Chapter 6
What Else Is in an Evolved Name? Exploring Evolvable Specificity with SignalGP

Alexander Lalejini and Charles Ofria

6.1 Introduction

In Genetic Programming Theory and Practice IX, [20] explored the use of tag-based naming in evolving modular programs. In this chapter, we continue exploring tag-based naming with SignalGP [14]; we investigate the importance of inexactness when making tag-based references: How important is imprecision when calling an evolvable name? Additionally, we discuss possible broadened applications of tag-based naming in the context of SignalGP.

What's in an evolved name? How should modules (e.g., functions, sub-routines, data-objects, etc.) be referenced in evolving programs? In traditional software development, the programmer hand-labels modules and subsequently refers to them using their assigned label. This technique for referencing modules is intentionally rigid, requiring programmers to precisely name the module they aim to reference; imprecision often results in syntactic incorrectness. Requiring evolving programs to follow traditional approaches to module referencing is not ideal: mutation operators must do extra work to guarantee label-correctness, else mutated programs are likely to make use of undefined labels, resulting in syntactic invalidity [20]. Instead, is genetic programming (GP) better off relying on more flexible, less exacting referencing schemes?

Inspired by John Holland's use of "tags" [8–11] as a mechanism for matching, binding, and aggregation, Spector et al. [20–22] introduced and demonstrated a tag-based naming scheme for GP where tags are used to name and reference

A. Lalejini (✉) · C. Ofria
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA
e-mail: lalejini@msu.edu; ofria@msu.edu

W. Banzhaf et al. (eds.), *Genetic Programming Theory and Practice XVI*,
Genetic and Evolutionary Computation,
https://doi.org/10.1007/978-3-030-04735-1_6

program modules. Tags are evolvable labels that are mutable, and the similarity (or dissimilarity) between any two possible tags is quantifiable. Tags allow for *inexact* referencing. Because the similarity between tags can be calculated, a referring tag can always link to the program module with the *most similar* tag; further, this ensures that all possible tags are valid references. Because tags are mutable, evolution can incrementally shape tag-based references within evolving code. Spector et al. demonstrated the value of an evolvable name, showing that the tag-based naming scheme supports the evolution of programs with complex, modular architectures by allowing programs to more easily reference and make use of program modules [20].

We previously extended Spector et al.'s tag-based naming scheme, broadening the application of tags to develop SignalGP [14], a GP technique designed to provide direct access to the event-driven programming paradigm. In Spector et al.'s original implementation [22], tags were used as an evolvable mechanism to label and later refer to code fragments. At there core, tags provide general-purpose, evolvable specificity—an evolvable way to specify zero or more tagged entities. SignalGP broadens the application of tags, using them to specify the relationships between events and event handlers (i.e., program modules that process events). However, the application of tag-based naming can be *further* broadened. For example, tag-based naming could be used to label and refer to particular instructions, other agents, or other virtual hardware components (e.g., registers, locations in memory, etc.). In this broader context, tags are still mutable labels with well-defined tag-tag similarity measures, allowing for inexact referencing. The context of a referring tag can limit the valid set of tagged entities with which it can match. For example, in the context of a function call, a referring tag might only match to function tags, whereas in the context of a memory access, a referring tag might only match to a tagged location in memory.

In this chapter, we investigate the importance of inexactness when making tag-based references, and we propose possible extensions to SignalGP that use broader applications of tag-based naming. In Sect. 6.2, we give a brief overview of SignalGP. In Sect. 6.3, we use an environment coordination toy problem to investigate the effectiveness of different thresholds of allowed imprecision when performing tag-based referencing. We compare the fitness effects of requiring different levels of tag similarity when matching referring tags to referents, ranging from requiring exact matches between tags for a successful reference to placing no restrictions on tag similarity for a successful reference. We find that, indeed, allowing for some inexactness when performing tag-based referencing is crucial. In Sect. 6.4, we demonstrate that requiring a minimum threshold of similarity for tags to match is important when programs must evolve to ignore irrelevant or misleading environmental signals. In addition to providing access to the event-driven programming paradigm, the way SignalGP programs are organized is well-suited for several interesting extensions. In Sect. 6.5, we speculate on several possibilities for how SignalGP can be extended to support module regulation, multi-representation programs, and major transitions in individuality.

6.2 SignalGP

SignalGP defines a way of organizing and interpreting genetic programs to provide computational evolution direct access to the event-driven programming paradigm. The event-driven programming paradigm is a software-design philosophy where software development focuses on the processing of events (often in the form of messages from other processes, sensor alerts, or user actions) [2, 4, 5]. Events are processed by segments of code called event handlers. In traditional event-driven programming, some identifying characteristic associated with the event (e.g., its name or type) determines the most appropriate event handler to trigger for processing the event, and the programmer is responsible for labeling event handlers such that they process the appropriate types of events. Software development environments that support the event-driven paradigm often abstract away the logistics of monitoring for events and triggering event handlers. This technique simplifies the code that must be designed and implemented by the programmer in domains that require on-the-fly reactions to signals from the environment or other agents.

SignalGP provides similarly useful abstractions to *evolving* programs. In SignalGP, signals (events) trigger the execution of program modules (functions) to respond to those signals. SignalGP applies tag-based referencing techniques to specify which function is triggered by each signal, allowing the relationships between signals and functions to evolve over time.

Here, we give a general overview of SignalGP in the context of linear GP, wherein programs are represented as sequences of instructions; however, the underlying organization and interpretation of SignalGP programs is generalizable across a variety of evolvable representations of computation (see Sect. 6.5.2). Figure 6.1 is provided to visually guide our discussion of SignalGP. A more detailed discussion can be found in [14].

SignalGP programs (agents) are explicitly modular, composed of a set of functions, each of which associates a tag with a linear sequence of instructions.

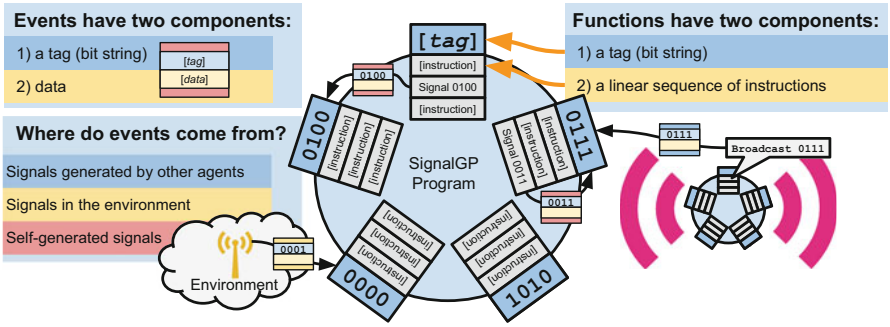

Fig. 6.1 A high-level overview of SignalGP. Programs are defined by a set of functions. Events trigger functions with the most similar tag, allowing programs to respond to signals. SignalGP agents handle many signals simultaneously by processing them in parallel

SignalGP makes explicit the concept of events. Each event is associated with a tag (indicating the event type) as well as additional event-specific data. In our work, we represent tags as fixed-length bit strings where tag similarity is quantified as the proportion of matching bits between two tags (simple matching coefficient). Because both events and functions are tagged, SignalGP uses tag-based referencing to determine the most appropriate function to process an event: events trigger the function with the closest matching tag as long as its within a fixed threshold. When an event triggers a function, the function is run with the event's associated data as input. In this way, functions act as event handlers, and tag-based referencing is used as an evolvable mechanism to determine the most appropriate function to trigger in response to an event. SignalGP agents handle many events simultaneously by processing them in parallel. Events may be generated internally, by the environment, or by other agents, making SignalGP particularly well-suited for domains that require programs to respond quickly to their environment or other agents.

The underlying instruction set is crafted to allow programs to easily trigger internal events, broadcast external events, and to otherwise work in a tag-based context. In our implementation of SignalGP, instructions are argument based, and as in traditional linear GP representations, arguments modify the effect of an instruction, often specifying memory locations or fixed values. In addition to evolvable arguments, each instruction has an evolvable tag, which may also modify the effect of an instruction. For example, instructions that refer to functions do so using tag-based referencing, and when an instruction generates an event (e.g., to be used internally or broadcast to other agents), the instruction's tag is used as the event's tag. The set of SignalGP instructions used in this work are documented in our supplemental material, which can be accessed via GitHub at https://github.com/amlalejini/GPTP-2018-Exploring-Evolvable-Specificity-with-SignalGP [13].

In Spector et al.'s original conception of tag-based referencing, as long as a program had at least one tagged module, all referential tags could successfully reference *something* [22]. The tag-based referencing employed by SignalGP, however, can be configured to only match tags whose similarity exceeds a threshold, allowing programs to ignore events by avoiding the use of similar tags. This similarity threshold allows us to adjust the degree of exactness required for tag-based references to succeed.

6.3 The Value of Imprecision in Evolvable Names

How important is imprecision when calling an evolvable name? Tag-based referencing has built-in flexibility, not requiring tags to *exactly* match to successfully reference one another. In Spector et al.'s initial implementation of tag-based referencing [22], referring tags *always* matched to the most similar receptor tag. Spector et al. speculated that tag-based referencing performed well because of

this inexactness: any tag-based reference is able to find a referent as long as one exists. We can, however, imagine different degrees of allowed imprecision when performing tag-based referencing, ranging from only identical tags being allowed to reference one another, to any two tags being allowed to match as long as they are the most similar pair. Indeed, any minimal level of tag-similarity for successful referencing can be imposed (e.g., requiring tags to be at least 50% similar before they can be considered as the best match).

Here, we explore the importance of imprecision in tag-based referencing using SignalGP. We evolve SignalGP agents to solve an environment coordination problem under a range of similarity thresholds, spanning from 0% (no similarity requirement) to 100% (requiring perfect matches).

6.3.1 The Changing Environment Problem

The changing environment problem is a toy problem that we designed to test GP programs' capacity to respond appropriately to environmental signals. We have previously used this problem to demonstrate the value of the event-driven paradigm using SignalGP [14].

The changing environment problem requires agents to continually match their internal state with the current state of a stochastically changing environment. The environment is initialized to a random state, and at every subsequent time step, the environment has a 12.5% chance of randomly changing to any of 16 possible states. To be successful, agents must monitor the environment for changes, adjusting their internal state as appropriate.

Environmental changes produce signals (events) with environment-specific tags that will trigger an appropriate SignalGP function; in this way, SignalGP agents can respond to environmental changes. Each of the 16 environment states is associated with a distinct tag that is randomly generated at the beginning of a run. Agents adjust their internal state by executing one of 16 state-altering instructions (one for each possible environmental state). Thus, the optimal solution to this problem is a 16-function program where each function is triggered by a different environment signal, and functions, when triggered, adjust the agent's internal state appropriately. An example solution to the changing environment problem is documented in our supplemental material, which can be accessed via GitHub at https://github.com/amlalejini/GPTP-2018-Exploring-Evolvable-Specificity-with-SignalGP [13].

To explore the value of imprecision in tag-based referencing, we evolved 30 replicate populations of SignalGP agents under nine treatments, each requiring a different similarity threshold for events to trigger functions: 0%, 12.5%, 25%, 37.5%, 50%, 62.5%, 75%, 87.5%, and 100%. Note that when performing a tag-based reference, if the closest matching tag is not greater than or equal to the required similarity threshold, the reference fails.

6.3.1.1 Hypothesis

A 100% similarity threshold is equivalent to exact-name referencing; thus, we expected it to perform poorly. A 0% similarity threshold is equivalent to what [22] used in their original demonstration of tag-based referencing; thus, we expected to it perform well. However, are intermediate thresholds just as effective? They provide varying degrees of allowed imprecisions while allowing programs to passively ignore some incoming signals. In prior work using SignalGP [14], a 50% similarity threshold performed well on the changing environment problem; thus, we expected treatments with intermediate thresholds to perform better than runs requiring exact tag-matching for references to succeed.

6.3.1.2 Experimental Parameters

For each treatment, we evolved 30 replicate populations of 1000 agents for 10,000 generations, starting from a simple ancestor program consisting of a single function with eight no-operation instructions. We initialized all replicates with a unique random number seed. Each generation, we evaluated all agents in the population three times (three trials). Each trial was composed of 256 time steps, and an agent's score for a single trial was equal to the number of time steps the agent's internal state matched the environment state. Thus, possible scores ranged from 0 to 256. An agent's fitness was the minimum score achieved after three trials, thus selecting agents that performed consistently. We used a combination of elite and tournament (size four) selection to determine which agents reproduced asexually each generation.

Offspring were mutated using SignalGP-aware mutation operators. We used whole-function duplication and deletion operators, applied at a per-function rate of 0.05; these operators allowed evolution to tune the number of functions in a SignalGP program. We mutated instruction- and function-tags at a per-bit mutation rate of 0.005. We applied instruction and argument substitutions at a per-instruction/argument rate of 0.005. We applied single-instruction insertion and deletion operators at a per-instruction rate of 0.005; when a single-instruction insertion occurred, we inserted a random instruction with random arguments and a random tag. In addition to single-instruction insertions and deletions, instruction sequences could be inserted or removed via slip-mutation operators [15]. When triggered, slip-mutations can either duplicate or delete multi-instruction sequences within a function. We applied slip-mutations at a per-function rate of 0.05.

Agents were limited to a maximum of 16 total functions, each of which were limited to a maximum length of 32 instructions. Agents were limited to a maximum of 32 parallel-executing threads. Agents were further limited to 128 call states per call stack. All tags were represented as length-16 bit strings.

6.3.1.3 Data Analysis

We analyzed evolving populations at two time points during the evolutionary process: generation 1000 and generation 10,000. For every population analyzed, we extracted the best-performing program and evaluated it 100 times (to account for environmental stochasticity), using its average performance as its representative fitness. For each time point (generation 1000 and 10,000) analyzed, we compared the performances of evolved programs across treatments. To determine if any of the treatments were significant ($p < 0.05$) within a set, we performed a Kruskal-Wallis test. For a time point in which the Kruskal-Wallis test was significant, we performed a post-hoc pairwise Wilcoxon rank-sum test, applying a Bonferroni correction for multiple comparisons. All statistical analyses were conducted in R 3.3.2 [18].

All visualizations of our results were generated using the seaborn Python library [23]. The code to run our experiments, perform statistical analyses, and generate visualizations is publicly available on Github [13].

6.3.2 *Results and Discussion*

Figure 6.2 gives the results for the changing environment problem early during our experiment (generation 1000) and at the end of our experiment (generation 10,000). At both generation 1000 and generation 10,000, programs evolved under different similarity thresholds had significantly different performance (Gen. 1000: Kruskal-Wallis test, Chi-squared = 161.27, $p < 2.2e{-}16$; Gen. 10,000: Kruskal-Wallis test, Chi-squared = 221.72, $p < 2.2e{-}16$). Table 6.1 gives the results of a post-hoc pairwise Wilcoxon rank-sum test for our results at both generation 1000 and generation 10,000.

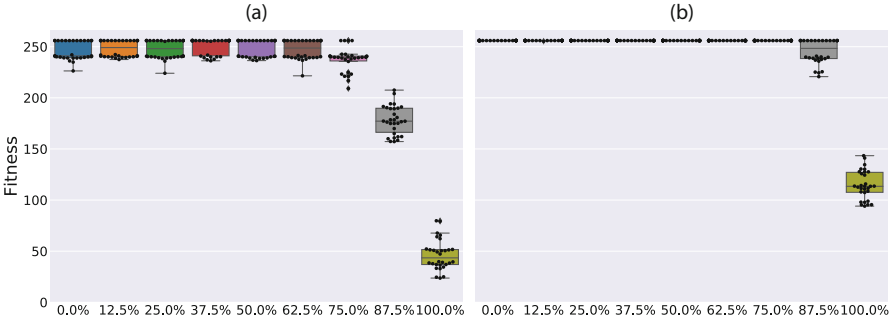

Fig. 6.2 Changing environment problem results at: (**a**) generation 1000 and (**b**) generation 10,000. The box plots indicate the fitnesses (each an average over 100 trials) of the best performing programs from each replicate across a range of minimum similarity thresholds

Table 6.1 Pairwise Wilcoxon rank-sum test results for the changing environment problem at generation 1000 and generation 10,000

Tag Similarity Threshold	0.0%	12.5%	25%	37.5%	50.0%	62.5%	75.0%	87.5%	100.0%	Generation 10,000
0.0%		1.0	NONE	NONE	NONE	NONE	NONE	**0.00027**	**2.5e-11**	
12.5%	1.0		1.0	1.0	1.0	1.0	1.0	**0.00079**	**3.6e-11**	
25%	1.0	1.0		NONE	NONE	NONE	NONE	**0.00027**	**2.5e-11**	
37.5%	1.0	1.0	1.0		NONE	NONE	NONE	**0.00027**	**2.5e-11**	
50.0%	1.0	1.0	1.0	1.0		NONE	NONE	**0.00027**	**2.5e-11**	
62.5%	1.0	1.0	1.0	1.0	1.0		NONE	**0.00027**	**2.5e-11**	
75.0%	0.3559	**0.0109**	0.0956	**0.0013**	0.0654	0.2005		**0.00027**	**2.5e-11**	
87.5%	**8.6e-10**	**7.6e-10**	**8.1e-10**	**5.2e-10**	**7.0e-10**	**7.6e-10**	**1.1e-09**		**4.4e-10**	
100.0%	**8.6e-10**	**7.6e-10**	**8.1e-10**	**5.2e-10**	**7.0e-10**	**7.6e-10**	**1.1e-09**	**1.1e-09**		
Tag Similarity Threshold	Generation 1,000									

Each row/column corresponds to a tag similarity threshold treatment. Each entry in the table indicates the Bonferroni-adjusted p value for a given comparison between two treatments. Statistically significant relationships ($p < 0.05$) are bolded. 'NONE' indicates that two treatments have identical distributions of data. Results for generation 1000 are in red, below the table's diagonal (in black). Results for generation 10,000 are in yellow, above the table's diagonal

At After 10,000 generations of evolution, programs evolved in the 87.5% and 100.0% similarity threshold treatments still perform significantly worse than those evolved in treatments with lower similarity thresholds. However, by 10,000 generations, some replicates evolved under the 87.5% similarity threshold treatment were able to produce optimal programs. No optimal programs evolved in the 100.0% similarity threshold treatment (exact name matching). Fully detailed statistical results can be found in our supplemental material [13].

Allowing for *Some* Imprecision is Crucial When Calling a Tag-Based Name

Because we limited agents to a maximum of 16 total functions, optimally solving the 16-state changing environment problem required programs to dedicate each of their 16 possible functions to responding to a particular environment state. Each function must be tagged such that only a single environment state change could trigger it, and when triggered, the function must immediately update the agent's internal state appropriately. If exact tag-matching (100% similarity threshold) is required for events to trigger functions, each function's tag must evolve to match a single environment state tag bit-for-bit. As expected, our results demonstrate that requiring tags to exactly match for successful references impedes evolution: after 10,000 generations, no optimal programs evolved under the 100.0% similarity threshold treatment.

In treatments that allow for inexactness when performing tag-based referencing, each function's tag must evolve to closely match (above a given similarity threshold) a single environment state; higher minimum required similarity thresholds require evolution to more precisely tune function tags. As demonstrated by the 87.5% similarity threshold treatment, requiring exceedingly high levels of precision can impede evolutionary adaptation.

By 10,000 generations, there was no significant difference in program performance among all treatments with similarity thresholds lower than 87.5%. While allowing for inexactness in tag-based referencing is crucial for evolving programs to solve the changing environment problem, intermediate levels of required precision (12.5%, 25.0%, 37.5%, 50.0%, 62.5%, and 75.0% similarity thresholds) proved just as effective as not imposing any tag similarity constraints (0.0% similarity threshold).

6.3.2.1 Illuminating Solution Space with MAP-Elites

We use the MAP-Elites [17] evolutionary algorithm to further illuminate the importance of inexactness when using tag-based naming schemes. In MAP-Elites, a population is structured based on a set of chosen traits of evolving solutions. Each chosen trait defines an axis on a grid of cells where each cell represents a distinct combination of the chosen traits; further, each cell maintains only the most fit (elite) solution discovered with the cell's associated combination of traits. A MAP-Elites grid is initialized by randomly generating solutions and placing them into their appropriate cell in the grid (based on the random solution's traits). After initialization, occupied cells are randomly selected to reproduce. When a solution is selected for reproduction, we generate a mutated offspring and determine where that offspring belongs in the grid. If the cell is unoccupied, the new solution is placed in that cell; otherwise, we compare the new solution's fitness to the current occupant, keeping the fitter of the two. Over time, this process produces a grid of prospective solutions that span the range of traits we used to define our grid axes.

Dolson et al. extended the use of MAP-Elites to examine GP representations [3]. By selecing MAP-Elites grid axes that correspond to program architecture, we can get a snapshot of what types of programs are capable of succeeding at a task and what tradeoffs might exist between the chosen traits. We use this approach to explore the role of inexactness in SignalGP: we apply the MAP-Elites algorithm to the changing environment problem, using minimum similarity threshold for tag-based referencing and the number of unique functions used by a program during evaluation to define our MAP-Elites grid axes. In our more traditional evolution experiment, we locked in the minimum required similarity threshold for each treatment. In our MAP-Elites analysis, we allow the minimum similarity threshold for a program to evolve between 0.0% and 100.0%. Further, we increased the allowed number of functions in a program from 16 to 32.

We initialized our MAP-Elites grid with 1000 randomly generated SignalGP programs. We ran the MAP-Elites algorithm for 100,000 generations where each generation represents 1000 reproduction events. We ran 50 replicate MAP-Elites runs, giving us 50 grids of diverse solutions for the changing environment problem. At the end of each run, we filtered out any program unable to solve the problem perfectly in each of our 50 runs. The heat map in Fig. 6.3 shows the density of optimal programs (aggregated across runs) within our chosen trait space.

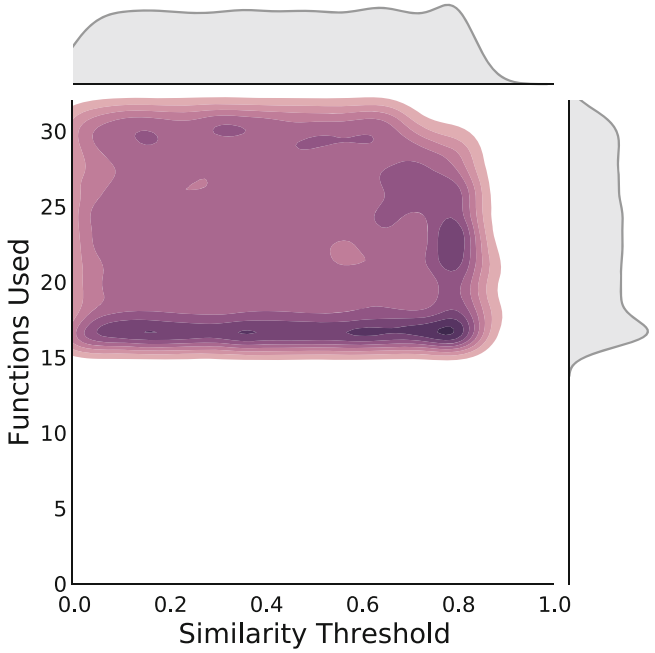

Fig. 6.3 Heat map of SignalGP programs evolved to solve the changing environment problem using MAP-Elites. Locations in the heat map correspond to distinct combinations of the following two program traits: the number of unique functions used by a program during evaluation and the program's minimum tag similarity threshold. Darker areas of the heat map indicate a higher density of perfect solutions found with a particular trait combination

From Fig. 6.3, we can see that all optimal programs use 16 or more functions. This is not surprising, as the changing environment problem cannot be optimally solved with fewer than 16 functions. The highest similarity threshold among all evolved solutions represented in Fig. 6.3 was 87.4657%, supporting the idea that requiring too much precision when performing tag-based referencing can impede evolution.

6.4 The Value of Not Listening

What's the value of ignoring signals in the environment? In some problem domains, the capacity to completely ignore distracting, irrelevant, or misleading signals while monitoring for others is crucial. For example, selective attention at a crowded restaurant allows us to ignore background noise and pay attention to a single conversion.

Here, we incorporate misleading distraction signals into the changing environment problem to demonstrate the value of ignoring signals in the context of SignalGP. In SignalGP, a 0% similarity threshold for tag-based references prevents agents from passively ignoring signals (events) in the environment. SignalGP programs can still be organized to *actively* ignore signals by having appropriately tagged, ineffectual functions to consume signals or by filtering signals based on event-specific data. In SignalGP, a 100% similarity threshold for tag-based references causes SignalGP programs to ignore any event whose tag is not an exact match with one of the agent's function tags, which we have shown to impede evolution (Sect. 6.3). Intermediate similarity thresholds, however, allow SignalGP agents to passively ignore signals in the environment without impeding adaptive evolution. We explore the value of allowing varying degrees of passive signal-discrimination via different similarity thresholds in SignalGP using the distracting environment problem.

6.4.1 The Distracting Environment Problem

The distracting environment problem is identical to the changing environment problem (described in Sect. 6.3.1) but with the addition of randomly occurring distraction signals. Like the changing environment problem, the environment can be in one of 16 states at any time with a 12.5% chance to change each update. Every time step there is also a 12.5% chance of a distraction event occurring, independent of environmental changes. Just as we randomly generate 16 distinct tags associated with each of the 16 environment states, we also generate 16 distinct distraction signal tags, which are guaranteed to not be identical to environment-state tags. Thus, to be successful, agents must monitor the environment for changes (adjusting their internal state as appropriate) while ignoring misleading distraction signals.

We repeated the experiment described in Sect. 6.3 with identical experimental treatments and parameters, but in the context of the distracting environment problem instead of the changing environment problem.

6.4.1.1 Hypothesis

As in the changing environment problem, optimal performance in the distracting environment problem requires 16 functions, each tagged such that it is triggered by a single environment-state signal; once triggered, a function must adjust the agent's internal state appropriately. However, the distracting environment problem also requires agents to ignore distraction signals. If a distraction signal is able to trigger a function, the agent cannot reliably maintain an internal state that matches the current environment state. Given that agents must dedicate 16 functions to adjusting internal state in response to environmental changes, they must be able to passively ignore distraction signals to avoid triggering an erroneous internal state. As such, the

0% similarity threshold treatment cannot produce optimally-performing programs. Further, intermediate similarity thresholds must be high enough to allow agents to passively discriminate between distraction signals and environmental changes. We expect treatments with higher intermediate similarity thresholds to be able to achieve optimality.

6.4.1.2 Statistical Methods

Our statistical methods for analyzing these data are identical to those described in Sect. 6.3.1.3.

6.4.2 Results and Discussion

Figure 6.4 gives the results for the distracting environment problem early during our experiment (generation 1000) and at the end of our experiment (generation 10,000). At both generation 1000 and generation 10,000, programs evolved under different similarity thresholds had significantly different performance (Gen. 1000: Kruskal-Wallis, Chi-squared = 144.3, $p < 2.2e{-}16$; Gen. 10,000: Kruskal-Wallis, Chi-squared = 193, $p < 2.2e\text{-}16$). Table 6.2 gives the results of a post-hoc pairwise Wilcoxon rank-sum test for our results at both generation 1000 and generation 10,000.

As in the changing environment problem, runs requiring exact name matching (the 100% tag similarity threshold treatment) produce programs that perform significantly worse than those evolved in all other treatments. At generations 1000 and 10,000, programs evolved in the 75% tag similarity threshold treatment significantly outperform programs evolved in all other treatments. By 10,000

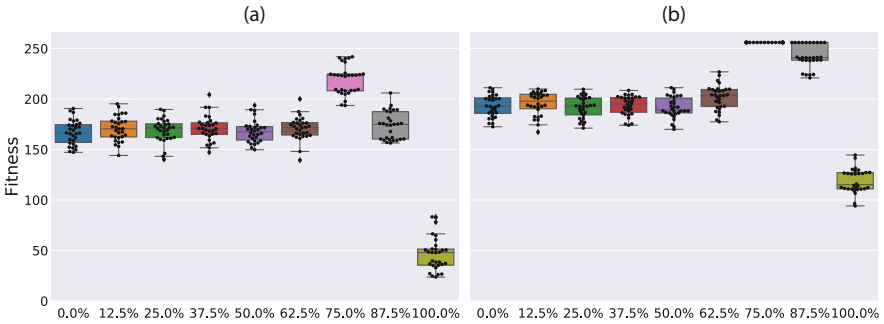

Fig. 6.4 Distracting environment problem results at: (**a**) generation 1000 and (**b**) generation 10,000. The box plots indicate the fitnesses (each an average over 100 trials) of the best performing programs from each replicate across a range of minimum similarity thresholds

Table 6.2 Pairwise Wilcoxon rank-sum test results for the distracting environment problem at generation 1000 and generation 10,000

Tag Similarity Threshold	0.0%	12.5%	25.0%	37.5%	50.0%	62.5%	75.0%	87.5%	100.0%
0.0%		1.0	1.0	1.0	1.0	0.240	**4.4e-11**	**1.4e-09**	**1.1e-09**
12.5%	1.0		1.0	1.0	1.0	1.0	**4.4e-11**	**1.4e-09**	**1.1e-09**
25.0%	1.0	1.0		1.0	1.0	0.072	**4.4e-11**	**1.4e-09**	**1.1e-09**
37.5%	1.0	1.0	1.0		1.0	0.183	**4.4e-11**	**1.4e-09**	**1.1e-09**
50.0%	1.0	1.0	1.0	1.0		0.065	**4.4e-11**	**1.4e-09**	**1.1e-09**
62.5%	1.0	1.0	1.0	1.0	1.0		**4.4e-11**	**2.7e-09**	**1.1e-09**
75.0%	**1.1e-09**	**1.3e-09**	**1.1e-09**	**1.5e-09**	**1.2e-09**	**1.5e-09**		**7.2e-06**	**4.4e-11**
87.5%	1.0	1.0	1.0	1.0	1.0	1.0	**3.3e-09**		**1.4e-09**
100.0%	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.1e-09**	**1.6e-09**	

Tag Similarity Threshold (Generation 1,000) — Generation 10,000

Each row/column corresponds to a tag similarity threshold treatment. Each entry in the table indicates the Bonferroni-adjusted p value for a given comparison between two treatments. Statistically significant relationships ($p < 0.05$) are bolded. Results for generation 1000 are in red, below the table's diagonal (in black). Results for generation 10,000 are in yellow, above the table's diagonal

generations, only the 75% and 87.5% tag similarity threshold treatments produced perfectly optimal programs. Fully detailed statistical results can be found in our supplemental material, which can be accessed via GitHub [13].

Requiring *Some* Precision When Calling a Tag-Based Name Can Be Important, Too

These data are not surprising: we designed the distracting environment problem as a toy problem to demonstrate the idea that sometimes requiring some amount of precision when using tag-based referencing can be important. Because we limited programs to 16 functions and all 16 functions were required to monitor for environment changes, solving the distracting environment problem required programs to have the capacity to discriminate between true, meaningful signals and irrelevant, meaningless signals. However, even in this case where signal discrimination was crucial, requiring exact tag-matching for signals to successfully trigger program functions was still too harsh a requirement for well-performing programs to evolve.

For both the changing environment and distracting environment problems, the 75% tag similarity threshold treatments produced optimally performing programs, allowing for sufficient signal discrimination in the distracting environment problem while not too badly impeding evolution's ability to bootstrap program responses to true signals. However, these data do not necessarily imply anything general about a 75% tag similarity threshold. The critical tag similarity threshold for the distracting environment problem depends on the number of distraction signals that must be ignored versus the number of true signals the programs must respond to, the number of bits composing a tag (here, we used 16), as well as the number of functions SignalGP programs are allowed to have.

6.4.2.1 Illuminating Solution Space with MAP-Elites

As we did for the changing environment problem, we again use the MAP-Elites evolutionary algorithm [17] to illuminate the solution space for the distracting environment problem. We apply MAP-Elites to the distracting environment problem exactly as described in Sect. 6.3.2.1, using minimum tag similarity threshold for tag-based referencing and the number of unique functions used during program evaluation as our MAP-Elites grid axes. The heat map in Fig. 6.5 shows the density of optimal programs evolved using MAP-Elites within our chosen trait space.

Figure 6.5 confirms our intuition about the solution space in the distracting environment problem, showing that many strategies with a wide range of minimum tag similarity thresholds exist that use around 32 functions where extra 'dummy' functions can consume distraction signals. Indeed, there are optimal solutions that use only 16 functions; however, these solutions seem require high minimum tag similarity thresholds.

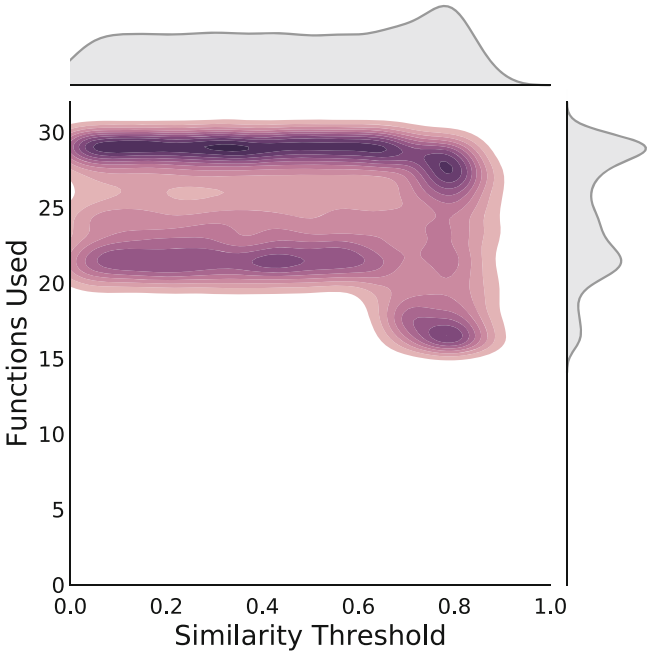

Fig. 6.5 Heat map of SignalGP programs evolved to solve the distracting environment problem using MAP-Elites. Locations in the heat map correspond to distinct combinations of the following two program traits: the number of unique functions used by a program during evaluation and the program's minimum tag similarity threshold. Darker areas of the heat map indicate a higher density of solutions found with a particular trait combination

6.5 What Else Is in an Evolved Name? Broadened Applications of Tag-Based Naming in SignalGP

Thus far, we have explored the importance of inexactness in evolvable names in the context of SignalGP. In this section, we discuss several extensions to the SignalGP framework that are possible because of the evolvable specificity afforded by its tag-based naming scheme.

6.5.1 SignalGP Function Regulation

Bringing together ideas from GP and gene regulatory networks is not novel [1, 16]. The capacity to regulate genotypic expression is valuable in both biological and computational systems, allowing environmental feedback to alter phenotypic traits within an individual's lifetime.

SignalGP is easily extended to model gene regulatory networks where functions can be up-regulated (i.e., be made more likely to be referenced by a tag) or down-regulated (i.e., be made less likely to be referenced by a tag). For example, a function that would normally not be triggered by an event can be up-regulated to increase its priority over other function that have closer match. We can add regulatory instructions to the instruction set that increase or decrease function regulatory modifiers, using tag-based referencing to determine which function should be regulated by a particular instruction.

Gene regulation provides yet another mechanism for phenotypic flexibility, allowing SignalGP programs to alter referential relationships in response to environmental feedback. Such a mechanism might be useful for problems that require within-lifetime learning or general behavioral plasticity.

6.5.2 Multi-Representation SignalGP

In this work and in prior work, we have exclusively used SignalGP in the context of linear GP: SignalGP functions associate a tag with a linear sequence of instructions. However, in principle, SignalGP is generalizable across a variety of evolutionary computation representations.

SignalGP programs are composed of a set of functions where each function is referred to via its tag. We can imagine these functions to be black-box input-output machines: when called or triggered by an event, they are run with input and can produce output by manipulating memory or by generating signals. We have exclusively used linear GP in SignalGP functions; however, we could have just as easily used other types of representations capable of receiving input and producing output (e.g., other GP representations, artificial neural networks, Markov Brains [6], hard-coded modules, etc.). We could even employ a variety of representations within a single agent.

The evolvable specificity afforded by SignalGP's tag-based naming scheme allows us to use this sort of black-box metaphor. Functions composed of different representations can still refer to one another via tags, and events are agnostic to the underlying representation used to handle them, requiring only that the representation is capable of processing event-specific data. Allowing for these types of multi-representation agents may complicate the SignalGP virtual hardware, program evaluation, and mutation operators, but it would provide evolution with a toolbox of diverse representations.

Hintze et al. proposed and demonstrated the evolutionary Buffet Method where Markov Brains [6] could be composed of heterogeneous computational substrates, allowing evolution to work out the most appropriate representation for a given problem [7]. Further, Hintze et al.'s Buffet Method demonstrated the success of hybrid solutions. Multi-representation SignalGP provides an unexplored, alternative approach to evolving multi-representation agents, bringing the Buffet Method into an event-driven context.

6.5.3 Major Transitions in SignalGP

In a major evolutionary transition in individuality, formerly distinct individuals unite to form a new, more complex lifeform, redefining what it means to be an individual. The evolution of eukaryotes, multi-cellular life, and eusocial insect colonies are all examples of transitions in individuality. Often the individuals that make up the higher-level entity are limited to local information, lacking direct access to the global state of the higher-level unit; lower-level units must rely on signaling and sensory information to coordinate their roles in the group [19, 24]. In a computational sense, a major transition in individuality is the evolution of a distributed system. Capturing these types of transitions in GP would give evolution a mechanism to incrementally form distributed systems from formerly individual programs.

In the previous section, we described how SignalGP could be extended to allow multi-representation programs where functions (modules) can be of any representation capable of receiving input and producing output. We can take this approach to multi-representation SignalGP one step further: any module within a SignalGP agent could be *another* (former) SignalGP agent. This approach is conceptually similar to Tangled Program Graph representation [12].

We can imagine a mutation operator that, when applied, induces transitions in individuality by injecting co-evolving SignalGP programs as self-contained, tagged modules into the program being mutated, allowing single individuals to be aggregates of lower-level individuals. Further, transitions in individuality can be applied *hierarchically*. Biological evolution has examples of such hierarchical transitions: eusocial insect colonies are composed of many multicellular individuals, each which are composed of many eukaryotic cells, which in turn are composed of organelles (many of which are thought to have been formally distinct individuals).

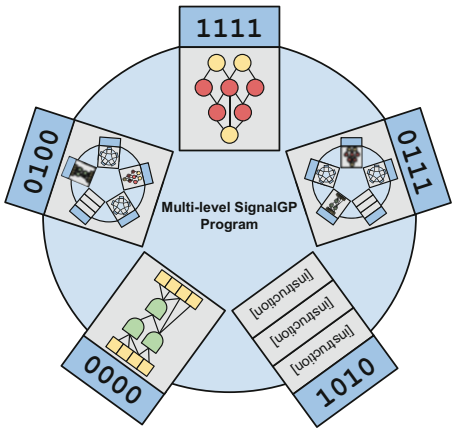

Fig. 6.6 Example of a multi-level SignalGP program. In this example, the agent is composed of five modules, including a neural network, a Markov Brain, a linear GP representation, and two multi-module programs at a lower level of organization

An individual SignalGP program may be composed of many SignalGP program modules, which may themselves be composed of many SignalGP programs, and so on (Fig. 6.6).

Implementing a mutation operator capable of inducing arbitrary numbers of hierarchical transitions in individuality requires us to answer the following questions: How should formerly individual programs interact when forced into an aggregate? And, how should an evolutionary algorithm handle evaluating both individuals and aggregates of individuals?

From the evolutionary algorithm's perspective, a multi-level SignalGP program is indistinguishable from a single-level. However, just as biological organisms composed of lower-level units of individuality require more energy to subsist, multi-level SignalGP programs require many more CPU cycles than single-level SignalGP programs. This is consistent with biology where major transitions disproportionately occur in energy-rich environments [19].

Extending SignalGP to support hierarchical transitions in individuality would provide a useful model for studying biological evolutionary transitions, allowing us to ask general questions about their dynamics. A transition in individuality mutation operator would also allow us to solve problems that might be best solved by a distributed system without knowing the optimal configuration of that distributed system a priori.

6.6 Conclusion

In this chapter, we explored the importance of inexactness when calling a tag-based name in GP. We show that allowing for inexactness when performing tag-based references is crucial for rapid adaptive evolution. Conversely, when some signals need to be ignored (such as in our distracting environment) it can be critical

to prevent dissimilar tags from finding incorrect matches. As such, intermediate thresholds for tag similarity may be ideal for optimal evolution in a broad range of environments. The most appropriate similarity thresholds for a given problem will depend on the specifics of the problem and the representation used. For example, we would need to consider the ways tags are used in a particular problem, as well as how those tags are represented, mutated, and compared.

Interestingly, while exact naming is the most intuitive referencing mechanism for human programmers, evolution is far more successful when program references are allowed to be inexact. In fact, mutation-selection balance may prevent exact references from being stably maintained over evolutionary time. If tags are mutated such that we expect at least one of a program's tags (referring or referent) to be mutated per reproduction event, the relationships between referring and referent tags are unlikely to be stably maintained.

Both SignalGP and our proposed extensions to SignalGP are inspired by biological systems and processes. As we continue to develop SignalGP, our goal is to continue to push the boundary of GP and to use SignalGP as a tool to study the natural systems that inspired its development, such as the evolution of modularity, gene regulation, cell signaling, and major evolutionary transitions in individuality.

Acknowledgements We extend our thanks to the members of the Digital Evolution Laboratory at Michigan State University and the attendees of the 2018 Genetic Programming Theory and Practice Workshop for thoughtful discussions and feedback on this work. This research was supported by the National Science Foundation (NSF) through the BEACON center (Cooperative Agreement DBI-0939454), a Graduate Research Fellowship to AL (Grant No. DGE-1424871), and NSF Grant No. DEB-1655715 to CO. Michigan State University provided computational resources through the Institute for Cyber-Enabled Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or MSU.

References

1. Banzhaf, W.: Artificial Regulatory Networks and Genetic Programming. In: Genetic Programming Theory and Practice, pp. 43–61. Springer US, Boston, MA (2003)
2. Cassandras, C.G.: The event-driven paradigm for control, communication and optimization. Journal of Control and Decision **1**, 3–17 (2014)
3. Dolson, E., Lalejini, A., Ofria, C.: Exploring genetic programming systems with MAP-Elites. In: Banzhaf, W., Spector, L., Sheneman, L. (eds.) Genetic Programming Theory and Practice XVI. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-04735-1_1
4. Etzion, O., Niblett, P.: Event Processing in Action. ISBN: 9781935182214. Manning Publications (2010)
5. Heemels, W.P., Johansson, K.H., Tabuada, P.: An introduction to event-triggered and self-triggered control. Proceedings of the IEEE Conference on Decision and Control pp. 3270–3285 (2012)
6. Hintze, A., Edlund, J.A., Olson, R.S., Knoester, D.B., Schossau, J., Albantakis, L., Tehrani-Saleh, A., Kvam, P., Sheneman, L., Goldsby, H., Bohm, C., Adami, C.: Markov Brains: A Technical Introduction. arXiv **1709.05601** (2017)

7. Hintze, A., Schossau, J., Bohm, C.: The evolutionary Buffet method. In: Banzhaf, W., Spector, L., Sheneman, L. (eds.) Genetic Programming Theory and Practice XVI. Springer International Publishing, Cham (2018). http://10.1007/978-3-030-04735-1_2
8. Holland, J.: The effect of labels (tags) on social interactions. Santa Fe Inst., Santa Fe, NM, Working Paper pp. 93–10 (1993). URL http://samoa.santafe.edu/media/workingpapers/93-10-064.pdf
9. Holland, J.H.: Genetic Algorithms and Classifier Systems: Foundations and Future Directions. In: Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA87), pp. 82–89 (1987)
10. Holland, J.H.: Concerning the emergence of tag-mediated lookahead in classifier systems. Physica D: Nonlinear Phenomena **42**, 188–201 (1990). DOI 10.1016/0167-2789(90)90073-X
11. Holland, J.H.: Studying complex adaptive systems. Journal of Systems Science and Complexity **19**, 1–8 (2006)
12. Kelly, S., Heywood, M.I.: Emergent Tangled Graph Representations for Atari Game Playing Agents. In: European Conference on Genetic Programming (EuroGP-2017), pp. 64–79 Springer (2017)
13. Lalejini, A.: amlalejini/GPTP-2018-Exploring-Evolvable- Specificity-with-SignalGP (2018). Available on GitHub at https://github.com/amlalejini/GPTP-2018-Exploring-Evolvable-Specificity-with-SignalGP
14. Lalejini, A., Ofria, C.: Evolving Event-driven Programs with SignalGP. arXiv **1804.05445** (2018)
15. Lalejini, A., Wiser, M.J., Ofria, C.: Gene Duplications Drive the Evolution of Complex Traits and Regulation. Proceedings of the European Conference on Artificial Life (ALIFE-2017), 4–8 (2017)
16. Lopes, R.L., Costa, E.: Genetic programming with genetic regulatory networks. In: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation - GECCO '13, p. 965. ACM Press, New York, New York, USA (2013)
17. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites. arXiv **1504.04909**, 1–15 (2015)
18. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2016). URL https://www.R-project.org/
19. Smith, J.M., Szathmary, E.: The major transitions in evolution. Oxford University Press (1997)
20. Spector, L., Harringtion, K., Martin, B., Helmuth, T.: What's in an Evolved Name? The Evolution of Modularity via Tag-Based Reference. In: R. Riolo, E. Vladislavleva, J.H. Moore (eds.) Genetic Programming Theory and Practice IX, Genetic and Evolutionary Computation, chap. 1, pp. 1–16. Springer New York, New York, NY (2011)
21. Spector, L., Harrington, K., Helmuth, T.: Tag-based modularity in tree-based genetic programming. GECCO '12: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation pp. 815–822 (2012)
22. Spector, L., Martin, B., Harrington, K., Helmuth, T.: Tag-based modules in genetic programming. GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation pp. 1419–1426 (2011)
23. Waskom, M., et al.: mwaskom/seaborn: v0.8.1 (september 2017) (2017). URL https://doi.org/10.5281/zenodo.883859
24. West, S.A., Fisher, R.M., Gardner, A., Kiers, E.T.: Major evolutionary transitions in individuality. Proceedings of the National Academy of Sciences **112**, 10,112–10,119 (2015)