente interrompida durante a atualização de software.

A execução cooperativa dos módulos da máquina virtual permite que ela execute corretamente sem a necessidade de outros controles de acesso. As alterações do programa aplicativo só ocorrem quando o módulo *Dispatcher* é executado, durante essa alteração todos os *Resources* são interrompidos. Porém, o módulo *Dispatcher* executa somente parte da reconfiguração a cada ciclo de execução, pois as aplicações no dispositivo não podem permanecer interrompidas durante todo o processo de reconfiguração.

A reconfiguração da máquina inicia quando o *Dispatcher* verifica que há uma mensagem de reconfiguração na fila de mensagens de entrada, a qual é implementada no módulo *Network Interface*. O *Dispatcher* chama o

método *net_rcv* para ler uma mensagem da fila, como mostrado na chamada 1 e 3 da Figura 25.

Figura 25 – Diagrama de sequência para reconfiguração da máquina virtual.



Dependendo do que se deseja alterar na aplicação, o *Manager* pode atuar sobre o *Resource* ou diretamente no sistema de arquivos (chamadas 2.1 e 4.1). Quanto se-trata da alteração de instâncias e conexões, o comando é enviado ao *Resource* que fará a modificação (chamada 2.1 na Figura 25). Quando a alteração refere-se ao tipo de bloco de funções, o *Manager* pode atuar diretamente sobre o arquivo desse bloco, escrevendo nele as instruções que serão recebidas na mensagem. Apenas uma mensagem é processada a cada execução do *Dispatcher*, permitindo que os demais módulos executem durante uma reconfiguração.

A máquina virtual faz algumas operações extras antes de executar um determinado comando de reconfiguração, pois é necessário manter a aplicação consistente. Por exemplo, antes de executar um comando para excluir um tipo de bloco de funções, a máquina virtual irá primeiro excluir todas as instâncias desse bloco bem como suas conexões.

O módulo *Manager* é responsável por executar a reconfiguração dinâmica da máquina virtual. Todo o gerenciamento do programa aplicativo é feito através desse módulo. Ele pode criar e excluir tipos de blocos, instâncias de blocos e conexões. Também pode alterar os estados de cada instância e informar o estado atual da máquina virtual.

Os comandos aceitos pelo *Manager* são enviados num formato próprio, diferente do formato *xml* como definido na norma IEC 61499. Pois a plataforma alvo da máquina virtual pode não ter recursos necessários para processar o formato *xml*. O processamento do *xml* fica a cargo da ferramenta de atualização, a qual transforma o *xml* num formato mais compacto e o envia para o *Manager* na máquina virtual. A descrição da interface e dos comandos reconhecidos pelo *Manager* são mostrados no Apêndice 3.

3.1.3 O Bloco de Funções Executado pela Máquina Virtual ICARU-FB

Cada bloco de funções é definido por um conjunto de instruções que foram previamente compiladas. O bloco compilado é enviado para a máquina virtual ICARU-FB que o salva em um arquivo. O conjunto de instruções completo da máquina virtual é mostrado no Apêndice 2.

A execução do bloco de funções é feita por uma máquina de pilha que é implementada no módulo *Resource*, cada bloco de funções é executado isoladamente e possui também uma área de memória reservada para os seus dados que são as variáveis do bloco de funções, essa área foi alocada previamente pela máquina virtual. O bloco de funções indexa as suas variáveis a partir da posição 0, pois a máquina virtual abstrai a real localização dessa área de memória.

Quando a instância for executada pela primeira vez, a execução começa à partir da segunda instrução no código do bloco de funções. Portanto, o compilador deve colocar nessa posição o algoritmo que corresponde a inicialização do bloco de funções, como por exemplo, os valores iniciais do ECC e outras variáveis.

Ainda no código de inicialização, o compilador deve colocar a instrução ALLOC, que indica a máquina virtual quanto de memória RAM o bloco de funções precisará para armazenar os seus dados. Ao executar essa instrução, cujo parâmetro é quantidade em bytes que o bloco necessita, o *Resource* irá criar um arquivo com o tamanho necessário e guardar o ponteiro para o mesmo numa tabela de instâncias. Esse arquivo criado contém os dados do bloco de funções. A partir da segunda execução, o bloco será sempre executado a partir da primeira instrução, onde se encontra o código do ECC do bloco.

Os algoritmos e o ECC do bloco de funções podem acessar apenas as variáveis declaradas no próprio bloco. Então, o *Resource* executa o código do bloco sobre os dados desse bloco.

O bloco de funções básico mostrado na Figura 8, ao passar pelo compilador da infraestrutura, irá gerar um código com as instruções da máquina virtual. Para facilitar a compreensão do código gerado em instruções da máquina virtual, será mostrada uma versão em C desse código na Figura 26.

Figura 26 – Representação em linguagem C do código gerado para um bloco de funções básico.

```
void ALG_INIT(){
    REF = FALSE;
}
void ALG_READ(){
    if(temp  > 30,0){
        REF = TRUE;
    }
    if(temp < 20,0){
        REF = FALSE;
    }
}
void run(){
    if(__est == 0){
        if(__alg){
            __alg = 0;
        }
        if(INIT){
            __est = 1;
            __alg = 1;
            INIT = 0;
        }else
        if(READ){
            __est = 2;
            __alg = 1;
            READ = 0;
        }
    }
    if(__est == 1){
        if(__alg){
            __alg = 0;
            ALG_INIT();
            INITO = 1;
        }
        if(1){
            __est = 0;
            __alg = 1;
        }
    }
    if(__est == 2){
        if(__alg){
            __alg = 0;
            ALG_READ();
            Q = 1;
        }
        if(1){
            __est = 0;
            __alg = 1;
        }
    }
}
void initialize(){
    alloc(8);
    __est = 0;
    __alg = 1;
}
```

A função *run()* é gerada a partir do ECC, as funções ALG_READ() e ALG_INIT() representam os algoritmos ALG_READ e ALG_INIT. Para a inicialização das variáveis foi gerada a função *initialize()*, essa função contém a instrução ALLOC que informa a máquina virtual quanto de memória é necessária para armazenar as variáveis do bloco de funções. A função *run()* é uma versão em C do ECC do bloco, essa função corresponde ao ECC mostrado na Figura 7 (pag. 33). As variáveis *__est* e *__alg* controlam os estados do ECC. A variável *__est* armazena o estado atual do ECC. E *__alg* verifica se os algoritmos associados ao estado atual já foram executados.

Na Figura 27, o bloco de funções é mostrado na forma de bloco de instruções. Esse bloco é independente e pode ser executado pela máquina

virtual. As instruções 0 e 1 fazem um desvio para *run()* e *initialize()* respectivamente, ao instanciar um bloco de funções, a máquina virtual fará a primeira execução do bloco a partir da segunda instrução que corresponde a rotina de inicialização. As demais execuções do bloco são feitas a partir da primeira instrução, que corresponde a chamada do ECC do bloco de funções.

Figura 27 – Instruções geradas pelo compilador para um bloco de funções básico.

```
    ; goto run()
    0: GOTO          18
    ; goto initialize()
    1: GOTO          72
;ALG_INIT
    2: CONST          0
    3: STORE          REF
    4: RET            0
;ALG_READ
    5: LOAD           TEMP
    ...
    instruções
    omitidas
    ...
   17: RET            0
;run()
   18:
   ...
   instruções do ECC
   omitidas
   ...
   71: RET            0
;initialize()
   72: ALLOC          8
   73: CONST          0
   74: STORE          __alg
   75: CONST          1
   76: STORE          __est
   77: RET            0
```

O bloco de instruções acessa um bloco de memória alocado pela máquina virtual, no qual estão suas variáveis. Para o bloco de instrução, a memória começa na posição 0. Dessa forma um bloco de instrução não tem nenhuma relação direta com os demais, a única relação existente será por meio das conexões que são processadas pela máquina virtual separadamente.

3.1.4 O Módulo *FileSystem*

Enquanto os módulos *Ram* e *Disk* abstraem o acesso ao hardware de memória volátil e não volátil respectivamente, o módulo *FileSystem* abstrai o acesso a esses dois módulos em arquivos. Pois o sistema de arquivos, implementado pela máquina virtual, é bastante simples, o que permite que ele seja usado também como uma forma de gerenciar a memória RAM.

As memórias volátil e não volátil foram abstraídas em arquivos para facilitar o gerenciamento dos dados e do código dos blocos de funções em execução na máquina. Dessa forma é possível manipular os blocos de funções como arquivos, o que permite, por exemplo, adicionar e excluir blocos. O sistema de arquivos também é utilizado como uma forma

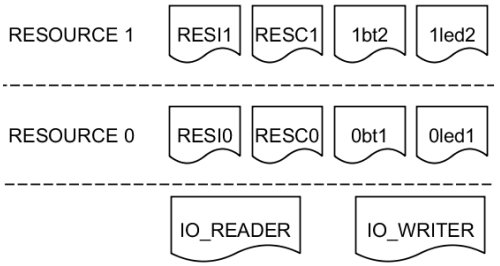
de proteger os dados dos blocos de funções, assim, a execução de um determinado bloco não pode acessar os dados de outros blocos.

Os dados armazenados em variáveis em um bloco de funções são manipulados através do módulo *FileSystem*, o acesso a esses dados é feito usando a mesma interface definida para a manipulação de arquivos. Para cada instância de um bloco, por exemplo, existe um arquivo na memória RAM no qual são armazenados seus dados. O conjunto de instruções de um bloco de função é armazenado em um arquivo separado e acessado através da mesma interface.

Durante a inicialização da máquina virtual, toda a memória não volátil é copiada para a RAM, dessa forma, todas as operações ocorrem sobre a memória RAM. As alterações que necessitam de armazenamento permanente, são executadas também sobre a memória não volátil (que é uma memória não volátil). Um exemplo disso é quando uma conexão entre dois blocos é criada, a alteração é armazenada também na na memória não volátil.

A Figura 28 mostra parte da memória RAM da máquina virtual. Nesse caso, estão armazenados dois tipos de bloco (IO_READER e IO_WRITER) e instâncias desses (0bt1, 1bt2, 0led1, 1led2). Detalhes sobre os arquivos RESCx e RESIx são apresentados na seção 3.1.5.

Figura 28 – Organização da memória em arquivos.



Os arquivos que representam um tipo de bloco são compartilhados entre todos os *Resources*. Esse arquivo contém o ECC e os algoritmos do bloco de funções que foram gerados pelo compilador.

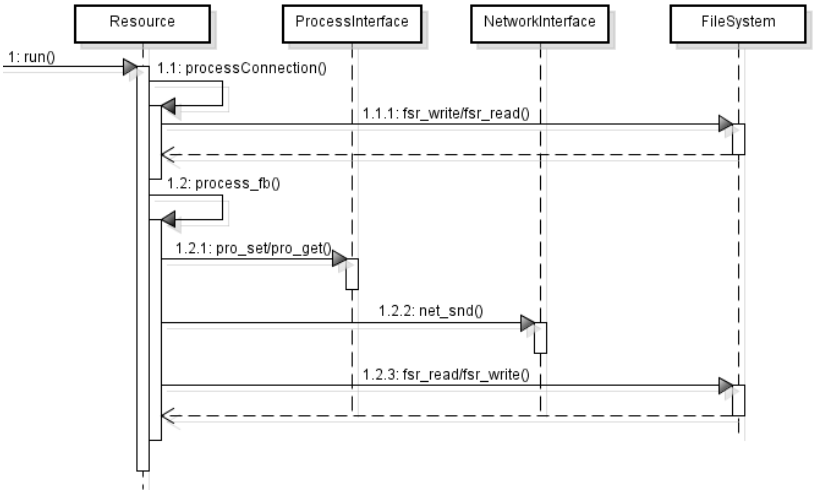
Os arquivos de dados das instâncias são perdidos sempre que a máquina reinicia, mas os arquivos RESIx, RESCx e de tipos de blocos permanecem sempre armazenados.

3.1.5 O Módulo *Resource*

O módulo *Resource* executa redes de blocos de função. A execução consiste em processar as conexões entre os blocos e, em seguida, processar os blocos. A Figura 29 mostra um diagrama de sequência de execução do método *run()* do *Resource*, que é chamado pelo *loop* principal da máquina virtual já visto na Figura 24.

Esse diagrama mostra a relação do módulo *Resource* com os demais módulos. Primeiro são processadas as conexões na chamada 1.1, como as conexões e os dados são armazenados em arquivos, faz-se necessária a comunicação com o sistema de arquivos da máquina virtual, visto na chamada 1.1.1.

Figura 29 – Diagrama de sequência da execução de um *Resource*.



Em seguida ocorre o processamento dos blocos de função na chamada 1.2. Novamente, como os dados dos blocos se encontram em arquivos, instruções que alteram os dados do bloco resultam em chamadas ao sistema de arquivos (chamada 1.2.3). Do mesmo modo, existem instruções que podem alterar dados no módulo *ProcessInterface* e também enviar dados pela interface de rede (chamadas 1.2.1 e 1.2.2).

O *Resource* possui duas tabelas para controlar a execução. A primeira, chamada RESI, contém todas as instâncias que o *Resource* deve executar. A segunda, chamada RESC, contém todas as conexões que devem

ser processadas. Essas tabelas são armazenadas em arquivos chamados RESIx e RESCx respectivamente, onde *x* é uma identificação do *Resource*. A tabela de instâncias é mostrada na Tabela 2.

Tabela 2 – Estrutura da tabela de instâncias.

Tipo: String	Nome: String	ram_data: IVMFILE	ram_code: IVMFILE
IO_READER	button1	xbutton1	IO_READER
IO_WRITER	led1	xled1	IO_WRITER

Para cada instância, a tabela armazena o tipo de bloco, o nome da instância, um ponteiro para o arquivo no qual se encontram os dados dessa instância e um ponteiro para o arquivo que contém as instruções do bloco.

A execução de uma instância consiste em ler o arquivo de código e processar as instruções sobre o arquivo de dados, o qual contém as variáveis da instância. As instruções executadas pelo módulo *Resource* são mostradas no Apêndice 2.

Dessa forma, a execução de um bloco de funções fica restrita a uma área de memória que é acessada como um arquivo.

Porém, esses arquivos de dados precisam se comunicar entre si, então utiliza-se as conexões definidas na norma IEC 61499. Uma conexão conecta uma variável de um bloco a variável de outro. No arquivo de dados, as variáveis estão dispostas sequencialmente, nesse caso uma conexão conecta uma posição *x* de um arquivo de dados a posição *y* do arquivo de dados de destino. A Figura 30 mostra a estrutura da tabela de conexões.

Figura 30 – Estrutura da tabela de conexões.

iori: string	vori: int8	ides: string	vdes: int8	→
button1	5	led1	2	

→

vsize: int8	fori: IVMFILE	fdes: IVMFI	isEvent
1	xbutton1	xled1	1

Onde:

iori:	Nome da instância de origem.
vori:	Posição da variável de origem no arquivo *ram_data*.
ides:	Nome da instância de destino.
vdes:	Posição da variável de destino no arquivo *ram_data*.
vsize:	Tamanho da variável a ser transferida.
fori:	Ponteiro para o arquivo de dados de origem.
fdes:	Ponteiro para o arquivo de dados de destino.
isEvent:	Indica se é uma variável de evento.

Há dois tipos de conexões entre os blocos de função, as conexões de evento e as de dados. Para diferenciar o tipo de conexão, a tabela de conexões possui o indicador *isEvent*.

Uma variável de evento só é transferida quando a origem tem valor 1, indicando que há um evento. Quando ocorre uma transferência de evento, a origem, que era 1, passará a ser 0 e o destino será 1.

O próprio bloco de funções é responsável por atribuir 0 a uma variável de evento de entrada. Isso irá ocorrer durante o processamento do ECC do bloco. Para os eventos de saída, o valor 0 é atribuído automaticamente pelo *Resource*.

Quando se trata de uma variável de dados, o valor é simplesmente copiado da origem para o destino.

3.1.6 O Módulo *ProcessInterface*

O módulo *ProcessInterface* fornece acesso as entradas e saídas do hardware. Essas entradas podem conter sensores, atuadores e outros elementos desse tipo. Esse módulo disponibiliza o acesso ao hardware em um vetor de bytes, que pode ser acessado através de métodos de escrita e leitura (*get* e *set*). Cada posição nesse vetor corresponde a algum elemento do hardware. A Figura 31 apresenta a interface desse módulo.

Figura 31 – Interface do módulo *Process Interface*

```
int1 pro_set(int8 address, int8 val);
int1 pro_get(int8 address, int8 *val);
int1 pro_init();
```

O método *pro_set(...)* permite que a máquina virtual atribua um valor a uma determinada saída. Ela pode atribuir 1, por exemplo, a uma saída digital o que pode ocasionar o acionamento de um atuador. Para ler entradas do hardware, utiliza-se o método *pro_get()*, através do qual pode-se obter, por exemplo, os dados de um sensor.

A implementação desse módulo depende totalmente do hardware. Por exemplo, em alguns compiladores de linguagem C para microcontroladores, o acesso as saídas digitais é dados por funções como *output_high() e output_low()*. Assim, o módulo *ProcessInterface* poderia ser implementado como mostra a Figura 32.

Figura 32 – Implementação da função *pro_set* do módulo *Process Interface*

```
int1 pro_set(int8 address, int8 val){
    if(address == 10){
        if(val) output_high(pin_d0);
        else output_low(pin_d0);
    }
    return 1;
}
```

O método verifica se o endereço é 10, e então verifica o valor de *val* e aciona hardware utilizando métodos fornecidos pela biblioteca da plataforma.

Segundo o conjunto de instruções da máquina virtual, pode-se criar uma aplicação que acessa os métodos do módulo *ProcessInterface*. A Figura 33 mostra isso.

Figura 33 – Instrução capaz de chamar o método *pro_set*

```
...
CONST 1
CONST 10
ARSTORE

...
```

A instrução CONST coloca uma constante na pilha de dados. A instrução ARSTORE faz acesso ao módulo *ProcessInterface*. Ela remove da pilha de dados o endereço que será acessado e, em seguida, o valor que será colocado nesse endereço. No exemplo da Figura 33, está se colocando 1 no endereço 10.

A portabilidade dos blocos de função entre diferentes versões da máquina virtual, pode ser comprometida devido a implementação do módulo *ProcessInterface*.

Por exemplo, se a máquina virtual possui uma aplicação que atribui o valor *True* para uma saída booleana cujo endereço é 10, então será chamada a função *pro_set(10,True)*. Para que essa aplicação seja compatível com outras versões da máquina virtual, essas máquinas também devem implementar uma saída booleana no endereço 10. O módulo *ProcessInterface* é quem decide o que é cada endereço.

3.1.7 Aderência a Norma IEC 61499 Através da Máquina Virtual

Nesse trabalho optou-se por utilizar uma máquina virtual que fosse capaz de executar os blocos de funções da norma IEC 61499 nativamente, como não foi encontrada nenhuma máquina virtual para esse fim na literatura presente, para esse trabalho foi projetada e implementada uma máquina virtual. Existem versões da máquina virtual Java que executam em plataformas semelhantes a máquina virtual desse trabalho, como em Brouwers, Corke e Langendoen (2008) e Bob (2014).

Brouwers, Corke e Langendoen (2008) apresenta a máquina virtual Java Darjeeling que é capaz de executar em microcontroladores de 8-bit, porém, não foi implementada a especificação completa da máquina virtual Java. Por exemplo, devido as limitações da plataforma alvo, essa máquina não utiliza o sistema de *class loader* presente no Java, ao invés disso, utiliza-se um modelo de ligação estático, também ainda não há suporte para ponto flutuante. Em Bob (2014), o *bytecode* do Java é transformado em estruturas na linguagem C que pode ser compilado diretamente para uma plataforma alvo como o Arduino.

Tanto Brouwers, Corke e Langendoen (2008) quanto Bob (2014), não podem suportar a execução da norma IEC 61499, principalmente com relação a reconfiguração dinâmica. A implementação dos blocos de funções em classes Java cria uma dependência entre os blocos que não pode ser alterada em tempo de execução. Ao compilar o código inteiro para a plataforma alvo, como em Bob (2014), torna-se inviável alterá-lo em tempo de execução.

Assim, parece que a solução mais viável para executar os blocos de funções em hardwares limitados, atendendo a todos os seus requisitos, é utilizar uma máquina virtual que suporte os blocos de funções nativamente.

3.2 A FERRAMENTA DE ATUALIZAÇÃO DINÂMICA

Como citado anteriormente, a máquina virtual desse trabalho executa sobre plataformas com poucos recursos computacionais. Devido a escassez de recurso, é inviável a execução de um interpretador *xml* ou compilador para os algoritmos internos do bloco de funções, a norma IEC 61499 não faz nenhuma restrição com relação a linguagem utilizada para implementar esses algoritmos, o que exige que a plataforma tenha um compilador para cada linguagem utilizada.

Assim a máquina virtual não é capaz de receber diretamente uma definição de sistema em *xml*. Logo, a definição em *xml* é interpretada por uma ferramenta de software que executa em um computador PC. Essa ferramenta de atualização dinâmica, com o auxílio de um ou mais compiladores, gera os arquivos binários dos blocos de funções.

As definições de blocos de função são enviadas, pela ferramenta, a um compilador que irá gerar um executável independente para cada bloco, como foi mencionado na seção 3.1.3. Nesse trabalho, foi implementado um compilador, que tem como linguagem fonte a linguagem ST e linguagem destino o conjunto de instruções da máquina virtual. A implementação de outros compiladores, que gerem código nesse conjunto de instruções, pode permitir que a máquina virtual aceite outras linguagens além da linguagem ST.

O bloco compilado é enviado às máquinas virtuais que executarão esse bloco, a ferramenta de atualização dinâmica é responsável por esse processo. Dependendo da configuração que está no *xml*, nem todas as máquinas virtuais receberão todos os blocos de função, cada máquina recebe apenas o necessário para executar a sua parte do sistema.

As demais definições no arquivo *xml* como instâncias e conexões, são processadas pela ferramenta de atualização que as envia para as respectivas máquinas virtuais. Essas definições irão formar as tabelas dos *Resources* como mostrado na seção 3.1.5.

3.2.1 O Compilador

Do ponto de vista da máquina virtual, o bloco de funções é apenas um bloco com instruções a serem executadas. A máquina virtual não conhece o que é um ECC com estados ou transições, para processar o bloco, a máquina virtual executa somente operações de busca, decodificação e execução de instruções.

Por isso, o compilador precisa transformar todo o bloco de funções em um único algoritmo ou em um conjunto de algoritmos. Essa transformação é apresentada na Seção 3.1.3 (pag. 61).

O formato *xml* da IEC 61499 permite especificar a linguagem utilizada nos algoritmos dos blocos de função através de marcações específicas como *<ST Text="A:=30;"/>*, que é utilizada para a linguagem ST.

O pré-processador extrai os blocos de funções da definição em *xml* e os transforma em código fonte na linguagem ST de acordo com a norma

IEC 61131-3 (IEC, 2003). Esse código fonte gerado é enviado ao compilador que gera os binários para a máquina virtual. A gramática da linguagem ST utilizada pelo compilador desse trabalho é mostrada no Apêndice 1. A Figura 34 mostra o código ST a ser compilado. Esse bloco de funções, o TMP_CONTROL, foi utilizado como exemplo anteriormente na seção 2.2.1, na qual foi explicado o seu funcionamento.

Figura 34 – Código ST para um bloco de funções básico.

```
FUNCTION_BLOCK TMP_CONTROL
  EVENT_INPUT
   INIT;
   READ WITH TEMP;
  END_EVENT
  EVENT_OUTPUT
   INITO;
   Q WITH REF;
  END_EVENT
  EC_STATES
    START; (* Initial state*)
    ST_INIT:ALG_INIT->INITO;
    ST_READ: ALG_READ -> Q;
  END_STATES
  EC_TRANSITIONS
    START TO ST_INIT := INIT;
    ST_INIT TO START := 1;
    START TO ST_READ := READ;
    ST_READ TO START := 1;
  END_TRANSITIONS
  VAR_INPUT
    TEMP: REAL;
  END_VAR
  VAR_OUTPUT
    REF: BOOL;
  END_VAR
  ALGORITHM ALG_INIT IN ST
    REF := FALSE;
  END_ALGORITHM
  ALGORITHM ALG_READ IN ST
    IF TEMP > 30.0 THEN
      REF := TRUE;
    ELSIF TEMP < 20.0 THEN
      REF := FALSE;
    END_IF
  END_ALGORITHM
END_FUNCTION_BLOCK
```

Como mencionado, o ECC deve ser transformado num algoritmo e compilado normalmente. Uma possível transformação do ECC em algoritmo é mostrada na Figura 35, esse ECC já foi mostrado em linguagem visual na Figura 7 (pag. 33).

Figura 35 – Transformação do ECC em algoritmo.

```
void run(){
    if(__est == 0){
        if(__alg){
            __alg = 0;
        }
        if(INIT){
            __est = 1;
            __alg = 1;
            INIT = 0;
        }else
        if(READ){
            __est = 2;
            __alg = 1;
            READ = 0;
        }
    }
    if(__est == 1){
        if(__alg){
            __alg = 0;
            ALG_INIT();
            INITO = 1;
        }
        if(1){
            __est = 0;
            __alg = 1;
        }
    }
    if(__est == 2){
        if(__alg){
            __alg = 0;
            ALG_READ();
            Q = 1;
        }
        if(1){
            __est = 0;
            __alg = 1;
        }
    }
}
```

Nessa figura, o ECC foi transformado numa função chamada *run()*. Cada estado do ECC recebeu uma numeração sendo 0 para o estado START, 1 para ST_INIT e 2 para ST_READ.

O algoritmo armazena o estado atual do ECC nas variáveis *__est* e *__alg*, a primeira armazena o estado atual do ECC e a segunda indica se o algoritmo do estado atual já foi executado.

Em cada estado é verificado se é necessário executar algum algoritmo associado ao estado atual e se há alguma condição de transição verdadeira, se houver, é atribuído 0 ao evento que gerou a transição e as variáveis *__est* e *__alg* são atualizadas.

Depois de transformar o bloco todo em algoritmos e compilá-lo, a ferramenta de atualização pode enviá-lo para as máquinas virtuais que irão executar esse bloco.

Criam-se as conexões entre os blocos, na máquina virtual, informando os blocos e variáveis de origem e destino. Porém, a máquina virtual conhece apenas os blocos por seus respectivos nomes, as variáveis não são identificadas.

Como mencionado anteriormente, a máquina virtual armazena os dados em arquivos. Portanto, para criar uma conexão, a ferramenta de atualização precisa informar a máquina virtual qual a posição das variáveis de origem e destino nesses arquivos de dados, bem como o tamanho da variável. Assim, o compilador também precisa gerar essa informação, que diz o

tamanho e a posição que foi escolhida para cada variável. A Figura 36 mostra essas informações geradas pelo compilador para o bloco TMP_CONTROL já visto anteriormente.

Figura 36 – Mapa das variáveis de um bloco de funções.

```
__alg:0:1,
__est:1:1,
INIT:2:1,
READ:3:1,
INITO:4:1,
Q:5:1,
TEMP:6:1,
REF:7:1,
```

Logo, se a origem de uma conexão é a variável REF de um bloco A do tipo TMP_CONTROL, a ferramenta de atualização deve enviar o valor 7 que corresponde a posição da variável e 1 que corresponde ao tamanho da variável.

Ao criar uma conexão, é necessário também que a ferramenta de atualização indique quais dessas variáveis são eventos, assim a máquina virtual pode tratar de forma diferente as conexões de eventos. Como a alocação das variáveis é decidida pelo compilador, é necessário que este gere também informações sobre quais variáveis são eventos.

O compilador desse trabalho gera um arquivo contendo o nome das variáveis de eventos de cada bloco, ele contém a posição dentro do arquivo de dados em que cada variável de evento foi alocada. Esse arquivo pode ser consultado pela ferramenta de atualização para criar as conexões.

A norma IEC 61499 permite utilizar qualquer linguagem de programação para implementar os algoritmos dentro dos blocos de funções. Para que não seja necessário criar vários compiladores distintos para a máquina virtual, a fim de permitir a utilização de cada linguagem de programação, pode-se implementar compiladores que gerem como linguagem alvo a linguagem ST. Assim, todos os esforços de desenvolvimento, como otimizações de código, podem ser concentrados em apenas um compilador.

O código fonte do compilador desenvolvido nesse trabalho está disponível em Pinto (2014).

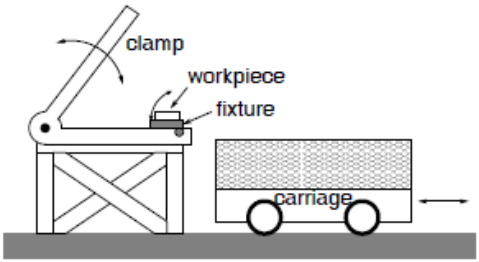
3.3 AMBIGUIDADES NA DEFINIÇÃO DA NORMA IEC 61499

Vários autores abordam os problemas relacionados a implementação da norma IEC 61499 (FERRARINI; VEBER, 2004; VYATKIN, 2009; SUNDER et al., 2006; STRASSER et al., 2011; CENGIC; AKESSON, 2010a). Segundo Yoong et al. (2009), a norma IEC 61499 não fornece uma semântica formal para a execução dos blocos de funções, ao invés disso, ela fornece uma descrição verbal para a execução desses blocos, o que resultou em múltiplas interpretações. Consequentemente, os blocos de funções podem apresentar comportamento distinto quando executados em ambientes de execução diferentes.

Em Cengic, Ljungkrantz e Akesson (2006), são citados dois desses problemas. O primeiro deles relacionado a ordem de execução dos blocos, e o segundo, relacionado a eventos contíguos.

A ordem de execução dos blocos de função não está definida pela norma IEC 61499. Como os blocos são executados de forma concorrente, supõe-se que qualquer ordem de execução pode ser escolhida. Para ilustrar esse problema, tem-se um exemplo na Figura 37.

Figura 37 – Exemplo de dispositivo para a realização de um processo automático.



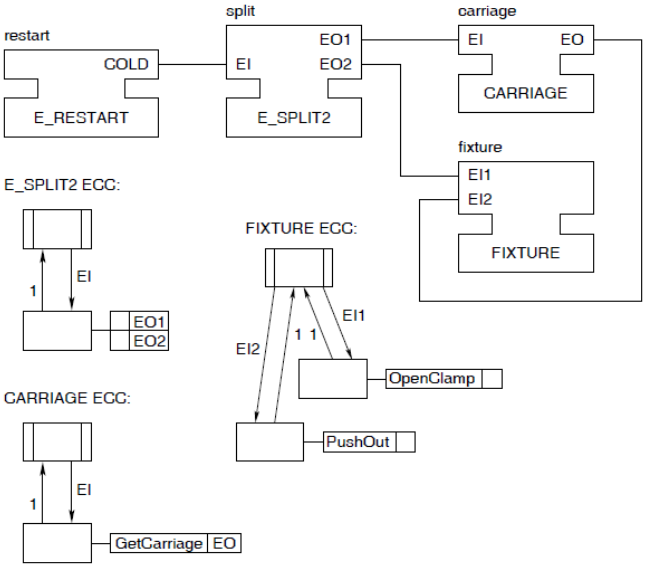
Fonte: (CENGIC; LJUNGKRANTZ; AKESSON, 2006)

O exemplo contém um dispositivo para a realização de um processo automático. A peça de trabalho é processada no suporte (*fixture*) que segura a peça enquanto um grampo (*clamp*) faz o trabalho. Ao terminar, o grampo é aberto e a peça é empurrada para um transporte (*carriage*)

Para controlar esse dispositivo, definiu-se a configuração mostrada na Figura 38. O bloco *carriage* chama o transporte e o bloco *fixture* abre o

grampo se receber um evento em EI1 e empurra a peça ao receber um evento em EI2.

Figura 38 – Blocos de funções para controle do dispositivo da Figura 37.



Fonte: (CENGIC; LJUNGKRANTZ; AKESSON, 2006)

Quando *split* recebe EI, *carriage* e *fixture* são ativados ao mesmo tempo. Mas, se *carriage* executar primeiro, ele irá ativar o evento EI2 do bloco *fixture*. Então, *fixture* terá EI1 e EI2 para processar, se optar por processar primeiro o evento EI2, então a peça será empurrada enquanto o grampo ainda está fechado, o que poderá destruir o dispositivo.

No modelo de execução da máquina ICARU-FB, quando *carriage* executa, o evento gerado fica armazenado em EO e somente será transmitido para *fixture* ao término da execução atual da rede. Portanto, quando *fixture* executar, haverá somente EI1 para processar. Nesse caso, essa aplicação executaria como o esperado.

Supondo um caso onde *fixture* recebe os dois eventos no mesmo ciclo de execução, a aplicação ainda funcionaria como o esperado na máquina ICARU-FB, pois, na implementação da máquina virtual, optou-se por processar os eventos de cima para baixo. Mas se a ordem dos eventos fosse inversa, o problema persistiria também na solução proposta para a máquina

ICARU-FB. Porém, esse tipo de problema pode ser ilustrado também em uma linguagem estruturada como C, isso é mostrado na Figura 39.

Figura 39 – Problema da execução de eventos simultâneos.

```
if(EI1){
 /* Abrir o grampo */
}
if(EI2){
 /* Empurrar a peça */
}
```

```
if(EI2){
 /* Empurrar a peça */
}
if(EI1){
 /* Abrir o grampo */
}
```

Nesse caso, a ordem de execução é definida pela sequência que o programador utilizou. Nesse trabalho, os eventos são verificados de cima para baixo, o que faria com que EI1 fosse processado primeiro. Porém essa definição não existe na norma IEC 61499.

O bloco de funções *fixture* possui dois eventos cuja ativação incorreta (primeiro EI2 e depois EI1) pode ocasionar a destruição da máquina. A implementação do próprio bloco poderia resolver esse problema. Quando a execução de um algoritmo é pré-requisito para execução de outro, essa checagem pode ser feita pelo próprio algoritmo, como mostrado na Figura 40.

Figura 40 – Algoritmo *PushOut* checando se o grampo está aberto (*ClampOpened*).

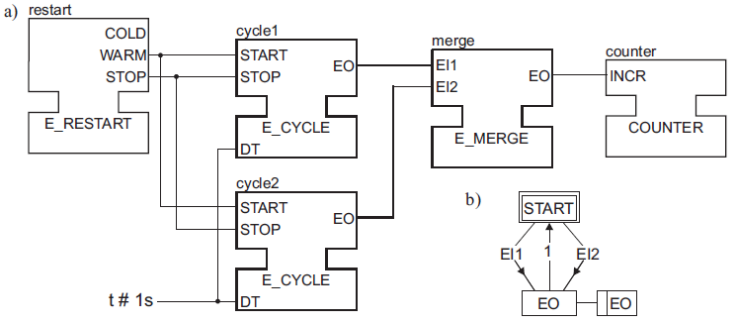
```
ALGORITHM PushOut in ST
  [...]
  IF NOT ClampOpened THEN
    OpenClamp();
  END_IF;
  [...]
END_ALGORITHM
```

Nesse exemplo, antes de empurrar a peça, o algoritmo *PushOut* verifica se o grampo já está aberto e abre-o se necessário. Foi usada a variável interna *ClampOpened* para indicar que o grampo está aberto, ela é alterada pelo algoritmo *OpenClamp* quando este efetua a abertura do grampo.

O segundo problema é com relação a eventos contíguos. Que são eventos que ocorrem simultaneamente (ou quase) em entradas diferentes ou na mesma entrada. Segundo Cengic, Ljungkrantz e Akesson (2006), os desenvolvedores do ambiente de execução ISaGRAF, interpretaram que

eventos que chegam, enquanto o ECC está ocupado, devem ser descartados. Enquanto que no ambiente de execução Fuber, foi interpretado que os eventos que chegam, enquanto o ECC está ocupado, devem ser armazenados numa em fila e processados posteriormente. Isso pode gerar comportamentos diferentes nas aplicações. Um exemplo é mostrado na Figura 41.

Figura 41 – a) Exemplo de aplicação produzindo eventos simultâneos. b) ECC para o bloco de funções E_MERGE.



Fonte: (CENGIC; LJUNGKRANTZ; AKESSON, 2006)

Nessa aplicação de exemplo, os blocos E_CYCLE produzem eventos que são enviados para o bloco E_MERGE ao mesmo tempo. Os eventos gerados por E_MERGE vão para o bloco COUNTER.

Executando a aplicação durante x segundos no ambiente ISaGRAF, *counter* terá contado x eventos. Enquanto que no ambiente Fuber terá contado $2x$. Pois a implementação do bloco E_MERGE em ISaGRAF descarta o segundo evento.

Esse problema pode ser resolvido alterando-se o bloco E_MERGE. Nesse caso o problema passa para o bloco COUNTER que terá de processar dois eventos em um intervalo de tempo muito curto.

Na implementação da máquina ICARU-FB, os eventos em entradas diferentes não são descartados, eles permanecem armazenados na própria de variável de entrada. Ao executar o ECC, o mesmo processa todos os eventos de entradas de cima para baixo.

O caso dos eventos na mesma entrada não ocorre na máquina ICARU-FB, pois não é possível que um ECC gere dois eventos na mesma saída no mesmo ciclo de execução. No caso de E_MERGE receber EI1 e EI2 ao mesmo tempo, o ECC irá processar EI1 e gerar um evento em EO num ciclo, somente no próximo ciclo ele irá processar EI2 e gerar novamente EO.

Outra ambiguidade da norma IEC 61499, que foi verificada nesse trabalho, é com relação ao posicionamento dos blocos na linguagem visual da norma. O posicionamento de cada elemento visual durante a edição dos blocos é armazenado nos atributos *x* e *y* no arquivo *xml*, porém não há uma padronização com relação a unidade de medida a ser utilizada nesses atributos, o que acarreta em posicionamento diferente dos blocos nos diferentes editores aderentes a norma.

4 ESTUDO DE CASO

Nesse capítulo são apresentados os resultados obtidos na implementação da infraestrutura descrita no Capítulo 3. Como estudo de caso, foram implementadas três versões da máquina virtual ICARU-FB e dois sistemas utilizando o editor FBE, já citado na Seção 2.3. As três versões da máquina virtual ICARU-FB são:

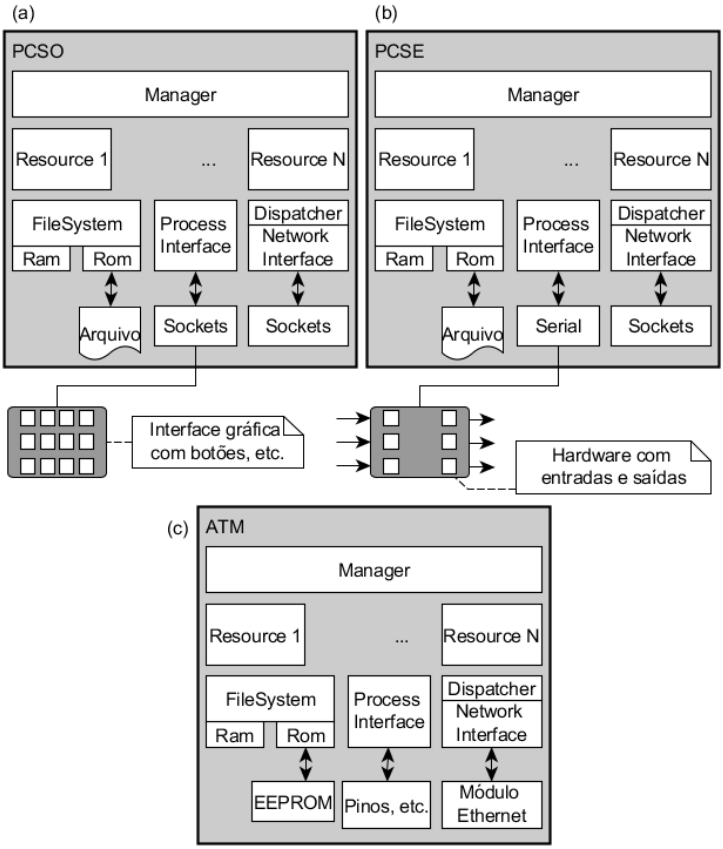
ATM: Essa versão, mostrada na Figura 42c, executa sobre a plataforma Arduino ATmega2560 (ARDUINO, 2014), a qual possui um microcontrolador ATmega2560. O módulo *ProcessInterface* dessa versão permite acesso aos pinos de E/S do Arduino. Para implementar o módulo *NetworkInterface*, utilizou-se uma biblioteca que se comunicar com um *Shield Ethernet* do Arduino.

PCSO: Essa versão executa sobre o sistema operacional Windows e é mostrada na Figura 42a. O módulo *ProcessInterface* dessa versão, através de *Sockets*, conecta-se a uma interface gráfica que é mostrada na Figura 43.

PCSE: Essa versão, mostrada na Figura 42b, também executa sobre o sistema operacional Windows. Porém, o módulo *ProcessInterface* conecta-se a uma interface de E/S através da porta serial do computador. Essa interface de E/S permite que sejam controlados elementos como lâmpadas e botões.

Apenas os módulos que dependem da plataforma apresentam implementações distintas. O módulo *Disk*, por exemplo, que implementa a memória não volátil da máquina virtual, utiliza-se de um arquivo para armazenar os seus dados nas versões PCSO e PCSE. Enquanto que na versão ATM é utilizada uma memória EEPROM, que é um tipo de memória não volátil interna do microcontrolador ATmega2560.

Figura 42 – Visão geral da implementação das máquinas virtuais.



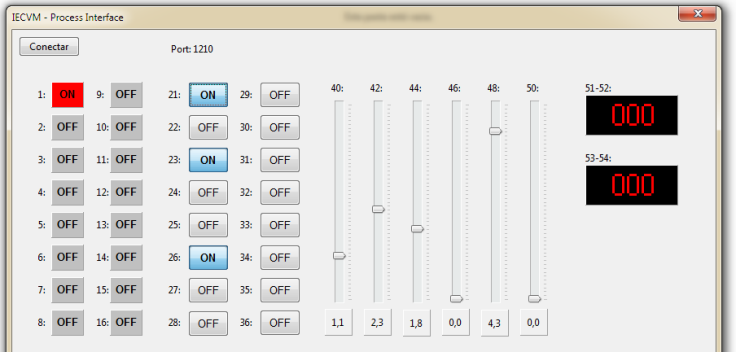
A comunicação entre as máquinas virtuais foi feita utilizando o protocolo TCP/IP. Um servidor foi desenvolvido para que as máquinas virtuais e a ferramenta de atualização se conectem e troquem informações.

As versões PCSO e PCSE diferem apenas no módulo *ProcessInterface*, pois PCSO acessa uma interface gráfica com botões e outros elementos através de sockets (Figura 43), enquanto PCSE acessa um hardware com entradas e saídas através da porta serial do computador.

A Figura 43 demonstra que a máquina virtual ICARU-FB pode executar sobre o sistema operacional Windows utilizando a interface gráfica desse sistema. Isso torna possível, por exemplo, o uso da máquina ICARU-FB para controlar também interfaces de usuário. Nesse caso, o módulo

ProcessInterface controla essa interface permitindo que a máquina virtual acesse os elementos gráficos.

Figura 43 – Interface gráfica para a máquina virtual PCSO.

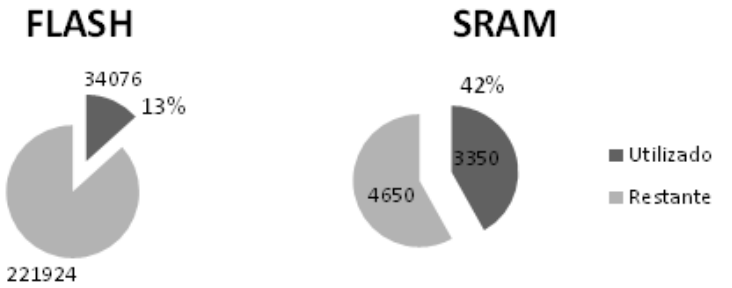


Na versão ATM, o módulo *ProcessInterface* acessa as entradas e saídas disponíveis no microcontrolador, que são os pinos digitais e entradas analógicas. Para implementar o módulo *NetworkInterface*, utilizou-se uma biblioteca de comunicação com um *Shield Ethernet* do Arduino.

As quantidades de memória ocupadas pelos módulos da máquina virtual ICARU-FB ATM são apresentadas na Figura 44, essas informações foram obtidas através da ferramenta *avr-size* que integra as ferramentas de desenvolvimento do Arduino. Essa tabela mostra que o código do núcleo da máquina virtual ocupa 13749 bytes da memória de programa (FLASH) do microcontrolador, e ocupa 1643 bytes da memória RAM (SRAM) do microcontrolador. O tamanho do núcleo é a soma de todos os módulos da máquina virtual exceto os módulos *Ram*, *Disk*, *ProcessInterface* e *NetworkInterface*.

Figura 44 – Memória ocupada pelos módulos da máquina virtual.

Nome do Módulo	FLASH (Bytes)	SRAM (Bytes)
Inicialização da máquina virtual e loop de execução	231	41
Dispatcher	151	11
Filesystem	4434	334
Manager	2653	467
NetworkInterface	1856	717
ProcessInterface	155	9
Ram	163	3507
Resource	6280	790
Rom	52	0
TOTAL dos módulos	**15975**	**5876**
TOTAL do núcleo	**13749**	**1643**
TOTAL do executável	**34076**	**6857**
TOTAL ATmega2560	**256000**	**8000**



O módulo *Ram* é responsável por abstrair o hardware de memória RAM, a memória usada por esse módulo varia em função da memória disponível, no caso do exemplo apresentado na Figura 44, o Módulo *Ram* ocupou 3507 bytes da SRAM que estavam disponíveis.

Portanto, o cálculo da memória consumida pela máquina virtual desconsidera a memória alocada a esse módulo, cuja implementação pode ocupar toda a memória SRAM restante. Isso totalizou 3350 bytes de SRAM ocupados pela máquina virtual, o que corresponde a aproximadamente 42% do total disponível no ATmega2560.

Ao considerar o tamanho final da máquina virtual, incluindo todos os módulos e bibliotecas necessárias que são fornecidas pela IDE do Arduino,

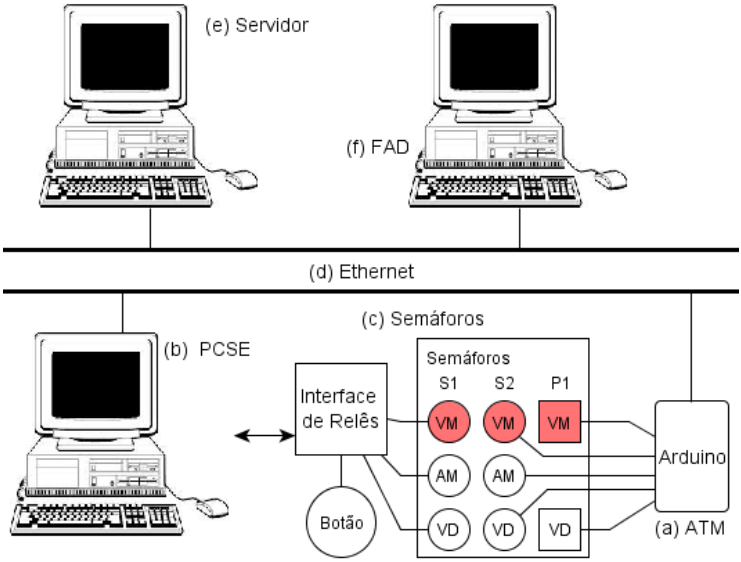
o tamanho do código foi de 34076 bytes com 6857 bytes para a memória de dados.

Como estudo de caso foram desenvolvidos dois exemplos, um sistema de semáforos e um controle PID. Os blocos de funções que implementam esses sistemas foram definidos usando o editor FBE (HARBS, 2012), o qual salva as especificações em formato *xml* de acordo com a norma IEC 61499. Para uma confirmação de que os exemplos estão em conformidade com a norma IEC 61499, eles também foram editados no editor FBDK descrito na Seção 2.3.

4.1 O SISTEMA DE SEMÁFOROS

O sistema de semáforos implementado consiste em dois semáforos para carros e um para pedestres. Nesse exemplo foram utilizadas duas máquinas virtuais, uma executando no Arduino e outra executando num computador. A configuração física do sistema é mostrada na Figura 45.

Figura 45 – Configuração física do sistema.



A configuração física usada nesse exemplo consiste em:

a) Um Arduino Atmega2560 o qual contém uma versão da máquina virtual;

b) Um computador com interface de saídas e entradas para a máquina virtual PCSE;

c) Os semáforos que serão controlados;

d) Rede para comunicação TCP/IP;

e) Um computador para executar o servidor de máquinas virtuais;

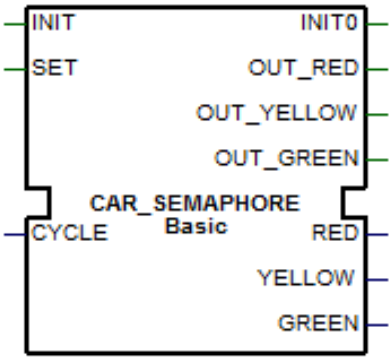
f) Um computador para executar a ferramenta de atualização dinâmica (FAD).

A comunicação entre as máquinas virtuais é feita através do protocolo TCP/IP. Quando acionadas, as máquinas virtuais se conectam ao servidor e iniciam a execução da aplicação. Para configurar as máquinas virtuais com suas devidas aplicações, a ferramenta de atualização conecta-se ao servidor e envia mensagens com os comandos de configuração, cada mensagem tem como destino uma máquina virtual, o servidor se encarrega de entregar cada mensagem ao seu destino. Quando a aplicação precisa enviar mensagens usando blocos PUBLISHER, por exemplo, a máquina virtual as envia para o servidor, o qual irá enviar essa mensagem a todas as máquinas virtuais e, por fim, cada máquina se encarrega de entregar a mensagem aos SUBSCRIBERs correspondentes.

Para controlar cada um dos dois semáforos para carros, foi criado um bloco de funções chamado CAR_SEMAPHORE. Esse bloco contém três entradas, sendo INIT para inicialização do mesmo, SET e CYCLE para controlar o funcionamento do semáforo.

Quando SET é acionado e CYCLE tem o valor 0, o bloco atribui *True* para a saída RED, *False* para as demais e dispara os eventos OUT_RED, OUT_YELLOW e OUT_GREEN. Se CYCLE tiver o valor 1 ou 2, o mesmo ocorre, porém atribui-se True para YELLOW ou GREEN respectivamente. A Figura 46 mostra esse bloco.

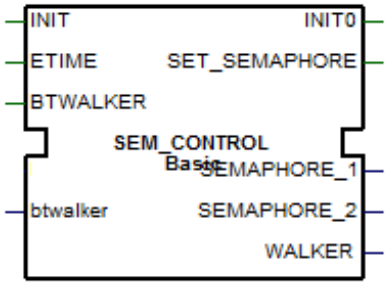
Figura 46 – Bloco de funções que controla um semáforo para carros.



Para executar o controle geral de todos os semáforos, foi criado um segundo bloco de funções chamado SEM_CONTROL, esse bloco de funções controla até dois semáforos para carros e um semáforo para pedestres.

Sua funcionalidade consiste em alternar os estados dos semáforos a cada ocorrência do evento ETIME. Se ETIME ocorre a cada 5 segundos, por exemplo, então os estados dos semáforos serão alterados a cada 5 segundos. A ocorrência do evento BTWALKER indica que o botão para pedestres foi pressionado, assim, quando terminar o ciclo dos semáforos para carros, eles serão fechados e será aberto o semáforo para pedestres. A Figura 47 mostra o bloco de funções SEM_CONTROL.

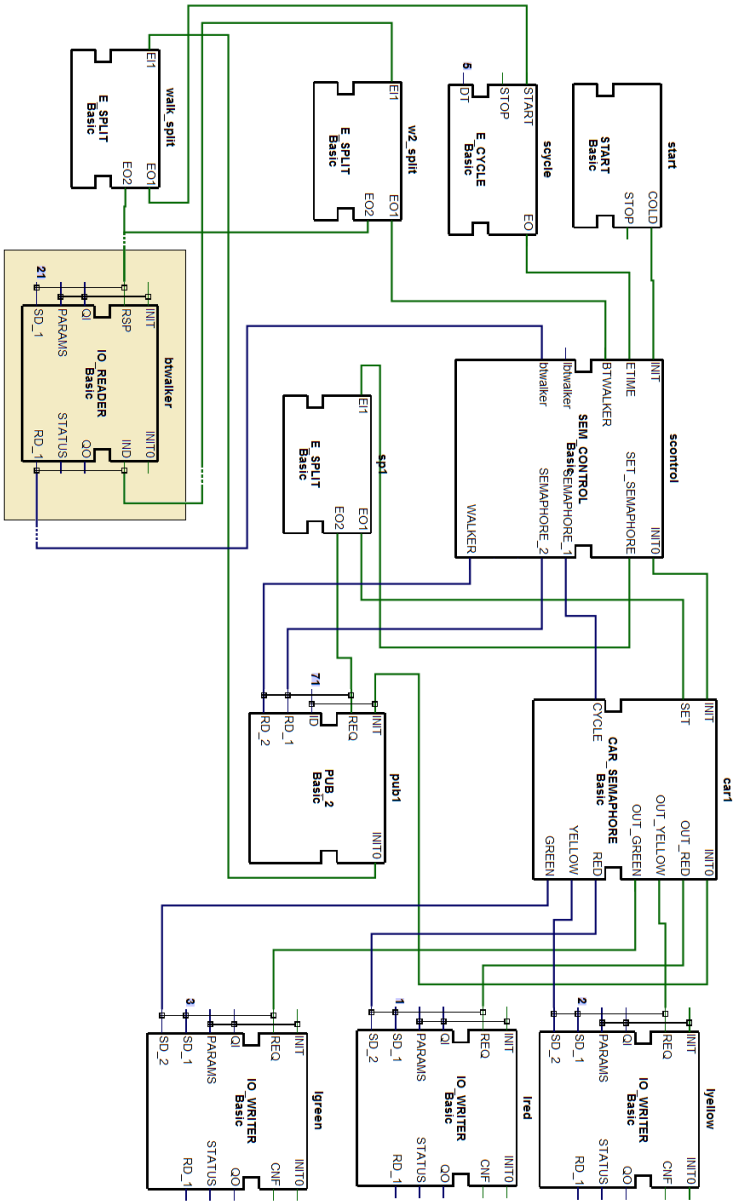
Figura 47 – Bloco de funções que controla até dois semáforos para carros e um para pedestres.



O sistema de semáforo é apresentado nas Figuras 48 e 49. A Figura 48 descreve os blocos que executam no PC, que consiste no bloco

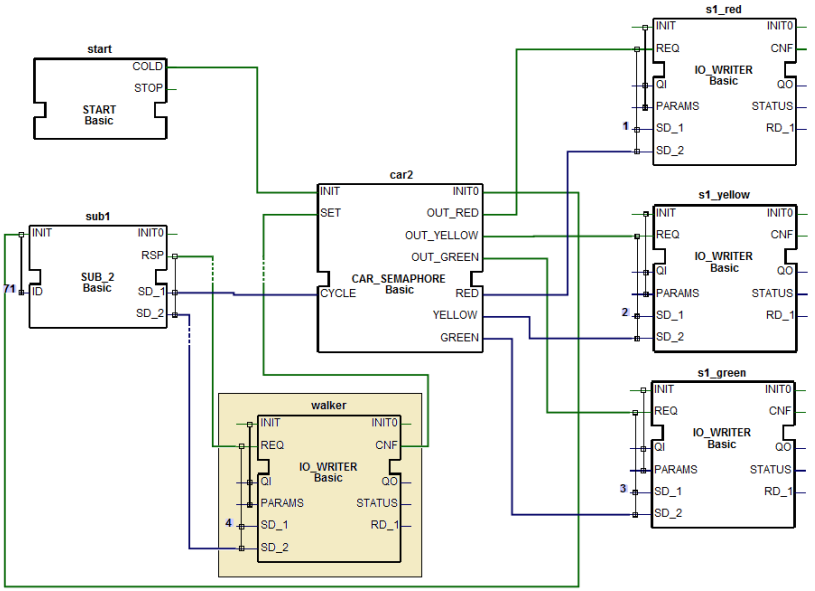
SEM_CONTROL sendo ativado por um bloco E_CYCLE a cada 5 segundos e um bloco CAR_SEMAPHORE. Portanto, PCSE fará o controle geral dos semáforos e também irá controlar um dos semáforos para carros, os demais dados de SEM_CONTROL são enviados através de um bloco PUBLISHER chamado *pub1*.

Figura 48 – Aplicação executada na máquina virtual para PC.



A máquina virtual Atmega2560, que executa no Arduino, irá processar a parte do sistema mostrada na Figura 49. Essa parte do sistema controla um semáforo para carros e um semáforo para pedestres, sendo que os dados para controle do semáforo são recebidos por um bloco do tipo SUBSCRIBER chamado *sub1*.

Figura 49 – Aplicação executada na máquina virtual para Atmega2560.



As partes em destaque nas Figuras 48 e 49, indicam os blocos e conexões que serão enviados para as máquinas virtuais em tempo de execução.

O suporte a pedestres consiste num botão de acionamento, que será lido pelo bloco *btwalker* (Figura 48), e por duas luzes (verde e vermelho) que são controladas pelo bloco *walker* (Figura 49). Na Figura 49, como na configuração inicial o bloco *walker* ainda não existe, então haverá uma conexão entre *sub1.RSP* e *sem.SET*, isso faz com que *sem1* seja acionado.

Esse experimento pretende demostrar que a infraestrutura proposta pode atender aos três requisitos da norma IEC 61499, que são: portabilidade, configurabilidade e interoperabilidade.

A portabilidade é demostrada ao executar blocos como CAR_SEMAPHORE nas duas máquinas virtuais implementadas, cada uma

para um hardware diferente. O suporte a configuração dinâmica é demostrada ao acrescentar o semáforo para pedestres com o sistema em funcionamento.

A infraestrutura interpreta um sistema descrito em formato *xml* de acordo com a norma IEC 61499.

4.1.1 Execução do Sistema de Semáforos

A execução do sistema de semáforos consiste em acionar o servidor e as máquinas virtuais, em seguida o arquivo *System.xml*, o qual contém uma descrição em *xml* do sistema de semáforos, é passado como entrada para a ferramenta de atualização dinâmica chamada FAD.

Inicialmente, a ferramenta FAD interpreta o *xml* extraindo blocos e a configuração do sistema, isto é, a forma com que os blocos estão distribuídos entre os dispositivos.

Os blocos extraídos são transformados para a linguagem ST e em seguida compilados por um compilador, o qual foi projetado para compilar os blocos e gerar instruções para a máquina virtual. Em seguida é gerada uma lista que é um roteiro de configuração inicial do sistema, parte dessa lista é mostrada na Figura 50.

Figura 50 – Parte do *xml* gerado para configurar o sistema.

```
<Requests>
  <Request Action="Create">
    <FB Name="start" Type="START" Device="DEV10"
    Resource="RES0"/>
  </Request>
  <Request Action="Create">
    <FB Name="sem1" Type="CAR_SEMAPHORE" Device="DEV10"
    Resource="RES0"/>
  </Request>
  <Request Action="Create">
    <FB Name="s1_red" Type="IO_WRITER" Device="DEV10"
    Resource="RES0"/>
  </Request>
  <Request Action="WRITE">
    <Connection Source="1" Destination="s1_red.SD_1"
    Device="DEV10" Resource="RES0"/>
  </Request>
  [...]
```

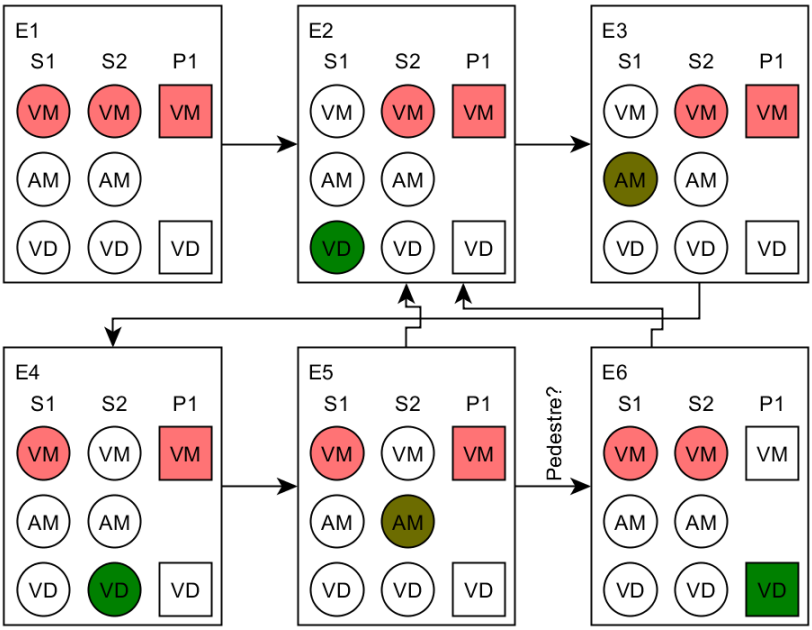
Nesse trecho da lista são criados os blocos *start*, *sem1* e *s1_red*, e também a conexão de uma constante com a variável *SD_1* do bloco *s1_red*. Em seguida, o usuário pode invocar um comando para que o roteiro seja

executado. Nesse caso a ferramenta FAD carrega esse roteiro, conecta-se ao servidor de máquinas virtuais e executa-o.

Após carregar o sistema inicial, que foi mostrado nas Figuras 48 e 49, sem os blocos *btwalker* e *walker* respectivamente que corresponde ao semáforo do pedestre, verificou-se que a infraestrutura atendeu aos requisitos de portabilidade e interoperabilidade. Pois, exatamente os mesmos blocos de função foram executados pelas duas máquinas virtuais envolvidas, sem a necessidade de recompilar os blocos para as diferentes versões da máquina.

A execução do sistema inicial é mostrada na Figura 51, a qual descreve os estados dos semáforos. Na configuração inicial do sistema, o estado E6 ainda não existe.

Figura 51 – Estados dos semáforos durante a execução do sistema. VD representa a lâmpada verde do semáforo, AM a amarela e VM a vermelha. S1 e S2 indicam as três lâmpadas dos semáforos para carros e P1 indica o semáforo para pedestres.



No estado inicial E1, os semáforos são alterados para o estado vermelho. Em seguida, o semáforo 1 inicia o ciclo de verde, amarelo e, por fim, vermelho. Quando o semáforo 1 muda para o estado vermelho, o

semáforo 2 tem seu estado alterado para verde, esse irá executar seu ciclo e ao terminar, o semáforo 1 inicia novamente seu ciclo.

Nessa configuração inicial, o semáforo do pedestre não funciona. Pois não há nenhuma ação atribuída ao botão do pedestre ou ao seu semáforo.

No passo seguinte, executa-se uma configuração que irá acrescentar os blocos em destaque nas Figuras 48 e 49 com suas respectivas conexões.

Observou-se que essa reconfiguração ocorre sem a paralisação ou reinicialização do sistema inicial que já está em execução. Isso atende ao requisito de configurabilidade da norma IEC 61499.

Essa reconfiguração alterou o sistema para que o mesmo suporta-se o semáforo para pedestres. Isso incluiu o estado E6 ao sistema.

Na Figura 51, ocorre um desvio para o estado E6 quando o botão do pedestre é pressionado, nesse estado os semáforos para carros são fechados, o de pedestres é aberto e fechado após 5 segundos (tempo definido no bloco *scycle* da Figura 48). Em seguida, o sistema retorna ao estado E2.

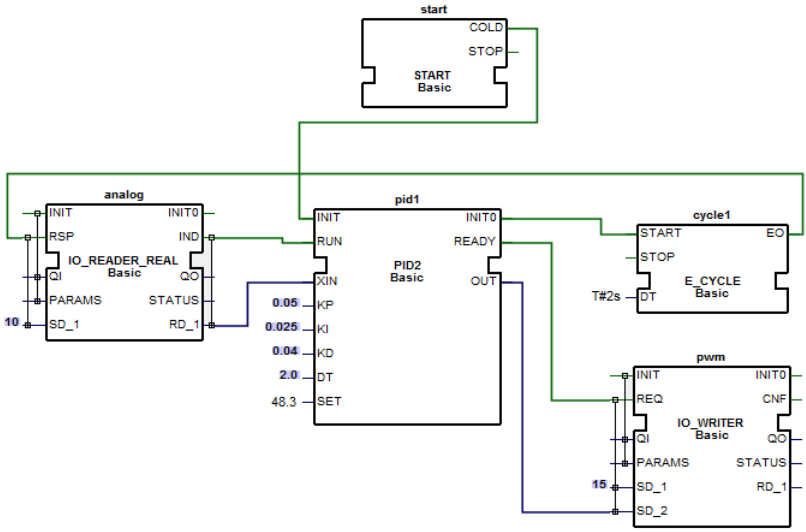
Também se verificou nesse experimento que a velocidade da rede *Ethernet* pode facilmente encher o *buffer* de entrada de rede das máquinas virtuais que executam em plataformas como o Arduino. A infraestrutura implementada já é capaz de contornar isso, simplesmente aguardando que a máquina virtual envie respostas para cada comando recebido. Entretanto, isso demonstra que a rede não está sendo bem aproveitada e, como forma de acelerar o processo de configuração, pode-se configurar mais de uma máquina virtual ao mesmo tempo, o que é sugerido como um trabalho futuro.

4.2 O SISTEMA DE CONTROLE PID

O controle PID (*Proportional-Integral-Derivative*) é um controlador de 3-termos que, devido a sua intuitividade, relativa simplicidade e performance satisfatória no controle de muitos processos, tem sido adotado como um padrão no controle de processos industriais (VISIOLI, 2006). Por serem amplamente usados na indústria, dois cenários envolvendo esse tipo de controle foram testados.

Para executar o controle PID foi utilizado um bloco de funções chamado PID, o qual está disponível no FBE. Esse bloco recebe como entrada o valor do sensor de temperatura e os parâmetros para o algoritmo PID. E como saída, emite um valor para controlar o ambiente até que atinja o valor de SET. A configuração do sistema para o controle PID é mostrada na Figura 52.

Figura 52 – Blocos de funções para controle PID.



O bloco *start* inicia o bloco *pid1* que gera o evento de START no bloco *cycle1*. O bloco *cycle1* dispara um evento a cada 2 segundos, esse evento executa o bloco *analog* que faz uma leitura de um sensor do processo controlado. O valor do sensor de temperatura é enviado para *pid1* que executa o algoritmo da Figura 53. O resultado é enviado para o bloco *pwm* que escreve numa saída que irá atuar sobre o processo controlado.

Figura 53 – Algoritmos do bloco de funções PID.

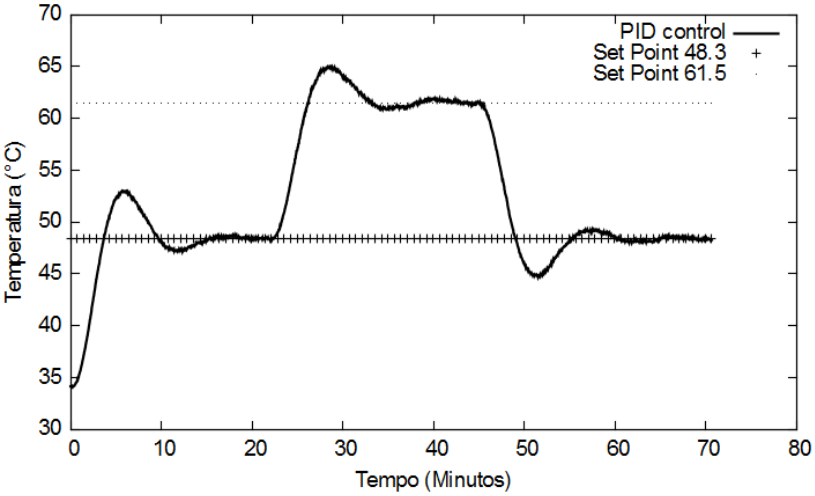
```
ALGORITHM ALG_INIT IN ST:
  previousError := 0.0;
  integral := 0.0;
END_ALGORITHM


ALGORITHM ALG_RUN IN ST:
  error := SET - XIN;
  integral := integral + error*DT;
  derivative :=
    (error - previousError)/DT;
  OUT := (KP*error + KI*integral +
    KD * derivative);
  previousError := error;
END_ALGORITHM
```

O primeiro teste consiste em controlar a temperatura sobre uma resistência elétrica através da aplicação de potência elétrica na mesma.

Para esse experimento, a rede de blocos precisa ser executada a cada 2 segundos, já que a resposta de um controle de temperatura é lenta como pode ser visto no gráfico da Figura 54, na qual o experimento teve duração de 70 minutos.

Figura 54 – Aplicação do controle PID sobre um processo de controle de temperatura.

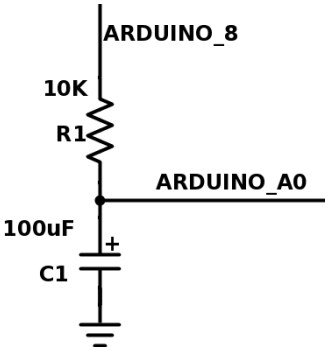


Nessa execução também foi verificada a capacidade da infraestrutura proposta de alterar o programa aplicativo em tempo de execução. Para isso, foi alterada a constante de valor 48.3 para 61.5 aos 23 minutos. Aos 45 minutos, a constante foi novamente alterada para 48.3

Um segundo teste com o controlador PID foi feito em um experimento que pudesse ter o comportamento simulado, permitindo que os resultados obtidos pudessem ser comparados com os resultados dessa simulação. A simulação foi feita usando o MATLAB (MathWorks Inc., 2013).

Esse experimento consiste na aplicação do controlador PID sobre um circuito RC, que possui dois componentes eletrônicos, um resistor de 10KOHMS e um capacitor de 100 μF (Microfarad). O objetivo do controle PID nessa aplicação é controlar a tensão elétrica sobre o capacitor. A Figura 55 mostra o diagrama elétrico utilizado para os testes.

Figura 55 – Diagrama elétrico do circuito RC.



A conexão ARDUINO_8 indica que o pino 8 do Arduino foi conectado nesse ponto, o mesmo vale para a conexão ARDUINO_A0. A função *AnalogWrite(int)* do Arduino foi utilizada para gerar uma tensão no pino ARDUINO_8 de 0V a 5V. E a função *AnalogRead(pin)* foi utilizada para fazer a leitura da tensão no pino ARDUINO_A0.

Na máquina virtual ICARU-FB, a interface de processo fornece acesso para *AnalogRead(A0)* e *AnalogWrite(8)* nos endereços 10 e 15 respectivamente. Portanto, como mostrado na Figura 52, o valor do pino A0 é enviado como entrada do bloco *pid1* na variável XIN, e a saída OUT é enviada para o pino ARDUINO_8.

Com esse circuito RC foram executados dois testes cujos resultados foram comparados com uma simulação no MATLAB. Utilizou-se a rede de blocos mostrada na Figura 52 com os parâmetros da Tabela 3.

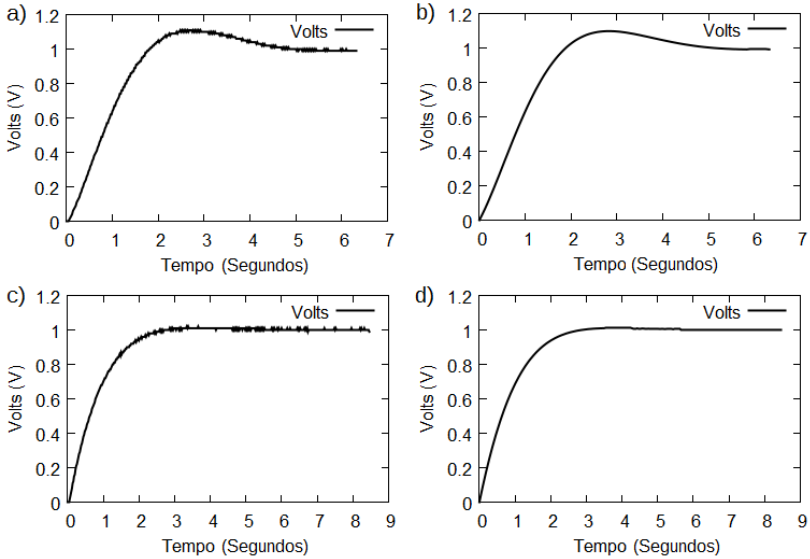
Tabela 3 – Parâmetros do controlador PID para os dois testes executados.

	KP	KI	KD	SET	DT
Teste 1	0,5	1,5	0	1,0	0,02s
Teste 2	1,0	1,3	0	1,0	0,02s

O teste com o controlador PID e o circuito RC consiste em acionar a máquina virtual ICARU-FB, efetuar o envio dos blocos e monitorar separadamente a tensão elétrica no pino ARDUINO_A0. Ambos os testes foram executados da mesma forma, a única diferença está nos parâmetros.

Os resultados desses testes são mostrados na Figura 56, na qual a) mostra o resultado do Teste 1 e sua simulação em b), e c) mostram o resultado para o Teste 2 e sua simulação em d).

Figura 56 – Resultados dos testes com o controlador PID sobre o circuito RC mostrados em a) e c), acompanhados da simulação no MATLAB em b) e d).



Observa-se que os resultados do experimento são bastante similares aos obtidos com a simulação.

Os testes com o controlador PID demonstraram a capacidade da infraestrutura de atender ao requisito de configurabilidade da norma IEC 61499, a qual foi mostrada ao alterar em tempo de execução a constante conectada na variável SET do bloco PID.

4.3 DISCUSSÕES

A máquina virtual ICARU-FB permite a portabilidade das aplicações ao manter a mesma implementação do seu núcleo em todas as plataformas. Entretanto a portabilidade só é garantida quando as diferentes implementações do módulo *ProcessInterface* são compatíveis, isso é mostrado no sistema de semáforo, o qual executa o mesmo bloco IO_WRITER, por exemplo, tanto no Arduino ATmega2560 quanto no PC. Assim, nas duas versões da máquina virtual, o acesso a saída de endereço 1 corresponde a lâmpada vermelha do semáforo.

Foi verificado também, através da implementação da interface mostrada na Figura 43, que a máquina virtual ICARU-FB pode atuar também em interfaces com o usuário, bastando que o módulo *ProcessInterface* seja implementado para essa finalidade. Assim, é possível utilizar a IEC 61499 desde a interface com o usuário até o controle de sensores e atuadores na linha de produção. E, considerando que o núcleo da máquina foi mantido intacto em todas as versões, não deverá ocorrer divergências na execução da norma IEC 61499.

A norma IEC 61499 não define um mecanismo para garantir a consistência, da aplicação, após a execução de uma reconfiguração dinâmica. Portanto, a remoção de um bloco pode causar inconsistências no sistema, apesar do código executar corretamente. No sistema de semáforos, por exemplo, não há mecanismos capazes de verificar se a remoção de um bloco pode, por exemplo, manter um dos semáforos sempre aberto (luz verde acionada), o que seria uma inconsistência.

Observou-se que, em alguns casos, o bloco de funções só pode ser removido em determinados estados do ECC, que seriam estados seguros. Por exemplo, um bloco de funções que controla um semáforo de veículos, só pode ser removido se o estado atual do semáforo é fechado. Uma possível abordagem é a definição de um estado seguro do ECC como parâmetro do comando de remoção.

A marcação de estados é capaz de resolver apenas um problema local, relacionado a remoção de um bloco de funções, mas não é capaz de resolver um problema global, por exemplo, impedir que a remoção do bloco afete outra parte do sistema. Assim como, também não pode resolver os possíveis problemas causados pela remoção de uma conexão. A verificação da consistência global da aplicação exige o uso de técnicas de verificação formal de sistemas e não está no escopo desse trabalho.

5 CONCLUSÃO

Foi apresentado um projeto e implementação de uma infraestrutura de software capaz de executar um sistema especificado em conformidade com a norma IEC 61499. Essa infraestrutura é composta de um pré-processador, um compilador e uma máquina virtual, sendo que a máquina virtual foi implementada para executar tanto em um PC quanto em um Arduino ATmega2560. Essa solução foi integrada ao editor de blocos de funções FBE.

Dois estudos de caso foram implementados, para verificar o suporte a características definidas na norma como: portabilidade, configurabilidade e interoperabilidade. Esses estudos de caso consistem em um sistema de controle PID e um sistema de controle de semáforos, ambos foram projetados usando o editor FBE. Os exemplos demonstraram que o ambiente é capaz de executar especificações feitas de acordo com a norma IEC 61499, uma vez que os arquivos *xml* foram definidos usando o editor de blocos FBE, portanto os sistemas estão em conformidade com essa norma.

Sistemas de controle PID são amplamente utilizados na indústria, e os resultados esperados são facilmente simulados em ferramentas como MATLAB, o que torna fácil a verificação dos resultados obtidos com a execução do sistema. Com o sistema de controle de semáforos foi possível verificar características como interoperabilidade e portabilidade, uma vez que parte do sistema foi executada em um PC e outra no Arduino ATmega2560. Nos dois casos, durante a execução, foram alteradas as conexões entre os blocos, sendo que no sistema de controle de semáforos também foram alterados alguns blocos de funções que estavam em execução, essas alterações demostram a possibilidade de reconfiguração dinâmica. Esses estudos de caso são uma evidência de que a solução proposta é viável e que a norma IEC 61499 pode ser implementada em ambientes com recursos extremamente limitados.

5.1 LIMITAÇÕES

Durante os testes da infraestrutura proposta pôde-se observar algumas limitações relacionadas ao hardware. A principal é baixa capacidade de memória da plataforma escolhida para os experimentos (microcontrolador Atmega2560), isso é uma característica comum em microcontroladores de 8 bits. Esse problema pode ser contornado acrescentando hardware específico para aumentar a memória da plataforma, entretanto, a execução de vários blocos pode degradar o tempo de resposta, uma vez que o microcontrolador possui baixa capacidade de processamento.

Outro problema é a forma com que a máquina gerencia o compartilhamento do tempo de execução de cada bloco, com a finalidade de implementar uma máquina virtual que consumisse poucos recursos optou-se por um modelo de compartilhamento de tempo cooperativo, a execução de um bloco de funções nunca é interrompida pela máquina virtual, ao invés disso, cada bloco deve liberar a máquina para que esta inicie a execução do próximo bloco. Com essa solução evita-se a necessidade da máquina virtual armazenar o estado da execução de cada bloco.

5.2 TRABALHOS FUTUROS

Como continuação desse trabalho podemos listar:

- A implementação do suporte a blocos compostos que são definidos na norma IEC 61499;
- Implementação da marcação de estados seguros para a remoção de um bloco de funções, como mencionado na Seção 4.3;
- A definição de um mecanismo para verificar a consistência global do sistema no caso de reconfiguração do mesmo;
- Estudos de casos usando memória externa, com a finalidade de verificar o desempenho da máquina virtual;
- Automatizar os passos de reconfiguração dinâmica, da forma que está implementado atualmente, é responsabilidade do programador especificar os passos com que os blocos e conexões serão atualizados no caso de uma reconfiguração dinâmica, em sistemas de maior porte isso pode ser bastante complexo;
- Possibilitar a configuração de várias máquinas virtuais simultaneamente, o processo de configuração das máquinas virtuais está sendo feito de forma sequencial, uma máquina de cada vez, o principal gargalo para configuração da máquina virtual é o hardware que está recebendo os blocos, não o que está gerenciando o processo, a atualização simultânea de várias máquinas poderia otimizar bastante esse processo, aproveitando melhor a rede de comunicação;
- Propor uma definição formal para algumas partes em que a norma permite projetos ambíguos. É possível definir redes de blocos de funções com comportamentos distintos, dependendo do ambiente em

que essa definição seja executada. Uma definição formal e uma ferramenta de verificação capaz de identificar a ocorrência dessas situações seria uma importante contribuição;

- Implementar um tratamento de exceções para a máquina virtual, pois a detecção e a localização de erros é importante para corrigir rapidamente possíveis problemas em sistemas de controle.

- Tratar requisitos de tempo real que são necessários em grande parte das aplicações industriais.

REFERÊNCIAS

4DIAC-CONSORTIUM. *Framework for Distributed Industrial Automation and Control*. 2014. Access Date July 2013. [Online]. Disponível em: ⟨http://www.fordiac.org/⟩.

ARDUINO. **Arduino: Open-Source Electronics Prototyping Plataform**. 2014. Access Date Jan 2013. [Online]. Disponível em: ⟨http://www.arduino.cc/⟩.

ARM Ltd. **ARM: The Architecture for the Digital World**. 2014. Access Date September 2013. [Online]. Disponível em: ⟨http://www.arm.com/⟩.

ARNOLD, K.; GOSLING, J.; HOLMES, D. **A Linguagem de Programação Java**. 4st. ed. São Paulo: Pearson Education, 2006. ISBN 0-321-34980-6.

ATMEL. **8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash**. 2014. Access Date Jan 2014. [Online]. Disponível em: ⟨http://www.atmel.com/images/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf⟩.

BOB, G. **HaikuVM: a small JAVA VM for microcontrollers**. 2014. Access Date September 2013. [Online]. Disponível em: ⟨http://haiku-vm.sourceforge.net/⟩.

BROUWERS, N.; CORKE, P.; LANGENDOEN, K. Darjeeling, a java compatible virtual machine for microcontrollers. In: **Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion**. New York, NY, USA: ACM, 2008. (Companion '08), p. 18–23. ISBN 978-1-60558-369-3. Disponível em: ⟨http://doi.acm.org/10.1145/1462735.1462740⟩.

CENGIC, G.; AKESSON, K. On formal analysis of iec 61499 applications, part a: Modeling. **Industrial Informatics, IEEE Transactions on**, v. 6, n. 2, p. 136–144, May 2010. ISSN 1551-3203.

CENGIC, G.; AKESSON, K. On formal analysis of iec 61499 applications, part b: Execution semantics. **Industrial Informatics, IEEE Transactions on**, v. 6, n. 2, p. 145–154, May 2010. ISSN 1551-3203.

CENGIC, G.; LJUNGKRANTZ, O.; AKESSON, K. Formal modeling of function block applications running in iec 61499 execution runtime. In: **Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on**. Prague: IEEE, 2006. p. 1269–1276. ISBN 0-7803-9758-4.

CHOUINARD, J.; BRENNAN, R. W. Software for next generation automation and control. In: **IEEE International Conference on Industrial Informatics**. Singapore: IEEE, 2006. p. 886–891. ISBN 0-7803-9700-2.

DUBININ, V.; VYATKIN, V. Semantics-robust design patterns for iec 61499. **Industrial Informatics, IEEE Transactions on**, v. 8, n. 2, p. 279–290, May 2012. ISSN 1551-3203.

eCosCentric Ltd. **eCos**. 2014. Access Date September 2013. [Online]. Disponível em: ⟨http://ecos.sourceware.org/⟩.

FERRARINI, L.; VEBER, C. Implementation approaches for the execution model of iec 61499 applications. In: **Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on**. Berlin: IEEE, 2004. p. 612–617.

HALL, K.; STARON, R.; ZOITL, A. Challenges to industry adoption of the iec 61499 standard on event-based function blocks. In: **Industrial Informatics, 2007 5th IEEE International Conference on**. Vienna: IEEE, 2007. v. 2, p. 823 –828. ISSN 1935-4576.

HARBS, E. **CNC-C2: um controlador aderente às normas ISO 14649 E IEC 61499**. 2012. Access Date Jan 2014. [Online]. Disponível em: http://www.tede.udesc.br/tde_busca/arquivo.php?codArquivo=3141.

HOLOBLOCK Inc. **HOLOBLOCK.com - FBDK - Function Block Development Kit**. 2014. Access Date July 2013. [Online]. Disponível em: ⟨http://www.holobloc.com/⟩.

ICS Triplex. **ISaGRAf**. 2014. Access Date Jan 2014. [Online]. Disponível em: ⟨http://www.isagraf.com/⟩.

IEC. **Inernational Standard IEC 61131-3: Programmable Logic Controllers - Part 3: Programming languages**. 2003.

IERUSALIMSCHY, R. **Programming in Lua**. Rio de Janeiro: Lua.org, 2006. ISBN 9788590379829.

KUO, M. et al. Determining the worst-case reaction time of iec 61499 function blocks. In: **Industrial Informatics (INDIN), 2010 8th IEEE International Conference on**. Osaka: IEEE, 2010. p. 1104–1109.

LEWIS, R. **Modelling Control Systems Using IEC 61499: Applying function blocks to distributed systems**. 1th. ed. London: Institution of Engineering and Technology, 2008.

LINDHOLM, T. et al. **The Java Virtual Machine Specification, Java SE 8 Edition**. RedWood City: Pearson Education, 2014. ISBN 9780133922721.

MathWorks Inc. **MATLAB: The Language of Technical Computing**. 2013. Access Date September 2013. [Online]. Disponível em: ⟨http://www.mathworks.com/products/matlab/⟩.

NEGRI, G. H. **GASR-FBE**. 2013. Access Date Jun 2014. [Online]. Disponível em: ⟨http://sourceforge.net/projects/gasrfbe/⟩.

nxtControl GmbH. **nxtRT61499F - run time systems for distributed control engineering**. 2014. Access Date Jan 2014. [Online]. Disponível em: ⟨http://www.nxtcontrol.com/en/products/nxtrt61499f.html⟩.

PINTO, L. I. **ICARU-FB**. 2014. Access Date May 2014. [Online]. Disponível em: ⟨http://sourceforge.net/projects/icarufb/⟩.

STRASSER, T. et al. Modeling of reconfiguration control applications based on the iec 61499 reference model for industrial process measurement and control systems. In: **Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on**. Prague: IEEE, 2006. p. 127–132.

STRASSER, T. et al. Framework for distributed industrial automation and control (4diac). In: **Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on**. Daejeon: IEEE, 2008. p. 283 –288. ISSN 1935-4576.

STRASSER, T. et al. Design and execution issues in iec 61499 distributed automation and control systems. **Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on**, v. 41, n. 1, p. 41–51, 2011. ISSN 1094-6977.

SUNDER, C. et al. Usability and interoperability of iec 61499 based distributed automation systems. In: **Industrial Informatics, 2006 IEEE International Conference on**. Singapore: IEEE, 2006. p. 31–37.

TIEGELKAMP, K.-H. J. **IEC 61131-3: Programming Industrial Automation Systems**. 2th. ed. Berlin: Springer, 2010.

VISIOLI, A. **Practical PID Control**. Brescia, Italy: Springer, 2006. ISBN 978-1-84628-586-8.

VYATKIN, V. The iec 61499 standard and its semantics. **Industrial Electronics Magazine, IEEE**, v. 3, n. 4, p. 40–48, 2009. ISSN 1932-4529.

VYATKIN, V. Iec 61499 as enabler of distributed and intelligent automation: State-of-the-art review. **Industrial Informatics, IEEE Transactions on**, v. 7, n. 4, p. 768–781, Nov 2011. ISSN 1551-3203.

VYATKIN, V. **IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design**. 2th. ed. New Zealand: International Society of Automation, 2012. ISBN 978-1-936007-93-6.

YOONG, L. H.; ROOP, P.; SALCIC, Z. Efficient implementation of iec 61499 function blocks. In: **Industrial Technology, 2009. ICIT 2009. IEEE International Conference on**. Gippsland, VIC: IEEE, 2009. p. 1 –6.

YOONG, L. H. et al. A synchronous approach for iec 61499 function block implementation. **Computers, IEEE Transactions on**, v. 58, n. 12, p. 1599–1614, Dec 2009. ISSN 0018-9340.

ZOITL, A. et al. Executing real-time constrained control applications modelled in iec 61499 with respect to dynamic reconfiguration. In: **Industrial Informatics, 2005. INDIN '05. 2005 3rd IEEE International Conference on**. Perth, Australia: IEEE, 2005. p. 62 – 67.

ZOITL, A. et al. A device and resource execution model for iec 61499 control devices. In: **Industrial Informatics, 2007 5th IEEE International Conference on**. Vienna: IEEE, 2007. v. 2, p. 1143–1149. ISSN 1935-4576.

ZOITL, A.; SUNDER, C.; TERZIC, I. Dynamic reconfiguration of distributed control applications with reconfiguration services based on iec 61499. In: **Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on**. Prague: IEEE, 2006. p. 109–114.

Apêndices

APÊNDICE A - GRAMÁTICA DA LINGUAGEM ST

```
StFile:
    Blocks System
    | Blocks
    | System
    ;

Blocks:
    Blocks FunctionBlock
    | FunctionBlock
    ;

FunctionBlock:
    T_FUNCTION_BLOCK T_ID
        EventInputListContainer
        EventOutputListContainer
        EcStatesListContainer
        EcTransListContainer
        VarInputListContainer
        VarOutputListContainer
        VarListContainer
        FbContinue
    T_END_FUNCTION_BLOCK
    ;

FbContinue:
      AlgorithmList
    | FbsListContainer
      EventConnListContainer
      DataConnListContainer
    ;


EventInputListContainer:
    /* empty */
    | T_EVENT_INPUT
        EventList
```

```
    T_END_EVENT
    ;

EventOutputListContainer:
    /* empty */
    | T_EVENT_OUTPUT
        EventList
    T_END_EVENT
    ;
EventList:
    /* */
    | EventList  Event T_SEMICOLON
    ;
Event: T_ID;

EcStatesListContainer:
    T_EC_STATES
        EcStatesList
    T_END_STATES
    ;
EcStatesList:
    EcStatesList EcStates
    | EcStates
    ;

EcStates:
    T_ID  EcActionsList T_SEMICOLON
    ;

EcActionsList:
    /* */
    | T_TWO_POINTS EcActionsListComa
    ;

EcActionsListComa:
      EcActionsListComa T_COMA EcAction
    | EcAction
    | /* */
    ;
```

```
EcAction:
       T_ID T_ARROW T_ID
    |  T_ARROW T_ID
    |  T_ID T_ARROW
    ;

EcTransListContainer:
    T_EC_TRANSITIONS
        EcTransList
    T_END_TRANSITIONS
    ;
EcTransList:
    /* */
    | EcTransList EcTrans
    ;
EcTrans:
    T_ID T_TO T_ID T_ASSIGNMENT  Expr
        T_SEMICOLON
    ;

VarInputListContainer:
    /* empty */
    | T_VAR_INPUT
        VarList
    T_END_VAR
    ;
VarOutputListContainer:
    /* empty */
    | T_VAR_OUTPUT
        VarList
    T_END_VAR
    ;
VarListContainer:
    /* empty */
    | T_VAR
        VarList
    T_END_VAR
    ;
VarList:
    /* */
```

```
    | VarList VarDecl
    ;
VarDecl:
    T_ID T_TWO_POINTS T_ID VarDeclAssig T_SEMICOLON
    ;
VarDeclAssig:
    /* empty */
    | T_ASSIGNMENT T_NUMBER
    ;

System:
    T_SYSTEM T_ID
        DeviceList
    T_END_SYSTEM
    ;

DeviceList:
    Device DeviceList
    | Device
    ;

Device:
    T_DEVICE T_ID T_TWO_POINTS T_ID
        ResourceList
    T_END_DEVICE
    | T_DEVICE T_ID T_TWO_POINTS T_ID  T_OP T_CP
        ResourceList
    T_END_DEVICE
    ;

ResourceList:
    ResourceList Resource
    | Resource
    ;

Resource:
    T_RESOURCE T_ID T_TWO_POINTS T_ID T_OP T_CP
        FbsListContainer
        EventConnListContainer
        DataConnListContainer
```

```
    T_END_RESOURCE
    ;

FbsListContainer:
    T_FBS
        FbsList
    T_END_FBS
    ;

FbsList:
    FbsList FbsDeclaration
    | FbsDeclaration
    ;

FbsDeclaration:
    T_ID T_TWO_POINTS T_ID T_SEMICOLON
    | T_ID T_TWO_POINTS T_ID T_OP  ConstAssigList
      T_CP T_SEMICOLON
    ;

ConstAssigList:
    ConstAssigList ConstAssig
    | ConstAssig
    ;
ConstAssig:
    T_ID T_ASSIGNMENT T_NUMBER T_SEMICOLON
    ;

EventConnListContainer:
    T_EVENT_CONNECTIONS
        ConnectionList
    T_END_CONNECTIONS
    ;

DataConnListContainer:
    T_DATA_CONNECTIONS
        ConnectionList
    T_END_CONNECTIONS
    ;
```

```
ConnectionList:
    ConnectionList Connection
    | Connection
    ;
Connection:
    T_ID T_POINT T_ID T_TO T_ID T_POINT T_ID
      T_SEMICOLON
    ;


/* ST Algorithms */
AlgorithmList:
    /* */
    | AlgorithmList Algorithm
    ;

Algorithm:
    T_ALGORITHM T_ID T_IN T_ID T_TWO_POINTS
        StatementList
    T_END_ALGORITHM
    ;

StatementList:
    /* empty */
    | StatementList Assignment
    | StatementList If_stat
    | StatementList While_stat
    | StatementList Call_stat T_SEMICOLON
    | StatementList AsmBlock
    ;

AsmBlock:
    T_ASM AsmCode T_END_ASM
    ;

AsmCode:
    /* */
    | AsmCode Instruction
    ;
```

```
Instruction:
      T_LOAD AsmIdNum
    | T_CONST T_NUMBER
    | T_STORE AsmIdNum
    | AsmIf T_GOTO  AsmIdNumGoto
    | T_NISND
    | T_OR
    | T_NIRCV
    | T_GOTO AsmIdNumGoto
    | T_ID T_TWO_POINTS
    ;

AsmIf:
    T_IF
    | T_IFEQ
    ;

AsmIdNumGoto:
    T_ID
    | T_NUMBER
    | T_ADD T_NUMBER
    ;

AsmIdNum:
    T_ID
    | T_NUMBER
    | T_ADD T_NUMBER
    ;

Call_stat:
    T_BI_SET_REG T_OP Expr T_COMA Expr T_CP
    | T_BI_GET_REG T_OP Expr T_CP
    ;


While_stat:
    T_WHILE Expr T_DO
    StatementList T_END_WHILE T_SEMICOLON
    ;
```

```
If_stat:
    T_IF If_format
    ;

If_format:
         Expr T_THEN  StatementList T_END_IF
           T_SEMICOLON
        |  Expr T_THEN  StatementList  T_ELSE
            StatementList T_END_IF T_SEMICOLON
        ;



Assignment:
      T_ID T_ASSIGNMENT Expr  T_SEMICOLON
    ;

Expr:
    Logical_Expr
    ;

Logical_Expr:
      Logical_Expr T_OR Logical_Expr_p2
    | Logical_Expr_p2
    ;

Logical_Expr_p2:
      Logical_Expr_p2 T_AND Logical_Expr_p1
    | Logical_Expr_p1
    ;

Logical_Expr_p1:
     /* */
    | Relational_Expr
    ;

Relational_Expr:
      Relational_Expr Relational_Operator
        Arithmetic_Expr
```

```
    | Arithmetic_Expr
    ;

Arithmetic_Expr:
      Arithmetic_Expr T_ADD Arithmetic_Expr_p1
    | Arithmetic_Expr T_SUB Arithmetic_Expr_p1
    | Arithmetic_Expr_p1
    ;

Arithmetic_Expr_p1:
      Arithmetic_Expr_p1 T_MUL Factor
    | Arithmetic_Expr_p1 T_DIV Factor
    | Factor
    ;

Factor:

      T_NUMBER
    | T_OP Expr T_CP
    | T_SUB Factor
    | T_NOT Factor
    | T_ID
    | Call_stat
    ;


Relational_Operator:
      T_GT
    | T_GE
    | T_LT
    | T_LE
    | T_EQ
    ;
```

APÊNDICE B - CONJUNTO DE INSTRUÇÕES DA ICARU-FB

A máquina virtual possui um conjunto de instruções pequeno e parecido com a máquina virtual Java. Como a máquina virtual é uma máquina de pilha, todas as instruções operam sobre essa pilha.

FORMATO DAS INSTRUÇÕES DA MÁQUINA ICARU-FB

O tamanho das instruções é de 3 bytes, sendo 1 byte para a instrução e 2 para os parâmetros.

Para as instruções de operações aritméticas e relacionais, os parâmetros devem indicar qual o tipo dos operandos e se alguma conversão deve ser feita. Essas instruções são escritas da seguinte forma:

```
<instrução> <c>, <t>
```

O parâmetro <*c*> indica para qual tipo os dois operandos devem ser convertidos. Os quatro bits mais significativos (MSB) indicam para qual tipo o primeiro operando deve ser convertido, os quatro bits menos significativos (LSB) indicam para qual tipo o segundo operando deve ser convertido.

O parâmetro <*t*> indica os tipos dos parâmetros envolvidos na operação. Sendo que os quatro bits mais significativos indicam o tipo do primeiro operando, e os demais indicam o tipo do segundo operando.

Operações Aritméticas

ADD c, t	**SUB c, t**
Operação: push(popA()+popB())	**Operação:** push(popA()-popB())
Descrição: Retira dois elementos da pilha, efetua a soma e insere o resultado na pilha.	**Descrição:** Retira dois elementos da pilha, subtrai o segundo elemento retirado (B) do primeiro (A), e insere o resultado na pilha.

MUL c, t

Operação: push(popA()*popB())

Descrição: Retira dois elementos da pilha,multiplica-os, e insere o resultado na pilha.

DIV c, t

Operação: push(popA()/popB())

Descrição: Retira dois elementos da pilha, divide o primeiro elemento retirado (A) pelo segundo (B), e insere o resultado na pilha.

MOD c, t

Operação: push(popA() % popB())

Descrição: Retira dois elementos da pilha, calcula o resto da divisão do primeiro elemento retirado (A) pelo segundo (B), e insere o resultado na pilha.

Operações Lógicas

AND

Operação: push(popA() && popB())

Descrição: Retira dois elementos da pilha, efetua a operação lógica 'E', e insere o resultado na pilha.

OR

Operação: push(popA() || popB())

Descrição: Retira dois elementos da pilha, efetua a operação lógica 'OU', e insere o resultado na pilha.

NOT

Operação: push(! popA())

Descrição: Retira um elemento da pilha, efetua a operação lógica 'NÃO', e insere o resultado na pilha.

Operações Relacionais

EQ c, t

Operação: push(popA() == popB())

p16: Parâmetro de 16 bits

Descrição: Remove dois elementos da pilha e empilha 1 se os elementos forem iguais, 0 caso contrário.

GT c, t

Operação: push(popA() > popB());

p16: Parâmetro de 16 bits

Descrição: Remove dois elementos da pilha, e empilha 1 se o primeiro elemento A for maior que B, 0 caso contrário.

LT c, t

Operação: push(popA() < popB());

p16: Parâmetro de 16 bits

Descrição: Remove dois elementos da pilha, e empilha 1 se A ¡ B, 0 caso contrário.

GE c, t

Operação: push(popA() >= popB());

p16: Parâmetro de 16 bits

Descrição: Remove dois elementos da pilha, e empilha 1 se A >= B, 0 caso contrário.

LE c, t

Operação: push(popA() <= popB());

p16: Parâmetro de 16 bits

Descrição: Remove dois elementos da pilha, e empilha 1 se A <= B, 0 caso contrário.

Chamada/Retorno de Subrotina e Saltos

IF p16

Operação: if(popA()! =0) goto p16;

p16: Parâmetro de 16 bits

Descrição: Remove um elemento da pilha, se este for diferente de 0 salta para a posição p16.

CALL p16

Operação: push(MSB of PC); push(LSB of PC); PC = p16;

p16: Parâmetro de 16 bits

Descrição: Salta para a posição p16 salvando o ponteiro PC na pilha. Como a pilha é de 8 bits, primeiro coloca-se na pilha a parte mais significativa do PC (MSB), em seguida a menos significativa (LSB).

GOTO p16

Operação: PC = p16;

p16: Parâmetro de 16 bits

Descrição: Salta para a posição p16.

RET

Operação: PC = popB() << 8 || popA();

Descrição: Restaura o PC obtendo o novo valor da pilha, que foi salvo anteriormente pela instrução CALL. Se a pilha estiver vazia, a execução do bloco de funções é terminada.

Operações da Pilha e Outros Módulos da Máquina Virtual

DROP

Operação: popA();

Descrição: Remove um elemento da pilha.

CONST p

Operação: push(p);

Descrição: Empilha uma constante p.

STORE p

Operação: MemoryRam[p] = popA();

Descrição: Remove um elemento da pilha e coloca-o na posição p da memória RAM.

LOAD p

Operação: push(MemoryRam[p]);

Descrição: Empilha o valor da posição p da memória RAM.

ARLOAD p

Operação: push(pro_get[p]);

Descrição: Empilha o valor da posição p do módulo *Process Interface*.

ARSTORE

Operação :
A = popA();
B = popB();
pro_set(B,A);

Descrição: Atribui um valor A para a posição B do módulo *Process Interface*.

ALLOC p

Operação: ALLOC p;

Descrição: Aloca um espaço com tamanho p para o bloco de funções.

NISND

Operação :
id = pop()
data_size = pop()
(array)data = pop *n* bytes
net_send(data)

Descrição: Envia dados para rede, utilizada por PUBLISHER.

NIRCV p

Operação :
id = pop()
data_size = pop()
push((array)data)

Descrição: Carrega os dados para um SUBSCRIBER se existirem no módulo *Dispatcher*. Após executar essa instrução os dados recebidos sob o id *id* estarão disponíveis na pilha.

APÊNDICE C - INTERFACE E COMANDOS DO *MANAGER*

A Figura 57 mostra a interface do módulo *Manager*. A função `man_init()` é utilizada pela máquina virtual para inicializar o módulo. E a função `man_newMsg()` é utilizada pelo *Dispatcher* quando este precisa entregar alguma mensagem ao *Manager*.

Figura 57 – Interface do módulo *Manager*

```
int8 man_init();
int8 man_newMsg(int8 *msg, int8 size);
```

Fonte: Próprio autor

As mensagens enviadas ao *Manager* tem o seguinte formato:

$$< comando >< parametros >$$

Os comandos de reconfiguração dinâmica são mostrados na tabela 58. Na qual $<s:$ ... $>$ representa uma string cujo último carácter é nulo (0), *i8* representa um inteiro de 8 bits, *i16* representa um inteiro de 16 bits e *n* representa uma sequência de dados com *n* bytes.

Figura 58 – Comandos para reconfiguração dinâmica.

Código	Formato	Descrição
0xD1	0xD1 <s:nome do bloco> <i16:tamanho em bytes>	BLOCK_CREATE
0xD2	0xD2 <s:nome do bloco> <i8:posição> <i8:tamanho dos dados> <n:dados>	BLOCK_WRITE
0xD3	0xD3 <s:nome do bloco> <s:nome da instância> <i8:resource>	INSTANCE_CREATE
0xD4	0xD4 <s:nome da instância de origem> <i8:posição da variável de origem> <s:nome da instância de destino> <i8:posição da variável de destino> <i8:tamanho da variavel> <i8:resource> <i8:isEvent> <n:value>	CONNECTION_CREATE
0xD7	0xD7 <s:nome da instância> <i8:estado> <i8:resource>	INSTANCE_SET_STATE
0xE1	0xE1 <s:nome do bloco>	BLOCK_DELETE
0xE3	0xE3 <s:nome da instância> <i8:resource>	INSTANCE_DELETE
0xE4	0xE4 <s:nome da instância de origem> <i8:posição da variável de origem> <s:nome da instância de destino> <i8:posição da variável de destino>	CONNECTION_DELETE

Ao receber um comando, a máquina virtual responde informando se o comando foi executado com sucesso ou se houve algum erro. A resposta para sucesso é a sequência de dois bytes *0xA0 0x0A* e para erro é *0xA0 0x0C*.

Para enviar a resposta, a máquina virtual utiliza o módulo *NetworkInterface*.

A norma IEC 61499 define um modelo de desenvolvimento para automação e controle industrial, ela estabelece uma linguagem visual que pode facilitar a implementação de sistemas de controle distribuídos. Esse trabalho apresenta a proposta e implementação do ICARU-FB, um ambiente *Open Source* multiplataforma, que é capaz de executar a linguagem definida na norma IEC 61499 em arquiteturas com poucos recursos computacionais. Uma máquina virtual foi projetada e implementada para executar redes blocos de funções em plataformas de 8 bits com o mínimo de recursos. Ela também foi portada para executar em um computador de 64 bits. Dois estudos de caso foram realizados a fim de verificar a conformidade com a norma IEC 61499. Através desses estudos de caso, foi verificado que é possível atender aos requisitos da norma IEC 61499, como configurabilidade, interoperabilidade e portabilidade. Os estudos de caso também demonstraram a habilidade do ambiente de reconfigurar o software em tempo de execução.

Orientador: Cristiano Damiani Vasconcellos

Coorientador: Roberto Silvio Ubertino Rosso Jr.

Joinville, 2014