Teleological Modeling and Reasoning for Automated Software Adaptations

Joshua Jones, Ashok Goel and Spencer Rugaber

College of Computing
Georgia Institute of Technology
Atlanta, USA 30332
{jkj, goel, spencer}@cc.gatech.edu

1 Background, Motivation and Goals

The design of a long-living software artifact evolves through many versions. Changes in the design 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 design 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).

AI research on conceptual design of physical devices has revealed the central role that teleological knowledge and reasoning play in automated adaptive design [1] [2] [3] [4] [5] [6] [7] [8]: (i) a declarative teleological model of a physical device that explicitly captures the teleological relationships among the structure, behaviors and functions of its design enables localization of the modifications needed to the structure to achieve new functions, and (ii) the ontology of teleological models provides a vocabulary for classifying, representing, indexing and accessing specific design cases, generic adaptation plans, primitive design components, and abstract design patterns. In analogy to physical devices, we view software artifacts as abstract devices, i.e., as abstract teleological artifacts with structures and behaviors that result in the accomplishment of desired functions [9] [10] [11] [12] [13]. Recent AI research on self-adaptation in software agents makes a further analogy between adaptive design of physical and software devices [14] [15] [16] [17] [11] [18] [19] [20]. This leads to our research hypothesis: *function and teleology are basic organizational principles of adaptive software design.*

[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.

Of course, software artifacts are also quite different from physical devices because the notions of structure, behavior, function and teleology in the two domains are different. Thus, the goals of this research are to address the following six questions [21]:

1. What are productive notions of structure, behavior, function and teleology in software design?
2. How might a declarative teleological model of a software artifact specify its functional architecture?
3. How might teleological analysis enable localization of the modifications needed to the architecture of a software artifact to achieve new functions?
4. How might the ontology of the teleological models of software artifacts support classification, representation, indexing and retrieval of generic adaptation plans, primitive software components, and abstract design patterns?
5. How might a teleology-driven process of adaptive software design look?
6. How might these building blocks be put together to enable interactive software design more easily, efficiently and effectively than is the case at present?

In particular, the project (i) focuses on changes in the functional design requirements of game-playing software agents (as opposed to their performance or operating requirements), and (ii) emphasizes software design at the architectural level of abstraction (as opposed to the algorithmic and code levels). In order to insure task generality, we intend to study both proactive and retrospective design adaptations. The following discussion uses proactive design adaptations as an example because proactive adaptations typically are harder than retrospective adaptations.

2 Game-Playing Domain

The domain for initial experimentation is the computer-based strategy game, FreeCiv (http://www.freeciv.org). The 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 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.

2.1 Adaptation Tasks

Table 1, adapted from [22] [23], 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 this 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. The proposed project will not only enhance this preliminary taxonomy, for example by adding subclasses to the classes, but will also identify generic adaptation plans and abstract design patterns corresponding to the classes (and subclasses) 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 and/or E units*
3. Extrapolating	*Change composition of F and/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*

As an example, consider the following adaptation scenario: The Ancients modification package for Freeciv (www.cs.utexas.edu/users/bdbryant/freeciv/ancients.html) 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 adapt itself to play the Ancients version of Freeciv? Note that this is an example of the eighth adaptation task (Abstracting) in Table 1.

2.2 Adaptation Data

Our initial analysis of the FreeCiv Subversion software repository is based on taxonomies drawn from existing literature in the software engineering community [24] [25]. Several aspects of the taxonomy presented by Mens, et al. are features of a software system as a whole, as opposed to features of a specific modification. Further, some of the dimensions of variance in software evolution that are discussed are not informative in terms of distinguishing amongst changes that have been made to the FreeCiv code base. However, two features, the 'why' (intent) and 'where' (scope) aspects of changes, are useful starting points for this work in terms of understanding the evolution of FreeCiv and the impact that model based reasoning can have on the types of changes identified.

There are 7 categories of change along the 'why' axis.

- *Groomative* changes are intended to increase the legibility of code, e.g. improving comments or indentation.
- *Preventive* changes are intended to prevent or 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.
- *Reductive* changes remove user-experienced functionality from a software system.
- *Corrective* changes are "bug fixes", correcting some problem with user-experienced functionality.
- *Enhancive* changes improve or extend user-experienced 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.

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 human analysis of the changes and associated Subversion comments. There were a total of 304 changes during this time period that we categorized. [26]

Figure 1 shows a breakdown of changes by their intent. As can be seen, groomative, enhancive, and corrective changes make up the bulk of those that occurred in the time period studied. Figure 2 illustrates the distribution of the changes studied over three scoping categories. *File* scope changes are those that touch code only within a single file, *module* scope changes are those that affect code only within one directory in the source hierarchy of the project, and *cross-module* changes are those that involve modifications to code in more than one directory. As might be expected, there are progressively fewer changes as scope

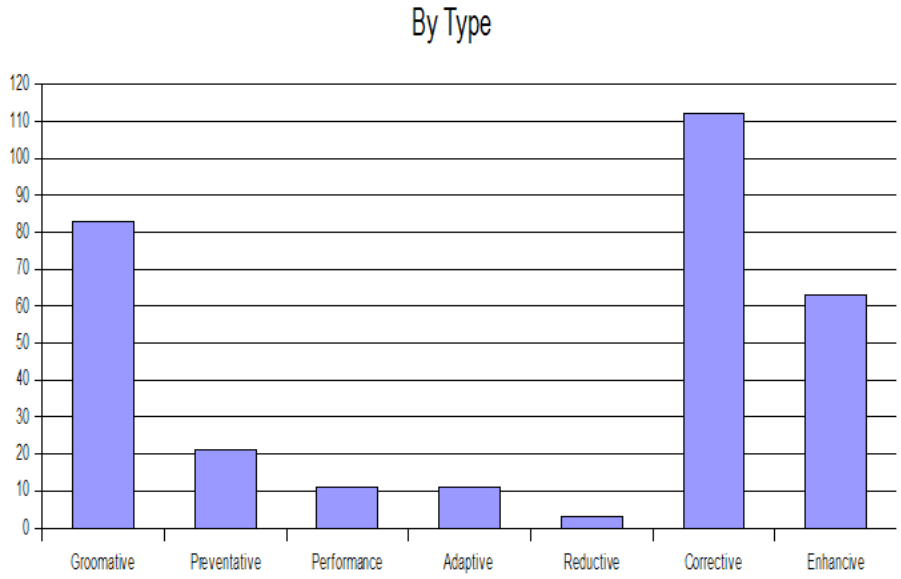

Fig. 1. Changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by intent.

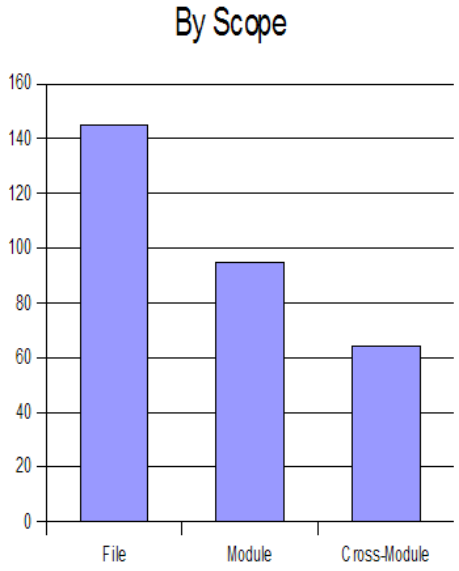

Fig. 2. Changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by scope.

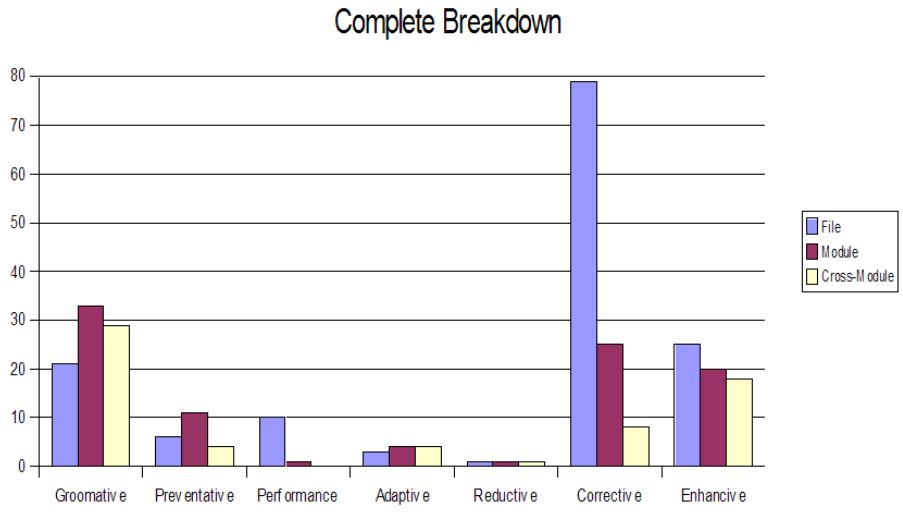

Fig. 3. Changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by both intent and scope.

increases. We were interested in the nature of the cross module changes – since these are the changes with highest impact, any automation we are able to provide with respect to these changes will be most significant. To this end, we more closely examined the cross-module changes found within the sample period, and further broke them down across three categories. *Reasonable* changes are those that clearly need to impact multiple modules, e.g. that affect the interfaces between components, or groomative changes like adjusting variable names in multiple modules to be in line with convention. *Reorganizational* changes are those that have to do with migrating functionality from one module to another. *Dubious* cross-module changes are those that could have been intra-module changes if different organizational decisions had been made. Figure 4 depicts the results of this analysis.

For the same purpose, we wanted to understand how cross-module changes specifically broke down across the 'why' dimension in the basic taxonomy used for analysis. Figure 5 illustrates these results.

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 due to GUI-

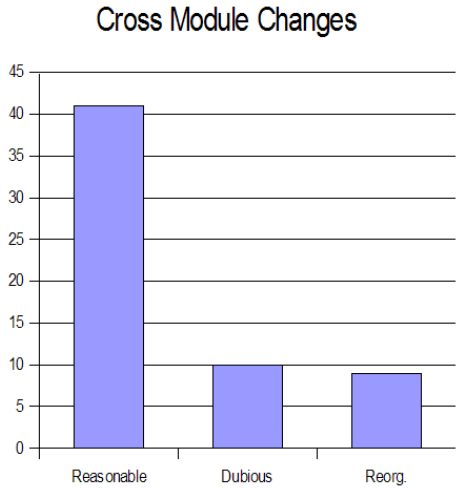

Fig. 4. Cross-module changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by (subjective) cause.

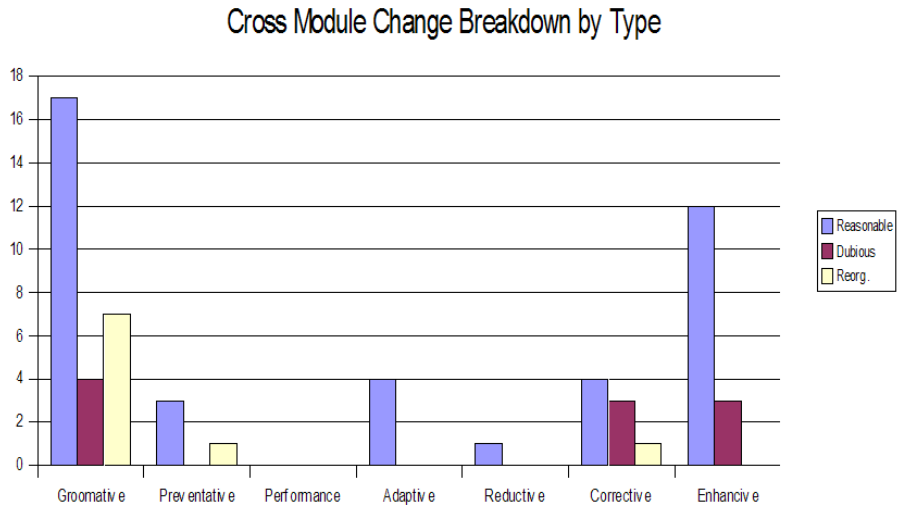

Fig. 5. Cross-module changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by intent.

based changes, with which it is less likely that automated reasoning can have significant impact, and core feature changes, where the potential for impact seems higher. Figures 6 and 7 show the results of this aspect of the study.

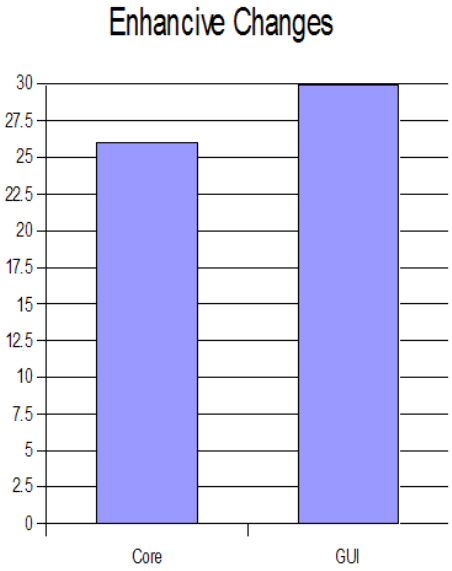

Fig. 6. Enhancive changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by GUI-based or core functionality modification.

As Figure 7 illustrates, 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.

3 Teleological Modeling and Reasoning

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) [15] [16] [17] [10] [11] [27], that makes use of a knowledge representation called Task-Method-Knowledge Language (TMKL) [28] [29] [30]. 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.

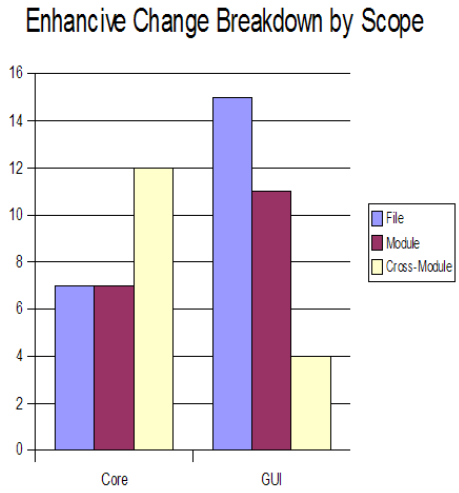

Fig. 7. Enhancive changes in the FreeCiv Subversion Repository 3/15/99 to 7/28/99, broken down by GUI-based or core functionality modification and scope.

3.1 Task-Method-Knowledge Language

Software systems modeled in TMKL are divided into tasks, methods, and knowledge. A task describes user intent in terms of a computational goal producing a specific result. A method is a unit of computation that produces a result in a specified manner. 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. 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 knowledge 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 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 binding. Unimplemented tasks cannot be executed until the model is modified. 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 first 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 [31], 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 an enormous 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. Hoang, Lee-Urban and Munoz-Avila [32] have shown that TMKL has a simpler syntax with the expressive power of Hierarchical Task Networks in classical planning [33] [34] [35].

Figure 8 and Table 2 illustrate an example of a TMKL model that REM is able to use in reasoning [36]. The piece of 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 8, 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 at the procedural level for execution as described below. The Defend-City task is executed each turn that the agent is not building a defensive unit in a particular

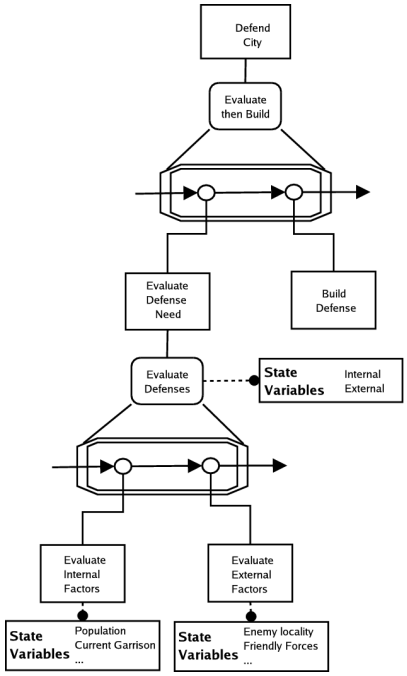

Fig. 8. Depiction of the *Defend-City* model

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 the discussion in this section is neither to describe how software engineers reason about adaptive design problems, nor to prescribe how they should do so. Instead, the goals are to suggest how an automated system may

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

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 written in TMKL.

Figure 9 illustrates a process for adaptive software design. The process begins with (a) 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 TMK model of the agent's functional architecture, and (b) 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 TMK model for teleological analysis, localizes the needed modifications in the TMK model (and the program code), and spawns adaptation goals corresponding 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.

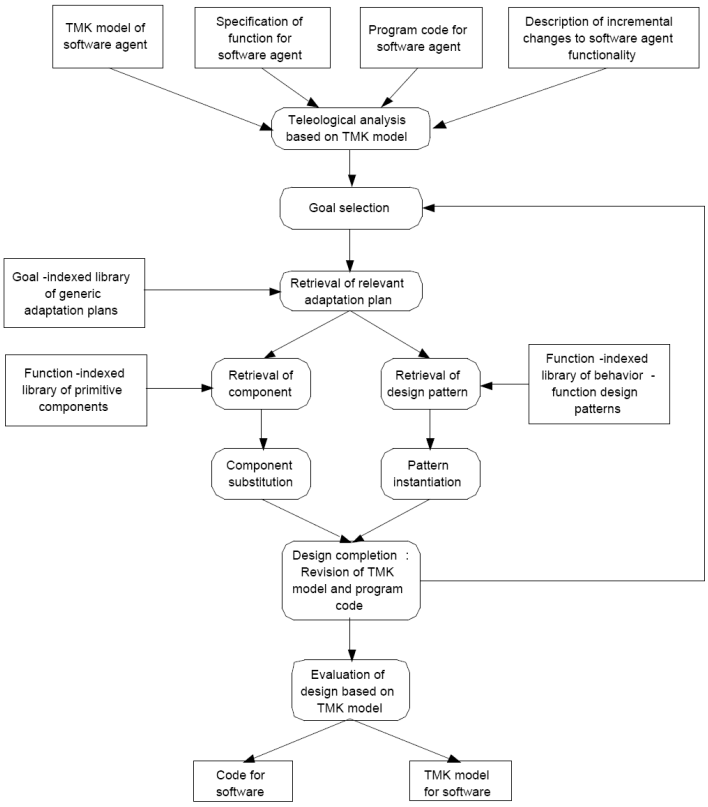

Fig. 9. A Partial Description of a Process for Adaptive Software Design

Each adaptation's goal is used to retrieve generic adaptation plans applicable to it. 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 TMK model of the Defend-City agent. Let us suppose that such a procedure is not available in the component library. The second adaptation plan 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 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 plan 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. In the next section, we discuss possible adaptation strategies for various types of software changes.

5 Potential Applications

This section details the potential that we believe that teleological reasoning has to support software adaptation, and also discusses a perspective on software evolution that arises from these considerations.

5.1 Groomative

It is unlikely that teleological model-based adaptation will have significant application to groomative changes, as these changes do not affect the functionality of the code, but are basically purely syntactic changes. However, these kinds of changes also have the tendency to be straightforward to implement, even if they are broad in scope. So, it is likely that other kinds of tools will be or already are effective in supporting this kind of change.

5.2 Preventive

Like groomative changes, preventative changes are primarily semantic manipulations that do not affect functionality, e.g. "factoring out" duplicated code into a single function. One could imagine some use for a TMKL-like model in identifying places where this kind of change might be useful, perhaps by searching the model for tasks with very similar characteristics (goals, knowledge requirements, etc). However, this class of modifications does not appear to be a primary application area for model-based adaptation, though there is the potential for some impact here.

5.3 Performance

Performance changes also do not modify the tasks achieved by a software system. However, these changes do specifically modify the methods used to achieve tasks, in order to be more efficient in terms of resource usage. Teleological reasoning could be useful in making such adaptations, if a library of methods is available. Then, task descriptions can be used to index this library in order to retrieve methods with desired performance characteristics. However, the extent to which it is possible to build a library of methods for reasonably complex tasks is not clear, and is an outstanding research issue.

5.4 Adaptive

Adaptive changes, like performance changes, will require changing the methods used to achieve tasks. As with performance changes, it might be possible to create libraries of methods for adaptive modification to various environments that could then be used to automate the porting of software to new contexts. These method libraries could in principle be developed once and then applied to multiple software systems, and can be seen as serving the same function as abstraction layers. One benefit of performing adaptation in this way is that holes in the adaptive method library can be automatically highlighted by teleological reasoning.

5.5 Reductive

It is likely that reductive changes can be assisted through teleological reasoning over TMKL-like models. Because the relationships between tasks in terms of control and knowledge flow are explicit in TMKL models, the impact of removing a piece of functionality on other tasks and methods should be identifiable, allowing an automatic reasoner to at least identify other sections of code that may require modification due to the removal, if not also assist further with these secondary modifications.

5.6 Corrective

Teleological reasoning can potentially be useful in identifying tasks that are not being satisfied by their associated methods, and in providing some information about the specific way in which the methods are failing. In this sense, TMKL-like models can assist with localization of faults that are the target of corrective modifications. However, it is not clear that the corrective modification itself can be automated via teleological reasoning, as this is likely to require knowledge of implementation specifics that are beyond those captured by the model. However, to the extent to which localization of faults can be performed, teleological analysis will have application to this type of change.

5.7 Enhancive

Given a high-level specification of an intended enhancement in terms of goals for functionality, model-based reasoning should be able to decompose those goals into a set of subgoals consisting of task modifications and/or additions at various places in the system model. As with corrective changes, it is not clear that the actual modifications or additions can be automated, but at least some useful guidance could be automatically generated.

5.8 Summary

Based on the rationale outlined in this section, teleological reasoning could have impacted the following proportion of changes in the sample taken from the FreeCiv repository:

- 66% of overall modifications
- 49% of cross-module changes
- 54% of multi-file changes within modules
- 81% of changes within files

At this time, it is not clear how well the sample set of changes studied reflects the typical breakdown of changes in game-playing agents in general. However, this analysis in conjunction with the data we have gathered from FreeCiv does lend support to the idea that teleologival reasoning can have impact in terms of (partial) automation of software evolution.

5.9 Towards A New Taxonomy of Software Evolution

In the above analysis, a few patterns of automation of software evolution through teleological model-based reasoning arise. Based on these patterns, we have begun to outline a new taxonomy of software evolution in terms of TMKL-like models. In this light, we have identified the following categories:

- Superficial Changes – These changes do not impact either functionality or the means by which functionality is achieved, but rather are primarily geared to human readability of a codebase. Groomative changes fall into this category.
- Task-Method Reorganization – These changes are primarily concerned with modifications to the dependencies among tasks and methods. For instance, factoring out common code used to achieve two tasks within a software system into a common function has a primary impact of collapsing a portion of the task-method hierarchy. Some preventative changes will fall into this category.

– Method Modification – These changes alter the procedures by which task goals are achieved. Changing the implementation specifics but not the functional aspects of a software system are method modifications. Performance, adaptive, and many corrective changes will fall into this category.

– Task Modification – These changes alter the functional goals of (portions of) a software system. Enhancive and reductive changes will fall into this category, and sometimes corrective changes will as well.

These categories are presented roughly in order of the degree of impact on a system. It is interesting to note that the more impactful changes will sometimes mandate the less impactful; that is, each change type listed can require the changes listed above it (task modification will often require method modification, etc). So in some sense each change type subsumes those above it, though this is not a hard and fast principle.

We will continue to refine and reflect on this taxonomy as work proceeds on this project in order to glean new insights into the nature of software change.

6 Current Work

The goals of ongoing work on this project are broadly to address those issues enumerated in the introduction – to offer new perspectives on understanding software evolution in game-playing agents, and to apply teleological reasoning to automate provide assistance with the implementation of those adaptations wherever possible. In particular, 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 [37]. This suggests that TMKL could be useful for representing game-playing scripts.
– Mapping Object-Oriented Software Design - In using TMKL to represent to game-playing agents, we have found that game-playing agents often use object-oriented software design. This raises the issue of building maps between object-oriented software designs and TMKL. One potential answer to this issue is to assign functional roles to the objects in object-oriented designs, and use these functional roles as maps to TMKL models.
– Generality of REM - We are evaluating and enhancing the generality of REM 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 REM - In experimenting with various software agents, we have found that REM often is too slow for use in many game-playing situations. This is in part because REM 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 REM currently delivers. In the example of Defind-City task described earlier, we ported the needed portions of REM's capabilities to a specialized agent to achieve this kind of efficiency [36].

- Integration with Generative Planning - REM uses GRAPHPLAN [38] both as an alternative to self-adaptation using teleological reasoning, and to complete partial solutions generated by self-adaptation. GRAPHPLAN is very fast, general-purpose generative planner. We are also experimenting with the use of FASTFORWARD [39], another very fast, general-purpose generative planner.
- Integration with Reinforcement Learning - REM also uses Q-Learning [40], a form of reinforcement learning [41] [42], as another alternative to self-adaptation using teleological reasoning, and to complete partial solutions generated by self-adaptation. We are experimenting 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 [36].
- 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 action in a game. We are developing a domain specification 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 [43] [44]. 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.
- Libraries of Tasks and Methods - Chandrasekaran has described a functional analysis of intelligent agents in terms of patterns of primitive tasks (called Generic Tasks) that occur again and again in intelligent agents [45] [46] [47]. More recently, he has proposed an ontology of generic tasks and generic methods [48] [49]. We are developing a library of generic tasks and generic methods useful in games such as Freeciv. This kind of library is needed for composing game-playing agents and for conducting task and method modifications.

Acknowledgments

This research is supported by a NSF Science of Design Grant (#0613744) on Teleological Reasoning in Adaptive Software Design. This paper has benefited from discussions with members of the Self-Adaptive Game-Playing Agents project,

including Summer Adams, Niyant Krishnamurthy, Chris Parnin, Derek Richardson, and Rucheek Sangani.

References

1. Bhatta, S., Goel, A.K.: A functional theory of design patterns. Proc. 15th International Joint Conf. on Artificial Intelligence (IJCAI-97) (1997) 294–300
2. Bhatta, S., Goel, A.K.: Learning generic mechanisms for innovative design adaptation. Journal of the Learning Sciences **6**(4) (1998) 367–396
3. Freeman, P., Newell, A.: A model for functional reasoning in design. Second Int. Jnt. Conf. on AI (IJCAI-71) (1971) 621–640
4. Gero, J., Tham, K., Lee, H.: Behavior: A link between function and structure in design. Intelligent Computer Aided Design (1992) 193–225
5. Goel, A., Bhatta, S.: Design patterns: An unit of analogical transfer in creative design. Advanced Engineering Informatics, Special Issue on Ontologies **18**(2) (2004) 85–94
6. Goel, A.K., Chandrasekaran, B.: Functional representation of designs and redesign problem solving. 11th International Joint Conf. on Artificial Intelligence (IJCAI-89) (1989) 1388–1394
7. Umeda, Y., Tomiyama, T.: Functional reasoning in design. IEEE Expert **12**(2) (1997) 42–48
8. Umeda, Y., Takeda, H., Tomiyama, T., Yoshikawa, H.: Function, behavior, and structure. AIENG '90 Applications of AI in Engineering (1990) 177–193
9. Allemang, D., Chandrasekaran, B.: Functional representation and program debugging. Knowledge-Based Software Engineering (1991) 136–143
10. Murdock, J.W., Goel, A.: Localizing planning using functional process models. Proc. International Conference on Automated Planning and Scheduling (ICAPS-03) (2003)
11. Murdock, J.W., Goel, A.: Meta-case-based resaoning: Self-improvement through self-understanding. Journal of Experimental and Theoretical Artificial Intelligence (in press)
12. Stroulia, E., Goel, A.: Functional representation and reasoning in reflective systems. Journal of Applied Intelligence, Special Issue on Functional Reasoning **9**(1) (1995) 101–124
13. Stroulia, E., Syst, T.: Dynamic analysis for reverse engineering and program understanding. Applied Computing Review **10**(1) (2002)
14. Goel, A., Stroulia, E., Chen, Z., Rowland, P.: A model-based approach to reconfiguration of schema-based reactive control architectures. Proc. AAAI Fall Symposium on Model-Based Autonomous Systems (1998)
15. 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
16. 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
17. Murdock, W., Goel, A.: Learning about constraints by reflection. Proc. 14th Biennial Conference of Canadian AI Society (2001) 131–140
18. 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)

19. 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
20. 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
21. Goel, A., Rugaber, S.: Teleological reasoning in adaptive software design. Proposal submitted to the NSF Program in Science of Design (2006)
22. Aha, D.W., Molineaux, M., Ponsen, M.J.V.: Learning to win: Case-based plan selection in a real-time strategy game. (ICCBR)
23. : Darpa proposer information pamphlet for transfer learning. BAA 05-29 (2005)
24. Mens, T., Buckley, J., Zenger, M., Rashid, A.: Towards a taxonomy of software evolution (2003)
25. Chapin, N., Hale, J.E., Kham, K.M., Ramil, J.F., Tan, W.G.: Types of software evolution and software maintenance. Journal of Software Maintenance **13**(1) (2001) 3–30
26. Jones, J., Goel, A., Rugaber, S.: Automating software evolution. Proc. of the Science of Design Symposium (2007)
27. Murdock, J.W.: Self-Improvement Through Self-Understanding: Model-Based Reflection for Agent Adaptation. PhD thesis, Georgia Institute of Technology (2001)
28. Murdock, J.W.: Modeling computation: A comparative synthesis of tmk and zd. College of Computing Technical Report GIT-CC-98-13 (1998)
29. Murdock, J.W.: Semi-formal functional software modeling with tmk. College of Computing Technical Report GIT-CC-00-05 (2000)
30. Murdock., J.W.: Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation. PhD thesis, College of Computing, Georgia Institute of Technology (2001)
31. MacGregor, R.: The Loom Knowledge Representation Language. University of Southern California. Marina del Rey, CA, USA. (1987)
32. 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)
33. Erol, K., Hendler, J., Nau, D.: Htn planning: Complexity and expressivity. Proc. Twelfth National Conference on Artificial Intelligence (AAAI-94) (1994)
34. Sacerdoti, E.: A structure for plans and behaviors. (1977)
35. Tate, A.: Generating project networks. Proc. International Joint Conference in Artificial Intelligence (1977) 888–893
36. 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)
37. Adams, S., Goel, A.: Stab: Making sense of vast data. Proc. IEEE Conference on Intelligence and Security Informatics (2007)
38. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence **90**(1-2) (1997) 281–300
39. 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
40. Watkins, C.: Modeling Delayed Reinforcement Learning. PhD thesis, Psychology Department, Cambridge University, United Kingdom (1989)
41. Barto, A., Sutton, R., Brouwer, P.: Associative search network: A reinforcement learning associative memory. Biological Cybernetics **40**(3) (1981) 201–211

42. Sutton, R., Barto, A.: Reinforcement learning: An introduction. (1998)
43. Jones, J., Goel, A.: Knowledge organization and structural credit assignment. Proc. IJCAI-05 Workshop on Reasoning, Representation and Learning in Computer Games (2005)
44. Jones, J., Goel, A.: Structural credit assignment in hierarchical classification. Proc. International Conference on Artificial Intelligence (2007)
45. Chandrasekaran, B.: Generic tasks in knowledge-based reasoning: High-level building blocks for expert systems design. IEEE Expert **1**(3) (1986) 23–30
46. Chandrasekaran, B., Johnson, T.R.: Generic tasks and task structures: History, critique and new directions. Second Generation Expert Systems (1993) 232–272
47. Chandrasekaran, B., Johnson, T., Smith, J.: Task structure analysis for knowledge modeling. Communications of the ACM **33**(9) (1992) 124–136
48. Chandrasekaran, B., Josephson, J.R., Benjamins, V.R.: Ontology of tasks and methods. Banff Knowledge Acquisition Workshop (1998)
49. Chandrasekaran, B., Josephson, J.R., Benjamins, V.R.: What are ontologies and why do we need them? IEEE Intelligent Systems **14**(1) (1999) 20–26