Verificação de modelos com *streams* de dados sobre conectores Reo

Mateus A. Torres[1], **Bruno Lopes**[1]

[1]Instituto de Computação
Universidade Federal Fluminense
Niterói, Brazil

mateustorres@id.uff.br, bruno@ic.uff.br

Abstract. *System modeling and certification, crucial for critical systems, are high complexity tasks. Using tools for exempt such complexity from developers may grant increased productivity on software construction and validation. Reo is a graphical language focused on coordination, promising to aid modeling of system components interaction. Time Data Streams denote, formally, the data flow in the aforementioned language, allowing model checking against a system execution, performed on this work by the tool nuXmv. Thus, this work proposes a high-level language that allows generic specification for Time Data Streams, along with a translation to the nuXmv model.*

Resumo. *A modelagem e certificação de sistemas, cruciais em sistemas críticos, são tarefas de alto teor de complexidade. O uso de ferramentas que isentem desenvolvedores dessa complexidade permite maior agilidade na construção e validação de software. Reo é uma linguagem gráfica focada em coordenações, promissora para facilitar a modelagem da interação de componentes de sistema. Timed Data Streams denotam, formalmente, o fluxo dos dados nessa linguagem, possibilitando a verificação de um modelo em relação a uma execução de sistema, feita nesse trabalho pela ferramenta nuXmv. Diante disso, este trabalho propõe uma linguagem em alto nível que permita a especificação genérica para Timed Data Streams, junto de uma tradução para o modelo nuXmv.*

1. Introdução

Com o desenvolvimento de computadores mais modernos e descentralizados e também de técnicas sofisticadas de desenvolvimento de *software*, ocorre atualmente uma grande difusão de sistemas móveis e distribuídos em larga escala. Esse fenômeno traz à tona maior preocupação com diferentes paradigmas computacionais, como o desenvolvimento baseado em componentes e orientado a serviços, culminando com a existência de robustos sistemas que também devem levar em conta propriedades como, sincronização, desacoplamento, transformação de dados, troca de mensagens etc [Kokash 2012].

Seguindo as tendências dessa dinâmica, deseja-se cada vez mais criar grandes sistemas composicionalmente com o uso de instâncias de serviços menores e desacoplados. Considerando essas particularidades, surgem também desafios para o desenvolvimento e certificação de sistemas, visto que essas instâncias não são facilmente "encaixadas". Consequentemente, surgem "brechas" de comunicação entre as instâncias, que devem ser tratadas para garantir o sucesso da cooperação entre essas. Esse tratamento é feito com

desenvolvimento adicional de *software*, como mecanismo de "coordenação" entre essas instâncias, feito de maneira externa a essas, para garantir independência das mesmas.

Assim, no caso de sistemas críticos, em que a modelagem e certificação formal são cruciais e complexas, as dificuldades se agravam, a medida que diferentes cenários complexos no paradigma de coordenação de componentes devem ser considerados . Uma maneira de garantir a corretude da implementação em relação a uma especificação de sistemas desse tipo, consiste em focar no estudo formal de modelos que possuem "interação" como um conceito primordial [Arbab 2006]. Aderindo essas questões foi proposto em [Arbab 2004], para facilitar as modelagens de sistemas de componentes, a linguagem gráfica *Reo* que é o principal formalismo que o trabalho deste artigo se baseia.

Anteriormente foi implementado um compilador de modelos *Reo* para o *model checker nuXmv* [Arena 2019] (https://github.com/frame-lab/Reo2nuXmv). Porém, o fluxo de dados das portas de *input*/*output* do modelo, especificado por *Timed Data Streams* (*TDS*), apresentadas na Seção 3, ainda não era modelado. Consequentemente, o modelo terá sua expressividade diminuída, assim como a visão "caixa preta" para um usuário que deseje verificar propriedades a respeito desse.

Com o objetivo de complementar a representação, permitindo verificar propriedades sobre o estado das portas de I/O, consequentemente, se raciocinando sobre a execução do sistema modelado, este trabalho apresenta uma linguagem alto nível. Essa linguagem permite especificar *TDS*s de maneira genérica bastante para expressar grande variedade de comportamentos para as portas com diferentes tipos de dados, e realiza a conversão automaticamente para o modelo *nuXmv* de maneira integrada ao compilador já existente. O novo compilador para a linguagem de *TDS* está sendo implementado em C, e seu código fonte, exemplos e instruções de uso, e também a gramática da linguagem podem ser encontrados em: https://github.com/frame-lab/tdsRepLanguage.

2. Reo

Reo é uma linguagem de modelagem gráfica baseada em coordenação exógena, onde essa possui como proposta a construção composicional de conectores complexos que impõe regras de comunicação, partindo de outros mais simples. Os princípios de *Reo* se baseiam na analogia introduzida em [Arbab 2004], em que uma forma eficiente e com alta reusabilidade de se construir modelos de coordenação em *glue code* consistiria em, criar diversos *scripts* mais simples e os integrar composicionalmente de forma genérica. Além de seguir essa analogia, *Reo* se demonstra promissor por "implementar" nativamente um *framework* de regras, propriedades e operações que facilitam a modelagem de sistemas distribuídos.

Um sistema modelado em Reo é composto por instâncias de componentes que interagem entre si por meio dos conectores de coordenação. Essas instâncias podem ser um conjunto de entidades abstratas de *software* ou componentes quaisquer que realizam diversas operações, somente importando as operações de *I/O* para o modelo. Os conectores de *Reo* são composições de diversos conectores "primitivos" denominados canais, que por sua vez são conexões ponto a ponto, possuindo nós de identificadores únicos e tipos distintos, sendo esses *source* (entrada), *sink* (saída), *mixed* (ambos). Onde esses canais são responsáveis por implementar semanticamente as propriedades do *framework* da linguagem.

Vale ainda ressaltar sobre *Reo*, que as propriedades do *framework* e a linguagem em si são extensíveis, possibilitando além da composição e combinação de conectores para formar regras de coordenação próprias, como também propriedades personalizadas. Essa extensão de expressividade é possível desde que regras e operações básicas do *framework* sejam seguidas, para não "extrapolar" o propósito da linguagem e manter os modelos semanticamente corretos [Arbab 2004]. Além disso a linguagem fornece conectores canônicos (Figura 1) que especificam comportamentos predefinidos, geralmente suficientes para implementação de modelos relativamente complexos.

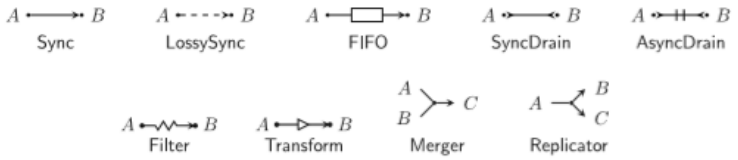

Figura 1. Conectores Canônicos Reo fornecidos por [Kokash 2012]

3. Constraint Automata

Considerando as vantagens do uso de *Reo*, como a notação gráfica intuitiva e aderência a propriedades de coordenação, ainda se faz necessário fornecer semântica formal a essa linguagem, para possibilitar o uso de *model checkers*. *Constraint Automata* [Baier C and Ruttend 2006] é um formalismo para descrever o comportamento e o possível fluxo de dados em modelos de coordenação de componentes. Onde esse pode ser visto como uma variação do autômato finito, onde transições são rotuladas pelo conjunto de nós (portas) de *I/O* de um determinado canal e predicados lógicos que representam as regras aplicadas sobre esse chamadas de *data constraints*.

Essas transições disparam de forma síncrona, baseadas nas avaliações das *data constraints* sobre um conjunto de dados do domínio da aplicação para determinado instante de tempo, representando o fluxo de dados do canal e o seu comportamento como um todo. Os estados assumidos pelo autômato representam as configurações assumidas pelo canal, por exemplo, qual tipo de dado (0 ou 1) está presente no conector *FIFO* de capacidade um, necessitando um estado para cada dado possível.

Definição 1 (*Constraint Automata*). Um *Constraint Automata* (*CA*) é uma tupla $A = (S, \mathit{Names}, s_0, \rightarrow)$ composta de: um conjunto de estados *S*, um conjunto de rótulos das portas existentes *Names*, um estado inicial $s_o \in S$, e uma relação de transição $\rightarrow$ sobre os conjuntos $S \times 2^{\mathit{Names}} \times \mathit{DC} \times S$. Onde o conjunto *DC* representa as *data constraints* citadas anteriormente, sendo um conjunto de predicados lógicos sobre as portas e dados do sistema. Assim uma transição é rotulada por um nome $n_i \in \mathit{Names}$ e um predicado $dc_i \in \mathit{DC}$, sendo escrita como: $s_0 \xrightarrow{n_i, dc_i} s_n$.

Um *CA* depende fortemente da noção de tempo para aplicar suas transições e representar o fluxo de dados e seu domínio corretamente. Seguindo esse princípio, foi introduzido em [Baier C and Ruttend 2006] *Timed Data Streams*, também chamadas de linguagens *TDS*, que são uma associação de um fluxo infinito de dados para qualquer canal *Reo*. Essa associação é feita considerando um *CA* como "aceitador" de *TDS*, isto é feito com o autômato "observando" os dados de I/O ocorrendo em um instânte t_x na porta

$n_i \in Names$. Assim baseando-se nas *data constraints* e na "observação" de uma tupla $(porta, dado, instante)$ é decidido se o fluxo de dados é válido, aceitando ou não a *TDS*.

Definição 2 (*Timed Data Streams*). *TDS* é definida por um par de funções (α, a) sendo cada uma *stream*, aplicada respectivamente, sobre os conjuntos: $Data$ um conjunto de dados possíveis, e R_+ os números reais positivos representando os instantes de tempo.

$$TDS = \{(\alpha, a) \in Data^w \times R_+^w : \forall i > 0 : a(i) \leq a(i+1)\}. \quad (1)$$

Onde uma *stream* A^w é definida como todas as sequências infinitas de números mapeadas sobre um conjunto A qualquer, sendo denotada por $A^w = \{\alpha | \alpha : \{0, 1, 2, ..\} \rightarrow A\}$. Sendo equivalente escrever individualmente para cada parâmetro (número) mapeado em A^w: $\alpha = \alpha(0), \alpha(1), \alpha(2)...$. Além disso é possível referenciar *streams* em função de outras (anteriores ou posteriores) usando sua forma "derivada" : $\alpha' = \alpha(1), \alpha(2), \alpha(3)...$, isto é, $\alpha^{(0)} = \alpha$ e $\alpha^{(i+1)} = \alpha^{(i)'}$.

Como estabelecido anteriormente, para as transições de fluxo de um *CA* é observado uma tupla $(porta, dado, instante)$. Ou seja, é necessário para formalizar o comportamento de *I/O* do modelo de coordenação, realizar uma associação entre as *TDS* e as portas envolvidas.

Definição 3 (*TDS*^{Names}). É uma tupla representando a associação entre cada par $\in$ *TDS* e cada rótulo de portas do sistema $n_i \in$ *Names*, denotada por:

$$TDS^{Names} = \{(\langle \alpha_1, a_1 \rangle, ..., \langle \alpha_1, a_n \rangle) : \langle \alpha_i, a_i \rangle \in TDS, i = 1, ..., \mid Names \mid\}. \quad (2)$$

Assim um *CA* de *TDS* $\subseteq$ *TDS*^{Names} pode ter sua semântica escrita em função dessa "linguagem", por exemplo, a linguagem *TDS* que denota o conector FIFO de capacidade 1:

$$L_{fifo} = \{(\langle \alpha, a \rangle, \langle \beta, b \rangle) \in TDS \times TDS | \beta = \alpha \wedge a < b < a'\}. \quad (3)$$

Onde $\beta = \alpha \wedge a < b < a'$ representa uma *data constraint* escrita em função de *streams* (anteriores e posteriores), sendo essas os instantes atuais e no futuro de a e b.

Além disso, um dos princípios de *Reo* é a construção composicional de conectores, a ideia equivalente para um *CA* é o produto entre autômatos, descrito brevemente como sendo a união de duas linguagens *TDS*. Seja L_1 e L_2 induzidas pelos autômatos A_1 e A_2, e dados dois circuitos *Reo* com conjunto de nós N_1 e N_2, a ideia por trás dessa operação é similar a operação *join* usada em bancos de dados relacionais, sendo realizada a junção dos pares de portas em comum $\in N_1 \cap N_2$ e mantendo as demais portas $\in N_1 \times N_2$, resultando em um autômato que induz a seguinte linguagem *TDS*:

$$L_r = \{(\langle \alpha, a \rangle, \langle \beta, b \rangle, \langle \delta, c \rangle) : (\langle \alpha, a \rangle, \langle \beta, b \rangle) \in L_1 \wedge (\langle \beta, b \rangle, \langle \delta, c \rangle) \in L_2\} \quad (4)$$

4. nuXmv

O *Model Checker nuXmv* foi escolhido por possuir características predominantes em sistemas distribuídos, como por exemplo, modularização e sincronização evidenciado pela criação de módulos lógicos (**MODULE**) que executam de maneira síncrona, e também a possibilidade de verficar *Fairness* e *Liveness* dos sistemas modelados [Cavada R 2014]. A construção de um modelo de sistema é ilustrada de forma resumida no Listing 1, que representa o módulo *main* de um *product automata* entre os conectores *Merger* e *FIFO*.

O fluxo de construção de um modelo geralmente consiste em criar o módulo *main* e outros associados, compostos de variáveis, *ENUM*s ou de outros módulos (linhas 2-4), e usar atribuições para definir o estado inicial e seguinte desses elementos (linhas 6-10). Onde essas atribuições podem ser feitas com expressões condicionais podendo representar transições de uma máquina de estados (linhas 10-12, expressões entre os delimitadores *case* e *esac*). Por último, pode-se especificar uma expressão de lógica temporal (*Computer Tree Logic* ou *Linear Temporal Logic*) que representa a propriedade a ser verificada, no exemplo, se para todos os caminhos do modelo as portas A e B não concorrem para o envio de dados, assim transferindo alternadamente (linha 11-12 pela diretiva **CTLSPEC**).

```
1  MODULE main
2  VAR time : 0..6;
3      modoEntrada : boolean;
4      CA : finalAutomata(time,modoEntrada);
5  ASSIGN
6   init(time) := 0;
7   init(modoEntrada):= FALSE;
8   next(time):= case    time = 6 : 0;  TRUE : time+1; esac;
9   next(modoEntrada):= case next(time) = 2 : TRUE; next(time) = 0: FALSE;
10                          TRUE : modoEntrada; esac;
11 CTLSPEC AG((CA.ports.a != NULL & CA.ports.b = NULL)
12              | (CA.ports.a = NULL & CA.ports.b != NULL));
13         . . .
```

Listing 1. Exemplo resumido de um modelo nuXmv

5. A linguagem proposta : *Tdsrepl*

5.1. Características da linguagem

A linguagem permitirá por meio de comandos simples, funções, e diretivas próprias, representar *TDS*s para portas quaisquer de um circuito *Reo*, e então expressar a relação de dependência entre elas, manipular e visualizar seus dados antes de converter para o *nuXmv*. A linguagem considera em sua gramática os comandos mais simples e comuns a outras linguagens de programação e a definição de procedimentos e funções.

Apesar das similaridades com linguagens de programação, o *design* dessa linguagem parte do principio de que para representar *TDS*s o usuário não é obrigado a criar um programa. Assim, apesar de que seja possível criar programas limitados com os comandos básicos oferecidos, a linguagem tem como foco principal a utilização de suas diretivas e representações simplificadas de *TDS* e funções. Um exemplo de uso é demonstrado no Listing 2, criando quatro *TDS*s com comportamentos distintos, associadas a diferentes portas, representando o fluxo de dados do conector composto *Merger-Fifo*.

A linguagem conta com "diretivas temporais" e "diretivas TDS", auxiliando a definição de intervalos de observação e comportamentos com noções de dependência temporal. As diretivas temporais: *I_TIME*, *C_TIME* e *F_TIME*, respectivamente, representam o instânte incial, atual e final de observação, e iniciam com valores *default* 0, 0 e 3, porém, as diretivas podem ser usadas e alteradas em diversos contextos, como visto no listing 2.

Vale ressaltar que, a diretiva *C_TIME* possui o valor *default* ou outro especificado somente fora do contexto das diretivas TDS. Onde essas diretivas, associam o rótulo

de uma porta qualquer aos comportamentos especificados que ditam presença de dados nessa, visto em atribuições à variáveis *TDS* do listing 2. Quando os comportamentos são definidos em função de *C_TIME*, essa possui o valor "variando" nesse contexto, isto é, o comportamento é aplicado N vezes entre o intervalo de observação definido, iniciando com o valor *default* ou outro atribuído e terminando a aplicação com o valor de *F_TIME*.

Os comportamentos das *TDS* podem ser gerados de diversas formas, seja utilizando procedimentos comuns a linguagens de programação (linhas 4-10), ou por uma função "matemática" (linha 17), ou uma lista de dados para cada segundo (linha 12), ou ainda por outra *TDS* já especificada. Onde no último caso, deve-se especificar o input para a nova tds, feito pela diretiva *linked*. Além disso, pode-se especificar "atraso" para as portas (ex: *fifo* Figura 1, formalização 3) com a diretiva *delayed* (linha 19).

```
1  I_TIME = 0
2  F_TIME = 6
3  modoEntrada = false
4  function fluxoTempo(modoEntrada){
5      iv = !modoEntrada
6      if(iv == false and C_TIME != F_TIME) {
7       return C_TIME % 2
8      }
9      return NULL
10 }
11 tdsA = { portname : 'A',
12           data-time: {0: 0, 1: 1, 2: 0} }
13 C_TIME = 2
14 modoEntrada = true
15 tdsB = {portname : 'B',
16           data-time : {function-domain: fluxoTempo(modoEntrada)} }
17 tdsBalt={portname:'Balt', data-time :{function-domain: C_TIME/C_TIME}}
18 C_TIME = 0
19 tdsC = {portname : 'C', linked : {tdsA, tdsB}, delayed : true }
20 tdsD = {linked: {tdsC} }
```

Listing 2. Mapeando portas usando, programas ,listas de dados e diretivas

5.2. Conversão para o nuXmv

As *TDS's* especificadas na linguagem são convertidas para *constraints* lógicas (similares a *Data constraints* da formalização 3) utilizadas nas atribuições do fluxo de construção do modelo *nuXmv*. Onde essas conversões seguem as regras estabelecidas pelo *reo2nuXmv*, consequentemente, esse processo só irá modificar os módulos *main* e o que representa as portas de I/O (*portsModule*), sendo essas modificações, respectivamente, reflexos dos comandos presentes nos escopos globais do código da linguagem e do escopo de contexto de cada variável *TDS*. Além disso, serão modificadas as referências ao *portsModule*, feitas por outro **MODULE** qualquer, onde esse representa o autômato equivalente aos conectores *Reo* (homônimo ao conector em questão ou finalAutomata em casos de produto). Os listings 1 (Seção 4) e 3 (ao fim dessa seção) ilustram as partes da conversão no módulo *main* e *portsModule* baseadas no exemplo da seção anterior 5.1 (Listing 2).

Todos os **MODULEs** recebem um parâmetro *time* criado no módulo *main* (Listing 1 linhas 2 e 4), que terá sua declaração e valores especificados pelas diretivas temporais da linguagem e seu intervalo de observação (Listing 2 linhas 1 e 2). Esse parâmetro

sera inicializado com o valor de *I_TIME* variando até *F_TIME* e depois reiniciar caso o usuário desejar continuar testando outros cenários do modelo (Listing 1 linha 8, primeira *constraint*. Além disso todas as demais variáveis que foram definidas por valores, ou seja, não apresentam dependência com outras variáveis, também serão reiniciadas após o fim de cada cenário, por meio da verficação da *constraint next(time) = 0*.

Comandos simples, como declarações seguidas de atribuições terão suas variáveis declaradas como uma **VAR** no *nuXmv*, possuindo o mesmo tipo do valor atribuído, e inicializadas com *init(var):= VALOR*, como visto na seção **ASSIGN** do listing 1 para o caso do escopo global, e listing 3 para o escopo da função associada a *TDS* de porta B para a varíavel "ivScopeB". Vale ressaltar que, caso *C_TIME* seja modificado antes de qualquer definição (ex: listing 2 linha 13), o "contexto temporal" será modificado, e a inicialização de *var* será feita utilizando condições lógicas **case** em função do parâmetro *time*, fazendo a atribuição não valer antes do parâmetro se igualar a *C_TIME*.

Reatribuições de variáveis declaradas serão convertidas para o comando *next(var):= VALOR*, onde o valor será o último utilizado no fluxo de especificação, ou caso tenha ocorrido mudança de "contexto temporal" *next* será atribuído em função do tempo (listing 1 linha 9). Análogo ao último caso, comandos mais complexos que referênciam variáveis, por exemplo, *if*, *else* e *loops*, serão convertidos para atribuiçoes com *init* ou *next* utilizando **case** (Listing 3 linha 18, equivalente ao if do Listing 2 linha 6).

As variáveis que recebem uma especificação das diretivas de *TDS* são convertidas para variáveis do módulo *portsModule*, desde que essas sejam utilizadas, isto é, sirvam como *streams* de I/O (Listing 3 linhas 2-6, porta "Balt" não é declarada). Essas varíaveis tem seus tipos de dados definidos como o conjunto de todos resultados de cada aplicação do comportamento da *TDS* associada, onde no caso da diretiva *linked*, o conjunto é a união dos conjuntos de todas as portas envolvidas.

Os comportamentos geradores de dados associados as *TDS* são convertidos de maneira análoga aos comandos de atribuição e aos de comandos mais complexos, visto na seção **ASSIGN**(listing 3 linhas 8-23). A diferença, é que as atribuições **case** são geralmente em função do parâmetro *time* ou algum parâmetro passado para a função associada a *TDS* (consequentemente também passada para o *portsModule*, no exemplo, o parâmetro "modoEntrada"), ou ainda por outra variável que representa uma porta. No último caso, essas são conversões do uso de *linked*, onde uma variável de porta tem sua atribuição *init* e *next* em função do valor de apenas uma das suas portas de input por vez, simulando o comportamento do conector *merger*, como pode ser visto para a porta C nos exemplos.

Vale ressaltar ainda algumas peculiaridades sobre a conversão dos comportamentos de uma variável *TDS*, onde no caso da diretiva *delayed* a conversão é analoga as especificações anteriores, porém, a atribuição *init* dessa variável será restringida (NULL) e uma verificação lógica (similar a formalização 3, vista no Listing 3 linha 23) será feita para a atribuição *next*. Além disso, os comportamentos das *tds* também podem ser afetados por "contexto temporal", com variáveis tendo os valores restringidos (Listing 3 linha 17, representa o contexto da linha 13 do Listing 2 antes de C_TIME = 2).

```
1 MODULE portsModule(time, modoEntrada)
2 VAR a: {0, 1, NULL};
3     b: {0, 1, NULL};
4     c: {0, 1, NULL};
```

```
5       d: {0, 1, NULL};
6       ivScopeB : boolean;
7   ASSIGN
8    init(a):= 0;
9    init(b):= NULL;
10   init(ivScopeB) := !modoEntrada;
11   init(c):= case a = NULL : b; b = NULL : a; TRUE : NULL; esac;
12   init(d):= NULL;
13
14   next(ivScopeB) := !next(modoEntrada);
15   next(a):= case next(time) = 1 :1; next(time) = 2 :0; next(time) = 0:0;
16                 TRUE : NULL;     esac; -- reset para valor init
17   next(b):= case next(time) < 2: NULL; -- mudou o contexto temporal
18         next(ivScopeB) = FALSE & next(time) != 6: next(time) mod 2; --if
19         TRUE : NULL; -- else
20         next(time) = 0 : NULL;   esac; -- reset para valor init
21   next(c):= case next(a)= NULL: next(b); next(b)= NULL : next(a);
22                 TRUE : NULL;     esac; -- reset para valor init
23  next(d):= case d = NULL: c; c = NULL & d != NULL : NULL; TRUE : d;esac;
24  MODULE finalAutomata(time, modoEntrada) --pelo tamanho do codigo gerado
25      . . .                       --as transicoes do automato foram ocultadas
```

Listing 3. MergerFifo de dados 0 e 1

6. Conclusão e trabalhos futuros

Este trabalho apresentou uma linguagem para formalizar *Timed Data Streams* e a lógica de um processo de tradução para o *nuXmv*, fazendo uso de conceitos de *Reo* e dos formalismos apresentados. Atualmente, o projeto já possui um *parser* funcional, e a implementação do processo de tradução está sendo terminada. Para trabalhos futuros, visa-se a integração completa com o *reo2nuXmv*, além de melhorias na gramática e recursos da linguagem, visando a usabilidade. Melhorias já pensadas incluem, um modo "interativo", similar ao usado pelo *nuXmv*, que permitirá o usuário verificar visualmente os comportamentos especificados para fluxos de dados antes do processo de conversão.

Referências

Arbab, F. (2004). *"Reo: a channel-based coordination model for component composition"*. Math. Struct. in Comp. Science, vol.14, pp.329–36. Cambridge University.

Arbab, F. (2006). *"Coordination for Component Composition"*. Electronic Notes in Theoretical Comp. Science vol.160, pp.15–40. Elsevier.

Arena, D. (2019). *"Um compilador de circuitos Reo para modelos nuXmv"*. Trabalho de conclusão de curso - Instituto de Computação, Universidade Federal Fluminense.

Baier C, Sirjanib M, A. F. and Ruttend, J. (2006). *"Modeling component connectors in Reo by constraint automata"*. Science of Comp. Programming, vol.61, pp.75–113. Elsevier.

Cavada R, e. a. (2014). *The nuXmv Symbolic Model Checker. In:Biere A., Bloem R.(eds)*. CAV 2014. Lecture Notes in Computer Science, vol 8559. Springer, Cham.

Kokash, N. (2012). *"Reo +mCRL2: A framework for model-checking dataflow in service compositions"*. Formal Aspects of Computing, vol.24, no2, pp.187-216. Hasso Plattner Institute.