Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees

Petter Ögren*

Swedish Defence Research Agency (FOI), Stockholm, SE-164 90, Sweden

In this paper, we argue that the modularity, reusability and complexity of Unmanned Aerial Vehicle (UAV) guidance and control systems might be improved by using a Behavior Tree (BT) architecture. BTs are a particular kind of Hybrid Dynamical Systems (HDS), where the state transitions of the HDS are implicitly encoded in a tree structure, instead of explicitly stated in transition maps. In the gaming industry, BTs have gained a lot of interest, and are now replacing HDS in the control architecture of many automated in-game opponents. Below, we explore the relationship between HDS and BTs. We show that any HDS can be written as a BT and that many common UAV control constructs are quite naturally formulated as BTs. Finally, we discuss the positive implications of making the above mentioned state transitions implicit in the BTs.

I. Introduction

Guidance and control of UAVs is an active research area,[1,2] and so far the main effort has been on how to make the UAVs do the tasks *right*, i.e. designing the controllers,[3–5] rather than doing the right *tasks*, i.e. designing the control architecture. However, when an increasing number of capabilities are added to a UAV system, the problem of how and when to switch, between different tasks and corresponding controllers, becomes increasingly important. Some tasks have to be done in the correct order, such as taxing and takeoff, while the execution of others depends on the situation and a given order of priority, such as collision avoidance and waypoint following.

This new problem area has long been the focus of computer game AI programmers, as computer game entities are virtual, and therefore less troubled by e.g. robustness issues related to real world sensing and control. It is therefore relevant to review some of the developments in the computer game industry, and see how they can be applied to the problems of the UAV control community.

In computer games, the control architecture of automated opponents if often designed using Finite State Machines (FSM).[6,7] Since these opponents themselves often move about in some kind of continuous space, the combined system can be modeled by a Hybrid Dynamical System (HDS).[8]

HDS have also been used in the field of robotics.[9–13] However, as the controllers become more complex, the number n of discrete states of the HDS grows, and it becomes increasingly difficult to keep track of all the $\mathcal{O}(n^2)$ possible transitions between those states. Thus, even though HDS are modular when it comes to the continuous dynamics, the discrete dynamic of the HDS is explicitly encoded in the transition functions, which makes the overall HDS much less modular that what it might seem at the first glance. For example, if one discrete state is removed from a HDS, *all* transitions to and from that state must be redesigned, just to ensure that the HDS is formally valid. Thus, the lack of modularity when it comes to state transitions heavily reduces scalability and reusability.

BTs is a way to encode a HDS that replaces the explicitly listed state transition functions of classic HDS with transitions that are implicitly given by a tree structure, see details below, and in references.[6,7,14–16] Thus, a given BT can be inserted as a subtree anywhere in any other BT and still be perfectly valid, and the transitions between the new and old parts of the combined tree will be implicitly given by the combined tree structure. Similarly, any part of a BT can be removed, as long as the remaining tree is connected, without causing invalid transitions.

*Deputy Reseach Director, Department of Information and Aerosystems

The main contribution of this paper is that we describe and formalize the development of BTs in the computer game industry and relate it to the HDS used in the UAV and robotics community. We furthermore show how a general HDS can be written in terms of a BT while keeping the modular structure of the continuous HDS-dynamics, and finally describe the benefits of making the HDS state transitions implicit in a BT.

The outline of this paper is as follows. In Section II we give a brief background on both HDS and BTs. Then, Section III describes the advantages of using BTs to model HDS. Section IV then gives a number of examples of BTs for UAV guidance and control and conclusions are drawn in Section V.

II. Background: Behavior Trees and Hybrid Control Systems

In this section we give formal descriptions of both HDS and BTs.

II.A. Hybrid Dynamical Systems

Following,[8] but using ordinary difference equations instead of ordinary differential equations for the continuos dynamics, in order to avoid technical issues regarding uniqueness and existence for the BTs, we let a hybrid dynamical system (HDS) be a system $H = (Q, \Sigma, A, G)$, with parts as follows:

- $Q = \{b_i\}$ is the countable set of discrete states.
- $\Sigma = \{\Sigma_q\}_{q\in Q}$ is the collection of systems of ordinary difference equations. Thus, each system Σ_q has a map $f_q : X_q \to X_q$, representing the continuous dynamics, evolving on $X_q \subset \mathbb{R}^{d_q},\ d_q \in \mathbb{Z}_+$, which are the continuous state spaces.
- $A = \{A_q\}_{q\in Q}$, where $A_q \subset X_q$ for each $q \in Q$, is the collection of autonomous jump sets.
- $G = \{G_q\}_{q\in Q}$, where $G_q : A_q \to S$ are the autonomous jump transition maps, said to represent the discrete dynamics.

Thus, $S = \bigcup_{q\in Q} X_q \times \{q\}$ is the hybrid state space of H. For convenience, we write $S_q = X_q \times \{q\}$, and $A = \bigcup_{q\in Q} A_q \times \{q\}$. Similarly, $G : A \to S$ is the autonomous jump transition map, constructed component wise in the obvious way.

The dynamics of the HDS H are as follows. The system is assumed to start in some hybrid state in $S\backslash A$, say $s_0 = (x_0, q_0)$, where $q_0 = b'$. It evolves according to $x_{n+1} = f_{b'}(x)$, $q_{n+1} = q_n$ until the state enters - if ever - $A_{b'}$ at the point $s_n = (x_n, q_n)$. It then transfers according to the transition map to $G_{b'}(x_n) = (x_{n+1}, q_{n+1}) \equiv s_{n+1}$, from which the process continues.

II.B. Behavior Trees

Loosely following,[6,7,15] we let a Behaviour Tree be a directed tree, with nodes and edges. If two nodes are connected by an edge, we call the outgoing node the parent and the incoming node a child. Nodes that have no children are denoted leaves, and the one node without parents is denoted the root node. Now, each node of the BT is labeled as belonging to one of the six different types listed in Table 1. If the node is not a leaf it can be one of the first four types, *Selector*, *Sequence*, *Parallel* or *Decorator*, and if it is a leaf it is one of the last two types, *Action* or *Condition*.

Upon execution of the BT, each time step of the control loop, the root of the BT is *ticked*. This tick is then progressed down (then back up) the tree according to the types of each node. Once a tick reaches a leaf node (Action or Condition), the node does some computation, possibly affecting some continuous or discrete states/variables of the BT, and then returns either *Success*, *Failure* or *Running*. The return status is then progressed up the tree, back towards the root, according to the types of each node. We will now describe how all the different node types handle the tick and processes the different return statuses.

Selector. Selectors are used to find and choose the first child that is successful. A Selector will return immediately with a status code success or running when one of its children returns success or running, see Table 1 and the pseudo code below. The child tasks are ticked in the order of importance. Figure 1 illustrates a simple BT with one selector and a set of Actions.

Table 1. The six node types of a BT.

Node type	Succeeds	Fails	Running
Selector	If one child succeeds	If all children fail	If one child returns running
Sequence	If all children succeeds	If one child fails	If one child returns running
Parallell	If N children succeeds	If M-N children fail	If all children return running
Decorator	Varies	Varies	Varies
Action	Upon completion	When impossible to complete	During completion
Condition	If true	If false	Never

```
SELECTOR (pseudocode)
For i = 1 to N (number of children)
   childStatus = Tick (Child(i))
   If childStatus == Running
      return Running
   else if childStatus == Success
      return Success
   end
end
return Failure
```

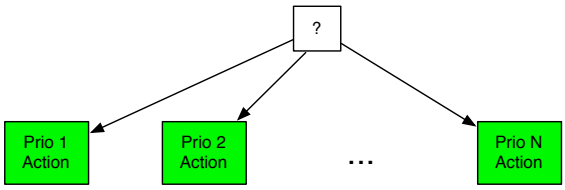

Figure 1. The Selector ticks its children in order until one returns Success or Running. Selectors are denoted by a white square with a question mark and Actions are denoted by a green square.

Sequence. A Sequence sequentially executes all its children in order. It will return immediately with a failure status code when one of its children fails, or with a running status code when one of its children returns running. As long as its children are succeeding, it will keep going. If it runs out of children, it will return success. See Figure 2.

```
SEQUENCE (pseudocode)
For i = 1 to N (number of children)
   childStatus = Tick (Child(i))
   If childStatus == Running
      return Running
   else if childStatus == Failure
      return Failure
   end
end
return Success
```

Parallel. A Parallel node executes all its children in parallel. It is configured to return success if more than a given fraction of the children returns success etc, see Figure 3

```
PARALLEL (pseudocode)
For i = 1 to N (number of children)
   childStatus(i) = Tick (Child(i))
```

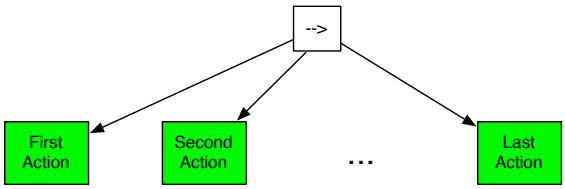

Figure 2. The Sequence ticks its children in order until one returns failure or running.

```
end
If allRunning(childStatus)
   return Running
else if moreThanMsuccess(childStatus)
      return Success
else return Failure
```

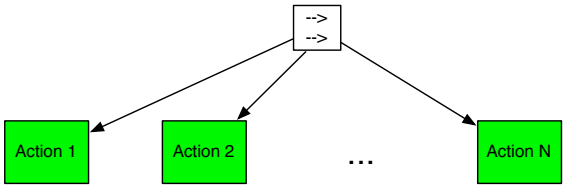

Figure 3. The Parallel node ticks all its children at the same time.

Decorators. Decorators are non-leaves that can have only one child. The decorator alters the behavior of its child by either manipulating its return status, or not ticking the child at all, and compute the return status based on some other criteria. An example can be found in Figure 4, where we have a decorator that starts a timer once its child returns success or running, it then returns whatever the child returns until a given time has passed, after which it return failure without ticking the child. Other examples of decorators include those that invert the output of their child, always return 'Success' or 'Failure', or counters that limit the number of times the child is allowed to execute.

```
DECORATOR (pseudocode)
if (some condition)
   childStatus = Tick (Child)
end
status = someFunction()
return status
```

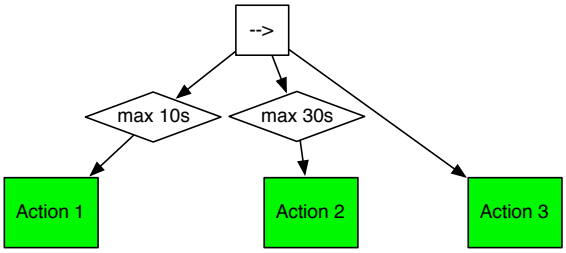

Figure 4. Two 'max-time' decorators, limiting the running time of Action 1 and Action 2 to 10 and 30 seconds respectively.

Action. An Action node performs an action, and returns Success if the action is completed, Failure if it can not be completed and Running if completion is under way.

Condition. A Condition node determines if a condition C has been met. Conditions are technically a subset of the Actions, but are given a separate category and graphical symbol to improve readability of the BT and emphasize the fact that they never return running and do not change any internal states/variables of the BT. Examples of Conditions can be found in Figure 5 below.

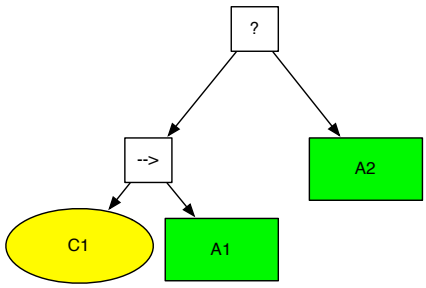

Figure 5. A Selector, a Sequence, a Condition (yellow) and two Actions (green). Action A1 is only performed when Condition C1 returns Success and Action A2 is only performed if C1 or A1 returns Failure.

III. Advantages of using Behavior Trees

As described above, the main advantage of using BTs to model HDS is that the transition functions of the HDS are made implicit, and captured by the tree structure, instead of being explicitly coded in the dynamics of each discrete state. This makes the BT truly modular. Actions, conditions and even subtrees can easily be added or removed from a BT without having to make careful revision of all transitions.

The two most important constructs, selectors and sequences, are furthermore extensions of the classical operators OR (selectors) and AND (sequences) and thus provide a flexible and powerful framework combined with the implications of the added Running return status. They furthermore provide easy encoding of two very natural HDS features, goal directed execution of one task after the other (sequences) and having a series of *fall back actions* if the current action is not applicable, or fails (selectors). Both these natural HDS features take a lot of work to encode in a standard HDS framework using only transitions, and even more works is needed when actions are added, or removed.

The advantages of using implicit, instead of explicit transitions can also be seen in the light of the following programming analogy. In early programming languages, e.g. BASIC, so-called *one way* transfers of control (goto-statements) were often encouraged, whereas most more recent programming languages, e.g. Java, are relying on *two way* transfers of control (function calls). The arguments against using goto-statements are perhaps best illustrated by the words of Dijkstra[17] "The unbridled use of the goto-statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. ... The goto statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program." In view of the above, one can argue that the classic transitions of the HDS are indeed *one way* transfers of control, similar to the goto-statement, whereas the implicit, *two way* transfer of control used in the BT are more similar to function calls, and that the drawbacks of the goto-statement are also true for classic HDS state transitions.

IV. Examples of UAV Control Behavior Trees

In this section we will describe a number of example UAV control BTs. First the BT-version of the Subsumption architecture, then the common construct of a number of tasks that has to be done in order, then an example showing how a general HDS can be written as a BT and finally a somewhat more complex Combat UAV controller example.

IV.A. Task that are prioritized - The subsumption architecture

The subsumption architecture for robot control made famous by Brooks[18] can be implemented by a single selector and a set of prioritized actions, as in Figure 1. This same architecture can also be interpreted in

terms of a number of fall back controllers. Using the labels of Figure 1, Prio 1 Action is the first choice controller, but if it is not applicable, or does not succeed for some reason, Prio 2 Action is invoked, and so on. An example where such a subtree would be relevant is when the first priority controller avoids collisions, and the second priority follows waypoints. The collision avoidance controller would then be invoked every time there are objects in the vicinity of the UAV, and the waypoint controller would be executed the rest of the time.

IV.B. Task that are ordered

Task that needs to be performed in a given order, such as taxing to the runway, takeoff, and initial climb to cruise altitude, are structured in a straightforward manner using the sequence operator and a set of actions (or subtrees), as in Figure 2.

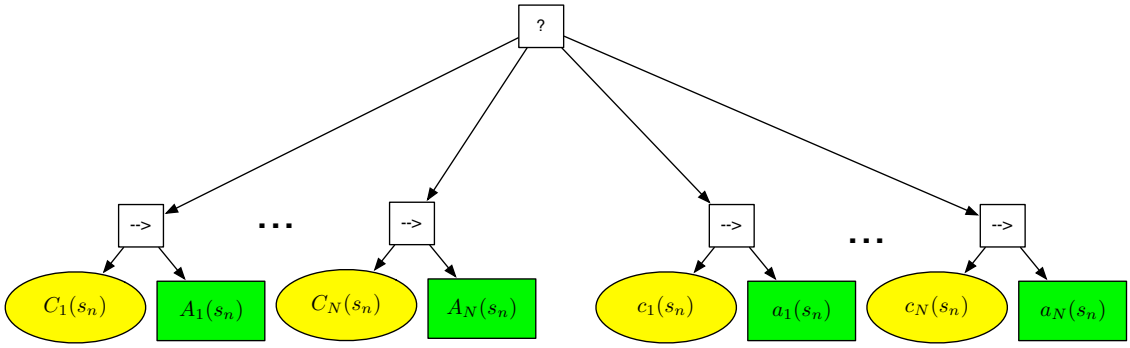

Figure 6. A general HDS written as a BT. The first N subtrees check if a transition is made in the HDS, and the last N subtrees take care of the continuous dynamics.

IV.C. A general HDS written as a BT

Technically, it is clear that all HDS can be written as BTs due to the fact that we have no restrictions of what kind of computations are allowed inside an Action. Thus the whole HDS can be included in one Action, and a BT can be created out of this single Action. However, the structure of the HDS can actually be translated into a BT without hiding everything in a single block. Let $s_n = (q_n, x_n)$ be the current state and define the (boolean) conditions

$$C_i(s_n) = (q_n == b_i) \text{ AND } (x_n \in A_{b_i}), \quad (1)$$
$$c_i(s_n) = (q_n == b_i), \quad (2)$$

and the Actions

$$A_i(s_n): \quad s_{n+1} = G_{b_i}(x_n), \quad (3)$$
$$a_i(s_n): \quad s_{n+1} = (f_{b_i}(x_n), b_i). \quad (4)$$

Then we define the BT as illustrated in Figure 6. The root is a Selector, connected to $2N$ subtrees composed of one Sequence, one Conditions and one Action. The N first subtrees use the Conditions C_i, checking the transition condition of the HDS and Action A_i executing those transitions, while the last N subtrees handle the continuous dynamics of the HDS using Condition c_i and actions a_i.

IV.D. A Combat UAV Controller

The controller of an air-to-air combat UAV might look something like the BT shown in Figure 7. As can be seen, the root node is a Selector with prioritized children from left to right. The highest priority (left) subtree is then composed of a Sequence first checking the Condition *Collision Warning* and then performing the Action *Avoid Ground*. The second priority subtree similarly checks if there is a *Missile Warning* and if so performes an *Evasive Maneuver*. The following subtree is somewhat more complex, it starts by checking

if there are *Enemies in Range* and if so first checks if *Odds OK*, i.e. if the number and estimated status of the enemies are comparable to or lower than the number and status (including fuel) of the friendly forces, and then proceeds into air-to-air combat. Otherwise a *Disengage* Action is invoked. If no enemies are in range, the next subtree checks a *Bingo Fuel* condition, i.e., if the current amount of fuel is not enough to fly home and land without violating the prescribed safety margin, if so, the *Fly Home* Action is invoked. Finally, if none of the above subtrees were applicable, the UAV keeps doing its routine duty, i.e. performing a Patrolling, Strike or Surveillance mission, and when that mission is complete the *Fly Home* Action is invoked, returning the UAV to its base.

As can be seen in the above example, it is quite easy to insert additional actions anywhere in the BT, at a fairly low cost in terms of increased complexity. Furthermore, the reliability of critical system functions, such as collision avoidance in Figure 7, is obviously not influenced by adding any other action or subtree at at branch of lower priority under the same selector.

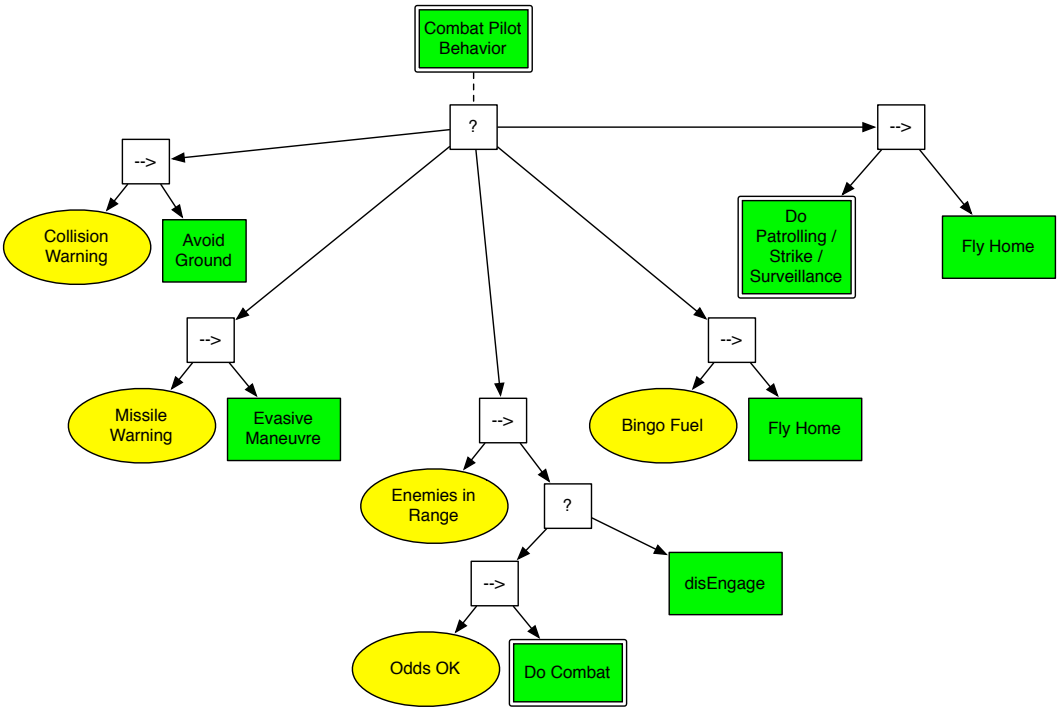

Figure 7. The BT of an air-to-air combat UAV

V. Conclusion

In this paper, we have argued that Behavior Trees (BT) is a formalism that presents many advantages, in terms of modularity, reusability and complexity, to the designer of UAV guidance and control systems. This is mainly due to the fact that BTs make the transitions of a HDS implicit in the tree structure, as well as *two way*, instead of explicitly encoded in a *one way* transition function. The implicit two way transitions substantially increase modularity, which in turn makes design and re-design much simpler. Finally, it also increases the readability of the algorithm, as discussed by Dijkstra in his paper regarding the Goto-statement.

References

[1]Kriegel, M., Brüggenwirth, S., and Schulte, A., "Knowledge Configured Vehicle – A layered artificial cognition based approach to decoupling high-level UAV mission tasking from vehicle implementation properties," *AIAA Guidance, Navigation, and Control Conference, Portland, Oregon*, 2011.

[2]Soto, Nava, and Alvarado, "Drone Formation Control System Real-Time Path Planning," *AIAA Infotech and Aerospace*

Conference, 2007.

[3]Dicheva, S. and Bestaoui, Y., "Route Finding for An Autonomous Aircraft," *49th AIAA Aerospace Sciences Meeting, Florida*, 2011.

[4]Enomoto, K., Yamasaki, T., Takano, H., and Baba, Y., "Guidance and Control System Design for Chase UAV," *AIAA Guidance, Navigation and Control Conference, Hawaii*, 2008.

[5]Keviczky, T. and Balas, G., "Software-Enabled Receding Horizon Control for Autonomous UAV Guidance," *AIAA Guidance, Navigation, and Control Conference*, 2005.

[6]Lim, C., Baumgarten, R., and Colton, S., "Evolving Behaviour Trees for the Commercial Game DEFCON," *Applications of Evolutionary Computation*, edited by D. Chio, Vol. 6024 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2010, pp. 100–110.

[7]Isla, D., "Handling complexity in the Halo 2 AI," *Game Developers Conference*, 2005.

[8]Branicky, M., "Introduction to hybrid systems," *Handbook of networked and embedded control systems*, 2005, pp. 91–116.

[9]Huber, M., *A hybrid architecture for adaptive robot control*, Ph.D. thesis, University of Massachusetts at Amherst, 2000.

[10]Fierro, R., Das, A., Kumar, V., and Ostrowski, J., "Hybrid control of formations of robots," *ICRA. IEEE International Conference on Robotics and Automation*, Vol. 1, IEEE, 2001, pp. 157–162.

[11]Conner, D., Rizzi, A., and Choset, H., "Composition of local potential functions for global robot control and navigation," *IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS)*, Vol. 4, IEEE, pp. 3546–3551.

[12]Connell, J., "SSS: A hybrid architecture applied to robot navigation," *IEEE International Conference on Robotics and Automation*, IEEE, 1992, pp. 2719–2724.

[13]Egerstedt, M., "Behavior based robotics using hybrid automata," *Hybrid Systems: Computation and Control*, 2000, pp. 103–116.

[14]Perez, D., Nicolau, M., O'Neill, M., and Brabazon, A., "Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution," *Applications of Evolutionary Computation*, 2011, pp. 123–132.

[15]Bojic, I., Lipic, T., Kusek, M., and Jezic, G., "Extending the JADE Agent Behaviour Model with JBehaviourTrees Framework," .

[16]Champandard, A., "Understanding Behavior Trees," *AiGameDev. com*, Vol. 6, 2007.

[17]Dijkstra, E. W., "Letters to the editor: go to statement considered harmful," *Commun. ACM*, Vol. 11, March 1968, pp. 147–148.

[18]Brooks, R., "Elephants don't play chess," *Robotics and autonomous systems*, Vol. 6, No. 1-2, 1990, pp. 3–15.