

BalancedGreedyStdAgent: An agent submitted to the ANAC 2026 SCM league

Takaya Nishigaki¹ Takanobu Otsuka²
^{1,2}Nagoya Institute of Technology, Aichi, Japan
¹`takaya0908takayawww@gmail.com`
²`otsuka.takanobu@nitech.ac.jp`

June 22, 2026

Abstract

This report describes **BalancedGreedyStdAgent**, an autonomous factory-manager agent for the Standard track of the Supply Chain Management League (SCML). The agent follows a deliberately conservative design: it concedes on price as a negotiation progresses, and it controls contract quantities using the per-step needs reported by the agent–world interface (`awi.needed_sales` and `awi.needed_supplies`) so that it largely avoids over-buying and over-selling. We evaluate the agent both with the **WorldRunner** utility across several market contexts and with an ANAC-style tournament. In the general (ANAC) context the needs-based quantity control raised the agent from last place among the tested agents to first, and in an ANAC-style tournament the agent finished first ahead of the built-in **GreedyStdAgent** and **SyncRandomStdAgent** baselines.

1 Introduction

The Supply Chain Management League (SCML) simulates a supply chain in which factories buy raw materials and sell finished products to one another. Each factory is run by an autonomous agent whose objective is to maximise its own profit by negotiating bilateral contracts with other agents. Negotiations are concurrent and their outcomes are interdependent: a contract to sell a product is only profitable if the corresponding input material has been (or can be) secured, and buying more material than can be sold incurs storage costs, while promising more product than can be produced incurs shortfall penalties.

This report presents **BalancedGreedyStdAgent**, our entry to the SCML Standard track. Rather than attempting sophisticated opponent modelling, the agent is built around two simple but robust ideas: (i) a time-based concession strategy for price, and (ii) quantity control driven by the agent’s actual remaining needs at each simulation step. The design philosophy is that in a volatile, concurrent market, avoiding loss-making contracts is at least as important as extracting the best possible price.

2 The Design of BalancedGreedyStdAgent

The agent inherits from **StdAgent** and implements the negotiation callbacks `propose` and `respond`. The remaining sections describe the three main aspects of its behaviour.

2.1 Negotiation Choices

Price. The agent uses a time-based concession strategy that is symmetric between proposing and responding. Let $t \in [0, 1]$ be the relative time within a negotiation, and let $p_{\min}$, $p_{\max}$ and $p_{\text{mid}} = (p_{\min} + p_{\max})/2$ be the minimum, maximum and middle unit prices of the negotiation issue. When selling, the agent proposes a price that starts near $p_{\max}$ and concedes towards p_{mid} as $t \rightarrow 1$; when buying, it starts near $p_{\min}$ and concedes towards p_{mid} . The acceptance threshold in `respond` follows the same schedule, so that early in a

negotiation the agent only accepts prices close to its ideal, and becomes more accommodating as the deadline approaches. This keeps the proposing and responding behaviour consistent.

Delivery time. When selling, the agent proposes a delivery time two steps ahead of the current step, leaving time to procure inputs and run production; when buying, it requests delivery at the current step so that the material can be used as soon as possible. Proposed times are always clipped to the valid range of the time issue, so the agent never proposes an infeasible date near the end of the simulation.

2.2 Concurrent Negotiation

Because many negotiations run in parallel and their outcomes interact, the key challenge is to avoid committing to more than the factory can absorb. The agent solves this by deriving its quantities from the agent-world interface rather than from any single negotiation. At each step it reads `awi.needed_sales` (how many units it should still sell) and `awi.needed_supplies` (how many units it should still buy). When proposing, the agent offers a quantity bounded by the relevant remaining need (and clipped to the issue’s range); when responding, it rejects any offer whose quantity exceeds the remaining need, and rejects all offers once the corresponding need has reached zero.

A limitation: within-step over-commitment. A subtle issue arises because the agent derives from `StdAgent`, whose `respond` callback is invoked once per offer rather than once per step. This raises the question of whether `needed_sales` and `needed_supplies` are updated incrementally as agreements are reached within a step, or only at step boundaries. We verified this empirically by logging the reported needs on every `respond` call. Within a single step the values are constant: for example, at step 0 we observed `needed_sales` = 8 across dozens of consecutive `respond` calls, with no decrease as offers were processed. The needs are therefore updated only at step boundaries. A consequence is that the per-offer check is sound for any *single* offer, but it does not prevent over-commitment from *multiple* accepted offers within the same step: each acceptance independently sees the same remaining need and judges itself to be within budget, so their accepted quantities can collectively exceed it. In other words, the needs act as a per-offer bound but not as a shared running constraint across simultaneous negotiations. Two remedies are possible: maintaining a within-step accounting of quantities already accepted (subtracting them from the reported need), or switching the base class to `StdSyncAgent`, which processes all of a step’s offers together and can allocate the remaining need across them. We did not implement either; we discuss them as future work (Section 4).

2.3 Risk Management

The agent’s risk management is a direct consequence of its needs-based quantity control. By never buying more than `needed_supplies` and never selling more than `needed_sales`, the agent limits two of the main sources of loss in SCML: holding unsold inventory (which incurs storage cost) and over-committing to sales it cannot supply (which incurs shortfall penalties). An earlier version of the agent capped quantities only at the number of production lines (`n_lines`) and tracked buy/sell commitments with its own counters; this proved fragile in markets where the agent had to both buy and sell, because the buy and sell limits were not linked to the agent’s true net position. Replacing the line-based cap with the interface’s needs resolved this problem (see Section 3). One gap remains, however: as discussed above, the needs-based bound is enforced per offer rather than across the simultaneous offers of a single step, so within-step over-commitment is still possible.

3 Evaluation

We evaluated the agent in two complementary ways.

Context comparison with WorldRunner. During development we used the `WorldRunner` utility, which runs several agents under identical conditions, across three contexts: the general `ANACStdContext`, `StrongSupplierStdContext` (the agent is at the first production level), and `StrongConsumerStdContext`. This makes it possible to see

how the agent behaves in different market positions. The decisive finding was that the agent performed competitively when its market role was fixed (second place in both the supplier and consumer contexts) but poorly in the general context, where its role varies and it must both buy and sell. Introducing needs-based quantity control (Section 2.3) raised the agent’s score in the general context from 0.33 (last among the tested agents) to first place, while leaving its strong performance in the consumer context essentially unchanged. We note that these are local evaluations: the opponents, random seeds, and tournament settings differ from the official competition, so absolute scores should be read as relative indicators rather than predictions of the official ranking.

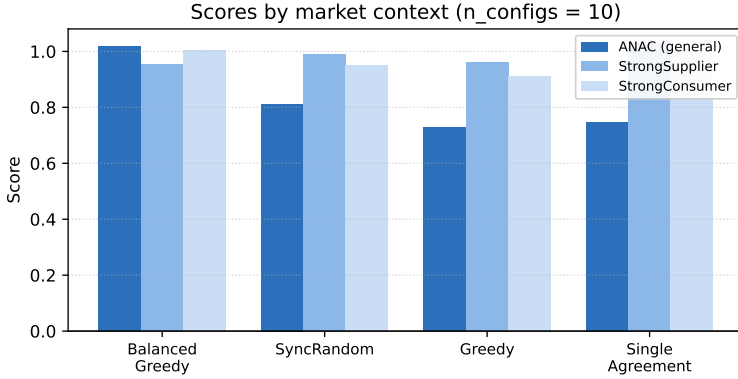


Figure 1: Scores of the main agents in three market contexts ($n_configs = 10$). The agent is competitive in every context and strongest in the general (ANAC) one.

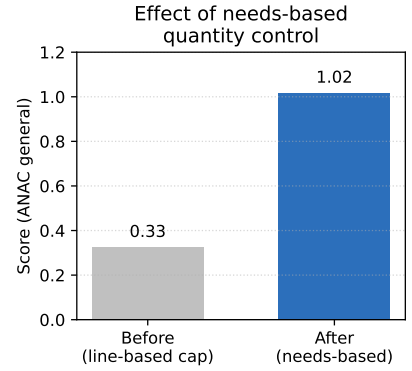


Figure 2: Score in the general context before and after introducing needs-based quantity control.

As Figure 1 shows, after the improvement the agent is no longer weak in any single context, and Figure 2 highlights the magnitude of the change in the general context that the needs-based control produced.

ANAC-style tournament. For a final evaluation closer to the official scoring method, we ran an ANAC-style Standard tournament using the SCML library’s `anac2024_std` utility, with `n_steps = 50`, `n_configs = 5` and `n_runs_per_world = 1`, against the built-in `GreedyStdAgent` and `SyncRandomStdAgent` baselines. The results are shown in Table 1. The agent finished first, ahead of both baselines. As with the context comparison, this is a local tournament rather than the official competition, but it uses the same scoring machinery and therefore gives a reasonable indication of the agent’s relative strength.

Agent	Score
BalancedGreedyStdAgent	1.0399
SyncRandomStdAgent	0.6793
GreedyStdAgent	0.6124

Table 1: Total scores in an ANAC-style Standard tournament ($n_steps = 50$, $n_configs = 5$, $n_runs_per_world = 1$).

4 Lessons and Suggestions

The most important lesson was that quantity control matters more than price tactics in this market. An early experiment that simply increased the proposed quantity (to match the number of production lines) made the agent worse, because it led to over-buying and over-selling; this showed that proposing larger quantities without regard to actual need is harmful. The breakthrough came from tying quantities to `needed_sales` and `needed_supplies`, which made the largest single improvement we observed.

We also learned to be careful about the variance of local evaluation. Scores fluctuate substantially between runs because each run generates new market configurations, so a single run can be misleading; comparing agents within the same run, and increasing the number of configurations, gives more reliable conclusions.

For future work we suggest, in order of priority: (i) closing the within-step over-commitment gap identified in Section 3, either by tracking quantities already accepted within a step or by moving to a **StdSyncAgent** base class that processes a step’s offers jointly, since this is the clearest correctness weakness of the current design; (ii) basing the price thresholds on the agent’s own production cost rather than the midpoint of the price range, so that acceptance decisions reflect the true break-even point; (iii) adapting the concession schedule to the opponent rather than using a fixed schedule for all partners; and (iv) revisiting performance in the supplier context, where the agent is competitive but slightly behind the strongest baseline and may be conceding sales it could profitably make.

Conclusions

BalancedGreedyStdAgent is a conservative SCML Standard agent that combines a symmetric time-based price concession with needs-based quantity control. The latter was the key design decision: by sizing contracts to the agent’s actual remaining needs, the agent avoids the inventory and shortfall costs that arise from over-committing, and this turned a last-place agent in the general market context into a first-place one. In an ANAC-style tournament the agent outperformed the built-in baselines. The results suggest that, in a volatile and concurrent market, robust quantity management is a particularly effective foundation on which more sophisticated negotiation tactics can later be built.