Towards Self-Adaptive Game Logic

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

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

Jared M. Moore
Grand Valley State University
Allendale, Michigan, USA
moorejar@gvsu.edu

ABSTRACT

Self-adaptive systems (SAS) can reconfigure at run time in response to changing situations to express acceptable behaviors in the face of uncertainty. With respect to game design, such situations may include user input, emergent behaviors, performance concerns, and combinations thereof. Typically an SAS is modeled as a feedback loop that functions within an existing system, with operations including monitoring, analyzing, planning, and executing (i.e., MAPE-K) to enable online reconfiguration. This paper presents a conceptual approach for extending software engineering artifacts to be self-adaptive within the context of game design. We have modified a game developed for creative coding education to include a MAPE-K self-adaptive feedback loop, comprising run-time adaptation capabilities and the software artifacts required to support adaptation.

CCS CONCEPTS

• **Software and its engineering** → **Requirements analysis**; • **Applied computing** → Interactive learning environments;

KEYWORDS

self-adaptive systems, software engineering, creative coding, game design

1 INTRODUCTION

Games are generally structured towards being reactive to the player to provide an immersive experience. For example, difficulty may scale as the player's character (PC) grows in strength or the environment and/or story may change as the PC makes in-game decisions. Such updates may be considered to be adaptations to enrich gameplay. In the case of an arcade-style game (i.e., where high score is the goal), such changes include reactions to the player's skill level, managing performance concerns, and optimizing overall game flow. This paper introduces a technique for incorporating software artifacts (e.g., requirements, goals, etc.) as first-class citizens in an adaptive gameplay loop. We apply this technique to our existing open-source game project to demonstrate its effectiveness.

Self-adaptive systems (SAS) are an approach for enabling reconfiguration at run time in systems facing uncertainty [11, 12], where the MAPE-K feedback loop [11] is one common technique for enabling adaptation. For instance, a system may change its configuration, algorithm, or decision-making engine as adversity manifests from combinations of system or environmental uncertainty (e.g., human interaction, unexpected external conditions, etc.). In terms of games, such configurations may include the number and intelligence of enemy characters, distribution of item pickups in relation to player level, and maintaining an appropriate frame rate as the number of instantiated objects increase. Moreover, software engineering techniques such as goal modeling [22] and run-time requirements monitoring [9] can enable introspection that supports self-reconfiguration strategies [3], however these self-reconfiguration techniques have not been intrinsically applied to a game loop.

This paper describes our application of a MAPE-K feedback loop to a game designed as a creative coding learning experience for students. We generated the necessary software artifacts (i.e., a goal model [21] with utility functions [9]), updated existing source code with a self-adaptive overlay, and added instrumentation for monitoring the performance of the overall system with respect to our defined metrics.

Our initial results demonstrate the feasibility of applying self-adaptation to a gameplay loop. The remainder of this paper is structured as follows. Section 2 discusses relevant background information on creative coding (focusing on our motivating example), SASs, and goal modeling. Section 3 then discusses our approach for modeling a game and its software artifacts as a self-adaptive feedback loop and presents our initial results. Lastly, Section 4 summarizes our results and presents future directions.

2 BACKGROUND AND RELATED WORK

This section describes our motivating example, SASs, and goal modeling. For each sub-section we also highlight related work.

2.1 Creative Coding (Motivating Example)

Creative coding is a form of programming in which the goal is generally considered to be some form of generative artwork or video game, typically created via abstractions over existing programming languages [10]. Processing (Java) and p5.js (JavaScript)[1] are two such abstractions that facilitate rapid development of creative coding applications. As such, creative coding can additionally be used to introduce students to high-level programming and game design concepts while minimizing the overhead and/or learning curve of graphics libraries (e.g., OpenGL, WebGL, SDL, etc.). Creative coding, as applied to this project, was intended to serve as a demonstration for undergraduates on game design concepts. We next describe `feesh`, our motivating example.

Motivating example - `feesh`: `feesh` is a relatively simple web-based game where the intention is to grow by eating smaller objects. This project was written in p5.js and can run in any browser without the need for a server. Additionally, the source code is available on GitHub.[2]

The gameplay loop for feesh is based on Fishy, a classic Flash game where the intent is to avoid large fish and eat smaller fish.[3] The intent of `feesh` was to replicate the core mechanics of Fishy

[1]See https://processing.org/ and https://p5js.org/, respectively.
[2]https://github.com/efredericks/feesh.
[3]See https://www.silvergames.com/en/fishy.

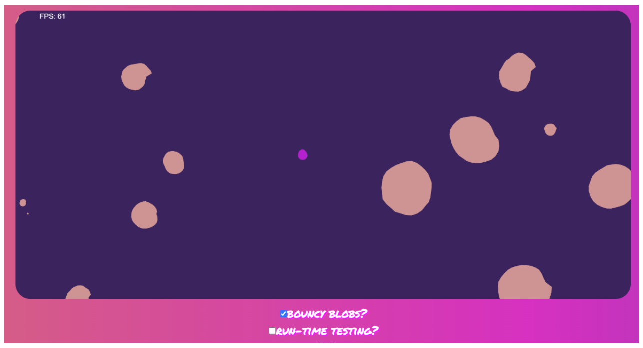


Figure 1: Screenshot of feesh gameplay. The pink blob is the player and the tan blobs are enemies of varying size. The checkboxes on the bottom can change game parameters at run time.

while introducing self-adaptation into the core gameplay loop. For additional processing complexity, all objects are formed via noise loops to give a "wobbly" impression, based upon the *TheCodingTrain*'s Blobby! video [18]. Collision detection is simplified to be circle-circle collision between all entities.

Creative coding is typically structured around introducing students to computer science concepts via generative art or simplified game design as a way for students to graphically visualize difficult problems [2, 15], with popular textbooks including *The Nature of Code* [19] and *Generative Art: A Practical Guide using Processing* [14]. Recently, creative coding has been applied to Internet of Things applications to further express art in real-world settings [23]. However, many papers centered around creative coding focus on teaching coding concepts, whereas we posit learning can be extended to additionally comprise complex software engineering concepts (where this argument will be explored in future works). Similarly, Bucchiarone *et al.* discuss gamification as a means for enhancing user engagement [4, 5], however for the purposes of this paper we aim to enhance gameplay via self-adaptation in contrast to adding gameplay elements an existing application.

2.2 Self-Adaptive Systems

SASs provide an approach for self-reconfiguring (e.g., configuration, algorithm, etc.) at run time [12, 13] in response to changing environmental and system conditions to continuously satisfy key objectives [7, 25]. The requirements for the system itself may also change over time, potentially necessitating updates in terms of patches, bug fixes, or configuration change. For the purposes of this paper we consider an SAS to comprise a set of configurable states connected via adaptive logic [28]. While there exist multiple approaches for enabling self-adaptation, we will follow the MAPE-K (Monitor-Analyze-Plan-Execute-Knowledge) feedback architecture [11]. We next describe our motivating example in the context of MAPE-K to highlight each aspect of its architecture.

Monitor: An SAS must monitor itself and its environment for decision-making purposes. We specified key metrics as relevant to the adaptive properties of our game engine, with *internal software monitors* being responsible for watching the state of each metric. Specifically, we manually identified *which* aspects of our game engine were amenable to adaptation (e.g., player size, enemy count) and which metrics *might* impact those aspects (e.g., FPS, execution time) A subset of these metrics, along with thresholds and reconfiguration strategies, are listed as follows in Table 1. Note, all values found in Table 1 were empirically derived.

Note that, for the purposes of this paper we are mainly focusing on features that directly impact the player during gameplay, however for future work we could examine past-game history (e.g., player behaviors, enemy encounters, etc.) to determine if adaptations are warranted.

Analyze: Each monitored attribute is then analyzed to determine if a violation has occurred or a threshold has been exceeded, thus necessitating a reconfiguration.[4] For this project, we defined thresholds for each monitored attribute to determine if a reconfiguration is warranted. For example, if the frame rate (i.e., frames per second, or `FPS`) falls below 30 then an adaptation is necessary to ensure the player has a smooth experience.

Plan: Based on identified adaptation requirements the *Plan* phase will select a reconfiguration strategy. Depending on the complexity of the system and feedback loop, this phase may involve the generation of many possible adaptations. For example, when the `FPS` drops below 30 the system will have the choice to remove the enemy-enemy collision (i.e., calculating collisions between enemies to induce "bouncing") or to remove a random number of enemies to reduce overall processing load.

Execute: The system will then execute the adaptation strategy based on monitored data and the decisions made in the *Analysis* and *Plan* phases. In the case of `feesh`, self-adaptation can comprise changes in complexity (e.g., increase the number of enemies given acceptable monitored performance), reduce the size of entities (e.g., scale down player size to improve playability), or change collision detection between enemies (e.g., removing enemy-enemy collision to improve performance). The Execute phase then directly applies the reconfiguration as needed.

Knowledge: For simplicity, *knowledge* of all aspects of the game and its self-adaptation mechanics are globally-accessible to all components of the MAPE-K loop.

Uncertainty: For the purposes of this paper, we will consider uncertainty to reflect human interaction (i.e., the player), the capabilities of the player's computer (e.g., browser choice, hardware specifications, frame rate, etc.), and random behaviors implemented within the `feesh` game engine. The combination of these types of uncertainty can result in different gameplay experiences, and as such, `feesh` has been modeled as self-adaptive.

Previously, the MAPE-K loop has been applied to a gaming application by Yamagata *et al.* [26], however their application focused mainly on the latency issue in multiplayer games, whereas our application focuses on including adaptation as part of the intrinsic game mechanics. Cámara *et al.* investigated how an SAS can optimize network latency via gamification and by modeling it as a stochastic multiplayer game [6]. Other methods of adaptation, among many, include the use of dynamic software product lines to

[4]In other systems, continuous learning may be implemented to make predictions, however self-reconfiguration in `feesh` is reactive.

Table 1: Monitored Metrics and Thresholds (N/A reconfigurations are only monitors).

Monitor	Threshold	Reconfiguration Strategy	Affected Goal(s)
Frame rate (FPS)	>=30	[Reduce enemy count, Disable enemy-enemy collision]	(B), (D), (E)
Playability	Maximize	[Reduce player size, Reduce enemy count]	(E), (F)
Score	Maximize	N/A	(H)
Number of enemies on screen	Maximize	N/A	(B), (E)
Enemy-enemy collision enabled?	T|F	N/A	(B), (D)
Execution time	Maximize	[Reduce player size, Reduce enemy count]	(A), (B), (C), (D), (E), (F)
Random event	2% chance	[Increase enemy count]	(A), (C), (E)

update behaviors at run time [1], Bayesian optimization with fuzzy systems [16], and artificial intelligence [20].

2.3 Goal Modeling

Goal-oriented requirements engineering (GORE) uses goals (i.e., high-level objectives and/or requirements) to model system behaviors and objectives [21]. GORE models a system-to-be via an acyclic graph comprising goals, refinements, requirements/expectations (i.e., leaf-level goals), and agents. Knowledge Acquisition in Automated Specification (KAOS) extends GORE via additional AND- and OR-refinements [8, 21] (e.g., Goals (A) and (B) in Figure 2, respectively).. AND-refined goals are satisfied only when *all* subgoals are satisfied and OR-refined goals are satisfied when at minimum *one* of its subgoals are satisfied. Figure 2 presents a KAOS [21] goal model for `feesh`, where KAOS is one approach to goal modeling.[5]

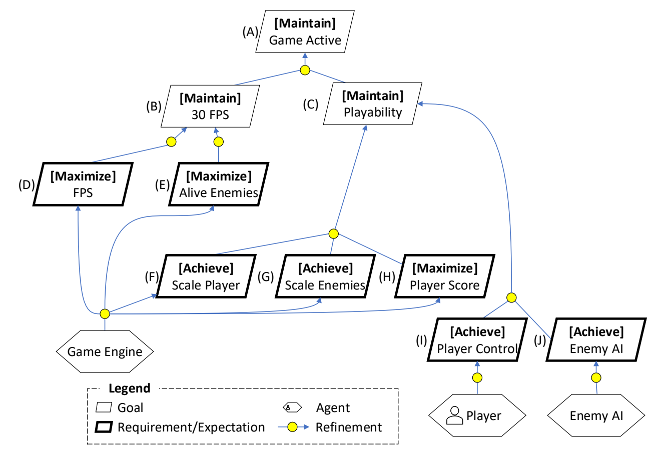

(A) [Maintain] Game Active
(B) [Maintain] 30 FPS
(C) [Maintain] Playability
(D) [Maximize] FPS
(E) [Maximize] Alive Enemies
(F) [Achieve] Scale Player
(G) [Achieve] Scale Enemies
(H) [Maximize] Player Score
(I) [Achieve] Player Control
(J) [Achieve] Enemy AI
Game Engine
Player
Enemy AI
Legend
Goal
Requirement/Expectation
Agent
Refinement

Figure 2: Initial goal model of `feesh` application.

The main goal of `feesh` is to extend time played (similar to an arcade game) (Goal (A)). To satisfy this goal, the game must be at an acceptable frame rate (Goal (B)) while ensuring it remains playable (Goal (C)). Goals (D) and (E) attempt to balance FPS by managing collision detection and the number of enemies on screen, respectively. Goals (F), (G), and (H) support Goal (C) by keeping the player at a manageable size while ensuring the score continually increases (to maximize player satisfaction). Moreover, Goals (I) and (J) additionally support (C) by enabling player and enemy control, respectively.

Goal (A) represents an AND-satisfied goal (i.e., Goals (B) and (C) must be satisfied for (A) to be satisfied) and Goal (B) represents an OR-satisfied goal (i.e., either Goals (D) or (E) must be satisfied for (B) to be satisfied). Moreover, goals designated as `Maintain` are considered to be *invariant* and those designated as `Achieve` are considered as *non-invariant*, where `invariant` goals cannot tolerate a violation and `non-invariant` goals can temporarily accept a transient violation.

Note that maximizing the number of enemies on screen while setting enemy-enemy collision to be enabled are competing concerns, as the collision detection algorithm implemented may significantly reduce frame rate. We would like to note that more "intelligent" collision detection algorithms can be implemented to minimize this issue (e.g., executed in parallel on a GPU, using local search/partitioning, etc.), however for the purposes of this paper a simpler implementation was preferred to demonstrate the issue.

We next describe utility functions, the approach we use for determining goal satisfaction at run time.

Utility Functions: Utility functions are mathematical formulae that have been used to quantify software requirement satisfaction [9, 17, 24]. In the context of an SAS, a utility function can serve as a metric for deciding if a reconfiguration is necessary (e.g., in the event that a goal is violated or under-performing). For example, a utility function can be derived for Goal (D) in Figure 2 as follows in Equation 1:

$$util_B = \begin{cases} 1.0 & \text{if } FPS \geq 40 \\ f(x) & \text{else if } FPS \geq 30 \\ 0.0 & \text{else} \end{cases} \tag{1}$$

, where a value of 0.0 indicates a violation, 1.0 indicates satisfaction, and any value in between denotes the degree of *satisficement*. For this project, each utility function yields a value normalized on [0.0, 1.0] and additionally implements a threshold, where any value below that threshold is considered inadequate and requires adaptation to resolve. In some cases (e.g., Goal (A), (G), (I), and (J)), the utility function is trivially true in that it always is 1.0. Values such as these are generally not useful as points of adaptation given that there is no change, whereas Goal (D) will change drastically over the course of the game execution.

[5]A similar approach for goal modeling is iStar [27].

3 APPROACH

This section describes how self-adaptation can be applied to a gameplay loop via our motivating example and further presents initial experimental results gathered during gameplay by the authors.

3.1 Instrumentation

We first developed a goal model of `feesh` as seen in Figure 2. For each goal we developed a utility function (e.g., Equations 1 and 2 for examples) to measure the satisfaction of each goal at run time. Additionally, we introduced adaptation mechanisms into `feesh` comprising software sensors (Monitoring) and logic for enabling self-reconfiguration (Analysis, Planning, Execution, Knowledge). Table 1 provides the adaptations currently available in `feesh`, however we anticipate that additional adaptation mechanisms can be introduced in future work. Each of these activities requires domain knowledge of the developer to (1) understand the underlying game logic/engine and know when/where software can be augmented with self-adaptive characteristics and (2) be able to derive new software artifacts and/or leverage existing software artifacts for use as first-class citizens at run time.

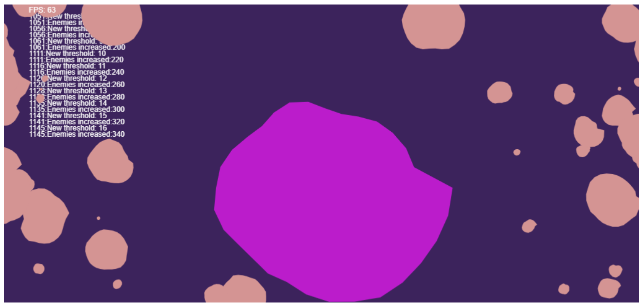
3.2 Adaptive Gameplay

We now present a sample use case for the MAPE-K loop within the context of the game. For example, consider Goal (C) in Figure 2. The utility function for this goal can be inserted into the main gameplay loop as part of the *Monitoring* phase. In the event that this goal is considered violated (e.g., $playerSize > width/2$) then the adaptation mechanism will determine which reconfiguration is necessary for Goal (C) to be satisfied again (note: such goals are considered *non-invariant* in that they can tolerate temporary failure; an *invariant* goal cannot tolerate failure and as such the program would need to go to a safety state - e.g., shutdown or forcing user input). Depending on the complexity of the Analysis and Planning phases, the SAS may generate one or many possible reconfiguration strategies and select the most appropriate course of action. In the case of Goal (C), the system reduces *playerSize* by 50% (where this value was determined empirically to extend gameplay). Moreover, if the *playerSize* exceeds the width of the canvas (i.e., nothing else is visible other than the player) then the game is considered to be *won* and the player is returned to the main menu.[6] Figures 3a and 3b illustrate this adaptation in-game.

3.3 Initial Experimental Results

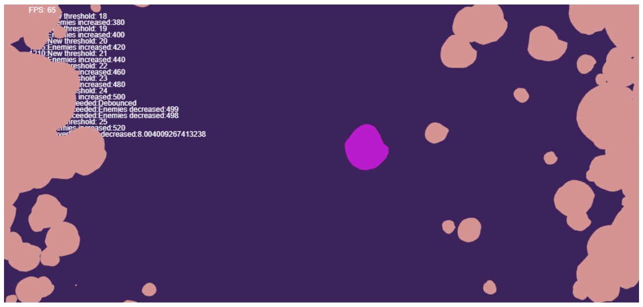
This section presents initial results on applying the MAPE-K feedback loop to `feesh`. For the purposes of this experiment, all utility values are normalized between $[0.0, 1.0]$ and execution time relies on the `frameCount` variable provided by p5.js (i.e., integer ticks since program start). `feesh` was run on a GitHub.io-hosted page (https://efredericks.github.io/feesh/) and accessed via Google Chrome (Version 97.0.4692.71 (Official Build) (64-bit)) on a modern laptop. We performed two experimental treatments: MAPE-K enabled and MAPE-K disabled (i.e., *Normal*). For each treatment we performed 50 replicates to establish statistical significance and we use the Wilcoxon-Mann-Whitney u-test with a significance (i.e.,

[6]With MAPE-K active, the *won* state is not attainable unless if the player gains mass quicker than can be processed.



```
FPS: 63
1051:New thres
1051:Enemies in
1056:New thresh
1056:Enemies incr
1061:New threshold:
1061:Enemies increased:200
1111:New threshold: 10
1111:Enemies increased:220
1116:New threshold: 11
1116:Enemies increased:240
112 New threshold: 12
1120:Enemies increased:260
1128:New threshold: 13
11 :Enemies increased:280
11 :New threshold: 14
1135:Enemies increased:300
1141:New threshold: 15
1141:Enemies increased:320
1145:New threshold: 16
1145:Enemies increased:340
```

(a) Sample of large player character that impedes on Goal (C).



```
FPS: 65
  threshold: 18
  emies increased:380
  threshold: 19
  E  ies increased:400
  New threshold: 20
  :Enemies increased:420
1210:New threshold: 21
  Enemies increased:440
  reshold: 22
  increased:460
  shold: 23
  ncreased:480
  shold: 24
  increased:500
  eeded:Debounced
  ceeded:Enemies decreased:499
  ceeded:Enemies decreased:498
  hreshold: 25
  emies increased:520
  ver  decreased:8.004009267413238
```

(b) Player size reduced as result of MAPE-K adaptation.

Figure 3: Reconfiguration resulting from Goal (C) violation.

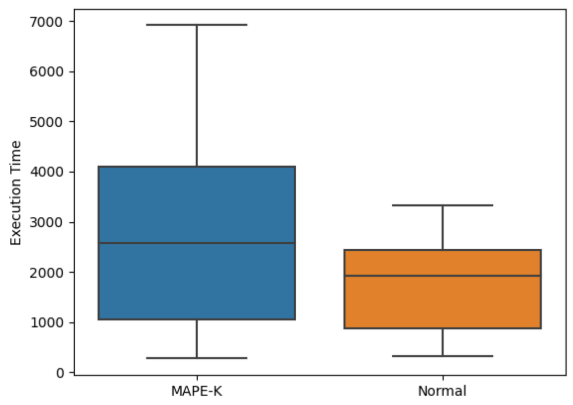
p-value) threshold of 0.05. The results were gathered by the authors of this paper. Additionally, we focus on Goal (F) to demonstrate feasibility of this approach, where we will explore additional metrics in future work.

Figure 4a presents boxplots of the overall execution time (i.e., `ticks`) for each treatment. As can be seen by these plots, players are engaged significantly longer by the version with MAPE-K enabled ($p < 0.05$), suggesting that the system is reconfiguring itself in support of its goals. Figure 4b presents the average utility values for Goal (F), where its utility equation is defined as:

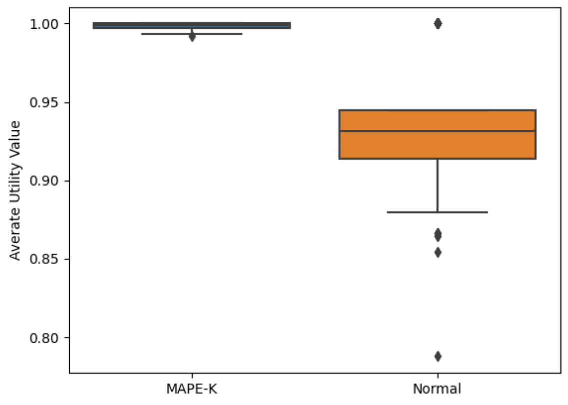
$$util_F = \begin{cases} 1.0 & \text{if } ps \leq w/2 \\ 0.0 & \text{else if } ps \geq w \\ 1.0 - abs(ps - (w/2))/(w/2) & \text{else} \end{cases} \quad (2)$$

, where ps is short for *playerSize* and w is short for the canvas width. As can be seen in Figure 4b, the average utility values for Goal (F) are significantly higher than those without MAPE-K ($p < 0.05$), suggesting that adaptations can continuously improve the satisficement of Goal (F). For the *Normal* system, the utility values remain high until the player wins the game (i.e., $playerSize \geq width$).

Initially we had anticipated that the FPS would be the most significant hindrance to game execution, given the additional overhead of the p5.js library with respect to graphics processing, the large amount of entities that can spawn, and lack of interaction with a graphics processor (i.e., shaders are not compiled for this application). The game ran at approximately 60 FPS with or without



(a) Total execution time between experimental treatments (seconds).



(b) Average utility values for Goal (F).

Figure 4: Feasibility experiments.

MAPE-K enabled (upon visual inspection), where the main limiting factor for goal satisfaction was *playerSize*. However, we did note that being overzealous with adaptations can in fact induce a performance decrease. For example, one reconfiguration strategy was in support of Goal (E) (i.e., *Maximize Alive Enemies*). A loop was added to scale the number of active enemies as appropriate to the player's size and score but as a result the FPS significantly dropped and would cause violations to Goal (B) (i.e., [Maintain] 30 FPS), resulting in a failed execution as Goal (B) is invariant.

4 DISCUSSION

This paper has presented our approach for incorporating the MAPE-K feedback loop into the core gameplay loop of a single-player game. We developed a goal model and associated utility functions for a browser-based game centered around creative coding concepts. We additionally deployed a MAPE-K feedback loop to enable self-adaptation at run time in response to player actions and game mechanics. Initial results indicate that applying a self-adaptive overlay to a gameplay loop can enable adaptation at run time as a result of software engineering artifacts (i.e., by elevating artifacts to be first-class citizens in the core loop).

Threats to Validity: This paper is a proof of concept to demonstrate the feasibility of run-time adaptation in the context of gaming. As such, we have identified the following threats to validity. For *internal* validity, the derivation of all artifacts and code were performed by the authors and may be subject to missing goals or unintentional bugs. Additionally, p5.js is not optimized for "fast" graphical applications and therefore suffers from bottlenecks when drawing to the canvas, where such issues are solved with more direct access (e.g., via pure JavaScript or WebGL shaders). However, the purpose of this example is for teaching and demonstration purposes and is therefore coded in an environment that is easily approachable. For *external* validity, the configuration of the player's computer may have an impact (hardware and software combined) on the game experience, as well as the player's comfort level with gaming in general. For *construct* validity, scalability and generalizability are possible threats. Specifically, we applied MAPE-K to a single application that is relatively simple in mechanics and limited in the number of derived goals and adaptations, however we envision that application to other types of games (both in nature and complexity) should be feasible.

Future paths of research for this project include human-focused studies to determine the effectiveness of self-adaptation in games (including player skill as an adaptation metric), incorporation of run-time search techniques for optimization and testing, and expansion of the `feesh` game engine for both teaching and research.

ACKNOWLEDGMENTS

This work has been supported in part by grants from the Michigan Space Grant Consortium (#80NSSC20M0124) and Grand Valley State University. The views and conclusions contained herein are those of the authors and do not necessarily represent the opinions of the sponsors.

REFERENCES

[1] Inmaculada Ayala, Alessandro V Papadopoulos, Mercedes Amor, and Lidia Fuentes. 2021. ProDSPL: Proactive self-adaptation based on Dynamic Software Product Lines. *Journal of Systems and Software* 175 (2021), 110909.
[2] Ilias Bergstrom and R Beau Lotto. 2015. Code Bending: A new creative coding practice. *Leonardo* 48, 1 (2015), 25–31.
[3] Kate M Bowers, Erik M Fredericks, Reihaneh H Hariri, and Betty HC Cheng. 2020. Providentia: Using search-based heuristics to optimize satisficement and competing concerns between functional and non-functional objectives in self-adaptive systems. *Journal of Systems and Software* 162 (2020), 50.
[4] Antonio Bucchiarone, Antonio Cicchetti, and Annapaola Marconi. 2019. Exploiting multi-level modelling for designing and deploying gameful systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 34–44.
[5] Antonio Bucchiarone, Tommaso Martorella, Diego Colombo, Antonio Cicchetti, and Annapaola Marconi. 2021. POLYGLOT for Gamified Education: Mixing Modelling and Programming Exercises. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 605–609.
[6] Javier Cámara, Gabriel A Moreno, David Garlan, and Bradley Schmerl. 2016. Analyzing latency-aware self-adaptation using stochastic games and simulations. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 10, 4 (2016), 1–28.
[7] 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.

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

[9] 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.

[10] Ira Greenberg. 2007. *Processing: creative coding and computational art.* Apress.

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

[12] 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.

[13] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. 1999. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE* 14, 3 (1999), 54 –62.

[14] Matt Pearson. 2011. *Generative art: a practical guide using processing.* Simon and Schuster.

[15] K Peppler and Y Kafai. 2005. Creative coding: Programming for personal expression. 30, 2008 (2005), 314.

[16] Michele Pirovano, Renato Mainetti, Gabriel Baud-Bovy, Pier Luca Lanzi, and Nunzio Alberto Borghese. 2012. Self-adaptive games for rehabilitation at home. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 179–186.

[17] 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.

[18] Daniel Shiffman. [n. d.]. Blobby! Coding Challenge 36. https://thecodingtrain.com/CodingChallenges/036-blobby.html

[19] Daniel Shiffman, Shannon Fry, and Zannah Marsh. 2012. *The nature of code*. D. Shiffman.

[20] Maciej Świechowski and Jacek Mańdziuk. 2013. Self-adaptation of playing strategies in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (2013), 367–381.

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

[22] A. van Lamsweerde and E. Letier. 1998. Integrating obstacles in goal-driven requirements engineering. In *Proceedings of the 1998 International Conference on Software Engineering*. 53–62.

[23] Lasse Steenbock Vestergaard, João Fernandes, and Mirko Presser. 2017. Creative coding within the Internet of Things. In *2017 Global Internet of Things Summit (GIoTS)*. IEEE, 1–6.

[24] 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.

[25] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.M. Bruel. 2009. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *17th IEEE International Requirements Engineering Conference (RE'09)*. 79–88.

[26] 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.

[27] E. S. K. Yu. 1997. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*. 226–235.

[28] Ji Zhang and Betty H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*. ACM, 371–380.