Execution Monitoring as Meta-Games for General Game-Playing Robots

David Rajaratnam and **Michael Thielscher**
The University of New South Wales
Sydney, NSW 2052, Australia
{daver,mit}@cse.unsw.edu.au

Abstract

General Game Playing aims to create AI systems that can understand the rules of new games and learn to play them effectively without human intervention. The recent proposal for *general game-playing robots* extends this to AI systems that play games in the real world. *Execution monitoring* becomes a necessity when moving from a virtual to a physical environment, because in reality actions may not be executed properly and (human) opponents may make illegal game moves. We develop a formal framework for execution monitoring by which an action theory that provides an axiomatic description of a game is automatically embedded in a *meta-game* for a robotic player — called the *arbiter* — whose role is to monitor and correct failed actions. This allows for the seamless encoding of recovery behaviours within a meta-game, enabling a robot to recover from these unexpected events.

1 Introduction

General game playing is the attempt to create a new generation of AI systems that can understand the rules of new games and then learn to play these games without human intervention [Genesereth *et al.*, 2005]. Unlike specialised systems such as the chess program Deep Blue, a general game player cannot rely on algorithms that have been designed in advance for specific games. Rather, it requires a form of general intelligence that enables the player to autonomously adapt to new and possibly radically different problems. General game-playing robots extend this capability to AI systems that play games in the real world [Rajaratnam and Thielscher, 2013].

Execution monitoring [Hähnel *et al.*, 1998; De Giacomo *et al.*, 1998; Fichtner *et al.*, 2003] becomes a necessity when moving from a purely virtual to a physical environment, because in reality actions may not be executed properly and (human) opponents may make moves that are not sanctioned by the game rules. In a typical scenario a robot follows a plan generated by a traditional planning algorithm. As it executes each action specified by the plan the robot monitors the environment to ensure that the action has been successfully executed. If an action is not successfully executed then some recovery or re-planning behaviour is triggered. While the sophistication of execution monitors may vary [Pettersson, 2005] a common theme is that the execution monitor is independent of any action planning components. This allows for a simplified model where it is unnecessary to incorporate complex monitoring and recovery behaviour into the planner.

In this paper, we develop a framework for execution monitoring for general game-playing robots that follows a similar model. From an existing game axiomatised in the general Game Description Language GDL [Genesereth *et al.*, 2005; Love *et al.*, 2006] a *meta-game* is generated that adds an execution monitor in the form of an *arbiter* player. The "game" being played by the arbiter is to monitor the progress of the original game to ensure that the moves played by each player are valid. If the arbiter detects an illegal or failed move then it has the task of restoring the game to a valid state. Importantly, the non-arbiter players, whether human or robotic, can ignore and reason without regard to the arbiter player while the latter becomes active only when an error state is reached.

Our specific contributions are: (1) A fully axiomatic approach to embedding an arbitrary GDL game into a meta-game that implements a basic execution monitoring strategy relative to a given physical game environment. This meta-game is fully axiomatised in GDL so that any GGP player can take on the role of the arbiter and thus be used for execution monitoring. (2) Proofs that the resulting meta-game satisfies important properties including being a well-formed GDL game. (3) Generalisations of the basic recovery behaviour to consider actions that are not reversible but instead may involve multiple actions in order to recover the original game state and that need to be planned by the arbiter.

The remainder of the paper is as follows. Section 2 briefly introduces the GDL language for axiomatising games. Section 3 presents a method for embedding an arbitrary GDL game into a GDL meta-game for execution monitoring. Section 4 investigates and proves important properties of the resulting game description, and Sections 5 and 6 describe and formalise two extensions of the basic recovery strategy.

2 Background: General Game Playing, GDL

The annual AAAI GGP Competition [Genesereth *et al.*, 2005] defines a general game player as a system that can understand the rules of an n-player game given in the general Game Description Language (GDL) and is able to play those games effectively. Operationally a game consists of a central

```
1  (role white)  (role black)
2
3  (init (cell a 1 white))  ...  (init (cell h 8 black))
4  (init (control white))
5
6  (<= (legal white (move ?u ?x ?u ?y))
7      (true (control white))     (++ ?x ?y)
8      (true (cell ?u ?x white))  (cellempty ?u ?y))
9  (<= (next (cell ?x ?y ?p)) (does ?p (move ?u ?v ?x ?y)))
10
11 (<= terminal (true (cell ?x 8 white)))
12 (<= (goal white 100) (true (cell ?x 8 white)))
13 (<= (goal black 0)   (true (cell ?x 8 white)))
14
15 (<= (cellempty ?x ?y) (not (true (cell ?x ?y white)))
16                       (not (true (cell ?x ?y black))))
17 (++ 1 2) ... (++ 7 8)
```

Figure 1: An excerpt from a game description with the GDL keywords highlighted.

controller that progresses the game state and communicates with players that receive and respond to game messages.

The declarative language GDL supports the description of a finite n-player game ($n \geq 1$). As an example, Fig. 1 shows some of the rules for BREAKTHROUGH, a variant of chess where the players start with two rows of pawns on their side of the board and take turns trying to "break through" their opponent's ranks to reach the other side first. The two **roles** are named in line 1. The **initial state** is given in lines 3–4. Lines 6–8 partially specify the **legal moves**: White, when it is his turn, can move a pawn forward to the next rank, provided the cell in front is empty. Line 9 is an example of a **position update** rule: A pawn when being moved ends up in the new cell. A **termination** condition is given in line 11, and lines 12–13 define the goal values for each player under this condition. Additional rules (lines 15–17) define auxiliary predicates, including an axiomatisation of simple arithmetics.

GDL is based on standard logic programming principles, albeit with a different syntax. Consequently, we adopt logic programming conventions in our formalisation, while maintaining GDL syntax for code extracts. Apart from GDL keyword restrictions [Love *et al.*, 2006], a GDL description must satisfy certain logic programming properties, specifically it must be *stratified* [Apt *et al.*, 1987] and *safe* (or *allowed*) [Lloyd and Topor, 1986]. As a result a GDL description corresponds to a state transition system (cf. [Schiffel and Thielscher, 2010]). We adopt the following conventions for describing some game state transition system properties:

- $S^{\texttt{true}} \stackrel{\text{def}}{=} \{\texttt{true}(f_1), \ldots, \texttt{true}(f_n)\}$ consists of the fluents that are true in state S.
- $A^{\texttt{does}} \stackrel{\text{def}}{=} \{\texttt{does}(r_1, a_1), \ldots, \texttt{does}(r_n, a_n)\}$, consists of the role-action statements making up a joint move A.
- $l(S) = \{(r, a) \mid G \cup S^{\texttt{true}} \models \texttt{legal}(r, a)\}$, where each a is an action that is legal for a role r in a state S for the game G.
- $g(S) = \{(r, n) \mid G \cup S^{\texttt{true}} \models \texttt{goal}(r, n)\}$, where each n is a value indicating the goal score for a role r in a state S for the game G.

Finally, while the syntactic correctness of a game description specifies a game's state transition system, there are additional requirements for a game to be considered *well-formed*

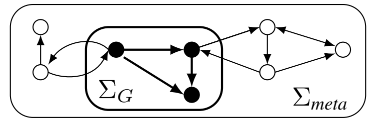


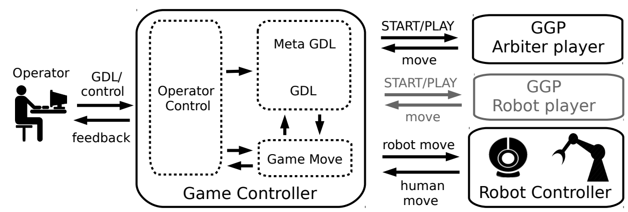
Figure 2: Embedding a game Σ_G into a meta-game Σ_{meta}.

[Love *et al.*, 2006] and therefore usable for the GGP competitions. In particular a well-formed game must terminate after a finite number of steps, every role must have at least one legal action in every non-terminal state (*playability*), every role must have exactly one goal value in every state and these values must increase monotonically, for every role there is a sequence of joint moves that leads to a terminal state where the role's goal value is maximal (*weakly-winnable*).

3 Meta-Games for Execution Monitoring

3.1 Systems Architecture

The following diagram illustrates the intended overall systems architecture for execution monitoring of general games played with a robot in a real environment.



It comprises the following components.

Game Controller. Accepts an arbitrary GDL description of a game (from a human operator) and (a) *automatically generates a meta-GDL game for the arbiter to monitor its execution*; (b) interacts with one or more general game-playing systems that take different roles in the (meta-)game; and (c) interacts with the robot controller.

Robot Controller. Serves as the low-level executor of the robot by (a) processing sensor data in order to recognise actions; (b) commanding the object manipulator to execute any instance of a move action defined for the environment.

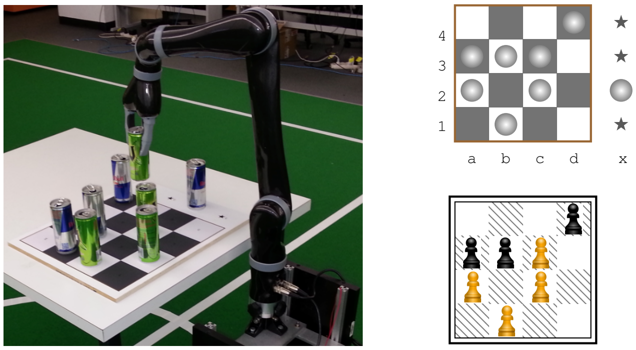
GGP Robot Player. Plays the desired game in the desired role for the robot.

GGP Arbiter Player. *Monitors the execution of the game by playing the automatically constructed meta-game.*

In the following we describe in detail an axiomatic method of embedding a given game and a specific execution monitoring strategy into a meta-game. Our aim is to fully axiomatise this meta-game in GDL, so that any GGP player can play the role of the arbiter and thus be used for execution monitoring.

3.2 Automatic Construction of Meta-Games

The general approach of embedding a given "source" game into a meta-game is graphically illustrated in Figure 2. Games can be viewed as state transition systems. Execution errors correspond to changes in the environment that are physically possible but do not correspond to legal moves in a game. To account for—and recover from—such errors, we embed the state machine for the original game into one that describes all possible ways in which the game environment can evolve.



(a) (b)

Figure 3: Game environment with robot, and breakthrough chess projected onto this environment.

Some of these transitions lead outside of the original game, e.g. when a human opponent makes an illegal move or the execution of a robot move fails. The role of the arbiter player, who monitors the execution, is to detect these abnormalities and when they occur to perform actions that bring the system back into a normal state. Hence, a meta-game combines the original game with a model of the environment and a specific execution monitoring strategy. Its GDL axiomatisation can therefore be constructed as follows:

$$\Sigma_{meta} \; = \; \tau(\Sigma_G) \cup \Sigma_{env} \cup \Sigma_{em} \text{ , where}$$

- Σ_G is the original GDL game and $\tau(\Sigma_G)$ a rewritten GDL allowing for it to be embedded into the meta-game;
- Σ_{env} is a GDL axiomatisation describing all possible actions and changes in the physical game environment; and
- Σ_{em} is a GDL axiomatisation implementing a specific execution monitoring strategy.

Redefining the source game $\tau(\Sigma_G)$

In order to embed a given game G into a meta-game for execution monitoring, simple rewriting of some of the GDL keywords occurring in the rules describing G are necessary. Specifically, the meta-game extends the fluents and actions of the original game and redefines preconditions and effects of actions, as detailed below. Hence:

Definition 1 *Let Σ_G be a GDL axiomatisation of an arbitrary game. The set of rules $\tau(\Sigma_G)$ are obtained from Σ_G by replacing every occurrence of*

- `base`(f) *by* `source_base`(f)*;*
- `input`(r, a) *by* `source_input`(r, a)*;*
- `legal`(r, a) *by* `source_legal`(r, a)*; and*
- `next`(f) *by* `source_next`(f).

All other keywords, notably `true` and `does`, remain unchanged.

Environment models Σ_{env}

The purpose of the game environment axiomatisation is to capture all actions and changes that may happen, intentionally or unintentionally, in the physical environment and that we

```
1  (env_coord a 1) (env_coord b 1) ... (env_coord x 4)
2  (env_coord a-b 1) (env_coord b-c 1) ...
3  (<= (env_base (env_can ?x ?y))
4      (env_coord ?x ?y))
5  (<= (env_base (env_fallen ?x ?y))
6      (env_coord ?x ?y))
7  (<= (env_input ?r noop)
8      (role ?r))
9  (<= (env_input ?r (move ?u ?v ?x ?y))
10     (role ?r) (env_coord ?u ?v) (env_coord ?x ?y))
11 (<= (env_input ?r (move_and_topple ?u ?v ?x ?y))
12     (role ?r) (env_coord ?u ?v) (env_coord ?x ?y))
13
14 (init (env_can a 1)) ... (init (env_can d 4))
15 (env_legal noop)
16 (<= (env_legal (move ?u ?v ?x ?y))
17     (true (env_can ?u ?v))
18     (not (true (env_can ?x ?y))) (env_coord ?x ?y))
19 (<= (env_legal (move_and_topple ?u ?v ?x ?y))
20     (true (env_can ?u ?v))
21     (not (true (env_can ?x ?y))) (env_coord ?x ?y))
22
23 (<= (env_next (env_can ?x ?y))
24     (or (does ?p (move ?u ?v ?x ?y))
25         (does ?p (move_and_topple ?u ?v ?x ?y)))
26         ((true (env_can ?u ?v))
27          (not (env_can_moved ?u ?v))))
28 (<= (env_next (env_fallen ?x ?y))
29     (or (does ?p (move_and_topple ?u ?v ?x ?y))
30         ((true (env_fallen ?u ?v))
31          (not (env_can_moved ?u ?v)))))
32 (<= (env_can_moved ?u ?v)
33     (or (does ?p (move ?u ?v ?x ?y))
34         (does ?p (move_and_topple ?u ?v ?x ?y))))
35
36 (env_reverse noop noop)
37 (<= (env_reverse (move ?u ?v ?x ?y) (move ?x ?y ?u ?v))
38     (env_coord ?u ?v) (env_coord ?x ?y))
39 (<= (env_reverse (move ?u ?v ?x ?y)
40                  (move_and_topple ?x ?y ?u ?v))
41     (env_coord ?u ?v) (env_coord ?x ?y))
```

Figure 4: An example GDL axiomatisation Σ_{env} of the game environment of Figure 3, including an axiomatisations of failure actions (`move_and_topple`) and how to reverse the effects of actions (axioms 36–41).

expect to be observed and corrected through execution monitoring. Consider, for example, the game environment shown in Fig. 3(a). It features a 4×4 chess-like board with an additional row of 4 marked positions on the right. Tin cans are the only type of objects and can be moved between the marked positions. They can also (accidentally or illegally) be moved to undesired locations such as the border between adjacent cells or to the left of the board etc. Moreover, some cans may also have (accidentally) toppled over. A basic action theory for this environment that allows to detect and correct both illegal as well as failed moves comprises the following.

Actions: `noop`; moving a can, `move`(u, v, x, y); and toppling a can while moving it, `move_and_topple`(u, v, x, y); where $u, x \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{x}\} \cup \{\mathtt{a-}, \mathtt{a-b}, \dots, \mathtt{x-}\}$ and $v, y \in \{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\} \cup \{\mathtt{1-}, \mathtt{1-2}, \dots, \mathtt{4-}\}$. The additional coordinates $\mathtt{a-}, \mathtt{a-b}, \dots, \mathtt{1-}, \dots$ are qualitative representations of "illegal" locations to the left of the board, between files a and b, etc.

Fluents: Any location (x, y) can either be empty or house a can, possibly one that has fallen over. Figure 3(b) illustrates one possible state, in which all cans happen to be positioned upright and at legal locations.

Action `noop` is always possible and has no effect. Actions

$\texttt{move}(u, v, x, y)$ and $\texttt{move\_and\_topple}(u, v, x, y)$ are possible in any state with a can at (u, v) but not at (x, y). The effect is to transition into a state with a can at (x, y), fallen over in case of the second action, and none at (u, v).

A corresponding set of GDL axioms is given in Figure 4. It uses env_base, env_input, env_legal and env_next to describe the fluents, actions, preconditions, and effects.

Note that this is just one possible model of the physical environment for the purpose of identifying—and correcting—illegal and failed moves by both (human) opponents and a robotic player. An even more comprehensive model may account for other possible execution errors such as cans completely disappearing, e.g. because they fell off the table.[1]

We tacitly assume that any environment model Σ_{env} includes the legal actions of the source game description Σ_G as a subset of all possible actions in the environment.[2]

Execution monitoring strategies Σ_{em}

The third and final component of the meta-game serves to axiomatise a specific execution monitoring strategy. It is based on the rules of the source game and the axiomatisation of the environment. The meta-game extends the legal moves of the source game by allowing anything to happen in the meta-game that is physically possible in the environment (cf. Figure 2). As long as gameplay stays within the boundaries of the source game, the arbiter does not interfere. When an abnormality occurs, the arbiter takes control and attempts to correct the problem. For ease of exposition, in the following we describe a basic recovery strategy that assumes each erroneous move n to be correctible by a single reversing move m as provided in the environment model through the predicate $\texttt{env\_reverse}(m, n)$. However, a more general execution monitoring strategy will be described in Section 5.

Fluents, roles, and actions. The meta-game includes the additional arbiter role, the actions are that of the environment model,[3] and the following meta-game state predicates.

```
(role arbiter)
(<= (input ?r ?m) (env_input ?r ?m))
(<= (base ?f) (source_base ?f))
(<= (base ?f) (env_base ?f))                    (1)
(<= (base (meta_correcting ?m))
    (env_input ?r ?m))
```

The new fluent $\texttt{meta\_correcting}(m)$ will be used to indicate an abnormal state caused by the (physically possible but illegal) move m. Any state in which a fluent of this form is true is an *abnormal* state, otherwise it is a *normal* state.

Legal moves. In a normal state—when no error needs to be corrected—the meta-game goes beyond the original game in considering *any move that is possible in the environment* for players. The execution monitoring strategy requires the arbiter not to interfere (i.e., to noop) while in this state.

```
(<= (legal ?p ?m)
    (not meta_correcting_mode)
    (role ?p) (distinct ?p arbiter)
    (env_legal ?m))
(<= (legal arbiter noop)                         (2)
    (not meta_correcting_mode))

(<= meta_correcting_mode
    (true (meta_correcting ?m)))
```

In recovery mode, the arbiter executes recovery moves while normal players can only noop.

```
(<= (legal arbiter ?n)
    (true (meta_correcting ?m))
    (env_reverse ?n ?m))
(<= (legal ?p noop)                              (3)
   meta_correcting_mode
    (role ?p) (distinct ?p arbiter))
```

State transitions. The meta-game enters a correcting mode whenever a bad move is made. If multiple players simultaneously make bad moves, the arbiter needs to correct them in successive states, until no bad moves remain.

```
(<= (next (meta_correcting ?m))
    (meta_player_bad_move ?p ?m))
(<= (next (meta_correcting ?m))
    (true (meta_correcting ?m))
    (not (meta_currently_correcting ?m)))

(<= (meta_currently_correcting ?m)               (4)
    (does arbiter ?n)
    (env_reverse ?n ?m))
(<= (meta_player_bad_move ?p ?m)
    (not meta_correcting_mode)
    (does ?p ?m) (distinct ?p arbiter)
    (not (source_legal ?p ?m)))
```

The environment fluents are always updated properly. Meanwhile, in normal operation the fluents of the embedded game are updated according to the rules of the source game.

```
(<= (next ?f) (env_next ?f))
(<= (next ?f)
    (not meta_bad_move) (source_next ?f)
    (not meta_correcting_mode))                  (5)

(<= meta_bad_move
    (meta_player_bad_move ?p ?m))
```

When a bad move has just been made, the fluents of the embedded game are fully preserved from one state to the next throughout the entire recovery process.

```
(<= (next ?f) (source_base ?f) (true ?f)
              meta_bad_move)
(<= (next ?f) (source_base ?f) (true ?f)       (6)
              meta_correcting_mode)
```

This completes the formalisation of a basic execution monitoring strategy by meta-game rules. It is game-independent and hence can be combined with any game described in GDL.

---

[1] We also note that, for the sake of simplicity, the example axiomatisation assumes at most one can at every (qualitatively represented) location, including e.g. the region below and to the left of the board. A more comprehensive axiomatisation would either use a more fine-grained coordinate system or allow for several objects to be placed in one region.

[2] This corresponds to the requirement that games be *playable in a physical environment* [Rajaratnam and Thielscher, 2013].

[3] Recall the assumption that these include all legal actions in the source game.

4 Properties

The previous section detailed the construction of a meta-game Σ_{meta} from an existing game Σ_G and a corresponding physical environment model Σ_{env}. In this section we show that the resulting meta-game does indeed encapsulate the original game and provides for an execution monitor able to recover the game from erroneous actions.

Proposition 1 *For any syntactically correct GDL game Σ_G and corresponding environment model Σ_{env}, the meta-game Σ_{meta} is also a syntactically correct GDL game.*

Proof: Σ_G and Σ_{env} are required to satisfy GDL keyword restrictions, be safe, and internally stratified. $\tau(\Sigma_G)$ does not change these properties and it can be verified that Σ_{em} also satisfies them. For example, `legal` would depend on `does` only if `source_legal` depended on `does`, which is only possible if Σ_G violated these restrictions.

Now, verifying that $\Sigma_{meta} = \tau(\Sigma_G) \cup \Sigma_{env} \cup \Sigma_{em}$ is stratified. The properties of Σ_{env} (resp. Σ_{em}) ensures that Σ_{meta} will be unstratified only if $\tau(\Sigma_G) \cup \Sigma_{em}$ (resp. $\tau(\Sigma_G) \cup \Sigma_{env}$) is unstratified. Hence Σ_{meta} could be unstratified only if $\tau(\Sigma_G)$ was unstratified. □

The syntactic validity of the meta-game guarantees its correspondence to a state transition system. Now, we consider terminating paths through a game's state transition systems.

Definition 2 *For a game G with transition function δ, let $\langle s_0, m_0, s_1, m_1, \ldots, s_n\rangle$ be a sequence of alternating states and joint moves, starting at the initial state s_0 and assigning $s_{i+1} = \delta(m_i, s_i)$, $0 < i \leq n$, such that s_n is a terminal state. Such a sequence is called a* run *of the game.*

To show that the meta-game encapsulates the original we first establish that a run through the original game maps directly to a run in the meta-game.

Proposition 2 *For any run $\langle s_0, m_0, s_1, m_1, \ldots, s_n\rangle$ of the game Σ_G there exists a run $\langle s'_0, m'_0, s'_1, m'_1, \ldots, s'_n\rangle$ of the game Σ_{meta} such that:*

- $m_i'^{\texttt{does}} = m_i^{\texttt{does}} \cup \{\texttt{does}(\texttt{arbiter}, \texttt{noop})\}, 0 \leq i \leq n-1,$
- $s_i^{\texttt{true}} \subseteq s_i'^{\texttt{true}},\ 0 \leq i \leq n,$
- $g(s_n) \subseteq g(s'_n)$.

Proof: $\langle s'_0, m'_0, s'_1, m'_1, \ldots, s'_n\rangle$ is constructed inductively.

Base case: observe that $\tau(\Sigma_G)$ does not change the fluents that are true in the initial state so $s_0^{\texttt{true}} \subseteq s_0'^{\texttt{true}}$. Furthermore no `meta_correcting` fluent is specified for the initial state of Σ_{meta} so s'_0 is a normal state.

Inductive step: select m'_i such that $m_i'^{\texttt{does}} = m_i^{\texttt{does}} \cup \{\texttt{does}(\texttt{arbiter}, \texttt{noop})\}$. Now, Σ_G is required to be physically playable in the environment model Σ_{env} so from Axioms (2) and s'_i being a normal state it follows that $l(s_i) \subseteq l(s'_i)$ and $\{\texttt{legal}(\texttt{arbiter}, \texttt{noop})\} \subseteq l(s'_i)$. Hence m'_i consists of legal moves and its execution leads to a game state s'_{i+1}. Furthermore since the moves in $m_i^{\texttt{does}}$ are `source_legal` then s'_{i+1} will be a normal state (Axioms 4) and furthermore $s_{i+1}^{\texttt{true}} \subseteq s_{i+1}'^{\texttt{true}}$. Finally, $\tau(\Sigma_G)$ does not modify any goal values or the `terminal` predicate so s'_{i+1} will be a terminal state iff s_{i+1} is a terminal state and any goal values will be the same. □

Proposition 2 captures the soundness of the meta-game with respect to the original game. A similar completeness result can also be established where any (terminating) run of the meta-game corresponds to a run of the original, when the corrections of the execution monitor are discounted.

Proposition 3 *For any run $\langle s_0, m_0, s_1, m_1, \ldots, s_n\rangle$ of the game Σ_{meta} there exists a run $\langle s'_0, m'_0, s'_1, m'_1, \ldots, s'_o\rangle$, where $o \leq n$, of the game Σ_G such that:*

- *for each pair (s_i, m_i), $0 \leq i \leq n$:*
  - *if s_i is a* normal *state then there is a pair (s'_j, m'_j), $0 \leq j \leq i$, s.t. $s_j'^{\texttt{true}} \subseteq s_i^{\texttt{true}}$ and $m_i^{\texttt{does}} = m_j'^{\texttt{does}} \cup \{\texttt{does}(\texttt{arbiter}, \texttt{noop})\}$,*
  - *else, s_i is an abnormal state, then for each non-arbiter role r, $\texttt{does}(r, \texttt{noop}) \in m_i^{\texttt{does}}$ and $\texttt{does}(r, a) \notin m_i^{\texttt{does}}$ for any $a \neq \texttt{noop}$.*
- $g(s'_o) \subseteq g(s_n)$.

Proof: Similar to Proposition 2 but need to consider the case where a joint action in Σ_{meta} is not legal in Σ_G. Note, the axioms for Σ_{em} and Σ_{env} do not provide for termination so the terminal state s_n must be a normal state.

Base case: note $s_0'^{\texttt{true}} \subseteq s_0^{\texttt{true}}$ and s_0 is a normal state.

Inductive step: assume s_i is a normal state and there is an s'_j, $j \leq i$, s.t. $s_j'^{\texttt{true}} \subseteq s_i^{\texttt{true}}$. There are two cases:
1) If all non-arbiter actions in m_i are `source_legal` then construct m'_j from m_i (minus the arbiter action). It then follows that $s_{j+1}'^{\texttt{true}} \subseteq s_{i+1}^{\texttt{true}}$ (Axioms 5) and s_{i+1} will be a normal state (Axioms 4). Furthermore, s'_{j+1} will be terminal iff s_{i+1} is terminal, in which case $g(s'_{j+1}) \subseteq g(s_{i+1})$.
2) If some non-arbiter action in m_i is not `source_legal` then a bad move has occurred (Axioms 4). Hence the `source_base` fluents will persist to the next state (Axioms (6)) which will be an abnormal state (Axioms (4)). Because a run must terminate in a normal state, there must be some minimal $i+1 < l < n$ s.t. s_l is a normal state. The `source_base` fluents will persist through to this state (Axioms (6)). Here all erroneous moves will have been reversed so $s_i^{\texttt{true}} = s_l^{\texttt{true}}$ (i.e., $s_i = s_l$). This process can repeat, but in order to reach termination at some point there must be `source_legal`-only actions taken from a normal state. At that point the first case would apply. □

Propositions 2 and 3 establish that, except for the arbiter correcting moves, the meta-game is a faithful encoding of the original game. However, it should be emphasised that even when the original game is well-formed the meta-game is not. Firstly, there is no termination guarantee for the meta-game. The simplest example of this is the case of a player that simply repeats the same incorrect move while the arbiter patiently and endlessly corrects this mistake. Secondly, the meta-game is not weakly-winnable or monotonic since no goal values have been specified for the arbiter role.

This lack of well-formedness is not in itself problematic, as the intention is for the meta-game to be used in the specialised context of a robot execution monitor rather than to be played in the GGP competition. Nevertheless, satisfy these properties, and in particular the guarantee of termination, can have practical uses. Consequently, extensions to construct a well-formed meta-game are described in Section 6.

5 Planning

The current requirements of the environment model Σ_{env} ensure that the arbiter only has a single action that it can take to correct an invalid move: it simply reverses the move. This allows even the simplest GGP player, for example one that chooses any legal move, to play the arbiter role effectively. However, this simplification is only possible under the assumption that all unexpected changes are indeed reversible and that Σ_{env} explicitly provides the move. This cannot always be satisfied when modelling a physical environment.

Instead, an existing environment model, such as the board environment of Figure 3, can be extended to a richer model that might require multiple moves to recover a normal game state. For example, correcting a toppled piece could conceivably require two separate actions; firstly, to place the piece upright and subsequently to move it to the desired position.

Secondly, some environments simply cannot be modelled with only reversible moves. For example, reversing an invalid move in CONNECTFOUR, a game where coloured discs are slotted into a vertical grid, would involve a complex sequence of moves, starting with clearing a column, or even the entire grid, and then recreating the desired disc configuration.

Our approach can be easily extended to allow for these more complex environment models. Most importantly, in an abnormal state the arbiter needs to be allowed to execute all physically possible moves. Formally, we remove the `env_reverse` predicate and replace the first axiom of (3) by

```
(<= (legal arbiter ?m)
    (env_legal ?m) meta_correcting_mode)
```
(7)

We also substitute the second and third axioms of (4) by

```
(<= (next (meta_prev ?f))
    (true ?f) (env_base ?f)
    (meta_player_bad_move ?p ?m))
```

This replaces `meta_currently_correcting`(m) by the new fluent `meta_prev`(f), whose purpose is to preserve the environment state just prior to a bad move. The goal is to recover this state. Hence, the game remains in correcting mode as long as there is a discrepancy between this and the current environment state:[4]

```
(<= (next (meta_prev ?f))
    (true (meta_prev ?f))
    meta_correcting_mode)
(<= (next (meta_correcting ?m))
    (true (meta_correcting ?m))
    (true (meta_prev ?f))
    (not (env_next ?f)))
(<= (next (meta_correcting ?m)
    (true (meta_correcting ?m))
    (env_next ?f)
    (not (true (meta_prev ?f))))
```

Consider an environment model Σ_{env} for CONNECTFOUR that includes the actions of clearing an entire column and of dropping a single disk in a column. The arbiter can always perform a sequence of actions that brings the physical game back into a valid state after a wrong disk has been slotted into the grid. This flexibility requires the execution monitor to plan and reason about the consequences of actions in order to return to the appropriate normal state. As part of the meta-game, this behaviour can be achieved by defining a goal value for the arbiter that is inversely proportional to the number of occurrences of bad states in a game. A skilled general game-playing system taking the role of the arbiter would then plan for the shortest sequence of moves to a normal state.

Worthy of note is that the generalised axiom (7) also accounts for the case of failed actions of the execution monitor itself. Whenever this happens, the game remains in a bad state and the arbiter would plan for correcting these too.

6 Termination and Well-Formed Meta-Games

As highlighted in Section 4, currently the constructed meta-games are not well-formed. This can be undersirable for a number of reasons. For example, a lack of guaranteed game termination can be problematic for simulation based players that require terminating simulation runs (e.g., Monte Carlo tree search [Björnsson and Finnsson, 2009]).

A simple mechanism to ensure game termination is to extend Σ_{em} with a limit on the number of invalid moves that can be made by the non-arbiter roles. Firstly, a strikeout counter is maintained for each non-arbiter player.

```
(<= (init (strikeout_count ?r 0))
    (role ?r) (distinct ?r arbiter))
(<= (next (strikeout_count ?r ?n))
    (not (meta_bad_move ?r))
    (true (strikeout_count ?r ?n)))
(<= (next (strikeout_count ?r ?n))
    (true (strikeout_count ?r ?m))
    (meta_bad_move ?r) (++ ?m ?n))
(++ 0 1) ... (++ 2 3)
```

Next, a player strikes out if it exceeds some number (here three) of invalid moves, triggering early game termination.

```
(<= (strikeout ?r)
    (true (strikeout_count ?r 3)))
(<= strikeout (strikeout ?r))
(<= terminal strikeout)
```

Beyond this termination guarantee, we can also ensure that games are weakly winnable and goal scores are monotonic, by defining goal values for the arbiter in all reachable states.

```
(<= (goal arbiter 100) terminal)
(<= (goal arbiter 0) (not terminal))
```

However, early game termination can be problematic if we treat the meta-game as the final arbiter of who has won and lost the embedded game. A player with a higher score could intentionally strikeout in order to prevent some opponent from improving their own score. To prevent this we, firstly, extend Definition 1 to replace every occurrence of `goal`(r, v) by `source_goal`(r, v). Next, axioms are introduced to remap `goal` scores for the original players, ensuring that a strikeout player receives the lowest score.

```
(<= (goal ?r 0) (role ?r) (not terminal))
(<= (goal ?r ?n)
    (source_goal ?r ?n) terminal
    (not strikeout))
```

[4] Since bad moves are no longer taken back individually, if they happen simultaneously then for the sake of simplicity all instances of `meta_correcting`(m) remain true until the preceding valid state has been recovered.

```
(<= (goal ?r 0) (strikeout ?r))
(<= (goal ?r 100)
    (role ?r) (distinct arbiter ?r)
    (not (strikeout ?r)) strikeout)
```

Proposition 4 *For any syntactically correct and well-formed game Σ_G and corresponding environment model Σ_{env}, the extended meta-game Σ_{meta} is also syntactically correct and well-formed.*

Proof: It can be verified that the extra rules don't change the syntactic properties of Σ_{meta}. Similarly, it is straightforward to observe that Σ_{em} now guarantees termination, is weakly-winnable, and has monotonic goal scores. □

Note that the modification to allow for early termination does have consequences for Proposition 3. Namely, it will only hold if the run of a meta-game terminates in a normal state. Trivially, if a meta-game terminates early then the embedded game will never complete and there will be no terminal state in the corresponding run of the original game.

7 Conclusion and Future Work

We have presented an approach for embedding axiomatisations of games for general game-playing robots into meta-games for execution monitoring. This allows for the seamless encoding of recovery behaviours within a meta-game, enabling a robot to recover from unexpected events such as failed action executions or (human) players making unsanctioned game moves. The approach is general enough to encompass a full range of behaviours. From simple, prescribed strategies for correcting illegal moves through to complex behaviours that require an arbiter player to find complex plans to recover from unexpected events. Alternatively, even for specifying early-termination conditions, for example, when a game piece falls off the table or a frustrated human opponent throws it away. Our method can moreover be applied to games axiomatised in GDL-II [Thielscher, 2010], which allows for modelling uncertain and partially observable domains for an arbiter and hence to account for, and recover from, execution errors that a monitor does not observe until well after they have occurred. An interesting avenue for future work is to consider the use of an Answer Set Programming (ASP) solver (see for example [Gebser *et al.*, 2012]) to help in this failure detection and recovery. Based on an observation of a failure state the ASP solver could generate candidate move sequences as explanations of how the game entered this state from the last observed valid game state. Finally, the general concept of automatically embedding a game into a meta-game has applications beyond execution monitoring, for example, as an automatic game controller that runs competitions and implements rules specific to them such as the three-strikes-out rule commonly employed at the annual AAAI contest for general game-playing programs [Genesereth and Björnsson, 2013].

Acknowledgements

This research was supported under Australian Research Council's (ARC) *Discovery Projects* funding scheme (project number DP 120102144). The second author is also affiliated with the University of Western Sydney.

References

[Apt *et al.*, 1987] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1987.

[Björnsson and Finnsson, 2009] Y. Björnsson and H. Finnsson. CADIAPLAYER: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[De Giacomo *et al.*, 1998] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *Proceedings of KR*, pages 453–465. Morgan Kaufmann, 1998.

[Fichtner *et al.*, 2003] M. Fichtner, A. Großmann, and M. Thielscher. Intelligent execution monitoring in dynamic environments. *Fundam. Inform.*, 57(2-4):371–392, 2003.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on AI and Machine Learning. Morgan & Claypool Publishers, 2012.

[Genesereth and Björnsson, 2013] M. Genesereth and Y. Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–111, 2013.

[Genesereth *et al.*, 2005] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Hähnel *et al.*, 1998] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX — bridging the gap between logic (GOLOG) and a real robot. In *Proceedings of KI*, volume 1504 of *LNCS*, pages 165–176. Springer, 1998.

[Lloyd and Topor, 1986] J.W. Lloyd and R.W. Topor. A basis for deductive database systems II. *The Journal of Logic Programming*, 3(1):55 – 67, 1986.

[Love *et al.*, 2006] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. Stanford University, 2006.

[Pettersson, 2005] O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[Rajaratnam and Thielscher, 2013] D. Rajaratnam and M. Thielscher. Towards general game-playing robots: Models, architecture and game controller. In *Proceedings of AI*, volume 8272 of *LNCS*, pages 271–276. Springer, 2013.

[Schiffel and Thielscher, 2010] S. Schiffel and M. Thielscher. A multiagent semantics for the Game Description Language. In *Agents and AI*, volume 67 of *CCIS*, pages 44–55. Springer, 2010.

[Thielscher, 2009] M. Thielscher. Answer set programming for single-player games in general game playing. In *Proceedings of ICLP*, volume 5649 of *LNCS*, pages 327–341. Springer, 2009.

[Thielscher, 2010] M. Thielscher. A general game description language for incomplete information games. In *Proceedings of AAAI*, pages 994–999, 2010.