Reducing Code Complexity in Hybrid Control Systems

Louise. A. Dennis*, Michael Fisher*, Nicholas K. Lincoln**, Alexei Lisitsa*, Sandor M. Veres**

* Department of Computer Science, University of Liverpool, UK
e-mail: L.A.Dennis@liverpool.ac.uk, mfisher@liverpool.ac.uk, A.Lisitsa@liverpool.ac.uk

** School of Engineering, Univeristy of Southampton, UK
e-mail: S.M.Veres@soton.ac.uk, N.K.Lincoln@soton.ac.uk

Abstract

Modern control systems are limited in their ability to react flexibly and autonomously to changing situations by the complexity inherent in analysing situations where many variables are present.

We present an architecture based on a combination of agent languages and hybrid systems for managing high level decisions in such systems. A preliminary case study suggests that the complexity of the code of such a system increases much more slower in the face of inreasing complexity of the underlying system, than in a more traditional approach based on finite state machines.

1 Introduction

Modern control systems are limited in their ability to react flexibly and autonomously to changing situations by the complexity inherent in analysing situations where many variables are present.

We are particularly interested in the control of satellite systems. Consider the problem of a single satellite attempting to maintain a geostationary orbit. Current satellite control systems maintain orbits using feedback controllers. These implicitly assume that any errors in the orbit will be minor and easily corrected. In situations where more major errors occur, e.g. caused by thruster malfunction, or where changes in mission priorities occur, it is desirable to modify or change the controller or other aspects of the physical system. The complexity of the decision task is a challenge to the imperative programming approach.

There is a long standing tradition, pioneered by the PRS system [14], of using agent languages (and other logic programming approaches – e.g. [22]) to control and reason about such systems. We therefore approach the problem from the perspective of rational agents and hybrid systems. We consider a satellite to be an *agent* which consists of a discrete (rational decision making) engine and a continuous (calculation) engine. The rational engine uses the *Belief-Desire-Intention* (BDI) theory of agency [20] to both generate discrete abstractions from continuous data and to use those abstractions to govern the high level decisions about when to generate new feedback controllers or modify hardware. The continuous, calculational engine is used to derive controllers, perform predictive simulations and to calculate information from continuous data which can be used in forming abstractions.

1.1 BDI Agents

We view an agent as an *autonomous* computational entity making its own decisions about what activities to pursue. Often this involves having goals and communicating with other agents in order to accomplish these goals [23]. *Rational agents* make decisions in an explainable way, making it easier for debugging, diagnostic and monitoring processes to account for an agent's actions at a high level.

Following BDI theory, we often describe each agent's *beliefs* and *goals* which in turn determine the agent's *intentions* (a set of actions it intends to take). Such agents make decisions about what action to perform next, given their current beliefs, goals and intentions.

1.2 Control Systems

A fundamental component of control systems technology is the *feedback controller*. This measures, or estimates, the current state of a system through a dynamic model and produces subsequent feedback/feedforward control signals. In many cases difference/differential equations can be used to elegantly manage the process. These equations of complex dynamics make changes to the input values of sub-systems and monitor the outcomes on various sensors.

We are investigating systems which require some decision making system to be integrated with the feedback controller. It is by now well established that using a separate *discrete* and *logical* decision making process for this aspect is preferable to greatly extending the basic control system [1, 2]. Overall systems with these characteristics are often referred to as *hybrid control systems*, in that they integrate discrete, logical decision processes with physical system dynamics.

Unfortunately, the control of hybrid systems using traditional programming methods can become increasingly unwieldy. Often the decision process is represented as an inflexible tree (or graph) of possible situations. Execution then involves tracing through a branch of this tree which matches the current situation and then executing the

feedback controller (or making other changes to the system) found at the relevant leaf of the tree.

Programming these decisions from state to state is often time-consuming and error prone and can lead to the duplication of code where the same actions need to be taken in several slightly different situations.

2 Architecture

Our aim is to produce a hybrid system embedding existing technology for generating feedback controllers and configuring satellite systems within a decision making part based upon agent technologies and theories. The link is to be controlled by an *abstraction layer* which converts data between continuous values appropriate for real time control and discrete values appropriate for reasoning.

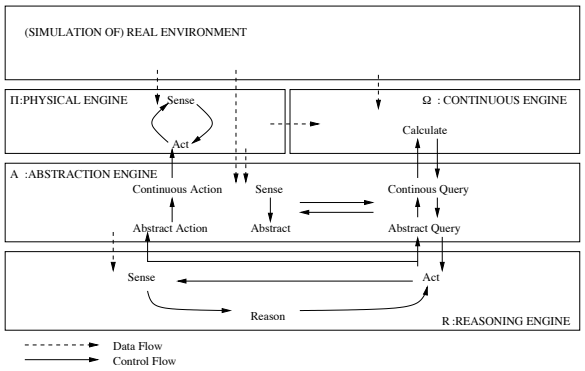

Figure 1. Hybrid Agent Architecture

Figure 1 shows an architecture for our system. Real time control of the satellite is governed by a traditional feedback controller drawing its sensory input from the environment. This forms a *Physical Engine* (Π). This engine communicates with an agent architecture consisting of an *Abstraction Engine* (A) that filters and discretizes information. To do this A may a use a *Continuous Engine* (Ω) to make calculations involving the continuous information. Finally, the *Rational Engine* (R) contains a "Sense-Reason-Act" loop. Actions involve either calls to the Continuous Engine to calculate new controllers (for instance) or instructions to the change hardware configuration of the Physical Engine. These instructions are passed through the Abstraction Engine for reification.

In this way, R is a traditional BDI system dealing with discrete information, Π and Ω are traditional control systems, typically generated by MatLab/Simulink, while A provides the vital "glue" between all these parts.

3 Scenario: Maintaining Geostationary Orbit with Thruster Failure

A Simulink model of a satellite in geostationary orbit [18], was implemented. MatLab functions, composed via sEnglish [21], were made available to the continuous part of the agent. These functions were capable of completing trivial computations such as whether a given set of coordinates were within an acceptable distance of the satellite's desired orbital position, `comp_distance`, as well as more complex processing tasks such as computing a fuel optimal return path to a desired orbital position, `plan_approach_to_centre`.

The satellite was simulated with three thrusters (X, Y and Z), each of which contained two fuel lines; one of these fuel lines was redundant enabling the agent to switch fuel lines if the other was ruptured (detectable by a drop in pressure on the output fuel line). Redundant thrusters (up to five in the X direction) were also introduced, allowing the agent to switch to a redundant thruster if both fuel lines appeared broken.

Controls were made available in the Physical Engine which could send a particular named *activation plan* to the feedback controller, `set_control`, switch thrusters on and off, `set_x1_main`, `set_x2_main`, `set_y1_main`, etc., control the valves that determined which fuel line was being utilised, `set_x1_valves`, etc. and change the thruster being used in any direction `set_x_bank`, etc.

A BDI-style language was developed, based on the Gwendolen programming language [8] and used to program both the abstraction and reasoning engines. A key feature of this style of programming is that it allows reactions to several events, or circumstances, to be handled in an interleaved fashion, so the system can continue to monitor incoming data while, for instance, calculating a new path and can react to, say, the malfunction of two thrusters without needing to specify the precise order in which the malfunctions are dealt with.

The agent programming language was implemented in Java and communication between the MatLab and Java parts of the system was managed via sockets. MatLab sent information over the socket consisting of a tag followed by a stream of numbers, on the Java side this is constructed into a predicate to be used by the abstraction engine.

A semantics for interaction between the components of the system was implemented, based on that outlined in [9]. This included a set of *shared beliefs* that were accessible from both the abstraction and reasoning engines.

3.1 The Abstraction Engine

The Abstraction Engine code consisted of two parts. There was a generic part which was used in all examples in the case study and a specific part which was modified each time a new thruster was added.

A, slightly tidied up, version of the generic code follows:

Code fragment 3.1 Geostationary Orbit:Abstraction Engine

```
+location(L1, L2, L3, L4, L5, L6) : {B bound_info(V1)} ←
        calc(comp_distance(L1, L2, L3, L4, L5, L6), Val),
        +bound_info(Val);

+bound_info(in) : {B proximity_to_centre (out)} ←
        −bound_info(out),
        −Σ proximity_to_centre (out),
        +Σ proximity_to_centre (in);
```

```
+bound_info(out) : {B  proximity_to_centre (in)} ←                10
        −bound_info(in),                                          11
        −Σ  proximity_to_centre (in),                             12
        +Σ  proximity_to_centre (out);                            13
                                                                  14
+!maintain_path : {B  proximity_to_centre (in)} ←                 15
        run( set_control (maintain));                             16
+!execute(P)     : {B  proximity_to_centre (out)} ←               17
        run( set_control (P));                                    18
                                                                  19
+! plan_approach_to_centre (P) :                                  20
   {B  location (L1, L2, L3, L4, L5, L6)} ←                       21
   calc( plan_approach_to_centre (L1, L2, L3, L4, L5, L6), P),    22
        +Σ  plan_approach_to_center (P);                          23
                                                                  24
−broken(X) :                                                      25
   {B  thruster_bank_line (X, N, L),                              26
    B  thruster (X, N, C, V, P), P1 < 1} ←                        27
        +Σ(broken(X));                                            28
                                                                  29
+ thruster (X, N, C, V, P):                                      30
   {˜ B broken(X),                                                31
    B  thruster_bank_line (X, N, L), P1 < 1} ←                    32
        +Σ broken(X);                                             33
+ thruster (X, N, C, V, P):                                      34
   {B broken(X),                                                  35
    B  thruster_bank_line (X, N, L), 1 < P1} ←                    36
        −Σ broken(X).                                             37
                                                                  38
+! change_fuel_line (T, 1) :                                      39
   {B  thruster_bank_line (T, B, 1)} ←                            40
        run( set_valves (T, B, off, off, on, on)),                41
        −Σ  thruster_bank_line (T, B, 1),                         42
        +Σ  thruster_bank_line (T, B, 2),                         43
        −Σ broken(T);                                             44
+!change_bank(T) : {B  thruster_bank_line (T, B, L)} ←           45
        B1 is B + 1;                                              46
        run(set_bank(T, B1)),                                     47
        run(set_main(T, B, off)),                                 48
        run(set_main(T, B1, on)),                                 49
        −Σ  thruster_bank_line (T, B, L),                         50
        +Σ  thruster_bank_line (T, B1, 1),                        51
        −Σ broken(T);                                             52
```

We use a standard BDI syntax: $+b$ indicates the addition of a belief; $!g$ indicates a goal, g, and $+!g$ the commitment to the goal. A plan $e : \{g\} \leftarrow b$ consists of a trigger event, e, a guard, g, which must be true before the plan can be executed and a body b which is executed when the plan is selected. The use of $\mathcal{B}b$ in a plan guard indicates a test that b is believed by the agent.

In addition to regular BDI syntax we use $+_\Sigma b$ and $-_\Sigma b$ to indicate the addition and removal of *shared beliefs* which are used by both the Abstraction and the Reasoning engines. The actions `calc` and `run` trigger communication with the MatLab processes. `calc` requests the calculation of a value from the Continuous Engine achieved by calling an `M-file` with the appropriate name, while `run` activates controls in the physical engine.

The Abstraction engine performs two functions, converting the data from the Physical and Continuous Engine into a form suitable for reasoning (e.g. the location information is converted into the abstract judgment of whether the satellite is within bounds in lines 1–13 and judgments over whether a thruster is broken are made in lines 30–37) and converting requests from the reasoning engine into instructions for the physical or continuous engine (e.g. the `change_bank` requestion is converted into a sequence of three `run` instructions in lines 45–52).

Requests from the reasoning engine are modelled as goal commitments. So `+!change_fuel_line(T, 1)` indicates that the abstraction engine has received a request from the reasoning engine to change a fuel line.

For example the code in lines 45–52 describes how to change a thruster in bank T following a request from the reasoning engine, if the thruster used by the bank is believed to be B. The Physical engine is instructed to set the bank to thruster $B+1$ (`set_bank(T, B1)`) then switch off thruster B, switch on thruster $B+1$, and then change the shared beliefs so it no longer beliefs that the bank is using thruster B but believes the bank is using thruster $B+1$. At the same time it removes any beliefs that the thruster is broken.

The code in fragment 3.1 was the same in all versions of the system, but for each additional thruster we had to add code to convert from the input data about that thruster to a more generic predicate. Below is the code used for the 1st thruster in the X bank.

Code fragment 3.2 Geostationary Orbit:X Thruster 1 Code

```
+ xthruster1 (L11, L21, P1, Volt1, Curr1):                        1
   {˜ B  thruster (x, 1, L2, L1, P, V, C)} ←                      2
        + thruster (x, 1, L11, L21, P1, Volt1, Curr1);            3
+ xthruster1 (L11, L21, P1, Volt1, Curr1):                        4
   {B  thruster (x, 1, L2, L1, P, V, C)} ←                        5
        − thruster (x, 1, L2, L1, P, V, C),                       6
        + thruster (x, 1, L11, L21, P1, Volt1, Curr1);            7
```

As can be seen the data coming from the physical engine tags each thruster's data with a label specific to the thruster (`xthruster1` in this case) but the abstraction and reasoning engine wish to apply the same reasoning to all thrusters and so convert this into a predicate, `thruster`, that is parameterised by the bank (`x` in this case) and the thruster within that bank (`1` in this case). Two cases are needed depending on whether or not the abstraction engine already has a belief about this thruster.

3.2 The Reasoning Engine

The reasoning engine code is as follows and remained the same for any number of redundant thrusters:

Code fragment 3.3 Geostationary Orbit: Reasoning Engine

```
+ proximity_to_centre (out) : {⊤} ←                               1
        − proximity_to_centre (in),                               2
        +! get_to_centre ;                                        3
+ proximity_to_centre (in) : {⊤} ←                                4
        − proximity_to_centre (out),                              5
        perform( maintain_path );                                 6
                                                                  7
+! get_to_centre  : {B  proximity_to_centre (out)} ←              8
        query( plan_approach_to_centre (P)),                      9
        perform(execute(P)),                                      10
        −Σ  plan_approach_to_centre (P);                          11
                                                                  12
+broken(X): {B  thruster_bank_line (X, N, 1)} ←                   13
        perform( change_fuel_line (X, N));                        14
+broken(X): {B  thruster_bank_line (X, N, 2)} ←                   15
        perform(change_bank(X, N));                               16
```

We use the same syntax as we did for the Abstraction Engine. Here the actions, '`perform`' and '`query`', request that the Abstraction Engine forward an instruction to the

Reasoning engine or a calculation to the continuous engine (respectively).

The architecture lets us represent the high-level decision making aspects of the program in terms of the beliefs and goals of the agent and the events it observes. So, for instance, when the Abstraction Engine observes that the thruster line pressure has dropped below 1, it asserts a shared belief that the thruster is broken. When the Reasoning Engine observes that the thruster is broken, it then either changes fuel line, or thruster bank. This is communicated to the Abstraction Engine which then sets the appropriate valves and switches.

4 Comparison to Traditional Hybrid Control Systems

As well as constructing a BDI style controller for thruster malfunction we constructed a traditional finite state machine controller using MatLab's stateflow package. As we added additional redundant thrusters we were able to compare how the size of the code increased in the two systems, and hence the programming burden and probability of error increased.

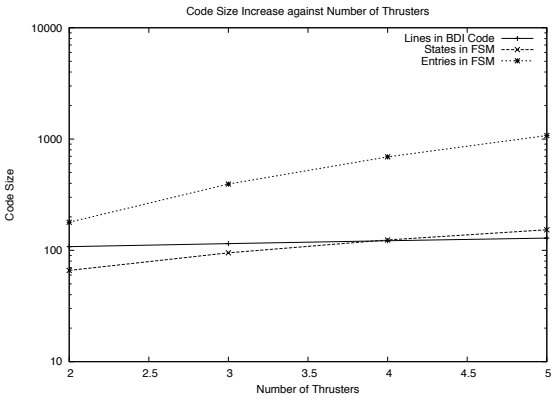

Figure 2. Comparing how Code complexity scales (Logscale on y axis)

As can be seen the increase in code size for the BDI system is linear (the additional seven lines of code show in fragment 3.2 which convert specific thruster predicates into more general predicates) while the FSM increases more than linearly as more redundant thrusters are added.

5 Future Work

The work on hybrid agent systems with declarative abstractions for autonomous space software is only in its initial stages and considerable further work remains to be investigated.

5.1 Further Case Studies.

We are keen to develop a repertoire of case studies which will provide us with benchmark examples upon which to examine issues such as more sophisticated reasoning tasks, multi-agent systems, forward planning, verification and language design.

We have already started work on a more sophistacted study involving a group of satellites attempting to maintain or change formation in low Earth orbit.

5.2 Custom Language.

At the moment the BDI language we are using for the Abstraction Engine is not as clear as we might like and it may prove that the BDI paradigm is not appropriate for the abstraction task, which is not one based around decision making. We are investigating the use of stream processing technologies (from e.g. [3, 15]) and the use of temporal logic statements as a better mechanism for forming abstractions.

We are also interested in investigating other programming languages for the Reasoning Engine – e.g. languages such as *Jason* [6] or 3APL [7] are similar to the one we employ, but better developed and supported. Alternatively it might be necessary to extend the custom language with, for instance, the concept of a *maintain* goal. Much of a satellite's operation is most naturally expressed in terms of *maintaining* a state of affairs (such as a remaining on a particular path).

5.3 Planning and Model Checking.

At present the `M-file` employed to create a new controller that will return the satellite to the desired orbit uses a technique based on hill-climbing search [17]. We are interested in investigating temporal logic and model-checking based approaches to this form of planning for hybrid automata based upon the work of Kloetzer and Belta [16]. We are also interested in the use of simulation as a form of predictive modelling that can assist in decision making.

Model checking techniques also exist [5] for the verification of BDI agent programs which could conceivably be applied to the Reasoning Engine. Abstraction techniques would then be required to provide appropriate models of the continuous and physical engines and it might be possible to generate these automatically from the abstraction engine.

There is also a large body of work on the verification of hybrid systems [1, 12] which would allow us to push the boundaries of verification of such systems outside the limits of the Reasoning Engine alone.

5.4 Multi-Agent Systems.

We are interested in extending our work to multi-agent systems and groups of satellites that need to collaborate in order to achieve some objective. For instance, there are realistic scenarios in which one member of a group of satellites loses some particular functionality meaning that its role within the group needs to change and the group itself needs to find a new formation. We believe this provides an interesting application for multi-agent work on groups, teams, roles and organisations [10, 13, 11, 19], and also provides an interesting test bed for using forward

planning and simulation techniques to inform the decision making process.

5.5 Implementation in Hardware

We hope to evaluate our software on a physical satellite simulation environment developed at the University of Southampton. Although this environment constrains the satellites to operate with 5 degrees of freedom, it allows the software to be tested in a real physical environment and to assess its ability to handle decision-making outside of an entirely virtual implementation. This will be of particular interest when evaluating the predictive simulation aspects of the system since the ability to handle differences between the simulated result of some action and the actual result of some action will be a key requirement.

6 Conclusion

This paper has presented a hybrid-style achitecture for the control of satellite systems.

A simple case study is presented demonstrating how this style of programming copes with the increasing complexity of the underlying system better than more traditional approaches to hybrid system programming based on on Finite State Systems. This reduced complexity follows from the systems ability to make use of parameterised subtasks and to specify that sub-tasks are triggered by specific system states and to allow several sub-tasks to be executed in an interleaved fashion.

6.1 Acknowledgment

Work funded by EPSRC grants EP/F037201/1 and EP/F037570

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[2] R. Alur, T. A. Henzinger, G. Lafferriere, George, and G. J. Pappas. Discrete abstractions of hybrid systems. In *Proceedings of the IEEE*, pages 971–984, 2000.

[3] A. Arasu, S. Babu, and J. Wisdom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report 2003-67, Stanford, 2003.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.

[5] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 69–78, L'Aquila, Italy, September 2008.

[6] R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the Golden Fleece of Agent-Oriented Programming. In Bordini et al. [4], chapter 1, pages 3–37.

[7] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming Multi-Agent Systems in 3APL. In Bordini et al. [4], chapter 2, pages 39–67.

[8] L. A. Dennis and B. Farwer. Gwendolen: A BDI Language for Verifiable Agents. In B. Löwe, editor, *Logic and the Simulation of Interaction and Reasoning*, Aberdeen, 2008. AISB. AISB'08 Workshop.

[9] L. A. Dennis, M. Fisher, N. K. Lincoln, A. Lisitsa, and S. M. Veres. Declarative abstractions for agent based hybrid control systems. In A. Omicini, S. Sardina, and W. Vasconcelos, editors, *Declarative Agent Languages and Technologies (DALT'10)*, May 2010.

[10] J. Ferber and O. Gutknecht. A Meta-model for the Analysis and Design of Organizations in Multi-agent Systems. In *Proc. Third International Conference on Multi-Agent Systems (ICMAS)*, pages 128–135, 1998.

[11] M. Fisher, C. Ghidini, and B. Hirsch. Programming Groups of Rational Agents. In *Proc. International Workshop on Computational Logic in Multi-Agent Sytems (CLIMA)*, volume 3259 of *LNAI*. Springer, November 2004.

[12] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[13] J. F. Hübner, J. S. Sichman, and O. Boissier. A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In *Proc. Sixteenth Brazilian Symposium on Artificial Intelligence (SBIA)*, pages 118–128, London, UK, 2002. Springer.

[14] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992.

[15] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Aetintemal, M.Cherniack, R. Tibbetts, and S. Zdonik. Towards a Streaming SQL Standard. In *Proceedings of Very Large Databases*, pages 1397–1390, Auckland, New Zealand, August 2008.

[16] M. Kloetzer and C. Belta. A Fully Automated Framework for Control of Linear Systems From Temporal Logic Specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

[17] N. Lincoln and S. Veres. Components of a Vision Assisted Constrained Autonomous Satellite Formation Flying Control System. *International Journal of Adaptive Control and Signal Processing*, 21(2-3):237–264, October 2006.

[18] M.J. Sidi. *Spacecraft Dynamics and Control: A Practical Engineering Approach*. Cambridge University Press, 2002.

[19] D. V. Pynadath, M. Tambe, N. Chauvat, and L. Cavedon. Towards Team-Oriented Programming. In *Intelligent Agents VI — Proc. Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, volume 1757 of *LNAI*, pages 233–247. Springer, 1999.

[20] A. S. Rao and M. Georgeff. BDI Agents: From Theory to Practice. In *Proc. First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, USA, June 1995.

[21] S.M. Veres. *Natural Language Programming of Agents and Robotic Devices: Publishing for Humans and Machines in sEnglish*. SysBrain Ltd, 2008.

[22] R. Watson. An Application of Action Theory to the Space Shuttle. In G. Gupta, editor, *Proceedings of Practical Aspects of Declarative Languages, First International Workshop (PADL '99)*, volume 1551 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 1999.

[23] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.