2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems

Teleological Software Adaptation

Joshua Jones, Chris Parnin, Avik Sinharoy, Spencer Rugaber & Ashok K. Goel
Design & Intelligence Laboratory, College of Computing
Georgia Institute of Technology, Atlanta, USA 30332

***Abstract*—We examine the use of teleological metareasoning for self-adaptation in game-playing software agents. The goal of our work is to develop an interactive environment in which the game designer generates requirements for a new version of a game, and the legacy software agents from previous versions of the game adapt themselves to the new game requirements. We are developing and testing our metareasoning technique for adapting game-playing agents in Freeciv, a mature program in the domain of turn-based, multi-player strategy games (www.freeciv.wikia.com). In this paper, we first present an analysis of adaptations to FreeCiv, next describe our general approach, and then describe a specific adaptation scenario.**

I. Introduction

Designs of long-living interactive games evolve through many versions. Changes from one version of a game to the next typically are incremental and often small Δs. A game designer (or a team of game designers and software engineers) formulates the requirements of the new version of the game, adapts the software for previous versions of the game to meet the new requirements, and implements and evaluates the modified designs of the game and the software. Typically the game designer uses high-level scripting languages to define the game environment (percepts, actions, rules, constraints) as well as the behaviors of various virtual agents (robot player) in the game. An interesting research issue in game playing is how might a virtual agent adapt its design, and thus its behaviors, to small Δ changes in its game environment. If the changes in the game environment can be arbitrarily large and complex then this becomes an "AI-complete problem," i.e., the problem becomes equivalent to the general AI problem. However, even if the changes to the game environment are incremental and small, this is a hard computational problem because changes to the environment can be of many types, modifications to the agent design can be of many types, there is no one-to-one mapping between changes to the environment and modifications to the agent design, and any modification to the agent design needs to be propagated down to the level of program code so that the new software is directly executable in the game environment.

In previous work, we have investigated the use of metareasoning for self-adaptation in software agents, which may offer an effective technique for adapting virtual agents in game playing. It is useful to make a few distinctions here. Firstly, adaptations to an agent can be retrospective (i.e., when the agent fails to achieve a goal in its given environment; [1] [2] [3] [4] [5], or proactive (i.e., when the agent is asked to operate in a new task environment; e.g., [6] [7]. Secondly, adaptations can be either to the deliberative element in the agent architecture [1] [2] [3] [4] [7], the reactive element [8], or both. Thirdly, adaptations to the deliberative element may be modifications to its reasoning process (i.e., to its task structure, selection of methods, or control of reasoning; e.g., [2] [3] [4] [7]), to its domain knowledge (i.e., the content, representation and organization of its knowledge; e.g., [9]), or both.

In this paper, we examine the use of teleological metareasoning for proactive adaptation of an agent's deliberative processing, specifically in the context of game-playing software agents. Our hypothesis is that a declarative self-model that captures the teleology of the agent's design may enable localization, if not also identification of the elements in the reasoning process responsible for a given behavior. (The term *metareasoning* refers to reasoning about reasoning. *Teleology* is a term from philosophy describing the study of design. The word's etymological roots connote goals and purpose. A *teleological model* of a system is one that situates the system's design and implementation in the context of its goals. *Teleological metareasoning* refers to reasoning about the teleology of reasoning.) The basic theme of our work on metareasoning for self-adaptation in intelligent agents is that *teleology is a central organizing principle of knowledge representations that enable self-adaptation of reasoning processes.* This work uses teleological self-models to allow the metareasoning process to perform self-diagnosis. Having done this, the metareasoning process can then perform self-adaptation at the identified location(s).

The goal of the work described here is to develop an interactive environment called GAIA (for Game Agent Interactive Adaptation) in which the game designer generates requirements for a new version of a game, and the legacy agents from previous versions of the game adapt themselves to the new game requirements in cooperation with the human designer, who provides guidance where automation is not possible or not implemented. We are developing and testing our metareasoning technique for adapting a mature program in the domain of turn-based, multi-player, strategy games, specifically FreeCiv (www.freeciv.wikia.com).

II. Analysis of Adaptations to Interactive Games

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

978-0-7695-3794-8/09 $26.00 © 2009 IEEE
DOI 10.1109/SASO.2009.33

destroying the civilizations of all opponents but can also be achieved through more peaceful means by building a civilization with superior non-military characteristics, such as scientific development. We have chosen FreeCiv as a domain for research because the game has an open Subversion 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. The open source development style also lends itself to this analysis, for obvious reasons.

Our initial analysis of the FreeCiv Subversion software repository is based on taxonomies drawn from existing literature in the software engineering community [10][11]. Mens et al. discuss categories of software change along two dimensions –*scope* and *intent*. The former expresses the degree of impact a change has in terms of the number of places in the software system that are touched by a modification. With respect to intent, the authors partition software changes into seven categories ranging from the purely cosmetic to those changes that enhance functionality:

- *Groomative* changes are intended to increase the legibility of code, e.g. improving comments or indentation.
- *Preventive* changes are intended to facilitate future maintenance, for instance factoring out common code into a function.
- *Performance* changes affect the resource usage (e.g. time or memory) of software without modifying functionality in terms of I/O behavior.
- *Adaptive* changes adjust software so that it can execute in a new environment; the task of applying these changes is commonly referred to as "porting" software[1].
- *Reductive* changes remove user-visible functionality from a software system.
- *Corrective* changes are bug fixes, correcting some problem with user-visible functionality.
- *Enhancive* changes improve or extend user-visible functionality.

The other pertinent dimension that we measured on a sample of changes in the FreeCiv Subversion repository is change scope. This dimension captures the impact of a change in terms of the number of places in the software system that are touched by a modification. File scope changes touch code only within a single file, module scope changes affect code only within one directory in the source hierarchy of the project, and cross-module changes involve modifications to code in more than one directory.

To get a sense of the distribution of changes to the FreeCiv codebase, a sample of changes over an approximately four month period from March 15, 1999 to July 28, 1999 was examined and broken down across the two dimensions described above, through detailed analysis by hand of the changes and associated Subversion comments. There were a total of 304 changes during this time period that we categorized [12]. Figure 1 shows a breakdown of changes by their intent and scope. As can be seen, groomative, enhancive, and corrective changes make up the bulk of those that occurred in the time period studied. As might be expected, there are progressively fewer changes as scope increases.

Based on this data, the areas with highest potential impact are groomative and enhancive changes, with secondary impact in the corrective and adaptive areas. However, it should also be noted here that this data does not reflect the difficulty of making a change. Since enhancive changes seem to be a potential impact area, based both on the data presented above and on our understanding of the application of teleological reasoning to software adaptation, we also examined the enhancive changes from the sample period more closely. Specifically, we were interested in determining the proportion of the enhancive changes to the GUI, with which it is less likely that automated reasoning can have significant impact, and core feature changes, where the potential for impact seems higher. We found that just over 50% of the enhancive changes are GUI specific. Further, the changes with broadest scope more commonly fall into the core functionality modification category. These results suggest that enhancive changes may indeed be a significant impact area for teleological reasoning about software adaptation.

III. Architecture for Adaptation

We have taken a teleological approach to software adaptation, addressing the problem of adapting software by connecting a teleological model to the source code. The teleological model is expressed in a language we have devised called TMKL [7]. Further, we have built a development environment called GAIA to automate the adaptation process. GAIA has two interesting features that we describe: its reasoning engine, REMng, and its code generation mechanism.

A. GAIA

GAIA is an interactive development environment, implemented in Eclipse (http://www.eclipse.org/) that supports a modeler in building TMKL models and then using them to adapt the modeled software. Its architecture is depicted in Figure 2.

The user-facing portion of GAIA is called SAGi, which appears in the center of the diagram. Besides providing overall control of the adaptation process, it supports users in building TMKL models. To the right of SAGi is a module responsible for managing TMKL models and persisting them. Persistence is currently provided via Eclipse's EMF package. The other two parts of GAIA are its inference engine, REMng (at the top of the diagram), and its code generation mechanism (under SAGi). The code-generation mechanism is based on a domain specific language (DSL) that provides a means of expressing objects and relations that exist within the game domain. We intend to support the addition of off-the-shelf machine learning tools, such as planners and learners, via

[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. In this paper we normally intend the latter, but in this case, the former.

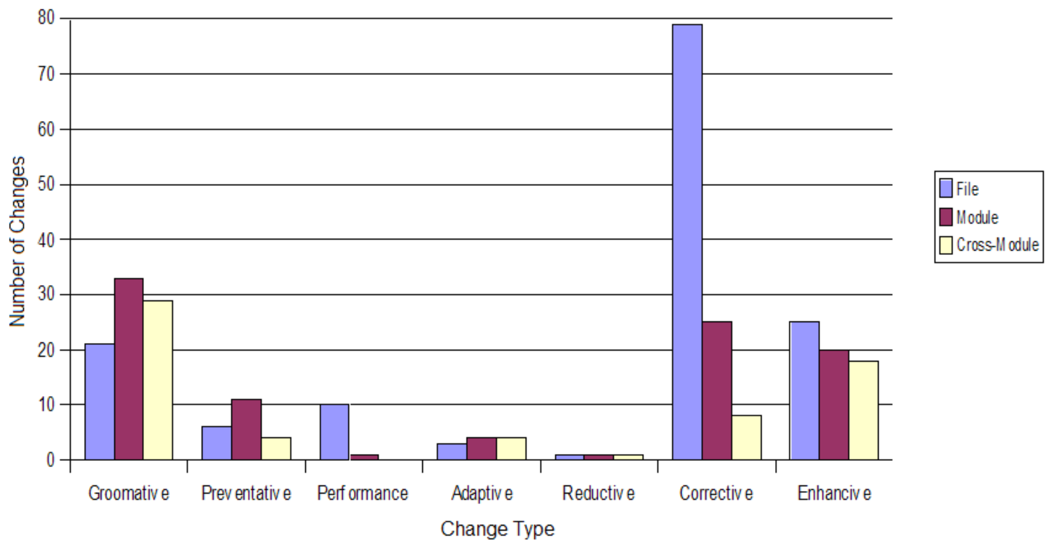

Fig. 1. Number of changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by both intent and scope.

an interface to REMng. To the left of SAGi in the figure is an event log for communicating run-time information back to REMng.

B. TMKL

TMKL [7] is a teleological modeling language intended to support automated reasoning about software systems. The name is an abbreviation for Task-Method-Knowledge Language. *Tasks* in TMKL express a system's goals in terms of inputs and outputs. *Methods* describe the mechanisms by which a system accomplishes its goals. *Knowledge* comprises the application-domain concepts and relations on which the system operates.

Tasks and Methods are defined in alternating hierarchical layers. The topmost Tasks describe a system's ultimate goals as black-box functional specifications. Each Task is implemented by any one of a set of alternative Methods, which it superordinates. Methods, in turn, organize the operation of a set of sub-Tasks. At the bottom of the hierarchy are primitive Tasks, available as executable units of code.

TMKL Tasks are defined by five pieces of information: input Knowledge elements, output Knowledge elements, required input conditions (pre-conditions), produced output conditions (post-conditions), and implementing Methods. Methods are defined similarly in terms of their pre-conditions, sub-Tasks and ordering constraints, represented by finite state machines. Finally, Knowledge in TMKL is defined in terms of application-domain concepts and their relationships. An example of a TMKL model is depicted in Figure 4 and discussed below in the section describing our case study.

C. REMng

GAIA's inferencing mechanism is called REMng. REMng works in conjunction with the code generation mechanism

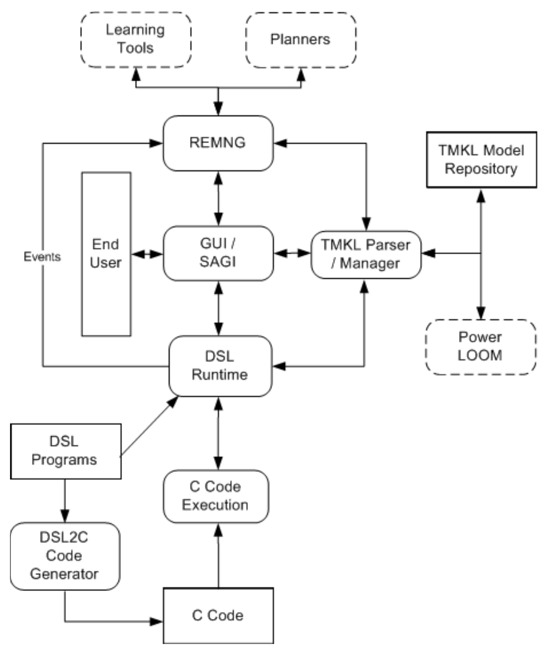

Fig. 2. GAIA Conceptual Architecture

described in the next subsection. That is, REMng, when given a program model expressed in TMKL and an adaptation goal, produces an adapted model as output. The model is then used to generate new code implementing the adapted model.

Figure 3 illustrates the process for software adaptation implemented by REMng. The process begins with a specification of program goals and their connections to the program code, represented via a TMKL model, and a specification of the differences between the goals delivered by and desired of the

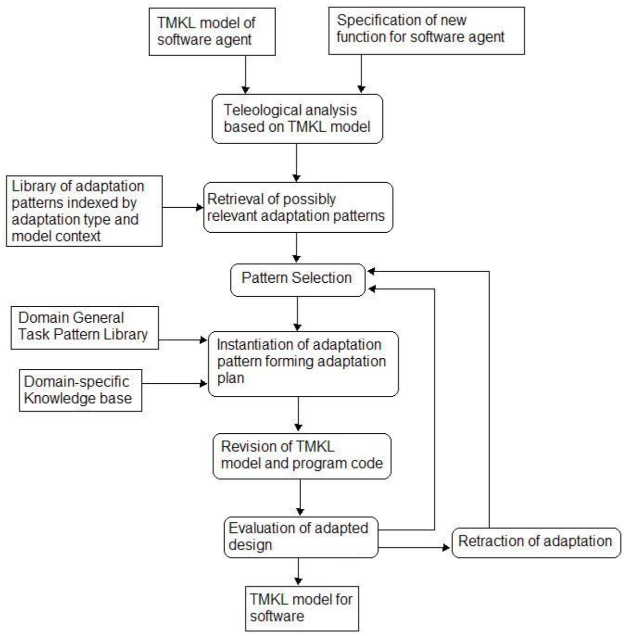

Fig. 3. REMng's process for adaptive software design

software. REMng uses the TMKL model, localizes the needed modifications in the model and identifies adaptation patterns corresponding to the needed modifications. The applicaton of a retrieved adaptation pattern is localized to specific components.

Adaptation patterns have two parts: rules for determining the applicability of the pattern in a particular problem context and procedures for changing model elements to effect the adaptation. Each adaptation pattern is parameterized according to the specifics of the requirement change that caused its retrieval as well as the local context in which it is being applied. This parameterization step results in a fully instantiated and executable set of *adaptation plans*. The parameterization process of an adaptation pattern, for example, may try to find a replacement for a primitive Task in the TMKL model. If such a primitive Task is not available in a component library, the pattern may be rejected and REMng may instead try to parameterize a different retrieved adaptation pattern that abstracts the adaptation goal. If such an abstract primitive Task is available in the pattern library, REMng would be able to parameterize the adaptation pattern to form an adaptation plan, which in turn would instantiate this abstract primitive Task and the specific portion of the program code it points to and adjust them accordingly.

The adaptation is completed by iteratively executing the adaptation plans that were successfully instantiated and asking the user to confirm or reject each change. The TMKL model is correspondingly revised by propagating the effects of the changes to the original model, and the revised model is passed to the code generation component for translation into an executable form. An example of REMng's processing of an adaptation request is presented below in section IV.

D. Generating program code

The primary purpose of the code generation component of GAIA is to provide a mechanism for realizing the effects of adaptations during program execution. A secondary purpose is to provide infrastructure for abstracting away programming-language and implementation-specific details from TMKL models. As a result, models and explanations of purpose can be expressed in a domain-specific manner, and the process of reasoning and adapting TMKL models can take place at a high level of abstraction.

Code generation consists of four phases. The first phase converts the TMKL elements into a type-checked semantic model. In the second phase, the elements of the model are translated into an abstract syntax tree (AST) representation. Nodes in the tree correspond to grammatical constructs in the target programming language. At this point, constraints from the game environment and programming language are absent from the model. Run-time details relevant to the execution of the model, but not necessarily its semantics, must now be reintroduced. Examples include dynamic allocation and freeing of game resources, safe storage and retrieval of game objects, iteration over game objects and how a domain concept in the knowledge base is actually mapped into run-time elements in the game environment. To introduce these concerns, the third phase traverses the AST of the model and expands any domain concepts or operators with the necessary constraints. Also during this phase, probes for producing feedback events to be sent to REMng are inserted into the AST. This feedback will allow REMng to determine whether failure to achieve goals expressed in the TMKL model has occurred, and, if so, to start a retrospective adaptation process. However, this retrospective adaptation is not yet implemented. Finally, the fourth phase takes in the completely augmented AST translated from the original TMKL model and emits pretty-printed code files. Using SAGi, the user can then request that the resulting code be compiled, linked and run.

IV. Case Study

We have studied the teleological approach to software adaptation by applying it to a specific adaptation scenario for the FreeCiv game. To illustrate the approach, we first describe FreeCiv and then the scenario. We then evaluate the approach in the following section.

A. FreeCiv

FreeCiv is written in the C programming language and comprises 157K lines of source code. One component of the game is an AI game-playing agent used when not enough human players have joined the game. The AI agent that is the target of our adaptation experiment comprises 20K lines of code.

B. FreeCiv adaptation scenario

A typical strategy undertaken by the FreeCiv AI agent is to grow the cities until they are big enough to support advanced technology research. A problem with large cities is that citizens can become dissatisfied (unhappy).

In our experiment, we changed the rules of FreeCiv. We added a new rule involving luxury items. Luxuries are special resources that may be randomly found on the FreeCiv game map. The added rule states that if a player's city is connected to a luxury item by a road, the *happiness* value of citizens of that city is increased. The happiness value of a city has secondary impacts, including an effect on population growth of the city.

C. REMng

In the luxury adaptation scenario, REMng works with the TMKL model depicted in Figure 4, without any of the portions contained within the three bold-outlined boxes. The portions within these boxes were added by REMng during the adaptation process, resulting in a new, adapted agent that is able to exploit the new game rule that is added in this adaptation scenario. In the figure, the containing, named rectangles are TMKL Methods. The contained, unlabeled rectangles are Tasks. The lines between Tasks in the Methods depict control flow (state transitions) while the lines from Tasks to Methods illustrate which Methods implement particular Tasks. This figure depicts the ”Largepox” strategy for playing FreeCiv, where the goal is to first build a number of cities (*Initial_M*), grow the population at those cities (*Growth_M*), achieve appropriate technology for military production (*Research_M*), exploit that technology to build a strong military (*Production_M*), and then employ that military to destroy the opponents (*MarineRush*).

In order to produce the adaptations depicted within the bold-outlined boxes of Figure 4, REMng must decide how the game-playing agent should be adapted in order to take advantage of the altered game dynamics of the luxury adaptation scenario. First, REMng must localize the adaptations within the TMKL model, as per the first reasoning step in Figure 3. In order to localize the adaptations, REMng must look into its knowledge base to determine all secondary impacts of increasing the happiness value of a city. Once it obtains this information, REMng must inspect the model of the agent to determine whether there are any goals expressed within the model that could potentially be affected by exploiting the new rule. In this example, REMng's knowledge base contains the information that increasing the happiness value at a city will also increase the population growth rate at that city. By forward chaining from the new rule added in the luxury adaptation scenario, REMng determines that connecting a city to a luxury resource via a road will have a secondary impact of increasing the population growth rate at that city. REMng's subsequent search of the TMKL model will then identify the Growth task (the task parent of the *Growth_M* method depicted in Figure 4) as having a relevant goal, since the goal of this task is to increase population at cities.

Given that one or more such locations are identified, REMng must now look into its library of adaptation patterns to determine whether it knows of any ways to adapt the identified model location(s) to achieve some enhancement relevant to the newly added rule. REMng has now reached the second reasoning step in Figure 3, pattern retrieval. If REMng locates some potentially applicable adaptation patterns, it must then try to instantiate them for the current situation (the following step in Figure 3). Adaptation patterns are general, and thus require parameterization to make them game-, system- and scenario-specific. In this scenario, REMng does find an applicable pattern, *insert-action-for-goal*, which will try to insert processing in the agent's design that will result in the execution of an action to help achieve a particular goal. This pattern is instantiated by locating a specific action to be executed (found in the game-specific knowledge base), *ConnectByRoad*. This action will cause a worker unit to build a road connecting two locations on the FreeCiv game map. REMng must further constrain the parameters of the action to be executed based upon the specific adaptation scenario. Here, REMng sees that the rule specifically requires the connection of a city location to a luxury resource location, and thus constrains the two location parameters of the *ConnectByRoad* action to those that contain the appropriate items.

Now that REMng has an instantiated the pattern to form an adaptation plan, it can actually apply changes to the TMKL model. The application of this plan includes several more plan-specific reasoning steps, including (1) an analysis of the control flow within the portion of the model to be modified, (2) insuring that there are no resource conflicts between existing actions and those that will be inserted, (3) potentially adding secondary actions that establish preconditions of the action to be inserted and (4) the generation of heuristics to control the execution of the action as well as secondary actions arising from (3).

Step (1) finds that the state machine flow in the *Growth_M* method of Figure 4 is sequential and unconditional, and step (2) finds no resource conflicts. Based on these findings, REMng adds a new task to the end of the existing state machine in *Growth_M*. The expanded structure of this new task is based upon REMngs analysis of the action to be executed (*ConnectByRoad*), its parameters, and the constraints on those parameters. Step (3) finds that we may not have the worker units necessary to execute *ConnectByRoad*, so it inserts the new task under the *GrowthImprovements_M* method after a recursive invocation of the add-task-for-goal adaptation pattern set up with a new parameterization for this secondary adaptation. Step (4) is handled by making use of a cost annotation that is attached to the *ConnectByRoad* action within the domain-specific knowledge base. At this time REMng will prefer to execute the lowest-cost parameterizations of *ConnectByRoad*. This is not necessarily the best heuristic, and step (4) is likely a place where the human designer may wish to intervene in order to insert a better criteria for ranking of possible action parameterizations.

REMng then proceeds to the Evaluation step of Figure

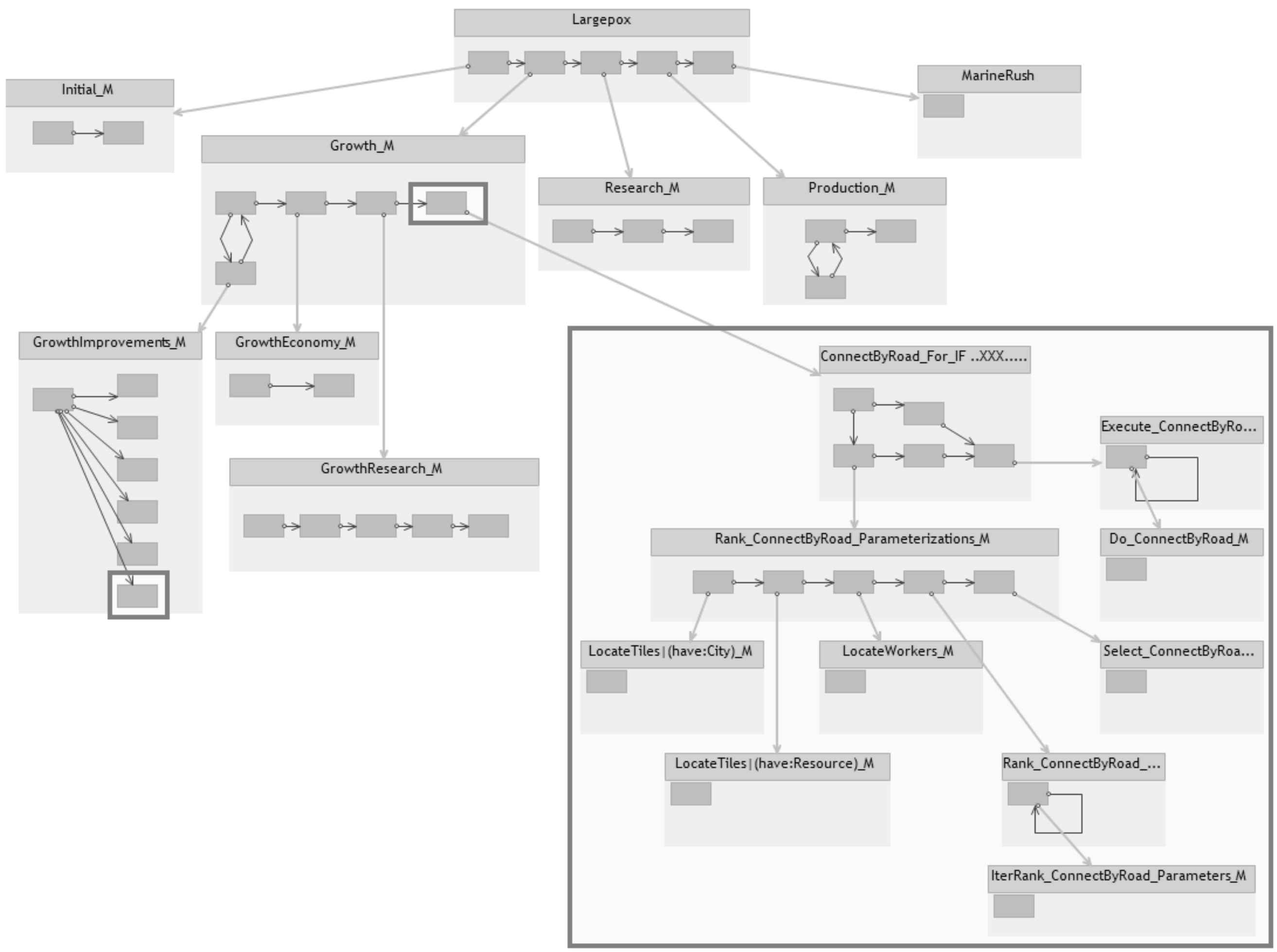

Fig. 4. Detailed TMKL model for agent player in the FreeCiv case study. In this figure, the containing, named rectangles are TMKL Methods. The contained, unlabeled rectangles are Tasks. The lines between Tasks in the Methods depict control flow (state transitions) while the lines from Tasks to Methods illustrate which Methods implement particular Tasks. Areas enclosed in bold boxes are added during adaptation in this case study.

3 and displays the model to the user for validation. If the user rejects the adaptation, REMng retracts it and, in this case, terminates, as there are no further retrieved adaptation patterns to be tried. If the user accepts REMng's adaptation, the result is a new, adapted TMKL model to be run in the modified FreeCiv environment containing the new game rule. This model is handed off to code generation mechanism to produce executable code for the adapted FreeCiv agent.

D. Code generation

FreeCiv and its unadapted AI player are written in the C programming language. Adapting the C code directly requires knowledge of both C's syntax and also of low-level implementation details. For the experiment described in this paper, we were more concerned with adaptation at the strategic level. We therefore decided to separate the programming-language-specific issues of reverse engineering and code embedding from the core research question of reasoning about adaptation. To effect this separation, we devised a domain-specific language for modeling games. The language contains both an ontology of game concepts and a reference architecture for game-playing agents. We then manually reverse engineered the FreeCiv AI agent into this language for use by REMng. After REMng produces an adapted model, the code generator produced an updated C program capable of playing the game with the luxury rule described above.

The input to the code generation process is the adapted TMKL model produced by REMng. The result of generating code from the model is a set of C files. These can then be compiled and linked with both our run-time library and the FreeCiv game engine to produce an executable program capable of autonomously playing FreeCiv. The generated code comprises a file containing implementations of the finite state machines inherent in the TMKL Methods and a file containing a set of queries of (accesses into) the TMKL knowledge base. The run-time library comprises two components responsible for realizing the domain-specific language. The first is a generic player capable only of communicating with the game engine on a turn-by-turn basis, sending it any outstanding requests through the game engine's API. The second component contains implementations for *commands*. Commands denote TMKL primitive Tasks for performing actions in the game. When executed, the state machines produce a sequence of commands that are queued for transmittal to the game engine by the generic player. When the game engine has completed one turn in the game, the state machines can resume their activities, possibly making use of the compiled queries to learn about changes in the game state. Execution progresses in this fashion, alternating between agent action and game engine updates until the game is completed.

V. Discussion

We have learned many things while conducting our research with TMKL and FreeCiv. Specifically, we can now begin the process of designing a next-generation teleological language. Interesting extensions include program structures, enhanced inferencing and knowledge modeling, the ability to express non-functional requirements and automated reverse engineering used to produce the original, unadapted TMKL models.

Program structures: TMKL is strictly hierarchical and sequential. Unfortunately, real programs are more complex. In FreeCiv it is often the case that conflicting goals must be reconciled. It makes sense to be able to reason about the goals separately and modularly and then use a reconciliation mechanism to arbitrate. For example, having more soldiers supports safety and ultimate victory, but taxes are required to support them, thereby reducing the happiness and productivity of citizens. Furthermore, to model agents capable of concurrent processing or to model independent, concurrently executing agents, it makes sense to add supporting program structuring mechanisms.

Enhanced reasoning: REMng is specialized for situations that arise when adapting FreeCiv. The following generalizations are contemplated.

- The ability to handle hypothetical situations.
- The ability to plug in external reasoners such as planners and learners.
- The ability to provide explanations for decisions.
- The ability to support classification and generalization such as would be provided by an off-the-shelf knowledge representation and reasoning system like PowerLoom (http://www.isi.edu/isd/LOOM/PowerLoom/).
- The ability to reason effectively at the meta-level in situations when making strategic decisions involving multiple conflicting goals is required. One possibility is to devise a TMKL model of REMng so that it can reason about itself.

Knowledge modeling: The level of abstraction provided by the Knowledge component of TMKL is low. In particular, our experiments with FreeCiv suggest the addition of the following devices to the language.

- The ability to aggregate similar domain elements.
- The ability to aggregate domain operations by, for example, providing by a macro capability.
- A specialized notation for expressing the adaptation to be made. With the FreeCiv experiment, the adaptation amounted to a rule change, but *rule* is a domain concept, and TMKL needs a more general mechanism.

Design rationale and non-functional requirements: TMKL has been historically concerned with functional requirements. Non-functional concerns, such as performance and security, which require modeling time and information flows, have not been included. Typically, non-functional concerns are addressed during program design by introducing additional program elements, for example, caches and security monitors, but TMKL does not explicitly support the connection between a non-functional requirement and the additional program element. Moreover, the reasons for making a specific choice, its *design rationale,* are also not documented.

Reverse engineering: The original, unadapted TMKL model of the AI FreeCiv player described above was produced manually. A more powerful software adaptation environment would support the automatic analysis of the subject program in order to construct the corresponding knowledge base and Methods. Such automation is well beyond the current state of the art for reverse engineering. Nevertheless, it may be possible to generate abstract approximations that would provide sufficient insight to take advantage of REMng's reasoning capabilities. A related challenge is the fact that we currently completely regenerate the code for the entire AI agent, even when only a part of its structure was changed by adaptation. Incremental code generation may be necessary to scale our approach to handling larger systems.

VI. Conclusion

We have described a language, TMKL, and a tool, GAIA, that supports the adaptation of game-playing agents. We have experimented with our approach to agent adaptation in the context of an open-source game called FreeCiv. We have adapted its AI game-playing agent in the situation where a new game rule has been added. A key enabler for this approach is the close connection between the game requirements, as expressed by TMKL Tasks, and the mechanisms by which the requirements are implemented, TMKL Methods. The connections between these requirements and the mechanisms that achieve them are represented by the use of a teleological modeling language, TMKL. These teleological agent models allow an adaptation process to localize required changes within the modeled agent's design, and then to effect adaptation of the agent at identified locations. The initial work on the adaptation scenario described here lends support to our overarching hypothesis that teleology is a central organizing principle of knowledge representations that enable self-adaptation of reasoning processes. In addition to the enhancements described in Section 5, we plan to experiment first with other strategy games and then move to other types of games, such as first-person shooters.

Acknowledgements

We gratefully acknowledge support by the National Science Foundation, Science of Design Program, under Grant #0613744. We thank Sameer Indrapu, Lee Martie and Derek Richardson for their contributions to the GAIA project.

References

[1] M. Genesereth, "An overview of meta-level architecture," *In Proceedings of the Third National Conference on Artificial Intelligence*, pp. 119–123, 1983.

[2] L. Birnbaum, G. Collins, M. Freed, and B. Krulwich, "Model-based diagnosis of planning failures," *In Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 318–323, 1990.

[3] E. Stroulia and A. Goel, "Functional representation and reasoning in reflective systems," *Journal of Applied Intelligence, Special Issue on Functional Reasoning*, vol. 9, no. 1, pp. 101–124, 1995.

[4] D. B. Leake, "Experience, introspection and expertise: Learning to refine the case-based reasoning process," *J. Exp. Theor. Artif. Intell.*, vol. 8, no. 3-4, pp. 319–339, 1996.

[5] J. W. Murdock and A. K. Goel, "Learning about constraints by reflection," *Canadian Conference on AI*, pp. 131–140, 2001.

[6] ——, "Localizing planning with functional process models," *ICAPS*, pp. 73–81, 2003.

[7] ——, "Meta-case-based reasoning: self-improvement through self-understanding," *J. Exp. Theor. Artif. Intell.*, vol. 20, no. 1, pp. 1–36, 2008.

[8] E. Stroulia and A. K. Goel, "Evaluating psms in evolutionary design: the autognostic experiments," *Int. J. Hum.-Comput. Stud.*, vol. 51, no. 4, pp. 825–847, 1999.

[9] J. Jones and A. Goel, "Metareasoning for adaptation of classification knowledge," in *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, May 2009, pp. 1145–1146.

[10] T. Mens, J. Buckley, M. Zenger, and A. Rashid, "Towards a taxonomy of software evolution," 2003. [Online]. Available: citeseer.ist.psu.edu/mens02towards.html

[11] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance*, vol. 13, no. 1, pp. 3–30, 2001.

[12] A. Goel, J. Jones, C. Parnin, S. Rugaber, and A. Sinharoy, "Teleological reasoning in software adaptation: A case study in game-playing agents," *Proc. of the Third International Conference on Design Science Research in Information Systems and Technology (DESRIST-08)*, pp. 365–379, May 2008.