Especificação Formal e Validação de um Kernel Paralelo Tempo Real

Cléver Ricardo Guareis de Farias

Centre for Telematics and
Information Technology (CTIT)
University of Twente (UT)
PO Box 217, 7500 AE, Enschede,
The Netherlands
E-mail: farias@cs.utwente.nl

Wanderley Lopes de Souza

Grupo de Sistemas Distribuídos e
Redes de Computadores (GSDR)
Departamento de Computação (DC)
Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676, 13565-905, São Carlos (SP)
E-mail: desouza@dc.ufscar.br

Abstract. This paper reports on a project where the Formal Description Technique (FDT) Language of Temporal Ordering Specification (LOTOS), a standard by International Organization for Standardization (ISO), was successfully used in the specification of an industrial operating system kernel for real-time applications. LOTOS was used on the specification of TRANS-RTXC, a real-time parallel kernel developed by Intelligent Systems International and currently commercialized as Virtuoso™ by Eonic Systems. The TRANS-RTXC LOTOS description developed in this project was validated for correctness by using the MiniLite toolset in a combinative approach of simulation and verification.

Resumo. Este trabalho apresenta os resultados de um projeto no qual a Técnica de Descrição Formal (TDF) Language of Temporal Ordering Specification (LOTOS), um padrão da International Organization for Standardization (ISO), foi utilizada com sucesso na especificação de um kernel industrial para aplicações de tempo real. LOTOS foi utilizada na especificação do TRANS-RTXC, um kernel paralelo tempo real desenvolvido pela Intelligent Systems International e atualmente comercializado como Virtuoso™ pela Eonic Systems. A descrição LOTOS do TRANS-RTXC, desenvolvida neste projeto, foi validada através de uma abordagem que combinou atividades de simulação e verificação, empregando-se o conjunto de ferramentas MiniLite.

Palavras Chave. Especificação, Validação, TDF, LOTOS, Sistemas de Tempo Real.

1. Introdução

O papel das Técnicas de Descrição Formais (TDFs) não está mais restrito ao projeto de protocolos de comunicação e sistemas distribuídos. Algumas de suas características, tais como abstração, expressividade e poder de análise, também podem ser empregadas para a produção de descrições completas, concisas e sem ambiguidades em outros domínios, tais como aplicações distribuídas com requisitos de tempo real.

Nos Sistemas de Tempo Real (STRs) a exatidão dos resultados depende não só da correção lógica de tais sistemas como também do tempo no qual esses resultados são obtidos. Para garantir que a implementação de um STR atenda aos requisitos pré-estabelecidos, uma abordagem de projeto precisa e sistemática faz-se necessária. Essa abordagem pode ser centrada na utilização de uma TDF, que servirá como base para todo o processo de desenvolvimento (especificação, validação, implementação e teste).

A TDF Language of Temporal Ordering Specification (LOTOS) [8] foi originalmente concebida para a especificação formal dos serviços e protocolos de comunicação do modelo de referência Open Systems Interconnection (OSI). Entretanto, os conceitos e construções de LOTOS são genéricos o suficiente para serem utilizados em outros tipos de sistemas distribuídos. Além disso, diversos tipos de extensões para LOTOS foram concebidas nos últimos dez anos, sendo que algumas delas foram consideradas por um grupo da ISO e

incorporadas a uma nova proposta de TDF, denominada Enhancements to LOTOS (E-LOTOS) [9], cuja padronização está prevista para o início de 1999.

Há muita controvérsia em relação à necessidade de estender LOTOS. Os que são a favor advogam que essa TDF poderia ser utilizada para a especificação de uma variedade maior de sistemas, desde que certas características ausentes, tais como, tipos concretos de dados, conceitos de módulo, interface e, sobretudo, tempo, fossem supridas. Os que são contra, advogam que essa TDF já atende a uma boa variedade de sistemas e que se, por um lado, o aumento do poder de expressão de uma linguagem aumenta o escopo de suas aplicações, por outro lado, dificulta também o seu aprendizado podendo vir a inibir potenciais usuários (LOTOS já é bastante complexa pois envolve dois tipos de álgebras).

A motivação principal para o desenvolvimento desse projeto foi a de avaliar se LOTOS padrão, sem qualquer tipo de extensão, é adequada para a especificação de certas partes dos STRs. Em particular, a atenção esteve voltada para o kernel, uma vez que a preocupação principal dessa parte é com o gerenciamento de tarefas de uma aplicação com requisitos de tempo real e não com os aspectos temporais em si. Geralmente, tais requisitos são tratados pela própria aplicação e ferramentas, tais como analisadores temporais, analisadores de escalonabilidade e depuradores [11], estão disponíveis para o projetista da aplicação lidar com essas restrições temporais.

Neste sentido, LOTOS foi empregada na especificação do kernel paralelo tempo real TRANS-RTXC. Este kernel foi desenvolvido pela Intelligent Systems International, mas atualmente está sendo comercializado como Virtuoso™ pela Eonic Systems. O TRANS-RTXC tem sido utilizado no desenvolvimento de várias aplicações industriais de tempo real, tais como satélites, sistemas militares, sistemas de acompanhamento médico e sistemas de radares. Alguns resultados preliminares desse projeto podem ser encontrados em [4, 5].

Este artigo está estruturado da seguinte forma: a seção 2 apresenta uma visão geral do contexto da especificação desenvolvida para o TRANS-RTXC, enquanto que a seção3 apresenta a especificação propriamente dita. A seção 4 discute como esta especificação foi validada, enquanto que a última seção apresenta as conclusões e perspectivas futuras para esse projeto.

2. Uma Máquina Paralela Genérica

A fim de fornecer uma visão geral do contexto no qual esse projeto está inserido, esta seção apresenta uma especificação LOTOS de uma máquina paralela genérica para aplicações de tempo real.

Tal máquina pode ser geralmente definida como um sistema composto por múltiplos processadores, dispositivos de entrada/saída e *links* de comunicação, que fazem a conexão entre os processadores. Por sua vez, cada processador é formado por um hardware específico (contendo CPU, dispositivos de armazenamento, *links* de comunicação, temporizadores, etc), por um kernel paralelo e por um conjunto de tarefas da aplicação. O kernel paralelo tem como função gerenciar os recursos de hardware, provendo assim uma interface amigável para o desenvolvimento de uma aplicação de tempo real.

Uma especificação LOTOS foi desenvolvida, utilizando o estilo orientado a recursos, para a modelagem dessa máquina paralela genérica em termos de seus componentes físicos. O propósito dessa especificação é somente prover um melhor entendimento do inter-relacionamento e da estruturação dos componentes da máquina paralela. Assim sendo, essa especificação não é completa, pois descreve apenas a estruturação dos componentes básicos do sistema e não o comportamento dos mesmos. Cada componente está representado por uma instanciação de processo *vazia*, isto é, sem a definição do seu comportamento.

A Figura 1 apresenta a especificação LOTOS de uma máquina paralela genérica. Nessa especificação, *ParallelMachine* está estruturada como a composição paralela dos processos *Devices*, representando os dispositivos de entrada/saída de dados da máquina, *Processors*, representando os vários processadores presentes na máquina, e *CommunicationLinks*, representando os *links* de comunicação utilizados para prover a conexão entre esses processadores. A comunicação entre *Processors* e *Devices* é realizada através da porta *IO*, enquanto que a comunicação entre *Processors* e *CommunicationLinks* é realizada através da porta *communicationlinks*.

```
SPECIFICATION ParallelMachine[init,dev]: noexit

(* ADTs definition *)

BEHAVIOUR

  HIDE IO, communicationlinks IN
   init ?number_of_processors: Nat ?number_of_links: Nat;
    ( Devices[IO, dev]
      |[IO]|
      Processors[IO, communicationlinks, init](number_of_processors, number_of_links)
      |[communicationlinks]|
      ComunicationLinks[communicationlinks](number_of_links)
    )

  WHERE

    (* Definition of the processes Devices and CommunicationLinks *)

   PROCESS Processors[IO, communicationlinks, init](number_of_processors: Nat,
                        number_of_links: Nat): noexit:=

   init !number_of_processors ?routingtable: RoutingTable;
    ([number_of_processors gt 0] ->
      (   SingleProcessor[IO, communicationlinks](...)
       ||| Processors[IO, communicationlinks, init](...)
      )
    [] [number_of_processors le 0] ->
       SingleProcessor[IO, communicationlinks](...)
    )

   WHERE

   PROCESS SingleProcessor[IO, communicationlinks](number_of_processor: Nat,
                              number_of_links: Nat, rtable: RoutingTable): noexit:=

    HIDE rtxcint, chanin, chanout, kernelmemoryaccess, rtxcinports, rtxcoutports, tasksmemoryaccess IN
       ApplicationTasks[tasksmemoryaccess, rtxcint](...)
       |[tasksmemoryaccess, rtxcint]|
       ParallelKernel[kernelmemoryaccess, rtxcint, chanin, chanout, rtxcinports, rtxcoutports](...)
       |[kernelmemoryaccess, rtxcinports, rtxcoutports]|
       Hardware[IO, communicationlinks, tasksmemoryaccess, kernelmemoryaccess, rtxcinports,
                rtxcoutports](...)

      WHERE

       (* Definition of the processes ApplicationTasks, ParallelKernel, and Hardware *)

    ENDPROC (* SingleProcessor *)

ENDSPEC (* ParallelMachine *)
```

Figura 1. Especificação LOTOS de uma máquina paralela.

A porta *init* é utilizada para a ativação do sistema. Através dessa porta é fornecido o número de processadores e *links* de comunicação disponíveis no sistema. Estes dados são passados como parâmetros para *Processors*, que também recebe através da porta *init* a tabela de roteamento relativa a um determinado processador. *Processors* está estruturado como a composição em paralelo sem sincronização de *SingleProcessor* com o próprio *Processors*. Através dessa estruturação, a existência simultânea de um número qualquer de processadores torna-se possível.

SingleProcessor representa um único processador da máquina paralela. Esse processo está estruturado como a composição em paralelo de *ApplicationTasks*, representando as tarefas da aplicação, *ParallelKernel*, representando o kernel paralelo tempo real do processador, e *Hardware*, representando os vários componentes físicos do sistema, tais como: CPU, registradores, *links* de comunicação, dispositivos de armazenamento (memória) e temporizadores.

A comunicação entre um processador e o seu ambiente (dispositivos de entrada/saída de dados e demais processadores) é realizada através das portas *IO* e *communicationlinks*, respectivamente. A configuração do sistema determina se os *links* de comunicação do processador, representados pelas portas *rtxcinports* (entrada de dados) e *rtxcoutports* (saída de dados), estão conectados aos dispositivos do sistema ou à rede de processadores.

A comunicação entre as tarefas da aplicação e o kernel paralelo é realizada através das portas *rtxcint*, *chanin* e *chanout*. Chamadas às funções providas pelo kernel são realizadas através da porta *rtxcint*, enquanto que solicitações para a transferência de dados (entrada e saída), entre o processador e os dispositivos da máquina paralela, são realizadas através das portas *chanin* e *chanout* respectivamente. A transferência de dados entre os processadores é realizada somente através de chamadas às funções do kernel.

O acesso à memória do processador é realizado através das portas *kernelmemoryaccess*, que conecta o kernel aos dispositivos de armazenamento, e *tasksmemoryaccess*, que conecta as tarefas da aplicação aos dispositivos de armazenamento.

3. Especificação LOTOS do TRANS-RTXC

O TRANS-RTXC é um kernel multitarefa, que executa em uma máquina paralela, e possui um escalonador preemptivo. Esse kernel é constituído de uma coleção de *threads* paralelos de alta prioridade, cada qual com uma função específica. A comunicação entre esses *threads* e entre os *threads* e as tarefas da aplicação é realizada através da troca de mensagens, utilizando-se os canais de comunicação da máquina paralela, e através de memória compartilhada. A Figura 2 ilustra uma arquitetura para o TRANS-RTXC baseada no seu conjunto de *threads*.

O TRANS-RTXC provê um conjunto padrão de estruturas para a construção de aplicações para tempo real, tais como tarefas, semáforos, filas, mensagens, recursos, blocos de memória e temporizadores. Cada estrutura possui um conjunto de chamadas do kernel para a sua manipulação. Maiores informações sobre o TRANS-RTXC podem ser encontradas em [14].

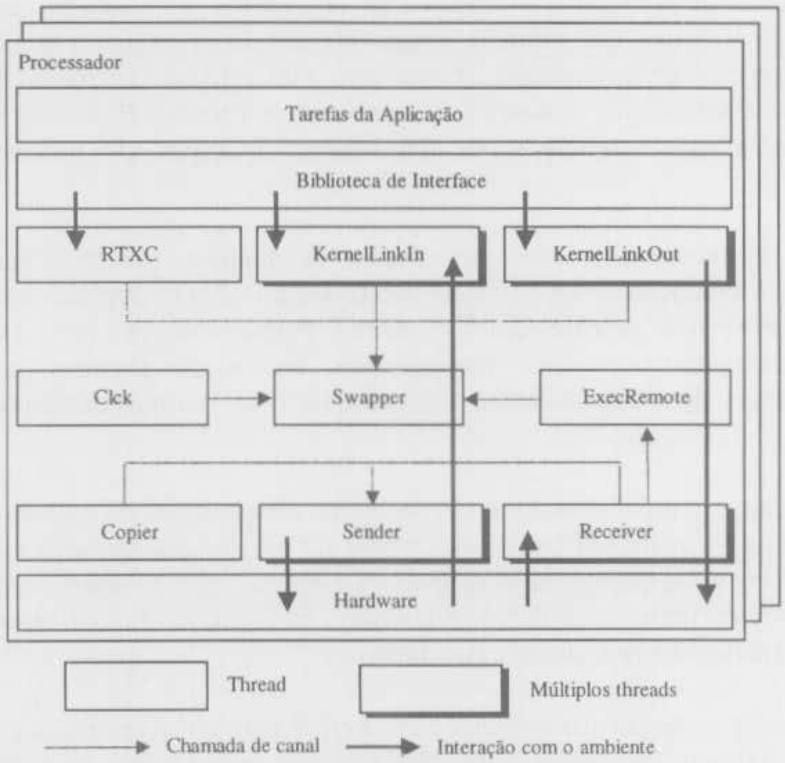

Figura 2. Arquitetura do TRANS-RTXC em um processador.

3.1. Metodologia de Desenvolvimento

Em se tratando de sistemas complexos, alguns princípios genéricos de projeto devem ser utilizados para nortear o projetista, aumentando assim a confiabilidade e a qualidade do sistema a ser desenvolvido. Em [15], três princípios qualitativos de projeto são apresentados: *ortogonalidade*, segundo o qual requisitos arquitetônicos independentes devem ser especificados através de definições independentes; *generalidade*, onde definições de propósito gerais, isto é, que possam ser reutilizadas, são preferíveis às definições de caráter específico; e *extensibilidade*, onde uma flexibilidade deve ser conferida ao projeto, de tal forma que se permita uma fácil expansão e modificação.

O desenvolvimento da especificação LOTOS do TRANS-RTXC partiu de sua descrição informal. O sistema foi construído incrementalmente, numa sequência de passos de projeto, até a obtenção de uma especificação final que preenchesse os requisitos do mesmo. O primeiro passo do projeto produziu a descrição mais abstrata do TRANS-RTXC. Cada um dos passos posteriores resultou em uma descrição mais completa e mais detalhada. As principais decisões utilizadas em cada passo de projeto foram baseadas nos princípios qualitativos acima descritos.

Vários estilos de especificação para suportar os princípios qualitativos de projeto foram definidos para LOTOS [15]. Estes estilos de especificação podem ser vistos como ferramentas que utilizam de maneira eficiente os conceitos e construções de LOTOS, guiando o projetista ao longo de uma trajetória de projeto que parte da análise dos requisitos dos usuários e termina com a implementação do sistema.

No desenvolvimento desta especificação foram utilizados principalmente dois estilos de especificação: o estilo orientado a recursos, para uma estruturação da mesma em termos de componentes, e o estilo orientado a restrições, para uma especificação abstrata desses componentes.

3.2. Especificação Formal

Nesta seção é demonstrado como LOTOS foi utilizada na especificação do conjunto de *threads* do TRANS-RTXC. A especificação produzida possui aproximadamente 3300 linhas de código LOTOS (incluindo comentários), dentre as quais 1400 linhas (42%) foram utilizadas na especificação dos Tipos Abstratos de Dados (TADs) e 1900 linhas (58%) foram utilizadas na especificação das expressões comportamentais. A especificação completa pode ser encontrada em [6].

As estruturas encontradas no TRANS-RTXC e definidas na especificação LOTOS desse kernel foram: tarefas, mensagens, temporizadores, semáforos e recursos lógicos. Estruturas representando filas e estruturas para o gerenciamento de memória não foram especificadas para efeitos de simplificação.

Essas estruturas foram especificadas através de TADs. Essa decisão de projeto foi baseada no fato de que a especificação das estruturas do kernel através TADs, e não através de processos LOTOS, confere uma maior flexibilidade na definição e manipulação das mesmas. Operações básicas podem ser definidas sobre as estruturas e facilmente podem ser combinadas, obtendo-se assim operações mais complexas. Outra vantagem a se destacar é a possibilidade de passar essas estruturas como parâmetros para os processos, ou através das interações que ocorram nas portas LOTOS da especificação.

A primeira influência que a especificação dos TADs sofreu por parte dos princípios de projetos foi com relação à generalidade. Deu-se preferência à especificação de um tipo parametrizado através de elementos formais, à especificação de múltiplas especializações de um mesmo tipo. Exemplos da aplicação desta política incluem a definição estruturas parametrizadas representando filas padrões e com prioridades.

Durante a especificação dos TADs, algumas vezes éramos forçados a nos decidir entre a especificação de uma estrutura única e abrangente ou a especificação de duas ou mais estruturas especializadas. Nestas situações, a aplicação do princípio da ortogonalidade foi decisiva na escolha da segunda alternativa. Como exemplo da aplicação deste princípio, destaca-se a definição de uma estrutura representando um pacote a ser enviado pela rede. Dado que o pacote poderia conter tanto dados quanto comandos, optamos pela definição de duas estruturas distintas: uma para o transporte de dados e outra para o transporte de comandos.

A principal aplicação do princípio da extensibilidade, nesta fase da especificação, ocorreu na definição dos TADs relacionados com as estruturas do kernel. Um cuidado especial foi dedicado à especificação destes TADs, de tal forma que a modificação de alguma característica destas estruturas ou mesmo a adição de novas estruturas, tais como filas e blocos de memória, possa ser feita sem alterações significativas nas definições dos demais TADs.

Na parte comportamental da especificação, um *thread* é representado por um processo LOTOS, enquanto que um canal de comunicação é representado por uma porta LOTOS. A Figura 3 apresenta o primeiro nível da especificação LOTOS do TRANS-RTXC. Esta arquitetura foi obtida baseando-se no conjunto de *threads* deste kernel. Contudo, neste primeiro nível da especificação *threads* com funções correlatas foram agrupados em um mesmo processo LOTOS. Nas próximas subseções, apresentaremos os aspectos mais relevantes da especificação de cada processo definido nessa especificação abstrata.

```
SPECIFICATION ParallelKernel[memoryaccess, rtxcint, chanin, chanout, rtxcinports, rtxcoutports](...): noexit
 (* ADTs definition *)
BEHAVIOUR
  HIDE swapping, copying, serving, sending, operationexecution IN
    memoryaccess ?taskQueue: TCBQueue ?kernelSemaphores: SemaList ?kernelResources: RheaderList;
    ( Swapper[swapping, memoryaccess](...)
      |[swapping]|
      PerformKernelCalls[operationexecution, copying, sending, swapping, memoryaccess, rtxcint](...)
      |[operationexecution, copying]|
      CommandServerProc[rtxcint, serving, swapping, operationexecution]
      |[serving]|
      Copier[copying, sending, memoryaccess, swapping](...)
      |[sending]|
      LinkServerProc[memoryaccess, rtxcinports, rtxcoutports, chanin, chanout, serving, sending, swapping](...)
      |||
      Clck[swapping, operationexecution](...) )
  WHERE
    (* Definition of the processes Swapper, PerformKernelCalls, CommandServerProc, LinkServerProc, and Clck *)
ENDSPEC (* ParallelKernel *)
```

Figura 3. Primeiro nível da especificação LOTOS do TRANS-RTXC.

3.1. *Swapper*

Uma tarefa no TRANS-RTXC é representada pelo seu Bloco de Controle de Tarefa (BCT). O BCT de uma tarefa foi definido como um TAD contendo todas as informações necessárias para o controle e manipulação da mesma.

Swapper pode ser considerado como o processo mais importante da especificação, uma vez que é responsável pela manipulação da fila de tarefas da aplicação (fila de BCTs) e pela decisão, baseando-se em uma política de escalonamento, de quais tarefas deverão ser escalonadas.

A política de escalonamento do kernel pode variar de acordo com as características da aplicação envolvida. É possível especificar e utilizar políticas com prioridades estáticas e dinâmicas. Para tanto, basta utilizar definições apropriadas para as operações e equações do TAD que representa a fila de tarefas da aplicação.

Na especificação formal do TRANS-RTXC foi utilizada uma política preemptiva baseada em prioridades estáticas. Segundo essa política, cada tarefa da aplicação recebe uma prioridade que se mantém inalterada. As tarefas prontas para serem executadas são armazenadas em uma fila (fila de prontos), de acordo com suas prioridades, e manipuladas através de um conjunto específico de operações. A tarefa em execução possui sempre prioridade igual ou superior às demais tarefas armazenadas na fila de prontos.

Se a tarefa em execução deve esperar pela ocorrência de um evento, tal como o fim de uma operação de entrada e saída ou o fim de uma transferência de dados, esta será bloqueada e transferida para uma fila de eventos. As tarefas bloqueadas são mantidas nessa fila até que os eventos desejados ocorram.

As mensagens enviadas para uma tarefa são armazenadas em sua caixa postal, localizada no seu BCT. Em função das características de LOTOS, foi decidido restringir a manipulação dos BCTs das tarefas da aplicação a um único processo (*Swapper*). Desta forma, este também é responsável pela inserção/remoção de mensagens nos BCTs das tarefas.

Todas estas restrições foram definidas como uma coleção de alternativas e agrupadas em três categorias: troca de contexto, bloqueio/liberação de tarefas e inserção/remoção de mensagens.

3.2. *PerformKernelCalls*

A principal função do *PerformKernelCalls* é a execução das chamadas do TRANS-RTXC. Como esse processo também é responsável pela manipulação e armazenamento do conjunto de semáforos e recursos lógicos do kernel, este foi refinado em três subprocessos comunicantes (princípio da ortogonalidade): *PerformFunctions*, responsável pela execução das chamadas do kernel, *SemaphoresManagement*, responsável pelo armazenamento e manipulação dos semáforos, e *ResourcesManagement*, responsável pelo armazenamento e manipulação dos recursos lógicos. A Figura 4 apresenta o refinamento do processo *PerformKernelCalls*.

Influenciados pelo princípio da extensibilidade, modelamos a execução das chamadas do kernel através de um conjunto de alternativas. Esta estruturação é especialmente adequada se levarmos em consideração o fato de que chamadas, bem como as estruturas providas pelo kernel, podem ser adicionadas ou removidas em função das características da aplicação envolvida.

```
PROCESS PerformKernelCalls[operationexecution, copying, sending, swapping, memoryaccess,
          rtxcint](processornumber: Nat, semaphores: SemaList, resources: RheaderList): noexit:=
 HIDE semaccess, resourceaccess IN
   PerformFunctions[operationexecution, semaccess, resourceaccess, copying, sending,
                    swapping, memoryaccess, rtxcint](processornumber)
   |[semaccess, resourceaccess]|
   SemaphoresManagement[operationexecution, semaccess](semaphores)
   |||
   ResourcesManagement[resourceaccess](resources)
 WHERE
   (* Definition of the processes PerformFunctions, SemaphoresManagement, and
      ResourcesManagement *)
ENDPROC (* PerformKernelCalls *)
```

Figura 4. Refinamento do processo *PerformKernelCalls*.

3.3. *CommandServerProc*

O TRANS-RTXC provê um conjunto de chamadas (comandos) para a manipulação de suas estruturas. Uma chamada pode resultar em uma operação local (chamada local) ou remota (chamada remota). Na ocorrência de uma operação remota, o kernel disponibiliza um buffer, insere os parâmetros adequados nesse buffer e o envia através dos *links* da rede. Dependendo da operação solicitada, a tarefa responsável pela chamada pode ser bloqueada até que uma resposta seja recebida. Quando um buffer é recebido através dos links da rede, a chamada nele contida é executada da mesma forma que uma chamada local.

CommandServerProc representa os *threads* servidores de chamadas do TRANS-RTXC, sendo definido como a composição paralela de dois subprocessos independentes (princípio da ortogonalidade): *Rtxc* e *ExecRemote*. Esses subprocessos lidam com as chamadas locais e remotas respectivamente. A Figura 5 apresenta os principais aspectos da especificação LOTOS de *CommandServerProc*.

Rtxc é o servidor das chamadas locais do kernel. Quando uma chamada local é recebida, este processo executa as seguintes ações: decodifica a chamada, provê a execução da chamada decodificada e analisa os resultados dessa execução. Para atender a esses requisitos, e baseados no princípio da ortogonalidade, refinamos o *Rtxc* em dois subprocessos comunicantes: *ReceiveLocalCall* e *HandleExecutionResults*.

```
PROCESS CommandServerProc[rtxcint, serving, swapping, operationexecution]: noexit:=
        Rtxc[rtxcint, swapping, operationexecution]
   |||  ExecRemote[serving, swapping, operationexecution]
 WHERE
 PROCESS Rtxc[rtxcint, swapping, operationexecution]: noexit :=
   HIDE transmitresults IN
     ReceiveLocalCall[rtxcint, operationexecution, transmitresults]
     |[transmitresults]|
     HandleExecutionResults[swapping, transmitresults]
   WHERE
   (* Definition of the processes ReceiveLocalCall and HandleExecutionResults *)
 ENDPROC (* Rtxc *)
 PROCESS ExecRemote[serving, swapping, operationexecution]: noexit :=
   HIDE transmitcall, transmitresults IN
     ReceiveRemoteCall[serving, transmitcall](initRemBuf)
     |[transmitcall]|
     HandleRemoteCall[operationexecution, transmitcall, transmitresults]
     |[transmitresults]|
     HandleExecutionResults[swapping, transmitresults]
   WHERE
   (* Definition of the processes ReceiveRemoteCall, HandleRemoteCall, and HandleExecutionResults *)
 ENDPROC (* ExecRemote *)
ENDPROC (* CommandServerProc *)
```

Figura 5. Visão geral do processo *CommandServerProc*.

ReceiveLocalCall espera pela ocorrência de um evento (porta *rtxcint*) contendo uma chamada da aplicação. Quando uma chamada é recebida, este processo a decodifica e a transfere (porta *operationexecution*), juntamente com os parâmetros correspondentes, para *PerformKernelCalls*. Após a execução de uma chamada, *PerformKernelCalls* retorna o resultado dessa execução (porta *operationexecution*), que é enviado (porta *transmitresults*) para *HandleExecutionResults*. O resultado da execução é analisado por *HandleExecutionResults*, e em função desta análise, este solicita ou não ao *Swapper* o bloqueio da tarefa responsável pela chamada.

ExecRemote é o servidor das chamadas remotas do kernel, tendo um comportamento similar ao *Rtxc*. Quando uma chamada remota é recebida, esta também deve ser decodificada, executada e os resultados dessa execução devem ser analisados. Entretanto, há uma diferença básica entre estes dois processos: o *Rtxc* pode receber apenas uma chamada por vez (uma chamada da tarefa em execução), enquanto que *ExecRemote* pode receber várias chamadas simultaneamente (provenientes dos vários processadores remotos).

Em função do exposto, e novamente de acordo com o princípio da ortogonalidade, *ExecRemote* foi refinado em três subprocessos comunicantes: *ReceiveRemoteCall*, que recebe (porta *serving*) e armazena as chamadas remotas, *HandleRemoteCall*, que decodifica e envia (porta *operationexecution*) uma chamada recebida para *PerformKernelCalls*, *HandleExecutionResults*, que analisa o resultado da execução de uma chamada e age em função desse resultado.

Todas as chamadas, quer sejam locais ou remotas, são executadas pelo *PerformKernelCalls*. Isto porque o comportamento da maior parte das chamadas é indiferente ao fato de serem executadas diretamente pelo processador local ou em um processador remoto. Assim sendo, foi definido um processo cuja função é a de executar todas as chamadas do kernel, evitando assim a duplicação de código na especificação.

3.4. *LinkServerProc*

Os canais de comunicação da máquina paralela podem estar conectados entre si formando uma rede de processadores. Três tipos de *links* entre os canais são possíveis: *Link de Rede (R)*,

conectando dois processadores, *Link de Entrada/Saída (ES)*, conectando o processador a dispositivos de entrada/saída ou ao hospedeiro e *Link Livre (L)*, quando não há conexões.

LinkServerProc representa os *threads* servidores de *link* do TRANS-RTXC: *KernelLinkIn* e *KernelLinkOut*, que lidam com a entrada/saída de dados com o hospedeiro ou com algum dispositivo de entrada/saída, e *Sender* e *Receiver*, que estão relacionados com o envio/recebimento de pacotes através da rede. A Figura 6 apresenta o trecho da especificação responsável pela criação dos processos servidores de *link*.

Dois aspectos foram levados em consideração na especificação deste processo: a necessidade de criar múltiplas instâncias dos processos servidores de *link* em função do número de *links* de comunicação disponíveis na máquina paralela, bem como a necessidade de criar diferentes instâncias destes processos em função do tipo do *link* disponível. Assim, tomando como base o princípio da generalidade, especificamos *LinkServerProc* de forma genérica, sendo este parametrizado com o número de *links* disponíveis no processador.

Primeiramente, estruturamos *LinkServerProc* como a composição em paralelo sem sincronização de *StartLinkProc* com o próprio *LinkServerProc*. Dessa forma, para cada *link* disponível uma instância de *StartLinkProc* é criada, operando em paralelo com as demais instâncias. A seguir, na especificação de *StartLinkProc* utilizamos uma expressão guardada para escolher entre os diferentes processos servidores de *link* que devem ser instanciados. Assim, se o *link* disponível for um *Link* de *Rede*, os processos *Receiver* e *Sender* são instanciados e associados a esse *link*. Por outro lado, se o *link* disponível for um *Link* de *Entrada/Saída* de dados com o hospedeiro ou um *Link Livre*, os processos *KernelLinkIn* e *KernelLinkOut* são instanciados e associados ao mesmo.

```
PROCESS LinkServerProc[memoryaccess, rtxcinports, rtxcoutports, chanin, chanout, serving, sending,
                       swapping] (rtable: RoutingTable, processornumber: Nat, numberofchan: Nat): noexit:=
  [numberofchan gt 0] ->    (* numberofchan represents the number of available channels *)
  ( StartLinkProc[memoryaccess, rtxcinports, rtxcoutports, chanin, chanout, serving, sending,
                  swapping](...)
    ||| i; LinkServerProc[memoryaccess, rtxcinports, rtxcoutports, chanin, chanout, serving, sending,
                          swapping](...) )
  [] [numberofchan eq 0] -> stop
WHERE
  PROCESS StartLinkProc[memoryaccess, rtxcinports, rtxcoutports, chanin, chanout, serving, sending,
                        swapping] (rtable: RoutingTable, processornumber: Nat, channumber:Nat): noexit:=
    memoryaccess !channumber ?typechan: Chan;
    ( [typechan eq R] ->    (* Routing link *)
      ( HIDE internalsending IN
          Receiver[memoryaccess, rtxcinports, serving, internalsending](...)
          |[internalsending]|
          Sender[rtxcoutports, sending,internalsending](...)  )
      [] [typechan ne R] -> (* Free link or input/output link *)
          (   KernelLinkIn[memoryaccess, chanin, rtxcinports, swapping](...)
          ||| KernelLinkOut[memoryaccess, chanout, rtxcoutports, swapping](...)  )  )
    WHERE
      (* Definition of the processes KernelLinkIn, KernelLinkOut, Sender, and Receiver *)
    ENDPROC (* StartLinkProc *)
ENDPROC (* LinkServerProc *)
```

Figura 6. Esquema de instanciação dos processos servidores de *link*.

3.4.1. *KernelLinkIn* e *KernelLinkOut*

KernelLinkIn e *KernelLinkOut* lidam com a entrada e saída de dados da máquina hospedeira, respectivamente. Duas tarefas determinam o comportamento desses processos: a recepção de

um pedido de transferência de dados e a transferência dos dados propriamente dita. Assim, de acordo com o princípio da ortogonalidade, *KernelLinkIn* foi refinado nos subprocessos comunicantes *ReceiveLinkInTransferRequest* e *HandleReceivedRequest*. A Figura 7 apresenta o refinamento de *KernelLinkIn*.

ReceiveLinkInTransferRequest recebe um pedido de transferência de dados (porta *chanin*) e solicita ao *Swapper* o bloqueio da tarefa responsável pelo pedido. *ReceiveLinkInTransferRequest* envia ao *HandleReceivedRequest* os parâmetros da transferência (porta *transmitrequest*) e este copia (porta *linkin*) os dados para a memória do sistema (porta *memoryaccess*). Ao término da transferência, *ReceiveLinkInTransferRequest* solicita ao *Swapper* a liberação da tarefa previamente bloqueada.

A especificação de *KernelLinkOut* é análoga à especificação de *KernelLinkIn*. A diferença básica é que no processo *KernelLinkOut* os dados são lidos da memória para a porta de saída (porta *linkout*).

```
PROCESS KernelLinkIn[memoryaccess, chanin, linkin, swapping](channumber: Nat): noexit :=
  HIDE transmitrequest IN
    ReceiveLinkInTransferRequest[chanin, swapping, transmitrequest](...)
    |[transmitrequest]|
    HandleReceivedRequest[transmitrequest, memoryaccess, linkin](...)
  WHERE
    (* Definition of the processes ReceiveLinkInTransferRequest and HandleReceivedRequest *)
ENDPROC (* KernelLinkIn *)
```

Figura 7. Refinamento do processo *KernelLinkIn*.

3.4.2. *Sender* e *Receiver*

Sender e *Receiver* estão relacionados com o envio e recepção de pacotes (dados e comandos) através da rede, respectivamente.

O mecanismo de roteamento é implementado através de tabelas estáticas fornecidas como parâmetros em cada processador. Cada pacote contém o número do processador destino. Todo processador da rede possui uma tabela contendo os demais processadores e seus respectivos *links* de saída. O processador conhece somente o *link* no qual o pacote deve ser enviado, sendo que o caminho restante não é de seu interesse. Essas tabelas são calculadas como a rota mais curta entre dois processadores, baseando-se nas conexões da rede.

Todos os pacotes encaminhados através da rede recebem a mesma prioridade de suas tarefas. Dessa forma, mensagens de tarefas mais prioritárias fluem mais rapidamente do que mensagens de tarefas menos prioritárias, caracterizando um aspecto fundamental de um STR.

Durante a especificação dos processos *Sender* e *Receiver* dois problemas surgiram: o primeiro foi a distinção entre a especificação de um pacote de comandos e a especificação de um pacote de dados, e o segundo foi o fato de que *Receiver* manipula de forma distinta pacotes de comandos e pacotes de dados. Dessa forma, baseando-nos no princípio da ortogonalidade, decidimos tratar pacotes de dados e comandos separadamente.

Sender foi refinado nos subprocessos independentes *SendDataBuffers* e *SendCommandBuffers*, que são responsáveis pelo envio de pacotes de dados e comandos através da rede, respectivamente. *SendDataBuffers* e *SendCommandBuffers* são análogos e cada um foi refinado (princípio da ortogonalidade) nos subprocessos comunicantes *ReceiveOutgoingBuffer* e *SendOutgoingBuffer*. A Figura 8 apresenta uma visão geral do refinamento do processo *Sender*.

```
PROCESS Sender[linkaddr, sending, internalsending](channumber: Nat): noexit :=
    SendDataBuffers[sending, internalsending, linkaddr](channumber)
 ||| SendCommandBuffers[sending, internalsending, linkaddr](channumber)
 WHERE
  PROCESS SendDataBuffers[sending, internalsending, linkaddr](channumber: Nat): noexit:=
   HIDE transmitbuffer IN
    ReceiveOutgoingBuffer[sending, internalsending, transmitbuffer](...)
    |[transmitbuffer]|
    SendOutgoingBuffer[transmitbuffer, linkaddr](...)
   WHERE
     (* Definition of the processes ReceiveOutgoingBuffer and SendOutgoingBuffer *)
  ENDPROC (* SendDataBuffers *)
  PROCESS SendCommandBuffers[sending, internalsending, linkaddr](channumber: Nat): noexit:=
    HIDE transmitbuffer IN
      ReceiveOutgoingBuffer[sending, internalsending, transmitbuffer](...)
      |[transmitbuffer]|
      SendOutgoingBuffer[transmitbuffer, linkaddr](...)
     WHERE
      (*Definition of the processes ReceiveOutgoingBuffer and SendOutgoingBuffer *)
  ENDPROC (* SendCommandBuffers *)
ENDPROC (* Sender *)
```

Figura 8. Visão geral do processo *Sender*.

Um pacote é recebido (portas *sending* e *internalsending*) e armazenado pelo processo *ReceiveOutgoingBuffer*. À medida em que os pacotes são enviados (porta *linkaddr*) por *SendOutgoingBuffer*, este solicita (porta *transmitbuffer*) um novo pacote a *ReceiveOutgoingBuffer*.

De forma análoga ao *Sender*, *Receiver* foi refinado nos subprocessos independentes *ReceiveDataBuffers* e *ReceiveCommandBuffers*, que são responsáveis pela recepção de pacotes de dados e de comandos provenientes da rede, respectivamente. *ReceiveDataBuffers* e *ReceiveCommandBuffers* também foram refinados em dois subprocessos comunicantes (princípio da ortogonalidade), nomeados *ReceiveBuffer* e *HandleReceivedBuffer*. A Figura 9 apresenta uma visão geral do refinamento do processo *Receiver*.

```
PROCESS Receiver[memoryaccess, linkaddr, serving, internalsending](rtable: RoutingTable,
                    processornumber: Nat, channumber: Nat): noexit :=
    ReceiveDataBuffers[memoryaccess, linkaddr, internalsending](...)
 ||| ReceiveCommandBuffers[linkaddr, serving, internalsending](...)
 WHERE
    PROCESS ReceiveDataBuffers[memoryaccess, linkaddr, sending](rtable: RoutingTable,
                                  pnumber: Nat, channumber: Nat): noexit:=
     HIDE transmitbuffer IN
       ReceiveBuffer[linkaddr, transmitbuffer](...)
       |[transmitbuffer]|
       HandleReceivedBuffer[transmitbuffer, memoryaccess, sending](...)
     WHERE
       (* Definition of the processes ReceiveBuffer and HandleReceivedBuffer *)
    ENDPROC (* ReceiveDataBuffers *)
    PROCESS ReceiveCommandBuffers[memoryaccess, linkaddr, sending](rtable: RoutingTable,
                                    pnumber: Nat, channumber: Nat): noexit:=
     HIDE transmitbuffer IN
       ReceiveBuffer[linkaddr, transmitbuffer](...)
       |[transmitbuffer]|
       HandleReceivedBuffer[transmitbuffer, memoryaccess, sending](...)
     WHERE
       (* Definition of the processes ReceiveBuffer and HandleReceivedBuffer *)
    ENDPROC (* ReceiveCommandBuffers *)
ENDPROC (* Receiver *)
```

Figura 9. Visão geral do processo *Receiver*.

Um pacote é recebido (porta *linkaddr*) e armazenado por *ReceiveBuffer*. Assim que os pacotes são recebidos, estes são enviados (porta *transmitbuffer*) para *HandleReceivedBuffer*. Quando *HandleReceivedBuffer* recebe um pacote (dados ou comandos), este processo verifica o destino do mesmo. Caso o destino do pacote seja um processador remoto, o mesmo é enviado (porta *sending*) para o processo *Sender*. Caso o destino do pacote seja o processador local, então duas situações são possíveis: se *HandleReceivedBuffer* está lidando com um pacote de dados, os dados são copiados diretamente para seu endereço destino (porta *memoryaccess*), e se *HandleReceivedBuffer* está lidando com um pacote de comandos, o comando é enviado (porta *serving*) para *ExecRemote*.

3.5. *Copier*

Copier lida com a transferência de blocos de dados de endereços locais para endereços remotos. Uma transferência de dados pode ser dividida em três fases: a recepção de um pedido de transferência, a execução da transferência e a liberação de uma tarefa à espera do fim da transferência.

De acordo com o princípio da ortogonalidade, poderíamos ter refinado *Copier* em três subprocessos, cada qual encarregado de executar uma das fases acima descritas. No entanto, *Copier* foi refinado somente em dois subprocessos comunicantes: *ReceiveCopyCall*, que é responsável pela recepção de um pedido de transferência e pela liberação da tarefa bloqueada que está esperando o fim da transferência; e *DataTransfer*, que é responsável pela transferência propriamente dita. Esta arquitetura foi escolhida porque tanto a recepção de um pedido de transferência, quanto a liberação da tarefa bloqueada, lidam de alguma forma com a tarefa que solicitou a transferência. A Figura 10 apresenta os principais aspectos da especificação LOTOS do processo *Copier*.

```
PROCESS Copier[copying, sending, memoryaccess, swapping](rtable: RoutingTable,
                    pnumber: Nat): noexit :=
 HIDE transmitcall IN
  ReceiveCopyCall[copying, sending, swapping, transmitcall](...)
  |[transmitcall]|
  DataTransfer[sending, memoryaccess, transmitcall](...)
WHERE
 PROCESS ReceiveCopyCall[copying, sending, swapping, transmitcall](rtable: RoutingTable,
                                pnumber: Nat): noexit:=
    ReceiveCall[copying, transmitcall](...)
   ||| UnLockWaitingTask[sending, swapping, transmitcall](...)
  WHERE
   (* Definition of the processes ReceiveCall and UnLockWaitingTask *)
 ENDPROC (* ReceiveCopyCall *)
 PROCESS DataTransfer[sending, memoryaccess, transmitcall](rtable: RoutingTable): noexit:=
   transmitcall ?copyinfo: Copyarg;
    MakeTransfer[transmitcall, sending, memoryaccess](...) >>
     DataTransfer[sending, memoryaccess, transmitcall](...)
  WHERE
   PROCESS MakeTransfer[transmitcall, sending, memoryaccess](rtable: RoutingTable,
                            copyinfo: Copyarg): exit:=
    HIDE transmitbuffer IN
     ReadBufferFromMemory[memoryaccess, transmitbuffer](...)
     |[transmitbuffer]|
     SendBuffer[sending, transmitbuffer, transmitcall](...)
    WHERE
     (*Definition of the processes ReadBufferFromMemory and SendBuffer *)
   ENDPROC (* MakeTransfer *)
  ENDPROC (* DataTransfer *)
ENDPROC (* Copier *)
```

Figura 10. Visão geral do processo *Copier*.

Embora *ReceiveCopyCall* possua duas funções relacionadas, estas são independentes. Como consequência, a aplicação do princípio da ortogonalidade na especificação deste processo resultou no refinamento do mesmo em dois subprocessos também independentes: *ReceiveCall*, que lida com a recepção (porta *copying*) e o armazenamento dos pedidos de transferência de dados, e *UnLockWaitingTask*, que libera a tarefa bloqueada que está aguardando o fim dessa transferência. Essa tarefa pode estar localizada tanto no processador local quanto em um processador remoto.

A função de *DataTransfer* é prover a transferência de dados entre o processador local e um processador remoto. Após a recepção dos parâmetros da transferência, *DataTransfer* instancia *MakeTransfer* que executará a transferência. Através de um raciocínio ortogonal lógico, *MakeTransfer* foi refinado nos subprocessos comunicantes *ReadBufferFromMemory* e *SendBuffer*. À medida em que os blocos de dados são lidos na memória por *ReadBufferFromMemory*, estes são enviados por *SendBuffer* ao processo *Sender*, que os transmitirá através da rede.

3.6. *Clck*

Clck é responsável pelo controle do tempo e dos temporizadores no TRANS-RTXC. Esse é o único processo da especificação que poderia exigir uma construção temporal para a modelagem concreta de um período de tempo. Entretanto, numa modelagem abstrata, semelhante a que foi realizada neste projeto, essa construção pode ser substituída por um evento interno. Assim, através da aplicação do princípio da ortogonalidade, *Clck* foi refinado nos subprocessos comunicantes *TicksCounter*, que controla o tempo, e *ClkblkHandler*, que controla os temporizadores. A Figura 11 apresenta uma visão geral do processo *Clck*.

```
PROCESS Clck[swapping, operationexecution](rtctick, ticker: Tick, clkqptr: ClkblkQueue): noexit :=
  HIDE tock IN
    TicksCounter[tock, operationexecution](...)
    |[tock]|
    ClkblkHandler[swapping, tock, operationexecution](...)
  WHERE
    PROCESS TicksCounter[tock, operationexecution]( rtctick, ticker: Tick): noexit:=
      (* TicksCounter behaviour *)
    ENDPROC (* TicksCounter *)
    PROCESS ClkblkHandler[swapping, tock, operationexecution](clkqptr: ClkblkQueue): noexit:=
      HIDE release IN
        ClkblkQueueControl[tock, release, operationexecution](...)
        |[release]|
        ClkblkRelease[swapping, release, operationexecution]
      WHERE
        PROCESS ClkblkQueueControl[tock, release, operationexecution](clkqptr: ClkblkQueue): noexit:=
          ( tock; CheckForReleasing[release](decreaseClkblkValue(clkqptr))
              >> ACCEPT newclkqptr: ClkblkQueue IN
                  ClkblkQueueControl[tock, release, operationexecution](newclkqptr) )
            [] operationexecution !rtxc_timer ?newtimer: Clkblk;  (* Insertion of a new timer *)
              ClkblkQueueControl[tock, release, operationexecution](addClkblk(newtimer, clkqptr))
            WHERE
            (* Definition of the process CheckForReleasing *)
        ENDPROC (* ClkblkQueueControl *)
        PROCESS ClkblkRelease[swapping, release, operationexecution]: noexit :=
          (* Behaviour of the process ClkblkRelease *)
        ENDPROC (* ClkblkQueueControl *)
    ENDPROC (* ClkblkHandler *)
ENDPROC (* Clck *)
```

Figura 11. Visão geral do processo *Clck*.

O TRANS-RTXC utiliza duas unidades de tempo: *ticks* e *tocks*. Um *tick* representa a quantidade de tempo entre as interrupções do relógio geradas pelo sistema. Um *tock* representa um número inteiro de *ticks*, sendo a unidade de tempo utilizada por todas as funções dependentes do tempo no sistema. A frequência de *ticks* em um intervalo de tempo, bem como o número de *ticks* em cada *tock* são determinados pelo projetista em função das características do processador paralelo e da aplicação que será executada.

TicksCounter conta o número de *ticks* no sistema. Um *tick* é representado pela ocorrência de um evento oculto chamado *timer*. *TicksCounter* aumenta o número de *ticks* até que um *tock* seja completado. Quando o número de *ticks* perfaz um *tock*, o processo *ClkblkHandler* é sinalizado.

Dois aspectos foram considerados durante a especificação de *ClkblkHandler*: a necessidade de controlar os temporizadores e a necessidade de liberar um temporizador quando seu tempo se esgota. Baseado nestas restrições, *ClkblkHandler* foi refinado (princípio da ortogonalidade) nos subprocessos comunicantes *ClkblkQueueControl*, responsável pelo controle da fila de temporizadores, e *ClkblkRelease*, responsável pelo tratamento do estouro de um temporizador.

ClkblkQueueControl mantém todos os temporizadores em uma lista encadeada pelo tempo. Quando o valor de um elemento (número de *tocks*) dessa lista é igual a zero, *ClkblkQueueControl* remove esse elemento da lista e o envia para *ClkblkRelease*. Este, por sua vez, sinaliza a tarefa associada ao temporizador e chama o processo *Swapper*.

4. Validação da Especificação

Esta seção apresenta uma breve descrição da abordagem utilizada na validação da especificação LOTOS do TRANS-RTXC. Para a validação deste estudo de caso, o conjunto de ferramentas MiniLite (versão 1996) [1] foi utilizado em uma abordagem combinando simulação e verificação.

Dada a quantidade de definições de tipos de dados presentes na especificação do kernel, uma atenção especial foi dedicada a sua validação. Uma ferramenta chamada Rep-ADT [2] foi utilizada para prover consistência à definição dos tipos de dados. Assim, durante o desenvolvimento da especificação do TRANS-RTXC, alguns axiomas críticos, que poderiam acarretar em comportamentos inesperados, foram identificados e substituídos por axiomas consistentes.

Após a verificação da consistência dos TADs, a ferramenta ADT-Interface foi utilizada na validação das operações dos mesmos. O ADT-Interface é parte integrante do Smile [2], o simulador simbólico interativo do MiniLite, e contém várias funcionalidades para a análise dos TADs de uma especificação. Essa ferramenta foi intensamente utilizada na simulação das operações dos TADs definidos neste projeto. Por exemplo, no caso da fila de Blocos de Controle de Tarefas (BCTs), que é o TAD que contém a política de escalonamento do kernel, e que necessita dispor de propriedades, tais como prioridade e política de first-in-first-out de cada prioridade[1], as equações, inicialmente definidas, violavam esta última propriedade. Este foi um dos tipos de erros que puderam ser detectados e corrigidos através do ADT-Interface.

Após a validação dos TADs da especificação, o próximo passo foi a validação do comportamento da mesma. Neste sentido, a primeira etapa foi a simulação de cada processo da especificação usando o Smile. Esta atividade procurou garantir a correção, com relação a sua

[1] Se duas ou mais tarefas igualmente prioritárias são inseridas na fila, a primeira a ser inserida deverá ser a primeira a ser escalonada.

descrição informal, de cada processc individualmente e não da especificação como um todo. Durante esta etapa, várias inconsistências e erros foram identificados. Para corrigir tais erros, um estudo mais profundo do kernel, incluindo a análise de seu código, foi necessário.

A segunda abordagem empregada foi a execução de toda a especificação em paralelo com um processo contendo sequências de testes. Cada sequência de teste foi provida manualmente e procurou cobrir cenários específicos, tais como o envio/recebimento de mensagens, bloqueio/liberação de recursos, temporização, operações envolvendo semáforos e transferência de dados. O sucesso de cada teste foi indicado por um evento especial não definido na especificação. Uma sequência de teste também poderia ter sido provida aleatoriamente pela ferramenta LOLA [12], mas tal recurso não foi utilizado tendo em vista que essa sequência não poderia modelar um cenário específico.

Como ilustração desta técnica, a Figura 12 apresenta os traços resultantes de uma sequência de teste envolvendo o bloqueio/liberação de um recurso. Neste teste, o comando *RtxcLock* (linha 2) é utilizado para bloquear um recurso representando uma saída de dados (e.g., impressora). Ao fim da transferência, este recurso é liberado (linha 16) através do comando *RtxcUnlock*. Eventuais tentativas de bloquear um recurso previamente bloqueado e não liberado (linha 8), resultam na suspensão da tarefa responsável pelo comando (linha 9).

```
1 - I(memoryaccess) !saveTask((TaskId(0,0), highPriority), (TaskId(Succ(0),0),
                      Dec(Dec(highPriority))), (TaskId(Succ(Succ(0)),0), Dec(highPriority))
                      !createSemaList !addRheader(..., createRheaderList)
2 - I(rtxcint) !rtxcLarg(Larg(rtxc_lock, highPriority, TaskId(0, 0), resource(Succ(0), 0), 0))
...
3 - I(resourceaccess) !changeResourceValue !Rheader(resource(Succ(0), 0), TaskId(0, 0), true,
                       initTaskHeaderQueue)
...
4 - I(chanout) !Succ(0) !Chanarg(TaskId(0, 0), highPriority, incaddress, Ant(Ant(bufsize)))
5 - I(swapping) !blocktask !linkwait !TaskHeader(TaskId(0, 0), highPriority)
6 - I(memoryaccess) !savecontext !Frame !TaskId(0, 0)
7 - I(memoryaccess) !restorecontext !Frame !TaskId(Succ(Succ(0)), 0)
8 - I(rtxcint) !rtxcLarg(Larg(rtxc_lock, Dec(highPriority), TaskId(Succ(Succ(0)), 0),
              resource(Succ(0), 0), Succ(0)))
...
9 - I(swapping) !blocktask !lockwait !TaskHeader(TaskId(Succ(Succ(0)), 0), Dec(highPriority))
..
10 - I(memoryaccess) !savecontext !Frame !TaskId(Succ(Succ(0)), 0)
11 - I(memoryaccess) !restorecontext !Frame !TaskId(Succ(0), 0)
12 - ... (* KernelLinkOut *)
13 - I(swapping) !releasetask !TaskId(0, 0)
...
14 - I(memoryaccess) !savecontext !Frame !TaskId(Succ(0), 0)
15 - I(memoryaccess) !restorecontext !Frame !TaskId(0, 0)
16 - I(rtxcint) !rtxcLarg(Larg(rtxc_unlock, highPriority, TaskId(0, 0), resource(Succ(0), 0), Succ(0)))
...
17 - I(swapping) !releasetask !TaskId(Succ(Succ(0)), 0)
...
18 - success
```

Figura 12. Traços de uma sequência de teste envolvendo o bloqueio e a liberação de um recurso.

Durante esta etapa de validação, poucos erros foram encontrados e corrigidos. Como os testes estão limitados a cenários previamente definidos, é possível afirmar que a especificação está correta para os cenários modelados, mas não é possível afirmar que toda a especificação está correta. Quanto mais abrangentes forem os cenários de teste, maior será a correção que os mesmos proporcionarão.

Finalmente, foi gerada e analisada a Máquina de Estados Finitos Estendida (MEFE) do comportamento de cada processo da especificação. Através do Smile, é possível gerar a MEFE

do comportamento de um dado processo através de sua simulação exaustiva, isto é, expandindo por completo todos os possíveis comportamentos. Entretanto, em alguns casos, tais como a combinação de uma chamada recursiva com o operador de paralelismo ou a indisponibilidade de recursos (e.g., falta de memória), não é possível gerar a MEFE de tal processo.

Após a geração da MEFE de um processo, essa máquina de estados é então transformada em um código de representação de autômatos, chamado FC2, que é a entrada para as ferramentas de verificação do MiniLite. Duas ferramentas foram utilizadas nesta etapa: Mauto [2], que é verificador comportamental, e Autograph [2], que é um visualizador gráfico de autômatos.

Mauto foi utilizada na redução do código FC2 através de bissimulação fraca. Algumas propriedades desse autômato reduzido, tais como a ausência de impasses, foram verificadas, sendo que esse autômato foi visualizado e analisado também através do Autograph, toda vez que as suas dimensões permitiram a utilização dessa ferramenta.

A Figura 13 ilustra a representação gráfica do autômato reduzido do processo *KernelLinkIn*.

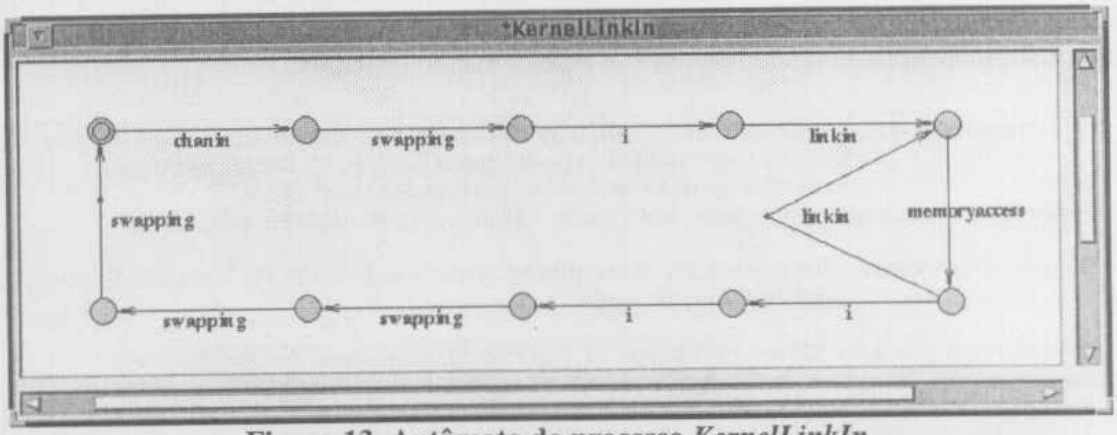

Figura 13. Autômato do processo *KernelLinkIn*.

Esta metodologia foi aplicada para cada processo da especificação. Durante esta atividade nenhum erro e nenhum impasse foram encontrados. A Tabela 1 contém algumas informações sobre os autômatos dos principais processos da especificação. Para cada processo é fornecido o número de estados e transições do autômato gerado e do autômato reduzido respectivamente. Esses dados foram obtidos utilizando uma máquina Sparc-Ultra1, com128 MBytes de RAM.

A especificação do processo *LinkServerProc* violava as condições necessárias para a geração de sua MEFE pois combinava uma chamada recursiva com o operador de paralelismo (ver Figura 6). Desta forma, este processo teve de ser modificado através da fixação do número de *links* disponíveis na máquina paralela. Os dados relativos a este processo foram obtidos utilizando apenas dois *links*.

Devido às limitações de memória e dado que o número de estados do autômato a ser gerado é potencialmente alto ($5 \times 144 \times 4497 \times 36 \times 42 \times 86 \approx 4.2\ 10^{11}$)[2], foi impossível gerar o autômato para a especificação como um todo. Assim, embora tenha-se verificado a inexistência de impasses nos principais processos da especificação, não foi possível constatar a inexistência dos mesmos no comportamento global da especificação. Maiores detalhes sobre as atividades de validação dessa especificação podem ser encontrados em [6].

[2] Este número foi obtido, tomando-se como base o número de estados das MEFEs reduzidas dos subprocessos que compõe a especificação.

Nome do processo	Estados	Transições	Estados	Transições
Swapper	12	54	5	12
CommandServerProc	14256	74988	144	696
Rtxc	108	231	12	23
ExecRemote	132	412	12	35
LinkServerProc	5114	30179	4497	23844
StartLinkServer	120	345	114	301
KernelLinkIn	10	11	10	11
KernelLinkOut	10	11	10	11
Sender	4	16	3	10
Receiver	4	16	4	16
Copier	36	113	36	113
Clck	60	184	42	124
PerformKernelCalls	513	1205	86	234
PerformFunctions	135	188	46	84
ResourcesManagement	2	3	1	1
SemaphoresManagement	3	5	3	5

Tabela 1. Dados dos autômatos gerados.

5. Conclusões

Este trabalho apresentou os resultados da utilização de métodos formais no desenvolvimento de um produto industrial. O propósito deste estudo de caso foi investigar se a TDF LOTOS e um conjunto de ferramentas de validação relacionadas a essa TDF são apropriadas para serem usadas na especificação de aplicações industriais distribuídas com requisitos de tempo real, tais como o kernel paralelo tempo real TRANS-RTXC.

O desenvolvimento de um kernel, para aplicações de tempo real, pode ser uma tarefa difícil e, em um mundo onde a tecnologia muda muito rapidamente, cresce a importância da adoção de um ciclo de desenvolvimento rápido e confiável. Neste sentido, a adoção de uma metodologia de desenvolvimento baseada em TDFs pode facilitar todo o projeto, uma vez que elas não estão restritas às fases de especificação, mas podem ser utilizadas como base para as demais fases do ciclo de vida desses sistemas. Outros trabalhos relacionados com a modelagem formal de sistemas operacionais podem ser encontrados em [3, 7, 13].

A especificação formal do TRANS-RTXC produzida neste trabalho serve de base para a análise e inclusão de novas características a esse kernel. Essas características podem ser adicionadas e avaliadas enquanto especificação, identificando-se assim a viabilidade de sua real implementação.

LOTOS foi utilizada com sucesso na especificação formal do TRANS-RTXC. O projeto formal do TRANS-RTXC foi realizado em um período de tempo relativamente curto. A produção da especificação LOTOS consumiu 13 homens-mês, incluindo o tempo necessário para aprender tanto a TDF LOTOS quanto o kernel TRANS-RTXC, enquanto que as atividades de validação consumiram 3 homens-mês, incluindo o tempo necessário para aprender a operar as ferramentas.

O desenvolvimento da especificação do TRANS-RTXC foi norteado por alguns princípios qualitativos genéricos de projeto. A adoção de tal metodologia de desenvolvimento proporcionou um aumento na qualidade da especificação desenvolvida, uma vez que as principais decisões de projeto foram fundamentadas em tais princípios. Tal fato tem influência direta nas fases de implementação e manutenção deste sistema, resultando em custos menores para as mesmas.

A utilização de LOTOS, por outro lado, trouxe algumas dificuldades para a especificação formal do TRANS-RTXC. A teoria de TADs de LOTOS confere uma grande flexibilidade à definição de tipos. Entretanto, essa flexibilidade se contrapõe à grande quantidade de tempo dispensada para a especificação e validação dos mesmos. A inclusão de tipos de dados concretos a LOTOS, como encontrados em E-LOTOS, resultaria em custos menores para esta fase de desenvolvimento.

O paralelismo de LOTOS também deixou a desejar quando da estruturação em paralelo com sincronização parcial dos processos que representam os *threads* do TRANS-RTXC. A disposição e a combinação dos mesmos com os operadores de paralelismo teve que ser realizada cuidadosamente, levando-se sempre em consideração que o resultado final da composição seria obtido através do conjunto total de restrições.

A especificação produzida foi validada através de uma abordagem que combinou atividades de simulação com atividades de verificação. Em função da grande quantidade de TADs encontrados na especificação, uma atenção especial foi dedicada à validação dessas definições. Neste sentido, as ferramentas Rep-ADT e ADT-Interface foram intensamente utilizadas na verificação de algumas propriedades dos TADs.

As atividades de validação do comportamento da especificação foram realizadas seguindo três abordagens distintas. A primeira delas procurou validar cada processo da especificação independentemente dos demais processos. A segunda abordagem utilizou processos contendo sequências de testes para testar toda a especificação. Finalmente, a última abordagem procurou, para cada processo da especificação, verificar a inexistência de impasses através da geração e análise da MEFE do comportamento da especificação. Infelizmente, foi impossível aplicar essa técnica para toda a especificação, devido ao problema da explosão de estados que ocorre na geração de sua MEFE.

A simulação, assim como o teste, limitam a validação das especificações a cenários de teste pré-definidos, não podendo fornecer resultados conclusivos, pois não podem ser empregados exaustivamente. Entretanto, essas técnicas podem ser empregadas em sistemas reais, o que muitas vezes não acontece com a verificação. O poder de detecção de erros será tanto melhor quanto maior for a cobertura dos cenários de teste pré-definidos.

Como trabalhos futuros, inicialmente pretende-se adicionar à especificação produzida outras estruturas, tais como filas e blocos de memória e seus respectivos conjuntos de chamadas. Novas políticas de escalonamento também deverão ser especificadas. Essas políticas podem ser avaliadas através do Smile, escolhendo-se assim a política mais apropriada para cada tipo de aplicação. De forma análoga ao desenvolvimento do kernel, a especificação dessas políticas não implicará na utilização de extensões temporais de LOTOS, uma vez que a unidade de tempo utilizada pelo TRANS-RTXC foi especificada abstratamente através de um evento interno.

Pretende-se também completar o ciclo de vida do TRANS-RTXC (fases de implementação e implantação) com base na sua especificação LOTOS. Uma implementação automática poderá ser gerada através do compilador TOPO [10]. Em princípio, este código necessitará de uma complementação manual, sendo que o mesmo poderá ser confrontado ao TRANS-RTXC original, visando, por exemplo, uma verificação do grau de automatização atingido pelo compilador TOPO ou uma comparação de desempenho entre essas duas implementações.

6. Agradecimentos

Gostaríamos de agradecer o apoio recebido da CAPES, FAPESP e CNPq para o desenvolvimento deste trabalho.

7. Referências Bibliográficas

[1] T. Bolognesi, J.v.d. Lagemaat e C. A. Vissers (Eds). *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, the Netherlands, 1995.

[2] M. Caneve, E. Salvatori (CPR)(Eds), *Lite User Manual: Final Deliverable*, Lo/WP2/N0034/ V08. ESPRIT Ref: 2304, 1992.

[3] T. Cattel. Modelization and verification of a multiprocessor realtime OS Kernel. In D. Hogrefe and S. Leue (Eds), *Seventh International Conference on Formal Description Techniques*, pp 35-50, 1994.

[4] C.R.G. de Farias, W. L. de Souza and C. E. Moron. Formal Specification and Automatic Implementation of a Real-Time Parallel Kernel. In *Proceedings of the XXI Workshop on Real-Time Programming (WRTP'96)*, Gramado-RS, pp 175-176, Novembro/1996.

[5] C.R.G. de Farias, W. L. de Souza and C. E. Moron. Design of a Real-Time Parallel Kernel Using LOTOS. In *Proceedings of the V Workshop on Parallel and Distributed Real-Time Systems*, Genebra (Suíça), Abril/1997.

[6] C.R.G. de Farias. *Especificação e Validação de um Kernel Tempo Real Paralelo Utilizando LOTOS*. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal de São Carlos, Agosto/1997.

[7] O. Gonzalez, et Al. Design of operating systems using the FDT ESTELLE. In D. Hogrefe and S. Leue (Eds), *Seventh International Conference on Formal Description Techniques*, pp 51-66, 1994.

[8] ISO IS 8807. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1989.

[9] ISO/IEC JTC1/SC21/WG7, Enhancements to LOTOS, *Final Commite Draft on Enhancements to LOTOS*, Project WI 1.21.20.2.3, Dezembro/1997.

[10] J. A. Manas, T. de Miguel, J. Salvachua, and A. Azcorra. Tool Support to Implement LOTOS Formal Specifications. *Computers Networks and ISDN Systems*, North Holland, 1993.

[11] C.E. Moron. *Designing Adaptable Real-Time Fault-Tolerant Parallel Systems*, Ph.D. Thesis, Department of Computer Science, University of York, 1994.

[12] S. Pavón and M. Llamas. The Testing Funcionalities of LOLA. In J. Quemada, J.A. Mañas, and E. Vázquez (Eds), *Formal Descriptions Techniques III*, Madri (Espanha), 1990.

[13] C. Pecheur. Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15 (2), pp 93-112, 1992.

[14] E. Verhulst and H. Thielemans. Transparent distributed real-time processing with TRANS-RTXC and transputers. In P. H. Welch, D. Stiles, T. L. Kunii and A. Bakkers (Eds), *Transputing '91 V.2*, IOS Press, 709-724, 1991.

[15] C.A. Vissers, G. Scollo, M. van Sinderen and E. Brinksma. On the use of specification styles in the design of distributed systems. In P.H.J.van Eijk, C.A.Vissers e M.Diaz (Eds), *The Formal Description Technique LOTOS*. North-Holland, 1989.