HarTAC– The Harvard TAC SCM'03 Agent

HarTAC - The Harvard TAC SCM '03 Agent

Rui Dong, Terry Tai and Wilfred Yeung
Harvard College,
Cambridge, MA 02138
{ruidong,tai,wyeung}@fas.harvard.edu

David C. Parkes
Division of Engineering and Applied Sciences,
Harvard University,
33 Oxford Street, Cambridge MA 02138
parkes@eecs.harvard.edu

Abstract

The Trading Agent Competition (TAC) is an annual event in which teams from around the world compete in a given scenario concerning the trading agent problem. This paper describes some of the key features and strategies employed by HarTAC, the Harvard University TAC SCM 2003 agent. HarTAC was built to be a moderately aggressive, production-oriented agent, combining a semi-conservative component purchasing strategy with a highly aggressive cycle-based sell side.

1. Introduction

The Trading Agent Competition (TAC) is an annual event in which teams from around the world compete in a given scenario concerning some form of the "trading agent problem." The generalized trading agent problem essentially asks the question: given a market situation with specific rules, how does one act to buy, sell, and produce goods to maximize expected profits? This is of particular interest to the AI community given automated trading and the increasing number of electronic markets. TAC was designed to spur research in this topic as well as "present difficult decision problems and admit a wide variety of potential bidding and negotiation strategies."[1] TAC Supply-Chain Management (SCM) and TAC Classic were scenarios constructed to elicit different strategies for solving the trading agent problem.

In this paper, we provide an in-depth discussion of HarTAC, the Harvard TAC SCM agent for TAC'03. An interesting characteristic of the TAC environment was that many teams ran their programs on public game servers, allowing knowledge to be shared between teams in the months leading up to the competition. Consequently, our agent's design was greatly influenced by this shared knowledge. Ultimately, we chose a moderately aggressive strategy for our agent based on aggressive production with added safeguards designed to protect ourselves against unfavorable game conditions. Central to our design is a steady-state analysis, with a control loop to achieve the desired factory production cycles. In the end, our agent made it to the semi-finals of the competition before being eliminated.[2]

1.1. TAC SCM BACKGROUND

In the TAC SCM scenario, contestants are presented with a virtual business world in which each contestant manages a PC manufacturing company. Six teams compete against one another in any given 220-day game, and the team whose company makes the most profit wins the game. In managing their company, each team must purchase components, schedule computer production at its factory, and manage the sale of computers to customers. Starting with no savings, no inventory, and no pending orders, each agent is endowed with limitless credit and its own factory.

Supply contracts are arranged by sending RFQ bundles (requests for quotes on a certain quantity of a type of component) to component manufacturers (automated global agents run by the game itself). These manufacturers respond with information on price and availability of the quantity requested, which the agent can then decide to either order or decline. Customer demands are similarly handled by receiving a different type of RFQ bundle from automated customers, which request computers by a specified due date. The agent responds with an offered price, and the customer determines the winner through a first price procurement auction. The winning agent must then produce and deliver the ordered goods by the specified due date to be paid or otherwise incur monetary penalties. Finally, the agent must submit a production schedule each day that indicates how many of each type of computer to build. Each computer type requires a different number of cycles to construct (besides

1 TAC Website, http://www.sics.se/tac/

2 In the semi-finals, we ran into technical difficulties with the computer we were using to run the agent that contributed to our loss. This will be discussed later on in the paper.

just different components), and the total number of cycles used per day may not exceed 2000.

In summation, on any given day, an agent needs to make the following decisions:

1. Which RFQs to issue to which providers
2. Which providers' offers (received in response to the RFQs sent the previous day) to accept
3. Which products to produce using the 2000 production cycles and the components in inventory
4. Which finished products to ship to customers
5. Which customer RFQs (received daily) to answer, and the price to be given to the customer

The agent that is able to make the most money at the end of the game is declared the winner.[3]

1.2. TAC SCM DESIGN AND METHODOLOGY

As seen in the description above, the TAC SCM scenario presents an agent with limited knowledge about its environment and other competitors. This partial knowledge, coupled with a large number of possible actions each day, makes the decision problem very complex. There is an intractable number of possible situations in which an agent might find itself, given what it knows each day about inventories, bankroll, prices, and RFQs. An agent also has a large number of possible actions each day.

With that in mind, we first modeled the agent's problem as one of control in a small, finite state space for which we specified rules to direct our agent's behavior. Control theory provides a straightforward model for dynamic systems, and we found this to be an intuitive simplification for the TAC scenario [1]. In truth, several of our state-specific mechanisms are modeled as control mechanisms as well. Our approach therefore shares some features with the methods of Kiekintveld et al. [5]

Our state space is designed with the assumption that there exists some steady state in which our agent will remain for most of the game. We define the steady state to be the game situation where we are in the best position to make a profit (a more precise definition will be given later). In that case, we construct our agent's behavior to maximize performance in this steady state and to maximize the time spent in the steady state. Meanwhile, we define states for the situations where our agent is knocked out of the steady state for various reasons (i.e. changing demand, changing supply, actions of other agents), and for each such state, we outline a set of actions for our agent to return to the steady state. We also need to specify states for start game conditions, which are designed with the goal of quickly reaching the steady state and of possibly preventing competitors from reaching their steady states (i.e. sabotage). Finally, we define states for end game conditions whose purpose is to maximize final revenue by selling off the remainder of the parts and products from the time spent in the steady state.

Since throughout the rest of the paper we refer to variables determined via experimentation, it is also important to note that our experimental methodology consisted of running numerous trials both on the public servers and on our own servers playing against agents that we designed to represent specific strategies. In analyzing our sell-side mechanism, for instance, we logged all of the prices at which orders were won when playing against a wide range of different agents. We then used statistical analysis (mostly simple linear regression) on the data from these games to determine the constants.

2. Organization of the Agent States

Central to the design of the HarTAC agent is the notion of the *steady state*. This defines an operating condition in which we believe we can make the most profit. Most of our behavior is geared towards achieving this state, towards remaining in this state, and towards maximizing our profit while in the steady state. We define the steady state as follows:

- Positive profits (calculated as the difference between current customer demand prices and current inventory costs) can be obtained by fulfilling orders.
- HarTAC has spent less than five days straight incurring penalties for unfulfilled orders.
- HarTAC possesses some minimal level of components (to be explained later, Section 4).
- We have not yet hit our end game state conditions (i.e. we have more than 45 days left in the game).

Under these conditions, we assume that it is to HarTAC's advantage to sell as many computers as possible, as constrained by our production capacity. This strong assumption is based upon empirical evidence that suggested that simple profit maximization of the price used in response to an RFQ yielded prices with winning probabilities that exceeded the factory capacity.

Due to our assumptions, the steady-state problem became a control problem, defined in terms of *targeted factory production*, to consistently win 2000 factory cycles worth of computers while maximizing the price for the orders won. In steady-state, our sell side uses a modified "proportional" control mechanism [1]. Thus, it is characterized by smooth functions determining the prices offered to customers through various changing factors indicating changing demand and level of competition. The buy side works to fulfill the needs of production and sales via a tiered control mechanism that maintains a good level of inventory, constantly adjusts its estimates to total needs of inventory, and

3 Interested readers should refer to the full specifications of the game in Sadeh et al. [6].

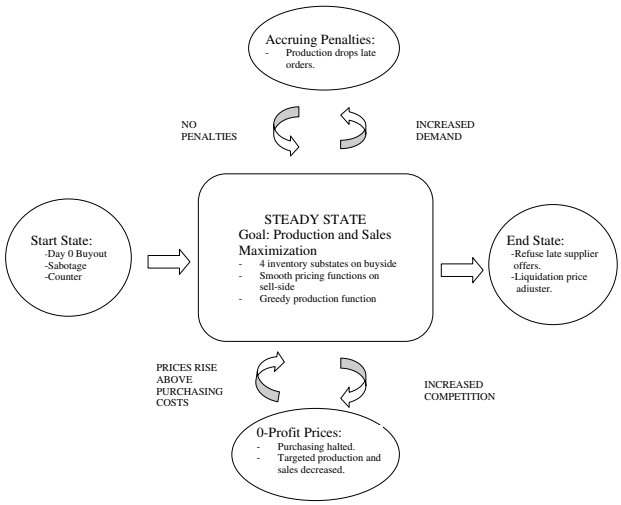

Figure 1. Organization of the agent states.

probes in hopes of finding good prices on components. Its behavior is determined by various sub-states of the inventory.

A host of other states work to maximize the time spent in the steady-state. These states are illustrated in Figure 1. The *start-state* procures the components necessary before passing onto steady-state. The *accruing-penalties* state alleviates the trend of penalties from the agent before passing it back to the steady-state, while the *negative-profit* state handles the situation when prices are pushed below costs. Finally, the *end-state* is an aggressive seller/conservative buyer strategy state where the agent tries to liquidate all inventory before exiting the game.

3. Steady State — Production

In order to sell as many computers as possible (i.e. 2000 cycles' worth daily), we need to be constantly producing computers. The steady-state production side is therefore a greedy algorithm: sell-side orders are processed in order of due date. Yet, should there be factory cycles available on a daily factory schedule, we fill the vacancy by producing computers uniformly at random across different types (subject to the availability of inventory).

4. Steady State — Purchasing

To fuel production, the purchasing of components was basically performed to maintain a reasonable quantity of all components in stock at all times.

In the short-run, a dearth of components is expensive as it stalls our production and sell side. On the other hand, due to the low inventory costs, a large amount of inventory does not hurt, as long as this inventory was bought at low costs. Inventory only becomes excessive when the agent cannot finish selling all of it by the end of the game. Considering these issues, our buy-side control problem focused on maintaining enough components in the short-run, determining the maximum inventory usable for the rest of the game, and maintaining the inventory level below this maximum.

We defined four substates for each possible component in our inventory: the *Critical substate* defines when it is urgent to purchase more inventory at any cost; the *Minimum substate* defines when the agent replaces inventory used while waiting for good deals to buy more components; the *Probing substate* defines when the agent merely probes for good deals; the *Maximum substate* defines when the agent stops buying totally, having surpassed the maximum level of inventory it can consume. These substates are illustrated in Figure 2. The goal substates for the agent are the Minimum and Probing substates. Note that the substates are component-specific. It is entirely possible for one particular component to be in the Critical substate while another component exists in Probing, or any combination thereof.

To define such substates, we experimentally defined three component constant thresholds—*critical, minimum*, and *maximum*. The four substates are placed in the four sections delineated by these three limits. The *critical* and *minimum* levels are fixed at 240 and 1600, respectively (for non-CPU components, and 120 and 800 for CPU components), while the maximum levels were determined during the game (as different games supported different maximum levels) and reduced during the course of a game as the end of the game approached.[4] Higher demand games allow more purchases with little risk, while lower-demand games might inflict severe punishment for the over-ordering of components, even at desirable prices. We set the maximum level, *Max* to be $\theta p_c d$, where p_c is the ratio of the average number of cycles HarTAC is managing to sell daily to the target of 2000, and d is the number of days left in the game. The constant, θ, is set to 160 units, which is roughly the expected number of any non-CPU component that the agent can use at maximal production in a single day. (There are twice as many options for CPUs, and we set the constant to 80 for CPU components). The role of p_c, the best estimate of how much the agent seems to be successfully selling, is to handle games with exceptionally low demand that prevented the agent from meeting the desired selling quota. The parameter was estimated during a game with a discounted moving average. When *Max* is less than the default value of 1600 for *Min*, then the *Max* thresh-

4 Roughly, at full capacity 2000 cycles/day would yield around 350 computers/day, and each computer can be constructed from 4 different CPU types and 2 different types of each component. This yields a full-capacity demand of roughly 80–85 for each CPU type, and 160–170 for each non-CPU component. Thus, a critical level of 240 for non-CPU components is a little less than the required capacity for around 1.5 days of continual production (similarly for the 120 critical level for CPU components).

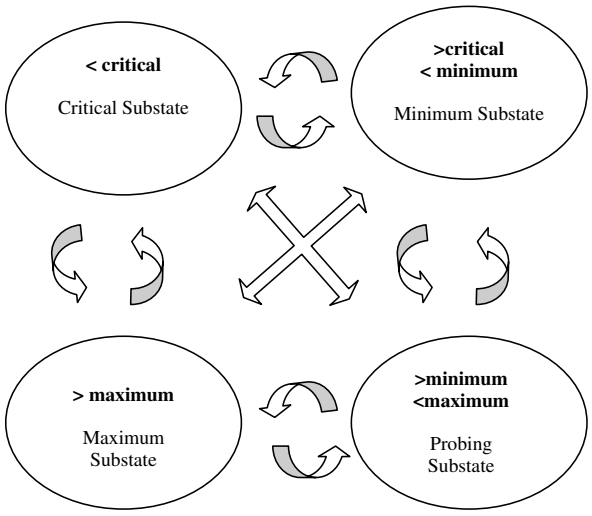

Figure 2. Organization of the inventory substates (one state machine for each component type).

Critical: Inventory ≤ 240	*Minimum*: Inventory [240, $\min\{1600, Max\}$]	*Probing*: Inventory [$\min\{1600, Max\}$, *Max*]	*Maximum*: Inventory $\geq Max$
$600(\max(1, r_i))$	$q_i(\max(1, r_i))$	$\max(0, 500(1-1/r_i))$	none

Table 1. Summary of the buy-side strategy for a component, which is defined according to the inventory substate (all numbers are for non-CPU components). Quantity r_i denotes the (*mean discounted historical price of component i*)/(*current price of component i*). Quantity q_i denotes the total quantity of component i sold (as a component of produced computers) on the current day.

old takes over.

The Probing substate uses small RFQs to try to identify low component pricing. Daily, we send RFQs to all suppliers requesting 1 component for 15, 25, and 75 days into the future. These three dates were chosen experimentally to offer a reasonable spread into the future. After we submit these RFQs, suppliers return on the following day with their estimated delivery dates and their prices. This pricing information is averaged over the days, with price information discounted by a multiplicative factor of 0.8 per elapsed day. From this, we can make informed decisions on when and from whom to buy components, based on their prices, delivery dates, and our current inventory stocks.

The quantity of each component purchased on a given day depends on the inventory substate for that component. The buy-side strategy is summarized in Table 1. Quantity r_i denotes the (*mean discounted historical price on component i*)/(*current price of component i*). Quantity q_i denotes the total quantity of component i sold (as a component within manufactured computers) on the current day. Notice that $r_i > 1$ when the current price is better than the historical average and $r_i < 1$ when the current price is worse than the historical average.

In the *Critical* substate we always purchase at least 600 units of each component, and more when $r_i > 1$, i.e. when the current price is better than historical prices. The quantity, 600, is chosen to represent the approximate number of components that allow continual production for 4 days (the corresponding purchase quantity is set to 300 for CPU components). In the *Minimum* substate we always purchase at least q_i units of each component, and more when $r_i > 1$ and current prices are better than historical prices. Given a target quantity, RFQs are constructed with the quantity divided equally across all suppliers with the required capability. In the *Probing* substate, we only purchase when $r_i > 1$ and current prices are better than historical prices, and we purchase from only the lowest-cost supplier. Moreover, we will submit an RFQ for *at most* 500 units of the component (c.f. 250 for non-CPU components) in the Probing substate. Finally, we never make any purchases in the *Maximum* substate, whatever the prices.

5. Steady State — Sell Side

For the general case, we wanted the sell side to sell as many computers as possible, given our constraints on production capacity. Aside from this, we desired to sell at the highest price possible. These desires and constraints were often in conflict, but in accordance to our production belief, we opted for selling more computers (up to 2000 cycles) instead of selling less computers at higher prices. Under the steady state assumptions of positive profits, reasonable inventory levels, and no accruing penalties, this desire to sell exactly 2000 factory cycles' worth of computers daily formed the cornerstone of the sell-side strategy.

The major question was then, how does one control the number of orders won and therefore the number of computers sold? We somehow needed to consistently regulate the orders won to 2000 cycles' worth daily. One popular scheme among existing agents at the time seemed to dictate bidding only on a fraction of all RFQs. Instead, we bid on all "sellable RFQs", and use price-based control to adjust the orders won to keep the associated number of cycles around 2000. The sellable RFQs are those for computers that we hold at least 80 in stock. Given that at least one other agent has submitted a bid on a particular RFQ, the winner is defined to be the agent that offered to complete the job for the least compensation. Assuming that there were

enough bidding agents at any time, an agent can use this scheme to control the orders won by simply adjusting its prices. Higher prices would inevitably lead to fewer orders won, while lower prices would increase the computers sold.

When successfully implemented, this method achieves high prices while winning around 2000 cycles' worth of orders per day. Had we raised prices any higher, we would have won too few orders, and vice versa. Also, our agent dynamically adjusts prices on a day-to-day basis, and proves sensitive to changes in the market and quick to respond. Thus, we reduced the price search space to a product-specific space, and used the same rules for each product irrespective of whether the product is in stock.

Our price-control loop adjusts prices separately, with one price for each computer type. We use two adjustors: the *cycle adjustor* and the *order-specific dampener*. The cycle adjustor translates into a variation of the classic "proportional" control mechanism. The order-specific dampener limits certain price changes and provides additional price stability.

The *cycle adjustor* uses the number of cycles' worth of orders won in the previous few days to adjust prices. Experimentally, we found that the chances of winning seemed to vary approximately according to the square of the price. Therefore, we took the derivative of our demand function with respect to price in order to determine the change in price required to effect the change in cycles won necessary to maintain the steady-state. This yields a linear function, and so we alter our bidding prices linearly with respect to the error in the number of cycles won in the past few days, with older data weight decaying exponentially with time. More technically, we used Δc, defined as the weighted average of the differences between the number of factory cycles associated with the orders won in recent days and 2000. Let t denote the current day. Δc is defined as:

$$\Delta c = \sum_{s=0}^{t-1} \alpha^{t-1-s}(2000 - c_s)$$

where c_s was the number of cycles won on day s, and α, our decay factor, was experimentally defined to be .5. The next day's bidding price for computer type i was then determined, as:

$$p_i^{t+1} = p_i^t[1 + (\beta \Delta c)] \quad (1)$$

where i denotes a computer type. Experimentally, we chose $\beta = .00015$.

Any day in which we overbid, we would automatically underbid the next few days, yielding an average of approximately 2000 cycles. The opposite was also true; those days during which we won few orders would generally be compensated for during the next few days, still yielding on average 2000 cycles.

The *order-specific dampener* acts on specific orders, and aims to obtain greater profits on RFQs disfavored by other agents. Throughout the game, we detect when we were winning certain types of RFQs with very high or very low probability. By categorizing all RFQs according to their product types, due dates, and penalties, we tracked our winning probability for each specific type of RFQ. We are always willing to raise prices if we always win particular orders, and we're willing to lower prices if we never win other order types. However, we only allowed price decreases on types of RFQs for which we win less than 10%. Symmetrically, only price increases were made on categories of RFQs that we win with a 90% probability or higher. In addition to allowing for different winning prices for different types of RFQs, this dampener had a more global effect of stabilizing our agent's pricing scheme. This stabilization prevented wild fluctuations in our prices (and in our RFQs won daily), allowing for the stability that preserved the steady-state.

6. Other States in the HarTAC System

Although our steady state controls act to smooth out shocks to the system, our agent is still subject to severe perturbations requiring more active corrective actions. To address such cases, we defined perturbation states to help the agent return more quickly to the steady state. In particular, we focused our attention on the states of *accruing penalties* and *negative-profits*. We also defined states for the *start game* and *end game* conditions. In these situations, we must be able to enter and exit the steady state quickly while still maximizing our profits.

6.1. Perturbation State 1: Accruing Penalties

Accruing penalties occur whenever the agent accepts cycles in far excess of 2000 in one particular day, but then on following days never undergoes a corresponding day of fewer than normal orders ($<$ 2000 cycles). As a result of this, we end up with more total orders than we can possibly fulfill. However, since the greedy delivery algorithms take into account only the dates of delivery, late penalties tended to spread *in perpetuum*, serving as a constant drain on our cash flow. This most often occurred directly as the result of dramatic upward shifts in demand or sudden decreases in competition.

However, determining when we're in this state is difficult. How do we know whether a few penalties are simply a temporary problem (that will be fixed by the price adjuster winning fewer orders), or actually accruing without end? In this regard, we simply counted the number of days that our agent experienced penalties. Experimentally, we found that 5 days of continued penalties with high probability signified accruing penalties.

Once in this accruing penalties state, we make efforts to return to the steady state simply by dropping orders. In this state, only orders that are not late are considered for factory

production and delivery. Let t denote the current day.

$$Do(\text{order}) = \begin{cases} \text{true} & \text{if } duedate(\text{order}) \geq t \\ \text{false} & \text{otherwise.} \end{cases}$$

By dropping all non-late orders, it gives us a chance to catch up on our order fulfillment, essentially forcing our agent to return to the steady-state, albeit at a loss from the penalties. Everything else, on both the buy side and sell side, remains unchanged.

6.2. Perturbation State 2: Negative Profits

On the flip side, very high competition levels and low customer demand may allow for the winning of RFQs only at prices below our costs. The price adjustors recommend sell-prices below costs, and we enter the negative profit state, where any sold computer represents a loss for the agent. Sadly, there is nothing that our agent can do about this case, as negative profits are a feature of the market itself. Therefore, the agent waits in this state for market conditions to improve, only offering computers to sell off our inventory before the game ends and freezing purchases on the buy side. The number of desired cycles, used by the sell side to set prices, is also reduced from the steady-state value to that specified by the *liquidation adjustor* (see Section 6.4). This state persists until prices rise above purchasing costs, at which point the agent returns to the steady state.

6.3. Start Game State

At the start of the game, we begin with a completely empty inventory and no orders taken from customers. Our goal is to transition as quickly as possible to the steady-state. This occurs once we have received enough of our first component shipments to begin producing, and once our sell side has adjusted prices so that we win at least 2000 cycles each day.

On the buy side of the agent, we needed to establish the minimum levels of inventory assumed in the steady state in order to begin producing and selling. Because buy-side prices were based on the previous days' demand, prices were always cheapest on the very first day (before there was any demand). This has been termed the *day-0 effect*. To take advantage of this day-0 effect, we order a sizable amount of components on day 0 in three separate orders, giving us a large amount of cheap components that come in at times spaced throughout the game.

Because of the delays in inventory delivery caused by massive day-0 purchases by competing agents, we found that HarTAC generally only managed to start producing around day 60 on average. As such, we wanted components for an average of 300 computers sold per day (empirically found, slightly lower than the 360 average number of computers equivalent to 2000 factory cycles) for 160 days (the remaining time in the game). This amounted to a total inventory of 48000 computers. Our buy-side RFQs then had this inventory distributed uniformly amongst the different types of computers (keep in mind that the sell side does not attempt to form niches in particular products). Any extra components that we needed due to either high demand or to spaces between our three major shipments were bought in the steady state based on our price probing.

We also devised a sabotage/counter-sabotage system to heighten our profits from day-0 ordering while preventing other agents from building up the same cushion of inventory. For sabotage, we sent to one supplier a single RFQ for a ridiculously huge quantity that suppliers could only finish by the last days of the game. As such, swamped with demand, the supplier will offer unreasonable dates for component delivery to all the other agents queued after us. We refuse the order sent to us by the supplier on the next day, with a small expense on our reputation (which is used to prioritize responses by our agent's suppliers). Since there is more than one supplier offering a single type of component for all components other than CPUs, we can recover the number of components needed through a different supplier.

Other agents also adopted sabotage strategies in the final games [3]. This can be a problem because the effect is that incoming orders can have very late delivery dates. To counter this effect, we go through all the supplier offers and choose to accept the combination of offers that minimizes a penalty function that penalizes deviation from an "optimal" continual replenishment of components. We determine the combination of orders that best satisfy our demand for components (i.e. an average number of 160 for non-CPUs, 80 for CPUs per day, at any time within the game), using a penalty function that places greater emphasis on avoiding short-term shortage than short-term excess. A "veto" is also used to avoid accepting an RFQ that would yield quantity that cannot be consumed even by the end of the game. This can be important because the quantities ordered per RFQ can be very large on the first day. As such, we mix the emphasis on avoiding short-term shortage with a greater emphasis on avoiding long-term excess.

At the beginning of the game, we also need to start selling quickly, to attain the volume required for steady-state. For our first bid to customers, we choose to overbid in order to avoid penalties, and set the initial price to exactly the reserve price. After bidding on all of the different types of computers, the steady-state adjustors kicked in (initially lowering the price) to attain and maintain the right level of orders won.

6.4. End Game State

At the end of the game, leftover inventory is essentially wasted money since its final value is 0, and so we aim to liquidate our entire inventory. Therefore, 45 days before the end of the game, HarTAC prepares to exit the game by mov-

ing into the end-game state.

On the buy side,we check that supplier offers will be delivered before the last two days of the game, to avoid receiving components that cannot be used in production.

On the sell side, we adopt an additional price adjuster– the *liquidation adjustor* –to ensure that we will be able to sell all current inventory before the end of the game. We consider the number of remaining days, and consider the number of cycles-worth of computers that are normally sold over that length of time (i.e. allowing for 2000 cycles/day). If there is an *excess* of computers to sell then we further reduce the price on those computers. In particular, for every 10 extra cycles to be sold per day we drop the price by an additional 1%, with $p_i^{t+1} = p_i^t[1 - \beta_l \Delta c_l]$ where $\beta_l = 0.001$ and Δc_l is the per-day excess inventory to shift. The liquidation adjustor clashes with the cycle adjustor (which is seeking to maintain a sell volume of 2000 cycles/day). However, the constant $\beta_l = 0.001$ dominates the constant $\beta = 0.00015$ in the cycle adjustor (Equation 1), and the liquidation adjustor dominates in the late game.

7. Performance in the Competition

HarTAC made it through to the semi-finals of the competition when hardware problems made for one catastrophic game from which the agent could not recover.[5] Yet, as recognized in other papers, HarTAC could have done much better had it not been for this particular game [4]. We will discuss in this part strategic peculiarities that distinguish HarTAC from other agents and contribute to its success.

On the sell side, HarTAC was one of the only agents in the semi-finals to bid on all the customer bundles for which it has inventory available. In principle, this is to help HarTAC achieve the highest prices for all inventory. In practice, HarTAC achieved excellent results in attaining the targeted production and sales level, with average penalty levels around 4.33%. Details of the semi-final round's delivery penalties are summarized in Table 2.

Yet, we did not yet find a satisfactory way to handle markets where HarTAC, lacking in full inventory, could only bid on a segment of the market. Thus, where the optimal number of bundles to win lies below the full production level, HarTAC's final profitability was lower than that of some other agents [4].

5 We ran HarTac from Acapulco, Mexico. The computer that we brought to the competition to run the agent turned out to be a bit unstable, so from time to time different programs would crash. Unfortunately, during the competition we thought that our agent had crashed, but it turned out that it was still running in the background. Consequently, when we restarted it, one version would cause the other version to disconnect, which would itself promptly attempt to reconnect. Upon reconnecting, it would a) disconnect the other agents, and b) perform our day-0 ordering. This made for the record of the single lowest scoring game in the entire competition.

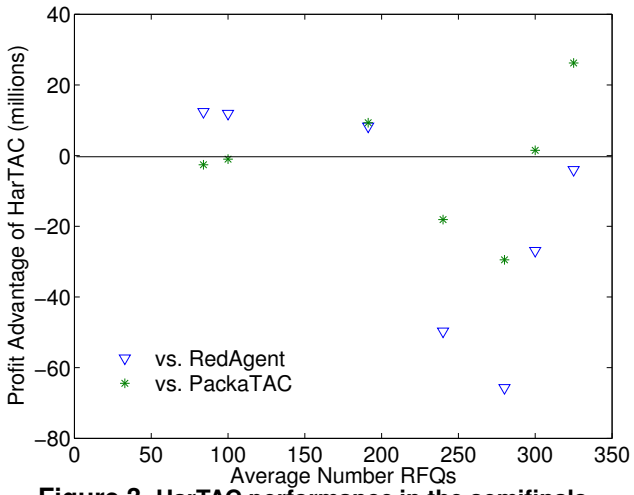

Figure 3. HarTAC performance in the semifinals.

Meanwhile, on the buy side, HarTAC distinguishes itself through a successful mix of aggressive and conservative strategies. Due to the day-0 buy-out effect, two main types of agents emerged in the finals of TAC SCM before the introduction of widespread counter day-0 strategies by Deepmaize. The first is an aggressive day-0 buyer, of which the most successful is RedAgent, the winner of TAC SCM 2003. Aggressive agents build large and cheap inventories at the beginning of the game in hopes of selling them later on. Such agents perform very well in mid to high demand games, but do much worse in low demand games due to the large sunk costs invested at the beginning of the game. On the opposite side, PackaTAC was built upon conservative strategies [2]. In this case, inventories were kept low during the game and bought in small quantities as inventory was sold by the sell side. PackaTAC did well in low demand games, but failed to capitalize upon the highly profitable markets due to the lack of cheap inventory.

HarTAC tried to mix in these two strategies by tuning down the day-0 demand and probing for good prices throughout the game, and only committing to a level of inventory after it has a good idea of the average market level. This gave us a high customer RFQ response rate, surpassed only by RedAgent in the semi-finals, while permitting flexibility for responding to low-demand markets [4]. Figure 3 compares our performance with RedAgent and PackaTAC in the semifinals. We did not include the record-low game during which our agent crashed.

Although based on a small amount of data, this analysis suggests that our strategy is somewhat successful in performing robustly across both low- and high-demand games. In games with lower demand, we are often able to outperform the aggressive bidding strategy of RedAgent, since this aggressive strategy results in many wasted components when it is unable to get all of the orders it needs. In games

	HarTAC	PackaTAC	RedAgent	Deepmaize	TAC-O-Matic	PSUTAC
Average % orders not delivered by due date	4.33	0	10.222	15.625	51.222	1

Table 2. Percentage of orders completed late in the semi-finals in TAC'03.

with higher demand, we are often able to perform better than the conservative bidding strategy of PackaTAC. However, the added costs of the in-game buying (i.e. probing costs and the fact that our probing does not guarantee the absolute cheapest purchases) are still quite significant, and RedAgent outperforms HarTAC in high-demand games. Meanwhile, we find that PackaTAC slightly outperforms us in the low demand games, when we waste some of our day-0 component purchases.

In general, HarTAC was moderately aggressive in its strategy. This allowed us to capitalize on some of the benefits resulting from high-demand, while avoiding the pitfalls of over-aggressiveness, resulting in moderate success overall.

8. Conclusions

For HarTAC, the state formulation provided a relatively simple formalization for us to work from, and our steady state definition provided us with a starting place from which we could define all our other states. Ultimately, the states also provided us with the intuition for our behavior rules. HarTAC is, we think, unique in its focus on *targeted production cycles* throughout the agent, even in adopting this approach to its sell-side mechanism.

Although encouraged by HarTAC's performance, the current state-based design is perhaps overly simplistic. With so many possible environmental conditions, it became difficult to keep the agent in the steady state. This was most clearly shown by the complicated sub-state system for our steady state, and in the sweeping behavior rules that we were forced to define that perhaps warranted more complex, context-specific behavior. With these issues in mind, we would be interested in pursuing a more fine-grained control model in future work.

References

[1] K J Astrom and B Wittenmark. *Adaptive Control*, Addison-Wesley, second edition, 1994.

[2] E. Dahlgren and P. R. Wurman. PackaTAC: A Conservative Trading Agent. *SIGecom Exchanges*, 4(3):33–40, 2004.

[3] J. Estelle, Y. Vorobeychik, M. Wellman, S. Singh, C. Kiekintveld, and V. Soni. Strategic interactions in a supply chain game. Technical report, University of Michigan, 2003.

[4] P. W. Keller, F.-O. Duguay, and D. Precup. RedAgent– Winner of TAC SCM 2003. *SIGecom Exchanges*, 4(3):1–8, 2004.

[5] C. Kiekintveld, M. P. Wellman, S. Singh, J. Estelle, Y. Vorobeychik, V. Soni, and M. Rudary. Distributed feedback control for decision making on supply chains. In *Proc. 14th Int. Conf. on Automated Planning and Scheduling*, 2004. To appear.

[6] N. M. Sadeh, R. Arunachalam, J. Eriksson, N. Finne, and S. Janson. TAC'03: A supply chain trading competition. *AI Magazine*, 24(1), Spring 2003.