Verificação de modelos Reo com nuXmv*

Daniel Arena Toledo[1], **Bruno Lopes**[1]

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

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

Abstract. *With the advent of critical systems in the modern world, the necessity to validate them also increased. That necessity comes from the fact that a system fail can cause catastrophic damages, therefore it's necessary to guarantee that those programs won't fail. Reo is a graphical modelling coordination language that takes advantage of the natural properties in distributed systems, such as data transfer and remote function calls to model such systems. Model checkers are tools used to verify and validate properties and characteristics about a model. This article proposes a compiler to generate a nuXmv (a symbolic model checker) model from a Reo model.*

Resumo. *Com o advento de sistemas críticos no mundo moderno, a necessidade de validá-los também aumentou. Essa necessidade se deve ao fato de que a falha desses sistemas podem acarretar consequências catastróficas, portanto é necessário garantir que esse programas não vão falhar. Reo é uma linguagem de coordenação de modelagem gráfica que toma vantagem das características naturais dos sistemas distribuídos como transferências de dados e chamadas de funções remotas para modelar esses sistemas. Verificadores de modelos são ferramentas utilizadas para verificar e validar propriedades e características a cerca de modelos. Esse artigo propõe um compilador para gerar um modelo nuXmv (um checador de modelos simbólico) a partir de um modelo Reo.*

1. Introdução

Com o advento da tecnologia, sistemas críticos estão se tornando cada vez mais comuns, e com isso, a necessidade de formalizar métodos para sua validação. Sistemas críticos [Knight 2002] são aqueles em que falhas podem trazer consequências relevantes, como perda de vidas, alta perda financeira ou destruição significante, ou seja, eles precisam ser confiáveis. Esses sistemas podem ser encontrados em várias áreas como biomédica, nuclear e aviônicos.

Como exemplos desses tipos de sistemas que já foram validados formalmente, temos a Linha 1 do Metrô de Paris [Gerhart et al. 1994], levando a um sistema totalmente automatizado. A Airbus também usa métodos formais para certificar os sistemas eletrônicos das famílias de aeronaves A318 e A340-500/60 [Bochot et al. 2009].

Essa necessidade de validar sistemas trouxe consigo a necessidade de ferramentas para auxiliar nesse processo, pois essa tarefa é tipicamente árdua e complexa. Estas ferramentas reduzem a complexidade além de tanto facilitar quanto agilizar essa validação.

*Os autores agradecem ao CNPq pelo apoio ao projeto

Dentre as categorias de ferramentas disponíveis, verificadores de modelos são uma delas e é amplamente utilizada.

Esses verificadores são ferramentas capazes de verificar se propriedades, normalmente modeladas por meio de uma lógica, estão presentes nos modelos dados como entrada. Dentre algumas propriedades que podem ser verificadas estão a presença de *deadlocks* no modelo e a alcançabilidade de algum estado modelado [Baier and Katoen 2008]. Um exemplo desses verificadores é o nuXmv[1], que é o verificador usado neste trabalho.

Como linguagem para modelar o sistema, temos o Reo [Arbab 2004], uma linguagem gráfica baseada em coordenações usada na modelagem de sistemas, com foco em modelos de sistemas distribuídos. Sua construção desenvolveu-se para que características como chamadas remotas de métodos e transferência de mensagens sejam nativas e intuitivas. Na Figura 1 temos os conectores canônicos da linguagem. *Constraint Automata* [Arbab et al. 2004] é um formalismo criado para denotar a semântica formal de Reo, com ele é possível construir autômatos que modelam um programa Reo.

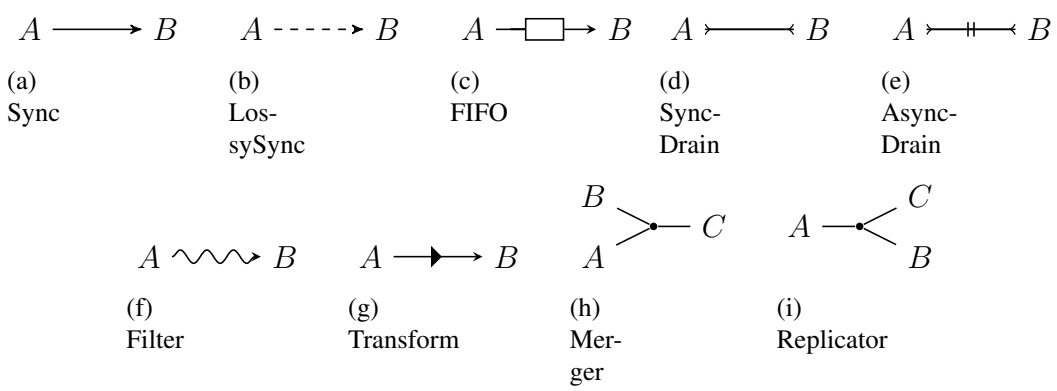

Figura 1. Conectores canônicos de Reo por [Arbab 2004]

Este trabalho consiste no desenvolvimento de um compilador para converter um modelo Reo em um modelo para o verificador de modelos nuXmv, ferramenta amplamente utilizada na indústria [Cavada et al. 2014].

2. Reo

Reo é baseado em canais onde coordenadores complexos são construídos a partir da composição de coordenadores mais simples [Arbab 2004]. Esses coordenadores complexos são conhecidos como conectores e são o princípio da linguagem. O objetivo de Reo é conectar instâncias de diferentes componentes, para agirem juntos num único sistema.

Como um modelo de coordenação, seu foco é nos conectores, suas composições e como se comportam. Não se importando com as entidades em si que estão conectadas, trabalhando juntas por esses conectores. Portanto essas entidades podem ser módulos de códigos, processos, objetos, *web services* entre outros componentes de software, conhecidas como instâncias de componentes.

[1] `https://nuxmv.fbk.eu/`

Conectores, conhecidos no Reo como canais, são definidos como uma ligação ponto-a-ponto entre dois nós, onde cada canal tem sem próprio comportamento pré-definido. Todo canal tem exatamente dois fins, e podem ser usados para compor conectores mais complexos. Além dos canais definidos formalmente pela linguagem, exemplificados na Figura 1, o usuário também pode definir seus próprios canais, e até combiná-los com os já definidos.

Um nó é definido no Reo como uma organização lógica que denota como as pontas dos conectores estão conectadas entre si. Um nó pode ser de origem, onde os dados entram no canal; sumidouro, onde os dados saem do canal; ou misto, onde os dados podem entrar ou sair, mas não simultaneamente. Essas pontas podem então ser usadas por qualquer entidade para tanto receber quanto enviar dados, dado que a instância a que pertence a entidade está conectada a uma dessas pontas.

3. Verificação de modelos

Verificadores de modelos são ferramentas usadas na verificação de propriedades formalmente caracterizadas acerca de sistemas previamente modelados. Essa verificação [Baier and Katoen 2008] se dá explorando todos os possíveis estados do modelo, ou seja, o verificador examina cada possível cenário do sistema de forma sistemática para mostrar que o modelo realmente satisfaz dada propriedade.

Essas propriedades a serem verificadas podem representar características de segurança, ou seja, propriedades que devem sempre valer para que algo indesejado nunca aconteça, como por exemplo a liberdade de *deadlock*; também podem representar características de vida, são propriedades para garantir a evolução do programa, ou seja, elas definem comportamentos que o sistema deve ter para sua execução; além de outras.

Ao se verificar uma propriedade, existem três possíveis resultados: a propriedade é dada como válida; a propriedade é dada como falsa; ou não foi possível verificar a propriedade, seja por falha na modelagem, ou porque o modelo explodiu, ou seja, se tornou muito grande para ser avaliado. Além disso, uma propriedade pode ser do tipo universal (ela vale para todos os estados do sistema), então em caso de falha o verificador apresentará um contra-exemplo; ou do tipo existencial (em algum estado do sistema ela vale), nesse caso o avaliador apresentará um exemplo no caso em que ela se dá como verdadeira.

O verificador de modelos usado neste trabalho é o nuXmv, um checador simbólico que verifica tanto sistemas síncronos finitos quanto infinitos. O nuXmv se divide em dois principais campos: verificações finitas, onde ele apresenta técnicas de verificação com algoritmos estado-da-arte; e sistemas infinitos, com tipos de dados como *Integers* e *Reals*, além de prover com verificações de modelos baseadas em SMT (*Satisfability Modulo Theory*) [Cavada et al. 2014].

A Figura 2 representa o que é feito essencialmente neste projeto, o compilador recebe um programa Reo, o formaliza para *constraint automata*, que é então compilado para um modelo nuXmv. Assim propriedades à cerca do programa podem ser verificadas pelo verificador de modelos.

O Listing 1 apresenta o módulo que modela um conector Reo FIFO. Esse conector é composto por um nó de origem, que aceita todos os dados, um *buffer* interno FIFO (*first-*

Listing 1. Conector FIFO (c) da figura 1 modelado no nuXmv

```
1 MODULE fifo(time)
2 VAR
3     cs : {q1,p1};
4     ports: portsModule;
5 ASSIGN
6     init(cs) := {q1};
7 TRANS
8     (((cs = q1) & (ports.c[time] = NULL) & (ports.a[time] = 0)) -> next(cs) = p1) &
9     (((cs = p1) & (ports.a[time] = NULL) & (ports.c[time] = 0)) -> next(cs) = q1);
```

in first-out) e um nó de sumidouro, por onde os operadores desse nó conseguem os dados do *buffer*. O trecho de código apresenta a parte mais importante do modelo nuXmv, um módulo onde estão codificadas os estados do conector, assim como suas transições.

Figura 2. Fluxo do compilador deste projeto.

4. Conclusões

O processo de validação formal, importante quando tratamos de sistemas críticos, é tipicamente árduo e complexo. Portanto, este projeto trata de um compilador de um programa Reo para o verificador de modelos nuXmv, cujo objetivo é facilitar e agilizar essa tarefa de validar formalmente propriedades a cerca desses sistemas.

Atualmente o processo do compilador é basicamente manual, no sentido em que cada etapa é feita separadamente. Pretende-se automatizar a integração dessas etapas em um mesmo programa, além de desenvolver uma interface para facilitar seu uso.

Referências

Arbab, F. (2004). Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366.

Arbab, F., Baier, C., Rutten, J., and Sirjani, M. (2004). Modeling component connectors in reo by constraint automata: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 97:25–46.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.

Bochot, T., Virelizier, P., Waeselynck, H., and Wiels, V. (2009). Model checking flight control systems: The airbus experience. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 18–27. IEEE.

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S. (2014). The nuxmv symbolic model checker. In Biere, A. and Bloem, R., editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer.

Gerhart, S., Craigen, D., and Ralston, T. (1994). Case study: Paris metro signaling system. *IEEE Software*, 11(1):32–28.

Knight, J. C. (2002). Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–550. ACM.