Um Framework de Sistema Operacional para a Implementação de Algoritmos de Controle Digital

Lucas Pires Camargo
Laboratório de Integração Software/Hardware
Universidade Federal de Santa Catarina
Joinville, Santa Catarina, Brasil
Email: camargo@lisha.ufsc.br

Giovani Gracioli
Laboratório de Integração Software/Hardware
Universidade Federal de Santa Catarina
Joinville, Santa Catarina, Brasil
Email: giovani@lisha.ufsc.br

***Resumo*—Grande parte dos sistemas de controle digitais possuem restrições de tempo real que somente podem ser atendidas por um sistema operacional de tempo real (SOTR). Geralmente, algoritmos de controle digital são implementados em um SOTR usando abstrações de tarefas periódicas, sensores e atuadores. Entretanto, desenvolvedores de sistemas de controle devem implementar o laço de controle completo, *i.e.,* amostragem dos dados dos sensores periódica, computação do controle e operações de atuação, sem qualquer suporte do sistema operacional para automatizar as operações de monitoramento dos sensores e atuação. Neste artigo, é proposto um framework de sistema operacional para automatizar a geração de algoritmos de controle digital (operações com sensores e atuadores e o laço principal de controle). Para demonstrar o uso do framework, foi implementado um algoritmo de controle do diferencial eletrônico automotivo e seu consumo de memória foi medido. Devido às sofisticadas técnicas de engenharia de software, tais como metaprogramação estática e projeto baseado em componentes, o framework consome pouca memória e provê uma interface simples para os desenvolvedores de sistemas de controle implementarem seus algoritmos.**

***Keywords*—Framework de sofware; Sistema Operacional; Algoritmos de controle;**

AGRADECIMENTOS

Este trabalho foi financiado pelo CNPq.

I. INTRODUÇÃO

Muitos sistemas de controle realimentados (*feedback*) são essencialmente periódicos, onde dados de entrada dos sensores e as saídas do controlador são amostrados em uma taxa fixa [1]. A Figura 1 mostra uma visão geral de uma estrutura básica de controle digital realimentado. A saída do controlador é enviada para um ou mais atuadores para controlar um processo específico. Um ou mais sensores realimentam o controlador com informações sobre o processo. Finalmente, o controlador avalia a diferença entre os valores lidos e o valor de referência para mudar as saídas do controlador [1].

Sistemas de controle digital baseados na teoria de controle realimentado estão presentes em diversos sistemas embarcados. Em automóveis, por exemplo, algoritmos de controle digital ajudam a melhorar a segurança. Sistemas de controle adaptativos e *airbags* são facilmente encontrados em carros comerciais e fornecem uma melhora significativa na segurança.

Comum aos algoritmos de controle digital é a existência de restrições de tempo real. Em um sistema de tempo real,

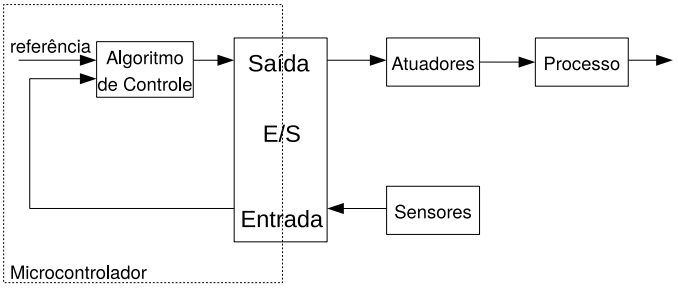

Figura 1. Visão geral de uma estrutura básica de controle digital realimentado.

a corretude não depende somente do seu comportamento lógico, mas também no tempo em que os cálculos são executados [2, 3]. Um sistema de tempo real crítico, como aqueles encontrados em um automóvel, nunca deve perder um prazo (*deadline*), caso contrário um dano catastrófico pode acontecer. Geralmente, o suporte a sistemas de tempo real é alcançado através de algoritmos de escalonamento tempo real e protocolos de compartilhamento de recursos, dentro de um sistema operacional de tempo real (SOTR) [3]. O desempenho e a estabilidade de um sistema de controle depende, entre outras características, das taxas de amostragem e os atrasos da computação (latência) [1]. Consequentemente, estão fortemente relacionados ao projeto do SOTR.

Tradicionalmente, algoritmos de controle digital são implementados em um SOTR usando abstrações de tarefas, nas quais permitem o sensoriamento de dados e atuação serem repetidas em uma frequência fixa. Entretanto, SOTRs atuais não fornecem uma estrutura para desenvolvedores de sistemas de controle implementarem seus controladores de forma eficiente e abstrata. Mais especificamente, desenvolvedores de sistemas de controle devem implementar o laço de controle completo, *i.e.*, amostragem periódica dos sensores, computação periódica do controle, e operações de atuação.

Com o intuito de atenuar essa limitação dos SOTRs atuais, neste trabalho é proposto uma framework de SO para automatizar a geração de algoritmos de controle digital. Através do uso de sofisticadas técnicas de engenharia de software, tais como projeto orientado a objetos (OOD) [4], metaprogramação estática [5] e projeto baseado em componentes (CBD) [6], o framework proposto permite automatizar a geração da amostragem periódica dos sensores e operações de atuação. Além disso, o framework provê uma interface para os desenvolvedores

facilmente implementarem seus algoritmos de controle em um SOTR. O framework foi implementado em um SOTR e sua viabilidade foi demonstrada medindo a saída de um algoritmo de controle do diferencial eletrônico de um carro usando hardware real. O consumo de memória do framework para este algoritmo também foi mensurado. Os resultados indicaram que o framework proposto facilita a implementação de algoritmos de controle digital, além de prover bom desempenho e baixo consumo de memória.

O restante deste artigo é organizado da seguinte forma. A Seção II revisa os principais conceitos correlacionados ao artigo. A Seção III apresenta o projeto do framework proposto. A Seção IV mostra a avaliação de um algoritmo de controle do diferencial eletrônico usando o framework implementado em um SOTR e executando sob uma plataforma de hardware real. A Seção V brevemente discute trabalhos relacionados. Finalmente, a Seção VI conclui o artigo e discute trabalhos futuros.

II. Background

Esta seção apresenta os principais conceitos usados pelo framework proposto. Os conceitos são divididos em dois tópicos principais que são detalhados nas próximas subseções: projeto de sistemas embarcados dirigidos pela aplicação e características configuráveis (*configurable features*) e metaprogramação estática em C++.

A. Projeto de Sistemas Embarcados Dirigidos pela Aplicação

O framework foi projetado usando conceitos da metodologia de projeto de sistemas embarcados dirigidos pela aplicação (ADESD) [7]. ADESD guia o desenvolvimento de sistemas operacionais dirigidos pela aplicação da análise do domínio a implementação. Ela é baseada nas estratégias de decomposição de domínio encontradas no projeto baseado em famílias (FBD) [8], projeto orientado a objetos (OOD) [4], programação orientada a aspectos (AOP) [5], metaprogramação estática [5] e projeto baseado em componentes (CBD) [6] para definir componentes que representam entidades de domínio significantes.

Na ADESD, o domínio do problema é analisado e decomposto em componentes independentes de plataforma ou abstrações[1] que são organizados em membros de famílias, como definido pelo FBD. Para reduzir as dependências e aumentar a reusabilidade das abstrações, a ADESD agrega a separação em aspectos da AOP ao processo de decomposição. Portanto, é possível identificar variações de cenários e propriedades não-funcionais para modelá-las como aspectos de cenário que cortam transversalmente todo o sistema. O adaptador de cenário envolve um componente e aplica o conjunto correspondente de aspectos a este componente [9].

A ADESD foi usada para fazer uma análise de domínio dos algoritmos de controle digital. Os principais componentes foram extraídos desta análise: monitoramento periódico dos sensores, computação do controle e atuação periódica. O framework de SO foi projetado seguindo esses principais componentes. O framework proposto é apresentado na Seção III.

[1]Aqui um componente é uma classe C++ com comportamento e interface bem definidos. Abstrações são componentes visíveis ao usuário.

B. Características Configuráveis e Metaprogramação Estática em C++

Adicionalmente às propriedades não-funcionais representadas como aspectos de cenário, na ADESD, diversos aspectos são identificados pelo projetista como *características configuráveis* (*configurable features*) dos componentes. Características configuráveis representam variações dentro de um componente e são usadas para mudar seu comportamento ou estrutura [7]. Classes *template*, chamadas de *Traits*, definem qual característica de um componente está ativa. A Figura 2 mostra a implementação de uma característica configurável usando uma estrutura de *template* da linguagem C++. Se a característica 1 (*feature_1*) está ativada (linha 2), comportamento adicional é executado dentro do método *Component::operation* (linha 7). Como o comando *if* é estaticamente avaliado pelo compilador, quando *feature_1* é falso, o compilador completamente remove o *if* do código executável final. Assim, é possível otimizar o código gerado e adicionar configurabilidade ao sistema.

```
1  template <> struct Traits<Component>
2      static const bool feature_1 = true;   // característica  está ativada
3      static const bool feature_2 = false ; // característica  está desativada
4  };
5  void Component::operation() {
6      ....
7      if ( Traits <Component>::feature_1) {
8          ....
9      }
10     ....
11 }
```

Figura 2. Exemplo de uma característica configurável usando uma classe *Trait* em C++.

Técnicas de metaprogramação também podem ser usadas para realizar computação e gerar código em tempo de compilação. Algoritmos são expressos usando recursão de *template* e especialização de classes de *template* [5]. A Figura 3 exemplifica a computação de um número fatorial usando um metaprograma em C++. *Factorial<7>* é instanciado em tempo de compilação (linha 10). Portanto, o compilador substitui *Factorial<7>::RET* pelo seu valor (5040) na linha 10 (*cout << 5040 << endl*). Isso significa que o programa fatorial não é executado em tempo de execução. Note que o compilador é usado como um processador para interpretar metaprogramas, simplesmente codificando dados como números e tipos [5].

```
1  template<int n> struct Factorial {
2      enum { RET = Factorial<n−1>::RET ∗ n };
3  };
4  // esta  especialização  termina  a  recursão
5  template<> struct Factorial<0> {
6      enum { RET = 1 };
7  };
8  //usa o programa Factorial
9  void main() {
10     cout << Factorial<7>::RET << endl; //imprime 5040
11 }
```

Figura 3. Metaprograma em C++ para calcular o fatorial de um número [5].

Metaprogramação estática é uma ferramenta poderosa para gerar código em tempo de compilação, reduzindo assim o sobrecusto em tempo de execução e as dependências de software. Essa característica é desejável em sistemas embarcados, onde os requisitos do sistema são geralmente conhecidos em tempo de projeto. Consequentemente, é possível gerar somente o código

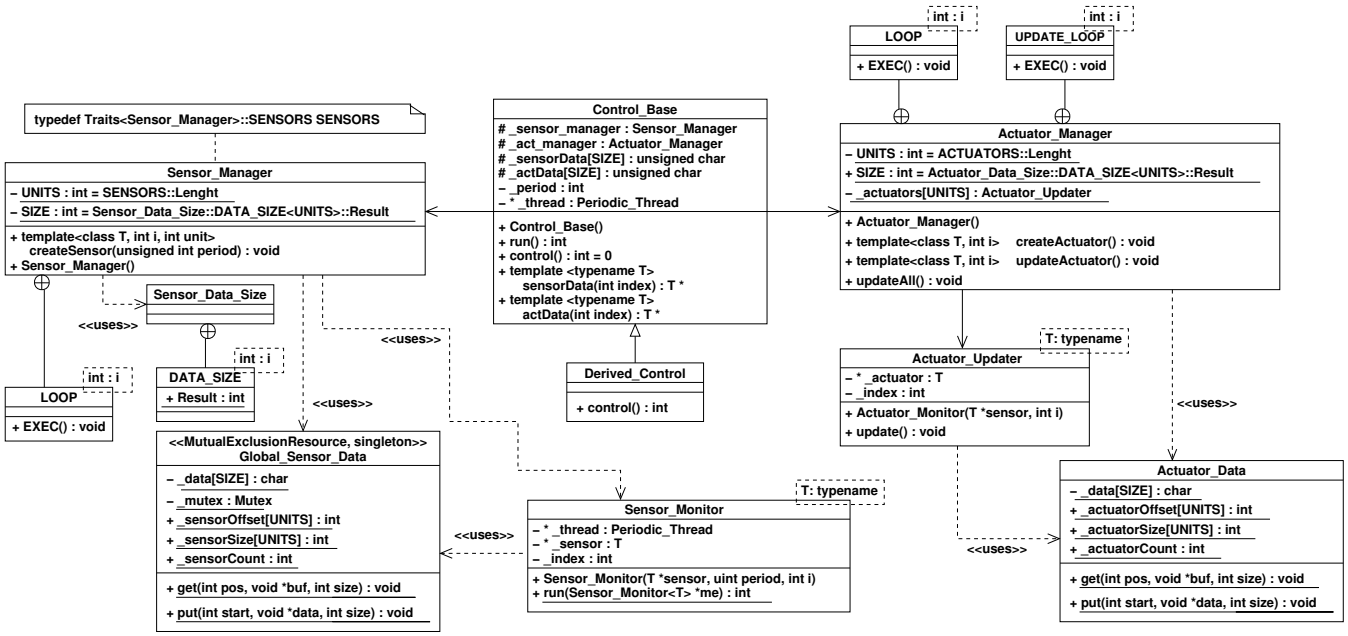

Figura 4. Diagrama de classes UML do framework proposto.

necessário para a aplicação, otimizando o desempenho. Como C++ provê um modelo orientado a objetos e metaprogramação estática, foi a linguagem escolhida para a implementação do framework de controle. Metaprogramação estática em C++ é usada extensivamente (geração de código e classes *traits*) para o desenvolvimento do framework.

III. O Framework Proposto

A Figura 4 apresenta uma visão geral do framework de controle proposto através de um diagrama de classes UML. O framework é compossto por 5 classes principais: `Control_Base`, `Sensor_Manager`, `Sensor_Monitor`, `Actuator_Manager`, e `Actuator_Updater`. Uma descrição detalhada de cada uma dessas classes é realizada abaixo.

A classe `Sensor_Manager` oferece uma abstração para o gerenciamento de sensores. É responsável por criar cada monitor de sensor (*i.e.,* uma thread periódica para ler os valores dos sensores) para cada unidade do sensor requerida pelo algoritmo de controle. A lista dos sensores e os períodos de leitura são definidos pelo desenvolvedor em tempo de compilação em uma classe *trait* (como exemplificado na Seção II-B). A Figura 5 mostra um exemplo da classe *trait* do `Sensor_Monitor` com dois sensores (`Sensor1` e `Sensor2`). Os sensores requeridos são inseridos em uma lista metaprogramada de tipos (linha 2)[2]. O período de cada monitor de sensor é inserido em uma estrutura estática na mesma ordem em que aparece na lista de sensores (linha 5). Por exemplo, o período de 10000 µs é definido como o período para o `Sensor1`. Note que a lista de tipos também é metaprogramada, sendo processada pelo compilador em tempo de compilação (*i.e.,* geração de código e *loop unrolling*). Assim, não há sobrecusto adicional em tempo de execução em qualquer operação da lista.

[2]Uma lista metaprogramada que gerencia tipos de classes ao invés de objetos em tempo de compilação. Veja [5] para uma revisão completa de listas metaprogramadas.

```
1  template <> struct Traits<Sensor_Manager> : public Traits <void> {
2    typedef LIST<Sensor1, Sensor2> SENSORS; // define os sensores
3    static const unsigned int* SENSORS_PERIODS(void) {
4        // define os períodos em microsegundos
5        static const unsigned int ret[SENSORS::Length] = {10000, 15000};
6        return ret;
7    }
8  }
```

Figura 5. Classe *trait* do componente `Sensor_Manager`.

A lista de tipos definida na classe *trait* é usada pelo `Sensor_Manager` (`typedef Traits<Sensor_Manager>::SENSORS SENSORS`) para iterar sobre a lista, criando um monitor para cada sensor. A classe interna `LOOP` é uma classe metaprogramada usada para iterar na lista de tipos. A Figura 6 mostra o método *EXEC* da classe *LOOP*. O construtor do `Sensor_Manager` chama `EXEC` (linha 15) passando o número total de sensores (`UNITS` - linha 4) como parâmetro. Após, o método `createSensor` é chamado para criar um `Sensor_Monitor` para cada sensor (linha 10). No final, o método *EXEC* é chamado recursivamente até o número de sensores alcançar 0 (linha 11). Uma especialização de *template* finaliza a recursão em *EXEC* (linha 19). Novamente, como *EXEC* é um metaprograma, todo código, incluindo as operações na lista, é gerado em tempo de compilação sem sobrecusto em tempo de execução.

A classe `Sensor_Data_Size` (Figura 4) é usada pelo `Sensor_Monitor` para calcular o tamanho total dos dados de todos os sensores definidos em tempo de compilação. Cada sensor deve definir seu tamanho total de dados (atributo estático). Assim, é possível calcular o tamanho dos dados requeridos por todos os sensores e gerar uma estrutura de dados para armazenamento desses dados. Todas as classes dos sensores devem ter a mesma interface para leitura dos valores (método *sample*). A classe `Global_Sensor_Data` é responsável pela leitura e escrita dos dados dos sensores. O

```
1  class Sensor_Manager {
2       ....
3       typedef Traits <Sensor_Manager>::SENSORS SENSORS;
4       static const unsigned int UNITS = SENSORS::Length;
5       ....
6       template <int i>
7       class LOOP {
8            static inline void EXEC() {
9                 ...
10                Sensor_Manager::createSensor<typename SENSORS::Get<i-1>::
                        Result, i-1,unit>(Traits<Sensor_Manager>::
                        SENSORS_PERIODS()[i-1]);
11                LOOP<i - 1>::EXEC();
12           }
13      }
14      ....
15      Sensor_Manager() { LOOP<UNITS>::EXEC(); }
16      ....
17 };
18 // especialização do LOOP para terminar a recursão quando i é 0
19 template <> inline void Sensor_Manager::LOOP<0>::EXEC() { }
```

Figura 6. Parte do código da classe `Sensor_Manager`.

estereótipo `MutualExclusionResource` do perfil UML-MARTE identifica que todos os métodos devem ser protegidos usando uma primitiva de sincronização (*Mutex*).

A classe `Sensor_Monitor` cria uma thread periódica para cada sensor (recebido como parâmetro de *template T*) usando o período definido no *traits* (Figura 5). Cada thread periódica executa o método *run*, no qual lê dados dos sensores e escreve os valores lidos em `Global_Sensor_Data`. Cada sensor é ciente da sua posição na classe `Global_Sensor_Data` pois ela armazena a posição inicial de cada sensor no vetor *_data* (veja Figura 4). A Figura 7 mostra parte do método construtor da classe `Sensor_Monitor`, no qual cria a thread periódica passando a referência a ele mesmo (*this*) e o período (linha 4). Um inteiro *i* é usado como índice para acessar a posição correta do dado dentro da classe `Global_Sensor_Data`.

```
1  template <typename T>
2  class Sensor_Monitor {
3       Sensor_Monitor(T * sensor, unsigned int period, int i) : _sensor(sensor)
              , _index(i) {
4            _thread = new Periodic_Thread(&run, this, period);
5       }
6       ....
7  };
```

Figura 7. Parte do código da classe `Sensor_Monitor`.

As operações de atuação são suportadas pela classe `Actuator_Manager`. Essa classe é similar à classe `Sensor_Manager`. Todos os atuadores também são definidos em uma lista metaprogramada dentro da classe *trait* (`typedef Traits<Actuator_Manager>::ACTUATORS ACTUATORS`). O construtor do `Actuator_Manager` cria um `Actuator_Updater` para cada atuador, usando a classe metaprogramada interna `LOOP`. Note que, diferentemente do `Sensor_Monitor`, o `Actuator_Updater` não cria uma thread periódica, pois as operações de atuação são iniciadas pelo algoritmo de controle. O método *update* envia a saída para o respectivo atuador. Todos os atuadores devem ter a mesma interface, ou seja, o método *update*. O método `updateAll` da classe `Actuator_Manager` usa a classe interna `UPDATE_LOOP` para chamar o método *update* de cada atuador. Todo código é gerado em tempo de compilação devido aos metaprogramas. A classe `Actuator_Data` centraliza todos os dados de atuação.

`Control_Base` é uma classe base para implementar algoritmos de controle (Figura 4). A classe tem objetos do tipo `Sensor_Manager` e `Actuator_Manager` que são responsáveis por criar todos os monitores dos sensores e os atualizadores dos atuadores automaticamente, como descrito nos parágrafos anteriores. A classe possui um método abstrato, *control*, que deve ser usado para implementar o algoritmo de controle. Basicamente, o desenvolvedor deve criar uma classe que estende `Control_Base` e deve implementar o método *control*. Além disso, a classe `Control_Base` tem métodos para acessar e escrever dados em `Global_Sensor_Data` em cada período de controle. A classe cria uma thread periódica que executa o método *run*, mostrado na Figura 8. Na linha 4, os valores dos sensores são lidos do atributo protegido `_sensorData`. Após, o método *control* da classe derivada é chamado para fazer os cálculos de controle (linha 5). Os dados de saída do controlador são escritos no atributo protegido `_actuatorData`, no qual é usado posteriormente para escrever em `Actuator_Data` (linha 6). Todas operações de atuação são executadas na linha 7. Finalmente, o método *wait_next* da thread periódica é chamado para esperar pelo próximo período (linha 8).

```
1  int Control_Base::run() {
2    while(1) {
3       //pega os valores dos sensores
4       Global_Sensor_Data::get(0, _sensorData, Global_Sensor_Data::SIZE);
5       if (!control()) return 0; //faz os cálculos de controle
6       Actuator_Data::put(0, _actuatorData, Actuator_Data::SIZE); // atualiza
              dados
7       _actuatorManager.updateAll(); // atualiza todos os atuadores
8       Periodic_Thread::wait_next(); // espera pelo próximo período
9    }
10 }
```

Figura 8. Implementação do método *run* da classe `Control_Base`.

Resumo. As atividades necessárias para um desenvolvedor usar o framework proposto são resumidas como segue. A primeira tarefa é criar uma lista de sensores e uma lista de atuadores nas classes *traits*. É assumido que o SO suporta todos os sensores e atuadores requeridos pelo algoritmo de controle. Após, o desenvolvedor deve criar uma classe derivada de `Control_Base` e deve implementar o método *control* (o algoritmo de controle). Os valores de saída do controlador devem ser escritos no atributo `_actuatorData`. O framework automatiza a criação de todos os monitores dos sensores e atuadores e provê uma interface simples para implementar algoritmos de controle, integrando-se aos SOTRs.

IV. AVALIAÇÃO

Para demonstrar a viabilidade do framework, o mesmo foi implementado dentro do *Embedded Parallel Operating System* (EPOS) [10]. EPOS é um SOTR multiplataforma, orientado a objetos e baseado em componentes, escrito em C++. EPOS suporta diversas políticas de escalonamento tempo real, incluindo o *Rate-Monotonic* (RM) e o *Earliest Deadline First* (EDF), e protocolos de inversão de prioridade [11]. O EPOS foi escolhido pois oferece todos os recursos de software necessários para a implementação do framework (*i.e.,* thread periódica, mutex, classes trait, projeto com C++ e tempo real).

Um algoritmo do diferencial eletrônico de um carro de corrida (*eRace*) para controlar a tração traseira de motores foi implementado usando o framework e o EPOS. Através desta implementação, foi possível mensurar o consumo de memória do framework e obter os valores de saída do controle, nos quais são discutidos nas próximas subseções.

A. Descrição do Algoritmo do Diferencial Eletrônico

O algoritmo foi executado em uma plataforma de hardware real, chamada EPOSMote II, na qual possui um microcontrolador ARM7 de 24 Mhz [10]. O escalonador escolhido foi o RM. O carro de corrida eletrônico requer sensores de posição dos pedais do freio e acelerador, bem como sensores do ângulo da direção, de velocidade das rodas e detecção do uso do pedal do freio. Os atuadores são os inversores da tração do motor, luzes de indicação e LEDs de status [12]. Devido à ausência de um hardware completo para implementar o controle, os principais sensores foram emulados: potenciômetros nas portas ADCs foram usados para simular os pedais de freio e acelerador, e um acelerômetro na porta SPI emulou o sensor de ângulo da direção. A atuação do inversor do motor foi emulado pela escrita dos dados de torque dinâmico em um barramento I^2C. No barramento, um microcontrolador ATMega128 foi usado para emular os controladores do motor e a aquisição dos dados enviados. Outros atuadores foram simulados por LEDs conectados aos pinos de GPIO.

A Tabela I mostra o mapeamento dos componentes do software de controle em threads de tempo real. *Control Loop* representa a classe `Derived_Control` da Figura 4. As outras entradas da tabela representam as instâncias do `Sensor_Monitor` (uma para cada sensor). O código da thread periódica foi instrumentado usando um temporizador, obtendo 64 amostras consecutivas do tempo de execução de cada thread, e extraindo o pior tempo de execução (WCET), o tempo médio de execução (AVG) e o desvio padrão (STD) dessas 64 execuções. A Tabela I apresenta os valores obtidos. Os monitores do pedal e freio têm baixa utilização pois o código é simples, somente copiando valores dos registradores para `Global_Sensor_Data`. O sensor da direção possui uma utilização maior que os outros sensores pois ele deve acessar o barramento e ler 6 bytes de dados.

Tabela I. MAPEAMENTO DAS TAREFAS PARA O SOTR.

Tarefa	AVG	WCET	STD	Período	Prioridade
Control Loop	859.1 µs	869 µs	1.246 µs	2 ms	2
Mon. Pedal Acel.	27 µs	27 µs	0 µs	1 ms	1
Mon. Pedal Freio	27 µs	27 µs	0 µs	1 ms	1
Mon. Direção	425 µs	432 µs	2.148 µs	2 ms	2
"Control Loop" dividido em "Cálculos" e "Atualização Atuadores":					
Cálculos	157.3 µs	162 µs	1.207 µs	2 ms	2
Atualização Atuadores	701.8 µs	707 µs	1.286 µs	2 ms	2

O teste de escalonabilidade suficiente do RM pode ser usado para verificar a escalonabilidade das tarefas ($\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$[3]. Para o conjunto de tarefas do experimento, n é 4, totalizando uma utilização máxima de aproximadamente 0.757. A utilização do conjunto de tarefas é 0.7045 (Tabela I). Consequentemente, a condição suficiente é verdadeira e o conjunto de tarefas escalonável segundo o RM. O sistema consegue taxas de amostragem adequadas e tem o comportamento esperado.

[3] C_i é o WCET e P_i é o período da tarefa i. n é o número total de tarefas [2].

B. Avaliação do Consumo de Memória

O consumo de memória dessa implementação do framework foi mensurado usando a ferramenta *GNU objdump*. A Tabela II apresenta os resultados obtidos em bytes. A seção `.text` refere-se à memória consumida pelo código executável, a seção `.rodata` às constantes do programa, `.bss` é a memória ocupada por variáveis estáticas e *heap* por dados alocados dinamicamente. A pilha refere-se aos dados armazenados na pilha do programa durante a execução no nível mais alto da aplicação (*main*). O consumo total de memória do sistema de controle é de 7480 bytes.

Tabela II. CONSUMO DE MEMÓRIA DO FRAMEWORK (EM BYTES).

Classe	.text	.rodata	.bss	heap	pilha
Actuator_Data	-	-	72	-	-
Actuator_Updater	728	88	-	32	-
BUS_Actuator	504	-	-	16	-
Control_Base	2696	-	-	32	44
Derived_Control	1052	4	-	-	-
Global_Sensor_Data	-	-	56	-	-
LED_Actuator	-	-	-	16	-
main (função)	680	-	-	16	-
Sensor_Monitor	1368	-	-	16	-
Traits (somente do framework)	-	44	-	16	-
Total	7028	136	128	144	44

Devido ao uso de metaprogramação estática e código *inlining*, o compilador otimizou algumas partes do código. Por exemplo, as classes `Sensor_Manager` e `Actuator_Manager` não apareceram no código resultante. Os códigos de amostragens dos sensores, encapsulados nas subclasses `Sensor`, foram embutidos nas respectivas especializações `Sensor_Manager`. O mesmo aconteceu com os atuadores em `Actuator_Updater`. Toda a otimização realizada pelo compilador beneficia o desempenho, evitando chamada a funções.

O consumo de memória para a inclusão de novos sensores no sistema depende se o tipo do sensor já está sendo usado ou não. Se o tipo do sensor está sendo usado (existe um monitor para o sensor), o consumo é limitado às estruturas estáticas das classes `Global_Sensor_Data` e `Control_Base` e na alocação de outro `Sensor_Monitor`. O consumo neste caso, em bytes, é de $12 + 4 \cdot SIZE$, onde $SIZE$ é a quantidade de armazenamento requerido pelo tipo do sensor. Se o tipo do sensor é novo no sistema, existe também o consumo de uma nova classe especializada para aquele sensor (`Sensor_Monitor`). Como `Sensor_Monitor` é uma classe parametrizada, o código é gerado para cada tipo recebido como argumento do *template*. O mesmo vale para os atuadores.

C. Saída do Controlador

Para atestar o correto funcionamento do controlador, foi projetado um cenário simples: o torque do carro é cada vez mais solicitado em uma estrada reta até atingir um valor desejado e então uma curva para a esquerda é efetuada. Para isso, foi criado um cenário simulado, onde os valores extraídos dos sensores do hardware são substituídos. Além disso, no final do laço de controle, os valores são escritos em uma porta serial. A saída serial gera um atraso no ciclo de controle, sendo necessária uma mudança no tempo do controle. As threads

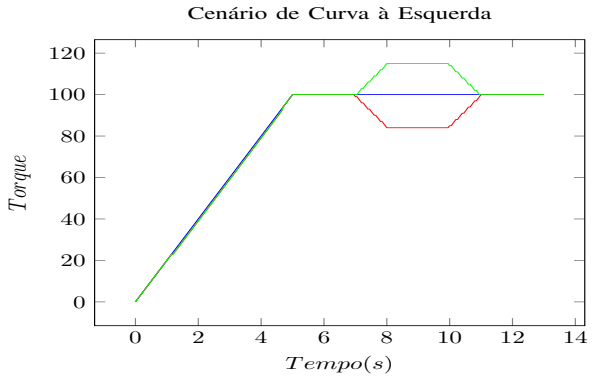

Figura 9. Saída do Algoritmo de Controle.

`Sensor_Monitor` foram executadas normalmente, mesmo quando seus dados não fossem utilizados, para não reduzir a utilização do sistema.

A Figura 9 apresenta a saída correspondente ao experimento detalhado acima. A linha em azul representa o torque requisitado pelo pedal, a linha em verde representa o torque desejado entregue à roda de tração direita e a linha em vermelho representa o torque desejado pela roda de tração esquerda. Como esperado em uma curva para a esquerda, o torque da roda esquerda é reduzido, enquanto o torque da roda direita aumenta. Neste cenário simulado, verificou-se o correto funcionamento da arquitetura de software proposta.

V. Trabalhos Relacionados

Ferramentas do Matlab/Simulink permitem a simulação e geração do código diretamente de um modelo de controle [13]. Broy *et al.* discute diferentes aspectos do fluxo de projeto de controladores multicamadas e baseados em modelos, envolvendo geração de código e configuração [14]. A ênfase do trabalho foi no fluxo de configuração da plataforma e arquitetura alvo, sendo que um modelo arquitetural não foi apresentado e a implementação de software não foi discutida.

Feedback Control real-time Scheduling (FCS) é um framework analítico que mapeia a qualidade de serviço do controle em sistemas de tempo real adaptativos à teoria de controle realimentado [15]. O framework compreende em uma arquitetura de escalonamento, especificações de desempenho e uma metodologia de projeto. O principal objetivo do framework é projetar algoritmos FCS que satisfazem as especificações dos sistemas de tempo real [15]. O framework proposto neste trabalho, diferentemente, foca na arquitetura de software dentro do SOTR para facilitar a implementação dos algoritmos de controle.

Frameworks similares para automatizar a geração de componentes do SO têm sido propostos recentemente. Por exemplo, um framework para geração de protocolos de controle de acesso ao meio (MAC) [16] e para a geração de protocolos de disseminação de dados em redes de sensores sem fio [17].

VI. Conclusão

Este artigo apresentou um framework de SO para automatizar a geração de algoritmos de controle digital. Através do uso de técnicas sofisticadas de engenharia de software, o framework automaticamente gera threads periódicas para ler dados dos sensores e também provê uma interface para facilmente implementar o laço de controle (e atuação). O framework foi avaliado em termos de consumo de memória através da implementação do controle do diferencial eletrônico de um carro, usando SOTR e hardware reais. O framework juntamente com o algoritmo de controle consumiu menos de 8 KB de memória, o que confirma sua usabilidade em plataformas de sistemas de controle embarcados com recursos restritos. O framework foi implementado em C++, podendo ser integrado em qualquer SOTR com suporte à linguagem. Como trabalho futuro, os sensores emulados serão substituídos por sensores reais e será estudado como plataformas multicore podem substituir pequenos microcontroladores e ainda assim prover as garantias de tempo real para algoritmos de controle. Outra possibilidade é a integração do sistema a ferramentas de modelagem de alto nível, como a SysML.

Referências

[1] K. J. Aström and B. Wittenmark, *Computer-controlled Systems: Theory and Design (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[3] J. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[4] G. Booch, *Object-Oriented Analysis and Design with Applications*. USA: Addison Wesley Co., 2004.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming." in *ECOOP'97*. Springer, 1997, pp. 220–242.

[6] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design & Test*, vol. 18, pp. 23–33, November 2001.

[7] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[8] D. Parnas, "On the design and development of program families," *IEEE Trans. on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, Mar 1976.

[9] A. A. Fröhlich and W. Schröder-Preikschat, "Scenario Adapters: Efficiently Adapting Components," in *4th WMSCI*, Orlando, Jul 2000.

[10] EPOS. (2014, May) Website. [Online]. Available: http://epos.lisha.ufsc.br

[11] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, 2013.

[12] A. Draou, "Electronic differential speed control for two in-wheels motors drive vehicle," in *Proceedings of POWERENG*, May 2013, pp. 764–769.

[13] MathWorks. (2014, Aug) MATLAB and simulink for technical computing. [Online]. Available: http://www.mathworks.com/

[14] M. Broy, S. Chakraborty, S. Ramesh, M. Satpathy, S. Resmerita, and W. Pree, "Cross-layer analysis, testing and verification of automotive control software," in *Proc. of the EMSOFT*, Oct 2011, pp. 263–272.

[15] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms*," *Real-Time Systems*, vol. 23, no. 1/2, pp. 85–126, Jul. 2002.

[16] L. Wanner, A. de Oliveira, and A. A. Fröhlich, "Configurable medium access control for wireless sensor networks," in *Proceeding of IESS*, vol. 231. Springer US, 2007, pp. 401–410.

[17] R. Steiner, G. Gracioli, R. de Cassia Cazu Soldi, and A. Frohlich, "An operating system runtime reprogramming infrastructure for wsn," in *Proceedings of ISCC*, July 2012, pp. 000 621–000 624.