Incorporating Multiple Self-Adaptive Agents in Games

Steven Streasick
streasis@mail.gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

Erik M. Fredericks
frederer@gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

Byron DeVries
devrieby@gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

Ira Woodring
woodriir@gvsu.edu
Grand Valley State University
Allendale, Michigan, USA

ACM Reference Format:
Steven Streasick, Erik M. Fredericks, Byron DeVries, and Ira Woodring. 2025. Incorporating Multiple Self-Adaptive Agents in Games. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696630.3730555

Abstract

A self-adaptive system (SAS) is capable of modifying its behavior at run-time to address uncertainty. For games, these self-adaptations can present a more dynamic experience (e.g., changing difficulty, optimizing performance), thereby enabling run-time updates to mitigate potential issues experienced during gameplay. For example, a self-adaptation may result in emergent behaviors that keep the player engaged or optimize performance to support a multitude of device configurations. Notably, games that leverage a run-time feedback loop have previously demonstrated success in optimizing a game's frame rate. However, multi-agent systems that incorporate self-adaptation remain largely unexplored in the video games domain. This paper demonstrates a novel approach for using multiple goal models with competing metrics for expressing optimal behavior in balancing and mitigating video game uncertainties. To support this goal, we adapt an existing browser-based game to a new framework that incorporates two distinct self-adaptive agents with potentially competing objectives.

1 Introduction

Games that are designed to adapt to player performance can create a more engaging and resilient game experience for the player. For example, a game that scales its difficulty to the player can provide a balance between keeping the player engaged and avoiding frustration, thereby achieving a flow state [8, 18]. This approach to game design is known as dynamic difficulty adjustment (DDA), where DDA can mitigate "staleness" a game may present if the player grows tired of its mechanics over time [31, 37]. However, other targets for optimization and adaptation can be useful for supporting a satisfactory player experience, such as maintaining an adequate frame rate and minimizing technical glitches. Here, dynamic performance adjustment (DPA) can be used to optimize player experience by adapting as a result of unexpected situations or negative behaviors expressed as a result of system limitations [22]. In this paper we focus on DPA, however we anticipate that DDA could be incorporated as well.

One approach for enabling adaptation is to deploy an application as a self-adaptive system (SAS). An SAS can modify its behavior to adjust for changes that occur within the system and environment [15, 21]. Self-reconfigurations occur autonomously as a result of decisions made by a controlling feedback loop to optimize a system and/or mitigate uncertainty. One common model for implementing a self-adaptive feedback loop is the *monitor*, *analyze*, *plan*, and *execute* with a shared *knowledge* base (MAPE-K) [15]. For example, a video game that employs a MAPE-K loop may self-reconfigure in the event that a network connection is laggy to reduce "rubber-banding" by offloading server-based decisions to a local (yet ideally secured) process. Additionally, a MAPE-K loop can be implemented to allow for adaptations based on real-time system metrics (e.g., CPU usage, memory consumption, request load) [11]. Previous works have demonstrated the feasibility of an SAS loop within game development [11, 31]. In practice, this often diminished the cohesion of the game, with several unrelated operations being held within the singular SAS loop.

In this paper we propose a novel approach for incorporating independent agents within a game environment that each have the capability to self-reconfigure with the aims of optimizing both performance and difficulty, where some metrics may be considered adversarial in nature. As such, we focus on providing framework support for incorporating DPA and DDA in our proof-of-concept system comprising multiple agents that each implement their own respective MAPE-K loop. The potential problem with including both DDA and DPA in a singular system is that an increase in difficulty (e.g., increasing the number of enemies) can in turn decrease the overall performance of the game. Likewise, a decrease in difficulty can increase the performance of the game. We created goal models for each SAS agent to formalize their objectives [9, 33], derived utility functions to quantify the performance of each goal [10, 27, 34], and developed self-adaptive overlays for the game entities to use at run time, respectively. The results of our study suggest that deploying multiple MAPE-K loops can be used to significantly improve game performance while supporting the objectives of DPA and DDA. The remainder of this paper is structured as follows. Section 2 presents background information for our motivating example, goal modeling, utility functions, and SASs. Section 3 details

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3730555

our approach for incorporating multiple MAPE-K agents within a game environment. Section 4 presents the results of our empirical study. Section 5 highlights related work in DDA, DPA, SASs, and multi-agent systems. Lastly, Section 6 discusses our results and presents future avenues for research. Our source code and data have been published as open source on our GitHub repository.[1]

2 Background

This section presents our motivating example and discusses relevant background information on goal modeling and SASs.

2.1 Motivating Example

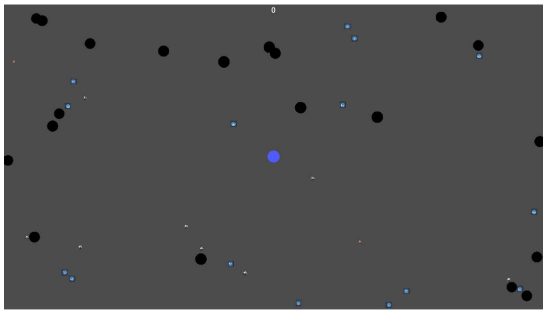
We developed a proof-of-concept game using Godot 4.3 and GDScript[2] to demonstrate our self-adaptive agents based on a prior browser-based iteration of our game [11]. Figure 1 shows a screenshot of our proof of concept game. The game concept is as follows: a player (blue circle) can eat enemies (dark circles) if they are smaller in size, thereby increasing the player's score and size, respectively. If a larger enemy touches the player, or if the player touches an environmental hazard, then the game is over. A player can collect powerups that allow the player to "eat" walls and become invincible for a short period of time.

In our game, enemies and the game engine each represent individual self-adaptive agents with their own governing feedback loops, where these loops enable self-adaptation (described next in Section 2.3) to reconfigure in response to needs for performance optimization (i.e., DPA) or player satisfaction (i.e., DDA). As such, DDA is an approach for automatically adjusting game difficulty based on player interactions to optimize towards player enjoyment and reduce frustrations [31, 37]. Within the context of our game we minimally implement DDA with the aim of optimizing satisfaction of each goal within our goal model (c.f., Section 2.2).

2.2 Goal Modeling

Goal modeling, or goal-oriented requirements engineering (GORE), provides a visual representation of key objectives and requirements, typically via an acyclic graph that includes goals, requirements and/or expectations, and agents [33]. Knowledge Acquisition in Automated Specification (KAOS) is one form of goal modeling that includes AND- and OR-refinements [9, 33], where an AND-refined goal must have all its child goals be satisfied and an OR-refined goal only requires that one child goal be satisfied. Figure 2 presents two goal models used in our application that demonstrate KAOS. For example, `Goal (E.a)` is only satisfied if both `(E.b)` and `(E.c)` are satisfied.

Figure 2 presents two separate KAOS goal models for individual agents in our game. The goal model on the left side specifies the objectives of an enemy within our game and the goal model on the right side specifies the game engine's objectives. Such a system can be considered a multi-agent system [14, 17], where agents in this application are autonomous yet not necessarily working towards the same overall goal. For example, the enemy's top-level goal (`(E.a)` *[Maintain] Playability*) aims to keep the player "happy"


(a) Zoomed-out view of game (no adaptation).

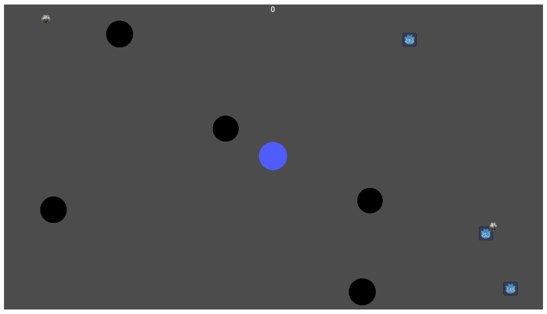

(b) Zoomed-in view of game (FPS optimization).

Figure 1: Screenshots of our proof-of-concept game. The blue circle in the middle represents the player, the dark circles represent enemies, and the other items represent walls (i.e., Godot logo), hazards (i.e., mines), and powerups (i.e., fire icons).

with their experience, yet the game engine's top-level goal (`(G.a)` *[Maintain] Game Active*) focuses on keeping the game active as long as possible. There is a dependency relationship between goals `(E.c)` and `(G.d)`, as both aim to provide the player with an adequate frame rate and use similar utility functions. Note that the player is not included as an agent within the goal models as their input has does not have a direct impact on the key objectives presented here.

Utility functions mathematically quantify the satisficement (i.e., degree of satisfaction, typically normalized within $[0.0, 1.0]$) of a requirement or goal [10, 27, 34]. For an SAS, a utility function can serve as a lightweight mechanism for the monitoring component to act as a software sensor. A sample utility function for `Goal (G.e)` is as follows in Equation 1, where the system designers selected a threshold of $FPS_{min} = 850$ to represent the minimum FPS allowed and a threshold of $FPS_{max} = 1400$ to denote an ideal maximum FPS. Additionally, threshold values of FPS_{t_1} and FPS_{t_2} were set to 1000 and 850, respectively, with corresponding utility value ranges of $(1.0, 0.73)$ and $(0.73, 0.0)$. Note that for presentation purposes we have simplified this goal, however its calculation additionally considers the current and prior zoom levels as well. These values were selected based on empirically-derived values in testing our game prototype and monitoring Godot's reported FPS values.[3]

[1]See https://github.com/StevenStreasick/Incorporating-Multiple-Self-Adaptive-Agents-in-Games.
[2]See https://godotengine.org/.

[3]"Typical" frame rates (e.g., 30 – 120) would result in minimal adaptations given our prototype's relative simplicity.

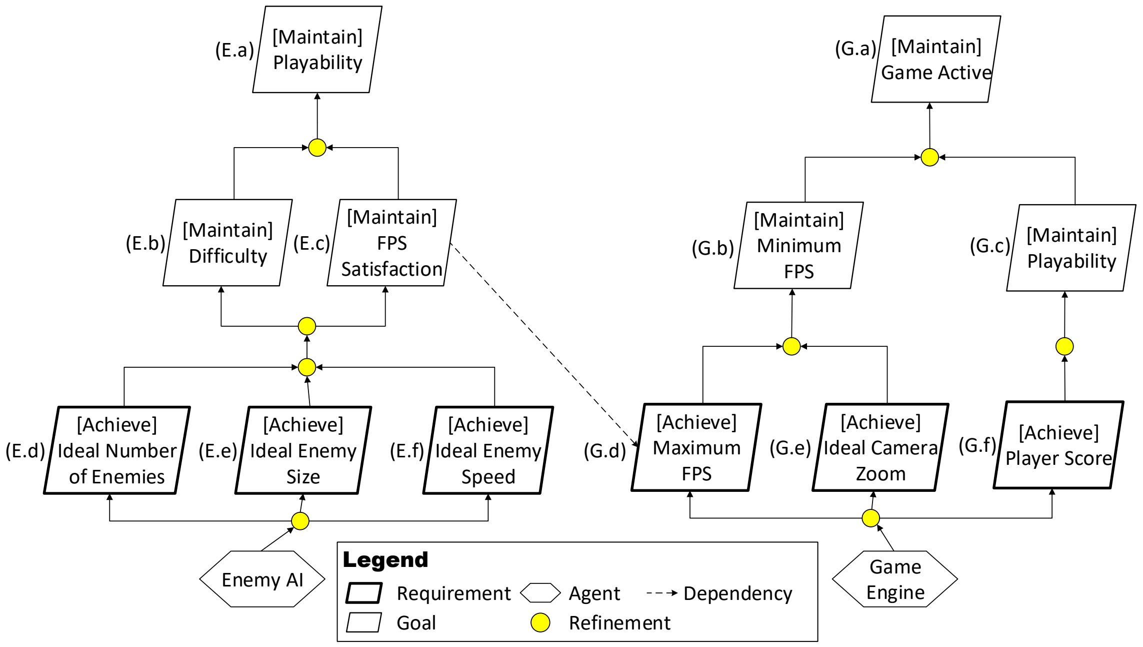

(E.a) [Maintain] Playability
(E.b) [Maintain] Difficulty
(E.c) [Maintain] FPS Satisfaction
(E.d) [Achieve] Ideal Number of Enemies
(E.e) [Achieve] Ideal Enemy Size
(E.f) [Achieve] Ideal Enemy Speed
Enemy AI
(G.a) [Maintain] Game Active
(G.b) [Maintain] Minimum FPS
(G.c) [Maintain] Playability
(G.d) [Achieve] Maximum FPS
(G.e) [Achieve] Ideal Camera Zoom
(G.f) [Achieve] Player Score
Game Engine
Legend
Requirement
Agent
Dependency
Goal
Refinement

Figure 2: Goal models describing key objectives for enemy agents and the game engine.

$$util_{G.e} = \begin{cases} 1.0 \ if \ FPS > FPS_{max} \\ (1.0, x, 0.73) \ if \ FPS_{max} < FPS < FPS_{t_1} \\ (0.73, x, 0.0) \ if \ FPS_{t_1} < FPS < FPS_{t_2} \\ 0.0 \ else \end{cases} \quad (1)$$

$util_{G.e}$ aims to continuously present an ideal zoom value that is defined as a result of the framerate, with a value of 1.0 indicating that the FPS exceeds the designer-specified threshold (i.e., 1400 FPS), a value of 0.0 indicating that a minimum threshold was violated (i.e., 850 FPS), and values in between are linearly scaled to represent the degree of satisficement for the goal.

2.3 Self-Adaptive Systems

SASs are applications that can self-reconfigure at run time in response to expected and/or unexpected issues that manifest as a result of uncertainty [15, 21]. Typically, an SAS is constructed with a feedback loop that enables adaptation with the aim of optimizing requirements satisfaction, where sample adaptations can include updating system configurations/parameters, changing algorithms, or updating system decision making [7, 28]. An SAS will perform these measures due to expressed uncertainty, where sources of uncertainty in this space can include unexpected player interactions, misconfigured system parameters, and differing hardware configurations [5, 6, 19, 26]. We implement the **M**onitor-**A**nalyze-**P**lan-**A**dapt-**K**nowledge (MAPE-K) loop [15], though there exist other forms of self-adaptive feedback loops [4]. With respect to our motivating example, our game will:

- *Monitor*: each agent within our system is responsible for monitoring the satisficement (i.e., the degree of satisfaction) for each goal within its respective goal model.
- *Analyze*: each agent monitors its utility values to determine if a reconfiguration is necessary; moreover, each agent has a defined set of reconfiguration strategies that can be triggered to improve itself or the system as a whole.
- *Plan*: each agent will determine which reconfiguration strategy is ideal based on the utility values indicating that an adaptation is necessary. For example, a violation of `Goal (E.d)` can result in a strategy to significantly increase the number of instantiated enemies on screen.
- *Execute*: the selected reconfiguration strategy is then either immediately implemented by its respective agent or scheduled, depending on its impact. For instance, if both `Goals (E.c)` and `(G.d)` were violated then only one zoom adaptation is necessary to cull drawn entities on screen. Ideally execution of a reconfiguration strategy is performed safely (i.e., without causing system degradation or violating system invariants).
- *Knowledge*: information regarding individual goal satisfaction (i.e, utility value calculation) and monitored system metrics are shared via a centralized entity within our game that is publicly accessible.

For example, one of our main system invariants (c.f., Figure 2, `Goal (G.b)`) is to *[Maintain] Minimum FPS*. For this case, a violation is considered to be a catastrophic failure and cannot be resolved via self-reconfiguration. However, one of the supporting non-invariants (`Goal (G.d)` - *[Achieve] Maximum FPS*) can temporarily tolerate transient failure and therefore is a target for adaptation. Moreover, satisfaction of this goal directly supports the satisfaction of its parent goal (`G.b`), though in combination with another goal (`Goal (G.e)` - *[Achieve] Ideal Camera Zoom*), to support this effort. In this case, if the utility function associated with `Goal (G.d)` is not satisfied then a reconfiguration strategy may be to "zoom in" on

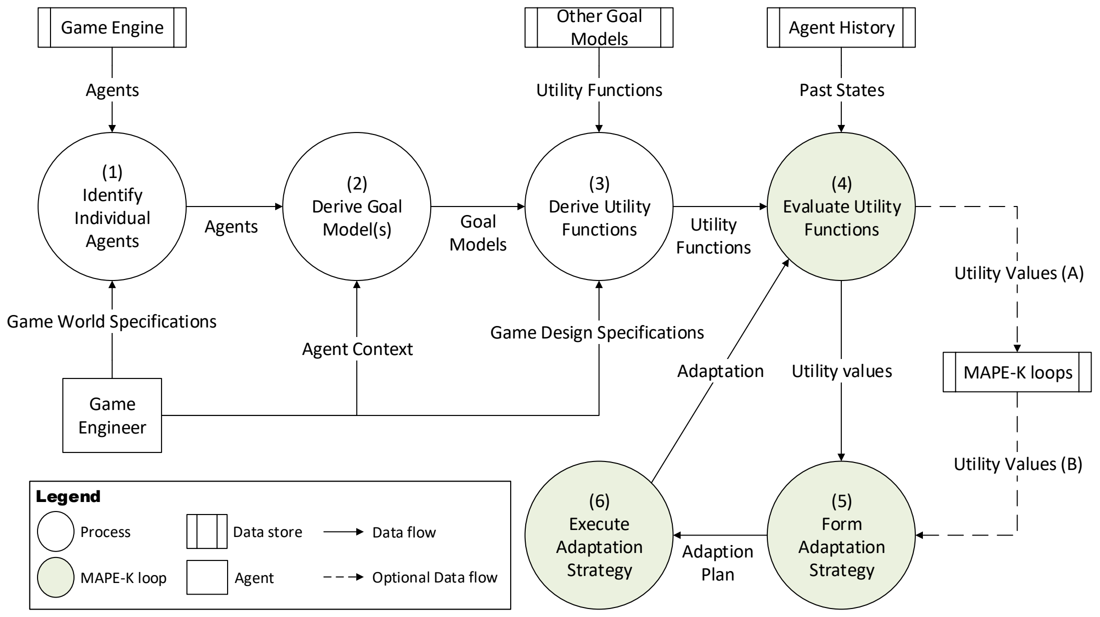


Figure 3: Data flow diagram describing our approach for incorporating multiple MAPE-K loops within our game prototype. Utility Values (A) and (B) denote two separate agents within our framework, though n sets of utility values may be defined as needed.

the player, thereby reducing the number of drawn entities on the screen and improving the framerate. In our approach, correlation of utility functions to adaptations is manually defined and more than one utility function can contribute to an adaptation decision.

3 Approach

This section describes our approach for incorporating multiple MAPE-K loops within a game environment. First, we describe the required inputs, expected outputs, and any assumptions necessary. Then, we describe our approach in detail.

3.1 Inputs, Outputs, and Assumptions

The system requires as input a set of goal models with respective utility functions for each agent and a game engine capable of supporting multiple adaptations. The goal model represents the functional and non-functional requirements of the SAS while the utility functions measure the agent's satisficement of each goal. Outputs include the logged utility values, system metrics, and number of adaptations performed over time. Assumptions include that the utility functions adequately quantify the performance of each goal, that goals in each goal model are well-aligned with an agent's key objectives, and that there exists no discrepancies in translating requirements to code. Additionally, we assume that the negative effects of including MAPE-K overlays into a game linearly impact performance metrics and can moreover be resolved by reconfiguration if necessary.

3.2 Adaptation Technique

This section details our technique for incorporating multiple self-adaptive agents within a game environment to optimize the game experience for the player. Figure 3 presents a data flow diagram that illustrates each step of the process.

(1) Identify Individual Agents: The agents that are chosen within this step are the entities responsible for the adaptations of the system. A key consideration when choosing agents are their relative independence within the framework, their ability to introspect on their behavior, and their support for self-reconfiguration at run time.

(2) Derive Goal Models: High-level objectives, including functional and non-functional system requirements, are formalized into goal model representations. The goal modeling syntax is a design decision, however for the purposes of this paper we follow the KAOS approach. Goal models may additionally include cross-cutting concerns that impact multiple agents (e.g., Figure 2, `Goals (E.c)` and `(G.d)`).

(3) Derive Utility Functions: Next, a utility function is derived to quantify the performance/behavior of each goal, where dependencies between models may use the same monitored values in support of their respective utility functions (e.g., *FPS* for `Goals (E.c)` and `(G.d)`). In this paper we normalize all utility functions to return a value on $[0.0, 1.0]$ to provide comparable metrics for analysis, where 0.0 indicates a violation, 1.0 indicates satisfaction, and $(0.0, 1.0)$ indicates a degree of satisficement.

(4) Evaluate Utility Functions: At run time, utility functions are evaluated as part of the *monitor* and *analysis* phases of the MAPE-K loop to determine if an adaptation is necessary. Depending on the goal, calculated utility values may be shared/reused if dependencies between goals exist. For example, Equation 1 measures the performance of `Goal G.e`. If the goal is violated (i.e., $FPS < FPS_{min}$, resulting in a value of 0.0), then a reconfiguration strategy will be necessary to improve zoom level and FPS to resolve the goal violation.

(5) Form Adaptation Strategies: If a reconfiguration strategy is determined to be necessary then the respective agent will select an appropriate adaptation. For example, the enemy agent can choose to reduce the number of enemy entities being instantiated or update the velocity and/or size of instantiated entities. The game engine can change the camera zoom level to reduce the number of entities drawn on screen. Following the example in (4), the camera will zoom in to reduce the number of drawn entities on the screen.[4]

(6) Execute Adaptation Strategies: Finally, the chosen adaptation strategy is executed and the MAPE-K loop continues. Adaptations are immediately executed in the context of our proof of concept game, however, depending on the application and the impact of the reconfiguration, system actions may be buffered or paused to minimize negative impacts.

4 Experimental Results

This section presents our results of incorporating multiple agents controlled by MAPE-K loops within a video game environment.

4.1 Experimental Configuration

For this study, our game was programmed within Godot 4.3 using GDScript. All recorded runs demonstrated "win" conditions where the player was alive for 45 seconds and was not consumed by a larger enemy. Metrics for only the last 30 seconds were captured to allow adequate time for the system to first stabilize. We used four separate hardware configurations and performed 10 experimental replicates per configuration.

- **Computer A** : Dell XPS 13 9310, Intel i7-1185G7 CPU, 16GB RAM, Intel Iris Xe, Windows 10 Pro
- **Computer B** : Desktop, Intel i5-9600K CPU, 16GB RAM, NVIDIA RTX 4060 Ti, Windows 10 Home
- **Computer C** : Dell Latitude 7400, Intel i7-8665U CPU, 32GB RAM, Intel integrated graphics, Pop_OS!
- **Computer D** : Lenovo Legion Pro 7, AMD Ryzen 9 7945HX, 32GB RAM, NVIDIA RTX 4080, EndeavorOS

4.2 Experimental Results

We recorded the number of adaptations that occurred within the models during our experiment. Figure 4 presents the total number of adaptations triggered as a result of goal violations, where the blue boxplots denote a zoom adaptation (i.e., zooming in to cull enemies drawn or zooming out to include more enemies on screen) and the orange boxplots denote an adaptation to the enemy agent (i.e., updating the number of enemies spawning). In our experiment, we found the runs produced by Computer D tended to have a higher framerate than our framerate maximum in both our camera zoom and FPS utility values, leading to no or few new adaptations being adopted by either model with the exception of a single run.

[4]As we used Godot for our game, off-screen entities are automatically culled and therefore FPS is improved.

The other configurations demonstrated more desirable behavior, with several adaptations being captured in each run in both the enemy and game engine goal models. Based on these trends, with the exception of Computer D, the enemy model's adaptations indicate that the model is achieving DPA. Our models also support adaptations in terms of enemy speed and size, however we leave formal studies of DDA in this system for future work.

We performed the Mann-Whitney U-test to determine statistical significance (i.e., $p < 0.05$) for each presented adaptation metric (i.e., `Zoom_Adaptations` between Computers A, B, C, and D, `Enemy_Adaptations` between Computers A, B, C, and D). There exists a significant difference between each computer configuration for both adaptation types, suggesting that the hardware configuration has a direct impact on the number of adaptations necessary for DPA-specific metrics.

We further examined the behavior of these adaptations by monitoring the utility values calculated for `Goal (E.d)` (i.e., *[Achieve] Ideal Number of Enemies*) and `Goal (G.e)` (i.e., *[Achieve] Ideal Camera Zoom*) over time. The utility function for `Goal (G.e)` was previously presented in Equation 1. Equation 2 is the utility function derived for `Goal (E.d)`, where the number of enemies that can spawn per second has a direct relationship with $util_{G.d}$ (i.e., maximize framerate) and x represents the current timestep. The utility value for $util_{G.d}$ is cubed in this function as a design choice with respect to observed game behavior. For presentation purposes we do not include the utility function for $util_{G.d}$.

$$util_{E.d_x} = \begin{cases} .95(util_{E.d_{x-1}}) + .05(util_{G.d})^3 \text{ if } smoothing \\ (util_{G.d})^3 \text{ if not } smoothing \end{cases} \tag{2}$$

Equation 2 has a direct relationship with its previous utility value if smoothing is enabled, where smoothing will determine if the value is directly set or gradually interpolated.

$$\begin{aligned} smoothing_x = {} & (util_{G.d} > .45) \text{ OR} \\ & (smoothing_{x-1} \text{ AND } util_{G.d} > .4) \end{aligned} \tag{3}$$

Equation 2 can be transformed into the number of enemies that are allowed to spawn using Equation 4. Here, we map the normalized utility value on $[0.0, 1.0]$ to the range of allowable enemies, where min and max are configured to be 0 and 2 enemies to spawn per second, respectively.

$$num\_enemies_x = (util_x)(|max - min|) + min \tag{4}$$

Figure 5 presents the utility values for `Goal (E.d)` (i.e., *[Achieve] Ideal Number of Enemies*, measured by $util_{E.d}$) and `Goal (G.e)` (i.e., *[Achieve] Ideal Camera Zoom*, measured by $util_{G.e}$) for a single experimental replicate. As can be seen from the figure, system agents begin to perform adaptations once the calculated utility values for each goal trend downward. While some adaptations immediately resolve the issue (i.e., enemy adaptation at timestep 35 for `Goal (E.e)`), others take longer to resolve the issue (i.e., both adaptations applied at timestep 30 for both goals).

In the latter case, multiple adaptations are required to improve utility values. Interestingly, there are also examples of adversarial adaptations in this figure. At approximately timestep 25 there is

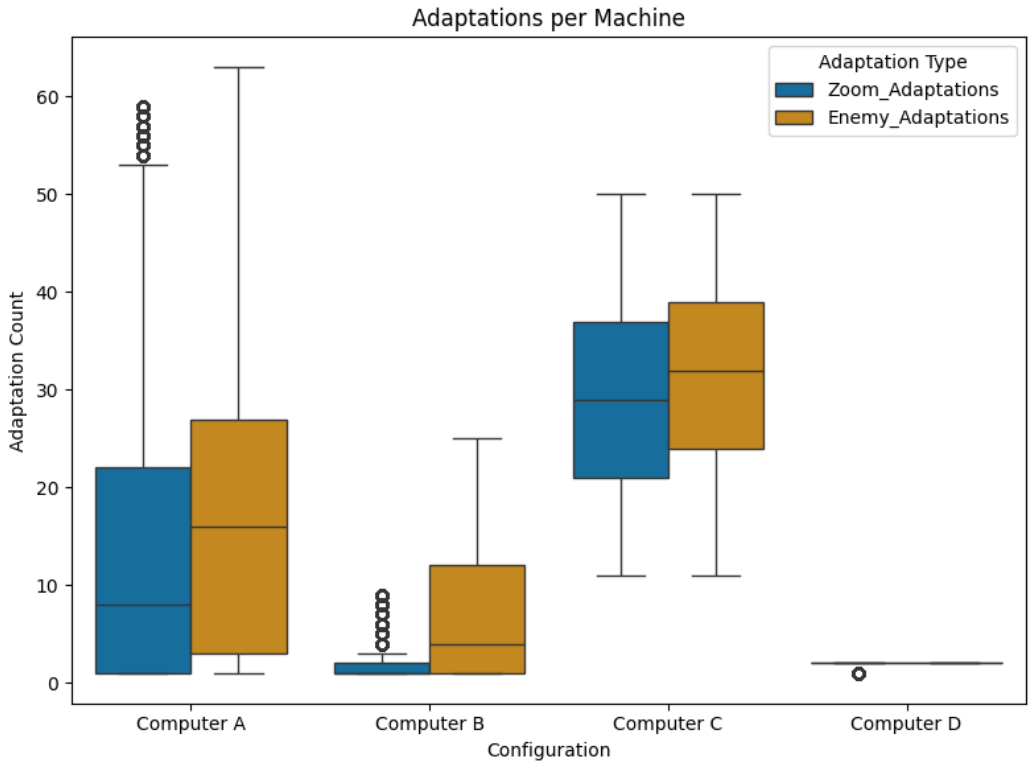


Figure 4: The number of adaptations between all runs for each configuration.

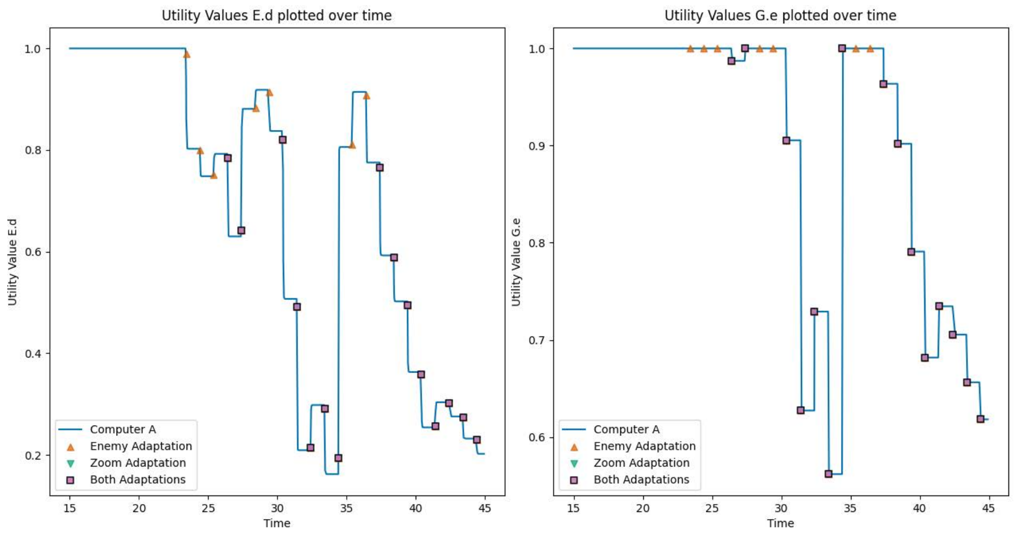


Figure 5: Utility values for Goals (E.d) and (G.e) with corresponding adaptations over time for a single experimental replicate with Computer A.

an enemy adaptation followed by both adaptations. $util_{E.d}$ immediately trends downwards, however $util_{G.e}$ trends upwards. Neither utility function presents a full violation, however, as our configured adaptations appear to be sufficient to maintain a level of goal satisficement.

The presented utility values are fairly chaotic and can lead to a player experience where the camera zooms in and out multiple times per-run to resolve FPS issues, leading to future work in "smoothing" the resolution of transient utility function violations/SAS reconfigurations to provide the player with a better visual experience. For presentation purposes we only show results from one run with Computer A, however these results are fairly similar across the replicates for configurations A–C. Interestingly, Computer D's calculated utility values over time for `Goal (E.d)` hold at 1.0 over the length of the game with a small number of minor adaptations occurring during the run. This result leads us to conclude that more study is necessary in generalizing our SAS implementation to a broad range of hardware configurations, where future work could consider automatically presenting a similar user experience across diverse hardware configurations. This further implies that the game may get more difficult the higher the framerate, though we leave a deeper study on DDA to our future work.

Threats to Validity: This study was a proof of concept to demonstrate the feasibility of incorporating multiple MAPE-K loops into a game environment. One threat to validity includes the small sample set of both users and device configurations used in the experiments, where future studies would include a large group of diverse participants focusing on engagement, flow state, etc. Another threat to validity lies in the reproducibility of results with respect to humans in the loop. While our game executions can be seeded for deterministic randomness, the behavior of humans cannot and again a larger test group of users is necessary for deeper studies. Our measurement of FPS may also be considered a threat to validity as we check the framerate each timestep using a built-in Godot function and do not perform any graphical or game-oriented optimizations in terms of CPU/GPU performance within our game. As such there may be processes happening within the Godot 4.3 game engine that we are not aware of to automatically optimize the application or calculate the framerate at different times. Another identified threat to validity includes the derivation and translation of goals, utility functions, and software artifacts within the Godot game engine as there may exist a misinterpretation between software design and implementation. Lastly, a threat to validity lies in the relative simplicity of the game we developed. A more complex game with multiple self-adaptive feedback loops may yield different results.

5 Related Work

This section presents related work on DDA, DPA, self-adaptation, and multi-agent systems in games.

5.1 Dynamic Difficulty and Performance Adjustment in Games

There exists a wide body of research in performing DDA to optimize the experience for the player in different facets [2, 23, 37]. Two of the main concerns with implementing DDA lie in measuring player satisfaction and adapting the game to maximize that satisfaction [29]. Zohaib presented a survey on implementing DDA within games, describing the flow channel and then presenting the state of the art with respect to several high-level classifications [37]. Hunicke presented an overview of requirements and metrics that can be used to support DDA activities [13]. Xue *et al.* present an approach for maximizing player engagement via DDA by using probabilistic graphs and modeling it as an optimization problem, demonstrating their technique's effectiveness on a globally-released mobile video game title [35]. There also exist techniques for using artificial intelligence [30], deep learning [24], and Bayesian modeling [12] in performing DDA in games. Most closely related, Souza *et al.* presented a self-adaptive approach for improving the player experience by maximizing their time within a monitored flow state, where difficulty comprises multiple "knobs" that can be adjusted by the game designer and autonomous adaptations can scale difficulty as needed [31]. While each of these works implement DDA in some fashion, our approach focuses on the interplay between performance optimization and player engagement via goal modeling and derived utility functions.

With respect to DPA there appears to be much less video game-focused research, however this may be explained by the fact that generic performance and application optimization can be considered to be applicable to a wider range of systems. Focusing on framerate, Sun and Wu explored methods for optimizing this metric in a cloud gaming environment where latency and geographic region are primary concerns [32]. Klein *et al.* investigated how frame timing can impact the perception of "smoothness" experienced in games that use a first-person perspective with a user-focused study that presented recommended graphics settings [16]. More broadly, DPA has been deployed to an Internet of Things application for optimizing message queuing concerns, thereby enhancing network performance [22]. Arcelli presented an approach that uses a multi-objective search heuristic to discover ideal self-adaptive architectures in an Internet of Things environment [1]. Our approach to DPA was to enable each separate MAPE-K agent in our system to influence performance with respect to the frame rate and number of spawned/drawn entities on screen.

5.2 Self-Adaptation in Games

While self-adaptation has traditionally been applied to safety-critical systems and networked architectures, there exists a body of work in leveraging the MAPE-K loop within gaming. The work previously described by Souza *et al.* incorporates a MAPE-K loop with the aim of maximizing player engagement by adapting difficulty to their needs and ensuring they stay within a flow state [31]. Yamagata *et al.* presented an empirical study in which a MAPE-K loop manages an online game with the aim of reducing network-oriented errors, where the components of the MAPE-K loop were distributed between the client(s) and server [36]. Our prior work [11] presented a proof of concept using a single MAPE-K as a mechanism for solely optimizing framerate. Our goal with this paper was to extend the concepts presented in each of these papers, however our focus was to incorporate multiple, distinct agents that aimed to optimize game performance as well as player engagement.

To that end, there also exists work in combining video games with multi-agent systems. Pons *et al.* presented work towards a multi-agent architecture as a model for learning and generating scenarios for the player to engage with [25]. Games have been conceptualized as an expression of a multi-agent system, where a game engine was developed to create games that exist as individual agents and are generated according to formal theorems [20]. Multi-agent systems have additionally been proposed as a mechanism for personalizing a (serious) game to player needs through an adaptive approach rooted in DDA [3]. While each of these techniques implements multiple agents, our approach was to enable each agent to have its own individual MAPE-K loop that can impact the system as a whole or the agent itself.

6 Discussion

This paper has presented a study on the feasibility of using multiple self-adaptive agents within a game environment. To demonstrate our approach we developed a proof-of-concept game where a player-controlled character was required to survive for a length of time by eating smaller enemies and avoiding larger enemies as adaptations were triggered as a result of monitored performance metrics. There were two self-adaptive feedback loops implemented: one controlled by the enemy entities and one controlled by the game engine. Both loops operated independently and could self-reconfigure as needed, based on governing goal models and derived utility functions. Experimental results suggest that multiple self-adaptive agents can positively impact a game experience for the player by improving the framerate and updating enemy characteristics at run time, thereby performing DPA and supporting DDA.

Future paths of research include extending the capabilities of our existing MAPE-K loops with deeper reconfiguration strategies, increasing the number of distinct agents within the game, designing and performing an empirical investigation into DDA and multiple self-adaptive agents, and performing a large user-focused study to focus on the impact of self-adaptation in games with respect to players of different skill levels with varying hardware configurations.

Acknowledgments

We gratefully acknowledge support from the Michigan Space Grant Consortium (award number 80NSSC20M0124) and Grand Valley State University.

References

[1] Davide Arcelli. 2020. A multi-objective performance optimization approach for self-adaptive architectures. In *European Conference on Software Architecture*. Springer, 139–147.

[2] Sander Bakkes, Chek Tien Tan, and Yusuf Pisan. 2012. Personalised gaming: a motivation and overview of literature. In *Proceedings of the 8th Australasian Conference on Interactive Entertainment: Playing the System*. 1–10.

[3] Spyridon Blatsios and Ioannis Refanidis. 2019. Towards an adaption and personalisation solution based on multi agent system applied on serious games. In *Artificial Intelligence Applications and Innovations: 15th IFIP WG 12.5 International Conference, AIAI 2019*. Springer, 584–594.

[4] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. 2009. Engineering self-adaptive systems through feedback loops. *Software engineering for self-adaptive systems* (2009), 48–70.

[5] Radu Calinescu, Raffaela Mirandola, Diego Perez-Palacin, and Danny Weyns. 2020. Understanding uncertainty in self-adaptive systems. In *2020 ieee international conference on autonomic computing and self-organizing systems (acsos)*. IEEE, 242–251.

[6] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. 2009. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems*. Springer-Verlag, Berlin, Heidelberg, 468–483.

[7] Lawrence Chung, B Nixon, E Yu, and J Mylopoulos. 2000. Non-functional Requirements. *Software Engineering* (2000).

[8] Mihaly Csikszentmihalyi. 2000. *Beyond boredom and anxiety.* Jossey-bass.

[9] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. 1993. Goal-directed requirements acquisition. *Science of computer programming* 20, 1 (1993), 3–50.

[10] Paul deGrandis and Giuseppe Valetto. 2009. Elicitation and Utilization of Application-level Utility Functions. In *Proc. of the 6th International Conference on Autonomic Computing* (Barcelona, Spain) *(ICAC '09)*. ACM, 107–116.

[11] Erik M. Fredericks, Byron DeVries, and Jared M. Moore. 2022. Towards self-adaptive game logic. In *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation (ICSE '22)*. ACM, 24–29. doi:10.1145/3524494.3527625

[12] Miguel González-Duque, Rasmus Berg Palm, and Sebastian Risi. 2021. Fast game content adaptation through Bayesian-based player modelling. In *2021 IEEE Conference on Games (CoG)*. IEEE, 01–08.

[13] Robin Hunicke. 2005. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*. 429–433.

[14] Nicholas R. Jennings. 1999. *Agent-Oriented Software Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–7. doi:10.1007/3-540-48437-X_1

[15] J.O. Kephart and D.M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (January 2003), 41 – 50.

[16] Devi Klein, Josef Spjut, Ben Boudaoud, and Joohwan Kim. 2024. Variable Frame Timing Affects Perception of Smoothness in First-Person Gaming. In *2024 IEEE Conference on Games (CoG)*. IEEE, 1–8.

[17] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. 2002. A Goal-Based Organizational Perspective on Multi-agent Architectures. In *Intelligent Agents VIII*. 128–140.

[18] Raph Koster. 2013. *Theory of fun for game design.* " O'Reilly Media, Inc.".

[19] Michael Austin Langford and Betty HC Cheng. 2019. Enhancing learning-enabled software systems to address environmental uncertainty. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 115–124.

[20] Carlos Marín-Lora, Miguel Chover, José M Sotoca, and Luis A García. 2020. A game engine to make games as multi-agent systems. *Advances in Engineering Software* 140 (2020), 102732.

[21] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B. H. C. Cheng. 2004. Composing adaptive software. *Computer* 37, 7 (July 2004), 56 – 64.

[22] Jie Meng, Xiaochao Wang, Jixin Hou, Zidong Wu, Wenbin Wang, Rui Zhang, and Zhu Qiao. 2022. A dynamic performance adjustment algorithm based on negative feedback mechanism of power internet of things. In *2022 IEEE 8th International Conference on Computer and Communications (ICCC)*. IEEE, 811–816.

[23] Fatemeh Mortazavi, Hadi Moradi, and Abdol-Hossein Vahabie. 2024. Dynamic difficulty adjustment approaches in video games: a systematic literature review. *Multimedia Tools and Applications* 83, 35 (2024), 83227–83274.

[24] Dvir Ben Or, Michael Kolomenkin, and Gil Shabat. 2021. Dl-dda-deep learning based dynamic difficulty adjustment with ux and gameplay constraints. In *2021 IEEE Conference on Games (CoG)*. IEEE, 1–7.

[25] Luc Pons, Carole Bernon, and Pierre Glize. 2012. Scenario control for (serious) games using self-organizing multi-agent systems. In *2012 IEEE International Conference on Complex Systems (ICCS)*. IEEE, 1–6.

[26] A.J. Ramirez, A.C. Jensen, B. H. C. Cheng, and D.B. Knoester. 2011. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 568 –571. (Preliminary work described in short paper)..

[27] Andres J. Ramirez and Betty H. C. Cheng. 2011. Automatically Deriving Utility Functions for Monitoring Software Requirements. In *Proceedings of the 2011 International Conference on Model Driven Engineering Languages and Systems Conference*. Wellington, New Zealand, 501–516.

[28] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. 2010. Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*. 95 –103.

[29] Gabriel K Sepulveda, Felipe Besoain, and Nicolas A Barriga. 2019. Exploring dynamic difficulty adjustment in videogames. In *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*. IEEE, 1–6.

[30] Mirna Paula Silva, Victor do Nascimento Silva, and Luiz Chaimowicz. 2015. Dynamic difficulty adjustment through an adaptive AI. In *2015 14th Brazilian symposium on computer games and digital entertainment (SBGames)*. IEEE, 173–182.

[31] Carlos Henrique R Souza, Saulo S de Oliveira, Luciana O Berretta, and Sergio T Carvalho. 2025. Extending a MAPE-K loop-based framework for Dynamic Difficulty Adjustment in single-player games. *Entertainment Computing* 52 (2025).

[32] Kairan Sun and Dapeng Wu. 2015. Video rate control strategies for cloud gaming. *Journal of Visual Communication and Image Representation* 30 (2015), 234–241.

[33] Axel van Lamsweerde. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley.

[34] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. 2004. Utility functions in autonomic systems. In *Proceedings of the First IEEE International Conference on Autonomic Computing*. IEEE Computer Society, 70–77.

[35] Su Xue, Meng Wu, John Kolen, Navid Aghdaie, and Kazi A Zaman. 2017. Dynamic difficulty adjustment for maximized engagement in digital games. In *Proceedings of the 26th international conference on world wide web companion*. 465–471.

[36] Satoru Yamagata, Hiroyuki Nakagawa, Yuichi Sei, Yasuyuki Tahara, and Akihiko Ohsuga. 2019. Self-Adaptation for Heterogeneous Client-Server Online Games. In *International Conference on Intelligence Science*. Springer, 65–79.

[37] Mohammad Zohaib. [n. d.]. Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review. *Advances in Human-Computer Interaction* 2018, 1 ([n. d.]). doi:10.1155/2018/5681652