Gerenciamento Eficiente de Recursos em Sistemas Embarcados

Roger K. Immich, Diego L. Kreutz e Antônio A. Fröhlich
Universidade Federal de Santa Catarina
Laboratório de Integração Software/Hardware
Florianópolis, Brasil, PO Box 476 88049-900
{roger,kreutz,guto}@lisha.ufsc.br

Abstract

Classical strategies for resource management in operating systems are often complex and innapropriate for embedded systems. Implementations for these strategies may use either virtual function tables or long conditional structures to provide transparent access to different resources. This overhead is unacceptable for embedded systems. The EPOS operating system provides flexible and transparent access to resources for applications without incurring in unnecessary overhead. Metaprogrammed structures are used to predict, according to application usage and in compile time, whether a resource must use a polimorphic representation or may be accessed through direct calls. This way, virtual function tables are only used in the system when strictly necessary, and thus saving resources. In this article, we show that this strategy is a viable alternative for resource management in embedded systems.

Keywords: Resource Management, Static Meta-programmming, Operating Systems, Embedded Systems

Resumo

Estratégias clássicas de gerenciamento de recursos em sistemas operacionais são complexas e inapropriadas para sistemas embarcados. Muitas vezes, as implementações utilizam métodos virtuais(polimorfismo), com o objetivo de alcançar transparência e reusabilidade de código, e/ou longas estruturas condicionais, que invariavelmente ocupam blocos de memória e processamento desnecessariamente, o que pode não ser aceitável em sistemas embarcados. O gerenciamento de recursos no EPOS(Embedded Parallel Operating System), é realizado de forma transparente a aplicação, flexível e sem adicionar sobrecarga desnecessária ao sistema. Através da utilização da técnica de metaprogramação estática é possível predizer em tempo de compilação se haverá necessidade da utilização de polimorfismo, ou se estes poderão ser substituídos por chamadas diretas. Dessa forma, somente serão utilizadas chamadas indiretas quando necessário, provendo economia de recursos. A utilização desta técnica mostrou-se uma alternativa viável para sistemas embarcados, provando que é possível utilizá-la isoladamente no sistema, continuando com a flexibilidade original e provendo as otimizações necessárias.

Palavras-chave: Gerenciamento de recursos, Metaprogramação estática, Sistemas Operacionais, Sistemas Embarcados

1 Introdução

Estratégias clássicas de gerenciamento de recursos em sistemas operacionais são, complexas e dependentes de um domínio de aplicação. Um dos principais propósitos de sistemas operacionais de uso geral é oferecer mecanismos básicos para a melhor utilização do hardware em questão, fazendo com que a aplicação execute da melhor forma possível. Devido ao fato de serem sistemas de uso geral, existe uma grande quantidade de dispositivos que necessitam de gerenciamento.

A fim de prover facilidades com reusabilidade e transparência do código, muitas vezes, opta-se pela utilização de polimorfismo, através da implementação de métodos virtuais (que realizam chamadas indiretas).

Outra forma muito utilizada para a implementação de estratégias de gerenciamento é através de estruturas condicionais. Ambas estratégias apresentadas, invariavelmente ocupam blocos de memória e ciclos de clock desnecessariamente, seja por resolver dependências somente em tempo de execução (obrigando a permanência de todos os recursos na memória mesmo que não sejam utilizados) ou pela necessidade de realizar múltiplas operações para a chamada de métodos virtuais, características estas que podem ser pouco aceitáveis em sistemas embarcados.

Seja qual for a abordagem utilizada incorre-se no dualismo entre sacrifício de recursos ou possibilidade de reuso do código. O primeiro pode ser um componente crítico em sistemas dedicados de pequeno porte, enquanto que o segundo pode ser pouco desejado entre desenvolvedores e mantenedores de sistemas. O reuso é uma das bases e premissas das linguagens de programação orientadas a objetos. Por outro lado, em sistemas embarcados tanto o reuso quanto o baixo consumo de recursos costumam ser requisitos essenciais. Logo, meios que possibilitam tanto o reuso de código quanto a menor sobrecarga possível sobre o sistema, são desejáveis.

O gerenciamento de recursos no EPOS é realizado de forma transparente a aplicação, flexível e sem adicionar sobrecarga desnecessária ao sistema. Através da utilização da técnica de metaprogramação estática é possível predizer em tempo de compilação se haverá necessidade da utilização de polimorfismo (com métodos virtuais) ou se estes poderão ser substituídos por chamadas diretas, desta forma, somente onde for necessário será utilizado o polimorfismo eliminando-o do resto do código, provendo economia de recursos.

A sessão 2 apresenta a estratégia de gerenciamento de recursos no EPOS. Na sessão 3 estão descritos os experimentos realizados e os resultados alcançados. Trabalhos correlatos são exibidos na sessão 4. Por último, na sessão 5 são discutidos os resultados obtidos.

2 Estratégia de gerência de recursos metaprogramada

A gerência de recursos em sistemas operacionais é um dos pontos cruciais para a evolução e manutenção destes sistemas. Em especial, quando se trata de sistemas orientados à aplicação, o gerenciamento dos diferentes componentes e recursos que podem fazer parte de uma determinada instância, para um determinado ambiente, é de vital importância. O usuário deve ter a possibilidade de selecionar os componentes desejados de maneira transparente e simples.

Um modo de atingir o gerenciamento de recursos de forma transparente e simples é através do uso de metaprogramação. Os metaprogramas permitem que a escolha dos componentes mais adequados à aplicação do usuário seja feita em tempo de compilação (estática) ou em tempo de execução (dinâmica). No caso de sistemas embarcados, que normalmente carecem de recursos computacionais em relação a computadores de uso geral, a opção pela resolução estática dos componentes a serem incluídos na instância final do programa torna-se a mais atrativa e desejada.

Com o objetivo de prover flexibilidade e ao mesmo tempo eficiência no gerenciamento de recursos, o EPOS é provido de uma biblioteca de metaprogração especialmente desenvolvida para ele[5]. Um dos recursos oferecidos por essa biblioteca, são as funções metaprogramadas de estruturas condicionais *IF-THEN-ELSE*, bem como a operação *EQUAL*, descritas na literatura atual [11]. Dentro do conjunto de funções metaprogramadas algumas são especialmente desenvolvidas para o gerenciamento de recursos do sistema, porém não restrito a ele, pois elas podem ser aplicadas e utilizadas onde houver necessidade.

Como ilutrado na figura 1, o metaprograma para gerenciamento de recursos no EPOS é aplicado em tempo de compilação. Através da analise da aplicação ele identifica os tipos de componentes que satisfazem os requisitos esperados pelo usuário. A segunda fase passa pela identificação dos componentes específicos que farão parte da instância final do sistema. Nessa fase, é objetivo do metaprograma eliminar todas as funções virtuais possíveis, reduzindo o código que seria necessário para gerenciar recursos em tempo de execução. Por exemplo, supondo que a aplicação do usuário utilize duas instâncias do tipo placa de rede. O desenvolvedor apenas instancia duas placas de rede e na hora da compilação, o metaprograma irá identificar o tipo placa eliminando o polimorfismo, substituido as chamadas indiretas por chamadas diretas às instâncias de placas de rede disponíveis para a plataforma alvo.

Em um exemplo prático, considerando três plataformas hipotéticas "A","B"e "C"e também duas placas de rede "X"e "Y". O usuário poderia utilizar um código semelhante ao apresentado na figura 2 para diferentes plataformas alvo. Caso a aplicação fosse compilada para a plataforma hipotética A, contendo duas placas de rede X, o polimorfismo dos objetos nic0 e nic1 serão substituidos por chamadas diretas às instâncias das placas X. O mesmo processo seria repetido para uma suposta arquitetura B, que disponibiliza duas placas de

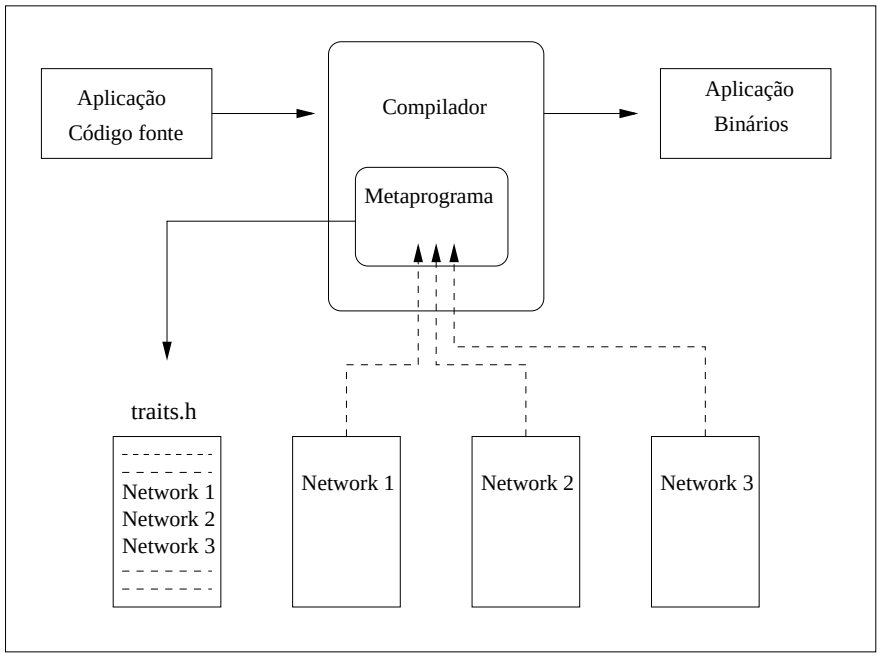

Figura 1: Como funciona o metaprograma

```
NIC nic0(0);
NIC nic1(1);

//thead 0
while(1){
    // ...
    nic0.send(BROADCAST, PROTOCOL, "A", 1);
}

//tread 1
while(1){
    // ...
    nic1.send(BROADCAST, PROTOCOL, "A", 1);
}
```

Figura 2: Aplicação de exemplo

rede Y. A grande vantagem é que a aplicação do usuário continua a mesma, transparente e com um alto grau de reusabilidade de código. Por outro lado, o polimorfismo não poderia ser eliminado para a plataforma C, pois, ela dispõe de uma placa de rede X e uma Y. Neste caso, somente é possível determinar quais chamadas referem-se a qual placa de rede em tempo de execução.

2.1 Funções metaprogramadas para gerenciamento de recursos

Uma das metafunções para o gereciamento de recursos chama-se *LIST*. A *LIST* é utilizada para gerar uma metalista, existente somente em tempo de compilação, dos recursos disponíveis no sistema. As informações que servem de base para a geração desta lista são encontradas em um arquivo que contém detalhes de configurações e características de todos os dispositivos que podem ser utilizados para uma determinada plataforma de hardware (traits.h). A Figura 3 ilustra um trecho do arquivo de configuração refente às placas de rede disponíveis para a arquitetura PC (Intel 386) no EPOS.

No exemplo apresenta-se três dispositivos de rede, PCNET32, E100 e C905, sendo que, cada um deles possui uma definição própria quanto ao número de buffers de envio e recebimento. Essas informações serão pertinentes em tempo de compilação, quando da definição de uma instância do respectivo componente.

```
template <> struct Traits<PC_NIC>: public Traits<PC_Common>
{
    typedef LIST<PCNet32, PCNet32> NICS;

    static const unsigned int PCNET32_UNITS = NICS::Count<PCNet32>::Result;
    static const unsigned int PCNET32_SEND_BUFFERS = 8;
    static const unsigned int PCNET32_RECEIVE_BUFFERS = 8;

    static const unsigned int E100_UNITS = NICS::Count<E100>::Result;
    static const unsigned int E100_SEND_BUFFERS = 8;
    static const unsigned int E100_RECEIVE_BUFFERS = 8;

    static const unsigned int C905_UNITS = NICS::Count<C905>::Result;
    static const unsigned int C905_SEND_BUFFERS = 8;
    static const unsigned int C905_RECEIVE_BUFFERS = 8;
};
```

Figura 3: Descrição das placas de rede no arquivo traits

```
template<typename NICS>
class Meta_NIC {
  //...
  public:
  typedef typename IF<polymorphic,
          NIC_Base,
          typename NICS::template
                   Get<0>::Result>::Result Base;
  // ...
};
```

Figura 4: Estutura condicional para remoção do polimorfismo

Outra metafunção é a *polymorphic*, que retorna um valor booleano obtido através do percorrimento e análise, em tempo de compilação, da metalista gerada anteriormente. Quando houver mais de um dispositivo para a mesma finalidade (por exemplo, placas de rede) de tipos diferentes (PCNet32, E100 e C905, como apresentado na Figura 3) incluídos na metalista, essa função retorna *TRUE* indicando que será necessária a utilização de polimorfismo. Quando a metalista tiver somente um elemento, ou vários elementos com dispositivos do mesmo tipo, essa função retorna *FALSE*, indicando que o polimorfismo pode ser eliminado e dessa forma podem ser realizadas chamadas diretas aos metodos dos dispositivos presentes na lista.

A utilização da metafunção *polymorphic* é realizada através de uma estrutura condicional metaprogramada, com o objetivo de definir o valor da variável *Base* (figura 4). Através do exemplo, essa variável servirá de referência para executar operações com as placas de rede. A variável *Base* poderá armazenar um apontador para métodos virtuais, no caso da utilização de polimorfismo, ou para um dispositivo, possibilitando chamadas diretas e consequentemente eliminando o polimorfismo.

3 Resultados da gerência de recursos no EPOS

Com o objetivo de testar a eficiência do gerenciamento de recursos no EPOS, foi utilizada uma aplicação simples que envia o caracter "A"para a interface de rede. Duas configurações de plataformas alvo foram utilizadas uma delas com somente uma interface de rede, e a outra com duas interfaces de rede distintas. Este experimento mostra a quantidade memória que é possível economizar com o gerenciamento de recursos, quanto este substitui as funções virtuais por chamadas diretas aos métodos.

A aplicação de exemplo foi desenvolvida e compilada para a aquitetura IA-32. A table 1 apresenta a quantidade de memória ocupada para o *Caso A* (não polimórfico) e para o *Caso B* (polimórfico). Estes valores demonstram que o gerenciamento de recursos no EPOS realizado em tempo de compilação, possibilita a melhor utilização da memória disponível, alocando espaço somente para os recursos que realmente serão utilizados na execução da aplicação em questão.

	Caso A	Caso B
.text	19308	19668
.data	88	88
.bss	432	432

Tabela 1: Tamanho em bytes no Caso A(uma interface) e B (duas interfaces diferentes)

	Teste Caso A	Teste Caso B
Tempo	13.6	14.61

Tabela 2: Tempo de acesso a interface de rede em microsegundos

No segundo experimento, as comparações de tempo de acesso à recursos dizem respeito a testes utilizando o EPOS. As medições foram realizadas adicionando-se um contador de tempo logo após a chamada do método *send* pela aplicação, finalizando depois da chamada do respectivo método *send* do drive da placa de rede. Foram executadas 100 interações, com 1.000.000 de leituras para amostragem. Os testes foram realizados no emulador VMWare em um computador Athlon64 3000+. A Tabela 2 apresenta a média dos tempos de acesso em microsegundos para ambos os casos.

Com as estratégias de gerenciamento de recursos no EPOS, o polimorfismo é eliminado sempre que possível. Desta forma, quando o sistema possuír somente um tipo de placa de rede(uma interface de rede ou duas do mesmo tipo), não existirá polimorfismo(*Caso A*), por isso, o tempo que a chamada levará da aplicação até o drive da placa de rede será menor, como pode ser visto na Tabela 2. Isso acontece pois os métodos virtuais foram substituídos por chamadas diretas. O polimorfismo só voltará a aparecer quando estiverem presentes no sistema duas placas de rede distintas(*Caso B*), levando assim mais tempo para realizar a mesma chamada.

Através da utilização das estratégias de gerenciamento de recursos do EPOS é possível prover economia de memória e melhorar o tempo de acesso a periféricos. A economia de memória é alcançada através da eliminação do que não será necessário a execução da aplicação, deixando o código final mais enxuto. A melhora no tempo de acesso à periféricos é obtida substituíndo-se, em tempo de compilação, métodos virtuais por chamadas diretas.

4 Trabalhos correlatos

Boost [7] C++ é uma biblioteca de metafunções que pode ser utilizada para os mais diversos propósitos. Essa biblioteca pode ser utilizada, por exemplo, para criar diferentes tipos de dimensões, tentando evitar possiveis operações incoerentes entre diferentes dimensões [1].

Boost busca agrupar abstrações importantes da programaçao genérica e funcional para construir um conjunto de ferramentas, por sua vez fáceis de usar, capaz de tornar a metaprogramação através de templates prática o suficiente para ambientes reais. Em sua concepção a biblioteca Boost é fortemente influenciada pela biblioteca STL (*Standard Template Library*). Enquanto a primeira é resolvida em tempo de compilação a segunda é resolvida em tempo de execução, o que proporciona um diferencial que poderá fazer a diferença de acordo com o contexto. Uma aplicação para um sistema embarcado poderia exigir a resolução da maior parte (ou todas) as dependências em tempo de compilação (pois sistemas embarcados normalmente possuem recursos limitados, sendo desprovidos de áreas especiais de armazenamento - como discos rígidos - e possuindo reduzidas quantidades de memória principal). Por outro lado, uma aplicação usual, destinada a microcomputadores de uso geral, pode não ter nenhuma restrição quanto ao uso de bibliotecas (metafunções, classes, algoritmos, etc) que serão resolvidos/carregados em tempo de execução.

A mineração de aspectos [3, 6, 13, 9] é uma linha de pesquisa que tem por objetivo a análise e desenvolvimento de recursos que permitam a identificação de aspectos candidatos a refatoração em sistemas existentes, baseados em linguagens de programação orientadas a objetos. A idéia básica é a busca por aspectos espalhados pelo código tentando agrupa-los por similaridade no intuito de separar as partes comuns, proporcionando a refatoração do software.

Apesar das várias pesquisas na área de mineração e refatoração de aspectos, ainda inexistem soluções ou

ferramentas capazes de identificar e isolar todos os aspectos passíveis de refatoração em um software. Essa é uma tarefa bastante complexa, que exige um alto grau de conhecimento acerca das linguagens de programação e suas capacidades de representação e organização de dados e algoritmos. Complementar a isso ainda existe a complicação oriunda das diferentes formas que um programador pode implementar e representar uma solução para um determinado problema. Esses fatores aumentam a dificuldade de identificação e refatoração de aspectos tornam-se ainda mais complicados em linguagens de programação flexíveis e ricas em recursos.

As linguagens de programação orientadas a objetos trazem consigo mecanismos que facilitam o reuso de código. Entre eles podem ser citados o polimorfismo, herança, metaprogramação, encapsulamento e linkagem dinâmica. Todos esses recursos buscam facilitar a vida dos desenvolvedores e/ou disponibilizar níveis de flexibilidade antes inatingíveis. No contexto de sistemas embarcados dois desses mecanismos são ao mesmo tempo desejáveis, porém inviáveis, o polimorfismo e a linkagem dinâmica. Esses mecanismos fornecem meios atrativos para a resolução de diferentes problemas computacionais, no entanto, trazem consigo sobrecargas (de memória e processamento) pouco aceitáveis na maioria dos sistemas embarcados. Uma situação ideal seria onde o desenvolvedor pudesse fazer uso desses recursos em tempo de codificação e os mesmos fossem eliminados em tempo de compilação, evitando o consumo extra de recursos em tempo de execução.

Na literatura atual existem algumas propostas de refatoração, browsers e mineração de aspectos. Porém estão em fase de desenvolvimento e/ou experimentação e ainda não é possível constatar, com o resultado obtido até o momento, se estas poderão ser aplicadas em sistemas embarcados. A refatoração de aspectos foi proposto por Hanenberg[8] e tem como objetivo reorganizar os aspectos, favorecendo a flexibilidade e a legibilidade do código. No entanto não elimina o polimorfismo, e desta forma, não é a mais adequada para sistemas embarcados.

A proposta de utilização de browsers de aspectos com o objetivo de auxiliar no reconhecimento de padrões e facilitar a navegação entre eles, também já foi explorada[6, 10]. No entanto ela requer que o desenvolvedor tenha um grande conhecimento tanto do paradigma de programação orientado a aspectos, como da ferramenta que será utilizada e igualmente a refatoração, ela não provê a identificação de pontos onde o polimorfismo pode ser substituído por chamadas diretas.

Mineração de aspectos[12] utiliza a técnica analise formal de conceitos[14] para encontrar automaticamente conjuntos de código com funcionalidades transversais ao sistema em métodos, classses e hierarquia de classes, que estão relacionados de alguma forma. Esta abordagem esta em fase inicial de experimentação e ainda não apresenta resultados expressivos para o caso de eliminação de polimorfismo.

O objetivo básico das funções virtuais (como o `virtual` em C++) é facilitar o reuso de código [2, 4]. No entanto, esse recurso pode produzir um grande aumento de complexidade. Nesse sentido, algumas iniciativas vêm sendo feitas na busca por soluções que visam reduzir, de forma transparente, o impacto no desempenho e tamanho do código quando do uso de funções virtuais. O objetivo de pesquisas como [2] e [4] é prover ferramentas para a análise e redução de código desnecessário. Alguns resultados mostram que é possível resolver até 71% as funções virtuais e redução do código compilado em 25%. Foi também demonstrado que alguns algoritmos são bastante rápidos e poderiam ser boas oportunidades para a inclusão em compiladores C++.

5 Conclusão

A utilização da técnica de metaprogramação estática com o objetivo de prover otimizações nas estratégias de gerenciamento de recursos do EPOS mostrou-se uma alternativa viável para sistemas embarcados. Apesar das dificuldades impostas pela metaprogramação tais como o aumento da complexidade, dificuldade de depuração e maior tempo de compilação, foi constatado que é possível utilizá-la isoladamente no sistema, apenas nos locais onde ela se faz necessária, mantendo a estrutura original nas demais partes, continuando com a flexibilidade original do sistema e provendo as otimizações necessárias.

Entre as otimizações alcançadas, estão a economia de memória e o tempo de acesso à dispositivos. A economia de memória foi obtida eliminando-se do código final do sistema todos os recursos que não serão utilizados pela aplicação em execução, deixando o sistema especializado para aquele caso. A redução no tempo de acesso aos dispositivos foi possível substituído-se funções virtuais, utilizadas no polimorfismo, por chamadas diretas aos métodos.

Referências

[1] David Abrahams and Aleksey Curtovoy. A Deeper Look at Metafunctions, Aug 2004. `http://www.artima.com/cppsource/metafunctions.html`.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM Press.

[3] Arie Van Deursen, Leon Moonen, and Marius Marin. Aspect mining and refactoring, November 05 2003.

[4] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–323, New York, NY, USA, 1996. ACM Press.

[5] Antônio Augusto Medeiros Froehlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, 1 edition, 2001.

[6] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, 3, 2000.

[7] Aleksey Gurtovoy and David Abrahams. The Boost C++ Metaprogramming Library. Technical report, 2005.

[8] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software, 2003.

[9] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 135–146. ACM Press, March 2005.

[10] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[11] Robert Robson. *Using the Stl - The C++ Standard Template Library*. Springer-Verlag, 2 edition, 1999.

[12] Tom Tourwe and Kim Mens. Mining aspectual views using formal concept analysis. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04)*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Arie van Deursen, Marius Marin, and Leon Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotdraw, March 05 2005. Comment: 25 pages.

[14] Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, 83:445–470, September 1981.