2
Trabalhos relacionados

Este capítulo apresenta os principais trabalhos relacionados à pesquisa realizada para esta tese. Primeiramente, vamos apresentar definições e abordagens utilizadas no processo de ensino e aprendizado de raciocínio computacional. Em seguida, vamos explorar tecnologias de apoio e mecanismos de colaboração no contexto dos ambientes de programação e documentação ativa. Depois disso, trataremos do *software* como forma de participação social e comentaremos as implicações dessa tendência. Por último, faremos uma relação entre o que está reportado nos trabalhos e a pesquisa desta tese.

2.1.
Definições e abordagens do ensino e aprendizado de Raciocínio computacional

O termo 'raciocínio computacional' (*computational thinking*) foi primeiramente definido por Wing em 2006. A autora defende que o raciocínio computacional é uma habilidade fundamental para todos, não apenas para graduados na área de computação, e envolve a formulação, a compreensão e a solução de problemas. O raciocínio computacional é a capacidade de usar a abstração e a decomposição para resolver tarefas complexas ou projetar sistemas complexos. A autora afirma que raciocínio computacional é sobre ideias, não sobre artefatos; é sobre entender as questões de modo conceitual, não é sobre programar, pois requer raciocínio em múltiplos níveis de abstração (Wing, 2006).

A Sociedade Internacional de Tecnologia na Educação publicou em sua página[8] uma definição operacional para '*computational thinking for k-12*[9] *education'*. No *website*, o raciocínio computacional é definido como um processo de resolução de problemas que inclui as seguintes características, mas não se limita a elas (ISTE, 2014): formular problemas de modo a nos habilitar a usar o

[8] https://www.iste.org/learn/computational-thinking/ct-operational-definition

[9] Denominação usada nos Estados Unidos para a escolaridade desde o pré-escolar até o último ano do ensino médio.

computador e outras ferramentas para resolvê-los; organizar e analisar dados de forma lógica; representar dados através de abstrações, tais como modelos e simulações; automatizar soluções através de pensamento algorítmico (como uma sequência de passos ordenados); identificar, analisar e implementar possíveis soluções com o objetivo de combinar passos e recursos de modo mais efetivo e eficiente; generalizar e transferir esse processo de resolução para uma ampla variedade de problemas.

A empresa Google também está envolvida na questão do raciocínio computacional no currículo K-12 para o aprendizado do que eles definem como um conjunto de habilidades para resolução de problemas e técnicas que engenheiros de *software* usam para escrever programas que estão em aplicações como busca, e-mail e mapas. Em sua página[10] '*Exploring Computational Thinking*', eles disponibilizam uma variedade de recursos para ajudar professores a aprender os conceitos básicos de raciocínio computacional e a incorporar as lições criadas por eles em sua atividade didática. Eles incentivam os professores, enfatizando a importância de atingir todas as pessoas, pois a necessidade desses conhecimentos para usar efetivamente os recursos tecnológicos é cada vez maior.

Barr e Stephenson (2011) dizem que uma definição de raciocínio computacional é útil quando ela está somada a exemplos que demonstrem como o assunto deve ser incorporado nas aulas. Eles organizaram conceitos e capacidades importantes para o raciocínio computacional com exemplos de utilização em disciplinas como ciências, matemática, artes da linguagem, estudos sociais e ciência da computação. Os conceitos destacados foram a coleta de dados, a análise de dados, a representação de dados, a decomposição de problemas, a abstração, os procedimentos e algoritmos, a automação, a paralelização e a simulação. Por exemplo, o conceito de representação de dados pode ser visto na ciência da computação como o uso de listas ou pilhas. Na matemática, este conceito é observado quando são usados gráficos de barras. Em ciências e estudos sociais, sumarizar os achados de um estudo é um modo de representar dados. Representar padrões de diferentes tipos de sentenças é uma aplicação do conceito de representação de dados na disciplina de artes da linguagem.

[10] https://www.google.com/edu/computational-thinking/

No relatório do NRC (2010) sobre raciocínio computacional, o tema foi observado sob diferentes perspectivas. Segundo eles o raciocínio computacional pode ser visto como: um conjunto de habilidades, conceitos, aplicações e ferramentas; uma linguagem para expressar ideias através da programação; uma forma de gerenciar a automação de abstrações; e uma ferramenta cognitiva.

Lee *et al.* (2011) reportaram exemplos de como o raciocínio computacional aparece entre jovens com diferentes bases socioeconômicas e culturais dentro e fora da escola. Os exemplos foram distribuídos entre as áreas de modelagem e simulação, robótica e desenvolvimento e *design* de jogos. Os autores indicaram que para apoiar o desenvolvimento do raciocínio computacional era importante usar ambientes de programação ricos, que permitem inspecionar e manipular abstrações e onde o aprendiz deixa de ser um mero usuário para se tornar um criador. Além disso, eles propõem usar um mecanismo de progressão em três estágios (usar, modificar e criar) para motivar os jovens diante dos ambientes de programação indicados.

Há um movimento chamado *Computer Science Unplugged*[11] que defende o ensino de ciência da computação por meio de jogos, quebra-cabeças e atividades que não fazem uso do computador (Bell *et al.*, 2009; Bell *et al.*, 2005). Eles têm como o objetivo divulgar a Ciência da Computação para o público jovem como uma disciplina interessante e estimulante. Entre os tópicos de computação abordados pelos fundadores do movimento estão a representação de dados, algoritmos, coloração de grafos e outros, mas as atividades propostas não envolvem recursos computacionais e incentivam a colaboração entre os aprendizes.

2.2. Tecnologias de apoio e colaboração

Como foi comentado na introdução, as principais tecnologias de apoio e colaboração relacionadas à pesquisa desta tese estão divididas entre: ambientes de programação e troca de experiências; ferramentas de auxílio à compreensão e depuração de programas; ambientes de distribuição e compartilhamento de

[11] http://www.csunplugged.org/

recursos; e documentação ativa. Cada uma dessas tecnologias será detalhada nas subseções a seguir.

2.2.1. Ambientes de programação, compartilhamento de recursos e troca de experiências

É grande a variedade de ambientes utilizados por escolas de ensino médio e fundamental voltados para a aquisição do raciocínio computacional. As abordagens variam, mas aulas de robótica, programação de jogos e animações e são amplamente difundidas. Nesta subseção apresentamos ambientes de programação e, em alguns casos, suas extensões na *Web* que apoiam os utilizadores fornecendo material didático e funcionam como meio para troca de experiências.

Papert (1980) oferecia uma alternativa de uso do computador direcionada para crianças. Ao invés de entregar os programas de computador às crianças, ele propôs que as crianças pudessem criar programas. Para o autor, um novo ambiente de desenvolvimento demandava um contato livre entre as crianças e os computadores, enquanto elas aprendiam a programar estavam aprendendo a pensar de um modo diferente. Naquela época, surgia a ideia do ensino de programação como um veículo para desenvolver habilidades de resolução de problemas, mas ainda não havia pesquisas para sustentar essa suposição. O ambiente LOGO foi uma dos primeiros construídos e utilizados no ensino de programação para crianças. A sua versão inicial era formada por um robô, similar a uma tartaruga, que obedecia a comandos e se movia formando desenhos geométricos na sua base.

A robótica é explorada até hoje no ensino de programação e raciocínio computacional voltado para resolução de problemas. Trabalhar com robótica envolve mais do que construir artefatos físicos, para dar vida a um robô é preciso saber criar programas de computador, ou seja, planejar algoritmos ou sequências de instruções que permitam que os robôs se movam, percebam e respondam ao ambiente (Bers, 2010). Por exemplo, o NXT LEGO® Mindstorms Robotics disponibiliza um dispositivo computacional físico que pode ser usado para resolver problemas usando motores e sensores em um ambiente de programação baseado em ícones (Imberman *et al.*, 2014).

Muitos trabalhos abordam a construção e uso de jogos no ensino de raciocínio computacional. O Scratch (Maloney *et al.*, 2010), um dos programas mais conhecidos nesse contexto, é uma ferramenta de programação visual que permite aos usuários aprender programação enquanto constroem jogos, histórias e simulações. O programa tem um grande apelo colaborativo e tem como missão "*ajudar os jovens a aprender a pensar de maneira criativa, refletir de maneira sistemática, e trabalhar de forma colaborativa - habilidades essenciais para a vida no século 21*", como está descrito seu *website*[12]. Wolz *et al.* (2010) usaram o Scratch como ferramenta em um curso de jornalismo interativo no ensino médio liderado por professores de artes e informática. Os estudantes realizam entrevistas e criam histórias usando texto, vídeo e animações no Scratch. Resultados preliminares indicaram como os estudantes percebem o que significa 'programar' e como ficam confiantes sabendo que são capazes de fazê-lo.

O Scratch tem um ambiente adicional de apoio e troca de experiências na *Web*[13]. No *site* é possível ver exemplos de programas, realizar testes, fazer *upload* dos programas e executar programas enviados por outros usuários. Em 2014, mais de seis milhões de projetos estavam compartilhados de acordo com o *site* da comunidade. Há espaços especialmente dedicados aos educadores e pais de estudantes. Entre os mecanismos mais importantes do *site* estão o fórum e a página de ajuda. O fórum permite o contato com a comunidade e disponibiliza diversos tópicos com discussões. A página de ajuda fornece material para iniciantes, guias, cartões e tutoriais em vídeo. Qualquer usuário pode baixar um projeto, fazer alterações e enviá-lo para o site em uma nova versão, no Scratch isso é chamado de *remix,* isto é, fazer o *download* de comportamentos de objetos específicos para usar em suas construções de mundos virtuais. Dados de 2010 informam que 28% dos projetos disponíveis são *remixes* e 68% são baseados em outros projetos (Monroy-Hernández e Hill, 2010). Brennan e Resnick (2012) destacam o reuso e o *remix* como uma das práticas recomendadas no desenvolvimento do raciocínio computacional.

O Greenfoot é um ambiente de desenvolvimento educacional integrado que tem como objetivo o ensino e aprendizado de programação (Kölling, 2010). Seu público-alvo é composto por estudantes a partir dos 14 anos, sendo recomendado

[12] http://scratch.mit.edu/about/
[13] http://scratch.mit.edu/

para o nível de ensino médio e universitário. A ferramenta combina uma visualização gráfica a uma linguagem de programação textual orientada a objetos. Os objetivos do Greenfoot são pontuados diante de duas perspectivas: dos estudantes e dos professores.

Para conquistar os estudantes, o Greenfoot foi projetado com os seguintes objetivos (Kölling, 2010): permitir que os usuários atinjam suas metas sem se prenderem a tarefas desnecessárias; habilitar os estudantes a descobrir as funcionalidades necessárias para atingir os seus objetivos; permitir incluir facilmente gráficos, animações e sons para atrair os aprendizes; ser flexível para suportar diferentes cenários de uso; permitir o desenvolvimento em pequenos passos com oportunidades de execução e observação com *feedback* visual do resultado; estar disponível para diferentes tipos de sistemas operacionais e *hardwares*; permitir a interação social para o compartilhamento com a comunidade incentivando a criatividade; ser extensível e permitir conexões com outros sistemas (serviços *Web*, bancos de dados, e outros) e outras plataformas e componentes de *hardware* (como controles de jogos).

Ainda segundo Kölling (2010), para apoiar os professores, o Greenfoot disponibiliza: a visualização, no sentido de que os conceitos importantes para o paradigma de programação devem ser diretamente visualizados; as interações no ambiente devem ilustrar conceitos de programação: todas as representações de princípios devem refletir corretamente o modelo de programação em que foram baseadas; o foco nos conceitos, deixando a questão sintática em segundo plano; mecanismos para evitar uma sobrecarga cognitiva, qualquer complexidade adicional à tarefa principal é escondida; auxílio aos professores com materiais didáticos.

O Greenfoot também tem um *site* na *Web*[14] onde estão disponibilizados diversos recursos para estudantes e professores. Os usuários têm acesso a tutoriais, livros, vídeos. Os professores têm uma página[15] restrita para educadores chamada Greenroom, cuja proposta é compartilhar experiências e material oferecendo uma plataforma de discussão e apoio. A página é um espaço para os professores entrarem em contato, encontrarem novos materiais e melhorarem o modo de ensinar.

[14] http://www.greenfoot.org/
[15] http://greenroom.greenfoot.org/door

Outra ferramenta para criação de jogos via uma linguagem de programação visual simples é o Kodu[16]. Ele foi desenvolvido pelo laboratório FUSE (*Future Social Experiences*) da Microsoft Research. A linguagem utilizada é considerada de alto nível e incorpora primitivas do mundo real como: colisão, cor e visão. O Kodu é recomendado para pessoas sem conhecimento em *design* ou programação a partir de 8 anos. O sistema pode ser utilizado com PC, com mouse e teclado, ou em Xbox 360, com controle para jogos. Os programas do Kodu são em três dimensões; a ideia é criar mundos, inserir personagens e construir as regras dos personagens através de condições e consequências.

Fristoe *et al.* (2011) utilizaram o Kodu para desenvolver uma pesquisa sobre como criar jogos baseados em relações dinâmicas, interações sociais e *storytelling*. As ferramentas de criação de jogos para iniciantes geralmente não oferecem mecanismos que facilitem essas relações. O trabalho descreveu uma extensão do Kodu com novos comandos para aumentar a expressividade dos programas de modo simples e fácil de entender. Os resultados indicaram que as crianças participantes se engajaram para utilizar as novas funcionalidades, e o mais importante, a nosso ver, é que elas explicitamente solicitaram a adição de um comando "falar" que fosse ilimitado, de modo que qualquer frase pudesse ser dita pelos personagens do programa. Isso é importante porque mostra como esta ferramenta poderia ser utilizada no contexto da autoexpressão através de *software*.

Na página do Kodu é possível encontrar recursos para utilizar a ferramenta sem o auxílio de professores. O site disponibiliza um curso com a duração de cinco semanas em arquivos que podem ser acessados livremente. Os professores também podem acessar materiais didáticos como apostilas e um canal no Youtube com vídeos exemplificando funcionalidades de projetos criados na ferramenta.

Ainda considerando ferramentas com visualizações mais sofisticadas, Alice é um ambiente de programação com gráficos interativos em 3D (Cooper *et al.*, 2000). Modelos de objetos são posicionados em um mundo virtual e a programação é similar à programação realizada em orientação a objetos. O usuário escreve *scripts* simples que controlam a aparência e o comportamento dos objetos. A ferramenta permite o aprendizado de conceitos fundamentais de

[16] http://www.kodugamelab.com/

programação no contexto de criação de animações para contar histórias e construção de jogos interativos.

O ambiente Alice tem uma extensão na *Web*[17] onde estão disponíveis para *download* a ferramenta nas suas diversas versões. O *site* também oferece material didático e acesso à comunidade de usuários do ambiente através de lista de e-mails, *blog*, página no Facebook e fórum. Além disso, é possível encontrar uma página como respostas a perguntas frequentes e *bugs* específicos de versões.

Baseada em Alice, surgiu a Storytelling Alice (Kelleher *et al.*, 2007), ferramenta mais direcionada para a criação de animações em vídeo. Segundo os autores, criar um vídeo animado e aprender a programar um computador pode ser fundamentalmente a mesma atividade. As duas ferramentas permitem aos usuários entrar em contato com conceitos de programação, incluindo laços, condicionais, métodos, parâmetros, variáveis, listas e recursão. O trabalho aponta que o uso da ferramenta Storytelling Alice pode motivar os estudantes a aprender programação.

O sucessor do Storytelling Alice é o ambiente The Looking Glass, que além de valorizar os aspectos relacionados à criatividade e expressão na criação das animações em 3D, tem como fundamento permitir o reuso de códigos. Segundo Gross *et al.* (2010), o ambiente permite que os usuários identifiquem a parte do código responsável por determinada funcionalidade, extraiam o código e o integrem em um novo contexto. Os autores realizaram um estudo exploratório confirmando que os usuários conseguem se apropriar da ferramenta construindo animações que implementam o reuso de código. Harms *et al.* (2012) criaram uma comunidade[18] para os usuários do ambiente. Além dos recursos típicos desse tipo de comunidade, a página exibe mensagens para fomentar o *remix*. Há grande incentivo à criação de histórias animadas, *remix* para aprendizado de novas habilidades e compartilhamento das histórias na *Web*.

Especialmente relacionado ao contexto desta pesquisa, é importante apresentar o projeto Scalable Game Design (SGD). Nos Estados Unidos, o SGD já atingiu mais de 10 mil participantes, superando o objetivo inicial que era alcançar 1200 estudantes em três anos[19]. O projeto tem como missão reinventar a ciência da computação em escolas públicas usando jogos e simulações. A estratégia foi

[17] http://www.alice.org/
[18] http://lookingglass.wustl.edu/
[19] Dados da wiki do SGD: http://sgd.cs.colorado.edu/wiki/Scalable_Game_*Design*_wiki

iniciar os cursos com a construção de jogos e partir para a transferência das habilidades de raciocínio computacional para a ciência.

Seu criador e coordenador, Repenning, em um de seus trabalhos apresenta uma lista de características que devem estar presentes em um curso cujo objetivo seja promover a aquisição de raciocínio computacional. Os itens envolvem a ferramenta, o currículo do curso e o treinamento dos professores (Repenning *et al.*, 2010): o estudante deve produzir um jogo rapidamente; o estudante deve fazer um jogo real que possa ser usado e que tenha um comportamento sofisticado, por exemplo, usando inteligência artificial; o currículo do curso deve oferecer a ajuda necessária para o gerenciamento das habilidades e desafios que acompanham a ferramenta; a ferramenta e o currículo do curso devem estar alinhados para oferecer a transferência do conhecimento adquirido para outros contextos do ensino; as atividades de construção de jogos devem ser acessíveis e motivacionais independente dos participantes; a ferramenta e o currículo do curso devem ser usados por todos os professores para ensinar todos os estudantes.

As pesquisas no SGD envolvem o uso de quatro sistemas: o AgentSheets (AS), o AgentCubes (Repenning e Ioannidou, 2006), a SGD Wiki e o SGD Arcade (Basawapatna *et al.*, 2010). O AS, na versão 3.0[20], é utilizado no SGD-Br. O AgentCubes é uma versão do AS para criação de jogos em três dimensões. A SGD Wiki[21] descreve o projeto e disponibiliza tutoriais sobre a construção de jogos e simulações, espaço com passo a passo para professores, informações sobre treinamentos e pesquisa. O SGD Arcade[22] é um sistema *Web* com infraestrutura de ambiente virtual de aprendizado com turmas que permite aos estudantes fazer *upload* e *download* dos jogos, além de jogar, comentar e classificar os jogos dos colegas. O SGD Arcade disponibiliza projetos, descrições e uma representação dos programas que objetiva a avaliação do aprendizado dos alunos.

Esta representação foi considerada em um estudo sobre mecanismos para avaliar o aprendizado de raciocínio computacional. Pesquisadores do SGD criaram uma ferramenta que gera um grafo para a visualização do que foi definido como 'padrões de raciocínio computacional' (Koh *et al.*, 2010). Quando um jogo é submetido ao SGD Arcade, uma análise semântica dos projetos é realizada

[20] Versão corrente do AgentSheets em abril de 2014
[21] http://sgd.cs.colorado.edu/wiki/Scalable_Game_Design_wiki
[22] http://scalablegamedesign.cs.colorado.edu/arcade/

automaticamente e um grafo específico do projeto é apresentado com o objetivo de indicar a existência de padrões de raciocínio computacional transferido dos jogos para simulações de ciências. Os padrões surgiram do exame de coleções de jogos e simulação construídos durante o projeto. A Figura 2.1 exemplifica o grafo resultante da análise de dois jogos criados no AS. Pode-se observar o aumento do uso de padrões como geração, absorção, colisão e outros. O grafo mostra que padrões são aplicados com base nos comandos utilizados e presume que o reuso dos padrões atesta a transferência do raciocínio computacional adquirido.

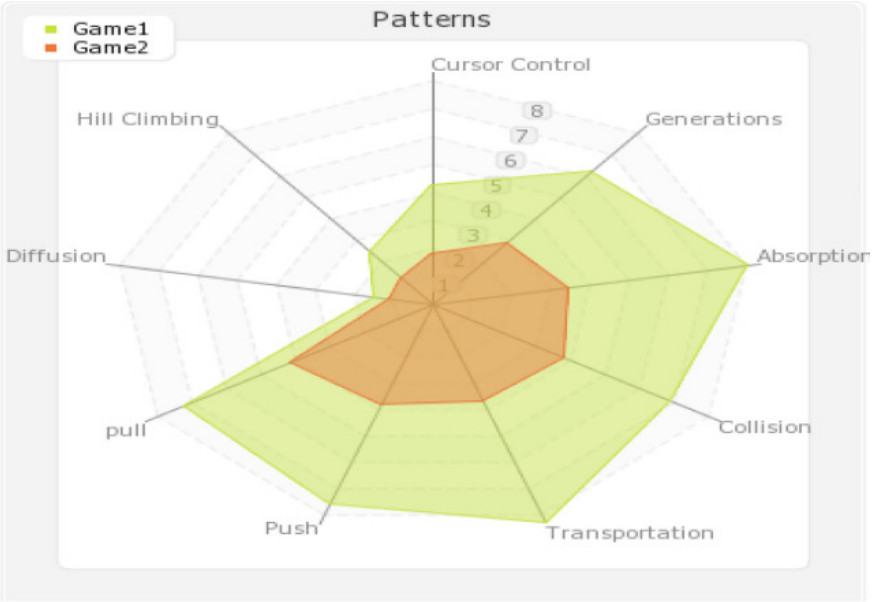

Figura 2.1 - Grafo de padrões de raciocínio computacional
Fonte: (Koh *et al.*, 2010)

2.2.2. Ferramentas de auxílio à compreensão e depuração de programas

Há um projeto sobre 'programação natural' liderado por Brad Myers na Universidade de Carnegie Mellon. Eles trabalham para tornar as linguagens de programação e os ambientes de desenvolvimento mais fáceis de aprender, mais eficientes e com menor probabilidade de erros. O objetivo deles é fazer com que as pessoas expressem suas ideias nos programas do mesmo modo que pensam sobre eles. Para isso, eles estudam como as pessoas pensam sobre tarefas de programação, usando uma abordagem completamente centrada no usuário (Myers *et al.*, 2004). Eles têm criado várias ferramentas de programação e depuração de

código. Por exemplo, o Euklas[23] (*Eclipse Users' Keystrokes Lessened by Attaching from Samples*), uma ferramenta que funciona como um *plug-in* do Eclipse para código em *JavaScript* onde é possível: destacar os erros de sintaxe no código; disponibilizar consertos rápidos com as alternativas possíveis; e visualizar longas explicações sobre o funcionamento de métodos.

Ko e Myers (2004) projetaram um mecanismo de depuração de programas dentro do ambiente Alice, denominado Whyline (*Workspace that Helps You Link Instructions to Numbers and Events*), baseados que foi baseada em uma espécie de depuração interrogativa. A ideia era que, em tempo de execução, o usuário podia acessar um menu com perguntas sobre eventos com os objetos presentes. Por exemplo, uma pergunta pode ser "*por que determinado objeto não pode ser redimensionado para o tamanho cinco*?" ou uma pergunta com negação "*por que determinado objeto não pode se mover três vezes para frente?*". Também é possível verificar um grafo que representa o programa em execução mostrando quais eventos foram disparados e o fluxo de controle lógico de um objeto. Neste estudo, dados empíricos de avaliação do uso da ferramenta mostraram que os usuários resolviam mais tarefas e em menor tempo, comparado com usuários que não tinham tal facilidade para depurar o código. Os autores continuaram este trabalho lançando versões do Whyline para Java (Ko e Myers, 2010), o que aumentou o escopo de aplicação da ferramenta.

Quando um programa é criado por usuários finais, não especializados em programação, problemas com confiabilidade e erros na programação são mais frequentes. Burnett *et al.* (2004) começaram a endereçar esse problema propondo uma abordagem chamada 'engenharia de *software* para usuários finais'. Entre as estratégias propostas pelos autores estão: o mecanismo *What You See Is What You Test* (WYSIWYT) que significa 'o que você vê é o que você testa', com ferramentas de ajuda para testes e localização de erros; o uso de assertivas; e um recurso de '*Surprise-Explain-Reward'* que seria como 'surpresa-explicação-recompensa' para motivar os usuários a utilizar os dispositivos disponibilizados. Os resultados indicaram que um ambiente de programação para usuários finais baseado nas estratégias citadas pode melhorar significativamente a habilidade dos usuários finais para criarem programas mais confiáveis e com menos erros.

[23] http://www.cs.cmu.edu/~euklas/literature.html

Outro mecanismo de programação por usuários finais que tem crescido nos últimos anos é a criação de *mashups*. *Mashups* são combinações de serviços na *Web* onde é possível processar a saída dos programas de diversos modos. Kuttal *et al.* (2013) abordou o apoio à depuração de programas direcionado aos usuários deste tipo de aplicação. Eles criaram uma versão modificada do '*Yahoo! Pipes*'[24] e realizaram um estudo pedindo que participantes realizassem tarefas na versão original do sistema e outro grupo utilizasse a versão modificada. O resultado das descobertas foi organizado em diretrizes para ambientes de *mashup* fornecerem assistência aos seus usuários. São elas: localização automatizada do erro, lista de *bugs* para serem corrigidos, evitar jargão técnico, referência cruzada entre *bugs* e mensagens de erro, mensagens de ajuda contextualizadas, nível incremental de assistência e auxílio à versionamento das alterações realizadas.

2.2.3. Documentação ativa

Os documentos ativos têm sido concebidos e definidos de diversas maneiras. Numa linha, temos documentos ativos como sistemas *Web* que servem para gerenciar documentos, automatizar processos e compartilhar conhecimento. Há empresas[25,26] que disponibilizam sistemas ou serviços para facilitar o desenvolvimento de aplicações que produzem documentos ativos. Inclusive a Microsoft[27] disponibiliza bibliotecas com essa finalidade. Estes sistemas permitem a criação de documentos com base em *templates*, não sendo é necessário se preocupar com formatação do arquivo, apenas preencher alguns dados e o documento completo é gerado automaticamente.

Em outra linha, a pesquisa de Aßmann (2005) propõe diferentes estilos de arquitetura para documentos ativos. O trabalho define documento ativo como um documento formado por: dados e programas; dados e macros; ou dados e *scripts*. O autor diz que um documento ativo pode ser manipulado interativamente, isto é, os campos refletem certas ações quando são preenchidos. Documentos ativos podem aparecer na *Web*, conter *servlets* ou *applets* em diferentes linguagens de *script*. Com frequência, o documento ativo reage imediatamente após mudanças

[24] https://pipes.yahoo.com/pipes/
[25] http://www.activedocs.com
[26] http://www.activedocuments.co.uk
[27] http://msdn.microsoft.com/en-us/library/bx9c54kf.aspx

do usuário. Além disso, os documentos ativos contêm componentes que são derivados automaticamente de um conjunto base e exploram o poder da programação para representar conteúdos de forma concisa.

Phelps (1997) trata a conversa entre os usuários do sistema e o sistema como um dos pontos fundamentais na documentação ativa. Neste trabalho, a autora relata uma experiência de uso de *wizards* para criar uma documentação sobre um adaptador de rede. Enquanto algumas formas de assistência, como sistemas de ajuda *online*, guiam o usuário pela interface do programa, os *wizards* fazem as tarefas pelos usuários, facilitando a comunicação direta entre eles e o programa. Em casos de *wizards*, a conversa é curta e vai direto ao ponto. O *wizard* eficiente, assim como o comunicador eficiente, sabe como manter uma conversa sobre o mínimo necessário para o entendimento das questões.

Outro trabalho, na área de Inteligência Artificial, apresenta o *design* de documentos ativos a partir da interpretação e aprendizado sobre as atividades comuns realizadas pelos usuários na aplicação (Garcia, 1992). A autora descreve uma abordagem para o *design* de documentação que aumenta a qualidade da documentação sem sobrecarregar os *designers* responsáveis pelo processo. A ideia central é criar um modelo de *design* que gere e explique decisões de *design* rotineiras. Ao invés de gravar as decisões deles, os *designers* ajustam o modelo de *design* inicial. O resultado desse processo é o documento ativo que contém o modelo de *design* habilitado a gerar automaticamente explicações para o *design*.

Os documentos ativos também têm sido explorados para organizar informações a partir da *Web.* Dado que a *Web* é uma enorme fonte de informação e é amplamente formada por documentos que contêm formulários, o conhecimento para processar os formulários pode ser utilizado para automatizar processos de negócios. Nam *et al.* (2003) propuseram uma abordagem de utilização de documentos ativos como método para automatizar processos de negócios baseados em documentos com formulários. O documento ativo deveria incluir conhecimento declarativo e as regras de negócios implicadas nos documentos para apoiar a automatização do processamento desses documentos. Além disso, o trabalho também sugere um *framework* para os documentos ativos. Os resultados mostraram que foi possível aumentar a inteligência das aplicações *Web*.

Ainda nessa linha, Zhuge (2003) apresenta um *framework* para organizar documentos na *Web* de modo que os serviços de recuperação e navegação por informações sejam melhorados. O trabalho defende que os documentos ativos podem apoiar os serviços de informação inteligentes como ensino *online*, assistentes de resolução de problemas e sistemas complexos de perguntas e respostas. Um documento é como um meio de comunicação entre as pessoas; durante o processo de transformação de conteúdo do escritor para o leitor, o conteúdo pode ficar distorcido, devido às diferenças no estilo de escrita e do entendimento mútuo dos significados das palavras e sentenças. Isto é semelhante ao processo que acontece na *Web*, restando então estruturar os documentos de acordo com a ontologia, com o conhecimento de *background*, com o conhecimento estrutural e com o conhecimento semântico.

Ko *et al.* (2000) tratam de *templates* de coleções de documentos ativos baseados em semântica para sistemas de gerenciamento de informação na *Web*. Os autores descrevem técnicas para representar semanticamente as coleções e os serviços de gerenciamento de informação que operam sobre as coleções. Essas técnicas ajudam os usuários do sistema a configurar análises complexas e estruturações de coleções de informações, além de adaptar o trabalho realizado para outras análises ou para diferentes coleções, e obter atualizações automáticas para coleções com conteúdos que mudam com o passar do tempo. A semântica das representações ajuda a identificar e sequenciar as análises e visualizações de serviços para uma dada tarefa.

2.3. *Software* como forma de participação social

Na década de 90, pesquisadores que já pensavam na produção de *software* como forma de participação social. Kalil (1996) vê a Internet como um supercomputador com processadores distribuídos operando em paralelo, que conecta não só microprocessadores, mas pessoas, repositórios de informação, sensores e agentes inteligentes. Desde aquela época, o autor tratava da importância de realizar mais pesquisas sobre tecnologias nas áreas de computação distribuída, trabalho colaborativo apoiado por computador (CSCW) e compartilhamento de conhecimento.

Preece e Shneiderman (2009) apresentaram um *framework* que categoriza níveis sucessivos de participação social onde os usuários podem ser leitores, contribuidores, colaboradores e líderes. Os autores descrevem como as pessoas entram nas mídias sociais, primeiramente, lendo, depois, contribuindo enviando comentários, fazendo *upload* de fotos ou avaliando restaurantes. Os passos seguintes são de colaboração explícita com outros, como na Wikipédia ou no Youtube. Por último, alguns usuários se tornam líderes organizando discussões (como curadores), ajudando novos usuários e promovendo a participação.

Outro trabalho nesta linha também classifica níveis de participação social (Denning e Yaholkovsky, 2008). O primeiro nível é o compartilhamento de informações que representa a troca de mensagens e dados através de *blog*, *chat*, *e-mail* e outros. A coordenação é o segundo nível, que significa ajustar componentes e pessoas para atuar com harmonia e acontece, por exemplo, em compras *online*, sistemas operacionais e protocolos da Internet. A cooperação é o terceiro nível e significa participar com outras pessoas de algo onde se aplicam as mesmas regras a todos, como é o caso de jogos *multiplayers*, *wikis* e fóruns. O último nível é o de colaboração, onde devem ser criadas soluções ou estratégias através da interação entre grupos de pessoas. Segundo os autores, há cinco etapas importantes na colaboração: explicitar uma questão a ser discutida pelo grupo; conectar pessoas interessadas; ouvir e aprender todas as perspectivas; permitir o envolvimento de todos; e a criação conjunta. Eles concluem que as ferramentas de compartilhamento de informação são mais usadas na colaboração atualmente. A seu ver, apesar do atraso nas tecnologias para fins colaborativos, a colaboração se dá e se sustenta mais por questões de solidariedade do que tecnologia.

O crescimento da computação social, baseada na produção social e colaboração em massa, facilitou a mudança da cultura do consumo para a cultura da participação (Jenkins, 2009). Na cultura do consumo os produtos eram todos finalizados e consumidos passivamente. Já na cultura da participação todas as pessoas são capazes de participar, contribuindo ativamente com problemas que têm, para elas, um significado pessoal.

De acordo com Fischer (2011), a cultura da participação oferece oportunidades para endereçar problemas da sociedade atual. Por exemplo, um problema que não seria possível resolver com grupos limitados de pessoas, como

é o caso da construção de prédios em três dimensões no Google SketchUp[28] e 3D Warehouse[29], repositórios de modelos de construções urbanas (ou não) criados por voluntários e organizados em coleções por curadores responsáveis pela inclusão de tais modelos no Google Earth[30]. O autor organiza os componentes que fazem parte do *framework* teórico da cultura da participação. São eles: *meta-design*, criatividade social e ecologias de participação. O objetivo do *meta-design* é a definição e a criação de infraestruturas sociais e técnicas onde novas formas de *design* colaborativo ocorrem. Na criatividade social todas as vozes envolvidas são consideradas na resolução de problemas complexos, no apoio a interação com outras pessoas, no compartilhamento de artefatos e na exploração de novos meios para colaborações interdisciplinares. As ecologias de participação consideram diferentes níveis de participação de acordo com o nível de interesse, motivação e conhecimento das pessoas em determinados contextos.

Fischer (2013) discute o impacto do *meta-design* no projeto de tecnologias educacionais. O *meta-design* permite às partes interessadas contribuírem e compartilharem conteúdo, dando seu *feedback*, para a evolução de objetos, processos, tecnologias, e assim por diante. Isso pode ser aplicável mesmo em sistemas de ensino onde muitas questões estão em aberto e ainda é preciso descobrir respostas. Todos os aprendizes atuam como *co-designers* em problemas pessoalmente significativos, fomentando assim a cultura de participação em que todas as vozes são ouvidas. O *meta-design* possibilita a criação de sistemas abertos que podem ser modificados pelos seus usuários, pois evoluem em tempo de interação. Estes sistemas devem permitir modificações significativas quando as necessidades aparecem; apesar dos melhores esforços em tempo de *design*, os sistemas sempre precisam evoluir para atender a novas necessidades que resultam da evolução dos próprios usuários.

A cultura de participação e o *meta-design* são claras oportunidade de autoexpressão através de programas. Turkle, em seu livro *Life on The Screen*, disse (Turkle, 2011): "*O computador não é o único modo de autoexpressão. Em cada ponto das nossas vidas, buscamos nos projetar no mundo. A criança vai usar lápis de cor para desenhar e argila para modelar. Nós pintamos,*

[28] http://www.sketchup.com/
[29] https://3dwarehouse.sketchup.com
[30] https://www.google.com/earth/

trabalhamos, mantemos diários, abrimos empresas, construímos coisas que expressam a diversidade de nossas sensibilidades intelectuais e pessoais. Mas, o computador nos oferece novas oportunidades como um meio de incorporar nossas ideias e expressar nossa diversidade". A autora também comenta sobre a necessidade de saber programação para se expressar através de computadores alguns anos atrás. No entanto, as ferramentas de criação de programas nos dias atuais facilitam essa expressão (são citadas Alice e The Looking Glass).

Os ambientes de programação de jogos, simulações e animações também podem ser vistos como formas de participação social. Como vimos na seção 2.2.1, vários deles têm *websites* onde há materiais adicionais e mecanismos de colaboração. Assim como o Scratch e o The Looking Glass, o aplicativo Catroid (Slany, 2012) também incentiva a *remixagem* de conteúdos dos jogos da comunidade. O Catroid foi inspirado no Scratch, mas foi construído apenas para dispositivos móveis. Os criadores desenvolveram uma comunidade *online* onde incentivam a colaboração e o *remix*. Todos os projetos da comunidade são de código aberto e as crianças são motivadas a realizar melhorias em projetos de outras pessoas. O grande desafio é fazer as crianças entenderem os princípios da *remixagem* e do código livre, pois muitas delas reagem negativamente e reclamam sobre plágio (Hill *et al.*, 2010). Para endereçar esse problema, os autores convidam o usuário a inserir um campo de créditos ao fazer o *upload* dos jogos na comunidade. Além disso, há uma *wiki* e um fórum onde questões sobre plágio, direitos autorais e código livre são abertamente discutidas. Os autores acreditam que o Catroid permite às crianças criarem conteúdo interativo nos seus dispositivos móveis ao invés de serem apenas consumidores. Isso dará a elas poder para dominar as habilidades necessárias para a autoexpressão através da programação em dispositivos que estão com elas o tempo todo (Gritschacher e Slany, 2012).

2.4. Relações dos trabalhos citados com a pesquisa

Com relação à definição de raciocínio computacional, a pesquisa desta tese reconhece a questão da resolução de problemas, automação de processos, construção de algoritmos proposta por pesquisadores, sociedades e empresas (Wing, 2006; ISTE, 2013, Google, 2014; Barr e Stephenson, 2011). No entanto,

nosso foco está em uma das perspectivas comentadas pelo NRC (2010), a expressão de ideias através de programas.

Nosso objetivo principal não é a apenas resolução de problemas através de mecanismos computacionais, mas o ensino de programação como forma de autoexpressão. Para isto, disponibilizamos um modelo para o *design* de tecnologias para fomentar a reflexão sobre os problemas, deixando a linha pedagógica para o ensino e aprendizado por conta dos professores.

As pessoas têm que aprender a pensar de certa forma para desenvolver a habilidade de usar linguagens de programação. Como há vários tipos delas, a separação entre aprender raciocínio computacional e aprender a programar usando uma ou outra linguagem. Na abordagem semiótica que esta tese adota (e que será detalhada no próximo capítulo), tanto é importante o raciocínio computacional, quanto a capacidade de programar usando uma ou outra linguagem. O raciocínio computacional determinará a competência da pessoa para elaborar o que ela tem intenção de comunicar através de um *software.* Enquanto isto, a capacidade de programar determinará a competência da pessoa para enunciar o que ela tem a dizer, de forma perceptível e compreensível para os destinatários desta comunicação.

Na nossa visão, anunciada na introdução desta tese, os produtores de *software* se comunicam com os consumidores (ou usuários); logo o *software* é tanto um meio como uma forma de expressão. Isto acontece em vários níveis. Por exemplo, o programador utiliza um ambiente de programação como o Eclipse para construir o sistema Joomla. Em determinado nível, o criador do Eclipse comunicou aos seus usuários que criou uma ferramenta de construção de programas. Já em outro, um usuário-desenvolvedor pode usar o Joomla para criar seu próprio ambiente na *Web*, comunicando aos usuários de seu software um outro conjunto de ideias. A sua mensagem se comunica através do conteúdo disponível no ambiente *Web*.

Quanto à abordagem, nos diferenciamos das ideias do *Computer Unplugged* de Bell *et al.* (2009) porque nosso intuito é, expressamente, o de utilizar as tecnologias computacionais, ao contrário dos autores que defendem o ensino de raciocínio computacional sem uso do computador. Porém, nossa proposta se aproxima da pesquisa de Lee *et al.* (2011) porque ambos incentivam o uso simulações e *design* de jogos. Lee *et al.* (2011) também sugerem o uso da

robótica, assim como Bers (2010) e Imberman *et al.* (2014). Entretanto, como a nossa pesquisa nasceu de uma ramificação do projeto SGD, utilizamos ambiente de programação visual AS, diferente de Imberman *et al.* (2014) que utilizam o NXT LEGO®.

Como reportado na subseção 2.2.1, há vários ambientes de programação de jogos e simulações disponíveis. O Greenfoot se destaca por ser mais parecido com um ambiente de programação profissional e ser direcionado para construção de código textual. Por outro lado, Scratch, Kodu, Alice, Storytelling Alice, The Looking Glass e AS são ambientes de programação visual.

A abordagem de programação como forma de autoexpressão foi explorada por Woltz *et al.* (2010) quando usaram o Scratch em um curso de jornalismo. Fristoe *et al.* (2011) mostraram a necessidade de crianças em busca de comandos que facilitassem a autoexpressão com o Kodu. Alice e seus sucessores também têm como característica importante a ideia de construir mundos onde histórias são contadas através dos personagens. O *storytelling* é uma forma de autoexpressão através de programas. No AS, essas características também podem ser exploradas. Ele é diferente do Scratch porque é baseado em regras de produção, deixando a lógica dos programas mais explícita. Quantos aos outros programas, o AS tem menos recursos gráficos, pois sua versão é em duas dimensões. Há, porém, um sucessor em 3D,- o AgentCubes, que tem basicamente as mesmas características do AS e com o qual o SGD-Br ainda não trabalha, mas poderá vir a trabalhar.

Quanto às comunidades *online*, grande parte das ferramentas comentadas têm extensões na *Web*. Elas se dedicam a oferecer material didático, tutoriais, vídeos, e apoio aos aprendizes. Algumas das comunidades oferecem mecanismos de *upload*, execução e *download* de projetos. O Scratch e o The Looking Glass têm o diferencial de motivar o *remix* de projetos. No Scratch é possível baixar os projetos, alterá-los e reenviá-los em uma nova versão, e enquanto que o The Looking Glass motiva o reuso de trechos com comportamentos de objetos nos mundos do ambiente. A proposta desta tese é auxiliar no *design* de extensões para ambientes de programação. As extensões devem conter múltiplas representações dos programas desenvolvidos e favorecer a reflexão dos aprendizes visando a autoexpressão.

Como parte da pesquisa no projeto SGD, Koh *et al.* (2010) apresentaram o grafo com o reconhecimento de ‘padrões de raciocínio computacional’ do AS

como uma forma de avaliar o aprendizado dos estudantes. De certo modo, este trabalho tem uma conexão especial com a proposta desta tese porque o grafo é uma nova representação sobre o programa e pode, portanto, ressaltar a presença de vários significados do *software*. Em nossa pesquisa, as múltiplas representações dos programas (ilustradas no capítulo introdutório) são propostas com o objetivo de explorar significados. Acreditamos que apenas uma análise sintática dos comandos não permite criar afirmações sobre o desenvolvimento das habilidades que envolvem o raciocínio computacional. Neste caso, o grafo de padrões não seria focado na avaliação do aprendizado, mas na reflexão sobre os programas.

Com relação às ferramentas de auxílio à compreensão e depuração de programas (subseção 2.2.2), diferente da linha seguida por Myers *et al.* (2004), nossa proposta não está embasada nos conceitos de *design* centrado no usuário. Como uma tese embasada na Engenharia Semiótica, nós vemos o programa como uma mensagem elaborada pelo *designer* enviada para o usuário final através do sistema. Nosso foco é no processo de comunicação, mediado pelo computador, entre o *designer* e o usuário. Já na abordagem de design centrado no usuário, o *designer* da tecnologia é invisível e ausente, por assim dizer. Toda a concepção de ação do usuário é formulada como uma interação que envolve estritamente a tecnologia e o usuário. Além do mais, não se enfatiza a comunicação ou as mensagens trocadas, mas sim o que o usuário faz e o quanto isto lhe custa cognitivamente (Norman, 1986).

Ko e Myers (2004) reportam em sua pesquisa um mecanismo de perguntas e respostas sobre dúvidas dos aprendizes durante a construção dos programas. Esta proposta se aproxima da nossa abordagem quando consideramos o *design* de uma documentação ativa como a elaboração de uma 'conversa' entre o programador e as representações do seu programa. No entanto, apesar da nossa 'conversa' conter todos e somente os caminhos propostos pelo *designer*, ela não é tão explícita como um mecanismo de perguntas (afirmativas e negativas) e respostas.

Burnett *et al.* (2004) e Kuttal *et al.* (2013) tratam da necessidade de desenvolver boas ferramentas para depuração de programas direcionadas aos usuários finais. Já nós reconhecemos o problema da falta de auxílio aos aprendizes, mas não propomos regras específicas para esse fim, mantemos o foco no *design* de representações que fomentem a reflexão sobre o código e sua

capacidade de comunicar ideias e intenções. Estamos propondo construir ferramentas que ajudem a pensar e não, necessariamente, aponte a linha em que um erro pode estar localizado.

Com relação à documentação ativa abordada na subseção 2.2.3, nossa definição está, em parte, alinhada com a de Aßmann (2005), quando ele fala sobre um conjunto de componentes derivados de uma base de componentes e sobre o poder dos documentos ativos para representar conteúdos de forma concisa. Mas, ele está propondo estilos de arquitetura para documentos ativos, enquanto que, nesta tese, seguimos uma linha completamente diferente; nós estamos apresentando um modelo de *design* da metacomunicação de documentação ativa no contexto de ensino de programação.

Assim como Phelps (1997), nós tratamos o documento ativo como uma 'conversa'. A proposta desta tese se concretiza em um modelo de *design* para documentação ativa que fomente a reflexão sobre programas; o objetivo é ajudar os *designers* a criar representações que auxiliem os aprendizes na resolução de problemas, apoiando a reflexão em/sobre a ação (Schön, 1983). Em outras palavras, queremos levar os aprendizes a refletir sobre programas, por meio de interações e 'conversas', tais programas, supondo que o *designer,* ao produzir o documento ativo, está promovendo e facilitando a comunicação de conceitos importantes sobre programação. O programa é tido como uma peça de autoexpressão, uma forma de comunicação. Como o documento ativo é a conversa sobre o programa construído pelo aprendiz, ele é (por ser também um programa, feito desta vez por um profissional e não um aprendiz) uma peça de metacomunicação. Trata-se de uma comunicação do designer (do documento ativo) sobre a comunicação do aprendiz-programador (autor de um programa de iniciante). Neste sentido específico o trabalho se distingue das propostas de documentação ativa de Garcia (1992), Zhuge (2003), Nam *et al.* (2003) e Ko *et al.* (2000).

A pesquisa não se propõe a contribuir com mais um ambiente de programação. Cada ambiente tem suas características específicas, pontos fortes e pontos fracos, cada um deles é mais ou menos adequado para determinados fins. Nossa proposta visa apoiar o *design* de mecanismos de reflexão sobre os programas desenvolvidos com as ferramentas existentes.

Em longo prazo, o objetivo da proposta é promover a autoexpressão através de *software* para atender a necessidade observada diante das pesquisas que mostram a importância da colaboração para a vida em sociedade. Desse modo, uma participação social consciente pode fazer com que os usuários deixem de ser apenas leitores para atuar de modo mais ativo como descrito por alguns pesquisadores (Preece e Shneiderman, 2009; Denning e Yaholkovsky, 2008).

A cultura da participação discutida por Fischer (2011) está cada vez mais presente e as pessoas que não estejam preparadas podem ser marginalizadas. Especialmente com relação ao *meta-design* (Fischer, 2013), para que participação social seja bem sucedida, é preciso que os usuários estejam minimamente alfabetizados computacionalmente. Eles precisam estar preparados para identificar corretamente o problema que estão enfrentando diante de um sistema, para, então, poderem opinar sobre possibilidades de melhoria. A produção de *software* como participação social está alinhada com a nossa visão de programação como forma de autoexpressão.

No contexto dos ambientes de programação, a participação social também acontece através do compartilhamento de recursos, reuso de código e *remix* de projetos, como o exemplo de incentivo à *remixagem* de jogos reportado por Gritschacher e Slany (2012) na seção 2.3. Promover a colaboração neste sentido é útil para o entendimento da programação a partir da criação de jogos por crianças, mas é um conhecimento que, se adquirido, pode ser transferido para contextos mais abrangentes, desde carreiras no desenvolvimento de *software* até uma atuação diferenciada na sociedade.