Hope for the Best, Prepare for the Worst: Multi-tier Control for Adaptive Systems

Nicolás D'Ippolito[†⋆], Víctor Braberman[⋆], Jeff Kramer[†],
Jeff Magee[†], Daniel Sykes[†], Sebastian Uchitel[†⋆]
[†] Department of Computing, Imperial College London, UK
[⋆] Departamento de Computación, FCEN, Universidad de Buenos Aires, Argentina
{n.dippolito,j.kramer,j.magee,d.sykes,s.uchitel}@imperial.ac.uk, vbraber@dc.uba.ar

ABSTRACT

Most approaches for adaptive systems rely on models, particularly behaviour or architecture models, which describe the system and the environment in which it operates. One of the difficulties in creating such models is uncertainty about the accuracy and completeness of the models. Engineers therefore make assumptions which may prove to be invalid at runtime. In this paper we introduce a rigorous, tiered framework for combining behaviour models, each with different associated assumptions and risks. These models are used to generate operational strategies, through techniques such controller synthesis, which are then executed concurrently at runtime. We show that our framework can be used to adapt the functional behaviour of the system: through graceful degradation when the assumptions of a higher level model are broken, and through progressive enhancement when those assumptions are satisfied or restored.

Categories and Subject Descriptors

D2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Design, Reliability, Theory

Keywords

Adaptive systems, controller synthesis, planning, reliability

1. INTRODUCTION

Many approaches for adaptive systems make use of various kinds of models, both in the design phase, and as an explicit artefact used during execution to guide decision-making processes. These models, these abstractions, are *necessarily* idealisations of the system or world that they describe, created with a certain purpose in mind [47]. The reasoning behind such idealisation is well understood: a 'complete' model would be intractable, both to create and to use computationally; and, moreover, there is a tacit understanding that the omitted detail is the very detail which is potentially incompatible with the purpose of the model. In other words, we make simplifying assumptions to ensure that the model supports the purpose we have in mind.

Any model, then, introduces uncertainty [21] and risk into the engineering process. We cannot be certain that our chosen assumptions are valid (and will always hold), risking undesirable consequences when the assumptions are broken. One of the aims behind research in adaptive systems is to mitigate this risk (arising from design-time uncertainty) by making decisions with information available at runtime, particularly in difficult, unpredictable environments that are liable to change [4].

In particular, certain approaches for adaptive systems use runtime representations of their operating environment [30, 22] and algorithmic techniques (e.g. planning [5, 44] and controller synthesis [38, 18]) that can produce, from environment models, operational strategies for the system to achieve its goals. Such models are idealised because the algorithmic complexity of dealing with a 'complete' model of the real world would be prohibitive in time and space, and because stronger assumptions can be relied upon to support more sophisticated system goals. The stronger the assumptions are, the more enhanced functionality can be guaranteed. Conversely, very little can be guaranteed in a world where 'everything' can go wrong.

Despite the important trade-offs involved in fixing environment assumptions, existing work on adaptive systems proposes designs in which only one environment model is permitted [44, 18, 45, 20] hence fixing the level of risk to be taken and the goals that can be achieved. Further, the resulting system loses robustness with respect to invalid assumptions: when the environment does not behave as assumed, either the system fails completely or continues executing but guarantees can no longer be provided.

In this paper we address the general question of how the planning layer in adaptive systems should be designed to support multiple environment models with different associated risk and guaranteeable levels of functionality, and how such a design can be used to provide graceful degradation and progressive functional enhancement at runtime according to the validity of the assumptions each model makes.

We propose a general multi-tier modelling, planning and enactment framework for adaptive systems and describe the conditions under which it guarantees graceful degradation and progressive enhancement. The framework is general in various dimensions. First, it defines the relations that must hold between tiers without fixing the specific techniques used at each tier to automatically construct goal achievement strategies, i.e. controllers. Secondly, it does not fix the number of tiers required. Thirdly it supports both state-sensing adaptive strategies such as in reactive plans [44] and event-sensing strategies such as in [18], or combinations of both.

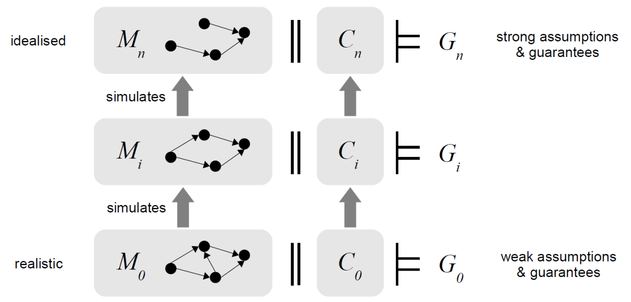

Figure 1: Multi-tier control problem

The key requirements the framework imposes in order to guarantee graceful degradation and progressive enhancement are that (see Figure 1): (i) higher-level environment models must be simulated by lower-level environment models, capturing a notion of idealisation of higher-level models; (ii) higher-level controllers used to achieve enhanced functionality must be simulated by controllers used at lower levels, ensuring a consistent overall strategy; (iii) the runtime infrastructure must be capable of detecting when an inconsistency between an environment model (in any tier) and the runtime environment occurs; (iv) a sound automated replanning procedure for each tier that is expressive enough to deal with the system goals for its tier must be provided, allowing progressive system enhancement after inconsistencies have been detected.

The framework's enactment procedure continuously monitors the environment and concurrently executes a stack of controllers giving priority to the controller of the uppermost enabled tier. It continuously updates the current state of all environment models based on monitored actions and sensed state, disabling tiers should an inconsistency be detected. At any point, to improve the provided functionality, a replanning attempt may be made for the lowest disabled tier. Based on the current state of the enabled tier immediately below, the state of the disabled tier is approximated and an attempt is made to build a controller that will work despite the uncertainty about the current state of the tier. Should a controller exist, it is put into the controller hierarchy and the tier is enabled.

In addition to an analytical validation of our approach, providing theoretical results for graceful degradation and progressive enhancement, we examine feasibility by instantiating the framework with three different combinations of goal types and synthesis algorithms. We show how the additional constraints on synthesis imposed by the framework can be achieved in general from multi-tier safety properties and how liveness can also be incorporated. The last instantiation also shows how very robust techniques such as reactive plans can be combined with less robust but highly expressive memory-based synthesis techniques.

The rest of the paper is summarised as follows. A motivating example is discussed in Section 2 and background definitions are given in Section 3. The framework is presented in Section 4 and validation is presented in Section 5. We then end with a discussion, related work and conclusions.

2. MOTIVATING EXAMPLE

We first motivate our work by considering the control system for an automated warehouse that fulfils orders consisting of a number of products selected from a wide range stocked by the warehouse. One or more mobile robots move products from the supplier area into their storage slots, and fulfil the orders by collecting products into a bag or parcel for delivery to the customer. Complete orders are moved to the delivery area. Suppliers are obliged to supply the warehouse with products, but supply issues or demand for some products may mean that the warehouse runs out of stock. In this case, it is permissible that incomplete orders are delivered. The warehouse is however obliged to store all products that are supplied. Within the warehouse a number of technical issues can arise. Factors such as wheel slip can impact the reliability of robot navigation, so that robots may arrive in unintended locations. In addition, the product storage slots can report an erroneous stock level, and can jam when depositing an item on a robot.

We wish to build a strategy (C_3) that guarantees the goal property (G_3) that the warehouse continually delivers complete orders. This strategy would involve the robot's visiting all the appropriate storage slots for the products in an order, taking an item from those slots, taking the complete order to the delivery point, and also moving products from the supply area at relevant times to maintain stock. Such a strategy makes various assumptions: (i) all products are (eventually) in stock; (ii) the storage slots report the correct stock levels and (iii) do not jam; and (iv) the robot moves reliably to all necessary locations. The environment usually satisfies these assumptions but can 'win' by breaking these assumptions in exhibiting technical problems or by not supplying products. This will cause C_3 to fail from time to time.

However, there is a weaker goal (G_2), which is that orders with at least one product in stock are continually delivered. The strategy (C_2) for this goal makes weaker assumptions: (i) one product is (eventually) in stock; (ii) one storage slot reports the correct stock level and (iii) does not jam; and (iv) the robot moves reliably to all necessary locations. The environment can however continue to frustrate the system with continual failure of the storage slots or the robot.

There is an even weaker goal (G_1) that can be maintained with failing storage slots, which is that supplied products are moved from the supply area to the storage slots. The corresponding strategy (C_1) assumes only that the robot can move reliably between the supply area and the storage slots.

Suppose that the system is initially executing strategy C_3, and one of its assumptions is violated, such that G_3 cannot be achieved. A common solution is to monitor the environment and detect when the assumptions are violated [48, 39]. But what should the system do when a violation is detected? How can it proceed from its current state? We would like the system to be designed in such a way that it continues from its current state with a strategy for the weaker G_2, or, in the worst case, ensures G_1 is maintained until an engineer can resolve the problem. In other words, the functionality of the system should undergo *graceful degradation* when environmental assumptions are violated.

In this example, the assumptions of C_3 are likely to hold most of the time, and so, after a period of degraded functionality, we would like the system to attempt to start providing G_3 again, without needing to reset or stop the system completely. We call this process *progressive enhancement.*

Our objective in this work is to develop an approach for adaptive systems that deals with multiple tiers of goals and assumptions, giving multiple tiers of functionality.

3. BACKGROUND

We use labelled transition Kripke structures to describe behaviour of the environment and the system. Transitions are labelled with names of actions, some of which the system can monitor or control. States have associated propositions which also may be monitored by the system.

DEFINITION 3.1. (Labelled Transition Kripke Structure) *A* labelled transition Kripke structure *(LTKS) is* $E = (S, A, P, \Delta, v : S \to 2^P, S_0)$*, where S is a finite set of states,* $A = A_C \uplus A_M$ *is the* communicating alphabet *which we assume is partitioned into controlled and monitored actions, P is a set of propositions,* $\Delta \subseteq (S \times A \times S)$ *is a transition relation,* $v : S \to 2^P$ *is a valuation function for states, and* $S_0 \subseteq S$ *is the set of initial states. A trace of E is* $\pi = s_0, \ell_0, s_1, \ell_1, \cdots$*, where s_0 is an initial state of E and, for every* $i \geq 0$*, we have* $(s_i, \ell_i, s_{i+1}) \in \Delta$*. We denote the set of infinite traces of E by* $\mathrm{Tr}(E)$*.*

In order to describe the enactment of controllers in later sections, we introduce notation to describe possible updates of the initial state of an LTKS

DEFINITION 3.2. (LTKS Updates) *Given an LTKS* $E = (S, A, P, \Delta, v, S_0)$ *with* $A = A_C \uplus A_M$*, we write* $E \xrightarrow{a_c}$ *for* $a_c \in A_C$ *if for all* $s \in S_0$ *there exists* $(s, a_c, s') \in \Delta$*. Otherwise we write* $E \overset{a_c}{\not\longrightarrow}$*. If* $a_m \in A_M$ *we (overload and) write* $E \xrightarrow{a_m}$ *if there is* $s \in S_0$ *such that there exists* $(s, a_m, s') \in \Delta$*. Otherwise we write* $E \overset{a_m}{\not\longrightarrow}$*. If* $E \xrightarrow{a}$ *for* $a \in A$ *then we write* $E \xrightarrow{a} E'$ *when an a action is taken from all states with outgoing a transitions (i.e.* $E' = (S, A, P, \Delta, v, S'_0)$ *and* $S'_0 = \{s' | (s, a, s') \in \Delta \wedge s \in S_0\}$*). We say that E is consistent with a set of propositions* $V \subseteq P$ *(denoted* $E \xrightarrow{V}$*) if there is* $s \in S_0$ *such that* $v(s) = V$*. We write* $E \xrightarrow{V} E'$ *when E' has a reduced set of initial states with respect to E based on V (i.e.* $E' = (S, A, P, \Delta, v, S'_0)$ *where* $S'_0 = \{s | s \in S_0 \wedge v(s) = V\}$*).*

DEFINITION 3.3. (Parallel Composition) *Let* $M = (S_M, A_M, P_M, \Delta_M, v_M, S_{M_0})$ *and* $E = (S_E, A_E, P_E, \Delta_E, v_E, S_{E_0})$ *be LTKSs with* $A_M = A^M_C \uplus A^M_M$ *and* $A_E = A^E_C \cup A^E_M$*.* Parallel composition $\|$ *is a symmetric operator such that* $E\|M$ *is the LTKS* $E\|M = (S, A_E \cup A_M, P_M \uplus P_E, \Delta, v, S_0)$*, where* $S = \{(s_e, s_m) \in S_E \times S_M | v(s_m) \cap P_E = v(s_e) \cap P_M\}$*,* $S_0 = \{(s_e, s_m) \in S | s_e \in S_{E_0} \wedge s_M \in S_{M_0}\}$*,* $v((s_e, s_m)) = v_M(s_m) \cup v_E(s_e)$*, and Δ is the smallest relation that satisfies the rules below, where* $\ell \in A_E \cup A_M$*:*

$$\frac{E \xrightarrow{\ell} E'}{E\|M \xrightarrow{\ell} E'\|M}\ \ell \in A_E \setminus A_M \qquad \frac{M \xrightarrow{\ell} M'}{E\|M \xrightarrow{\ell} E\|M'}\ \ell \in A_M \setminus A_E$$

$$\frac{E \xrightarrow{\ell} E',\ M \xrightarrow{\ell} M'}{E\|M \xrightarrow{\ell} E'\|M'}\ \ell \in A_E \cap A_M$$

We restrict attention to states in S that are reachable from S_0 using transitions in Δ.

DEFINITION 3.4. (Simulation) *Let $\wp$ be the universe of all LTKSs with communicating alphabet A. Given E and F in $\wp$, we say that E simulates F, written* $E \geq F$*, when* (E, F) *is contained in some* simulation relation $R \subseteq \wp \times \wp$ *such that for all* $\ell \in A$ *and* $(E, F) \in R$ *we have* $E \xrightarrow{\ell} E'$ *implies that there is F' such that* $F \xrightarrow{\ell} F' \wedge \forall s_E \in \mathrm{init}(E') \cdot \exists s_F \in \mathrm{init}(F') \cdot v'_E(s_E) = v'_F(s_F) \wedge (E', F') \in R$*.*

We are interested in distinguishing controlled from monitored actions and ensuring that when composing LTKSs, one controller never blocks the other's controlled actions. This notion captured by *interface automata* [17] is used here for LTKSs.

DEFINITION 3.5. (Legal Environment) *Given LTKSs E and F defined over communicating alphabets A_E and A_F that are partitioned into the sets* (A_{E_C}, A_{E_M}) *and* (A_{F_C}, A_{F_M}) *of controlled and monitored actions such that* $A_{E_C} = A_{F_M}$ *and* $A_{F_C} = A_{E_M}$*, we say that F is a legal environment for E if for all* $(s_E, s_F) \in E\|F$ *it holds that* $\Delta_E(s_E) \cap A_{E_M} \supseteq \Delta_F(s_F) \cap A_{F_C}$ *and* $\Delta_E(s_E) \cap A_{E_C} \subseteq \Delta_F(s_F) \cap A_{F_M}$*.*

Control problems are typically defined over a simplified form of LTKS in which the set of propositions is empty.

DEFINITION 3.6. (Labelled Transition Systems) *A* Labelled Transition System *(LTS) is an LTKS* $E = (S, A, P, \Delta, v, S_0)$*, where* $P = \emptyset$*. We will sometimes refer to an LTS as a tuple* $E = (S, A, \Delta, S_0)$*.*

The problem of controller synthesis is to automatically produce a state machine that restricts the occurrence of actions it controls, based on its observation of the actions that have occurred and the propositions it can sense from the current state of the environment, so that when deployed in a given environment a given goal holds. We do not prescribe the logic used to describe goals nor the algorithms used to solve control problems. Hence, we simply assume a satisfaction relation $\models$ for LTSs and the logic used to express goals. We abuse notation referring to control problems for LTKS when we are referring to the control problem resulting from removing state valuations from the LTKS.

DEFINITION 3.7. (LTS Control) *Given a specification for an environment in the form of an LTS* $E \in \wp$ *and a goal G expressed in some logic $\mathcal{L}$ and a satisfaction relation* $\models \subseteq \wp \times \mathcal{L}$*, a solution for the LTS control problem* $\mathcal{E} = \langle E, G \rangle$ *is a deterministic LTS C such that C is a legal environment for E,* $E||C$ *is deadlock free, and* $E||C \models G$*.*

4. MULTI-TIER CONTROL

We now define a general framework for adaptive systems that supports graceful degradation and progressive enhancement. We first explain the framework's preconditions and initialisation procedure, and then explain how it is executed and finally discuss more formally its guarantees regarding graceful degradation and progressive enhancement. A general overview of the framework is given in Figure 2. The overview is linked to the pseudocode in Figure 3 representing the runtime behaviour of the framework by means of the Ⓐ . . . Ⓜ markers to aid the reader. The pseudocode in turn uses definitions provided in this section.

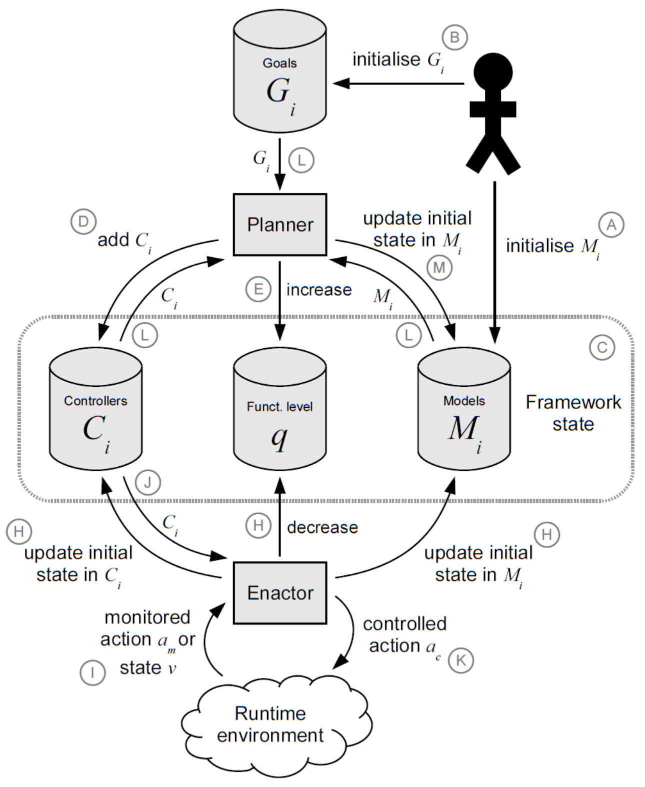

Figure 2: Framework

4.1 Preconditions and Initialisation

The framework requires initialisation through human intervention: a set of environment models and goals must be provided. The environment models are expected to be ranked in terms of the degree of idealisation of the environment they represent. We assume that the environment model M_0 is the least idealised model of the environment and require that environment models further up the hierarchy allow strictly less behaviour. This can be formally captured via a simulation relation, $M_i \geq M_j$ for $i < j$. Note that in order to simplify presentation we require environment models to have the same communicating alphabets partitioned identically into controlled and monitored actions. In summary, the less idealised the environment model is, the more behaviour (in terms of unexpected actions and non-determinism) may arise.

Each tier ($0 \leq i \leq n$) has an associated goal (G_i) to be achieved by the system assuming that the environment conforms to the environment model for that tier (M_i).

Each tier introduces a control problem $\mathcal{E}_i = \langle M_i, G_i\rangle$ to be solved. However, there is an additional constraint: each controller must be simulated by controllers in lower tiers ($C_i \geq C_j$ for $i \leq j$). Intuitively, this requires that a controller at a higher tier never do something that a controller of a lower tier would not do, thus ensuring that if a controller at a higher tier must be stopped, because the assumptions for its tier are discovered to not hold, decisions made by it up to that point have been consistent with lower tier controllers. In other words, it allows for graceful degradation, falling back to lower tier controllers when needed.

The requirement that the controller hierarchy preserve simulation may make solving the set of control problems $\mathcal{E}_i = \langle M_i, G_i\rangle$ more difficult. In particular the existence of a solution for each $\mathcal{E}_i$ does not in general guarantee that there exists a solution for each $\mathcal{E}_i$ that also preserves simulation between controllers. In Section 5 we discuss how this can be achieved for a variety of controller synthesis and planning algorithms. Note that an alternative, naïve requirement would be that higher-level goals entail lower-level goals ($G_i \implies G_j$ for $j \leq i$). However, this requirement is too weak as different tiers may chose different strategies to achieve lower-level goals.

We assume that the environment models and goals for tiers 1 to n are provided by engineers. However, tier 0 is set automatically (lines 2 and 3) to solve the control problem $\mathcal{E}_0 = \langle M_0, \texttt{true}\rangle$ where M_0 is the most general environment model (an LTKS where states represent all possible results of sensing the environment and all states are connected for all actions). M_0 is guaranteed to simulate any M_1 manually provided. The controller C_0 used to solve $\mathcal{E}_0$ is the most permissive controller, i.e. one that allows any controlled action at any point in time. C_0 is guaranteed to not restrict the possible solutions of $\mathcal{E}_1$ as it will simulate any controller.

We assume a planning operation (line 8 and Definition 4.1) that is executed bottom-up through the tiers (lines 6-10). The operation attempts to build a controller that solves the control problem in a tier while being simulated by the controller for the tier immediately below.

Definition 4.1. (Planning Operation) *The planning operation applied to tier $i > 0$ either returns* `null` *or a controller C_i such that it is a solution to the control problem $\mathcal{E}_i = \langle M_i, G_i\rangle$ and $C_i \geq C_{i-1}$.*

Note that the `initialise` procedure does not require that the planning operation be successful for all tiers (line 10). It is possible that the system starts in a degraded mode and later and progressively enhances its behaviour as controllers for higher tiers are built.

The algorithm in Figure 3 avoids, for simplicity, explicit treatment of synchronisation between the Enactor and Planner processes. We assume an implementation of the Planner which minimises the locking time of S so as not to block the Enactor's execution. One such implementation would be that S is only locked once a viable plan has been built and at the end of the execution of the Enactor's while block. To avoid interference, the new controller is inserted only if S has not changed while the Planner was computing the controller (i.e. no sensed or actuated actions).

4.2 Framework Execution

We now describe how the framework is executed. The key data structure the framework relies upon is called the *framework state.* It contains the environment models $M_0, \dots, M_n$ which have their initial states updated as controlled actions are actuated or changes in the runtime environment are sensed. The structure also contains controllers $C_0, \dots, C_n$ which are used to decide which controlled action to take and whose initial states are also updated. In addition, the data structure records the current functional level (q) of the system (i.e. the highest level for which the runtime environment is believed to conform to the environment model).

Definition 4.2. (Valid Framework State) *We define the framework state as a tuple $(M_0, \dots, M_n, C_0, \dots, C_n, q)$ where*

```
1  initialise($M_1$ ... $M_n$, $G_1$ ... $G_n$)
2    $M_0$ = Planner.getModel0();
3    $C_0$ = Planner.getController0();
4    s = (($M_0$, $M_1$ ... $M_n$), ($C_0$), 0); Ⓐ
5    q = 0;
6    do
7      q++;
8      $C_q$ = Planner.plan(s, q); Ⓛ
9      s.setController($C_q$, q); Ⓓ
10   while (c $\neq$ null && q $\leq$ n);
11   s.setFuncLevel(q); Ⓔ
12   fork enactor();
13   fork planner();
14
15 planner()
16   while (true)
17     q = s.getFuncLevel();
18     if (q < n)
19       $M_q$ = s.getModel(q);
20       $M_{q+1}$ = s.getModel(q+1);
21       states = $M_{q+1}$.getStatesSimulatedBy($M_q$);
22       if (! states.empty())
23         $M_{q+1}$.setInitStates(states);
24         $C_{q+1}$ = Planner.plan(s, q+1); Ⓛ
25         if ($C_{q+1}$ $\neq$ null)
26           s.setController($C_{q+1}$, q+1); Ⓓ
27           s.setModel(q+1, $M_{q+1}$); Ⓜ
28           s.setFuncLevel(q+1);
29
30 enactor()
31   while (true)
32     if (s.getFuncLevel() $\geq$ 1)
33       $v$ = runtime.senseState(); Ⓘ
34       s = (s $\triangleright v$); Ⓗ
35       if (s.getFuncLevel() $\geq$ 1)
36         $a_m$ = runtime.senseAction(); Ⓘ
37         if ($a_m$ $\neq$ null)
38           s = (s $\triangleright a_m$); Ⓗ
39         else
40           $C_q$ = s.getController(s.getFuncLevel());
41           $a_c$ = $C_q$.pickEnabledContAction(); Ⓙ
42           if ($a_c$ $\neq$ null)
43             runtime.actuate($a_c$); Ⓚ
44             s = (s $\triangleright a_c$); Ⓗ
```

Figure 3: Framework's runtime behaviour

M_i are the environment models and C_i are the controllers with $0 \leq i \leq n$, and $0 \leq q \leq n$. We say that a framework state is valid (with respect to goals $G_0, \ldots, G_n$) if for all $0 \leq i \leq q$ the following hold: i) C_i is a solution to $\mathcal{E}_i = \langle M_i, G_i \rangle$, ii) $M_j \geq M_i$ for $0 \leq j \leq i$, and iii) $C_j \geq C_i$ for $0 \leq j \leq i$. We say that an framework state is providing functional level i if $i = q$.

Succinctly, the enactment of the controller stack stored in the framework state is performed by executing *all* controllers concurrently whilst prioritising the decision of higher tier controllers. The Enactor executes a loop in which the runtime environment is sensed (lines 33 and 36) or actuated upon (line 43) and the framework state is updated accordingly (lines 34, 38 and 44).

The framework state update operation ($S \triangleright x$ where x represents either a monitorable action or sensed runtime environment state) updates each environment model and controller in the data structure and, crucially, decrements the functional level q appropriately to capture the highest level for which the sensed behaviour is consistent with the environment model.

Definition 4.3. (Framework State Update) *Let $S = (M_0, \ldots, M_n, C_0, \ldots, C_n, q)$ be a valid framework state. An update of S for x (denoted $S \triangleright x$) where x is an action a or a valuation V is a framework state S' such that $S' = (M'_0, \ldots, M'_n, C'_0, \ldots, C'_n, q')$ where $0 \leq q' \leq q$ is the largest q' such that $M_{q'} \xrightarrow{x}$, and where $M_j \xrightarrow{x} M'_j$ and $C_j \xrightarrow{x} C'_j$ for $0 \leq j \leq q'$, and $M_j = M'_j$ and $C_j = C'_j$ for $q' < j \leq n$.*

The `enactor` procedure actually first senses the runtime environment state and updates the framework state, then it senses if any monitorable actions have occurred and updates the framework state. Then if no monitorable actions have occurred it picks a controlled action to actuate. It is important to note that the controlled action is chosen based on the actions that the controller on tier q allows (lines 40 and 41). Thus, priority is given to the highest tier controller for which the corresponding environment model is in a consistent state with the runtime environment.

Thus, the framework is designed to support situations in which the assumptions made in the various environment models do not hold.

Definition 4.4. (Inconsistency) *An* inconsistency *occurs when the runtime environment is not simulated by the environment model of tier i and the system is running at a functional level $q \geq i$. Definition 4.3 detects inconsistency between environment models and runtime environment when (i) a monitored action ℓ occurs for which no outgoing transition from the current M_q state exists (i.e. $M_i \overset{\ell}{\not\longrightarrow}$) or when (ii) the sensed state ($V \subseteq P$) of the environment is inconsistent with the propositions of the current state of the environment model (i.e. $M_i \overset{V}{\not\longrightarrow}$).*

When an inconsistency is detected, the Enactor will disable the tier by lowering the functionality level to a $q' < i$ and continue executing with a degraded service in which $C_{q'}$ is used to decide which actions to take and hence only goals up to $G_{q'}$ will be guaranteed.

Note that as M_0 and C_0 are the most general environment and controller models and G_0 is `true`, it is always the case that setting $q = 0$ yields a valid framework state.

The proposed framework supports *progressive functional enhancement* by attempting to create new controllers for tiers above the current functional level q (see `planner` in Figure 3). The framework does not prescribe when replanning must be attempted. In principle this can be done at any time, however in practice replanning may be associated with a clock or with heuristics related to environment modelling.

One difficulty of replanning is that once a tier i has been disabled (i.e. $q < i$), the current state of the environment model M_i has since then not been updated (see Definition 4.3). Thus, the current initial state of M_i is invalid. This problem is solved by replanning bottom-up.

Consider the case in which the system is providing functional level $i - 1$. In order to replan, the current state of the environment according to M_i is re-established using the current state of model M_{i-1} and the fact that M_{i-1} is required to simulate M_i. The Planner computes the largest possible initial set of states for M_i such that M_{i-1} simulates M_i (lines 19 and 23). Recall that as tier 0 can never be disabled, replanning is always performed for tiers 1 and above.

Definition 4.5. (Initial State Inference) *Let $M = (S, A, P, \Delta, v, S_0)$ and N be environment models. We say that M' is the result of inferring the initial states of M based on N (denoted $M \downarrow N$) if $M' = (S, A, P, \Delta, v, S'_0)$ is the LTKS with the largest set of initial states such that $N \geq M$.*

If the initial states for tier i can be inferred from tier $i-1$ then the Planner is used to solve the control problem for level i as in Definition 4.1 (line 24). If successful, the framework state is updated with new versions of M_i and C_i (lines 26-27). In addition, the functional level of the system is incremented by one (line 28).

Definition 4.6. (Framework State Enhancement) *Let $S = (M_0, \ldots, M_n, C_0, \ldots, C_n, q)$ be a valid framework state with $q < n$. If $M'_{q+1} = M_{q+1} \downarrow M_q$ and C'_{q+1} are the result of replanning tier $q+1$ for a framework state $(M_0, \ldots, M'_{q+1}, \ldots, M_n, C_0, \ldots, C_n, q)$ then state enhancement results in $(M_0, \ldots, M'_{q+1}, \ldots, M_n, C_0, \ldots, C'_{q+1}, \ldots, C_n, q+1)$*

As indicated above, there is no guarantee that the functional level will be enhanced when attempting to replan. It may be impossible to infer the initial state of the next environment model up or there may not be a solution to the resulting control problem for that tier. Should either be the case, the framework state simply remains unchanged and no enhanced functionality is provided.

4.3 Framework Properties

It is possible to prove that the framework correctly provides graceful degradation and progressive enhancement. A proof is beyond the scope of this paper, however it is relevant to describe the framework's properties, the assumptions it relies upon and discuss their implication[1]. The key result is stated (informally) as follows:

Theorem 4.1. *(Graceful Degradation) If the initial framework state has a functional level q and the runtime environment exhibits behaviour π consistent with M_i with $i < q$ (i.e. $\pi \in \mathrm{Tr}(M_i)$) then all goals up to level i are satisfied (i.e. $\pi \models G_j$ for all $0 \leq j \leq i$).*

Theorem 4.1 relies mainly on the fact the update operation (Definition 4.3) preserves state validity (Definition 4.2) and particularly the functional level is set to exclude any higher-level controller or environment model that cannot follow the behaviour of the runtime environment. The key assumption required for the proof is that the framework executes fast enough so as to never block its environment. In other words we adopt the synchronous hypothesis [9] which is commonplace in literature for reactive systems (e.g. [29]) and is appropriate in the context of architecture-level adaptation for which this framework is proposed.

Progressive enhancement relies on the fact that if the framework state is valid and providing functional level i, a successful replanning procedure (Definition 4.6) at tier $i+1$ yields a valid framework state. The result follows straightforwardly from the definitions (Definition 4.1 and 4.6). Once replanned, Theorem 4.1 is applicable from the resulting framework state. As expected, replanning provides guarantees from the point of replanning onwards and not from the start of the execution of the system.

[1]For the reader's convenience an Appendix with more a formal exposition and proof sketch can be found in [1].

5. VALIDATION

In this section we aim to show the applicability of our framework, by showing that it can support in practice a variety of existing synthesis techniques, which differ in terms of expressiveness and robustness. We do not focus on computational complexity or scalability because—beyond the fact that each new tier introduces a further control problem—these are properties that emerge from the choice of expressive power of the languages used to express goals at each tier and the corresponding synthesis procedures needed to solve the resulting control problems.

We also aim to show that the additional requirements that the framework imposes, chiefly the simulation requirement between environment models and between controllers, can be accommodated in a systematic and modular way, even when the specific synthesis techniques used were not designed to support these additional requirements.

Finally, we demonstrate how our framework can provide a rigorous means to endow such synthesis techniques, which may not have been developed with adaptive systems in mind, with graceful degradation and progressive enhancement.

We discuss three instantiations of the framework, and apply them to the problem domain taken from the example in Section 2. Firstly we apply *backward propagation* [41] to synthesise controllers for three tiers of safety goals, where the uppermost is a bounded liveness goal. We assume no state sensing capabilities (the environment models are LTSs). Secondly, we replace the technique in the uppermost tier with *generalised reactivity* (GR(1)) [37, 18] to synthesise a controller for a liveness goal, while retaining backward propagation for the safety goals in two lower tiers. Finally, we replace the technique in the lowest tier with *reactive planning* [42, 44], retaining backward propagation and GR(1) for the two other tiers. Reactive planning requires that we allow sensing.

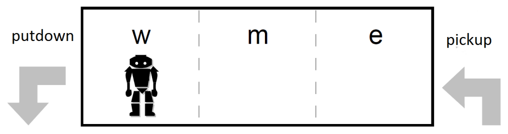

Figure 4: Transporting products.

The problem domain is as follows (Figure 4). A robot moves between three locations, and in certain locations the robot can pick up or put down objects (products). We do not explicitly model the passive objects that picking and putting manipulate. The robot is capable of sensing whether pick and put actions succeed or fail, but it has *no* capacity for sensing its location. We wish to build a controller for this robot that satisfies various requirements. Broadly our objective is to have the robot move between locations w and e, picking and putting at the appropriate moments. In addition there are two safety requirements that are aimed at preserving the physical integrity of the robot and the environment: *i*) picking should not be performed at putting locations and vice-versa, and *ii*) the robot should not pick if it is holding a product, nor put if it is not holding a product.

The assumptions required to achieve these goals differ. For example, succeeding in transporting products requires reliable move, pick and put operations. Avoiding picking at put locations requires reliable inference of the robot's current whereabouts. We define three tiers of models, where the upper tier (tier 3) supports our transportation objective, and the lower tiers support reduced functionality related to physical safety.

Each instantiation is arranged in the same three tiers, with some minor adjustments relating to the synthesis technique employed. Although we describe the tiers top-down, the framework is agnostic as regards methodology. The designer may start with a realistic model and progressively idealise it (bottom-up) towards satisfying a set of stronger goals, or start with a strong goal and progressively weaken it (top-down) towards something that can be satisfied in more realistic models. Nevertheless, as previously noted, once the models and goals are in place, synthesis happens bottom-up.

In all cases we discuss the domain models and goals informally but also accompany these descriptions with Finite State Processes (FSP) expressions [33] to describe domain models and safety goals formally, and Fluent Linear Temporal Logic (FLTL) expressions [24] to formally describe more expressive goals. Note that the FSP and FLTL are included only to illustrate the input to the synthesis tools used, hence background on these languages is not included.

5.1 Backward Propagation

In the first instantiation of the framework, we use backward propagation [41] implemented in LTSA [2] to generate controllers for three tiers of safety goals, where the goal for the uppermost tier is a bounded liveness property.

The backward propagation algorithm synthesises a controller by removing paths from the domain model LTS that lead to the error state. The error state is introduced by composition of the model with observer automata encoding the safety properties.

The backward propagation technique does not necessarily ensure that (individually generated) controllers satisfy the simulation requirement of our framework. Thus for each tier of the hierarchy we use the controller generated for the tier below to restrict the domain model through parallel composition. This ensures that the upper controller does not include any behaviour that is not present (i.e. is unsafe) in the lower controllers. This approach is only complete when using a synthesis technique that guarantees that the generated controllers are maximal (that is, they contain all strategies for achieving their goal), because a non-maximal controller would restrict the upper controller more than strictly necessary. Such is the case for safety properties.

5.1.1 Tier 3 Model (Most Idealised)

We start by modelling the problem domain at the most idealised level, before considering more realistic models. The `GOOD_MAP` is an idealisation of how the environment is expected to behave. The controlled actions are *move*, *pickup* and *putdown*, while the other actions indicate how the environment can respond. In particular, moving in some direction (east or west) is certain to lead to the correct location (w, m, or e). For example, moving east from location w means the robot will arrive in location m. In addition, *pickup* and *putdown* can only succeed in certain locations.

```
GOOD_MAP = (arrive['w] -> MAP['w]),
MAP['w] = (move['e] -> arrive['m] -> MAP['m]
          | move['w] -> arrive['w] -> MAP['w]
          | putdown -> putsuccess -> MAP['w]
          | pickup -> fail -> MAP['w]),
MAP['m] = (move['e] -> arrive['e] -> MAP['e]
          | move['w] -> arrive['w] -> MAP['w]
          | putdown -> putfail -> MAP['m]
          | pickup -> fail -> MAP['m]),
MAP['e] = (move['e] -> arrive['e] -> MAP['e]
          | move['w] -> arrive['m] -> MAP['m]
          | putdown -> putfail -> MAP['e]
          | pickup -> success -> MAP['e]).
```

The model of the (uncontrolled) robot is reused (without changes) in all tiers. We compose the map with a simple model of the robot to produce the idealised domain model.

```
ROBOT =
   (move[Direction] -> arrive[Locations] -> ROBOT
      | pickup -> {success,fail} -> ROBOT
      | putdown -> {putsuccess,putfail} -> ROBOT).
||GOOD_DOMAIN = (ROBOT || GOOD_MAP).
```

5.1.2 Tier 3 Goals

Our primary requirement in this scenario is that the robot repeatedly puts and picks. We encode this as a bounded liveness property and find a controller using the same mechanism as with the other safety goals.

The overall goal is encoded as `BOUNDLIVE`, which states that before the current time bound has been reached (`ENDED`), there must have been successful *pickup* and *putdown* actions. The goal relies on the definition of two *fluents*. For example, `BEEN_PICKING` becomes true after *success* and false after *reset*. The `COUNT` process counts controlled actions, up to the specified bound (`MaxTime`), at which point *ended* must happen.

```
fluent BEEN_PICKING = <success, {reset}>
fluent BEEN_PUTTING = <putsuccess, {reset}>
fluent ENDED = <ended, reset>
ltl_property BOUNDLIVE = [](ENDED ->
                  (BEEN_PICKING && BEEN_PUTTING))
const MaxTime = 6
COUNT = COUNT[0],
COUNT[i:Times] = (CONT -> count[i] -> COUNT[i+1]
                  | ended -> reset -> COUNT),
COUNT[MaxTime+1] = ERROR.
```

We then synthesise a controller, hiding the *arrive* actions as these are not actions the robot can monitor. As a result, the controller will have to infer the current location from the occurrence of other actions, while nonetheless satisfying the goal. We use the idealised `GOOD_DOMAIN` since the bounded liveness goal cannot be achieved in the non-ideal domains. The `LEVEL2_CONTROLLER` is added to the safety goal to ensure that the `LEVEL3_CONTROLLER` achieves the goals of, and can be simulated by, the lower controllers.

```
||LEVEL3_SAFETY = (COUNT || RUNNING || BOUNDLIVE ||
                                 LEVEL2_CONTROLLER).
deterministic ||SOGD3 =
 (GOOD_DOMAIN||LEVEL3_SAFETY)\{arrive[Locations]}.
 controller ||LEVEL3_CONTROLLER = (SOGD3)
           fluents {...} controls {CONT,ended}.
```

The controller is then synthesised using an algorithm that performs backward propagation from the error state.

5.1.3 Tier 2 Model

The next environment model, `BAD_MAP`, is less idealised, and describes the possibility that moving in a certain direction does not always lead to the expected location[2]. In this model, due to the middle location being sunken, moving out of the middle may not be successful and moving into it may lead to an acceleration that makes the robot overshoot its intended location. Later, when the *arrive* action is hidden, BAD_MAP will have a non-deterministic outcome for each *move*. The BAD_DOMAIN simulates the GOOD_DOMAIN, as our framework requires.

```
BAD_MAP = ...
MAP['w'] = (move['e'] ->
                  (arrive['m'] -> MAP['m'] |
                   arrive['w'] -> MAP['w'])
           |move['w'] -> ...
||BAD_DOMAIN = (ROBOT || BAD_MAP).
```

5.1.4 Tier 2 Goals

The non-determinism of robot movements means that eventually reaching either location cannot be guaranteed and thus product transportation cannot be achieved. However, as movements predictably never make the robot go in the opposite direction locations can be sufficiently approximated to avoid picking at the put location and vice versa.

```
fluent AT[x:Locations] = <arrive[x],
                          {move[Direction],reset}>
ltl_property NO_PICK_W = [](pickup -> !AT['w'])
ltl_property NO_PUT_E = [](putdown -> !AT['e'])
```

Again we restrict the environment of this controller using the controller from the tier below in order to ensure simulation, and we hide *arrive* actions making *move* actions non-deterministic.

```
||LEVEL2_SAFETY = (NO_PICK_W || NO_PUT_E
                               || LEVEL1_CONTROLLER).
deterministic ||SOBD2 =
 (BAD_DOMAIN || LEVEL2_SAFETY)\{arrive[Locations]}.
controller ||LEVEL2_CONTROLLER = (SOBD2)
                        fluents {} controls {CONT}.
```

5.1.5 Tier 1 Model (Most Realistic)

The least idealised environment model, VERY_BAD_MAP, allows movements to lead to any location and also includes the possibility that the *pickup* and *putdown* actions fail in the locations where they are meant to succeed (locations e and w respectively). The VERY_BAD_DOMAIN simulates the BAD_DOMAIN.

```
VERY_BAD_MAP = ...
MAP['w'] = (move[Directions] ->
                      arrive[x:Locations] -> MAP[x]
          | pickup -> {success,fail} -> MAP['m'] ...
||VERY_BAD_DOMAIN = (ROBOT || VERY_BAD_MAP).
```

5.1.6 Tier 1 Goals

The tier 2 goal cannot be achieved under the weaker assumptions of tier 1 as location cannot be reliably inferred. However, avoiding picking (putting) when holding (not holding) a product is still possible in this harsher environment.

```
fluent PICKED = <success, putsuccess>
ltl_property PICKONCE = []!(PICKED && pickup)
ltl_property PUTONCE = []!(!PICKED && putdown)
||ALTERNATE = (PICKONCE||PUTONCE).
```

Having defined LEVEL1_SAFETY goal, we can now synthesise a controller that is safe in VERY_BAD_DOMAIN.

[2]This can be due to wheel slip, or, in the case of our Nao H25 humanoid robot, which performs *trilateration* using visually-identified landmarks, due to camera noise, unstable motion, and an obscured line of sight.

5.1.7 Degradation & Enhancement

The resulting hierarchy of controllers provides a control system in which the most demanding bounded liveness goal (picking and putting) can be achieved if the environment encountered at runtime is compliant. If the runtime environment behaves as the BAD_DOMAIN or VERY_BAD_DOMAIN, the system is nonetheless guaranteed to achieve its safety goals. In other words, the system makes a best effort, given the uncertainty about the runtime environment.

In order to observe how the controller stack operates degrades and enhances its functional level we simulated the composition of $Enactor(S)$ with various environments. We used a model checker to generate traces that lead to Enactor states in which service was degraded. These traces, when the runtime environment is the VERY_BAD_DOMAIN, reveal situations where the tier 3 environment is unable to follow. One such trace is *arrive.w*, *move.e*, *arrive.w*, *count*.0, *move.e*, *arrive.w*, *count*.1, *pickup*, $fail$. In this case, the tier 3 controller expects that after performing two *move.e* actions that the robot will be in location e, and so *pickup* must succeed. In fact, the non-determinism in the VERY_BAD_DOMAIN means that the robot is in location w and *pickup* must fail. At this point the Enactor disables the LEVEL3_CONTROLLER and continues with level 2.

At some later stage, the Enactor may attempt to achieve its level 3 goal again. Interestingly, as the initial state of the GOOD_DOMAIN is unknown and little can be inferred from the current state of BAD_DOMAIN, the resulting controller for tier 3 needs to be quite smart. In fact, replanned controllers for tier 3 try to infer, through a series of recovery actions, what their current location is before attempting to achieve their goals as they would have, having known their current location. For example, the new level 3 controller can start by performing *putdown* because the success or failure of this action reveals to the controller whether it is in location e or not. Such recovery strategies are built automatically by the planner.

5.2 Generalised Reactivity

In our second instantiation of the framework, we take the three tiers of models as given in instantiation 1 and we change the synthesis technique in tier 3 from backward propagation to GR(1) [37]—an expressive subset of linear temporal logic that includes liveness—implemented in MTSA [18]. This technique allows us to specify liveness properties without a bound which may be useful if the bound is unknown and also may be computationally convenient if the bound is known to be high. On the downside, if there is a low bound, the additional computational complexity of solving GR(1) goals (polynomial) against backward propagation (linear) results in a significant overhead.

Replacing the bounded liveness goal and backward propagation technique of tier 3 with GR(1) and the MTSA synthesis algorithm is straightforward. As before, the controller for tier 3 is computed based on the composition of GOOD_DOMAIN and the controller for tier 2. This is complete because controllers for tier 1 and 2 are maximal. We discuss maximality further in Section 5.4.

Although GR(1) limits the controllers above the tier in which it is used, graceful degradation and progressive enhancement work in the same manner as with backward propagation. If the level 3 controller throws an exception, then execution continues with the level 2 controller. Likewise when attempting to resume level 3, GR(1) is capable of generating recovery actions from a choice of starting states, as happens with backward propagation.

5.3 Reactive Planning

In our final instantiation, we retain tiers 2 and 3 from the second instantiation, and change the synthesis technique in tier 1 from backward propagation to the more robust reactive planning [44], again implemented in LTSA.

Although reactive planning uses the same core algorithm as backward propagation (for safety properties), it differs in that it envisages a different enactment mechanism which, in addition to performing controlled actions and monitoring uncontrolled actions, also directly senses propositions about the environment to determine the current state. More specifically, the enactment mechanism we adopt (see Figure 3) uses sensing to verify that the valuation of sensed variables (in the runtime environment, not any model) matches the valuation in the current inferred (model) states. Note that this scheme differs from the traditional reactive plan enactment mechanism [42], which does not retain the model at runtime and consequently is stateless.

In this instantiation we assume that the system, rather than sensing the success or failure of *pickup* and *putdown* actions, can sense if it is holding a product. Thus, the resulting controller will be robust to unexpected events such as the accidental dropping of a product, or manual intervention placing a product into the robot's hand.

In order to transform our pure LTS model for tier 1 into an LTKS, required to perform reactive planning, we must label the states with valuations of the sensed variables. Space limitations prevent us from giving the full details here, but the essential aspect is that fluents (in this case *holding*) are defined to label the LTKS according to what can be sensed.

The enactment mechanism for reactive plans, in contrast to backward propagation and GR(1), can raise exceptions in two situations: when an unexpected monitored action occurs, and when the valuation of sensed variables does not match the valuation of the inferred current state. When attempting to resume operation in this tier, the sensed variables can be used to narrow down the set of possible starting states.

As no actions are performed "blindly", reactive plan enactment is robust with respect to the unexpected impact of actions on the environment. The limitation of reactive planning is that the goals which can be achieved without "remembering" what the system has done are limited. In contrast, the GR(1) technique used in tier 3 infers the state of the world from the history of controlled and uncontrolled actions that have occurred, without checking at runtime the correspondence of the inferred state with the runtime state. Such approaches are less robust as they risk taking decisions based on an incorrect assumption about what the current state of the world is.

These two techniques had not, up to now, been combined even in an *ad hoc* fashion. Our framework provides a general mechanism for combining these and other techniques and in this particular case results in a hybrid approach that addresses the limitations of both while retaining their strengths.

A critical aspect for the use of reactive planning in our framework is that the underlying LTKS (rather than a representation in the form of a condition-action table) is available. This is necessary to restrict the environment of the tier above (level 2), so that the controllers meet the simulation requirement. Further effort may be required to use reactive planning tools (e.g. [10]) that do not make this LTKS available in our framework.

5.4 Discussion

The demonstration shows how our framework can combine synergistically and support the use of different controller synthesis and planning techniques, particularly to handle graceful degradation and progressive enhancement even when these synthesis techniques were not developed specifically to address the challenges present in engineering adaptive systems. It also shows that the construction of the controller stack can be engineered in a modular fashion thereby avoiding an increase of complexity in the synthesis algorithms used in each tier.

The main limitation enforced by the modular bottom-up planning procedure is that, in order for the approach to be *complete*, the controller for the tier below must be maximal (i.e. encodes all possible strategies for achieving its goal). This is always possible when the lower tier has a safety property, but it is not always possible for liveness goals. This means that attempting to build a controller on top of one that achieves a liveness goal may fail due to the fact that the strategy used for liveness is inconsistent with the higher tier goal even though a different strategy for liveness would have allowed a controller for the higher-level goal. Note however that this incompleteness does not mean that tiers cannot be stacked on top of liveness goals. If the Planner can build a controller for the next tier up graceful degradation and progressive enhancement are guaranteed.

Although it is likely that in many practical applications safety properties will be expected to be achieved in the lower (more robust) tiers, we believe that more sophisticated complete synthesis techniques will have to be specifically tailored to allow stacking tiers on top of liveness properties. The lack of limitations on lower-tier safety properties allows flexible treatment of different kinds of safety properties such as those related to physical integrity of human life, of the system itself or other environment agents, or required and nice-to-have properties.

6. RELATED WORK

Our work in this paper touches on a range of related topics. We cannot give an exhaustive list here, but in the following we summarise some of the related work.

Our work concerns techniques for synthesising behaviour, a broad field that draws from program synthesis (e.g. [34]), model driven development (e.g [25, 8, 49, 32]), planning (e.g. [42, 10, 44]) and supervisory control (e.g. [37]).

Our framework, to be instantiated, relies on the extensive work that has been developed in the areas of discrete event planning [42, 10, 44] and supervisory control [37, 18] which has focused on augmenting the expressiveness of goals that can be automatically achieved and the complexity of building strategies for achieving them (e.g. [28]). We are bound by these results, which have been shown to allow application of these techniques in practice (e.g. [31]). Other planning-based approaches, particularly within the area of adaptive systems, include PLASMA [45], which employs reactive planning for architectural assembly, and others that use planning for architecture [6, 16, 35].

More broadly, the central feature of our framework is its ability to deal with enhancement and degradation of functionality when assumptions are broken. This relies on some means to monitor assumptions as in the work of Welsh *et al.* [48], and on appropriate recovery action. In much work (e.g. [40, 44]) this consists of switching to redundant alternative implementations of components, or applying alternative sequences of actions, as in the recent work by Carzaniga *et al.* [12]. In the context of behaviour synthesis it can involve rolling back recent actions, or generating a new strategy that includes extra recovery actions [6]. The common theme is however that there is a single requirement to be met, whereas we admit the possibility of a range of stronger and weaker requirements (cf. [49, 7, 27]) to deal with uncertainty, an approach envisaged in the SEAMS roadmap [14].

Degradation and enhancement based on service quality (e.g. [13]) has been studied extensively. Notably, for adaptive systems, Ghezzi *et al.* [23] mixes design-time analysis of a behavioural model with runtime decision-making that considers the probability of achieving the system requirements. When a requirement is under threat, a form of graceful degradation is allowed, by omission of optional functionality (while inclusion could be considered an enhancement). This approach is in the same spirit as ours in that a mechanism for detecting divergence between the runtime environment and the environment model is in place and replanning mechanisms are used to degrade and enhance. Nonetheless, the environment model is structurally fixed and only rates and probabilities of transitions are monitored. Our framework could be applied in order to handle violation of functional (as opposed to quantitative) assumptions.

The notion of replanning as a way of dealing with uncertainty has been studied in the planning community. For instance in [11], replanning occurs when a plan, based on a partial model or a weaker goal, is deemed to be improvable through the acquisition of new information. However, the notion of multiple tiers with clear guarantees on the goals achievable at each tier based on the tier's assumptions has not been developed.

Runtime monitoring of behaviour models has been studied extensively [3]. Our approach is multi-tier and is made possible by having all active controllers execute concurrently. In contrast, in [36] the case when system requirements change is addressed by automatically identifying the states in which it is safe to switch to a new controller. Zhang and Cheng [50] verify that manually specified transitions between states in the old and new controllers are safe.

Our approach was motivated by the desire to avoid setting fixed assumptions by adopting a single domain model. However, an alternative to the hierarchy of models we have proposed here is to use feedback from executing the controller to revise the model itself [19]. Two existing approaches (our previous work [43], which used inductive learning, and that of Epifani *et al.* [20], which used Bayesian estimation) took a probabilistic view of the environment and updated those probabilities according to observations (such as detected inconsistencies). The hierarchy of models we propose here could be seen as distinct discretisations of the underlying (unknown) probabilistic environment model. This suggests there is fruitful work to be done in combining the approaches, for instance, by enabling the approach of Epifani, which cannot alter the structure of the model, to switch between structurally different models.

Note that our work is orthogonal to the extensive work on continuous control and planning which also has a place in adaptive systems (particularly robotic ones). Continuous control typically establishes a feedback loop where control decisions are made solely based on sensing of the environment's state and are used to achieve short term goals. Hybrid and discrete controllers are typically built on top of these techniques. In these higher layers the controller can be seen as a feedforward control where it can anticipate the result of controlled actions and, hence, is able to plan in advance the necessary operations in order to achieve longer term goals (e.g. [26, 46]). This is also the principle behind the three-layer reference model for adaptive system [30], where our work would be situated in the goal management layer. In the MAPE-K reference model [15], our work would sit in the analyse or plan components.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a framework for the planning and enactment layers of reactive systems. The framework supports simultaneously multiple control problems each with its own set of environment assumptions and goals to be achieved. These tiers allow the engineering of adaptive systems to deal with multiple levels of risk, gracefully degrading services when environment assumptions are found not to hold and progressively enhancing services when possible, in the hope that stronger assumptions will hold again.

We have demonstrated how the framework can be instantiated for different combinations of properties (e.g. safety and liveness) and synthesis algorithms, in particular showing that additional constraints introduced by the framework can be systematically addressed and do not introduce further computational complexity. Finally, we have demonstrated how the framework can be used to combine synergistically rather different approaches such as reactive planning (a robust memoryless control approach for safety properties based on state-based sensing) and GR(1) (an expressive non-robust memory-based approach for liveness properties).

We believe that the framework lays the foundations and opens a research agenda for exploring multi-tier control frameworks for adaptive systems. There are a number of broad avenues of future work that the approach presented in this paper puts forward. Firstly, we believe the embedding of the tool chain for the different instantiations presented into a real sensing and actuating adaptive system is the next logical step in the evaluation of the approach. In addition, further generalisation is of particular interest. For instance, exploring weaker requirements than a total ordering of environment models (e.g. tree structures) should be possible and useful in various settings. In addition, a framework to support both qualitative and quantitative assumptions is essential to integrate much of the work that has been developed for planning and enactment of adaptive systems, and also to support more sophisticated adaptive degradation and enhancement.

8. ACKNOWLEDGEMENTS

This work was partially supported by the following grants: ANPCYT PICT 1774/11 and 0724/12, UBACYT W0813, ERC StG PBM FIMBSE and EU FP7 MEALS - 295261.

9. REFERENCES

[1] Appendix. http://www.doc.ic.ac.uk/~su2/tmp/shl.pdf.

[2] The Labelled Transition System Analyser (LTSA). http://www.doc.ic.ac.uk/ltsa/.

[3] *Workshop on Models at Runtime Series*. ACM/IEEE, 2006–2013.

[4] *ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*. ACM/IEEE, 2006-2013.

[5] P. E. Agre and D. Chapman. What are plans for? *Robotics and Autonomous Systems*, 6(1-2):17 – 34, 1990. Designing Autonomous Agents.

[6] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281, 2007.

[7] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy goals for requirements-driven adaptation. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 125–134, 2010.

[8] S. Bernardi, J. Merseguer, and D. C. Petriu. Dependability modeling and assessment in uml-based software development. *The Scientific World Journal*, 2012, 2012.

[9] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[10] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *Proceedings of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[11] R. I. Brafman and G. Shani. Replanning in domains with partial information and sensing actions. *J. Artif. Intell. Res. (JAIR)*, 45:565–600, 2012.

[12] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791. IEEE Press, 2013.

[13] L. Checiu, B. Solomon, D. Ionescu, M. Litoiu, and G. Iszlai. Observability and controllability of autonomic computing systems for composed web services. In *Applied Computational Intelligence and Informatics (SACI), 2011 6th IEEE International Symposium on*, pages 269–274, 2011.

[14] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 2009.

[15] A. Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006.

[16] C. E. da Silva and R. de Lemos. Using dynamic workflows for coordinating self-adaptation of software systems. *Software Engineering for Adaptive and Self-Managing Systems, International Workshop on*, 0:86–95, 2009.

[17] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120. ACM, 2001.

[18] N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 77–86, New York, NY, USA, 2010. ACM.

[19] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 7–16. ACM, 2010.

[20] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 111–121. IEEE, 2009.

[21] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems*, volume 7475 of *Lecture Notes in Computer Science*, pages 214–238. Springer, 2010.

[22] E. Gat. *Three-layer Architectures, Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997.

[23] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 33–42. IEEE Press, 2013.

[24] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[25] H. Giese and W. Schaefer. Model-driven development of safe self-optimizing mechatronic systems with mechatronicUML. Technical Report tr-ri-12-322, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany, Apr. 2012.

[26] J. H. Gillula, H. Huang, M. P. Vitus, and C. J. Tomlin. Design and analysis of hybrid systems, with applications to robotic aerial vehicles. In C. Pradalier, R. Siegwart, and G. Hirzinger, editors, *ISRR*, volume 70 of *Springer Tracts in Advanced Robotics*, pages 139–149. Springer, 2009.

[27] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. Cheng, and D. Hughes. Goal-based modeling of dynamically adaptive system requirements. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th*

Annual IEEE International Conference and Workshop on the, pages 36–45. IEEE, 2008.

[28] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata logics, and infinite games: a guide to current research*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[29] J. A. Hager. Software cost reduction methods in practice. *IEEE Trans. Softw. Eng.*, 15(12):1638–1644, Dec. 1989.

[30] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[31] H. Kress-Gazit, D. Conner, H. Choset, A. Rizzi, and G. Pappas. Courteous cars. *Robotics Automation Magazine, IEEE*, 15(1):30–38, 2008.

[32] E. Letier and W. Heaven. Requirements modelling by synthesis of deontic input-output automata. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE*, pages 592–601. IEEE / ACM, 2013.

[33] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. Wiley New York, 2006.

[34] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, Jan. 1980.

[35] A. D. Marco, P. Inverardi, and R. Spalazzese. Synthesizing self-adaptive connectors meeting functional and performance concerns. In M. Litoiu and J. Mylopoulos, editors, *SEAMS*, pages 133–142. IEEE / ACM, 2013.

[36] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 63–72. IEEE Press, 2013.

[37] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364–380, 2006.

[38] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[39] A. J. Ramirez, B. H. Cheng, N. Bencomo, and P. Sawyer. Relaxing claims: Coping with uncertainty while evaluating assumptions at run time. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 53–69. Springer Berlin Heidelberg, 2012.

[40] B. Randell. System structure for software fault tolerance. *Software Engineering, IEEE Transactions on*, (2):220–232, 1975.

[41] S. Russell and P. Norvig. Artificial intelligence: a modern approach. *New Jersey*, 1995.

[42] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *IJCAI*, volume 87, pages 1039–1046. Citeseer, 1987.

[43] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue. Learning revised models for planning in adaptive systems. In *Proceedings of ICSE*, 2013.

[44] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From Goals to Components: A Combined Approach to Self-Management. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'08, 2008.

[45] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. Plasma: a plan-based layered architecture for software model-driven adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 467–476. ACM, 2010.

[46] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[47] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, Oct. 2000.

[48] K. Welsh, P. Sawyer, and N. Bencomo. Towards requirements aware systems: Run-time resolution of design-time assumptions. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 560–563. IEEE, 2011.

[49] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009.

[50] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM Press.

APPENDIX

A. FRAMEWORK PROPERTIES

The aim of this section is to articulate more formally what the framework guarantees in terms of degradation and enhancement and also to sketch the key elements of a proof of such guarantees. To do so, we describe in a homogeneous semantic setting the behaviour of the framework discussed above together with the behaviour of the runtime environment in which the framework is executed. We use, as a semantic domain, labelled transition systems, which are a simplified form of LTKSs in which the set of predicates is empty.

First, we give an invariant of the execution of the framework which is the basis for the remainder of the results in this section.

Theorem A.1. *Let S be a valid framework state for goals $G_0, \dots, G_n$. If S' is the result of updating S according to an action or sensed state, then S' is a valid framework state for goals $G_0, \dots, G_n$.*

It is straightforward to show that the update operation (Definition 4.3) preserves state validity. This theorem relies on the fact that there is a simulation relation between environment models and also between controllers, and the fact that the functional level is set to exclude any higher level controller or environment model that cannot follow the sensed state, sensed action or actuated action.

We now define an LTS that models the behaviour of the Enactor. The states of the LTS represent valid framework states. The transitions of this LTS model framework state updates (i.e. $\triangleright$).

Definition A.1. (Enactor Behaviour) *Let $s = (M_0, \dots, M_n, C_0, \dots, C_n, max)$ be a valid framework state. and A_C, A_M, P represent the sets of monitored actions, controlled actions and sensed propositions of the framework. We define $Enactor(s)$ to be the smallest LTS (S, A, Δ, S_0) such that $S_0 = \{s\}$, $A = A_C \cup A_M \cup 2^P$, and $(s_1, a, s_2) \in \Delta$ if and only if $s_2 = (s_1 \triangleright a)$.*

It is relevant to ask if the definition above is a sound abstraction of the code in Figure 3. The key point to note here is that a state of $Enactor(s)$ can have outgoing transitions representing monitored and controlled actions; however, the code actually senses and actuates in sequence. This is not a problem if the synchronous hypothesis [9] is used which assumes that the system is fast enough to actuate so as to never block the environment. This is a common assumption in the design of reactive systems [29] that makes even more sense in architectural adaptation strategies. The Enactor code (see Figure 3) executes very cheap update operations when not sensing monitored actions, hence the synchronous hypothesis is reasonable in this setting. Note however that the execution time of the Planner is not negligible as planning operations are time consuming. We assume reasonably engineered implementation of the Planner which minimises locking time of S so to not block the Enactors execution. One such implementation would be that S is only locked once the a viable plan has been built and at the end of the execution of the Enactor's while block. To avoid interference and breaking the invariant of framework state validity, only if S has not changed while the Planner was computing the controller (i.e. no sensed or actuated actions) the new controller is inserted.

We are interested in properties about the composition of $Enactor(s)$ with its runtime execution environment E, which we will assume is an LTS that has transition labels modelling controlled and monitored actions (i.e. A_C and A_M) and state sensing (2^P).

In addition these properties must refer to conformance of the runtime execution behaviour to the user-provided environment models. Hence, we interpret LTKS environment models as LTSs by means of $sem(M_i)$.

Definition A.2. (LTS Semantics of LTKS) *Given an LTKS $M = (S, A, P, \Delta, v, S_0)$ with controlled actions A_C and monitored actions A_M, the LTS semantic interpretation of M, denoted $sem(M)$ is the smallest LTS $M = (S', A', \Delta', v', S'_0)$ where $S_0 = \{E\}$, $A = A_C \cup A_M \cup 2^P$, and $(s_1, a, s_2) \in \Delta$ if and only if $(s_1 \xrightarrow{a} s_2)$ or $(s_1 = s_2 \wedge a = v(s_1))$.*

Now we give a result that states that the Enactor when executing in a runtime environment that exhibits behaviour consistent with the environment model of tier i guarantees all goals up to that level. In other words, if $Enactor(s)$ composed in parallel with its runtime execution E exhibits behaviour that is consistent with $sem(M_i)$, then the goals G_0 through G_i are satisfied.

Theorem A.2. *Let $S = (M_0, \dots, M_n, C_0, \dots, C_n, q)$ be a valid framework state for goals $G_0, \dots, G_n$ and let E be an LTS representing the behaviour of the runtime environment. If $\pi \in \mathrm{Tr}(E||Enactor(S))$ and $\pi \in \mathrm{Tr}(sem(M_i))$ for some i such that $0 \le i \le q$ then $\pi \models G_j$ for all $0 \le j \le i$.*

The proof of this theorem follows from Theorem **??** and the fact that all controllers up to tier i are solutions to the control problem of their tier (see Definition 4.1). Note that a related result can be stated if the LTS E representing the runtime environment *conforms* to a human-provided environment model M_i (i.e. $sem(M_i) \ge E$). In this case, all traces $E||Enactor(S)$ are guaranteed to be traces of M_i (i.e. $Tr(E||Enactor(S)) \subseteq Tr(sem(M_i))$) and thus satisfy goals up to G_i.

However, a subtle but important issue worth discussing further is the fact that Theorem **??** is for infinite traces. This is required to support liveness properties. However, the requirement means that if the Enactor is deployed in a runtime environment that can prevent infinite traces no guarantees can be given. Hence, relevant questions are *i)* what characteristic should an environment have to guarantee infinite traces when composed with the Enactor? and *ii)* do these characteristics restrict applicability of the result?

If E is a legal environment (see Definition 3.5) of the Enactor then all traces of $E||Enactor(S)$ can be guaranteed to be extensible to infinite ones. Being a legal environment requires, on the one hand, that E should not block Enactor-controlled actions, an expectation that is not limiting (it is akin to assuming that the controller will always be able to perform a controlled action, which is not the same as assuming that the expected consequence of that controlled action will occur). On the other hand, the Enactor should not block monitored actions in E but this is trivially satisfied by definition of the update operation $\triangleright$ which is defined for all actions and valuations (see Definition 4.3).

Indeed, the notion of legal environment guarantees that "outputs" of one component are not blocked by the other.

However, it provides no guarantee that an "input" expected by a component will be provided. In other words, a deadlock is possible because E is waiting for $Enactor(S)$ to perform a controlled action while $Enactor(S)$ is waiting for a monitored action to occur. The following scenario exemplifies this: the Enactor is waiting for monitored actions that M_q assumes possible but in fact are not in possible in E. In addition, the Enactor refuses to perform any controlled action as C_q deems it convenient (to achieve G_q) to wait for a monitored action from E. Further, E has only controlled actions enabled. In this case, a deadlock ensues as the E.

Fortunately, there is a methodological workaround for this type of scenario and it corresponds to a typical strategy in adaptive systems: timeouts. If the system is equipped with a timeout event that signals lack of activity on the environment side this would be interpreted as an unexpected monitored action that evidences that E does not conform to M_q. Thus, Enactor would proceed, gracefully degrading to a more permissive level where a controlled action can be taken.

To present a result on *progressive enhancement* we first must show that if the framework state is valid and providing functional level i, a successful replanning procedure (Definition 4.6) at tier $i+1$ yields a valid framework state. The result follows straightforwardly from the definitions (Definition 4.1 and 4.6).

THEOREM A.3. *Let S be a valid framework state for goals $G_0, \ldots, G_n$ providing functional level $i < n$. If replanning tier $i+1$ is successful then the resulting framework state is valid.*

Once replanned, Theorem **??** is applicable from the resulting framework state. Note, however, that replanning provides guarantees from the point of replanning onwards and not from the start of the execution of the system: Suppose a finite trace α is exhibited by $E||Enactor(S)$ and α is not a valid finite trace of M_i (thus the framework is providing a degraded functional level, below i). Assume that after α the current framework state is S' and that of the environement is E'. If we now observe an infinite trace β by $E'||Enactor(S')$ where β is a valid infinite trace of M_i', it is not true that $\alpha\beta \models G_j$ for all $0 \leq j \leq i$, only that $\beta \models G_j$ for all $0 \leq j \leq i$.