FACULDADE DE INFORMÁTICA
PUCRS – Brazil
http://www.inf.pucrs.br

Uma Análise de Linguagens de Especificação para Sistemas Distribuídos

Lucio Mauro Duarte e Fernando Luís Dotti

TECHNICAL REPORT SERIES

Number 016
October, 2001

Contact:

lduarte@inf.pucrs.br

http://www.inf.pucrs.br/~lduarte

fldotti@inf.pucrs.br

http://www.inf.pucrs.br/~fldotti

Lucio Mauro Duarte is a graduate student at PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul – Brazil. He is member of the Parallel and Distributed Processing research group since 2000. He receives a federal graduate research grant from CAPES (Brazil) to support his research.

Fernando Luís Dotti works at PUCRS/Brazil since 1998. He is an associate professor and develops research in Computer Networks, Distributed Systems and Code Mobility. He got his Ph.D. in 1997 at Technical University of Berlin (Germany).

Published by the Campus Global – FACIN – PUCRS
Av. Ipiranga, 6681
90619-900 Porto Alegre – RS – Brazil

Uma Análise de Linguagens de Especificação para Sistemas Distribuídos

Relatório Técnico 016/2001

Lucio Mauro Duarte (mestrando)
Fernando Luís Dotti (orientador)

1 Introdução

O aumento nas dimensões das redes, possibilitado pelos avanços tecnológicos na área, exige, cada vez mais, a existência de sistemas compostos por componentes distribuídos na rede que permitam que operações possam ser realizadas à distância. Obter-se dados e serviços de uma fonte distante passou a ser uma coisa corriqueira e, portanto, desejável. Sistemas distribuídos cada vez maiores e mais complexos têm sido desenvolvidos para atender a variadas necessidades. Este crescimento em tamanho e em complexidade aumenta também a dificuldade do desenvolvedor no processo de teste e depuração destes sistemas. Mesmo para sistemas que executam localmente, dependendo da sua complexidade, é uma tarefa difícil verificar se ele funciona corretamente. Esta tarefa torna-se ainda mais árdua quando se tratam de sistemas que têm componentes executando em máquinas diferentes. Questões como comunicação, concorrência, paralelismo, compartilhamento de informações e tolerância a falhas devem ser consideradas.

Através da abordagem tradicional de construção de sistemas, os passos seguidos são a descrição e modelagem (informal) do sistema, o desenvolvimento e a fase de testes e depuração. Por essa seqüência de etapas, a maioria dos problemas de projeto do sistema só será identificada quando o sistema já estiver na fase de testes, exigindo retornar-se à etapa de modelagem, recomeçando o ciclo de construção. Isto porque a modelagem utilizada não possui uma base formal, a partir da qual se possam verificar, matematicamente, algumas das características desejadas para o sistema. Seguindo essa abordagem tradicional, saber se um sistema distribuído funciona corretamente não é trivial, pois requer a realização de diversos testes, reparos e alterações. Na maioria das vezes, os testes realizados não cobrem todas as situações possíveis e acabam acontecendo falhas não previstas, gerando a necessidade de manutenção constante do sistema.

Atualmente, temos os chamados *sistemas abertos*. Por *sistema aberto* entende-se um sistema autônomo quanto à sua execução, com ciclo de vida independente de outros sistemas (o que significa que ele pode não estar disponível em algum momento, caracterizando o seu dinamismo) e que possui capacidade de comunicação com os demais sistemas. Essa capacidade de comunicação com sistemas sobre os quais ele não tem, *a priori*, informação alguma é que o define como aberto. Ao considerarmos o dinamismo desses sistemas, temos uma dificuldade muito maior em prever e prevenir, segundo a abordagem tradicional, a ocorrência de determinadas situações e/ou comportamentos. Em sistemas como esses, uma aplicação pode gerar, com os mesmos dados de entrada, resultados diferentes a cada execução, dependendo do estado da rede.

A complexidade da tarefa de construção de sistemas aumenta ainda mais quando se atribui aos componentes do sistema a capacidade de mobilidade. Com isso, além de

possuirmos componentes distribuídos e autônomos, passamos a considerar componentes que podem estar transitando na rede e executando em vários pontos desta. A dúvida quanto ao funcionamento correto de sistemas que envolvem a utilização de componentes móveis, portanto, revela ainda maior a necessidade de se poder, antes mesmo da implementação do sistema, possuir uma descrição fiel (no sentido de que descreve exatamente o que é esperado do sistema) e precisa do comportamento do sistema. Este tipo de descrição constitui-se em uma garantia de que não será necessário retomar o projeto do sistema após a sua implementação, a menos que haja uma mudança na semântica esperada deste.

Por tudo isso, este trabalho apresenta um estudo sobre *linguagens de especificação formal de sistemas*, focando-se em especificações para sistemas distribuídos. Tais linguagens pretendem fornecer uma descrição não-ambígua de um sistema que permita testar-se e provar-se matematicamente que tal sistema possui as propriedades esperadas. Por descrição não-ambígua entenda-se uma descrição que é clara e precisa o suficiente, utilizando-se para isso de uma linguagem formal, para permitir apenas uma única interpretação de seu conteúdo. Esta interpretação única nem sempre é fornecida por descrições feitas em linguagem natural, onde diferentes interpretações podem gerar diferentes resultados. Para prover esta descrição não-ambígua, as linguagens de especificação formal fornecem um modelo matemático para descrição de sistemas, a partir do qual podem ser realizadas verificações, a fim de provar-se que o sistema possui certas propriedades como, por exemplo, a ausência de uma situação de *deadlock.*

Este trabalho tem, portanto, como objetivo principal apresentar algumas das linguagens de especificação que podem ser usadas para descrever sistemas distribuídos. Para alcançar tal objetivo, realizaram-se os objetivos específicos que eram: estudar algumas das linguagens de especificação formal possíveis de serem utilizadas para especificarem-se sistemas distribuídos; estabelecer critérios de comparação entre as linguagens estudadas; e analisar cada uma das linguagens em relação aos critérios estabelecidos.

O texto deste trabalho está organizado da seguinte forma: a seção 2 apresenta as linguagens de especificação formal *TLA (Temporal Logic of Actions), CSP (Communicating Sequential Processes), SDL (Specification and Description Language)* e *Gramáticas de Grafos*, fornecendo uma exemplificação de uso de cada linguagem; no seção 3 tem-se uma análise comparativa entre as linguagens apresentadas na seção 2; a seção 4 apresenta a conclusão do trabalho; e a seção 5 relaciona as referências bibliográficas.

2 Linguagens de Especificação

Para que seja possível verificar-se de maneira formal (matemática) se um sistema satisfaz certas propriedades desejadas, deve ser construído um modelo matemático que descreve o comportamento deste sistema, o que pode ser feito utilizando-se uma linguagem de especificação formal. Uma *especificação* é uma descrição de alto-nível (abstrata) do sistema a ser construído, a qual deve ser compacta, precisa e não-ambígua. Uma *especificação formal* [2] [3] é uma descrição de um sistema feita em uma linguagem com sintaxe e semântica precisamente definidas; ou seja, definidas utilizando-se conceitos matemáticos. Esta linguagem é chamada de *linguagem de especificação formal* (*LEF*). Dada uma especificação formal do sistema, pode-se provar a existência ou não de uma propriedade ou característica. Esta prova pode ser realizada

através de uma abordagem de *verificação formal*. A verificação formal consiste em utilizarem-se técnicas matemáticas para assegurar que um sistema computacional apresenta uma certa propriedade. O uso de técnicas matemáticas permite considerar todos os casos possíveis, mesmo quando o número de casos é muito grande [1]. Isto implica provar-se, em tempo de projeto, que o sistema apresenta a propriedade desejada em qualquer situação, sem a necessidade de realizarem-se testes posteriores na etapa pós-implementação, quando é mais difícil identificar e corrigir erros. Isto, é claro, só é possível se for possível garantir-se que a implementação correspondente fielmente à especificação realizada.

A seguir, são apresentadas algumas das LEF mais conhecidas e utilizadas. Tais LEF foram escolhidas por representarem diferentes classes de linguagens de especificação, tais como lógica temporal (TLA), cálculo de processos (CSP), *event-driven* (SDL) e *data-driven* (Gramáticas de Grafos).

2.1 Temporal Logic of Actions (TLA)

Sistemas concorrentes são geralmente descritos em relação a seus *comportamentos*, que definem o que eles realizam durante sua execução. Em 1977, Amir Pnueli, em [6], introduziu o uso de lógica temporal para descrever estes comportamentos. No fim dos anos 80, Leslie Lamport criou uma variação da lógica de Pnueli chamada *Temporal Logic of Actions* ou *TLA* [5]. TLA provê fundamentação matemática para a especificação de sistemas concorrentes, descritos através de fórmulas. TLA apresenta algumas idéias trazidas das linguagens de programação, como a divisão da especificação em módulos [4].

Uma especificação em TLA é uma fórmula em lógica temporal que expressa um predicado sobre comportamentos, onde *predicado* refere-se a uma expressão *booleana* construída a partir de variáveis e constantes. Um comportamento em TLA corresponde a uma seqüência infinita de *estados* [7], onde um estado representa atribuições de valores a variáveis [5]. A semântica de um sistema é o conjunto de comportamentos que ele pode assumir. Um par de estados, um representando o estado antigo e outro o novo, definem o que é chamado de *passo*.

Todas as fórmulas em TLA podem ser descritas através dos operadores matemáticos conhecidos, como os operadores aritméticos + (adição), - (subtração), * (multiplicação) e / (divisão), os operadores da lógica proposicional $\wedge$ (conjunção), $\vee$ (disjunção), $\equiv$ (equivalência), $\neg$ (negação) e $\Rightarrow$ (implicação), os operadores de comparação $<$ (menor), $>$ (maior), $=$ (igualdade), $\leq$ (menor ou igual), $\geq$ (maior ou igual), os quantificadores $\forall$ (quantificador universal) e $\exists$ (quantificador existencial) e os operadores de conjuntos $\cap$ (interseção), $\cup$ (união), $\subset$ (subconjunto) e \ (diferença).

Além dos operadores apresentados, são definidos os operadores ’ (*prime*), $\Box$ (*always*), $\Diamond$ (*eventually*) e ~> (*leads to*). O primeiro define o valor de uma certa variável no estado seguinte; ou seja, *v’* define o valor da variável *v* no próximo estado do sistema, onde *v’* é dita uma *primed variable*. Este operador é utilizado para definir a idéia de *ação*, a qual é definida por uma expressão de valor *booleano* formada por variáveis, *primed variables* e símbolos constantes. Uma ação representa uma relação entre estados antigos e estados novos, sendo que variáveis referem-se ao estado antigo e *primed variables* referem-se, como já citado, ao novo estado. Dessa forma, $x' = y + 1$ é uma relação que afirma que o valor de *x* no novo estado (representado por *x’*) é igual ao valor de *y* no estado anterior mais um. Os operadores unários $\Box$ e $\Diamond$ são usados na construção de *fórmulas temporais*, juntamente com os operadores *booleanos*. Dada uma fórmula *F* , a fórmula temporal $\Box F$ define que *F* é sempre verdadeira, enquanto que a

fórmula $\Diamond F$ define que F é eventualmente verdadeira. Estes dois operadores podem ser combinados, de forma que a fórmula temporal $\Box\Diamond F$ determina que a fórmula F é verdadeira com uma freqüência infinita e a fórmula temporal $\Diamond\Box F$ determina que eventualmente a fórmula F é sempre verdadeira. Dadas as fórmulas temporais F e G, define-se $F \rightsquigarrow G$ como igual a $\Box(F \Rightarrow \Diamond G)$. Ou seja, $F \rightsquigarrow G$ determina que quando F é verdadeira, G é verdadeira a seguir ou algum tempo mais tarde.

Definições em TLA são apresentadas como $Id \triangleq exp$, onde *Id* representa um identificador e *exp*, uma expressão. Uma *definição* determina que *Id* é um nome associado à expressão *exp*. Assim, a definição $x \triangleq a + b$ torna escrever $x * c$ similar a escrever $(a + b) * c$.

TLA não possui nenhuma primitiva de comunicação predefinida. Por isso, para representar a comunicação, pode-se assumir que as fórmulas de TLA usam variáveis compartilhadas como uma primitiva de comunicação [5]. Também se pode usar a própria TLA para definir as primitivas de comunicação que se queira, como a criação da idéia de canais apresentada em [4].

Lamport também propôs uma linguagem formal para escrever especificações em TLA chamada de TLA$^+$ [4]. Uma especificação em TLA$^+$ é particionada em módulos. Módulos definidos podem ser incorporados em novos módulos, permitindo a utilização de definições já especificadas. TLA$^+$ possui alguns módulos já definidos, como o módulo *Naturals* que define operadores de números naturais. Esta modularidade e incorporação de módulos previamente escritos a novos módulos permitem que, por exemplo, tenha-se um módulo em que o operador + seja definido como a adição de dois números naturais, como em *Naturals*, e um outro módulo em que tal operador poderia ser definido como a adição de duas matrizes. Com isso, cada símbolo presente em uma fórmula deve ser um operador padrão de TLA$^+$ ou ter sido definido ou incorporado e a sua semântica depende da definição atribuída a ele dentro do escopo do módulo em que ele é utilizado.

2.1.1 Exemplo de Especificação com TLA

Para exemplificar uma especificação de sistema feita com TLA, será apresentada a especificação de uma interface para transmissão de dados entre dispositivos assíncronos, apresentada em [4]. Esta mesma aplicação será utilizada nos exemplos das outras linguagens a serem descritas. Neste sistema, temos um *Sender* e um *Receiver* conectados conforme a Figura 1.

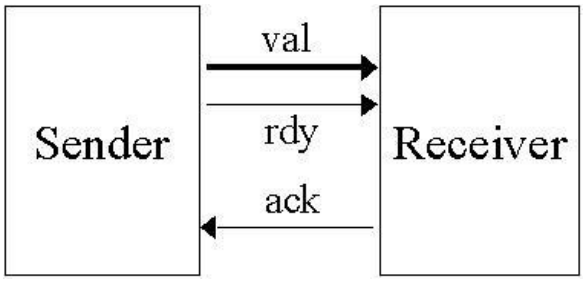

Figura 1. Ilustração do sistema de troca dados entre interfaces assíncronas.

Dados são enviados através de *val* e as linhas *rdy* e *ack* são usadas para sincronização. A interface segue o protocolo de *two-phase handshake*, segundo o qual o *Sender* envia um dado e espera-se a confirmação de recebimento pelo *Receiver* para

enviar o dado seguinte. Quando um dado é enviado, ele é transmitido através de *val* e o sinal em *rdy* é trocado, informando ao *receiver* que o *Sender* enviou um dado e espera confirmação de recebimento. O *Receiver* então confirma o recebimento trocando o sinal em *ack*. Um exemplo de execução deste protocolo é descrito na Figura 2.

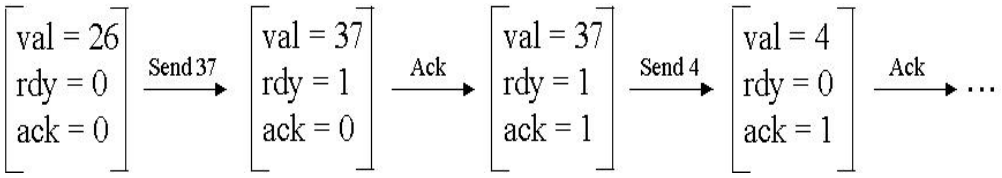

Figura 2. Exemplo de execução do protocolo *two-phase handshake*

A especificação em TLA$^+$ para a interface de comunicação assíncrona, conforme apresentado em [4], é descrita na Figura 3.

MODULE *AsynchInterface*

1. EXTENDS *Naturals*
2. CONSTANT *Data*
3. VARIABLES *val*, *rdy*, *ack*
4. *TypeInvariant* $\triangleq \land val \in Data$
5. $\land rdy \in \{0, 1\}$
6. $\land ack \in \{0, 1\}$

---

7. *Init* $\triangleq \land val \in Data$
8. $\land rdy \in \{0, 1\}$
9. $\land ack = rdy$

10. *Send* $\triangleq \land rdy = ack$
11. $\land val' \in Data$
12. $\land rdy' = 1 - rdy$
13. $\land$ UNCHANGED *ack*

14. *Rcv* $\triangleq \land rdy \neq ack$
15. $\land ack' = 1 - ack$
16. $\land$ UNCHANGED $\langle val, rdy \rangle$

17. *Next* $\triangleq Send \lor Rcv$
18. *Spec* $\triangleq Init \land \Box[Next]_{\langle val, rdy, ack \rangle}$

---

19. THEOREM *Spec* $\Rightarrow \Box TypeInvariant$

Figura 3. Especificação TLA para interface de comunicação assíncrona.

A especificação da Figura 3 apresenta o módulo *AsynchInterface*, que possui a especificação para a interface assíncrona de comunicação. Deve-se ressaltar que a especificação compreende apenas a interface de comunicação entre *Sender* e *Receiver*, sendo que estes dois módulos seriam especificados separadamente e utilizariam esta especificação da interface de comunicação para descreverem a troca de dados. A linha 1 apresenta a declaração do módulo predefinido a ser incorporado à especificação. Todos os módulos predefinidos a serem utilizados por um módulo novo são declarados após a palavra reservada EXTENDS. No caso deste módulo, determinou-se que *rdy* e *ack* podem assumir os valores 0 ou 1, que são valores naturais e, portanto, utiliza-se o módulo onde estão as definições dos naturais (*Naturals*). A linha 2 apresenta a lista de constantes do módulo seguidas da palavra reservada CONSTANT (ou CONSTANTS, para o caso de mais de uma). Para este módulo define-se apenas a constante *Data*, que determina o conjunto de valores que podem ser assumidos por *val*. Com isso, não é necessário definir o tipo de dado a ser transmitido; sabe-se apenas que ele pertence ao conjunto definido por *Data*. A linha 3 traz as variáveis que, neste caso são as já citadas: *val*, *rdy* e *ack*. As variáveis são precedidas da palavra reservada VARIABLES (ou VARIABLE, para o caso de uma só). As linhas 4 a 6 apresentam a definição de *TypeInvariant* que determina, de forma clara, os valores que podem ser assumidos pelas variáveis declaradas. Assim, sabe-se que *val* só assumirá valores pertencentes ao conjunto *Data* e que *rdy* e *ack* só poderão ter valor 0 ou 1. O símbolo $\wedge$ ao lado de cada linha determina a conjunção das condições estabelecidas. O mesmo ocorre para a disjunção de condições, onde cada linha é precedida do símbolo $\vee$.

Depois das declarações, seguem-se as definições principais. As linhas 7 a 9 definem *Init* como o predicado inicial, ou seja, condições iniciais da interface. O predicado inicial não apresenta *primed variables* por não representar uma mudança de estado e sim a configuração inicial do sistema. Assim, define-se que, inicialmente, val assume o valor de um elemento qualquer de *Data*, *rdy* possui valor 0 ou 1 e que *ack* possui o mesmo valor de *rdy*.

A seguir, são definidas as ações *Send* (linhas 10 a 13) e *Rcv* (linhas 14 a 16) que correspondem, respectivamente, às mudanças de estados em caso de envio e recebimento de dado. Ao contrário do que acontece com o predicado inicial, as ações possuem *primed variables* por representarem mudanças de estado e por definirem os valores a serem assumidos pelas variáveis no novo estado.

A ação *Send* é permitida em um estado a partir do qual é possível realizar-se um passo de envio. A condição que habilita um passo de envio está apresentada na linha 10. Isto é, uma ação *Send* só é possível se *rdy = ack*. Dessa forma, temos, dentro da definição de uma ação, condições de execução da ação (não possuem *primed variables*) e as definições dos novos valores assumidos pelas variáveis quando a ação é executada (contendo *primed variables* que expressam a mudança de estado). Sendo satisfeita a condição da linha 10, a linha 11 determina que *val* assume um novo valor a ser transmitido, o qual pertence ao conjunto *Data*. A linha 12 define que ao ser enviado um dado *rdy* tem seu valor trocado, de forma que, se seu valor era 0, passa ser 1 e vice-versa, já que a troca de seu valor sinaliza a ocorrência de um envio de dado do *sender* ao *receiver*, como visto anteriormente. A linha 13 determina, através da palavra reservada UNCHANGED, que *ack* permanece com o mesmo valor que possuía no estado antigo, isto porque ela não é afetada quando há um envio de dado. Cabe citar que UNCHANGED *ack* equivale a *ack'* = *ack*, mas a palavra reservada é usada para fins de maior clareza e entendimento. O uso de UNCHANGED se faz necessário porque os novos valores de cada uma das variáveis da especificação devem ser definidos em cada

ação. Assim, mesmo que a variável não mude seu valor, isto deve ser descrito usando a palavra reservada UNCHANGED.

A ação *Rcv* é permitida a partir de um estado em que é possível realizar-se um passo de recebimento. A condição para que esta ação seja habilitada é a definida na linha 14. Logo, se $rdy \neq ack$, sinalizando que houve um passo de envio, então pode haver um passo de recebimento. No recebimento, a única variável alterada é *ack*, que tem seu valor invertido (se 0, passa para 1 ou vice-versa), a fim de sinalizar o recebimento do dado (linha 15. A linha 16 apresenta que os valores de *val* e *rdy* não são modificados, já que os mesmo só são alterados quando há um passo de envio.

Como já dito, *Init* determina os possíveis estados iniciais da interface e *Send* e *Rcv*, as ações que devem ser realizadas em caso de passo de envio e recebimento. A fim de definir isto formalmente, tem-se na linha 17 a definição de *Next*, que representa a ação a ser executada no próximo estado, modificando o estado antigo. *Next* define que a ação a ser executada é a descrita por *Send* ou por *Rcv*. *Send* e *Rcv* não podem ser executadas paralelamente, sendo executada uma das duas ações a cada passo, isto devido a abordagem de *interleaving* adotada em TLA que define a execução alternada de ações. Assim, somente após o término de uma ação sendo executada é que outra ação poderá ter início.

A linha 18 define *Spec*. *Spec* descreve que os componentes válidos deste sistema são seqüências de estados nos quais, inicialmente, o predicado *Init* é verdadeiro e o predicado *Next* é sempre verdadeiro.

A linha 19 define um teorema (representado pela palavra reservada THEOREM) que afirma que, para todo comportamento que satisfaz o comportamento especificado em *Spec*, a definição *TypeInvariant* é sempre verdadeira. Ou seja, qualquer comportamento que satisfaça a especificação não deve atribuir valores às variáveis que não estejam entre os valores definidos em *TypeInvariant*.

2.2 Communicating Sequential Processes (CSP)

A linguagem CSP, proposta por Hoare em [8], é uma linguagem que permite a descrição de computação concorrente e distribuída. Um sistema distribuído é representado em CSP como um conjunto de processos que se comunicam por mensagens [12]. Um programa em CSP é um conjunto estático de processos, isto é, não é permitida a criação dinâmica de processos. Baseia-se na idéia de que, em uma linguagem que envolve concorrência, não se pode modelar a execução de um comando apenas como a transformação de um estado inicial para um estado final; é também necessário modelarem-se as interações contínuas que ocorrem com o meio enquanto um comando está sendo executado [9]. Assim, tem-se o conceito de *transição*. Uma transição de um processo é composta por um *estado inicial*, que corresponde ao estado do processo antes da transição, uma seqüência de interações entre o processo e o meio em que ele executa e um possível estado do processo após esta seqüência de interações. Os estados internos de um processo não são observáveis por seu ambiente.

A unidade fundamental no comportamento de um processo é um *evento*. Eventos são tratados como sendo instantâneos. O comportamento de um processo até um certo momento no tempo pode ser definido como a seqüência de todos os eventos dos quais ele participou, o que é chamado de *trace* do processo [9].

A comunicação entre processos dá-se através de suas identificações usadas em comandos de envio (*output*) e recebimento (*input*) de dados. Estas identificações são definidas pelo especificador. Assim, um processo identificado por *A* que envia dados a

um processo identificado por *B*, terá o comando de envio *B ! exp*. Este comando define que o processo identificado por *B* receberá o valor contido em *exp*. Para que este processo *B* receba o dado enviado pelo processo *A*, ele deve conter um comando de recebimento de dados *A ? val*. Este comando define que *B* receberá os dados do processo *A* e os guardará em *val*. O fluxo de comunicação é unidirecional, do transmissor para o receptor. A execução, tanto de um comando de envio como de recebimento, causa o bloqueio do processo que o executa até que o outro processo execute o comando complementar. É permitido o envio e recebimento de um número arbitrário de mensagens em um simples comando de envio ou recebimento. Com isso, podemos ter *B ! (x, y, z)*, o que significa que o processo *B* receberá, com um comando *A ? (x, y, z)*, três valores *x*, *y* e *z* do processo *A* com apenas um comando. A passagem de mensagens sem parâmetros, como em *A ? more()*, permite sincronização entre processos sem comunicação através de mensagens parametrizadas. Este tipo de mensagem é referido como *sinal* [12]. Deve-se salientar que, como a comunicação entre processos é feita utilizando-se identificações, é necessário que ambas as partes envolvidas conheçam a identificação uma da outra [10].

Comunicação concorrente pode ser modelada por *portas*, que servem como caixas de correspondências que recebem mensagens de um processo e de onde o processo destino da mensagem pode obtê-la. Cada porta é associada a apenas um processo transmissor e um processo receptor [12].

Os processos em CSP executam em paralelo. O operador || é usado para indicar processos que executam concorrentemente e são ditos *dependentes* (podem comunicar-se). Dessa forma, *A* || *B* define que *A* e *B* são concorrentes. Para definirem-se processos que executam concorrentemente e são *independentes*, i.e. processos que não interagem, usa-se o operador ||| [9]. Assim, *C* ||| *D* determina que *C* e *D* são concorrentes e independentes. Processos concorrentes não podem compartilhar variáveis. Isto facilita a utilização de CSP para ambientes de processamento distribuído [12].

CSP provê tipos de dados primitivos e estruturados. Os tipos estruturados compreendem registros (*records*) que podem possuir um nome e contêm campos que representam os valores do registro. Variáveis podem ser declaradas em qualquer ponto do programa.

Falha (*failure*) é um conceito importante em CSP, o qual usa falhas para identificação do término de um processo e para controle interno da execução de um processo [10]. Quando um processo executa um comando de recebimento de dados de um processo que terminou, o comando falha. Ocorre o mesmo se um processo executa um comando de envio de dados para um processo que terminou, causando a falha deste comando. Comandos condicionais em CSP tratam a falha como equivalente a um valor de falso (*false*). Em outro contexto, uma falha ocasiona o término do processo em que ela ocorreu.

CSP possui comandos de iteração, comandos com guarda e comandos seletivos. Um comando de iteração é o comando **[<test> → <action>]*, o qual corresponde a dizer-se "enquanto *test* for verdadeiro, execute *action*", sendo, portanto, similar a um comando *while-do*. O símbolo * especifica execuções repetidas do comando até que a condição *test* não seja verdadeira. CSP adota os comandos com guarda de Djikstra como a estrutura básica de controle. Comandos com guarda são comandos que possuem condição de guarda (*guarded clause*), a qual define a execução ou não do comando (comandos são não-determinísticos). Uma condição de guarda é uma série de expressões *booleanas*. Caso todas as expressões da condição de guarda sejam verdadeiras, o comando com guarda é executado. Dessa forma, $test_1; test_2;...;test_n \rightarrow$

cmd determina que *cmd*[1] é executado se $test_1, test_2,...,test_n$ forem verdadeiras. Comandos seletivos (*alternative commands*) são construídos concatenando-se comandos com guarda com símbolos □. Dessa forma, um comando seletivo é um tipo de comando com guarda. Neste caso, as condições de guarda são testadas e são executadas as ações correspondentes ao comando cuja condição de guarda é verdadeira, sendo ignorados os demais comandos. Um exemplo de comando seletivo seria $test_1 \rightarrow cmd_1 \square test_2 \rightarrow cmd_2$, onde cmd_1 ou cmd_2 seria executado dependo da avaliação das condições $test_1$ e $test_2$.

Uma possibilidade fornecida em CSP é a criação de um *array* de processos que executam um mesmo programa em paralelo. O tamanho do *array* deve ser uma constante definida em tempo de compilação. O *array* de processos pode ter múltiplas dimensões. Os processos de um *array* são referenciados pelo seu índice no mesmo.

Processos não podem ser recursivos, comunicando-se consigo mesmos. Isto é facilmente explicável pelo fato de que, como um comando de envio ou recebimento de dados deve ter um comando correspondente no outro processo comunicante, uma comunicação de um processo consigo mesmo poderia causar uma situação de *deadlock*. O uso de um *array* de processos permite criar-se o efeito de recursão, causando a comunicação entre dois programas iguais sendo executados por dois processos diferentes.

2.2.1 Exemplo de Especificação com CSP

A seguir tem-se, na Figura 4, a especificação CSP para a interface de comunicação assíncrona descrita em 2.1.1.

```
1.  SENDER ≡
2.      val : Data;

3.      *[ (Init ? ack() or Receiver ? ack()) →
4.              Receiver ! rdy();
5.              Receiver ! val
6.       ]

7.  RECEIVER ≡
8.      val : Data;


9.      *[ Sender ? rdy() →
10.             Sender ? val;
11.             Sender ! ack()
12.      ]


13. INIT ≡ Sender ! ack()

14. [Sender::SENDER || Receiver::RECEIVER || Init::INIT]
```

Figura 4. Especificação CSP para interface de comunicação assíncrona.

[1] *cmd* pode corresponder a uma série de comandos.

A especificação CSP é dividida em definições de programas a serem executados por processos. A definição de um programa é feita na especificação acima como ocorre na linha 1, tendo-se o nome associado ao programa descrito seguido do símbolo ≡ e dos comandos que compõem o programa definido. Assim, entre as linhas 1 e 6 tem-se a definição do programa associado ao nome *SENDER*. O símbolo ≡, portanto, determina que o nome *SENDER* é equivalente ao programa descrito. A linha 2 apresenta a declaração de uma variável *val* do tipo *Data*. As linha 3 a 6 apresentam um comando de repetição que faz com que o comando guardado descrito entre os colchetes (linhas 3 a 5) seja executado repetidas vezes. O comando guardado define que, caso seja recebido um sinal *ack* de um processo identificado por *Init* ou de um processo identificado por *Receiver* (linha 3), são executados os envios de um sinal *rdy* (linha 4) e do valor contido na variável *val* (linha 5) para o processo identificado por *Receiver*.

Entre as linhas 7 e 12 é definido o programa associado ao nome *RECEIVER*. Como acontece com *SENDER*, é declarada uma variável *val* do tipo *Data* (linha 8). Um comando de repetição é descrito entre as linhas 9 e 12. O comando repetido durante a execução deste comando é o comando guardado que determina que, caso seja recebido um sinal *rdy* de um processo identificado por *Sender*, é recebido um valor deste mesmo processo, o qual é armazenado em *val*, e é enviado um sinal *ack* também para o mesmo processo.

A linha 13 define o programa associado ao nome *INIT*, o qual realiza apenas o envio de um sinal *ack* para o processo identificado por *Sender*.

A linha 14 define a existência de um processo identificado por *Sender* que executa o programa *SENDER*, um processo identificado por *Receiver* que executa o programa *RECEIVER* e um processo identificado por *Init* que executa o programa *INIT*. Esta linha define o que pode ser chamado de um programa principal do sistema. O símbolos || entre as definições dos processos define que eles executam paralelamente.

O sistema descrito funciona da seguinte forma. Os três processos começam a executar ao mesmo tempo. *Init* envia um sinal *ack* para *Sender* e termina. Esse sinal é necessário para que *Sender* e *Receiver* não fiquem bloqueados nas condições de guarda de seus respectivos comandos, já que *Sender* espera por um sinal *ack* de *Receiver* e este espera por sinal *rdy* de *Sender*. Dessa forma, o sinal *ack* enviado por *Init* desbloqueia *Sender*, e este envia um sinal *rdy* para *Receiver*. *Receiver* é desbloqueado e espera por um valor para armazenar em *val*. *Sender* envia o valor constante em sua variável *val* para o *Receiver* e volta a ficar bloqueado, esperando por um novo sinal de *ack*, o qual, dado que *Init* foi finalizado, virá de *Receiver*. Ao receber o valor para armazenar em *val*, *Receiver* envia um sinal *ack* para *Sender*, desbloqueando-o e permitindo que ele envie um novo valor, recomeçando o processo de envio e recebimento descrito.

2.3 Specification and Description Language (SDL)

A *Specification and Description Language (SDL)* [13][15] é, como diz o próprio nome, uma linguagem de especificação e descrição de sistemas. Esta linguagem é padronizada pela *International Telecom Union (ITU)* e foi concebida para a especificação e descrição de sistemas de telecomunicações, sistemas distribuídos e sistemas embarcados [14]. Por isso, SDL foi, originalmente, uma linguagem utilizada por empresas de telecomunicações, mas já é hoje utilizada em vários setores industriais para descrição de sistemas de tempo real. SDL apresenta uma notação gráfica denominada SDL/GR e uma sintaxe textual padrão equivalente chamada de SDL/PR.

SDL apresenta vários tipos de construções para representar a descrição de um projeto, sendo que cada tipo de construção correspondente a uma *visão*. As diferentes visões são a visão arquitetural (*architecture view*), a visão da comunicação (c*ommunication view*), a visão comportamental (*behavior view*) e a visão da informação (*information view*). Estas diferentes visões formam um conjunto coerente e consistente que permite a descrição completa de um projeto [17] [19].

A arquitetura do sistema é representada por uma hierarquia de componentes. Em SDL, um sistema é composto por um conjunto de módulos interconectados chamados *blocos* (*blocks*). Blocos podem ser agrupados em novos blocos, criando uma estrutura de blocos aninhados [16]. Os blocos são divididos em *processos* (*processes*), os quais descrevem o comportamento do sistema.

SDL baseia-se em máquinas de estados finitas, logo o comportamento de cada processo é descrito como uma máquina finita de estados. A transição de estados ocorre quando o processo recebe um sinal. Durante a transição, é realizada uma seqüência de ações, que podem ser o envio de um sinal a outro processo, a manipulação de dados internos, etc. *Procedimentos (procedures)* são igualmente descritos por uma máquina de estados finita. Procedimentos podem ser recursivos e é suportada a chamada remota de procedimentos; ou seja, é possível realizar-se a chamada a um procedimento que executa no contexto de outro processo.

Blocos comunicam-se assincronamente entre si e com o ambiente através de *canais* (*channels*). Canais podem ser unidirecionais ou bidirecionais, sendo que um canal bidirecional pode ser considerado como dois canais unidirecionais independentes [12]. Os canais podem conter mensagens que são chamadas de *sinais* (*signals*). Um sinal é um pacote de informação que contém dados. Já a comunicação entre processos é estabelecida por vias de comunicação assíncrona denominadas *rotas de sinais* (*signal routes*). As rotas de sinais permitem realizar uma forma de multiplexação de canais. Isto é, um canal pode transmitir um grupo de sinais entre blocos e, dentro de cada bloco, um subgrupo deste grupo de sinais pode ser destinado, através de rotas de sinais, para cada processo. Dessa forma, podem-se ter diversas rotas de sinais que transmitem para processos sinais de um único canal.

A cada processo está associada uma *fila de recebimento* (*input queue*) de sinais, onde os sinais recebidos das diferentes rotas de sinais são agrupados. Os tipos de dados predefinidos e que podem ser transmitidos pelos sinais são *integer*, *boolean*, *real*, *character* e *time*, além dos tipos *string*, *powerset*, *array* e *struct*. Tipos abstratos de dados podem ser criados utilizando as definições dos tipos básicos. Tipos abstratos de dados possuem especificados um conjunto de valores possíveis, um conjunto de operações permitidas e um conjunto de axiomas que regulam a aplicação das operações [13]. Além de sinais assíncronos, SDL também suporta, como citado anteriormente, chamadas remotas de procedimentos, as quais são síncronas. O modelo de comunicação de SDL é independente da tecnologia de comunicação utilizada [18].

Os sinais que podem ser recebidos por um processo são especificados por *símbolos de entrada* (*input symbols*), sendo possível associar-se uma *condição de habilitação* (*enabling condition*) a cada símbolo de entrada. Dessa forma, um sinal associado a um símbolo de entrada só poderia ser recebido se a sua condição de habilitação fosse avaliada como verdadeira. Todos os sinais enviados a um processo são colocados na sua fila de recebimento, ordenados pela tempo de chegada, até que o processo alcance um estado no qual ele possa aceitar um sinal.

Em sua última versão, chamada SDL-2000, lançada em novembro de 1999, SDL passou a incluir também conceitos de modelagem orientada a objetos, suportando

encapsulamento, polimorfismo e herança simples (uma construção SDL pode herdar o comportamento de uma outra construção).

2.3.1 Exemplo de Especificação com SDL

Como ocorreu com CSP, será utilizada a especificação da interface assíncrona descrita em 2.1.1 como exemplo de especificação com SDL/GR. Esta especificação é apresentada na Figura 5 (em nível de bloco) e na Figura 6 (nível processo).

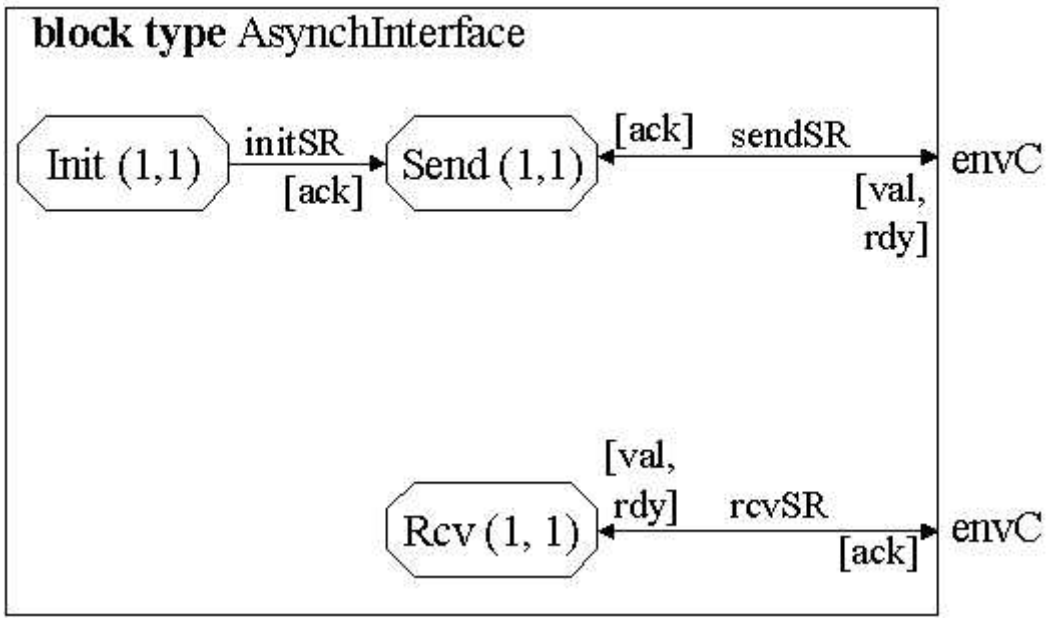

Figura 5. Especificação SDL para interface de comunicação assíncrona - nível bloco.

A Figura 5 apresenta a especificação em nível bloco. Para um sistema completo, dever-se-ia especificar o nível sistema, onde seriam descritos os canais, sinais e tipos de dados utilizados na especificação, bem como de que forma os canais ligam os blocos (simbolizados por retângulos) que compõem o sistema. Este nível não é apresentado aqui por interessar apenas a especificação da interface de comunicação que seria usada em um sistema. Logo, apresenta-se aqui a definição da interface como um bloco de SDL. A expressão *block type* antes do nome do bloco define que este bloco pode ser instanciado. Assim, poderiam existir várias instâncias de *AsynchInterface*. Os octógonos simbolizam os processos constantes no bloco. As cardinalidades *(m, n)* colocadas logo a seguir do nome de cada bloco definem que existem *m* instâncias daquele processo na inicialização do sistema e que um máximo de *n* instâncias podem existir durante a execução do mesmo. Dessa forma, para os processos definidos, existe sempre apenas uma instância que é criada na inicialização do sistema, sem haver criação dinâmica de novos processos. Caso não seja informado valor de *n* na cardinalidade, isto determina que não existe limite para a criação dinâmica de instâncias do processo ao qual ela está associada.

O escopo do bloco é definido pelo retângulo. Isto significa que *envC* determina um canal externo que introduz e recebe dados de processos do bloco. Neste caso, *envC* é multiplexado, podendo conter sinais destinados ao processo *Send* e/ou ao processo *Receive*. Ele também pode transmitir ao outro bloco ao qual ele está conectado sinais de qualquer um dos dois processos. Internamente ao bloco, a comunicação se dá através de

rotas de sinais, representadas por *initSR*, *sendSR* e *rcvSR*. Ou seja, os sinais externos trazidos pelo canal *envC* são transmitidos internamente através das rotas de sinais *sendSR* e/ou *rcvSR*. Os sinais que podem ser transmitidos por um canal ou rota de sinais são apresentados em colchetes. Ou seja, *[ack]* sobre um canal ou rota de sinais determina que o sinal *ack* pode ser transmitido por este canal ou rota de sinal. As setas nas pontas de canais e rotas de sinais apresentam a direção do fluxo de sinais. Assim, a rota de sinais *initSR* transmite sinais *ack* do processo *Init* para o processo *Send*, a rota de sinais *sendSR* transmite sinais *ack* de um processo conectado ao canal *envC* para o processo *Send* e sinais *val* e *rdy* deste último para o primeiro e a rota de sinais *rcvSR* transmite sinais *val* e *rdy* de um processo conectado ao canal *envC* para o processo *Rcv* e sinais *ack* no fluxo contrário.

Neste nível de bloco, como visto, tem-se uma visão mais detalhada da comunicação do sistema. Na especificação em nível de sistema ter-se-ia uma visão mais arquitetural do sistema. A visão de comportamento é apresentada no nível de processo, como mostra a Figura 6.

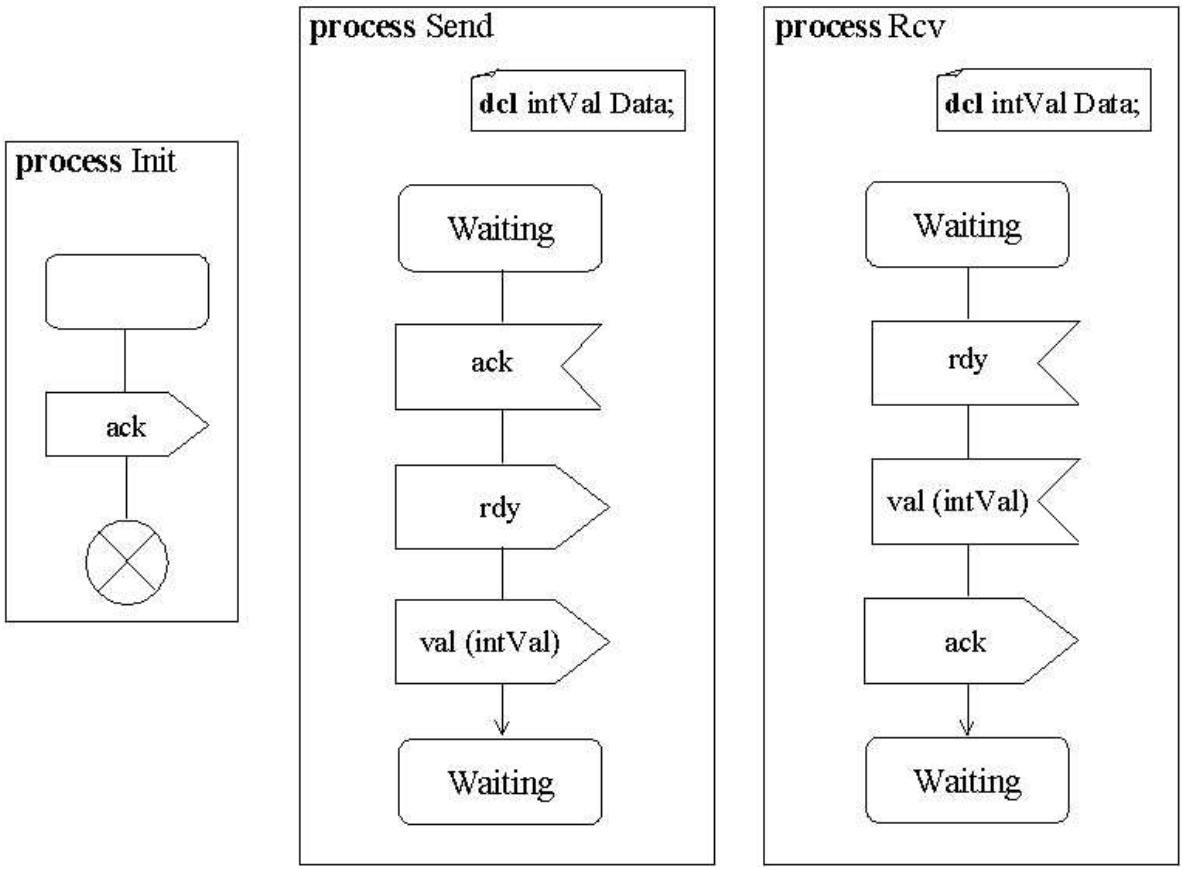

Figura 6. Especificação SDL para interface de comunicação assíncrona - nível processo.

Em nível de processo, os processos são definidos por retângulos dentro dos quais se apresenta o comportamento do processo. Os comportamentos são definidos por transições entre estados, como já relatado. Um estado é simbolizado por um retângulo de bordas arredondadas, contendo a identificação do estado em seu interior. Um estado inicial que não exige um sinal de entrada para realizar uma transição é um símbolo de estado sem a identificação do estado em seu interior. Como também já citado, a transição de estados ocorre através do recebimento de um sinal de entrada. Sinais de

entrada são simbolizados por retângulos com duas pontas do lado direito (como *ack* no processo *Send*). A transição de estado dispara ações por parte do processo que podem ser o envio de sinais de saída ou alterações no estado interno do processo. Sinais de saída são representados por retângulos com uma ponta na extremidade direita (como *rdy* no processo *Send*). O estado interno de um processo é mantido em variáveis declaradas no processo. Variáveis são declaradas dentro de símbolos representando notas com a ponta dobrada. Usa-se a palavra *dcl* para declarar uma variável, seguindo-se a esta o nome da variável e o seu tipo. No caso, declara-se em *Send* e em *Rcv* uma variável chamada *intVal* de tipo *Data*. Neste caso que tipo é *Data* não é relevante, mas ele deveria ser especificado no nível de sistema. Alterações nas variáveis internas do processo são representadas em retângulos cujo texto interno apresenta *variável := valor*, atribuindo-se *valor* a *variável*. O fim explícito de um processo é representado por um círculo com um **X** em seu interior.

Assim, o comportamento definido para o processo *Init* descreve que ele apenas envia um sinal *ack* e finaliza. Utilizando-se o que já foi visto no nível de bloco, sabe-se que este sinal é transmitido ao processo *Send*. O processo *Send* possui, como visto, uma variável interna. Ele inicia em um estado *Waiting* e permanece ali até receber um sinal *ack* que, como visto no nível de bloco, pode ser enviado por *Init* ou por *Rcv*. Ao receber este sinal, ele realiza o envio de um sinal *rdy* e do valor de sua variável interna *intVal* através do sinal *val* (que, como mostra a Figura 5, são enviados a *Rcv*). Neste caso, *val* é um sinal que carrega dados. Realizado isso, *Send* retorna ao estado *Waiting*. Já o processo *Rcv*, tal como *Send*, inicia em um estado *Waiting* e espera um sinal de entrada *rdy* (enviado por *Send*). Recebido o sinal, espera um sinal *val* contendo o valor a ser atribuído a sua variável interna *intVal* e depois envia um sinal de *ack* (para *Send*), voltando ao estado de *Waiting*.

Dessa forma, tem-se que o processo *Send* envia valores constantes em sua variável interna para *Rcv* que armazena estes valores em sua variável interna. Desconsideram-se aqui os valores transmitidos. O processo *Init* serve apenas para inicializar o sistema, impedindo que *Send* fique esperando por um sinal de *Rcv* e este por um sinal do primeiro.

2.4 Gramática de Grafos

Gramáticas de Grafos são uma generalização de gramáticas de Chomsky, substituindo as *strings* por grafos [20]. Diferente de regras em gramáticas de Chomsky, uma regra de grafos $r: L \rightarrow R$ não consiste somente dos grafos L (lado esquerdo) e R (lado direito), mas também de uma parte adicional: um mapeamento de vértices e arcos de L em vértices e arcos de R de maneira compatível. Assim, se um arco e_L for mapeado em um arco e_R, então o vértice origem de e_L deve ser mapeado para o vértice origem de e_R, ocorrendo o mesmo para o vértice destino. Gramáticas de grafos, segundo a abordagem algébrica [23], especificam um sistema em termos de estados (modelados por grafos) e mudanças de estados (modeladas por *derivações*) [1].

A interpretação operacional de uma regra $r: L \rightarrow R$, seguindo esta abordagem de especificação, é a seguinte:

- Itens de L que não têm imagem em R são *removidos*;
- Itens de L que são mapeados para R são *preservados*;
- Itens de R que não têm uma pré-imagem em L são *criados*.

O comportamento da especificação é determinado pela aplicação de regras aos grafos representando o estado real do sistema, partindo-se de um *grafo inicial*, no qual é representado o estado inicial do sistema. As aplicações de múltiplas regras podem ocorrer em paralelo se não houver conflito entre elas (duas ou mais regras não podem eliminar um mesmo item) . A aplicação de uma regra a um grafo *G* é chamada de *passo de derivação*. Um passo de derivação só é possível se existe uma ocorrência do lado esquerdo da regra no grafo *G* [4]. Ou seja, uma regra é aplicada somente se os elementos presentes no lado esquerdo das regras da gramática de grafos, ocorrem presentemente no grafo *G*. Além disso, pode-se adicionar outras condições para a realização de um passo de derivação. Uma regra, portanto, só é aplicada caso as condições relacionadas a ela sejam atendidas [21].

Uma restrição de Gramáticas de Grafos foi proposta em [20], onde foram definidas *Gramáticas de Grafos Baseadas em Objetos (GGBO)*. Esta abordagem considera a especificação de sistemas baseados em objetos. Os componentes desses sistemas são, como diz o nome, *objetos*, os quais são entidades autônomas que se comunicam e cooperam através da troca de *mensagens*. Os objetos possuem um estado interno não observável pelo ambiente e seu comportamento corresponde às reações executadas por ele ao receber mensagens. Múltiplas reações podem ser executadas paralelamente e podem gerar alterações no estado interno do objeto e/ou o envio de mensagens a outros objetos.

Seguindo esta abordagem, as principais entidades de GGBO são objetos e mensagens, que são representados como vértices dos grafos. Os arcos do grafo determinam as relações existentes entre as entidades. As entidades e as possíveis relações entre elas são definidas em um grafo chamado *grafo modelo*. Para cada aplicação baseada em objetos podem-se ter diversos tipos de mensagens e/ou objetos e, para representar estas entidades, define-se um *grafo de tipos da aplicação*. Neste grafo apresentam-se os tipos de mensagens e objetos específicos da aplicação em questão, sendo que o comportamento das entidades básicas (objetos e mensagens) mantém-se o mesmo, apenas sendo estendido para definir o comportamento particular de cada tipo de objeto e mensagem da aplicação. Por exemplo, caso se quisesse especificar uma aplicação envolvendo o controle de uma ferrovia, como descrito em [1] e [22], os objetos da aplicação poderiam ser trens e trilhos e as mensagens poderiam definir o avanço ou não dos trens através dos trilhos.

Haja vista a descrição apresentada, pode-se definir uma gramática de grafos como sendo composta por um grafo de tipos, um grafo inicial e um conjunto de regras.

2.4.1 Exemplo de Especificação com Gramáticas de Grafos

Como feito com as linguagens anteriores, usar-se-á a descrição da interface assíncrona apresentada em 2.1.1 para exemplificar o uso de Gramáticas de Grafos. Deve-se citar que neste exemplo serão utilizadas GGBO. Isto porque a especificação torna-se mais fácil e o uso de objetos, sendo o conceito de objeto bastante familiar, determina uma melhor compreensão da especificação.

A Figura 7 apresenta o grafo de tipos da especificação exemplo.

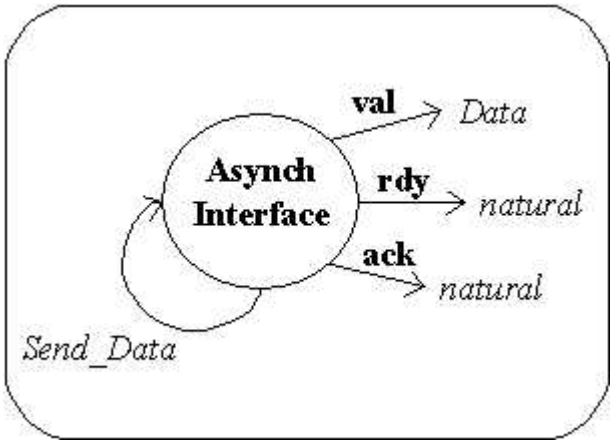

Figura 7. Grafo de tipos da especificação da interface assíncrona em GGBO.

Este grafo de tipos define que as entidades envolvidas na especificação são do tipo *AsynchInterface*. Uma entidade *AsynchInterface* possui os atributos *val* do tipo *Data*, *rdy* do tipo *natural* e *ack*, também do tipo *natural*. Uma *AsynchInterface* pode manter uma relação de *Send_Data* com outra *AsynchInterface*, definindo que uma entidade *AsynchInterface* pode enviar dados para outra.

A Figura 8 descreve o grafo inicial. O estado inicial do sistema apresenta duas entidades *AsynchInterface*: uma *Sender* e outra *Receiver*. *Sender* pode enviar dados para *Receiver*. O valores iniciais dos atributos das duas entidades são também apresentados no grafo inicial. Também existe, no grafo inicial, uma mensagem *Ack* para *Sender*. Esta mensagem serve para iniciar o sistema.

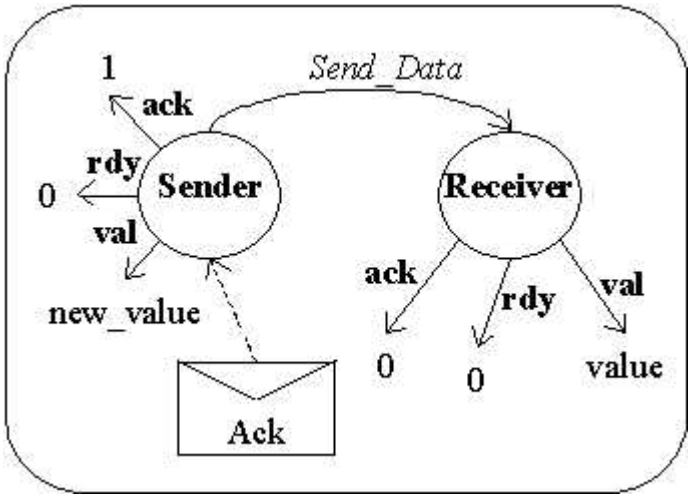

Figura 8. Grafo inicial da especificação da interface assíncrona em GGBO.

A Figura 9 apresenta o conjunto de regras da especificação. As regras são descritas por um lado esquerdo, que determina o grafo que deve existir no estado atual do sistema para que a mesma seja aplicada, uma seta que representa a transformação do grafo do lado esquerdo no grafo do lado direito, a qual apresenta o nome da regra e pode conter uma condição de aplicação desta, e um lado direito que determina o estado final do grafo após a aplicação da regra.

A primeira regra (*SendValue)* é aplicada sobre o estado inicial. Esta regra é, portanto, a primeira regra a ser aplicada. O resultado da aplicação desta regra é a

geração de uma mensagem *Value*, levando como argumento o valor *new_value* do atributo *val*, e de uma mensagem *Rdy*, ambas enviadas por *Sender* para *Receiver*. A aplicação da regra também causa a modificação dos valores de *rdy* e *ack*. Os valores destes atributos são invertidos apenas como forma de simbolizar que uma mensagem *Rdy* foi enviada e uma mensagem *Ack* foi recebida.

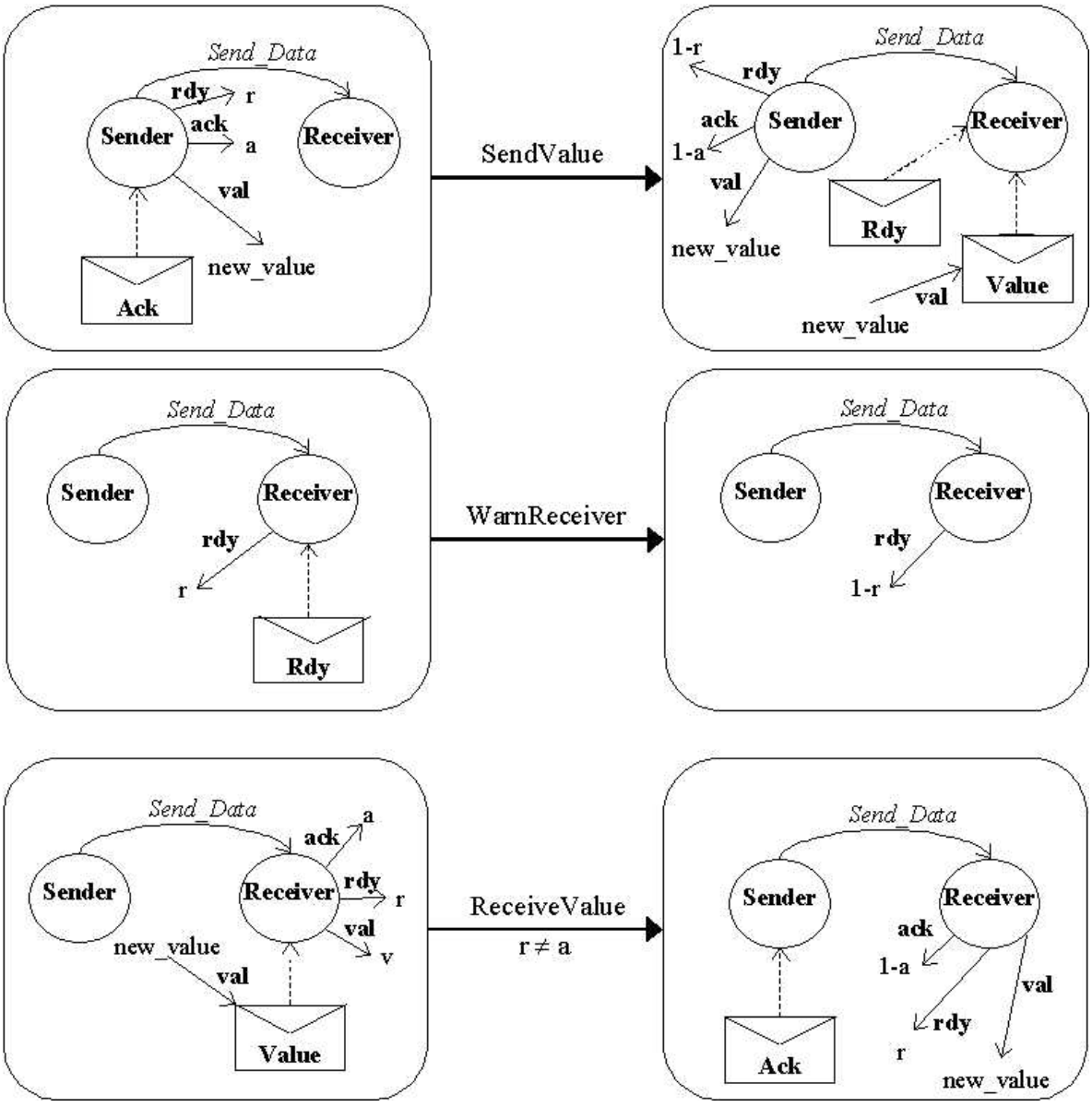

Figura 9. Conjunto de regras da especificação da interface assíncrona em GGBO.

A segunda regra (*WarnReceiver*) define que, tendo-se uma mensagem *Rdy* enviada para *Receiver*, esta regra é aplicada. A aplicação da mesma causa o consumo da mensagem e a alteração do atributo *rdy* de *Receiver*, simbolizando um sinal de aviso de que um novo valor foi enviado por *Sender*. Esta alteração inverte o valor *r* de *rdy*, tornando *r* diferente do valor *a* de *ack*.

A terceira e última regra (*UpdateSender*) define que, dada a situação da existência de uma mensagem *Value* enviada para *Sender* e, caso os valores *r* de *rdy* e *a* de *ack*

sejam diferentes, a regra é aplicada. A aplicação da regra causa o consumo da mensagem e a atualização do valor do atributo *val*, o qual recebe o valor recebido como argumento da mensagem. Além disso, o valor de *ack* é invertido e é gerada uma mensagem *Ack* para *Sender*, o que deixa o grafo com a configuração inicial.

3 Análise Comparativa das Linguagens

Nesta seção será apresentada uma análise comparativa das LEF vistas no capítulo anterior. Para realizar tal análise, serão estabelecidos critérios de avaliação e comparação das linguagens. Tais critérios são apresentados na seção 3.1. A seção 3.2 apresenta a análise de cada critério em relação às linguagens vistas e a seção 3.3 contém uma tabela com o resumo da análise realizada.

3.1 Critérios de Análise das Linguagens

Os critérios aqui utilizados correspondem ao que, neste trabalho, são consideradas como questões importantes quando se quer descrever um sistema distribuído. São analisados os critérios tidos como de maior importância e que mais podem influenciar a escolha por uma ou outra LEF por um desenvolvedor de sistemas distribuídos.

Os critérios a serem estabelecidos na análise são:

- **Representação de concorrência/paralelismo:** Sendo uma LEF para descrição de sistemas distribuídos, deve ser possível representar as idéias de concorrência/paralelismo, permitindo especificar-se, de forma clara, o que pode e o que não pode ocorrer simultaneamente[2] dentro do sistema descrito;

- **Abstrações oferecidas:** Cada LEF fornece algumas abstrações básicas com as quais o especificador pode trabalhar. A tais abstrações estão associados conceitos (orientação a objetos, orientação a processos, etc.) que definem os tipos das entidades com que se trabalha, bem como o comportamento assumido por elas;

- **Tipos abstratos de dados:** A LEF deve oferecer a possibilidade de trabalhar-se com tipos abstratos de dados, permitindo uma descrição mais completa dos tipos dos dados que serão manipulados pelo sistema descrito e dos tipos de dados que podem ser transmitidos entre as entidades do sistema;

- **Distribuição:** Obviamente, a questão da distribuição é essencial para uma LEF a ser utilizada para descrever sistemas distribuídos. Deve ser possível expressar-se a existência de distribuição;

[2] Paralelo, concorrente e simultâneo serão usados como sinônimos daqui por diante, apesar de, conceitualmente, não representarem a mesma coisa. Estes termos são aqui igualados apenas a fim de determinar se uma linguagem fornece ou não mecanismos que permitem simultaneidade de execução de componentes.

- **Comunicação:** A comunicação representa a forma de interação entre componentes de um sistema. Ela pode realizar-se síncrona ou assincronamente e localmente ou de forma distribuída. Os tipos de comunicação fornecidos por uma LEF, através de primitivas de comunicação, devem suprir as necessidades do desenvolvedor, preferencialmente, oferecendo comunicação síncrona e assíncrona, bem como comunicação local e distribuída;

- **Representação de comportamento do sistema:** Cada LEF deve prover uma forma de representar o comportamento assumido pelo sistema especificado, mostrando o que ocorre ou pode ocorrer durante a sua execução. A representação do comportamento de um sistema envolve descrever a evolução do estado do sistema desde a sua inicialização até uma possível finalização.

3.2 Análise das Linguagens

Dados os critérios estabelecidos na seção anterior, a seguir será discutido cada um deles em relação às LEF apresentadas na seção 2.

3.2.1 Representação de concorrência/paralelismo

Em TLA, o paralelismo ocorre implicitamente entre módulos. Não existe uma forma de demonstrar-se explicitamente que dois módulos ocorrem em paralelo. Além disso, não existe uma primitiva de comunicação que permita determinar-se a interação entre módulos de especificação, permitindo assumir-se que os módulos que interagem podem executar simultaneamente[3]. Apesar disso, pode-se subentender que dois módulos executam em paralelo por não haver qualquer restrição quanto a isso. Dentro dos módulos, porém, existem restrições quanto à execução de ações simultâneas. Isto porque TLA segue a abordagem de execução *interleaving*, determinando a execução alternada de ações. Logo, quando se escreve $Next == Send \vee Rcv$, está-se determinado que, a cada ocorrência de *Next*, ou *Send* ou *Rcv* será executada. Ou seja, não há possibilidade de as duas ocorrerem ao mesmo tempo.

Em CSP, como descrito na seção 2.2, a representação de concorrência é intrínseca a esta linguagem. Ao contrário do que ocorre em TLA, CSP possui primitivas de comunicação, o que já define a idéia de paralelismo através da possibilidade de interação entre dois processos. Mas, além disso, CSP provê uma forma de representação explícita de paralelismo e concorrência que é a utilização de símbolos $||$ e $|||$. O primeiro permite determinar que $P1 \,||\, P2$ significa que *P1* e *P2* executam paralelamente e são dependentes (podem interagir). Já $P1 \,|||\, P2$ afirma que os dois processos ocorrem em paralelo e de forma independente (não há interação entre eles).

SDL não apresenta uma forma explícita de representar concorrência/paralelismo, mas, como ocorre em TLA, pode-se assumir que existe simultaneidade. Dessa forma, blocos podem ocorrer em paralelo, bem como os processos contidos em um bloco também podem executar ao mesmo tempo. A existência de interação entre componentes

[3] Obviamente, a inexistência de uma primitiva de comunicação não determina a ausência de execução simultânea, pois pode haver execução concorrente sem interação (independente). Apenas considera-se que, existindo comunicação/interação, então é possível realizar-se execução simultânea.

é marcante em SDL, principalmente pela troca de sinais entre processos, o que torna claro o paralelismo entre estes componentes. Além disso, a possibilidade de transmissão de múltiplos sinais por um mesmo canal para diferentes rotas de sinais de diferentes processos, fornece a capacidade de ativação de transições em diversos processos simultaneamente.

Em Gramáticas de Grafos, a expressão de concorrência/paralelismo é implícita. A representação de simultaneidade se dá pela possibilidade de executar-se um conjunto de regras em um mesmo instante, desde que tais regras não conflitem. Ou seja, caso uma regra tenha suas condições de aplicação satisfeitas e não exista nenhuma regra executando ou para executar que conflite com ela, ela será executada. Logo, todas as regras que, em um dado instante, correspondam ao perfil descrito serão executadas concorrentemente. Caso haja conflito entre regras passíveis de execução, a escolha de qual das regras será executada é feita não-deterministicamente. A questão de conflito de regras é melhor esclarecida em 3.2.6.

3.2.2 Abstrações oferecidas

TLA oferece, como abstrações para a construção de suas especificações, módulos e ações. Os módulos podem ser vistos, por um certo ponto de vista, como classes que possuem atributos (variáveis e constantes) e podem herdar definições de outras classes. As ações seriam os métodos destas classes. Estas abstrações fornecem as idéias de modularidade e reusabilidade. A defecção de TLA é a ausência de uma abstração para comunicação, o que impede visualizar-se com clareza a interação entre módulos.

As abstrações de CSP são processos, o que facilita o trabalho com esta LEF visto que o conceito de processo é bastante familiar. Os processos em CSP também podem ser vistos como classes, sendo possível inclusive instanciarem-se processos que executam o mesmo programa. Abstrações para comunicação também são oferecidas, tais como as portas de comunicação, que podem ser vistas como caixas de correspondência. Além disso, o uso da identificação dos processos envolvidos em comandos de envio e recebimento de dados pode ser assumida como uma abstração de um canal associado a cada processo.

SDL possui uma hierarquia de abstrações com a existência de sistemas, blocos, processos e procedimentos. Sistemas são formados por blocos. Blocos são compostos por processos e procedimentos e podem conter outros blocos. Esta hierarquia torna possível visualizar-se um sistema em diversos níveis de abstração. Dessa forma, pode-se trabalhar em nível de sistema, descrevendo-se a relação entre os blocos que o compõem, em nível de blocos, mostrando os processos e suas relações e/ou em nível de processos, descrevendo o comportamento de cada processo e procedimento. Também existe a idéia de reuso de blocos, podendo-se criar instâncias de um bloco. O problema é que a existência de muitos níveis de abstração pode dificultar a identificação de em que nível certas coisas devem ser especificadas ou que tipo de abstração deve ser utilizada para representar uma parte do sistema especificado. Além disso, SDL fornece a abstração de sinais, usados como as mensagens que são transmitidas entre blocos e entre processos e que podem conter os tipos de dados existentes.

Em Gramáticas de Grafos, particularmente GGBO, as abstrações fornecidas são objetos e mensagens. Como os conceitos de orientação a objetos são bastante difundidos, existe facilidade em se trabalhar com estas abstrações. Também deve se considerar que, como tais abstrações são comumente encontradas em linguagens de programação conhecidas, sua utilização torna a especificação bastante compreensível e intuitiva.

3.2.3 Tipos abstratos de dados

O mecanismo de criação de tipos abstratos de dados está presente em TLA. Um exemplo é a idéia da criação de canais de dados apresentada em [4]. Um canal, por exemplo, poderia ter sido utilizado no exemplo apresentado em 2.1.1 para a controlar os valores de *val*, *ack* e *rdy*. Assim, poder-se-ia ter um canal *c* onde os sinais pudessem ser acessados como *c.val, c.rdy* e *c.ack*. Portanto, como se vê, pode-se criar estruturas de dados com campos de tipos de dados diferentes, que funcionam como tuplas de tipos híbridos.

Em CSP é possível construir-se um *array* de elementos dos tipos básicos, bem como de processos. CSP também suporta tipos estruturados sob a forma de registros, os quais podem conter variáveis dos tipos básicos que são acessadas como campos do registro.

SDL suporta a criação de tipos abstratos de dados a partir de tipos básicos e de outros tipos abstratos. Além disso, há suporte à criação de novos tipos de dados pelo desenvolvedor, permitindo a utilização dos mesmos em suas especificações. A dificuldade está na definição destes tipos novos, o que exige a especificação de valores possíveis, de operações que podem ser realizadas e do comportamento dos novos tipos através de axiomas. Ou seja, é necessário conhecer-se bem o meio de definição de tipos e todas as características do tipo que se está criando para defini-lo corretamente.

Gramáticas de Grafos permitem apenas a utilização de tipos básicos, tais como *natural*, *real* e *string*. Não há mecanismos para a criação de tipos abstratos de dados. Pode-se, no entanto, utilizar-se uma outra abordagem que forneça a criação de tipos abstratos de dados em conjunto com Gramáticas de Grafos, podendo-se utilizar nesta os tipos definidos.

3.2.4 Distribuição

Uma especificação em TLA não apresenta a idéia de distribuição. Assim como ocorre em relação à questão de concorrência/paralelismo, pode-se assumir a possibilidade de existência de componentes distribuídos, mas não é possível expressar isto claramente na especificação. Mesmo podendo-se subtender a execução de módulos em lugares diferentes, a inexistência de uma primitiva de comunicação dificulta a representação compreensível da interação entre eles. Como, pela falta de uma primitiva de comunicação, assume-se que se trabalha com variáveis compartilhadas, fica difícil trabalhar-se com a questão de variáveis compartilhadas por módulos que não executam no mesmo local, tendo-se uma arquitetura de memória compartilhada.

CSP não possui uma forma de expressão clara de distribuição, mas somente a existência de um mecanismo de comunicação que utiliza uma identificação única dentro do sistema já torna possível pensar-se em processos distribuídos. Dessa forma, tem-se a idéia de transparência de localização, sendo que apenas existe uma identificação do processo, através da qual ele se comunica. A comunicação, desta forma, pode ocorrer, implicitamente, tanto local como remotamente, o que permite dizer que os processos podem estar distribuídos.

Em SDL existe a especificação de chamadas de procedimentos remotos, o que já determina a existência de distribuição. Quanto aos outros componentes, não há como definir se eles atuam distribuidamente. Pode-se, no entanto, pensar nesses componentes como distribuídos. É possível dizer-se, por exemplo, que cada bloco é uma unidade de distribuição onde processos e procedimentos podem executar ou que cada processo de

um bloco executa em local diferente. Com isso, SDL não possui uma forma de expressar claramente a existência destes componentes de forma distribuída, mas também não impõe restrições a que isso ocorra segundo uma convenção do especificador. Logo, até mesmo pela estrutura hierárquica da especificação em SDL, esta LEF apresenta grande flexibilidade para adaptar-se a diversos tipos de modelos de distribuição, permitindo trabalhar-se com distribuição em nível de blocos ou em nível de processos ou em ambos.

Em GGBO, a comunicação entre objetos acontece através de mensagens. Não existe uma representação de distribuição mas, assim como ocorre com SDL e CSP, pode-se assumir que os objetos encontram-se distribuídos, trocando mensagens . Uma extensão para GGBO foi proposta em [20], na qual é possível representar-se a idéia de localização e, por conseqüência, tem-se uma representação de distribuição.

3.2.5 Comunicação

Em TLA, como já discutido nos itens anteriores, não existe primitiva de comunicação e trabalha-se como se as variáveis fossem compartilhadas pelos módulos de um sistema. Logo, TLA trabalha basicamente com uma arquitetura de memória compartilhada. No entanto, existe a possibilidade de criarem-se abstrações de comunicação a partir das construções da linguagem.

Ao contrário de TLA, CSP possui primitivas de comunicação. Tais primitivas utilizam as identificações dos processos para endereçar os dados transmitidos. Pode-se ver o uso das identificações de processos como o uso de um canal associado a cada processo e referido pela identificação do processo ao qual ele está associado. A necessidade de identificar-se explicitamente com qual processo outro processo se comunica pode dificultar a tarefa de especificação em CSP. Isto pode trazer o prejuízo de resultar em uma especificação que não seja tão genérica quanto seria desejado, haja vista que os processos que se comunicam devem ser previamente identificados. Estas primitivas funcionam de modo bloqueante, exigindo que haja um comando de recebimento do receptor para cada comando de envio do transmissor. Além disso, CSP provê também o conceito de portas, que podem trabalhar como caixas de correspondência, armazenando os dados recebidos por um processo. O uso de portas permite que um processo receba diversas transmissões de dados de diferentes processos concorrentemente, já que as portas são um mecanismo de comunicação não-bloqueante de CSP. Assim, CSP provê um mecanismo de comunicação bloqueante e outro não-bloqueante, ou pode-se dizer que CSP permite comunicação síncrona e assíncrona.

Em SDL a comunicação é explicitada pela descrição de canais entre blocos e rotas de sinais entre processos. Isto porque SDL fornece a chamada *visão de comunicação*. Dessa forma, é possível descrever claramente quais componentes podem se comunicar com quais componentes, além de ser possível saberem-se os tipos dos dados que serão transmitidos em cada canal e rota de sinal. Isto pode, como ocorre no caso de CSP, causar uma certa inflexibilidade da especificação, já que os canais e as rotas de sinais tem explicitamente os seus componentes comunicantes identificados. Por outro lado, esta explicitação das ligações entre componentes facilita a compreensão da especificação e torna o erro mais difícil, principalmente pela possibilidade de expressão gráfica fornecida pela linguagem. SDL fornece comunicação assíncrona e síncrona (através de chamadas remotas de procedimento).

Em GGBO as relações de comunicação não são explicitadas, sendo que o transmissor não precisa ser identificado. Ou seja, pode-se ter uma mensagem sem identificação do originador, o que torna a comunicação bastante flexível. Uma

mensagem possui um tipo e pode conter qualquer tipo de dado dentre os definidos na linguagem. O conteúdo da mensagem é determinado quando de seu envio, sendo definido através de argumentos passados para a mensagem, sendo que um mesmo tipo de mensagem possui sempre os mesmos tipos de argumentos.

3.2.6 Representação de comportamento do sistema

O comportamento de uma especificação TLA é descrito pela execução de ações. A execução de ações de um mesmo módulo segue a idéia de *interleaving*, ou seja, apenas uma ação ocorre em cada instante do tempo, tendo-se uma execução alternada de ações. Ações de módulos diferentes podem executar em paralelo. Logo, uma especificação determina que ações são executadas pelos módulos do sistema em cada momento dentro do tempo de execução deste.

Uma especificação CSP segue a idéia de seqüencialidade, sendo executado cada um dos comandos um após o outro. Esta seqüencialidade pode ser alterada pela existência de comandos de repetição e seleção. Dessa forma, um bloco de comandos pode ser executado repetidas vezes e pode-se determinar a seleção de comandos a serem executados, fazendo com que alguns comandos sejam ignorados.

Nas abstrações mais básicas dentro da hierarquia de SDL, os processos e procedimentos, o comportamento, segundo a *visão comportamental*, é descrito por uma máquina de estados finita. É realizada a transição entre os estados desta máquina pelo recebimento de sinais, o que dispara a execução de ações pelo processo ou procedimento, podendo causar emissão de sinais a outros processos, ocasionando uma transição de estados também nestes. Existe ainda a possibilidade de um processo, em um dado estado, realizar uma chamada a um procedimento, o qual também é descrito por uma máquina de estados finita. A execução de um procedimento causa o desvio da linha de execução do processo para o procedimento, retornando a execução para o processo quando o procedimento termina.

Em Gramáticas de Grafos, o comportamento da especificação é descrito pela aplicação das regras aos grafos que representam o estado atual do sistema. Como já falado no item 3.2.1, diversas regras podem ser aplicadas simultaneamente. Regras que possuam algum conflito (causando a eliminação de um mesmo componente, por exemplo) passam por um processo não-determinístico de escolha de qual delas será aplicada. Por isso, ao ocorrerem regras conflitantes, deve-se ter em mente que qualquer uma delas pode ser aplicada, sendo as outras ignoradas. A aplicação de regras pode causar a criação de novos componentes, a preservação de componentes já existentes ou a eliminação de componentes. Isto dá um caráter dinâmico a uma especificação em Gramáticas de Grafos, o que não ocorre nas outras LEF, onde os componentes que podem executar dentro do sistema são estaticamente definidos. O comportamento interno dos componentes é descrito dentro das regras através da apresentação das mudanças no estado interno (valores de atributos) do componente a cada aplicação de regra.

3.3 Resumo da Análise das Linguagens

De acordo com os critérios propostos e a análise desenvolvida sobre cada um deles em relação às LEF vistas, tem-se aqui a apresentação de uma tabela comparativa que resume a análise feita. Foi acrescido a esta tabela um critério de comparação o qual descreve se a LEF possui especificação textual ou gráfica. Todas as linguagens possuem especificação textual, mas apenas algumas têm especificação gráfica. Este critério não

foi incluído na seção anterior por não ser um critério exigido de uma LEF para sistemas distribuídos. Ele foi acrescido à tabela somente com uma informação a mais para a comparação, já que se pode ter a preferência por especificação textual ou gráfica. Para a comparação consideram-se Gramáticas de Grafos sob a abordagem de GGBO.

Os dados de comparação são apresentados na Tabela 1.

	Conc. / Paral.	**Abstrações oferecidas**	**Tipos abstratos**	**Distrib.**	**Comunic.**	**Comport.**	**Tipo**
TLA	Não	Módulos e ações	Sim	Não	Não	Execução de ações	Textual
CSP	Sim	Processos	Sim	Não	Sim (primitivas de comunic.)	Execução sequencial de comandos	Textual
SDL	Não	Blocos, processos e procedimentos	Sim	Não	Sim (canais e rotas de sinais)	Máquina de estados finita (nível processo)	Textual e Gráfica
GGBO	Não	Objetos	Não	Não	Sim (mensag.)	Aplicação de regras aos grafos do estado atual	Textual e Gráfica

Tabela 1. Comparação entre as linguagens apresentadas.

A interpretação da Tabela 1 é feita da seguinte forma: O item sobre *Concorrência/ Paralelismo (Conc./Paral.)* determina se existe alguma possibilidade de representação explícita de simultaneidade; o item *Abstrações Oferecidas* apresenta os tipos de abstrações fornecidos pela LEF; o item *Tipos Abstratos* define se existe a possibilidade criação de tipos abstratos de dados; *Distribuição (Distrib.)* informa se existe alguma forma de representação explícita de distribuição; *Comunicação (Comunic.)* descreve se existe alguma primitiva de comunicação; C*omportamento (Comport.)* informa o tipo de descrição de comportamento utilizado na LEF; e *Tipo* informa se a especificação é textual ou gráfica.

Cabe esclarecer que a questão de distribuição constante na tabela envolve, como dito, representação explícita de distribuição. Conforme discutido em 3.2.4, CSP, SDL e GGBO possibilitam que se pense em distribuição pela existência de comunicação entre as entidades, a qual pode ser local ou remota. Ou seja, implicitamente estas LEF permitem assumir-se a execução de entidades de forma distribuída sem, no entanto, ser possível mostrar-se isso claramente na especificação.

4 Conclusão

Este trabalho apresentou uma visão geral de linguagens de especificação formal de diferentes abordagens, descrevendo as características de quatro das linguagens representantes destas abordagens. Para cada linguagem foi apresentado um exemplo de especificação como forma de ilustrar as linguagens e permitir a comparação entre os estilos de especificação de cada uma delas.

Deve-se ressaltar que o problema usado na ilustração das linguagens foi retirado de [4], o qual provê um tutorial de TLA. Logo, é possível que tenha parecido ser mais fácil especificá-lo nesta linguagem. Mas o que pôde ficar claro com este trabalho é que um mesmo problema pode ser especificado segundo diferentes abordagens, sem, no entanto, estar incorreto. Cada linguagem pode ter uma abordagem diferente: TLA que possui a idéia de alteração de estados através da execução de ações que modificam os valores do estado; CSP segue o modelo de cálculo de processos, baseando o comportamento da especificação na execução de processos que interagem através de suas identificações; SDL que segue a idéia de *event-driven*, onde processos são descritos por máquinas de estados finitas que mudam de estado através da recepção de sinais; e GGBO que seguem a abordagem da execução de regras que modificam grafos que descrevem o estado do sistema. Apesar dessas diferenças, qualquer uma delas pode ser usada para descrever um sistema qualquer. A escolha por uma ou outra deve, entretanto, levar em consideração dois aspectos principais: o conhecimento do desenvolvedor da linguagem que será usada e o tipo de sistema a ser descrito. O primeiro aspecto diz respeito ao quanto o desenvolvedor estar habituado a lidar com uma certa linguagem. Isto é, uma especificação será melhor desenvolvida se o desenvolvedor souber trabalhar com os conceitos e abstrações da linguagem, conhecendo a semântica desta. O segundo aspecto compreende saber identificar que característica possui o sistema que se quer especificar, verificando-se qual abordagem utilizada nas linguagens fornece melhores recursos para descrevê-lo e quais possuem ferramentas disponíveis para auxiliar na especificação.

Para concluir, o que se pode afirmar é que a especificação formal de um sistema pode tornar mais fácil o processo de desenvolvimento de software, principalmente de sistemas distribuídos, permitindo corrigir-se erros de projeto antes da implementação. Mas, para que as vantagens propiciadas por ter-se uma especificação formal de um sistema sejam realmente obtidas, é necessário realizar-se um boa escolha quanto a melhor LEF a utilizar, de acordo com o tipo do sistema em questão. Isto porque uma escolha errada pode limitar a especificação, por causa de uma abordagem incompatível com o tipo de sistema, ou gerar um especificação incorreta do sistema devido à falta de conhecimento do desenvolvedor da linguagem com a qual ele está trabalhando.

5 Bibliografia

[1] DÈHARBE, D., MOREIRA, A. M., RIBEIRO, L., et al. Introdução a Métodos Formais: Especificação, Semântica e Verificação de Sistemas Concorrentes. *Revista de Informática Teórica e Aplicada*, v. 7, n. 1, Instituto de Informática - UFRGS, Porto Alegre, 2000, pp. 7-48.

[2] CLARKE, E., WING, J.. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, v. 28, n. 4, 1996, pp. 626-643.

[3] ASTESIANO, E., REGGIO, G.. Formalism and Method. In Proc. of TAPSOFT'97: Theory and Practice of Software Development, *Lecture Notes in Computer Science*, v. 1214, Springer, 1997, Lille, France, pp. 93-114.

[4] LAMPORT, L.. *Specifying Concurrent Systems with TLA*$^{+}$. Compaq, 1999.

[5] LAMPORT, L.. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, v. 16, n. 3, 1994, pp. 872-923.

[6] PNUELI, A.. The Temporal Logic of Programs. In *Proceedings of the 18*th *Annual Symposium on the Foundations of Computer Science*, IEEE, 1977, pp. 46-57.

[7] ABADI, M., LAMPORT, L., MERZ, S.. A TLA Solution to the RPC-Memory Specification Problem. Formal Systems Specification – the RPC-Memory Specification Case Study, *Lecture Notes in Computer Science*, v. 1169, Springer, 1996, pp. 21-65.

[8] HOARE, C. A. R.. *Communicating Sequential Processes*. Prentice Hall, 1985, 256 p..

[9] BROOKS, S. D., HOARE, C. A. R.. The Theory of Communicating Sequential Processes. *Journal of the ACM*, v. 31, n. 3, 1984, pp. 560-599.

[10] FILMAN, R. E., FRIEDMAN, D. P.. *Coordinated Computing Tools and Technologies for Distributed Software*. McGraw Hill, New York, 1984, 356 p..

[11] FIDGE, C.. A Comparative Introduction to CSP, CCS and LOTOS. *Technical Report Nº 93-24*, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994, 49 p..

[12] SPIRAKIS, P., TAMPAKAS, B., ANTONIS, K., et al.. *Specification Languages of Distributed and Communication Systems: State of The Art*. ALCOM-IT Report, Computer Technology Institute, University of Patras, Greece, 1996.

[13] ITU-T, ITU Recommendation Z.100. *The Specification and Description Language (SDL)*. ITU, Geneva, 2000.

[14] LEBLANE, P., OBER, I.. Comparative Case Study in SDL and UML. In *Proc. of 33rd Technology of Object-Oriented Languages and Systems (TOOLS 33)*, St. Malo, France, 2000, pp. 120-131.

[15] Web ProForum Tutorials. *Specification and Description Language (SDL)*. The International Engineering Consortium. Disponível na Internet em http://www.iec.org/tutorials/acrobat/sdl.pdf.

[16] SPITZ, S., SLOMKA , F., DÖRFEL, M.. SDL* - An Annotated Specification Language for Engineering Multimedia Communication Systems. In *Sixth Open Workshop on High Speed Networks*, Stuttgart, Germany, 1997. Disponível na Internet em http://www7.informatik.uni-erlangen.de/RP/pub.html.

[17] ENCONTRE, V.. SDL: A Standard Language for Ada Real-Time Applications. In *Proc. of Annual International Conference on Ada*, San Jose, USA, 1991, pp. 45-53.

[18] SHERRATT, E., LOFTUS, C.. Designing Distributed Services with SDL. *IEEE Concurrency*, USA, 2000, pp. 59-66.

[19] SALES, I., PROBERT, R.. From High-Level Behaviour to High-Level Design: Use Case Maps to Specification and Description Language. In *Anais do 18º Simpósio Brasileiro de Redes de Computadores*, Belo Horizonte, Brasil, 2000, pp. 457-471.

[20] DOTTI, F. L., RIBEIRO, L.. Specification of Mobile Code Systems Using Graph Grammars. *Formal Methods for Open Object-Based Distributed Systems IV*, Kluwer Academic Publishers, Stanford, USA, 2000, pp. 45-63.

[21] CORRADINI, A.. Concurrent Computing: From Petri Nets to Graph Grammars. Eletronic Notes in Theorical Computer Science 2, *Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation*, pp. 245-260.

[22] HOLZBACHER, A., PÉRIN, M., SÜDHOLT, M.. Modeling Railway Control Systems Using Graph Grammars: A Case Study. *INRIA Technical Report*, n. 3210, 1997, 24 p..

[23] EHRIG, H.. Introduction to the Algebraic Theory of Graph Grammars. *Lecture Notes in Computer Science*, v. 73, Springer, 1979, pp. 1-69.