Exploração do Espaço de Projeto e Síntese de Software Embarcado a partir de Modelos Alloy

Ronaldo Ferreira[1], Emilena Specht[1], Lisane Brisolara[2], Julio Mattos[2], Érika Cota[1] e Luigi Carro[1]

[1]Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil
[2]Departamento de Informática, Universidade Federal de Pelotas, Pelotas, Brasil
{rrferreira, emilenas, erika, carro}@inf.ufrgs.br, {lisane.brisolara, julius}@ufpel.edu.br

Abstract. Embedded systems have become very complex, mainly due to the amount of functions performed in software, despite the very limited hardware resources. On this domain, design automation tools must meet the demand for productivity and maintainability in embedded systems development, but constraints such as memory, power and performance must still be considered. In this work we introduce a new approach to automate embedded software development, which synthesizes Java source code from its Alloy model. The application model is formally verified, increasing overall quality on the final product. The source code generation from Alloy models, combined with an estimation tool, provides design space exploration to match tight embedded software design constraints, what is usually not taken into account by standard software engineering methods.

Palavras-chave: Alloy, Automação de Software, Exploração do Espaço de Projeto, Geração de Código, Java, Sistemas Embarcados.

1 Introdução

Sistemas embarcados (SE) desempenham um papel importante no dia-a-dia das pessoas. Os telefones portáteis, por exemplo, são sistemas embarcados que oferecem uma grande gama de aplicações além de telefonia: câmera digital, agendas, jogos, acesso à Internet, tocadores de arquivos MP3, entre outros. Segundo Graff [1], o domínio de SE é guiado por fatores como confiabilidade, custo e tempo de projeto. Estatísticas apontam que mais da metade dos projetos atuais de SE estão atrasados principalmente devido ao reduzido tempo de projeto e à crescente complexidade destes sistemas [2].

O software embarcado executa em dispositivos eletrônicos normalmente de recursos limitados e desenvolvidos para uma aplicação específica. Soluções em software têm sido preferidas a soluções em hardware, pois o software é mais flexível, mais fácil de atualizar e de reusar [1]. Adicionalmente, estatísticas formuladas pela indústria [3] revelam que no software embarcado a quantidade de linhas de código, métrica considerada como um indicativo da complexidade do software [4], cresce na

faixa de 26% ao ano. Por esta razão, nas aplicações atuais o software é responsável pela maior parte no gargalo de produtividade. Neste contexto, projetistas buscam novas metodologias e ferramentas para aumentar a produtividade e qualidade no projeto de software embarcado.

O uso de técnicas de modelagem e projeto baseadas em níveis de abstração mais altos é apontado como crucial para lidar com a complexidade encontrada na nova geração de SE [5]. Com abstração gerencia-se a complexidade, porém é preciso também automatização para aumentar a produtividade dos projetos. Na área de engenharia de software, ferramentas CASE (*Computer-Aided Software Engineering*) são utilizadas para automatizar o processo de desenvolvimento, gerando código a partir de modelos de alto nível.

Além das restrições físicas tradicionais, sistemas embarcados são freqüentemente sistemas de segurança crítica, onde aspectos de confiabilidade e segurança são mais importantes do que o desempenho [6]. Esse fato aponta a necessidade de um suporte cuidadoso à verificação e validação (V&V) desde a análise de requisitos até o teste do software desenvolvido. A qualidade do software é um requisito prioritário no domínio de SE [7]. Como técnicas de V&V são dispendiosas e podem representar 80% dos custos totais do software em sistemas críticos [6], deve-se estabelecer um compromisso entre qualidade, custo e tempo de desenvolvimento. Técnicas formais são indicadas para a verificação de sistemas críticos porque permitem uma melhor cobertura das falhas. Ferramentas que oferecem verificação formal ao projeto e não implicam no aumento adicional do tempo de projeto representam uma contribuição valiosa, embora atualmente elas ainda se encontrem distantes dos projetos reais.

O objetivo deste trabalho é aumentar o grau de automação e verificação no desenvolvimento de software embarcado, introduzindo uma nova abordagem para a geração automática de software aliada à exploração do espaço de projeto. A solução proposta baseia-se no uso de Alloy [8], uma linguagem declarativa dotada de ferramentas de verificação formal de modelos, para a modelagem das aplicações. Com isto, a abordagem garante abstração e correção das propriedades especificadas da aplicação, facilitando a verificação da qualidade final do produto, além de prover um fluxo de projeto automatizado. Esta abordagem pode ser inserida em um modelo de processo de software, reduzindo o esforço e aumentando a qualidade do software produzido.

2 Abordagem Proposta

Este trabalho propõe o uso do Alloy como linguagem de especificação de alto nível, e a geração automática de código Java otimizado para a plataforma de hardware a partir de modelos descritos em Alloy.

2.1 Fluxo de Projeto

A Figura 1 descreve o fluxo de projeto proposto para a abordagem de automação do desenvolvimento de software embarcado. A partir da descrição dos requisitos, o

modelo da aplicação é descrito em Alloy, o qual captura a especificação estrutural e comportamental do sistema. Utilizando o Alloy Analyzer o projetista pode verificar se as propriedades descritas pelos requisitos da aplicação foram corretamente especificadas no modelo. A ferramenta de tradução de modelos tem como entrada um modelo Alloy já verificado e aceito como correto, a partir do qual é gerado o código Java que implementa o modelo formal. Diferentes implementações Java são geradas para um mesmo modelo Alloy, variando-se a estrutura de dados. Para todas estas soluções, estimam-se propriedades tais como desempenho, energia e tamanho de memória requerida pelo programa quando executando na plataforma alvo. Com base nos requisitos não-funcionais e nas estimativas obtidas, o projetista determina qual a melhor solução. O resultado do fluxo é a solução escolhida mapeada para a plataforma alvo. Este trabalho adota a plataforma FemtoJava [9], que será detalhada na seção 3.1.

2.2 Linguagem Alloy

Modelos Alloy expressam tanto a estrutura quanto o comportamento das aplicações. A construção estrutural em Alloy é a *Signature*, que representa um conjunto de átomos. As *Signatures* podem também introduzir campos, os quais representam relações entre os átomos. Uma *Signature*, em linhas gerais, é uma estrutura de dados do tipo conjunto. A linguagem provê operadores para conjuntos, relações, bem como herança entre *Signatures*. Por ser baseada na solução de restrições, Alloy possui construções para representá-las, sendo elas os fatos, predicados e funções. Fatos são restrições que sempre são avaliadas como verdadeiras. Predicados e funções promovem o reuso de restrições, sendo utilizados para fins de simulação e válidos respeitando-se determinadas condições. Já as asserções são implicações a serem verificadas contra o modelo.

Uma importante classe de aplicações de software embarcado é a de sistemas reativos, a qual é composta por uma máquina de estados que, ao receber um evento, executa uma determinada ação. O elemento básico do Alloy é o átomo, o qual é um elemento de uma *Signature*. Como o átomo é imutável (o valor atribuído a um átomo não pode ser alterado), a mudança de estados em Alloy é modelada por um conjunto deles, cada um representando um tempo distinto. A máquina reativa modelada é uma Máquina de Venda de Bebidas que possui os seguintes requisitos funcionais (Figura 2): i) há três tipos de bebidas: água, chá e refrigerante; ii) cada bebida possui um preço; iii) a máquina funciona através da inserção de fichas; iv) uma venda só se completa quando a quantidade necessária de fichas para um dado tipo de bebida foi inserida; v) a máquina não vende bebidas que não estão mais disponíveis no reservatório de bebidas.

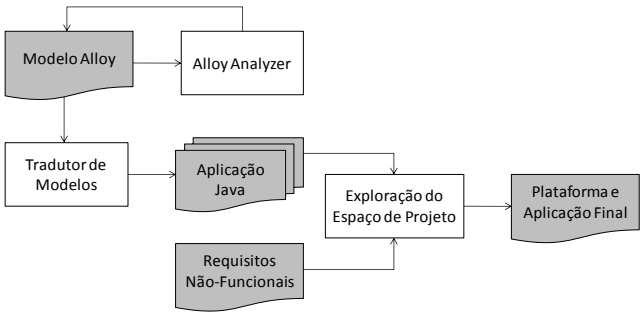

Figura 1. Fluxo de projeto da abordagem proposta

```
open util/ordering[State] as ord
abstract sig Drink { price: Int }
sig Water extends Drink {}
sig Softdrink extends Drink {}
sig Tea extends Drink {}
sig Coin {}
sig VendingMachine{
   coinStorage:   set Coin,
   drinkStorage:  set Drink
}
sig State { machine: VendingMachine }
fact InitialState {
   let s0 = ord/first.machine | no s0.coinStorage
   let s0 = ord/first.machine | all d:Drink | d in s0.drinkStorage
   some Tea and some Water and some Softdrink
}
fact DrinkBuying {
  all s: State, s': ord/next[s] | some d: Drink | some c: Coin |
    d.price<=#s.machine.coinStorage=>
           buyDrink[s.machine,s'.machine,d]
    else addCoin[s.machine,s'.machine,c]
}
fact Prices {
   all w: Water      | w.price = 1
   all s: Softdrink  | s.price = 2
   all t: Tea        | t.price = 3
}
fact MaximunCoinAmount { #Coin = (sum d : Drink | d.price) }
fact AlwaysDoSomething{ all s:State,s': s.ord/next |
      s.machine.coinStorage != s'.machine.coinStorage }
fact AllMachineInSomeState{ all m:VendingMachine |
      some s: State | m in s.machine }
pred buyDrink[m, m': VendingMachine, d: Drink] {
       m'.drinkStorage =  m.drinkStorage - d
      #m'.coinStorage  = #m.coinStorage  - d.price
}
pred addCoin[m, m': VendingMachine, c: Coin] {
       m'.coinStorage  = m.coinStorage + c
       m'.drinkStorage = m.drinkStorage
}
run { no ord/last.machine.drinkStorage and
      no ord/last.machine.coinStorage } for 14
run addCoin
run b
```

Figura 2. Modelo Alloy para a Máquina de Vendas de Bebidas

2.3 Geração de Código

Uma ferramenta de tradução de modelos Alloy em código Java foi desenvolvida. Nesta seção os algoritmos de tradução são apresentados e o processo de tradução é demonstrado através do modelo da Máquina de Vendas de Bebidas.

2.3.1 Geração de Classes e Atributos

As classes e os atributos são extraídos das *Signatures* e das relações declaradas nos modelos, respectivamente, e o algoritmo para geração estrutural é apresentado abaixo.

Passo 1: Para cada *Signature S* extraída do modelo Alloy é criada uma classe Java *C*, exceto se essa *Signature S* não possui campos.

Passo 2: Para cada relação *R* de *S* é inserido um atributo em *C*. Se *R* for unária e tiver como domínio uma *Signature* que não é vazia, cria-se uma coleção como atributo na classe *C*; se o domínio for vazio, cria-se um atributo do tipo *String*. Se *R* for n-ária, cria-se uma nova classe Java *C'* na qual as colunas de *R* são atributos de *C'*, usando as regras descritas no *Passo 2*.

A Figura 3 apresenta a parte estrutural da classe Java *VendingMachine*, a qual foi gerada a partir da *Signature* de mesmo nome. Essa classe, devido aos campos de multiplicidade *set* contém duas coleções. Para essas coleções, diferentes estruturas de dados podem ser usadas. A Figura 3 apresenta um exemplo onde a estrutura *FastList* foi escolhida para as duas coleções existentes (linhas 2 e 3).

```
1.  public class VendingMachine {
2.     protected FastList<Drink> drinkStorage;
3.     protected FastList<String> coinStorage;
4.     public VendingMachine() {
5.        this.drinkStorage = new FastList<Drink>();
6.        this.coinStorage   = new FastList<String>();
```

Figura 3. Porção estrutural da classe Java VendingMachine gerada

2.3.2 Geração de Métodos

Os métodos são extraídos dos predicados e das funções declaradas no modelo. O algoritmo para a geração de métodos Java é apresentado a seguir.

Passo 1: Para cada predicado *P* e para cada função *F* os quais possuem dois parâmetros formais do tipo *C* e que estendem a biblioteca Alloy *Ordering* vá para o *Passo 2*, senão vá para o *Passo 3*.

Passo 2: Cria-se um método *M* na classe Java *C* com o nome de *P* ou *F*. Se for predicado, o tipo de retorno de *M* é *void*; se for função, o tipo de retorno de *M* é o mesmo de *F*. A visibilidade de acesso de *M* é *public.*

Passo 3: Cria-se um método *M* na classe Java *Main* com o nome de *P* ou *F*. Se for predicado, o tipo de retorno de *M* é *void*; se for função, o tipo de retorno de *M* é o mesmo de *F*. A visibilidade de acesso de *M* é *public static.*

A inferência da classe a qual um método pertence é realizada através da lista de parâmetros formais dos predicados/funções do modelo Alloy. Como supracitado, a

modelagem de estados é realizada através de um conjunto de estados, onde cada estado representa uma configuração. As mudanças nas configurações são realizadas através dos predicados e funções. A lista de parâmetros formais dos predicados/funções que realizam essas alterações contém dois parâmetros da *Signature* que encapsulam os dados de um estado – um encapsula a configuração atual, outro a configuração do próximo estado. Se ocorrer essa construção, a ferramenta de tradução insere o método na classe gerada a partir da *Signature* que encapsula as configurações do estado. Caso contrário, o predicado/função é inserido como método estático da classe *Main.*

A Figura 4 apresenta os métodos gerados a partir dos predicados declarados no modelo Alloy (Figura 2). O processo de tradução trata o caso de polimorfismo paramétrico de maneira bastante simples: geram-se todas as combinações possíveis dos tipos polimórficos (linhas 6, 9 e 12 – Figura 4). Isso pode ser inadequado ao se considerar boas práticas de Engenharia de Software, mas é necessário para não tornar ambígua a escolha entre os estados possíveis da máquina que será sintetizada. Isto não interfere no reuso e na qualidade do processo, pois toda a abordagem é aplicada no modelo Alloy de alto nível, e não no código orientado a objetos gerado.

```
1.public class VendingMachine {
3.  public void addCoin(String c) {
4.      AlloyOperations.alloyUnion(this.coinStorage,c); }
6.  public void buyDrink_Water(Water d) {
7.         AlloyOperations.alloySubtraction(this.drinkStorage, new
           Water(d)); }
9.  public void buyDrink_Softdrink(Softdrink d) {
10.        AlloyOperations.alloySubtraction(this.drinkStorage, new
           Softdrink(d)); }
12. public void buyDrink_Tea(Tea d) {
13.        AlloyOperations.alloySubtraction(this.drinkStorage, new
           Tea(d)); }
```

Figura 4. Porção comportamental da aplicação gerada a partir dos predicados do modelo

2.3.3 Inferência e Síntese da Máquina de Estados

A máquina de estados da aplicação é sintetizada automaticamente a partir do modelo Alloy, assumindo o modelo reativo de computação. Para a geração da máquina de estados, o projetista indica apenas as operações que devem fazer parte da mesma. Isso é realizado através do comando Alloy *run*.

Passo 1: Cria-se uma estrutura *X* para a máquina de estados. Para cada comando *run E* do modelo Alloy, se *E* chama somente um predicado *P* ou função *F*, vá para o *Passo 2*.

Passo 2: Para cada *Signature S* que estende a biblioteca *Ordering* do Alloy, instancia um *s* do tipo *S* na classe Java *Main*. Se *P* ou *F* pertencerem à *S*, vá para o *Passo 3*. Senão, vá para o *Passo 4*.

Passo 3: Para cada parâmetro formal *A* de *P* ou *F* que estende uma classe Java gerada, cria-se uma opção em *X* para cada classe estendida por *A*.

Passo 4: Cria-se uma opção em *X* para *P* ou *F*.

Somente os comandos *run* que invocam predicados ou funções são utilizados no processo de geração da máquina de estados.

Uma porção da máquina de estados reativa sintetizada a partir do modelo da Máquina de Vendas de Bebidas é apresentada na Figura 5. A linguagem Alloy não oferece suporte a operações de entrada e saída (E/S). No entanto, é necessário tratá-las e gerá-las no código Java, para que a máquina reaja aos eventos. Para tal, são inseridas as construções de E/S em Java antes da chamada dos métodos que capturam dados (linhas 9 a 11 – Figura 5). A enumeração Java *menu_enum* é criada para armazenar o estado atual da máquina de estados. Após o término de qualquer operação, a máquina desvia para o estado *done* (linha 13 – Figura 5), onde uma nova ação pode ser tomada.

```
1.  public class Main {
2.   public static void main(String[] args) {
3.    VendingMachine machine = new VendingMachine();
4.    Menu menu_enum = Menu.done;
5.   InputStreamReader stdin = new
          InputStreamReader (System.in);
6.     BufferedReader console = new BufferedReader(stdin);
7.    try { while(true){
8.       if(menu_enum.ordinal() == Menu.addcoin.ordinal()) {
9.       try { System.out.println("Enter Coin:");
10.         String c = console.readLine();
11.         machine.addCoin(c); }
12.        catch (IOException e) { e.printStackTrace(); }
13.           finally { menu_enum = Menu.done; } }
```

Figura 5. Porção do código Java da máquina de estados sintetizada

3 Resultados Experimentais

Os resultados da geração e da execução do código Java são apresentados, além da plataforma alvo utilizada. A Máquina de Vendas de Bebidas contém porções *control* e *data-flow*. Um bloco de instruções é dito *control-flow* quando apresenta uma alta relação de saltos por instrução; caso contrário, o bloco é denominado *data-flow*. O processo de geração fornece diferentes códigos Java, os quais diferem quanto às estruturas de dados da Javolution [10] escolhidas: *FastList, FastMap e FastSet*. Para a Máquina de Vendas foram geradas cinco soluções distintas. Como optou-se por utilizar Javolution ao invés de bibliotecas Java padrão para estruturas de dados (como *Set* e *List*), o número de soluções diferentes geradas é $O(4^k)$, onde k é a quantidade de relações com multiplicidade *set* declarada no modelo Alloy.

3.1 Plataforma Alvo

A plataforma alvo considerada baseia-se em diferentes implementações de um processador Java chamado FemtoJava [9], o qual implementa uma máquina de execução Java em hardware através de uma máquina de pilha compatível com a

especificação da JVM (*Java Virtual Machine*). Neste trabalho foram utilizadas a versão multiciclo [9] e uma versão pipeline do processador FemtoJava desenvolvidas especificamente para o mercado de sistemas embarcados. A versão multiciclo é adequada para aplicações embarcadas que requerem uma pequena área e um pequeno desempenho, enquanto a versão pipeline fornece uma melhoria em termos de desempenho adicionando custos em área e potência. Ressalta-se que a escolha da plataforma FemtoJava foi devida à possibilidade de instrumentar o código gerado. No entanto, o fluxo proposto pode ser adaptado para utilizar J2ME ou outra plataforma.

Os resultados apresentados foram estimados através da ferramenta DESEJOS [11]. A calibração da ferramenta para desempenho foi realizada através de um simulador ciclo a ciclo dos processadores [12] e para consumo de energia através da ferramenta *Synopsis Power Compiler* a partir da síntese das descrições VHDL dos processadores.

3.2 Resultados

A Tabela 1 apresenta as diferentes soluções da Máquina de Vendas. As soluções 4 e 5 apresentam significativa diferença em termos de desempenho e energia consumida em relação às soluções 1, 2 e 3 (4 vezes mais eficiente). Isto é devido ao fato das soluções 1, 2 e 3 possuírem no seu código a manipulação de estruturas do tipo *FastList* e as soluções 4 e 5 manipularem apenas estruturas do tipo *FastMap* e *FastSet*. Em relação à potência, o processador pipeline dissipa mais potência que o processador multiciclo, devido principalmente ao tamanho da sua área. Entretanto, em termos de desempenho, a versão pipeline é em torno de 50% mais rápida. Comparado com aplicações que possuem muitas operações matemáticas este ganho é baixo (na ordem de 5 vezes), porém isto deve-se ao fato da quantidade de *bytecodes* que manipulam objetos ser alta, prejudicando o desempenho do processador.

Tabela 1. Resultados de desempenho, energia e potência da Máquina de Vendas. A primeira linha na tabela representa a solução 1, e assim sucessivamente.

Multiciclo			**Pipeline**		
Desempenho (ciclos)	**Energia Consumida (Joules)**	**Potência Média (mWatts)**	**Desempenho (ciclos)**	**Energia Consumida (Joules)**	**Potência Média (mWatts)**
33.466.145	0,1344	20,0855	18.259.424	0,1412	38,6782
34.575.240	0,1394	20,1674	18.902.957	0,1464	38,7503
34.512.434	0,1391	20,1578	18.870.747	0,1461	38,7239
8.314.412	0,0388	23,3420	5.025.923	0,0405	40,2973
8.251.699	0,0384	23,3265	4.993.165	0,0401	40,2140

4 Trabalhos Relacionados

Há diversas ferramentas, acadêmicas e comerciais, para automação de software baseadas em UML que extraem código a partir dos diagramas, *e.g.* Rational Rose,

UniMod e Rhapsody. Algumas delas são capazes somente de gerar o código estrutural a partir dos diagramas de classe. Uma porção bastante restrita de ferramentas acadêmicas gera o código comportamental a partir de uma combinação de diagramas UML, tal como a Rhapsody. Essas ferramentas provêem alta abstração para o projetista, mas nenhuma delas oferece verificação formal dos modelos e exploração do espaço de projeto da aplicação modelada.

Para modelagem de sistemas há abordagens como POLIS [13], Metropolis [14] e ForSyDe [15]. POLIS provê geração de código baseada no modelo de Máquina de Estados Concorrentes Finitos (CFSM). A especificação formal de POLIS permite a interface direta com algoritmos já existentes de verificação formal baseados em CFSM. Entretanto, o conjunto de aplicações gerado e o escalonador compõem um conjunto estático, logo, um comportamento dinâmico não é possível. Além disso, POLIS não permite a especificação de estruturas de dados e não oferece exploração do espaço de projeto. Metropolis [14] foi proposta como uma extensão de POLIS. A linguagem de especificação de Metropolis, a MetaModel, é baseada em redes de processos e destina-se à especificação em nível de sistema. Metropolis oferece duas técnicas de verificação: verificação formal usando o verificador de modelos SPIN e a simulação de traços de execução [16]. Porém, técnicas de verificação baseadas no SPIN geralmente representam um acréscimo no tempo de projeto, o que não é viável no domínio de aplicações embarcadas. Na abordagem aqui proposta, usa-se a sintaxe *lightweight* do Alloy para se ter qualidade dentro das limitações de tempo de projeto.

O ForSyDe [15] é um *framework* desenvolvido para modelagem e síntese de sistemas embarcados. As especificações neste *framework* são modeladas em Haskell, que oferece abstração e semântica formal para o modelo. A abstração proporcionada pela linguagem Haskell é similar à proporcionada por OCaml, sendo próxima a de linguagens de programação padrão.

Trabalhos propuseram o uso de Alloy para verificação formal de programas ou para especificação de sistemas. [17] propõe a tradução de diagramas de classe UML marcados com restrições OCL para Alloy, visando suportar a verificação desse diagrama. Como diagramas comportamentais não são traduzidos, essa abordagem verifica somente a estrutura. Outra abordagem usa a *Alloy Annotation Language* para anotar o código Java e dele gerar um modelo Alloy que pode ser verificado pelo *Alloy Analyzer*, o que permite análise estática do modelo [18], mas nenhum trabalho utiliza Alloy para exploração do espaço de projeto de software embarcado.

5 Conclusões e Trabalhos Futuros

Este artigo propõe uma abordagem para o desenvolvimento de software embarcado, onde a especificação em alto nível é descrita em Alloy, o que provê abstração e verificação formal. Após a verificação do modelo, gera-se automaticamente código Java que implementa a aplicação modelada. A ferramenta de tradução de código gera diferentes códigos Java, variando a estrutura de dados. Para cada solução gerada estimam-se as propriedades físicas, de forma que a solução mais adequada aos requisitos não-funcionais da aplicação possa ser determinada. Neste trabalho também

foi apresentada a integração de uma ferramenta de exploração do espaço de projeto ao fluxo de projeto do software embarcado, o que é essencial para suportar a avaliação de soluções e a definição de soluções que respeitem os requisitos relativos às restrições físicas presentes em sistemas embarcados.

Referências

1 Graaf, B., Lormans, M. and Toetenel, H. Embedded Software Engineering: The State of the Practice. IEEE Software, November/December, p. 61-69. (2003).
2. CMP Media. 2006 State of Embedded Market Survey. April. (2006).
3. VDC – Venture Development Corporation VIII: Embedded Systems market Statistics. The 2005 Embed. Soft. Strat. Market Intelligence. Prog. USA. (2006).
4. Kan, S. H. Metrics and Models in Software Quality Engineering. Addison-Wesley. (2002).
5. Selic, Bran. Models, Software Models and UML. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers. Chapter. 1, p. 1-16. (2003).
6. Koopman, P. Reliability, Safety, and Security in Everyday Embedded Systems. A. Bondavalli, F. Brasileiro, and S. Rajsbaum (Eds.): Dependable Computing. LNCS 4746, p. 1–2, Springer-Verlag Berlin Heidelberg (2007).
7. Woodward, Michael V; Mosterman, Pieter J. Challenges for embedded software development. Proc. of the NEWCAS 2007, Agosto 5–8, p. 630-633. (2007).
8. Jackson, D. Software Abstractions – Logic, Language and Analysis. The MIT Press, Cambridge, MA, EUA. (2006).
9. Ito, S. A., Carro, L. and Jacobi, R. P Making Java work for microcontroller applications. IEEE Design & Test of Computers, Set-Out, 18 (5), pp. 100-110. (2001).
10. Dautelle, J. Fully Deterministic Java. AIAA SPACE 2007 Conference and Exposition, Long Beach, California, Setembro. 18-20. (2007).
11. Mattos, Júlio C. B., Carro, Luigi. Object and Method Exploration for Embedded Systems Applications. Proc. of the SBCCI 2007, Set. New York: ACM Press. p.318-323. (2007).
12. Beck, A.C.S., Mattos, J. C. B., Wagner, F.R., Carro, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. Proc. of the SBCCI 2003, Set. IEEE Computer Society Press , p.349-359. (2003).
13. Balarin, F. et. al. Synthesis of Software Programs for Embedded Control Applications. IEEE, 18 (6), p. 834-849. (1999).
14. Sangiovanni-Vincentelli, A. L. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design, IEEE, 95 (3), p. 467-506. (2007).
15. Sander, I. and Jantsch, A. System modeling and transformational design refinement in ForSyDe. IEEE, 23 (1), p. 17-32. (2004).
16. Chen, X., Hsieh, H., Balarin, F. Verification Approach of Metropolis Design Framework for Embedded Systems. Int. Journal of Parallel Prog. 34 (1) p. 3-27. (2006).
17. Massoni, T., Gheyi, R., Borba, P. A UML class diagram analyzer. 3rd Workshop on Critical Systems Development with UML, p. 100-114. (2004).
18. Khurshid, S., Marinov, D., Jackson, D. An Analyzable Annotation Language. ACM SIGPLAN Notices, 37 (11), p. 231 – 245. (2002).