Robust Degradation and Enhancement of Robot Mission Behaviour in Unpredictable Environments*

Nicolas D'Ippolito[1], Victor Braberman[1], Daniel Sykes[2], Sebastián Uchitel[12]
[1] Departamento de Computación, FCEN, Universidad de Buenos Aires, Argentina
[2] Department of Computing, Imperial College London, UK

{ndippolito,vbraber}@dc.uba.ar, {daniel.sykes,suchitel}@imperial.ac.uk

ABSTRACT

Temporal logic based approaches that automatically generate controllers have been shown to be useful for mission level planning of motion, surveillance and navigation, among others. These approaches critically rely on the validity of the environment models used for synthesis. Yet simplifying assumptions are inevitable to reduce complexity and provide mission-level guarantees; no plan can guarantee results in a model of a world in which everything can go wrong. In this paper, we show how our approach, which reduces reliance on a single model by introducing a stack of models, can endow systems with incremental guarantees based on increasingly strengthened assumptions, supporting graceful degradation when the environment does not behave as expected, and progressive enhancement when it does.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Design

Keywords

Self-adaptive Systems, Controller Synthesis

1. INTRODUCTION

Controller synthesis and planning approaches based on temporal logic have proven useful for generating discrete event-based robot behaviours from high-level specifications (e.g. [4, 30, 29]). Such approaches rely on finite-state models that purport to represent the operating environment and how the robot can interact with it. However, any such model is by definition an abstraction of the real environment and its dynamics, and any such model entails a risk that it is not a true representation of the environment as encountered at runtime. In some scenarios, this risk, when materialised, may lead to catastrophic failure of the mission.

One means to cope with this uncertainty [12] is to use machine learning techniques that revise (or indeed generate from scratch) the models on which synthesis relies so that, over a period of time, the models converge upon a "realistic" description of the environment [27, 11, 14]. One drawback of using such techniques is the computational cost of learning, and the delay before the mission can begin in earnest, which may be prohibitive in some domains (e.g. safety-critical systems). Another drawback is that the learned model may be of such complexity that synthesis becomes computationally infeasible, and in the worst case nothing can be guaranteed in a world where anything can go wrong. There is therefore a benefit in having an element of manual abstraction involved in synthesising robotic behaviours. To that end, we have proposed an approach [7] in which models at different levels of abstraction are used to synthesise a controller capable of gracefully degrading its guarantees when the runtime environment diverges from one of the more abstract models, and progressively enhancing its guarantees when the environment behaves as envisaged in the more idealised models.

Our approach uses a stack of models where higher models are more idealised and can be simulated by the lower models. A mission requirement is associated with each tier of the stack. Higher tiers allow to produce controllers guaranteeing stronger requirements, while lower tiers only allow for controllers with weaker requirements because of their more realistic description of the environment dynamics. Each tier of the stack can be regarded as an independent controller synthesis problem, but our approach combines the resulting controllers in such a way that a failure in a higher controller can be handled by a graceful degradation to the controller of a lower tier, resulting in a lower guaranteed 'service level'. Likewise, if the environment conforms to a higher tier, we may attempt to synthesise a controller for a higher tier and so enhance the guaranteed service level.

In this paper, we show how synthesised controller stacks can be used to provide robust behaviour for robot missions from high-level temporal logic specifications. We apply it to an existing case study involving a robot engaged in a surveillance mission [28] and show how, in addition to automatic synthesis for cyclic missions (i.e. missions in which the goals are achieved infinitely many times, our approach enables the robot to handle invalid environment models. In

*This work was partially supported by grants ERC PBM-FIMBSE, ANPCYT PICT 2012-0724, UBACYT W0813, ANPCYT PICT 2011-1774, UBACYT F075, CONICET PIP 11220110100596CO, MEALS 295261.

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804342

other words, in this paper we report on an application of our previous technique to a known case study to evaluate its applicability and asses some of its properties.

In Section 3 we give an overview of our approach and the guarantees it makes. Section 4 describes our particular implementation of the general approach and how it achieves the general requirements. Sections 5 and 6 discuss how the approach has been applied to an existing surveillance case study. Finally, Section 7 comments on some of the related work, before Section 8 concludes.

2. BACKGROUND

We start with labelled transition systems a canonical representation of reactive components and systems.

DEFINITION 2.1. (Labelled Transition Systems) *A* Labelled Transition System *(LTS) is* $E = (S, A, \Delta, s_0)$*, where* S *is a finite set of states,* A *is its* communicating alphabet*,* $\Delta \subseteq (S \times A \times S)$ *is a transition relation, and* $s_0 \in S$ *is the initial state. We denote* $\Delta(s) = \{\ell \mid (s, \ell, s') \in \Delta\}$ *and* $\Delta(s, \ell) = \{s' \mid (s, \ell, s') \in \Delta_E\}$*. A trace of* E *is* $t{=}\ell_0, \ell_1, \ldots$*, where for every* $i \geq 0$ *we have* $(s_i, \ell_i, s_{i+1}) \in \Delta$*, We denote the set of traces of* E *by* $\text{TR}(E)$*. We say that an LTS is deterministic if* (s, ℓ, s') *and* (s, ℓ, s'') *are in* Δ *implies* $s' = s''$*.*

Reactive systems are built as compositions of multiple reactive components. Such composition is formalised as follows:

DEFINITION 2.2. (Parallel Composition) *Let* $M = (S_M, A_M, \Delta_M, s_{M_0})$ *and* $E = (S_E, A_E, \Delta_E, s_{E_0})$ *be LTSs. The* Parallel Composition ($\|$) *is a symmetric operator such that* $E\|M$ *is the LTS* $E\|M = (S_E \times S_M, A_E \cup A_M, \Delta, (s_{E_0}, s_{M_0}))$*, where* Δ *is the smallest relation that satisfies the rules below, where* $\ell \in A_E \cup A_M$*:*

$$\frac{(s,\ell,s') \in \Delta_E}{((s,t),\ell,(s',t)) \in \Delta} \ \ell \in A_E \setminus A_M \qquad \frac{(t,\ell,t') \in \Delta_M}{((s,t),\ell,(s,t')) \in \Delta} \ \ell \in A_M \setminus A_E$$

$$\frac{(s,\ell,s') \in \Delta_E,\ (t,\ell,t') \in \Delta_M}{((s,t),\ell,(s',t')) \in \Delta} \ \ell \in A_E \cap A_M$$

We restrict attention to states in $S_E \times S_M$ *that are reachable from the initial state* (s_{E_0}, s_{M_0}) *using transitions in* Δ*.*

There are various restrictions that can be imposed on the LTS to be composed using parallel composition. These restrictions vary in order to adequately capture different interaction models. We are interested in reasoning about what a controller can achieve in a possibly adversarial environment. Hence, the distinction between actions that are controlled or monitored by the controller is relevant. Thus we adopt the notion notion of legal LTS from *Interface Automata* [5] where a component may not block their environment from performing actions that they monitor.

DEFINITION 2.3. (Legal LTS) *Given LTSs* $M = (S_M, A, \Delta_M, s_{M_0})$ *and* $E = (S_E, A, \Delta_E, s_{E_0})$*, where* A *is partitioned into actions controlled and monitored by* M *(*$A = A_C \dot\cup A_M$*), we say that* M *is a legal LTS for* E *if for all* $(s_E, s_M) \in E\|M$ *it holds that* $\Delta_E(s_E) \cap A_C \supseteq \Delta_M(s_M) \cap A_C$ *and also that* $\Delta_E(s_E) \cap A_M \subseteq \Delta_M(s_M) \cap A_M$*.*

Simulation relation between two LTSs is formally defined as follows.

DEFINITION 2.4. (Simulation) *Let* $\wp$ *be the universe of all LTSs 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* $F \xrightarrow{\ell} F' \wedge \forall s_E \in \text{init}(E') \cdot \exists s_F \in \text{init}(F') \cdot (E', F') \in R$*.*

We fix Fluent Linear Temporal Logic (FLTL) [15] as the language for describing properties. A *fluent* Fl is defined as $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, where $I_{Fl} \subseteq A$ is the set of initiating actions, $T_{Fl} \subseteq A$ is the set of terminating actions and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent may be initially *true* or *false* as indicated by $Init_{Fl}$. Every action $\ell \in A$ induces a fluent, namely $fl_\ell = \langle \ell, A \setminus \{\ell\}, false \rangle$.

Let $\mathcal{F}$ be the set of all fluents over A. An FLTL formula is defined inductively using the standard Boolean connectives and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (strong until) as follows:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \psi$$

where $Fl \in \mathcal{F}$. Additionally, as it is usual, we define $\varphi \wedge \psi$ as $\neg\varphi \vee \neg\psi$, $\Diamond\varphi$ (eventually) as $\top\mathbf{U}\varphi$, $\Box\varphi$ (always) as $\neg\Diamond\neg\varphi$, and $\varphi\mathbf{W}\psi$ (weak until) as $(\varphi\mathbf{U}\psi) \vee \Box\varphi$.

Let Π be the set of infinite traces over A. The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent Fl at position i, denoted $\pi, i \models Fl$, if and only if one of the following conditions holds:

- $Init_{Fl} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \rightarrow \ell_j \notin T_{Fl})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \ \rightarrow \ell_k \notin T_{Fl})$

In other words, a fluent holds at position i if and only if it holds initially or some initiating action has occurred, but no terminating action has yet occurred.

For an infinite trace π, the satisfaction of a (composite) formula φ at position i, denoted $\pi, i \models \varphi$, is defined as follows:

$$\begin{array}{lcl}
\pi, i \models Fl & \triangleq & \pi, i \models Fl \\
\pi, i \models \neg\varphi & \triangleq & \neg(\pi, i \models \varphi) \\
\pi, i \models \varphi \vee \psi & \triangleq & (\pi, i \models \varphi) \vee (\pi, i \models \psi) \\
\pi, i \models \mathbf{X}\varphi & \triangleq & \pi, i+1 \models \varphi \\
\pi, i \models \varphi \mathbf{U} \psi & \triangleq & \exists j \geq i \cdot \pi, j \models \psi \wedge \\
 & & \forall\, i \leq k < j \cdot \pi, k \models \varphi
\end{array}$$

We say that φ holds in π, denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula $\varphi \in$ FLTL holds in an LTS E (denoted $E \models \varphi$) if it holds on every infinite trace produced by E.

An LTS control problems aims to find, given an LTS E modelling the environment to be controlled and a goal φ to be achieved, a *controller* M in the form of an LTS such that $E\|M$ does not restrict uncontrollable actions of E, does not have deadlocks and satisfies φ

DEFINITION 2.5. (LTS Control [6]) *Given a domain model in the form of an LTS* $E = (S, A, \Delta, s_0)$*, a set of controllable actions* $A_c \subseteq A$*, and an FLTL formula* φ*, a solution for the LTS control problem* $\mathcal{E} = \langle E, \varphi, A_c \rangle$ *is an LTS* $M = (S_M, A_M, \Delta_M, s_{0_M})$ *such that* M *is a legal LTS for* E*,* $E\|M$ *is deadlock free, and every trace* π *in* $E\|M$ *is such that* $\pi \models \varphi$*.*

3. APPROACH

The central concept in our approach is that of the *control stack*, which has in each *tier* a controller synthesis problem for a particular mission requirement and environment model. Overall the control stack specifies the robot's mission.

The key requirements the approach 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 lower levels controllers, 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 requirements for its tier must be provided, allowing progressive system enhancement after inconsistencies have been detected. Our implementation of the approach provides the runtime infrastructure (iii) and planning procedure (iv), guarantees controller simulation (ii), and checks that the models given in a control stack specification satisfy (i).

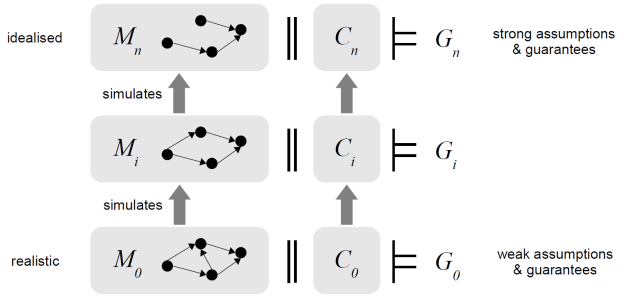

Figure 1: Multi-tier control problem

The environment models are expected to be ranked in terms of the degree of idealisation of the environment they represent. The environment model M_0 is the least idealised and require that environment models further up the hierarchy allow strictly less behaviour. This can be formally captured via a simulation relation [23], $M_i \geq M_j$ for $i < j$. We require environment models to have the same communicating alphabets partitioned identically into *controlled* and *monitored* actions. Controlled actions are those that the robot may choose to perform, while monitored actions are events that the robot observes in the environment. In summary, the less idealised the environment model is, the more behaviour (in terms of unexpected actions and nondeterminism) may arise.

Each tier i has an associated requirement (G_i) to be achieved by the system assuming that the runtime environment conforms to the environment model for that tier (M_i).

Each tier introduces a control problem $\mathcal{E}_i = \langle M_i, G_i \rangle$. A solution to a control problem (a controller) is a deterministic LTS that, when composed with its environment, guarantees requirement G_i (i.e. $M_i \| C_i \models G_i$). The control stack introduces 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 never do something that a lower-tier controller would not do, thus ensuring that if a controller must be stopped, because the assumptions for its tier are discovered not to hold, decisions made by it up to that point have been consistent with lower-tier controllers. This allows for graceful degradation, falling back to lower-tier controllers when needed. Section 4 describes how this constraint is satisfied.

Control stack synthesis is executed bottom-up through the tiers. 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. We do not require that control problems for all tiers have solution. It is possible that the system starts in a degraded mode, with controllers solving problems up to level i. The system, as the current state evolves, may progressively enhance its behaviour by synthesising controllers for tiers beyond tier i.

After synthesis, the enactment procedure continuously monitors the environment and concurrently executes the stack of controllers giving priority to the controller of the uppermost enabled tier. It continuously updates the current state based on monitored actions and sensed state, disabling tiers at level i and above should an inconsistency be detected at tier i (Section 4 shows how this is achieved). At any point, to *progressively enhance 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 automatically approximated and an attempt is made to build a controller that will work despite the uncertainty about the current state of the tier. This demands that the controller synthesis procedure be capable of solving problems exhibiting non-determinism. Should a controller exist, it is put into the controller hierarchy and the tier is enabled. The approach 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 the problem domain. For a detailed explanation of the enactment procedure the reader is referred to [7].

4. IMPLEMENTATION

The implementation of our framework consists of two main components: a planner, which implements the controller synthesis algorithms, and an enactor, which handles runtime execution of the control stack.

4.1 Planner

The Modal Transition System Analyser (MTSA) [8], is a tool for developing and analysing compositional models of concurrent systems, using the Finite State Processes (FSP) process algebra. Importantly for our approach, MTSA implements controller synthesis algorithms for Generalised Reactivity(1) (GR(1)) goals, which cover an expressive subset of linear temporal logic including safety and liveness properties [8]. Our general approach is agnostic as regards the synthesis procedure, but GR(1) is expressive enough for many domains. We extended MTSA to support the specification and synthesis of complete control stacks A control stack `C` is specified in MTSA as follows:

```
controlstack ||C@{Controlled} {
  tier(ENV, REQ)
  ...
  }
```

where `Controlled` refers to a set of controlled actions, and where each tier consists of environment model `ENV` and mission requirement specification `REQ`. A control stack may consist of any number of tiers ordered such that the last tier has the most realistic environment model.

Environment models and requirements are defined using existing support in MTSA for process and property specification in FSP and FLTL (fluent linear temporal logic), and examples of them can be found in Section 5.

Synthesis of the control stack is achieved by solving the controller synthesis problem of each tier bottom-up from the lowest tier. If no solution is found for the problem in a particular tier, synthesis of the stack terminates at that tier. The procedure also includes a sanity check that the environments of tiers simulate the one immediately above.

Synthesis for a single tier i consists of the following steps:

1. Compose the tier's environment model E_i in parallel with the controller C_{i-1} generated by the tier below (if there is a tier below) to create E_i'. This ensures that the controller for tier i will be simulated by the controller of the tier below.

2. Solve the GR(1) controller synthesis problem for the tier's requirement on E_i', to produce controller C_i.

3. *Complete* controller C_i to produce C_i'. The *completion* consists of considering the monitored actions enabled in each state of the controller, and adding transitions to a designated *exception state* for any monitored actions which are not enabled. These transitions capture behaviours of the environment that have not been anticipated in the present tier's environment model. If the runtime environment does not behave as the model describes, one of these transitions will be taken to the exception state. A single extra transition, which we call an *exception marker*, is added at the exception state which indicates to the enactor that a particular tier has been disabled. It is these transitions that enable the enactor to detect inconsistencies

The final control stack state machine CS is a parallel composition of the completed controllers C_i', i.e. $CS = complete(C_1)\|...\|complete(C_n) = C_1'\|...\|C_n'$. This composition guarantees the requirements of every tier of the stack until the exception marker for tier i occurs, at which point it only guarantees the requirements of the tiers up to $i-1$.

4.2 Enactor

The enactor extends [3] to execute control stacks rather than individual controllers. It keeps track of the stack's current state, executing controlled actions (via domain-specific action implementations as in [3]) and responding to monitored environment events. When the current state is controlled, the enactor selects an enabled action at random. When the state is uncontrolled, the enactor waits to receive an environment event. In states where the only enabled action is an exception marker for some tier i, the enactor notes the degradation of the service to $i-1$ and reports this to the rest of the framework. In effect, this disables the controller for tier i. The planner may attempt at any point an enhancement by re-synthesising a controller for tier i (or above).

5. CASE STUDY

In this section we apply our approach to an autonomous robot given the mission of surveying a set of regions of a city. This case study is inspired by that given in [28]. The environment consists of five regions of interest, and when each region is visited the robot may receive a reward or incur some damage. In the original problem, the environment also contained a number of obstacles but the task of avoiding these is handled with lower-level control, and so we omit them from our discrete specification here.

Our overall mission goal is to have the robot repeatedly collect a specific reward (`reward[1]`), and collect the other rewards if possible. Unfortunately, it is not possible to guarantee achievement of this goal in the environment due to two sources of uncertainty. The first concerns the motion of the robot, which is not always reliable, and the second concerns the rewards and damage that the environment may provide in each region. In the worst case, the robot may move to the wrong region and receive unexpected rewards or damage. This would make our overall goal unachievable. However, we can decompose the mission into a series of requirements, the weakest of which can be satisfied in the most realistic environment in the lowest tier, and then introduce further tiers above for stronger requirements in more idealised environments.

The tier 1 (most realistic) environment model available to us is given in FSP syntax below.

```
SLIPPY_ROBOT = (arrive['r5]->ENV->ROBOT['r5]),
ROBOT[p:Locations] =
 (goto[q:Locations]->arrive[q]->ENV->ROBOT[q] |
 ...
 goto['r3]->arrive['r3]->ENV->ROBOT['r3] |
 goto['r3]->arrive['r4]->ENV->ROBOT['r4]).

ORIGINAL_MAP = MAP,
MAP = (
 arrive['r1]->(reward[1]->MAP | reward[2]->MAP) |
 ...
 arrive['r4]->(reward[2]->MAP | damage[2]->MAP) |
 arrive['r5] -> base -> MAP)+{ENV}.
||SLIPPY_DOMAIN = (ORIGINAL_MAP||SLIPPY_ROBOT).
```

The model states that the robot (which starts in `r5`) can perform a controlled `goto` action, which is followed by a monitored environment event `arrive`, which indicates arrival at the destination. This model displays a degree of non-determinism representing unreliable motion of the robot (it can equally represent unreliable sensing of location). In particular, there are two groups of adjacent regions. When the robot moves towards one such region, it may arrive in a different region that is adjacent to the destination. For example, moving to `r3` may lead to arrival in `r3` or `r4`.

As specified in the `MAP` process, after arrival, the environment can respond with one of several events representing rewards or damage. For instance, in region `r1`, the environment may provide `reward[1]` or `reward[2]`. Different regions provide different rewards, and region `r5` provides event `base` to represent the base location. The unpredictable motion in this environment model means that we cannot guarantee the strong properties that we are interested in. However, in the worst case we want the robot to ensure its physical safety and so our tier 1 requirement `AVOID_DAMAGE` is to avoid receiving `damage[2]`. This property is specified as follows:

```
fluent DAMAGE[i:Damages] = <damage[i], base>
ltl_property NO_DAMAGE2 = []!DAMAGE[2]
controllerSpec AVOID_DAMAGE = {
  safety = {NO_DAMAGE2}
  controllable = {CONT} }
```

The controller satisfying this requirement will never allow the robot to attempt a `goto['r3]` as it may arrive to `r4` which can result in damage.

If we assume reliable motion it is possible to guarantee stronger requirements. We specify such an assumption by removing the non-determinism from the `ROBOT` process:

```
ROBOT = (arrive['r5] -> ENV -> ROBOT['r5]),
ROBOT[p:Locations] =
(goto[q:Locations]->arrive[q]
             -> ENV -> ROBOT[q]).
||ORIGINAL_DOMAIN =
(ORIGINAL_MAP||ROBOT).
```

We are now able to introduce a more interesting mission requirement. We are particularly interested in having the robot collect `reward[1]`, and hence our tier 2 requirement `REWARD_LIVE_GOAL` is to collect `reward[1]` infinitely often (i.e. $\square\diamond\ reward_1$), and to visit the base infinitely often ($\square\diamond\ base$). It is specified as follows:

```
fluent REWARD[i:Rewards] = <reward[i], base>
fluent AT_BASE = <base, goto[Locations]>
assert REWARDS = (REWARD[1])
assert VISITBASE = AT_BASE
assert REWARDFAULTS = REWARD[2]
controllerSpec REWARD_LIVE_GOAL = {
  failure = {REWARDFAULTS}
  liveness = {REWARDS,VISITBASE}
  controllable = {CONT} }
```

The desired reward is only available in region `r1`, and the environment model states that, instead of `reward[1]`, the environment may, with some probability, give `reward[2]`. Provided that these hidden probabilities are non-zero, they can be abstracted with the assumption that arriving in `r1` infinitely often will yield `reward[1]` infinitely often. Encoding this kind of assumption in GR(1) has been demonstrated in [8] where probabilistic failures (i.e. `reward[2]`) are treated non-quantitatively.

It would now be possible to create a control stack consisting of two tiers using the above models and requirements. However, the synthesised controller only guarantees that *eventually* the desired reward will be received. In a practical setting with a robot's limited power supply, this is too weak a guarantee. Instead, we wish to have a bound on how long it will take to receive the reward. In order to do this we must strengthen our assumptions about the environment and create an idealised model of it, resulting in a third tier in our stack. In this case, we estimate that the environment has a low probability of repeatedly failing to give `reward[1]` (e.g. if the probability of `reward[1]` is 0.5 then the chance of failing in four attempts falls rapidly to 0.0625). We therefore introduce the assumption that the environment will comply within a small bound on the number of attempts, accepting that there is a small risk that this bound may be broken at runtime. In our idealised tier 3 environment model `BOUNDED_FAILURE_DOMAIN` we introduce an FSP process that restricts the previously given `ORIGINAL_MAP` such that the number of `reward` actions before the desired reward is bounded.

```
BOUNDED_FAILURES(Reward=1,Bound=2) = BF[0],
BF[i:0..Bound] = (reward[Reward] -> BF[0]
| when (i < Bound)
{{reward[Rewards]}\{reward[Reward]}} -> BF[i+1]).
||BOUNDED_FAILURE_DOMAIN =
 (ORIGINAL_MAP||BOUNDED_FAILURES(1,5)||ROBOT).
```

We are now in a position to specify the stronger requirement for tier 3. The `REWARD_BOUNDED_GOAL` states that the reward must be received within a count of 7 controlled actions.

```
fluent ENDED = <ended, reset>
fluent REWARD_BOUNDED[i:Rewards] =
     <reward[i], reset>
ltl_property BOUNDREWARDS =
     [](ENDED ->
      (REWARD_BOUNDED[1] && AT_BASE))
||BOUNDEDREWARDS(Bound=8) =
     (RUNNING || COUNT(Bound)
      || BOUNDREWARDS).
||BR = BOUNDEDREWARDS(7).
controllerSpec REWARD_BOUNDED_GOAL = {
  safety = {BR}
  controllable = {CONT} }
```

The tier 3 controller will ensure that `reward[1]` is received within at most 7 controlled actions (resetting the count when received), assuming that the runtime environment behaves like the tier 3 model. If, however, the runtime environment does not behave in this idealised manner, the control stack will ensure graceful degradation from tier 3 to tier 2. More specifically, if `reward[1]` is not received within the bound, an exception marker transition (inserted by the completion of the tier 3 controller) will be taken, and the subsequent behaviour of the control stack will no longer conform to the tier 3 controller. Instead it will guarantee only the requirements of tiers 1 and 2.

In the final, uppermost tier we have a mission requirement which relies on the most idealised model of the environment. We would like, if the environment turns out to be an ideal one, that the robot collect any other rewards available. Hence, our tier 4 requirement is for the robot to have received `reward[1]` and either of the other rewards infinitely often (i.e. $\square\diamond(reward_1 \wedge (reward_2 \vee reward_3))$):

```
assert ALL_REWARDS =
     (REWARD[1] && (REWARD[2] || REWARD[3]))
controllerSpec ALL_REWARD_LIVE_GOAL = {
  liveness = {ALL_REWARDS}
  controllable = {CONT} }
```

This requirement can only be achieved by assuming an environment which gives rewards deterministically:

```
PREDICTABLE_MAP = MAP,
MAP = (
 arrive['r1] -> (reward[1] -> MAP) |
 arrive['r2] -> (reward[2] -> MAP) |
 ...
 arrive['r5] -> base -> MAP)+{ENV}.
```

The specification of the mission control stack composed of the four tiers is as follows:

```
controlstack ||STACK@{CONT}= {
  tier(PREDICTABLE_DOMAIN,ALL_REWARD_LIVE_GOAL)
  tier(BOUNDED_FAILURE_DOMAIN,REWARD_BOUNDED_GOAL)
  tier(ORIGINAL_DOMAIN,REWARD_LIVE_GOAL)
  tier(SLIPPY_DOMAIN,AVOID_DAMAGE)
}
```

The resulting control stack state machine, of 2427 states, was synthesised in 2951ms on a laptop with an Intel Core i5 2.3GHz CPU and 4Gb memory.

5.1 Graceful Degradation

The four tiers of our control stack mean that the level of service can be degraded, in response to uncertainty in the environment, three times before failing completely (provided that the assumptions of the higher tiers are violated before those of lower tiers). A controller synthesised for a single model and single goal will fail completely the first time an unexpected event is encountered.

When the control stack is operating in tier 4, one of the possible traces leading to degradation is:

```
arrive['r5], base, goto['r1], count[0], arrive['r1],
reward[2], tier_disabled4
```

The `tier_disabled4` event is the exception marker used by the enactor to track the current service level. The exception occurs in this case because the tier 4 environment model does not allow `reward[2]` in region `r1`, and yet this is what happened in the runtime environment. After this exception, the control stack is operating in tier 3, achieving the tier 3 requirement. It may continue in this tier indefinitely, in the case where the runtime environment matches the tier 3 abstraction. On the other hand, one possible trace (continuing from the above trace) leading to further degradation is:

```
goto['r1], count[1], arrive['r1], reward[2],
...
goto['r1], count[5], arrive['r1], reward[2],
tier_disabled3
```

This exception occurs because the runtime environment has broken the bound given in the model, which states that region `r1` must provide a `reward[1]` within 5 attempts.

The control stack continues to operate in tier 2. Again, execution may continue from this point achieving the tier 2 requirement indefinitely. There is however the possibility of a further degradation as follows:

```
goto['r1], arrive['r3], tier_disabled2
```

This leaves the control stack operating in tier 1. A further sequence of events that lead to an exception is as follows:

```
reward[3], goto['r2], arrive['r2], damage[2],
tier_disabled1
```

Note that although we have presented the degradation from tier to tier, it is possible that the tier 1 assumptions are violated immediately, bypassing the intermediate tiers. After all tiers are disabled, the control stack cannot guarantee any goals until progressive enhancement takes place.

5.2 Progressive Enhancement

Suppose now that the stack is operating in tier 2, that is, the bound related to `reward[1]` has been broken, but the motion of the robot remains reliable. A progressive enhancement may now be considered in order to raise the service level back into tier 3. Suppose that the rewards given by the environment while in tier 2 have been observed, either automatically under a machine learning scheme or through manual intervention. Suppose further that it has been determined that, after the transient disturbance which caused the degradation from tier 3 to tier 2, the environment does in fact provide `reward[1]` within the required bound[1]. In such a situation, the initial state of the tier 3 model would be approximated. For instance, if the last executed action (in tier 2) was `goto['r1]` then the state of the tier 3 model must be one where `arrive['r1]` can occur. An attempt would then be made to synthesise a controller, and if successful the service level would be raised to tier 3.

6. EXPERIMENT

Figure 2: Nao executing mission

We have experimented with our synthesis and enactment infrastructure [3] in various robotic settings, including an AR Drone 2.0, a Katana robotic arm and a Nao H25 humanoid robot. A video of the latter executing a synthesised mission control stack similar to the one described above can be found at
`http://www.doc.ic.ac.uk/~das05/quadrotor3.avi`.

Controller enactment for these settings requires implementing each of the controlled actions in the control stack specification in terms of the existing behaviours provided by the robot's API. For instance, for the control of the Nao robot, the detection of various types of reward is achieved by recognising balls of different colours using the Nao's on-board camera. The balls are presented to the Nao upon arrival in each region. The location of the Nao within the environment (an office) is determined using trilateration with respect to a number of landmarks in positions known *a priori* (i.e. a structured environment). The landmarks themselves are recognised using the on-board camera. Similarly, rewards and locations can be recognised on the AR Drone using its front and bottom cameras respectively.

[1]In the case of manual intervention, it would be reasonable to amend tier 3 to match a different observed bound.

The synthesised mission control stack is executed by the enactor, which starts by assuming the runtime environment behaves like the model in the upper tier. Initially, in the video, we allow this assumption to hold by providing the Nao with the reward it is expecting. Later, we break the bound on the number of damage events expected in the uppermost tier, forcing the enactor to gracefully degrade the level of service. Execution continues seamlessly such that the Nao immediately seeks a repair, as demanded by the lower tier requirement.

The experiments demonstrate that our general approach can be deployed in a robotics setting on top of a high-level API that encapsulates the complexities of, for instance, control of the system dynamics that allows stable movement of the AR Drone or the localisation of the Nao robot. The resulting system can then ensure that mission-level guarantees can be gracefully and automatically degraded (or enhanced) when necessary to cope with unexpected mission-level events in the environment.

7. RELATED WORK

There has been an increasing interest in the development of robot planning motion techniques based on temporal logic (TL) specifications [30, 4, 29, 19, 4, 28, 21]. One of the main features such techniques provide is the ability to guarantee satisfaction of the goals if the environmental assumptions are met. This is especially relevant in robotics where strict safety conditions must be ensured by robot motion plans, which has been acknowledged by the community as a key problem to be tackled [13].

In addition to safety properties, TL-based approaches provide efficient algorithms for generating plans for liveness goals that allows a wide range of mission objectives to be specified such as surveillance and navigation, among many others. Techniques such as [29] and [6] automatically synthesise high-level motion plans from discrete-event descriptions of the environment and goals described as GR(1) [24] properties. In [28] goals are specified as co-safety formulas [20] to be satisfied by a non-deterministic model of the environment.

Other approaches [21, 22, 30, 4] consider settings where the environment is represented with stochastic transition systems such as Markov decision processes and the goals are expressed with temporal logic supporting probabilistic reasoning such as probabilistic computation tree logic (PCTL). The resulting plans guarantee the satisfaction of the goals up to a certain probability or expected reward.

However, all the aforementioned techniques have a single environment model with a fixed level of risk. Hence, if the real environment diverges from its model the plan would fail with unforeseen consequences. Thus, engineers must balance carefully the increased risk of introducing strong assumptions that allow achieving sophisticated goals against the robustness of having weak assumptions at the expense of only being able to achieve simpler goals.

Controller hierarchies have been studied (e.g [16]), however focus is on synthesis-time scalability rather than runtime robustness to invalid environment models.

Although we have implemented our approach in the MTSA toolset [9], this is not the first tool in implementing LTL-based controller synthesis. Tools such as Lily [17], Acacia+ [2] or Unbeast [10] implement techniques for synthesising controller from general LTL specifications. Such specifications are known to be 2EXPTIME Complete. Consequently, we restrict attention to GR(1) as it allows for tractable synthesis procedures. A number of tools supporting synthesis from GR(1) specifications have been developed [18, 25, 1, 26]. However, none of them work for event-based models which are central to our specification procedure.

8. CONCLUSIONS

In this paper we have presented an approach for robust high-level control synthesis for robot missions, and applied it in a various scenarios. In contrast to the 'all or nothing' approach of other work based on temporal logic, our approach allows a mission specification to include a range of requirements of different 'strengths' which entail different levels of risk when operating in the runtime environment. Our implementation ensures that when the stronger requirements of higher tiers cannot be met due to environmental uncertainty, the level of service degrades gracefully to a level at which requirements can be guaranteed. It then permits progressive enhancement at a later stage.

In future work we are interested in quantifying the level of risk associated with the tiers of our control stack, and combining the approach with techniques that can learn appropriate environment models for disabled tiers in the stack before progressive enhancement.

9. REFERENCES

[1] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. Ratsy: a new requirements analysis tool with synthesis. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 425–429, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin. Acacia: a tool for ltl synthesis. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 652–657, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel. Controller synthesis: From modelling to enactment. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1347–1350. IEEE Press, 2013.

[4] I. Cizelj and C. Belta. Control of noisy differential-drive vehicles from time-bounded temporal logic specifications. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2021–2026, May 2013.

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

[6] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran. Softw. Eng. Methodol.*, 22, 2013.

[7] N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel. Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In *ICSE*, pages 688–699, 2014.

[8] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models for fallible domains. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 211–220. ACM, 2011.

[9] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 475–476, Washington, DC, USA, 2008. IEEE Computer Society.

[10] R. Ehlers. Symbolic bounded synthesis. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 365–379, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE 2009*, pages 111–121. IEEE, 2009.

[12] 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.

[13] T. Fraichard and J. J. K. Jr. Guaranteeing motion safety for robots. *Auton. Robots*, 32(3):173–175, 2012.

[14] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *ICSE*, pages 277–287, 2014.

[15] 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.

[16] R. C. Hill, J. E. R. Cury, M. H. de Queiroz, D. M. Tilbury, and S. Lafortune. Multi-level hierarchical interface-based supervisory control. *Automatica*, 46(7):1152–1164, July 2010.

[17] B. Jobstmann and R. Bloem. Optimizations for ltl synthesis. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '06, pages 117–124, Washington, DC, USA, 2006. IEEE Computer Society.

[18] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 258–262, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] H. Kress-Gazit, G. Fainekos, and G. Pappas. Temporal-logic-based reactive mission and motion planning. *Robotics, IEEE Transactions on*, 25(6):1370–1381, Dec 2009.

[20] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, Oct. 2001.

[21] M. Lahijanian, J. Wasniewski, S. Andersson, and C. Belta. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3227–3232, May 2010.

[22] A. Medina Ayala, S. Andersson, and C. Belta. Temporal logic control in dynamic environments with probabilistic satisfaction guarantees. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3108–3113, Sept 2011.

[23] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

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

[25] A. Pnueli, Y. Sa'ar, and L. D. Zuck. Jtlv: A framework for developing verification algorithms. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 171–174, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] V. Raman and H. Kress-Gazit. Synthesis for multi-robot controllers with interleaved motion. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 2014.

[27] 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.

[28] A. Ulusoy, M. Marrazzo, K. Oikonomopoulos, R. Hunter, and C. Belta. Temporal logic control for an autonomous quadrotor in a nondeterministic environment. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 331–336, May 2013.

[29] E. Wolff, U. Topcu, and R. Murray. Efficient reactive controller synthesis for a fragment of linear temporal logic. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5033–5040, May 2013.

[30] C. Yoo, R. Fitch, and S. Sukkarieh. Provably-correct stochastic motion planning with safety constraints. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 981–986, May 2013.