Runtime Verification meets Controller Synthesis[⋆]

Shaun Azzopardi[0000−0002−2165−3698], Nir Piterman[0000−0002−8242−5357], and Gerardo Schneider[0000−0003−0629−6853][⋆⋆]

University of Gothenburg, Gothenburg, Sweden
{name.surname}@gu.se

Abstract. Reactive synthesis guarantees correct-by-construction controllers from logical specifications, but is costly cost — 2EXPTIME-complete in the size of the specification. In a practical setting, the desired controllers need to interact with an environment, but the more precise the model of the environment used for synthesis, the greater the cost of synthesis. This can be avoided by using suitable abstractions of the environment, but this in turn requires appropriate techniques to mediate between controllers and the real environment. Runtime verification can help here, with monitors acting as these mediators, and even as activators or orchestrators of the desired controllers. In this paper we survey literature for combinations of monitors with controller synthesis, and consider other potential combinations as future research directions.

Keywords: Reactive synthesis · Controllers · Runtime verification · Monitors.

1 Introduction

Developing programs correctly is difficult, always requiring iteratively finding and repairing bugs. The traditional formal solution is model checking, which can be employed to confirm the correctness of programs with respect to some specification (often in linear temporal logic). Recently, significant effort has instead gone into *reactive synthesis*, the study and development of techniques that automatically produce correct-by-construction programs from LTL specifications [26, 25, 19]. See, for example, [13] for an introduction to reactive synthesis.

In the setting of an adversarial environment, where certain goals are desired, reactive synthesis techniques can be used to determine if these goals are always achievable and if the answer is positive, to construct a program that achieves these goals. This kind of program is called a *controller*, and both the goals and the environment are usually described in linear temporal logic (LTL).

⋆ This research is funded by the ERC consolidator grant D-SynMA under the European Union's Horizon 2020 research and innovation programme (grant agreement No 772459)

⋆⋆ Corresponding Author.

Unfortunately, practical considerations can make this costly — already for LTL specifications the problem is 2EXPTIME-complete [26]. In what concerns cost, on the other hand, formal verification allows one to verify whether an existing system satisfies the desired specification, at a cheaper price (e.g., PSPACE-complete for model checking of LTL [28]). This is even more true for runtime verification on single execution prefixes at runtime [18]. However, reactive synthesis is still more alluring, since it promises to solve the problem of automatically developing a controller for whatever the environment does, when possible, at least in a theoretical setting — the price is worth paying.

Applying reactive synthesis in a practical setting however may not be easy. One problem is that a desired controller may need to interact with systems whose full representation in LTL can be very large, the size of which is a parameter to the 2EXPTIME-complete complexity of reactive synthesis. This problem can be handled by using appropriate abstractions of the environment. For example, if the environment is at times engaged in a process that requires several actions in sequence, but the reactive synthesis problem is only interested in the point this process ends, then we can do with only one proposition that is true when the process ends. This avoids the need to have a proposition for every possible state of the environment. However, applying such high-level controllers in practical settings requires an interface with the real environment that adapts real-world events to the expected higher-level, abstract events. In literature we find *monitors* used in such a manner.

Monitors are objects that observe a system. In runtime verification they are used to output, given a certain execution prefix, whether the system currently satisfies or violates a certain property [18]. Synthesising such monitors is cheap but when combined with reactive synthesis they open up the possibility to solve problems not expressible in LTL (see Section 3), or at least to reduce the size (and thus cost) of the reactive synthesis problem. Essentially, their use can be to abstract away parts of specification, to reduce the synthesis problem to a kernel of the original specification.

In literature we find several ways in which monitors are used thusly. In this paper we set out to survey these combinations, mostly based on our own experience and contributions in this line. More concretely, we classify the approaches found in literature as according the following:

i) Monitors to identify what guarantee should start holding and when (Section 3);
ii) Monitors for environment assumptions (Section 4); and
iii) Monitors as orchestrators between controllers (Section 5).

Before presenting these, in the next section we give some brief background to runtime verification and reactive synthesis, and after in Section 6 we consider and propose other combinations providing future directions for research, and we give our concluding remarks in Section 8.

2 Background

We give a brief background on runtime verification and reactive synthesis, with a running example of a monitor and a reactive synthesis problem.

2.1 Runtime Verification

Runtime verification (RV) is a lightweight verification technique, consolidated in between informal non-exhaustive techniques like testing and formal static verification techniques like model checking, static verification and theorem proving. In a way, RV is understood to be a good complementary approach surmounting the weaknesses of those techniques.

RV techniques allow the construction of monitors for the execution of software systems in order to ensure the validity of a given property, or the detection of its violation, at runtime [15, 18]. These monitors can be constructed from higher-level formal specifications, usually LTL, but also from more operational and more expressive symbolic automata.

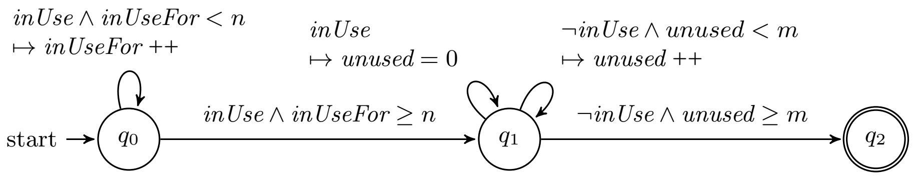
$inUse \wedge inUseFor < n$
$\mapsto inUseFor$ ++
$inUse$
$\mapsto unused = 0$
$\neg inUse \wedge unused < m$
$\mapsto unused$ ++
start
q_0
$inUse \wedge inUseFor \geq n$
q_1
$\neg inUse \wedge unused \geq m$
q_2

Fig. 1. Monitor that checks that the room has been in use for n time steps, after which when there is a period of m time steps where the room is empty it flags.

Symbolic automata monitors are finite-state automata, with internal variables. Rather than labelling transitions by events, these automata's transitions are labelled by propositional guards over events and internal variables, and actions on the internal variables. This allows succinct program-like specifications, leaving it up to the writer how much information to encode in the control-flow and how much to make symbolic.

An example is the monitor illustrated in Fig. 1. This monitor represents operationally some implementation that, through sensors, detects when people are in the room (*inUse*). Moreover, it deduces that the room needs cleaning depending on how long the room has been in use and after that waits for enough time to pass until the room no longer has any foot traffic ($\neg inUse$). Reaching q_2 is the point at which the monitor identifies the room as being dirty, and thus ready to be cleaned.

There are several variations on these monitors that include richer features, like a notion of timers and communication over channels [11]. In runtime verification they have been used to monitor programs written in different programming languages, e.g., Java [11], and Solidity [4].

```
INPUTS { ROOMCLEAN; INROOM; DOORLOCKED; }
OUTPUTS { GOTOROOM; LEAVEROOM; CLEAN; LOCKDOOR; }

ASSUMPTIONS {
    G F (!GOTOROOM || INROOM);
    G F (!LEAVEROOM || !INROOM);
    G F (!(LOCKDOOR && INROOM && CLEAN) || ROOMCLEAN);
    G ((DOORLOCKED && ROOMCLEAN) -> X ROOMCLEAN);
    G ((!ROOMCLEAN && !CLEAN) -> X !ROOMCLEAN);
    G ((!INROOM && !GOTOROOM) -> X !INROOM);
    G ((INROOM && !LEAVEROOM) -> X INROOM);
    G (LOCKDOOR -> X DOORLOCKED);
    G (!LOCKDOOR -> X !DOORLOCKED);
}
GUARANTEES {
    F (ROOMCLEAN && (X F !INROOM) && (X F !DOORLOCKED));
}
```

Fig. 2. Snippet in TLSF [16] format for the reactive synthesis of a cleaning robot.

2.2 Reactive Synthesis

Controllers are programs that control for a certain goal in the context of some environment. Focusing on reactive controllers, the problem of *reactive synthesis* is to construct controllers from *linear temporal logic* (LTL) specifications [26]. We assume faimiliarty with LTL, but for completeness, LTL is the language over a set of atomic propositions AP, with negation, conjunction, the unary operator next (X), and the binary operator G. The eventually operator (F) is often used, where $F\phi \overset{\text{def}}{=} \mathit{true} U \phi$.

Specifications in reactive synthesis are of the form:

$$\mathcal{A} \to \mathcal{G},$$

where $\mathcal{A}$ is the *assumption* specification, restricting the way variables can be controlled by the environment, and $\mathcal{G}$, the *guarantee* specification, describing behaviour the desired controller should induce. The environment is assumed to be adversarial, and when for every possible environment behaviour the controller can always ensure $\mathcal{G}$ then problem is said to be *realisable*, otherwise it is said to be *unrealisable* and then no controller exists. This problem of LTL realizability is known to be 2EXPTIME-complete [26].

Consider Fig. 2 as an example of a reactive synthesis problem for a robot to clean a room. We have a partition of propositions into *input* (*uncontrollable*) and *output* (*controllable*) propositions. The assumptions are an abstraction of the robot's environment and the effect of the robot's actions (outputs) on the environment. The guarantees require that a robot synthesised using this specification will eventually clean the room, and then leave the room while leaving the room unlocked. This problem is realisable: Fig. 3 shows a controller for it.

3 Monitoring for Guarantees

Monitors can be used to identify the point at which a certain property is satisfied or violated. In reactive synthesis we often want to activate the obligation for the controller to satisfy some guarantee once the environment behaves in some way. In literature we find two ways in which monitors are exploited to aid this. Initially, we look at our own work [5], that consider *a priori* monitors that identify points where we require some guarantees. We also look into an approach by Finkbeiner et al. [14], where guarantees become known only at runtime. In that work monitors are used to ensure outstanding co-safety obligations arising from the initially required guarantees hold even when requirements change at runtime and a new controller needs to be generated and used.

Monitor Triggers We start with an example, based on the cleaning robot example. Consider we have an *a priori* precise notion of when the room is dirty and needs cleaning, given as a program-like monitor, as illustrated in Fig. 1. Consider the problem of finding a strategy for a cleaning robot to ensure a room is clean. The question then is how to synthesise a strategy for a cleaning robot to ensure the room is eventually cleaned *whenever* this obligation starts holding (at q_2).

In [5] we propose to specify such problems with the combination of symbolic monitors ($\mathcal{M}$) and LTL (assumptions A and guarantees ϕ), in two ways: simple triggering $\mathcal{A} \to \mathcal{M} : \phi$ and triggering with repetition $\mathcal{A} \to (\mathcal{M}; \phi)^*$. The point at which a monitor triggers the obligation to satisfy ϕ is clear, given appropriate final states in $\mathcal{M}$. However, an important aspect, is that it is not obvious at which point a controller for ϕ should return back control to $\mathcal{M}$. For general LTL formulas there is no specific point at which an obligation is satisfied, e.g., an invariant $G\phi'$ would have to continue enforcing ϕ' forever. However, for co-safety formulas there is such a natural point: the end of a tight or *satisfaction-informative* prefix (the dual of the violation-informative prefix of [17]). [5] gives a 2EXPTIME-complete algorithm to construct a controller that finishes executing when such a point has been reached, remaining within the same complexity class as reactive synthesis. Simply plugging in such a controller $\mathcal{C}$ for $\mathcal{A} \to \phi$, with $\mathcal{M}$, results in $(\mathcal{M}; \mathcal{C})^*$, a controller for the original formula.

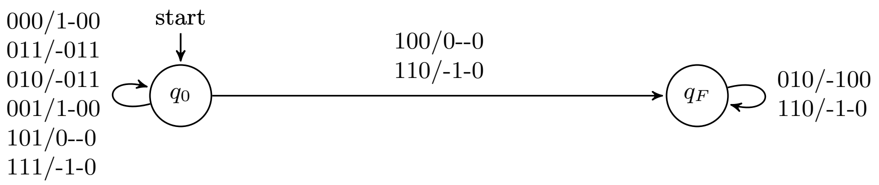

Fig. 3. Cleaning robot controller, as given by Strix [19] — inputs (outputs) on left(right)-hand side of transition labels, with 0 (1) at nth position for input (output) denotes nth proposition in input (output) list (from Fig. 2) being false (true).

This approach has limitations. Assumptions about the environment have to be stateless, given the controller synthesis is done for $A \to \phi$. For stateful assumptions one would have to reason about their state at points where the monitor triggers, and at the point the required controller gives back control to the monitor. Moreover, to have many such trigger constructs running in parallel is problematic, since it does not allow to re-use the compositional synthesis method we described, e.g. the synthesis of a controller for $(M; \phi) \wedge (M'; \phi')$ cannot be delegated to just synthesising a controller for ϕ and for ϕ' independently, but requires synthesising both together.

On the other hand, properly abstracting the monitor in LTL (which is possible in this case) results in very large controllers even for small values of n and m in the monitor. For example, setting $n = m = 2$ in $\mathcal{M}$, and combined with $\mathcal{G}$ appropriately, Strix [19], the state-of-art tool in LTL reactive synthesis, produces a controller with around four times the transitions as that of the symbolic monitor and the tight controller for $\mathcal{G}$. Moreover, this continues to increase as n and m increases, while with our approach changing the value of the parameters does not change the size of the final controller.

Obligation Monitors In our work, there is an *a priori* description on when an obligation is in effect, and of the obligated LTL requirement itself. These may however both be relaxed. In general, reactive synthesis assumes that the guarantees do not change at runtime. However this is not necessarily the case in a practical setting — often requirements change and systems are taken offline. Monitors have been used to aid the process of updating such obligations.

Finkbeiner et al. [14] consider the problem of *live synthesis*, where a system may be replaced by another at runtime. They consider that at the point where the replacement occurs, there may be outstanding obligations from the requirements of the first system that would be violated if it is simply replaced with a synthesised controller for the new requirements. Instead, they require that the new controller also satisfies these outstanding obligations. The problem then is how to identify what the obligations are at the exact point of replacement.

For example, consider a setting where we want to re-assign the cleaning robot to clean other room. If we do the handover between the old and new strategy naïvely, the robot could leave the first room unclean. A more natural requirement on this handover would ensure the robot achieves one last time its co-safety requirements in the first room, before moving to the second room. For example, consider a setting where we want to re-assign the cleaning robot to clean other room. If we do the handover between the old and new strategy naïvely, the robot could leave the first room unclean. A more natural requirement on this handover would ensure the robot achieves one last time its co-safety requirements in the first room, before moving to the second room.

Considering outstanding obligations as co-safety properties, Finkbeiner et al. [14] use monitors to consider any remaining outstanding obligations. They synthesise monitors at the same time as the original controller is synthesised, that monitors for the remaining obligations given any execution prefix of the controller. Then, when a system update must be effected then the new controller is synthesised to also satisfy the obligations the monitor signals as outstanding.

In our example, if the cleaning robot has not yet managed to clean the room, then the new strategy will take this into account and clean the room, unlock the door, and leave the room, before leaving for its new mission. Similarly, if it has cleaned the room but not yet unlocked the door and left the room, it will do so before attempting the next mission.

Compare this also with the approach of [24], where additional specifications are required for the transition between the new and old specifications. This work also assumes additional signals that control the transition process.

4 Monitoring for Assumptions

As mentioned, commonly, a reactive synthesis problem has two parts: *guarantees* ($\mathcal{G}$) we wish a controller to control for in the context of some *assumptions* ($\mathcal{A}$) about the environment, i.e. the specification $\mathcal{A} \implies \mathcal{G}$. If the assumptions can be invalidated by the controller then the specification becomes true, and the controller has 'won' since the implication then becomes trivially true whatever the controller does after. This is not always ideal. Often, we want to exclude solutions to a reactive synthesis problem where the controller invalidates the assumption, since it is not usually the intention of the designer. Excluding these kinds of solutions usually requires expertise in modelling the environment in the assumptions appropriately. There is however work to exclude these automatically, e.g., for the GR(1) subset of LTL [20].

Moreover, reactive synthesis is adversial, i.e., the environment is assumed not to invalidate its own assumptions, but in a practical setting the real-world environment may actually violate the constraints that were assumed. This may be due to imprecise modelling or unpredicted changes that occur during the lifetime of the system. Theoretically this is not problematic, since the controller simply wins in this case. Practically this can be a problem — an assumption violation may go unnoticed and lead to a controller working based on information that is not accurate. This can lead it to trying to fulfil a now impossible goal, resulting in a waste of resources at best and at worst interfering with the goals of other agents. In this setting, the controller would ideally halt or change its behaviour once the environment no longer satisfies our original assumptions about it, for which we can exploit monitors. Monitors for environment assumptions can be run concurrently with the controller, stopping the controller (or taking other actions) if an environment assumption is violated. We consider two works along this direction.

Unmodelled Environment Behaviour In a practical setting, the designer may not give exact or even sound environment assumptions, but instead give a necessary abstraction of the environment assumptions. This measure may be taken to reduce complexity, or in the case that the environment model is not discrete or expressible in LTL. In this case we do not fulfil our goals simply by the environment remaining within the bounds of our necessary version of the assumptions.

Here, having an environment assumption monitor running concurrently with the controller also can be useful, to signal possible failure of the controller, due to unmodelled environment behaviour. This setting is considered by Zudaire et.al. in [32], and described further below.

In a hybrid setting, a description of the environment and guarantees are given in a language richer than LTL (e.g., with events carrying real numbers). To use automatic reactive synthesis techniques entails discretising these descriptions, and thus abstracting them. In this case, monitoring the LTL version of the environment assumption may not be enough. For example, instead of describing a sensor's behaviour in detail, its LTL abstraction may simply be abstracted with one boolean variable that is set as true when the sensor's value is above a certain value. However, at runtime a sensor may become miscalibrated due to external factors (e.g., weather), although in the reactive synthesis problem we may choose not to model this (since we may not necessarily be able to control for the non-modelled factors).

In [32], Zudaire et.al. handle the above problem by combining temporal task planning with streaming runtime verification. Discrete task specifications are enriched with an explicit representation of the assumptions about the continuous world allowing, in this way, the monitoring of the validity of these assumptions, and thus reacting to it accordingly. See [32] for an application of the technique to three different scenarios in the context of Unmanned Aerial Vehicles.

In this case, instead of monitoring for the complete LTL abstraction, we can monitor for a richer specification of the abstraction that captures the non-modelled behaviour. This monitor can then be used to notify when realisability of the concrete problem is no longer ensured, due to non-modelled factors. It can also be used in a *predictive* manner, where the possibility of failure is detected before it occurs and (continuous) control measures are taken to prevent it (e.g., recalibration of sensor is carried out when accuracy falls beneath a certain level, before a collision occurs).

Unknown Environment In another case, the environment may not be known. A sound assumption to make here would be simply *true*, but this would make any interesting goal unrealisable, given it puts no limitations on the environment. However, there are techniques to identify environment assumptions for which a goal is realisable (e.g., [7, 8]). If these assumptions are reasonably minimal and weak, they can be used to synthesise a controller for the goal that can work in a number of settings. Moreover, in the general case, beliefs about the environment may turn out to be false and at runtime unexpected behaviour may be exhibited.

Environment monitors in these cases can also be used to monitor for the environment assumptions used for synthesis, but moreover they can be used at runtime to adjust these assumptions. In this line of work, Wong et al. [30] propose that when a violation is detected, the environment behaviour at runtime can be used to re-synthesise a new controller that takes this new behaviour into account. Before this, if the assumption violation still allows progress, the controller maintains safety and takes action to allow progress when the environment returns to the expected behaviour.

5 Monitors as Orchestrators in Contextual Missions

In [22] Maoz and Ringert list three main challenges concerning the application of reactive synthesis to the robotics domain. The first challenge concerns the scalability of using high-level declarative specification languages like LTL. Reasoning in LTL is not something developers are used to — they are used to more imperative languages, while LTL specifications can quickly become very complex. Moreover, writing appropriate assumptions (and guarantees) is an art in and of itself, and inadequate assumptions can lead to problematic implementations, as discussed in the previous sections. The second challenge is the need for techniques to abstract from time and complex data, to allow for more tractable synthesis. The final challenge relates to having appropriate support for the development process, e.g., a tool integrating both verification and testing of the controller, to allow the developer to check the controller respects their intentions, or techniques that allow re-using previous synthesis artefacts when the specification evolves (to avoid full re-synthesis).

The use of monitors has been proposed to help tackle these challenges (see an initial proposal by Mallozzi et al. in [21, Chap. 7]). In particular: (1) monitors add a level of imperativeness to a specification (as we also saw in Section. 3); (2) reasoning about time, data, and complex behaviour can be relegated to monitors, while still being represented in a formal structure, amenable to verification; and (3) the use of monitors as orchestrators between different guarantees/controllers can allow different parts of a specification to be synthesised separately (with some caveats). In this section we briefly describe an application of monitors with reactive synthesis as described in [21, Chap. 7] — a framework using monitors as orchestrators in robotic missions with contextual missions/guarantees.

Let us consider a very simple example: an robot (or agent here) needs to clean rooms r_1 and r_2 (task 1) one after the other during the day, and rooms r_3 and r_4 (task 2) during the night (see Fig. 4). Switching between tasks happens when the *context* switches (from day to night and vice-versa).

Synthesising a controller for the above simple example is within reach using existing techniques, however the synthesis needs to be done at a global level and not compositionally, thus a certain context's task changing (e.g., the layout of furniture in one room is changed) requires re-synthesis for the whole mission. Ideally, when possible, one would want to synthesise only for the changed tasks. Moreover, should the context not be expressible in LTL, appropriate abstractions (perhaps of time or complex data) would need to be used, adding to the complexity of the problem.

However, in these kinds of specifications one can attempt to do better. Under different contexts, the robot has different tasks, which suggests that it would be useful if the controller for each tasks could be synthesised separately and then integrated later somehow with appropriate logic for context monitoring. This first solution is not without its problems thought: trying to do a certain task may have an effect on the state of another tasks. In a monolitic reactive synthesis approach this is handled smoothly (the controller is aware of the states of every task), but attempting to do things compositionally requires extra work

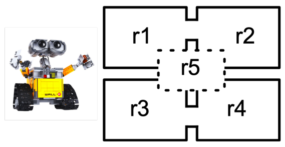

Fig. 4. A robot cleaning 4 regions.

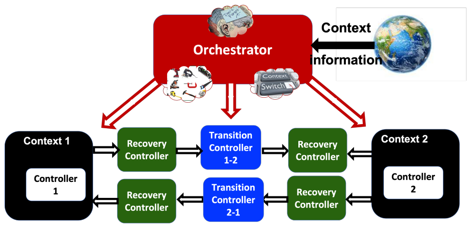

Fig. 5. A high-level representation of contextual robot missions for 2 contexts.

to transition smoothly between tasks. This however may be unavoidable when the specification required goes beyond LTL, or involves large numbers of variables such as to make the synthesis problem intractable.

For that, Mallozzi et al. in [21, Chap. 7] suggest an architecture and development environment composed of the following key components:

1. *Agents:* The agent has tasks (LTL specifications), each associated with a certain context (explained next). These tasks are achievable through appropriately synthesised controllers acting under different contexts[1]. They can be automatically synthesised from LTL specifications, or other specification languages like Spectra [23]. The authors also allow programs in general.
2. *Contexts:* One assumes a set of contexts (in the example here *day* or *night*), and a function that determines the current context from some knowledge about the current state of the world. For example contexts can be geographical, like the neighbourhood of a city the controller is currently in; or something more complex, like being in a certain neighbourhood at a certain time of day. An important aspect is that context can change, i.e. the current context depends on the current point in time, and the previous behaviour of the environment and the system.
3. *Recovery controllers:* Agents have no control on which context they are in at any time-step. Thus, when the context changes, they are expected to transition seamlessly to making progress in the task of the next context. However, the state of the task of this next context may have changed since the last time the agent was in that context (e.g., someone may have moved some things around in a room), and thus the agent might need to do some *recovery* task to either get the agent or the environment in a state where the agent can continue where they left off (e.g., returning the things moved around to their previous position). Similarly, before leaving a context, it is reasonable to pre-specify that the agent leaves it in a safe state (e.g., turned off any lights). Recovery controllers serve these purposes.
4. *Transition controllers:* This is activated after the recovery controller leaves the state in the previous context in a sane state, and activates the recovery controller to prepare the state of the new context for the agent to continue.

[1] Some of these may be written manually, or provided *a priori*, and appropriately model checked.

5. *Orchestrator:* This is in charge of notifying the controllers of the change in context in the real-world, and triggering of the actual change in context in the system, as illustrated in Fig. 5.

Such a framework integrates the use of reactive synthesis in a much richer setting. Using runtime monitors as orchestrators (or context monitors) in this case adds considerable power: one can easily write simple monitors with the capabilities of counting, accumulating, computing averages, memorising current states, etc., besides the fact of allowing for parameterised specifications. There is then a tradeoff between the strong guarantees about what is synthesised and the expressive power of the orchestration against the requirement to have to specify recovery transitions However, this can be managed in this setting through the use of verification, which is possible for domains beyond synthesis.

Note that [21, Chap. 7] only presents an *ad hoc* solution to the problem given in Fig. 4 when contexts are given by propositional variables ("night" and "day"). A more general solution along the lines of the architecture presented in (Fig. 5) for more general contexts and that can handle dynamic changes, is still missing.

6 Other Potential Combinations

We have seen monitors used effectively alongside controllers in extending the scope of reactive synthesis. In each of these uses of monitors, the interaction between monitors and controllers is, however, very specific and limited. Limiting interaction has its benefits — it preempts the need for reasoning about complex interactive behaviour. However, it also prevents interesting deeper applications. We consider further possible combinations here.

Concurrent Monitors and Controllers We can imagine discrete environment events that can be controlled by monitors, e.g., a monitor written in a rich language can be used to monitor the continuous world to notify a discrete controller about some event. Some approaches, as we saw in Section 4, simply ignore these more complex specifications, and use complete restrictions of them (e.g., that a sensor can never become miscalibrated), with a monitor attempting to take care of their violation at a higher-layer. This approach however trades overall correctness guarantees of the hybrid controller, for easier automatic synthesis, since the monitor's behaviour may be crucial for realisability of the full problem.

As a first step, we are currently exploring the incorporation of more program-like descriptions of the environment (encoded as symbolic monitors) in a reactive synthesis problem. This would allow including complex specifications or implementations of the environment in a reactive synthesis, and through an appropriate (sound) abstraction-refinement loop automatically synthesising a controller. This will reduce the current necessary abstraction decisions the developer would have to do, while it extends the scope of reactive synthesis to richer domains.

We are also considering the use of monitors to abstract away parts of a given LTL formula that are purely about the environment. Such an approach could enable to reduce a synthesis problem by replacing some sub-formula with a new variable, and composing the controller for the abstracted formula with the monitor for the removed sub-formula. The crucial aspect of this problem is that assumptions about the new variable may need to be added to make the abstract problem realisable, and in the worst case full knowledge of the hidden sub-formula may be needed. For example, consider a co-safety formula about the environment ϕ, if we have a guarantee $G(\phi \implies Fc)$, where c is a controller event. If this is all we have, then replacing ϕ in the guarantee by a fresh environment event e suffices, and the controller generated can be composed with a monitor for ϕ (that outputs e at every step ϕ is tightly/informatively satisfied [5]) to create a controller for the original problem.

Monitors for Discretising Data-Carrying Events Another problem to be tackled is that of controllers for specification that require more than a finite alphabet, but also data-carrying events. Consider a standard arbiter example, where we want every request to eventually be matched by a grant. A standard way of specifying this in LTL is to not allow new requests in between a request and the corresponding grant, given the limitations with regards to counting in LTL. A more general specification would be able to index an incoming request, and check that there is a grant with a matching index. Consider a monitor with a counter, that decorates a given request with the current value of the counter, increases the counter, and then triggers a controller to eventually perform the associated grant. The controller can be parametrised with a number, such that the events it generates are parametrised by this number, and the monitor can then activate such a distinct controller each time a request comes in. Such a system has an infinite alphabet, but re-uses simpler LTL reactive synthesis techniques.

Running controllers in parallel is not simple however. These controllers may be sharing resources, and thus the synthesis from a grant specification needs to be modified such that a controller is aware there may be other controllers working in parallel. A monitor here can also be used to mediate between the controllers. A controller only needs to be aware of its own version of the events (tagged by some number), and that the same events with a different number can occur. A naïve solution would seem to require the controller being aware of the whole infinite alphabet, however a monitor can be used to turn any event labelled by another number into a generic *other* event, that can still be enough for realising a controller.

7 Related Work

Monitors have been been used as enforcers of properties at runtime, where if a system, acting as a *shield* [6]. This is done through suppression and/or insertion of system events by the enforcer, ensuring the property holds at runtime. The setting of these approaches is usually that of an existing system and a shield that enforces some (co-)safety or infinite-renewal properties about the system. Reactive synthesis of LTL is more general that LTL runtime enforcement, given the limitations of monitoring with regards to liveness properties.

Runtime verification has been exploited in other contexts to manage the expense or limitations of other techniques, e.g., static verification [1, 9, 2, 3] and testing [10]. Moreover it has found applications beyond the dynamic analysis of software, see [27] for a survey.

Other work that combines in some way monitors with reactive synthesis is by Maoz and Ringert. In [23] they consider a language that includes a notion of monitors, but these become part of the specification to be controlled, not benefiting from treating monitors separately. Ehlers and Finkbeiner consider a monitoring approach where prefixes are classified into whether the specification remains realisable or not, or has been violated or fulfilled [12]. Chou et al do something similar, but for linear stochastic systems [31].

An *ad hoc* combination of monitors with reactive synthesis is given in [29]. In that paper, Ulus and Belta use runtime monitors to check and enforce safety properties (e.g., to avoid certain locations or collision for robots), while control techniques enforce the high-level mission. Monitored properties are expressed as regular expressions and past temporal logic formulas. Monitors are also used to discard unsafe trajectories from a proposed set of trajectories.

8 Conclusions

Reactive synthesis provides strong guarantees, but becomes harder the more precise and rich a specification language. In this paper we have seen how runtime monitoring techniques have been used to increase the scope of reactive synthesis without increasing the hardness, by relegating parts of an envisioned system to a monitor. We have surveyed techniques in literature along these lines, including our own previous work, and identified three main kinds of monitor-synthesis synergies: monitors to identify or trigger obligations; monitors that identify assumption violation; and monitors that act as orchestrators between controllers.

We identified some promising possible combinations. One is the use of monitors in parallel with controllers, where monitors can even correspond to some general non-regular property of the environment, or to a sub-formula of the LTL specification we want to synthesise. We have also described how monitors can have more control by dynamically activating copies of the same controller, possibly in parallel, and mediating between these (e.g., to allow use of shared resources), and decorating of their outputs according to some scheme, to ensure, for example, that the nth request is matched by a grant from the nth controller.

We hope the reader may find other novel and interesting ways to combine in a systematic and useful way runtime verification and reactive controller synthesis.

References

1. Ahrendt, W., Chimento, M., Pace, G., Schneider, G.: A specification language for static and runtime verification of data and control properties. In: FM'15. LNCS, vol. 9109, pp. 108–125. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_8

2. Ahrendt, W., Chimento, M., Pace, G., Schneider, G.: Verifying Data- and Control-Oriented Properties Combining Static and Runtime Verification: Theory and Tools. Formal Methods in System Design **51**(1), 200–265 (August 2017). https://doi.org/10.1007/s10703-017-0274-y
3. Azzopardi, S., Colombo, C., Pace, G.J.: A technique for automata-based verification with residual reasoning. In: MODELSWARD 2020. pp. 237–248. SCITEPRESS (2020). https://doi.org/10.5220/0008981902370248
4. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: Contractlarva and open challenges beyond. In: RV'18. pp. 113–137. LNCS, Springer (2018)
5. Azzopardi, S., Piterman, N., Schneider, G.: Incorporating monitors in reactive synthesis without paying the price. In: ATVA'21. LNCS, vol. 12971, pp. 337–353. Springer (2021). https://doi.org/10.1007/978-3-030-88885-5_22
6. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: - runtime enforcement for reactive systems. In: TACAS'15. LNCS, vol. 9035, pp. 533–548. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_51
7. Cavezza, D.G., Alrajeh, D., György, A.: Minimal assumptions refinement for realizable specifications. In: FormaliSE'20. p. 66–76. ACM (2020). https://doi.org/10.1145/3372020.3391557
8. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: CONCUR'08. pp. 147–161. Springer (2008)
9. Chimento, J.M., Ahrendt, W., Pace, G., Schneider, G.: STARVOORS: A Tool for Combined Static and Runtime Verification of Java. In: RV'15. LNCS, vol. 9333, pp. 297–305. Springer (2015). https://doi.org/10.1007/978-3-319-23820-3_21
10. Chimento, M., Ahrendt, W., Schneider, G.: Testing Meets Static and Runtime Verification. In: FormaliSE@ICSE'18. pp. 30–39. ACM (2018). https://doi.org/10.1145/3193992.3194000
11. Colombo, C., Pace, G.J., Schneider, G.: LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In: SEFM'09. pp. 33–37. IEEE Computer Society (2009)
12. Ehlers, R., Finkbeiner, B.: Monitoring realizability. In: RV'12. pp. 427–441. Springer (2012)
13. Finkbeiner, B.: Synthesis of reactive systems. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 45, pp. 72–98. IOS Press (2016). https://doi.org/10.3233/978-1-61499-627-9-72, https://doi.org/10.3233/978-1-61499-627-9-72
14. Finkbeiner, B., Klein, F., Metzger, N.: Live synthesis. In: ATVA'21. LNCS, vol. 12971, pp. 153–169. Springer (2021). https://doi.org/10.1007/978-3-030-88885-5_11
15. Havelund, K., Goldberg, A.: Verify your runs. In: VSTTE'05. LNCS, vol. 4171, pp. 374–383. Springer-Verlag (2005)
16. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Piskac, R., Dimitrova, R. (eds.) Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. EPTCS, vol. 229, pp. 112–132 (2016). https://doi.org/10.4204/EPTCS.229.10, https://doi.org/10.4204/EPTCS.229.10
17. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: CAV'99. p. 172–183. Springer (1999)
18. Leucker, M., Schallhart, C.: A brief account of runtime verification. J.of Logic Alg. Prog. **78**(5), 293–303 (2009)

19. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020)
20. Majumdar, R., Piterman, N., Schmuck, A.K.: Environmentally-friendly gr(1) synthesis. In: TACAS'19. pp. 229–246. Springer (2019)
21. Mallozzi, P.: Designing Trustworthy Autonomous Systems. Ph.D. thesis, Chalmers, Sweden (2020)
22. Maoz, S., Ringert, J.O.: On the software engineering challenges of applying reactive synthesis to robotics. In: RoSE'18. pp. 17–22 (2018)
23. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. Softw. Syst. Model. **20**(5), 1553–1586 (2021). https://doi.org/10.1007/s10270-021-00868-z
24. Nahabedian, L., Braberman, V.A., D'Ippolito, N., Honiden, S., Kramer, J., Tei, K., Uchitel, S.: Dynamic update of discrete event controllers. IEEE Trans. Software Eng. **46**(11), 1220–1240 (2020). https://doi.org/10.1109/TSE.2018.2876843, https://doi.org/10.1109/TSE.2018.2876843
25. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 364–380. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
26. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL'89. pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
27. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstić, S., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A Survey of Challenges for Runtime Verification from Advanced Application Domains (Beyond Software). Formal Methods in System Design pp. 1–57 (August 2019). https://doi.org/10.1007/s10703-019-00337-w
28. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3), 733–749 (jul 1985). https://doi.org/10.1145/3828.3837, https://doi.org/10.1145/3828.3837
29. Ulus, D., Belta, C.: Reactive control meets runtime verification: A case study of navigation. In: RV'19. p. 368–374. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_21
30. Wong, K.W., Ehlers, R., Kress-Gazit, H.: Correct high-level robot behavior in environments with unexpected events. In: Robotics: Science and Systems X (2014). https://doi.org/10.15607/RSS.2014.X.012
31. Yoon, H., Chou, Y., Chen, X., Frew, E.W., Sankaranarayanan, S.: Predictive runtime monitoring for linear stochastic systems and applications to geofence enforcement for uavs. In: RV'19. LNCS, vol. 11757, pp. 349–367. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_20
32. Zudaire, S., Gorostiaga, F., Sánchez, C., Schneider, G., Uchitel, S.: Assumption monitoring using runtime verification for uav temporal task plan executions. In: ICRA'21. pp. 6824–6830. IEEE (2021). https://doi.org/10.1109/ICRA48506.2021.9561671