Self-Adaptation in Software Agents: An Initial Case Study in Game-Playing Agents

Ashok K. Goel, Joshua Jones, Christopher Parnin, Spencer Rugaber, Avik Sinharoy

Design Intelligence Laboratory
School of Interactive Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA

Abstract. As the task environment of a software artifact evolves, so must its design. For example, as the task environment in a computer game evolves, so must the design of the software agent that plays the game (or the agent's behavior is likely to become more suboptimal than before). We are exploring how a software artifact may adapt itself as it's task environment evolves incrementally. In particular, we are investigating how a game-playing agent may adapt itself as the percepts, actions, rules and constraints of its environment evolve from one version of the game to the next. A core research question in our work is what must an agent know about its design so that it can identify and make the right self-modifications to meet the needs of the new task environment? Our hypothesis is that the agent's self-knowledge of its teleology (i.e., the mechanisms by which it's design achieves its functions) may support the process of self-adaptation. In this paper, we describe the preliminary design of an interactive environment called GAIA in which a human game engineer and a game-playing software agent cooperatively adapt the agent's software design and program code. As the game-playing agent uses it's self-knowledge of it's teleology to identify modifications to it's design and code, the game engineer may (or may not) accept specific modifications and thus guide the process of self-adaptation. We also illustrate a first, simple example from FreeCiv, an interactive turn-based strategy game, at a high-level of specification.

In *Proceedings of the Third International Conference on Design Science Research in Information Systems and Technology.* (V. Vaishnavi & R. Baskerville, Eds). May 7-9, 2008, Atlanta, Georgia: Georgia State University. ©The Authors 2008. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee.

1 Background, Motivation and Goals

The design of a long-living software artifact evolves through many versions. Changes in the requirements from one version to the next typically are incremental and sometimes quite small (*deltas*). A software engineer (or a team of software engineers) formulates the requirements of a new version, adapts the design of the previous versions to meet the new requirements, implements and evaluates the modified design. Of course, the ordering of these tasks is not necessarily linear; the requirements, for example, may evolve during the design episode, and if the proposed design fails in the evaluation task, it may need to be redesigned. Thus, adaptive[1] design includes both proactive adaptation (adapting a design to meet new requirements) and retrospective adaptation (redesigning a proposed design).

In previous work, we have addressed both retrospective [1] [2] [3] and proactive adaptations [4] [5] [6] [7]. The earlier work had led to a knowledge representation language (called TMKL) for capturing a software agent's knowledge of its teleology, and a knowledge-based engine (called REM) for adapting the agent's design.[2] The ongoing work described here began with a case study of changes made to an open-source game [8]. The current project focuses on proactive adaptations to the design of game-playing software agents as their game environment evolves incrementally, e.g., incremental changes to the percepts and actions in the game or to the rules and constraints of the game. The earlier work on REM and TMKL also leads to our research hypothesis for the current project: *teleology, the explicit connection of functions to goals, is a basic organizational principle of adaptive software design.*

2 Game-Playing Domain

The domain for initial experimentation is computer-based strategy games, and the first case study examines FreeCiv[3]. FreeCiv is an open source variant of a class of Civilization games with similar properties. The aim in these games is to build an empire in a competitive environment. The major tasks in this endeavor are exploration of the randomly initialized game environment, resource allocation and development, and warfare that may at times be either offensive or defensive in nature. Winning the game is achieved most directly by destroying the civilizations of all opponents, but can also be achieved through more peaceful

[1] The term *adaptation* has several meanings. In the Software Engineering community, it is largely synonymous with *porting* to a new hardware platform or software environment. In the Artificial Intelligence community, however, it normally means changes made to alter functionality. We will try to make clear by context which of the two interpretations we intend.

[2] *Agent* is an AI term referring to software viewed as receiving percepts from an environment and taking actions in that environment; thus, the term connotes a perspective on a software system.

[3] http://www.freeciv.org

means by building a civilization with superior non-military characteristics, such as scientific development. We have chosen FreeCiv as a target for research because the game has an open Subversion[4] repository recording a lengthy revision history (development began on November 14, 1995), providing a rich source of data for us to analyze in order to understand the evolution of a specific software project. In addition, the AI agents included with the game are good vehicles for testing adaptation.

2.1 Adaptation Tasks

Table 1, adapted from [9] [10], provides a preliminary taxonomy of proactive adaptation tasks in adaptive software design with examples from the domain of interactive strategy games. In general, the adaptation task becomes harder as one moves down the table. In fact, almost any good game-playing software agent is likely to cover the first five of these tasks as part of its design, and thus these tasks are not of much interest in our project. Further, the eleventh and last adaptation task is outside the scope of this project because playing a real-time, first-person shooter game such as Unreal Tournament is outside the competence of any software agent designed for playing a turn-based game such as FreeCiv. Thus, the proposed project will focus on adaptation tasks six (6) through ten (10) in Table 1. Our research will not only enhance this preliminary taxonomy, for example by adding subtypes to the types enumerated below, but will also identify generic adaptation patterns and abstract design patterns corresponding to the types (and subtypes) of adaptation tasks.

Table 1. A Preliminary Taxonomy of Adaptation Tasks for Game-Playing

Key: So is start state, Sg is goal state; F is friendly units; E is enemy units; M is Map; TBG is a turn-based game such as FreeCiv and C-evo; RTG is a real-time game such as Unreal Tournament.

Type	***Example***
1. Memorization	*Same So and Sg*
2. Parameterization	*Change initial locations for F or E units*
3. Extrapolating	*Change composition of F or E units*
4. Restyling	*Vary non-combatants on M*
5. Extending	*Vary number of units for F and/or E*
6. Restructuring	*Design is for one kind of map, adapt for another*
7. Composing	*Design is for only mounted or only dismounted soldiers, adapt for both*
8. Abstracting	*Design is for one set of weapons and armor, adapt for another set*
9. Generalizing	*Design is for using deception only for unit locations, adapt for when deception is also used for weapon identities*
10. Reformulating	*Design is for one TBG, adapt for another TBG*
11. Differing	*Design is for a TBG, adapt for a RTG*

[4] http://subversion.tigris.org/

As an example, consider the following adaptation scenario: The Ancients modification package for FreeCiv [5] is a version of the game in which only pre-gunpowder weapons are used. Now suppose that the program code of a software agent that can play regular FreeCiv is available. How may the game-playing agent be adapted to play the Ancients version of FreeCiv? Note that this is an example of the eighth adaptation task (Abstracting) in Table 1.

3 Teleological Modeling

The starting point for the teleological methods that we will use to (partially) automate and assist with software adaptation is a system called Reflective Evolutionary Mind (REM) [4] [5] [6] [11] [12] [13], that makes use of a teleological knowledge representation called Task-Method-Knowledge Language (TMKL) [14] [15] [16]. Though we intend to develop a new system, incorporating insights from our study of change in software systems, we will draw heavily on experience with REM, and leverage existing technology wherever possible. This section describes TMKL, the language used to create models of software systems over which REM operates. Hoang, Lee-Urban and Munoz-Avila [17] have shown that TMKL has a simpler syntax with the expressive power of Hierarchical Task Networks in classical planning [18] [19] [20].

3.1 Task-Method-Knowledge Language

TMKL models of software systems are expressed in terms of tasks, methods, and knowledge. A *task* describes user intent in terms of a computational goal producing a specific result. Tasks encode functional information – the production of the intended result is the function of a computation. It is for this reason that the models specified in TMKL are *teleological* – the purpose of computational units is explicitly represented. A *method* is a unit of computation that produces a result in a specified manner. The *knowledge* portion of the model describes the different concepts and relations that tasks and methods in the model can use and affect as well as logical axioms and other inferencing information involving those concepts and relations. Formally, a TMKL model consists of a tuple (T, M, K) in which T is a set of tasks, M is a set of methods, and K is a knowledge base.

A task is a tuple $(in, ou, gi, ma, [im])$ encoding input, output, given condition, makes condition, and (optionally) an implementation, respectively. The input (in) is a list of parameters that must be bound in order to execute the task (e.g., a task involving movement typically has input parameters specifying the starting location and destination). The output (ou) is a list of parameters that are bound to values as a result of the task (e.g., a task that involves counting a group of objects in the environment will typically have an output parameter for the number of objects). The given condition (gi) is a logical expression that must hold in order to execute the task (e.g., that a robot controlled by the software is at

[5] http://www.cs.utexas.edu/users/bdbryant/freeciv/ancients.html

the starting location of a movement task to be executed). The makes condition (ma) is a logical expression that must hold after the task is complete for the execution to have been successful (e.g., that the robot is at the destination). The optional implementation (im) encodes a representation of how the task is to be accomplished. There are three different types of tasks depending on their implementations: non-primitive tasks, primitive tasks, and unimplemented tasks. *Non-primitive tasks* have a set of methods as their implementation. *Primitive tasks* have implementations that can be immediately executed such as program code, logical assertions or knowledge bindings. *Unimplemented tasks* cannot be executed until the model is refined.

A method in TMKL is a tuple (pr, ad, st) encoding a provided condition, additional results condition, and a state-transition machine. The two conditions encode incidental requirements and results of performing a task that are specific to that particular method of doing so. The state-transition machine in a method contains states and transitions. The execution of a method involves starting at the entry transition, going to the state it leads to, executing the subtask for that state, selecting an outgoing transition from that state whose applicability condition holds, and then repeating the process until a terminal transition is reached.

Knowledge representation (K) in TMKL is based on Loom [21], an off-the-shelf knowledge representation (KR) framework. Loom provides not only all of the KR capabilities found in typical AI planning system (the ability to assert logical atoms, to query whether a logical expression holds in the current state, etc.), but also a variety of more advanced features (logical axioms, truth maintenance, multiple inheritance, etc.). In addition, Loom provides a top-level ontology for reflective knowledge. Through the use of a formal framework such as Loom, dependencies between the knowledge used by tasks as well as dependencies between tasks themselves can be described in such a way that an agent will be able to reason about the structure of the tasks.

Figure 1 and Table 2 illustrate an example of a TMKL model that REM is able to use in reasoning [22]. The software modeled in the example is a portion of a FreeCiv playing agent. The portion depicted is responsible for assisting in decisions about the type of unit that should be manufactured at a city controlled by the AI player. Specifically, this portion of the agent determines the urgency of need for defensive units that will help to defend the city in case an attack is staged by an opposing player. We call this task 'Defend-City'.

As shown in Figure 1, the overall Defend-City task is decomposed into two sub-tasks via the Evaluate-then-Defend method. These subtasks are respectively responsible for evaluating the defensive needs of a city and initiating the building of a particular structure or unit at that city. One of the subtasks, the Evaluate-Defense-Need task, is further decomposed through the Evaluate-Defense method into two additional subtasks, a task to check internal factors in the city for defensive requirements and a task to check factors outside the immediate vicinity of the city for defensive requirements. These subtasks are then implemented for execution as described below. The Defend-City task is executed each turn that

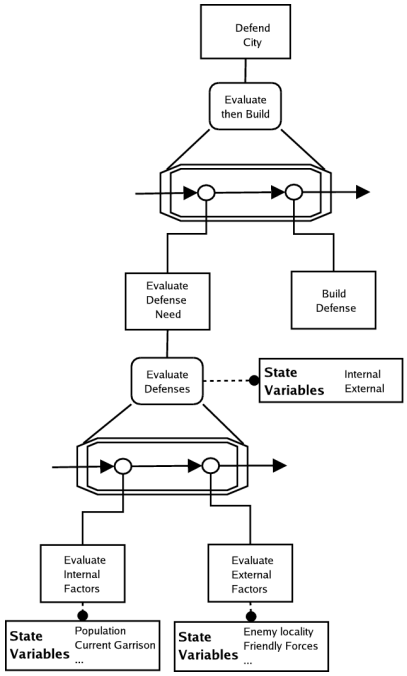

Fig. 1. Depiction of the *Defend-City* model

the agent is not building a defensive unit in a particular city in order to determine if production should be switched to a defensive unit. It is also executed each turn that a defensive unit has finished production in a particular city. The internal evaluation task utilizes knowledge concerning the current number of troops that are positioned in and around a particular city to determine if the city has an adequate number of defenders without regard to any external circumstances. The external evaluation of a city's defenses examines the area within a specified radius around a city for nearby enemy combat units. It utilizes the knowledge of the number of units, their distance from the city, and the number of units currently allocated to defend the city in order to provide an evaluation of the need for additional defense. These tasks produce knowledge states in the form of defense recommendations that are then used by the task that builds the appropriate item at the city. The Build-Defense task makes use of the knowledge states generated by the evaluation subtasks, knowledge concerning the current status of the build queue, and the technology currently available to the agent to determine what should be built for a given iteration of the task.

4 The Process of Adaptive Software Design

The goal of this section is neither to describe how software engineers reason about adaptive design problems, nor to prescribe how they should do so. Instead, the

Table 2. TMKL Model of Defend-City Task

TMKL Model of the Defend-City Task	
Task	Defend-City
by	Evaluate-Then-Build
makes	City-Defended
Method	Evaluate-Then-Build
transitions:	
state: *s1*	Evaluate-Defense-Need
success	s2
fail	fail
state: *s2*	Build-Defense
success	success
fail	fail
additional-result	City-Defended, Unit-Built, Wealth-Built
Task	Evaluate-Defense-Need
input	External/Internal-Defense-Advice
output	Build-Order
by	UseDefenseAdviceProcedure
makes	DefenseCalculated
Method	Evaluate-Defense-Need
transitions:	
state: *s1*	Evaluate-Internal
success	s2
fail	fail
state: *s2*	Evaluate-External
success	success
fail	fail
additional-result	Citizens-Happy, Enemies-Accounted, Allies-Accounted
Task	Evaluate-Internal
input	Defense-State-Info
output	Internal-Defense-Advice
by	InternalEvalProcedure
makes	Allies-Accounted, Citizens-Happy
Task	Evaluate-External
input	Defense-State-Info
output	External-Defense-Advice
by	ExternalEvalProcedure
makes	Enemies-Accounted
Task	Build-Defense
input	BuildOrder
by	BuildUnitWealthProcedure
makes	Unit-Built, Wealth-Built

goals are to suggest how an automated system may reason about design adaptations and to indicate how a software engineer might use teleological knowledge

in her reasoning. For this discussion, we assume that the program code of the software agent is modeled with TMKL.

Figure 2 illustrates a process for adaptive software design. The process begins with a specification of the functions delivered by a software agent (e.g., the Defend-City agent for regular FreeCiv), the program code for the agent, and the TMKL model of the agent's functional architecture and a specification of the differences between the functions delivered by and desired of the agent (e.g., a specification of the differences between the Defend-City task for the regular and the Ancients version of FreeCiv). The automated system uses the TMKL model for teleological analysis, localizes the needed modifications in the TMKL model (and the program code), and spawns adaptation goals appropriate to the needed modifications. The adaptation goals correspond to the taxonomy of adaptation tasks but are localized to specific components (or subsystems) or the agent.

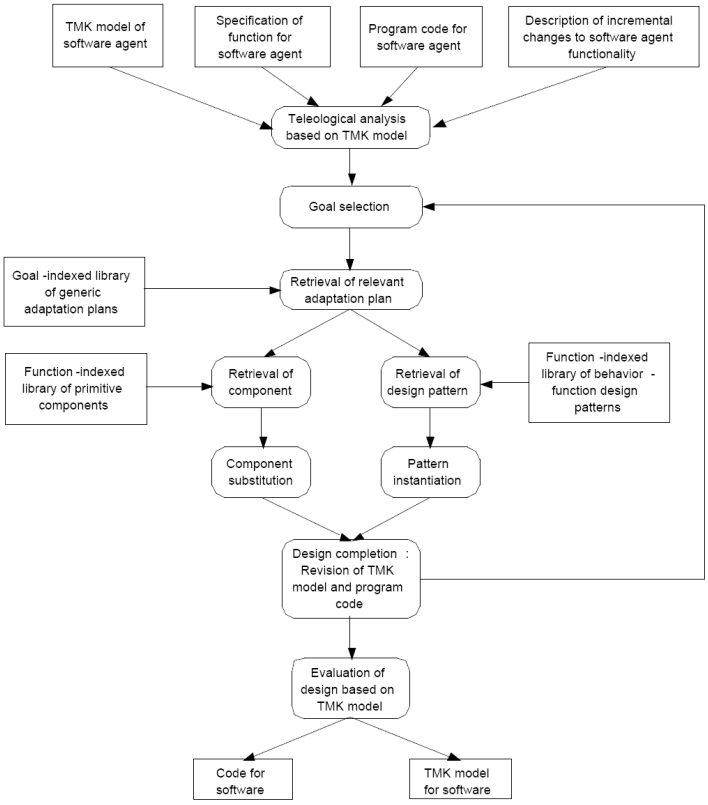

Fig. 2. A Partial Description of a Process for Adaptive Software Design

Each adaptation goal is used to retrieve generic candidate adaptation patterns applicable to it. These patterns are retrieved from a library of adaptation patterns stored as part of the automated system. The first adaptation, for example, may try to access a primitive component (i.e., a procedure) for the Ancients version of FreeCiv to replace the corresponding procedure in the Evaluate-External-Factors primitive task in the TMKL model of the Defend-City agent. Let us suppose that such a procedure is not available in the component library. The second adaptation pattern may then abstract the adaptation goal, and try to access an abstract design pattern for defending cities against enemy attacks. Let us suppose that such a design pattern in the form of an abstract procedure is available in the adaptation pattern library. The pattern may specify, for example, that the number of defenders in a city should exceed or at least equal the number of enemy combat units within a specific radius of the city, where the radius is directly proportional to the speed of travel of the enemy units. The adaptation pattern instantiates this procedure in the Evaluate-External-Factors task and the specific portion of the program code it points to, and, given that the speed of combat units in Ancients is lower than the speed of such units in regular FreeCiv, it adjusts the radius around the city accordingly. An important principle of this adaptation process is minimalism of change to the structure of the design to achieve an adaptation goal. In general, the adaptation strategy selected will depend upon the nature of the change to be made.

4.1 GAIA Design

We have developed a system called GAIA that realizes the adaptation process described above. At this time, the implementation of the system is partial, but we do have a working test case that exercises the various components within the system. In this test case, the user wishes to add a constraint to the agent; specifically, the user wishes the agent to build more than 300 fighting units before commencing the attack against the opposing player. We will first describe the major components in the system architecture, and then give a more detailed account of the system's processing for this specific scenario. The system architecture is depicted in figure 3.

REMng (REM the Next Generation) is the reasoning engine responsible for selecting and applying adaptations based on the TMKL model of the software system to be adapted and the specification of the adaptation requirements. The adaptation requirements take the form of either an anticipated environmental change, in the case of proactive adaptation, or information about failures that are to be corrected through adaptation in the case of retrospective adaptation. REMng interacts with the GUI (SAGi) in order to obtain information about adaptation requirements in the case of proactive adaptation, to provide feedback about proposed adaptations to the user, and to receive the user's responses to proposed adaptations. In the case of retrospective adaptation, REMng needs to interact with the Event Log Manager to retrieve traces of the system under adaptation executing in the environment. These traces provide information to REMng's reasoning process about the failures that are to be corrected. REMng

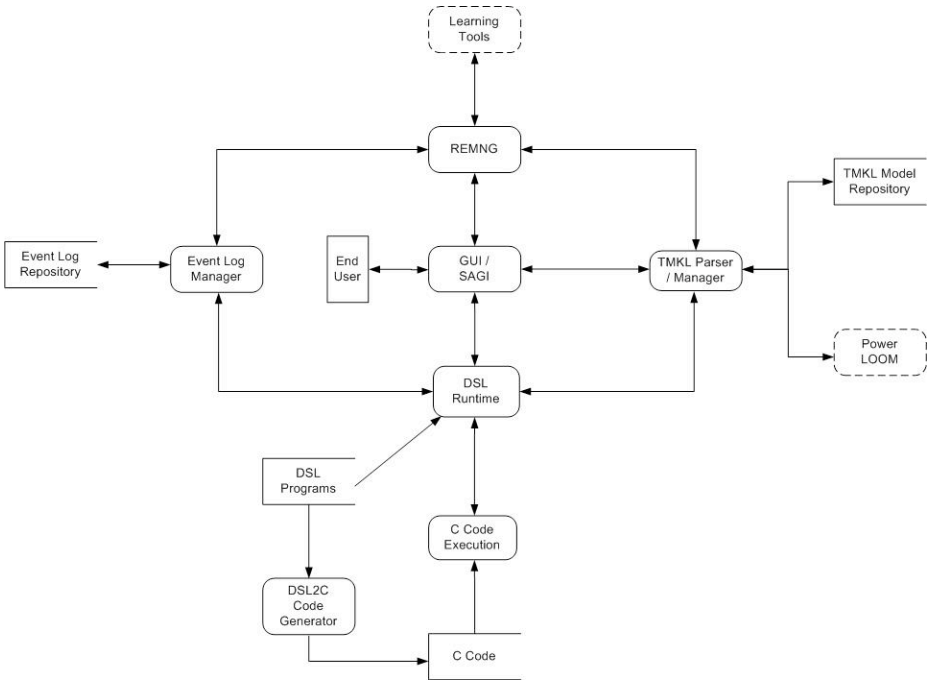

Fig. 3. GAIA Architecture Diagram

retrieves the TMKL models of agents to be adapted from the TMKL Manager and registers new models reflecting (candidate) adaptations with this module. Finally, REMng may interact with external tools such as reinforcement learning libraries or generative planners in order to execute some of its adaptation strategies. As alluded to above, a major component of REMng is a library of adaptation patterns. Each pattern within this library encodes applicability information that may depend upon the type of adaptation that is required as well as the context within the agent model where adaptation is required. Each pattern also contains a domain-inspecific adaptation procedure, parameterized with needed domain and context specific information, that can be executed in order to produce the appropriate change to the agent.

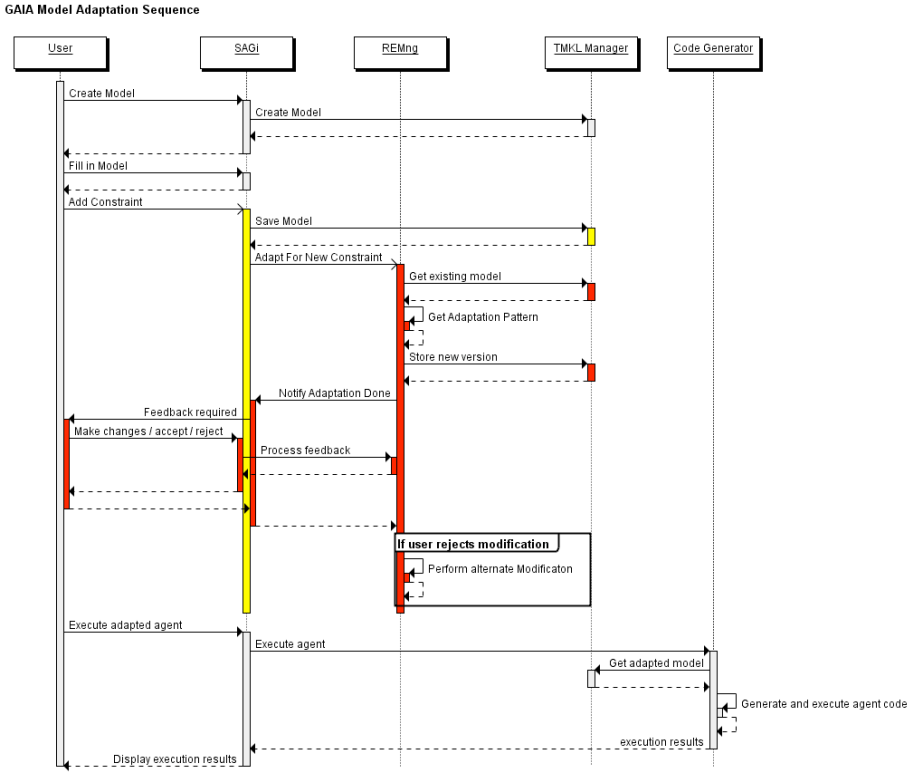

Fig. 4. Example GAIA Adaptation Sequence Diagram

The sequence of messages that passes among the various GAIA components for the particular proactive adaptation example described above is depicted in figure 4. This scenario serves as a proof of concept for our architecture, as well as our overall hypothesis that teleological models support software agent adap-

tation. In this sample adaptation, the user wishes a game playing agent to be adapted for a new setting that is characterized by the addition of a new constraint (more than 300 fighting units have been built) in the execution of certain actions (declare war). This new constraint is entered by the user via the SAGi GUI. This results in a call from SAGi to REMng requesting the adaptation and delivering the information provided by the user. REMng then retrieves the TMKL model of the agent from the TMKL Manager. Using the model and the information provided by the user, REMng selects a candidate adaptation pattern and applies it to the agent. In this case, the adaptation pattern selected is parameterized over the constraint being added and a task in the TMKL model with a makes condition that matches the constrained behavior. The adaptation pattern's process will modify the provided condition of the task. In addition, preconditions attached to transitions entering the task are adjusted to reflect the new constraint. The resulting model is then registered with the TMKL Manager. A notification is also sent to SAGi, causing the model reflecting the candidate adaptation to be displayed to the user. The user can then choose to execute the adapted agent, resulting in a call to the Code Generation component. The Code Generator will produce an executable version of the agent, translated from the TMKL model. The results of execution are then returned to SAGi for inspection by the user. After the user views the information, they will decide either to accept or reject the adaptation proposed by REMng. In this example, the user is satisfied with the adaptation and so an accept notification is sent to REMng. If the user had rejected the adaptation, REMng would attempt to apply another adaptation pattern if another candidate was found in the pattern library, continuing the process. The user also has the option to execute the adapted model (or in fact any agent model known to the system). If execution is desired, the user will select this operation through the SAGi GUI. This results in a message being sent to the Code Generation component, which will retrieve the appropriate agent model from the TMKL Manager, perform code generation, execute the resulting agent code and finally return the results of execution for display back to the user via the GUI. This process can be used to help with the evaluation of a proposed adaptation.

5 Current Work

Our current efforts focus on the following issues:

- Generality of TMKL - We are evaluating and enhancing the generality of TMKL for teleological modeling of software agents including game-playing agents. For example, in addition to the portion of the FreeCiv game-playing agent for the Defend-City Task, we are building models of the agent for the tasks of placing a city and attacking enemy units and cities. We are also using TMKL for representing story plots [23]. This suggests that TMKL could be useful for representing game-playing scripts.

– Generality of REMng - We are evaluating and enhancing the generality of REMng for teleological reasoning in self-adaptive software agents. For example, we are experimenting with teleological reasoning about agents that address constraint satisfaction problems.

– Efficiency of REMng - In experimenting with various software agents, we have found that REMng often is too slow for use in many game-playing situations. This is in part because REMng keeps track of not only each task and method that it executes but also each knowledge state it produces. Many games however require much faster responses than REMng currently delivers. In the example of Defend-City task described earlier, we ported the needed portions of REMng's capabilities to a specialized agent to achieve this kind of efficiency [22].

– Integration with Generative Planning - REM uses GRAPHPLAN [24] both as an alternative to self-adaptation using teleological reasoning and to complete partial solutions generated by self-adaptation. GRAPHPLAN is a fast, general-purpose generative planner. We plan to extend REMng's library of adaptation patterns to include some that make use of GRAPHPLAN. We are also experimenting with the use of FASTFORWARD [25], another fast, general-purpose generative planner.

– Integration with Reinforcement Learning - REM also uses Q-Learning [26], a form of reinforcement learning (RL) [27] [28], as another alternative to self-adaptation using teleological reasoning, and to complete partial solutions generated by self-adaptation. We have experimented with the use of REM's teleological reasoning to localize the use of Q-Learning. The variant of REM for the Defend-City task uses teleological reasoning for this purpose [22]. Again, we plan to extend REMng's library of adaptation patterns to include some that use RL to refine task implementations.

– Knowledge Specification - An important issue in building teleological models of game-playing agents using TMKL is the specification of the domain knowledge including the percepts and the actions in a game. We are developing a domain specific language for specifying the domain knowledge in FreeCiv.

– Knowledge Modification - Given the specification of the knowledge and its organization for a task, another issue is how to automatically repair the knowledge organization if it turns out to be incorrect as the game is actually played. We have developed an automated technique for self-diagnosis of an agent's knowledge organization and evaluated it for the task of city placement in FreeCiv [29] [30]. In this method, the agent uses its knowledge to make predictions about the world, and if these predictions turn out to be incorrect, it uses a functional specification of its knowledge to diagnosis and repair the knowledge.

Acknowledgments

We thank Derek Richardson and Jason Taylor for their contributions to the GAIA project. This research is supported by a NSF Science of Design Grant (#0613744) on Teleological Reasoning in Adaptive Software Design.

References

1. Stroulia, E., A. Goel, A.: A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. Proc. National Conference on Artificial Intelligence - (AAAI-96) (1996)
2. Stroulia, E., Goel, A.: Redesigning a problem solver's operators to improve solution quality. Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI-97) (1997) 562–567
3. Stroulia, E., Goel, A.: Evaluating problem-solving methods in evolutionary design: The autognsotic experiments. International Journal of Human-Computer Studies, Special Issue on Evaluation Methodologies **51** (1999) 825–847
4. Murdock, J.W., Goel, A.K.: An adaptive meeting scheduling agent. Proceedings of the First Asia-Pacific Conference on Intelligent Agent Technology (IAT'99) (1999) 374–378
5. Murdock, J.W., Goel, A.K.: Towards adaptive web agents. Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering (ASE'99) (1999) 335–338
6. Murdock, W., Goel, A.: Learning about constraints by reflection. Proc. 14th Biennial Conference of Canadian AI Society (2001) 131–140
7. Murdock, J.W., Goel, A.K.: Meta-case-based reasoning: self-improvement through self-understanding. J. Exp. Theor. Artif. Intell. (2008)
8. Jones, J., Goel, A., Rugaber, S.: Automating software evolution. DESRIST (2007)
9. Aha, D.W., Molineaux, M., Ponsen, M.J.V.: Learning to win: Case-based plan selection in a real-time strategy game. (ICCBR)
10. : Darpa proposer information pamphlet for transfer learning. BAA 05-29 (2005)
11. Murdock, J.W., Goel, A.: Localizing planning using functional process models. Proc. International Conference on Automated Planning and Scheduling (ICAPS-03) (2003)
12. Murdock, J.W., Goel, A.: Meta-case-based resaoning: Self-improvement through self-understanding. Journal of Experimental and Theoretical Artificial Intelligence (in press)
13. Murdock, J.W.: Self-Improvement Through Self-Understanding: Model-Based Reflection for Agent Adaptation. PhD thesis, Georgia Institute of Technology (2001)
14. Murdock, J.W.: Modeling computation: A comparative synthesis of tmk and zd. College of Computing Technical Report GIT-CC-98-13 (1998)
15. Murdock, J.W.: Semi-formal functional software modeling with tmk. College of Computing Technical Report GIT-CC-00-05 (2000)
16. Murdock., J.W.: Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation. PhD thesis, College of Computing, Georgia Institute of Technology (2001)
17. Hoang, H., Lee-Urban, S., Muoz-Avila, H.: Hierarchical plan representations for encoding strategic game ai. Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05) (2005)

18. Erol, K., Hendler, J., Nau, D.: Htn planning: Complexity and expressivity. Proc. Twelfth National Conference on Artificial Intelligence (AAAI-94) (1994)
19. Sacerdoti, E.: A structure for plans and behaviors. (1977)
20. Tate, A.: Generating project networks. Proc. International Joint Conference in Artificial Intelligence (1977) 888–893
21. MacGregor, R.: The Loom Knowledge Representation Language. University of Southern California. Marina del Rey, CA, USA. (1987)
22. Ulam, P., Goel, A., Jones, J., Murdock, J.W.: Using model-based reflection to guide reinforcement learning. Proceedings of the IJCAI 2005 Workshop on Reasoning, Representation and Learning in Computer Games (2005)
23. Adams, S., Goel, A.: Stab: Making sense of vast data. Proc. IEEE Conference on Intelligence and Security Informatics (2007)
24. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence **90**(1-2) (1997) 281–300
25. Hoffman, J.: A heuristic for domain-independent planning and its use in an enforced hill-climbing algorithm. Proc. 12th International Symposium on Methodologies for Intelligent Systems (2000) 216–227
26. Watkins, C.: Modeling Delayed Reinforcement Learning. PhD thesis, Psychology Department, Cambridge University, United Kingdom (1989)
27. Barto, A., Sutton, R., Brouwer, P.: Associative search network: A reinforcement learning associative memory. Biological Cybernetics **40**(3) (1981) 201–211
28. Sutton, R., Barto, A.: Reinforcement learning: An introduction. (1998)
29. Jones, J., Goel, A.: Knowledge organization and structural credit assignment. Proc. IJCAI-05 Workshop on Reasoning, Representation and Learning in Computer Games (2005)
30. Jones, J., Goel, A.: Structural credit assignment in hierarchical classification. Proc. International Conference on Artificial Intelligence (2007)