Live Synthesis[‡][*]

Bernd Finkbeiner, Felix Klein, and Niklas Metzger

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{finkbeiner,felix.klein,niklas.metzger}@cispa.de

Abstract. Synthesis automatically constructs an implementation that satisfies a given logical specification. In this paper, we study the *live synthesis* problem, where the synthesized implementation replaces an already running system. In addition to satisfying its own specification, the synthesized implementation must guarantee a sound transition from the previous implementation. This version of the synthesis problem is highly relevant in "always-on" applications, where updates happen while the system is running. To specify the correct handover between the old and new implementation, we introduce an extension of linear-time temporal logic (LTL) called *LiveLTL*. A LiveLTL specification defines separate requirements on the two implementations and ensures that the new implementation satisfies, in addition to its own requirements, any obligations left unfinished by the old implementation. For specifications in LiveLTL, we show that the live synthesis problem can be solved within the same complexity bound as standard reactive synthesis, i.e., in 2EXPTIME. Our experiments show the necessity of live synthesis for LiveLTL specifications created from benchmarks of SYNTCOMP and robot control.

1 Introduction

The past decade has brought remarkable progress in the automatic synthesis of reactive systems from temporal specifications [13,5,17]. Traditionally, synthesis is seen as a one-off method: the generated implementation is guaranteed, by construction, to satisfy the specification. If the specification changes, the process is repeated from the start. For systems that are *always-on*, like banking systems, or controllers in power plants, this may, however, not be an option: when the requirements change, the system must be updated while it is still running, and the control must transition to the new version without disrupting the safety or functionality of the running system. While such *live updates* are a well-studied concern in operating systems research (cf. [9]), they are, somewhat surprisingly, still a novelty in formal methods.

[‡]An extended version of this paper is available at [6].

[*]This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), by the European Research Council (ERC) Grant OSARES (No. 683300), and by the German Israeli Foundation (GIF) Grant "Knowledge-based Synthesis" (No. I-1513-407./2019).

In this paper, we define a *live system* as sequence of implementations, each with a corresponding specification. The last element in the sequence is the currently executed system. Performing a live update terminates the currently active system and extends the sequence with a new implementation. The key challenge of live updates is that any obligations imposed by the specification of the terminated system that are not yet satisfied at the time of the update must be taken care of by the newly active system. This transfer of obligations is important to make the update transparent from the user's perspective. Consider, for example, an arbiter specified as the LTL formula $\square(\mathit{request} \rightarrow \diamondsuit\, \mathit{grant})$, which requires that every *request* is eventually followed by a *grant*. If the update occurs after some *request*, but before the corresponding *grant*, then the new implementation must still guarantee the occurrence of the *grant*.

The problem of *model checking* live updates is to check whether a given new implementation will result in a correct live update; the *synthesis* problem is to automatically find such an implementation. To specify the correct handover between the old and new implementation, we introduce an extension of linear-time temporal logic (LTL) called *LiveLTL*. A LiveLTL specification defines requirements on the two implementations and ensures that the new implementation satisfies, in addition to its own requirements, any obligations left unfinished by the old implementation. We consider two variants of the model checking and synthesis problems. In *finite-trace live updates*, we only require the update to be correct in a specific situation, i.e., after a specific execution of the previous implementation. In *universal updates*, we require that the update can occur at any time. We show that model checking live updates is PSPACE-complete in the initial and update specification. Synthesis is 2EXPTIME-complete in the combination of the specifications for both update variants.

We report on experience with a prototype implementation of our approach on a range of benchmarks, including examples taken from the synthesis competition and a robotic case study. In our experiments, live synthesis is used to construct live updates built on reasonable pairs of specifications. The results show the necessity of verifying live updates with the adapted semantics of LiveLTL and that every considered specification states obligations for the update.

2 Running Example – Relay Station

Consider the following setup: a satellite has been positioned in the orbit of Mars in combination with multiple base stations on the planet. The base stations take samples from the extraterrestrial environment, analyze them and submit their findings to the satellite. After the data has been sent by a station, it waits for instructions from the satellite: whether the sample must be further analysed, or whether it can be discarded and a new sample must be taken. The satellite, on the other hand, provides the stations with the corresponding instructions and collects the data of all stations for relaying it back to earth. To this end, the satellite takes care that always some data of all base stations has been collected to be present in the report for earth.

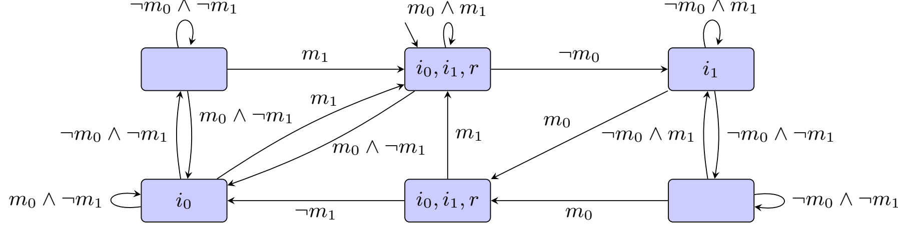

Fig. 1: Synthesized LTS for the satellite specification.

We formalize this behavior of the satellite in LTL. On the input side, the satellite receives n *measurements* m_j of every base station, where $0 \leq j < n$ ranges over the n deployed base stations on the planet. On the output side, the satellite outputs *instructions* i_j and can create a *report* r to be sent back to earth. The behavior is formalized using the following guarantees: First of all, every measurement m_i must be responded to eventually and instructions are only sent in response to received measurements.

$$\varphi_1 := \bigwedge_{j=0}^{n} m_j \rightarrow \bigcirc\Diamond i_j \qquad \varphi_2 := \bigwedge_{j=0}^{n} \Box\neg m_j \rightarrow \Diamond\Box\neg i_j$$

Furthermore, a report is generated as long as every base station submits a measurement regularly, while no report needs to be generated as long as some measurements are still missing.

$$\varphi_3 := (\bigwedge_{j=0}^{n} \Diamond m_j) \rightarrow \Diamond r \qquad \varphi_4 := (\bigvee_{j=0}^{n} \Box\neg m_j) \rightarrow \Diamond\Box\neg r$$

All guarantees φ_j must be satisfied at every point in time. We obtain the overall specification $\varphi := \bigwedge_{j=1}^{4} \Box\varphi_j$. The specification is realizable, as witnessed by the synthesized labeled transition system (LTS) for two base stations in Figure 1. We follow the transition system for $\Box\varphi_1$. Starting in the initial state, if m_0 and m_1 is received, we stay in the same state and $\bigcirc\Diamond m_0$ as well as $\bigcirc\Diamond m_1$ is satisfied. The transition system follows the $\neg m_0$ edge to the state labeled with i_1 to satisfy the subformula $\bigcirc\Diamond i_1$. Note that $m_1 \rightarrow \Diamond i_1$ would be directly satisfied in the initial state since the Moore semantics evaluates the formula based on the current state and next edge label. The states at the top right and bottom left ensure that φ_2 is satisfied, i.e., it waits for inputs before the corresponding output is set. Corresponding to φ_4, the top left and bottom right states control the output r which is only allowed to be true as long as all measurements are received. Consider a situation, where one of the base stations fails. The satellite controller must be updated, since the satellite would wait indefinitely for the data of the broken base station otherwise. The report generation would also be broken. However, we cannot just eliminate the broken base station from the original specification, synthesize again and restart the satellite with the new result. The reason is that there still may be an outstanding instruction of the satellite for one of the remaining base stations, for which this base station is

actively waiting. Therefore, the updated specification still needs to take this obligation of the old implementation into account.

We consider the necessary changes to the synthesis procedure that are required for a correct update of the specification and synthesized implementation. An adapted verification framework is introduced that enables the validation of live systems. We present a logic that avoids the break of the base stations and satellite due to the disregarded obligations of the old system during update.

3 Preliminaries

Linear Temporal Logic. Linear temporal logic (LTL) [19] is a logic for specifying correctness of linear-time systems. The syntax is a combination of state and path operators over a set of atomic propositions (AP) that define behavior over infinite time. Formulas in LTL are built according to the grammar $\varphi ::= \top \mid \bot \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$ where $a \in AP$. Temporal operators are *next* $\bigcirc$ and *until* $\mathcal{U}$, all other operators are boolean connectives. We assume every LTL formula to be in release positive normal form (PNF) where negations are only allowed in front of atomic propositions. For readability, implication $\rightarrow$ and equivalence $\leftrightarrow$ as well as the common abbreviations *eventually* $\diamondsuit a$ for $\top \mathcal{U} a$ and *globally* $\square a$ for $\neg\diamondsuit\neg a$ are used throughout this paper. Defining the LTL semantics, the operator $\vDash$ evaluates infinite traces σ and explicit index i against LTL formulas φ where traces are words over letters $\sigma \in (2^{AP})^\omega$. For example, σ satisfies $\bigcirc a$ if in the next step a holds in σ and $a \mathcal{U} b$ if a holds until b holds. A trace $\sigma = A_0 A_1 A_2 \ldots$ with $A_i \in 2^{AP}$ is an infinite sequence of sets of atomic propositions. We use the infix notation $\sigma[n, m]$ to crop the trace to the sub-trace from position n to m, $\sigma[n, m] = A_n A_{n+1} \ldots A_{m-1}$, where $A_i \in 2^{AP}$, and concatenate the finite trace σ_1 with the possibly infinite trace σ_2 with $\sigma_1 \cdot \sigma_2$. The semantic operator $\vDash$ builds a language of a specification φ with $\text{Words}(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma, 0 \vDash \varphi\}$. A trace σ that is terminated at an arbitrary position m, i.e., $\sigma[0, m]$, is a finite trace and denoted by η. The function $\mathit{expand} : LTL \rightarrow LTL$ uses the standard LTL expansion rules to unroll the given formula, expand_n repeats expand n times. For example, $\mathit{expand}_1(aUb) = b \vee (a \wedge \bigcirc(a \mathcal{U} b))$. The function $\mathit{after} : LTL \times 2^{AP} \rightarrow LTL$ evaluates the formula on a given atomic proposition assignment and returns the remaining formula, e.g. $\mathit{after}(a \mathcal{U} b, \{a\}) = a \mathcal{U} b$ and $\mathit{after}(a \mathcal{U} b, \{b\}) = \top$. $\mathit{after}(\varphi, \sigma[0, n])$ is defined as $\mathit{after}(\mathit{after}(\varphi, \sigma_0), \sigma[1, n])$ with $\mathit{after}(\varphi, \epsilon) = \varphi$. Explicit definitions of *expand* and *after* can be found in [6].

Transition Systems. The reactive model for LTL are transition systems where state labels correspond to the output of systems and transition labels correspond to the input of the environment. Given a finite set of directions Υ and a finite set of labels Σ, a Σ-labeled Υ-transition system is a tuple $TS = (T, t_0, \tau, o)$, consisting of a finite set of states T, an initial state $t_0 \in T$, a transition function $\tau : T \times \Upsilon \rightarrow T$, and a labeling function $o : T \rightarrow \Sigma$. Given AP and partition $AP = O \cup I$ for output and input atomic propositions, implementations for LTL specifications are 2^O-labeled 2^I-transition systems (TS). The paths of a

transition system start in t_0 and follow the transition function τ collecting input and output labels with the output function o. The traces of a transition system $\mathit{Traces}(TS)$ omit the state information of paths. We assume transition systems without terminal states and a deterministic transition function.
Model Checking and Synthesis. Model checking a transition system TS against a specification φ checks the relation $\mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi)$. The problem of automatically constructing a transition system that satisfies the model checking property is referred to as *synthesis*. In the course of this paper, we refer to the algorithms of LTL model checking and synthesis as black box algorithms. Similar to $\mathit{Traces}(TS)$, we denote the set of finite traces of TS by $\mathit{FinTraces}(TS)$.

4 Live Updates

Common formalisms for verification agree on the following assumption: different system versions are analyzed in isolation, i.e., everything that happened before the initial state of the new implementation is irrelevant for its correctness. For updates at runtime, this assumption is infeasible. The update system has to satisfy *obligations* that were stated during the execution of the previous system to be correct. In this section, we set the foundations for a specification language that is able to express correctness of a live update by defining the structure of two live update problems. We identify the factors affecting the update process and formalize the interplay of the components. The definitions are independent of specific temporal logics and can be adapted to various logics and system models.

Proving the correctness of systems either by model checking or synthesis assumes the existence of a starting point that is handled as the initial state. For live updates, the starting point of verification is not the initial state of the update system, but the initial state of the system running beforehand. Running systems create obligations that cannot be discarded when updated live, otherwise, for example, an observer would starve waiting for its response. The recent development of live systems enforces the sensibility of correctness algorithms to validate systems w.r.t. the *context* they are started in. For linear-time systems given as transition systems, we define the context as the finite execution of the previous system combined with its specification. The finite execution implicitly changes the state of the formula which we refer to as *active* formula. We capture this change to the formula with a function Ψ, which, given a finite trace and a specification, returns a specification that captures the obligations needed for the satisfaction of the update system. With defining Ψ, one is able to vary the impact of the initial system to the update system. Verifying an update system with standard LTL, one implicitly defines Ψ to be $\top$ for every input, enforcing no obligations on the update system.

Definition 1 (Finite Trace Live Update). *Let TS_I be an initial system, TS_U be an update system, φ be an initial specification, ψ be an update specification, and η be a finite trace of TS_I. TS_U is considered correct if it is correct w.r.t. ψ and the result of $\Psi(\eta, \varphi)$ for the function $\Psi : (2^{AP})^* \times LTL \rightarrow LTL$ defining the obligation.*

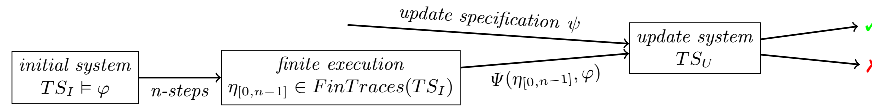

Fig. 2: The finite trace live update with φ as the initial specification, ψ as the update specification, and Ψ as the function computing the obligation for TS_U.

The finite trace live update handles the context of the update as white-box: the finite execution of the previous system is fully known. For this explicit execution, the obligation is computed and, together with the specification of the update system, verified against the update. Figure 2 shows the dependencies built by the finite trace live update where n is the number of discrete time-steps of the finite execution. However, the explicit finite execution of the initial system is not always available. Therefore, Definition 2 introduces update correctness for all possible finite paths of the initial system.

Definition 2 (Universal Live Update). *Let TS_I be an initial system, TS_U be an update system, φ be an initial specification, and ψ be an update specification. TS_U is considered correct if it is correct w.r.t. ψ and $\Psi(\eta, \varphi)$ for all possible finite traces η of TS_I.*

The context of the update is handled as black-box in the universal case. The explicit execution and the system's state of the update is unknown. Nevertheless, if all possible obligations are satisfied by the update system, the update is guaranteed to be correct. Definition 2 increases the number of possibilities to be verified, since arguing over an infinite set of finite traces cannot be performed directly. In comparison to the explicit live update, the length n is kept arbitrary since every finite trace may enforce its particular obligation.

Since we consider reactive systems, it is natural to aim for an update system that reacts to the update context, i.e., for each result of $\Psi(\eta, \varphi)$ the update system starts differently. This problem is covered by the finite trace live update if the number of different contexts is finite. One can solve the update problem for each context and combine the resulting update systems accordingly. In general, multiple other meaningful models of update correctness can be designed, e.g., enforcing the existence of an update point in the initial system's future. Nevertheless, finite trace and universal live updates suffice for the course of this paper and build a justifiable framework for live updates.

5 A Temporal Language for Live Updates

With the two live update problems defined, we introduce LiveLTL to state and verify the correctness of live updates. LiveLTL is an extension to LTL and specifies live update properties that automatically enforce the obligations of the previous execution on the update system. The syntax and semantics of LiveLTL as well as the language equivalence to LTL are shown. Moreover, we identify the class of obligations that can be stated by LiveLTL specifications.

5.1 LiveLTL

LiveLTL is designed according to three aspects: (1) the initial system is not able to enforce new obligations after termination, (2) all obligations stated before termination are satisfied by the update system, and (3) obligations are satisfiable in finite time. This guideline is a trade-off between independence of the previous system and incurring obligations from the initial specification to the update system. The definition of LiveLTL follows the finite trace update structure and builds the language for inputs as a combination of a finite and an infinite trace evaluation. The syntax is taken from LTL and we assume the set of atomic proposition for the initial system to be a subset of the atomic propositions of the update system. As extension to the semantic operator $\vDash$ of LTL, the operators $\vDash_{|\eta|,\mathcal{I}}$ and $\vDash_{|\eta|,\mathcal{U}}$ form the language for the initial system and the update system respectively. $\vDash_{|\eta|,\mathcal{U}}$ performs an index shift from time-step 0 to the update position and evaluates the changed formula with the LTL operator and is defined as $\sigma, i \vDash_{|\eta|,\mathcal{U}} \varphi$ iff $\sigma, i + |\eta| \vDash \varphi$. Since the update specification is only relevant for the update system, the shift of size $|\eta|$ enables the correct evaluation of the update system's part of the trace. $\vDash_{|\eta|,\mathcal{I}}$ inserts $|\eta|$ as upper bound for recurrent formulas, i.e., formulas with the *release* operator:

$$
\begin{array}{ll}
\sigma, i \vDash_{|\eta|,\mathcal{I}} \top & \sigma, i \nvDash_{|\eta|,\mathcal{I}} \bot \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} a & \text{iff } A_i \vDash a, \text{i.e. } a \in A_i \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} \neg a & \text{iff } A_i \nvDash a, \text{i.e. } a \notin A_i \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_1 \wedge \varphi_2 & \text{iff } \sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_1 \text{ and } \sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_2 \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_1 \vee \varphi_2 & \text{iff } \sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_1 \text{ or } \sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_2 \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} \bigcirc \varphi & \text{iff } \sigma, i+1 \vDash_{|\eta|,\mathcal{I}} \varphi \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff } \exists j, j \geq i.\ \sigma, j \vDash_{|\eta|,\mathcal{I}} \varphi_2 \text{ and } \forall k, i \leq k < j.\ \sigma, k \vDash_{|\eta|,\mathcal{I}} \varphi_1 \\
\sigma, i \vDash_{|\eta|,\mathcal{I}} \varphi_1 \,\mathcal{R}\, \varphi_2 & \text{iff } \forall j, |\eta| > j \geq i.\ \sigma, j \vDash_{|\eta|,\mathcal{I}} \varphi_2 \text{ or} \\
 & \quad \exists k, |\eta| > k \geq i.\ (\sigma, k \vDash_{|\eta|,\mathcal{I}} \varphi_1 \wedge \forall l, i \leq l \leq k.\ \sigma, l \vDash_{|\eta|,\mathcal{I}} \varphi_2)
\end{array}
$$

Informally, $\varphi_1 \,\mathcal{R}\, \varphi_2$ opens the *obligation* φ_2 in every execution step which contradicts (1) if evaluated after the update. As standard LTL semantics enables the specification to infinitely open new obligations, $\vDash_{|\eta|,\mathcal{I}}$ is built to limit this behavior to the actual finite execution of the initial system. The definition of $\vDash_{|\eta|,\mathcal{I}}$ mostly follows the definition of $\vDash$, except for the evaluation of *release* formulas. For all indices greater or equal to the length of the trace, $\varphi_1 \,\mathcal{R}\, \varphi_2$ is immediately satisfied, thus imposing the end of newly created obligations from the initial implementation. Therefore, the initial operator permits the transfer of finitely satisfiable obligations to the update system (2), but forbids the impact of the initial system after its termination (1). Note that for LTL formulas in PNF, all operators except *release* only specify finite behavior and all open obligations are satisfiable in finite time (3). The two operators define the language of LiveLTL.

Definition 3 (Language of LiveLTL). *Let φ, ψ be LTL formulas and let $\eta \in (2^{AP})^*$. The linear time property induced by φ, ψ, and η is*

$$Words(\varphi, \psi, \eta) = \{\eta \cdot \sigma \in (2^{AP})^\omega \mid \eta \cdot \sigma, 0 \vDash_{|\eta|,\mathcal{I}} \varphi \ \wedge \eta \cdot \sigma, 0 \vDash_{|\eta|,\mathcal{U}} \psi\}.$$

The language is dependent on the initial specification, the update specification, and the finite trace. Evaluating the inclusion of an infinite trace with the first $|\eta|$ elements being fixed consists of a combination of the operators $\vDash_{|\eta|,\mathcal{I}}$ and $\vDash_{|\eta|,\mathcal{U}}$. The initial LiveLTL operator is defined on the syntactic structure of the initial formula and is insensitive with respect to syntactic tautologies. Providing formulas without syntactic ambiguity that cannot be dissolved in $|\eta|$ time steps is left to the specifier. The following theorem relates LiveLTL and LTL.

Theorem 1. *LiveLTL and LTL are equally expressive.*

The proof is a reduction via encoding the initial trace into the LTL formula and is presented in the full version [6]. While being equally expressive, LiveLTL enables the direct evaluation of the newly introduced live update problems on a given context. Correctness for finite trace live updates follows from standard language inclusion.

Definition 4 (Finite Trace LiveLTL Update). *Let TS_U be an update system, φ be an initial specificaiton, ψ be an update specification, and η be a finite trace. TS_U is correct w.r.t. finite trace LiveLTL if $\eta \cdot \mathit{Traces}(TS_U) \subseteq \mathit{Words}(\varphi, \psi, \eta)$.*

Example 1. Interpreting the running example as finite trace LiveLTL update, we can obtain the finite trace $\eta = \{m_1, i_0, i_1, r\}, \{i_1\}, \{m_0, m_1\}$ as execution of the relay station. Evaluating η with $\vDash_{|\eta|,\mathcal{I}}$ shows that $\Diamond\, i_0$, $\Diamond\, i_1$, and $\Diamond\, r$ need to be satisfied by the update system, since both measurements are unanswered and no report was given after both base stations sent their measurements. Note that changing the last trace element to $\{m_0\}$ eliminates the obligations for the base station i_1 and the report r.

The finite trace update directly translates to the definition of LiveLTL, whereas the universal live update adds a level of quantification.

Definition 5 (Universal Live LTL Update). *Let TS_I be an initial system, TS_U be an update system, φ be an initial specification, and ψ be an update specification. TS_U is correct w.r.t. universal LiveLTL if*

$$\forall \eta \in \mathit{FinTraces}(TS_I) : \eta \cdot \mathit{Traces}(TS_U) \subseteq \bigcup_{\eta \in \mathit{FinTraces}(TS_I)} \mathit{Words}(\varphi, \psi, \eta).$$

To satisfy the universal update condition, the update system needs to be robust against every possible obligation of the initial system. We explore the model checking and synthesis problems of LiveLTL in Section 6.

5.2 Obligations

The impact of the initial system on the update system is declared by the operator $\vDash_{|\eta|,\mathcal{I}}$ and forms a class of temporal properties. We investigate this class and build a monitor that traces the open obligations during the execution of a system. In practice, the explicit update to be performed is unknown during the design of the

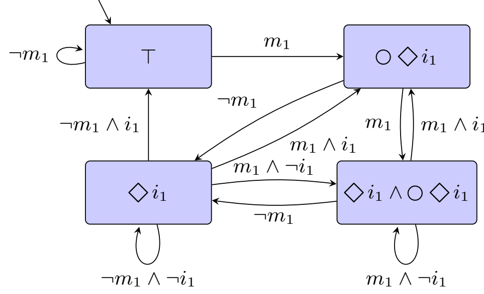

Fig. 3: The obligation monitor for φ_1 with one base station.

initial system. Therefore, one approach to face live updates is keeping track of *open* obligations while the system is executed. To obtain the expressivity of the obligations possibly enforced by LiveLTL, we introduce the *obligation property*.

Definition 6 (Obligation Property). *A linear time property P_{obl} over AP is called an obligation property if for all words $\sigma \in P_{obl}$ there exists a good prefix, i.e., for every $\sigma \in P_{obl}$ there exists a word $\sigma[0,m]$ s.t. $\forall x.x \in (2^{AP})^\omega : \sigma[0,m] \cdot x \in P_{obl}$. Obligation properties coincide with the class of co-safety properties.*

Obligations and co-safety properties describing the same language is a natural outcome of the LiveLTL semantics. To obtain the open obligations with constant cost during runtime, the construction of a monitor tracking the obligations provides a space bounded solution. The monitor is meant to be constructed simultaneously to the initial system.

Definition 7 (Obligation Monitor). *Let $strip : LTL \rightarrow LTL$ be a function syntactically substituting every $\mathcal{R}$ by $\top$. A deterministic obligation monitor for an LTL formula φ is the tuple $\mathcal{OM}_\varphi = (T, t_0, \Upsilon, after, o)$, where $T = \{\varphi' \mid \omega \in (2^{AP})^* : \varphi' = after(\varphi, \omega)\}$ is the set of states, $t_0 = strip(\varphi)$ is the initial state, $\Upsilon = 2^{AP}$ is the set of directions, after is the transition function defined over T and Υ, and $o(t) = strip(t)$ is the labeling function.*

Since the state space of $\mathcal{OM}_\varphi$ corresponds to the state exploration of φ, converting the formulas to obligations is achieved by *strip* and stored in the labeling function. This can be interpreted as the obligations that have to be satisfied by the update system if an update is initiated in this state. The obligation monitor only tracks states and does not guarantee that every reachable state corresponds to a reachable state of a correct implementation of φ. We justify this property by assuming TS_I is correct.

Example 2. Figure 3 displays the obligation monitor for $\varphi_1 = \Box(m_1 \rightarrow \bigcirc\Diamond i_1)$ of our running example with one base station. The monitor starts in an obligation free state corresponding to the state before the system is started and contains one direction for every element of 2^{AP}. Note that we denote directions symbolically. Whenever m_1 is received on an edge, the obligation $\bigcirc\Diamond i_1$ is raised. From the

$\bigcirc\Diamond i_1$ state, we differentiate between m_1 and $\neg m_1$ leading to another raise of the $\bigcirc\Diamond i_1$ obligation together with $\Diamond i_1$ or only $\Diamond i_1$ respectively. Returning to the obligation $\top$ is only possible if i_1 is set to $\top$ and m_1 is $\bot$ in the same step.

Note that an offset between initial system and obligation monitor is created. While transitions of the initial system consider environment inputs and states correspond to system outputs, elements of the state space of the obligation monitor are formulas and the transitions are defined by inputs and outputs combined. Residing in a state in the obligation monitor can be interpreted as taking a transition in the system and not yet reaching the next state. Figure 3 shows a monitor for a specification, where the implementation is unknown during construction and the obligation monitor over-approximates the reachable states of the implementation. One can limit the reachable states of the monitor to the paths in the transition system. Indeed, in regard of completeness, unreachable obligations need to be eliminated from the obligation monitor during verification.

6 Model Checking and Synthesis

In this section we solve the problems of model checking live updates and synthesis of live updates, i.e., live synthesis. We explore finite trace and universal updates for the problems and show the complexity of each result and multiple parameters.

6.1 Model Checking Live Updates

Model checking a transition system TS against an LTL formula φ corresponds to answering the question if TS satisfies φ, i.e., $TS \vDash \varphi$. For live systems, the evaluation of the update transition system starts with the initial finite execution and switches to the update system afterwards. Model checking the update system is therefore a language inclusion check of the traces of the transition system combined with η against the LiveLTL semantics.

Definition 8 (Model Checking Finite Trace Live Updates). *Let TS_U be an update system, φ be an initial specification, ψ be an update specification, and η be a finite trace. The problem of model checking finite trace live updates is defined as $\eta \cdot Traces(TS_U) \subseteq Words(\varphi, \psi, \eta)$.*

The model checking problem can be split into two separate parts, directly identifying the newly introduced conditions for live systems with the operators $\vDash_{|\eta|,\mathcal{I}}$ and $\vDash_{|\eta|,\mathcal{U}}$. In addition to that, TS_U combined with η needs to satisfy the update semantics of LiveLTL. Since both tasks can possibly be performed in isolation of each other, the overhead given by the live update semantics under the assumption of an update system already verified with LTL is an interesting topic but left open for future work. The complexity of the problem is stated w.r.t. the length of the trace and the combination of initial and update formula:

Theorem 2 (Complexity in φ, ψ, and η). *The model checking problem for finite trace live updates is* PSPACE*-complete in $|\varphi| + |\psi|$ and in* NL *in $\eta \cdot TS_U$.*

The proof is based on model checking the combination of η and TS_U and can be found in [6]. The universal live update is verified independently of specific initial traces. The condition is stronger than for finite trace updates, and the number of compatible initial and update systems is smaller. Given that the context is unknown, the executions starting in the initial state of TS_U need to satisfy every possible open obligation. Universal updates are relevant if neither the trace nor the obligation monitor are stored and computed respectively. Given the initial system, model checking universal update compatibility obtains the same complexity as finite trace updates.

Definition 9 (Model Checking Universal Live Updates). *Let TS_I be an initial system, TS_U be an update system, φ be an initial specification, and ψ be an update specification. The problem of model checking universal live updates is defined as $\forall \eta \in \mathit{FinTraces}(TS_I) : \eta \cdot \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi, \psi, \eta)$.*

The implicit update points in TS_I allow for the connection of both transition systems and model checking with a linearly increased formula.

Theorem 3 (Complexity in $\varphi + \psi$, and $TS_I \cdot TS_U$). *The model checking problem for universal live updates is* PSPACE*-complete in $|\varphi| + |\psi|$ and* NL *in $TS_I \cdot TS_U$.*

The complexity results from encoding the live update in the combined transition system $TS_I \cdot TS_U$ and an adapted formula. Based on the model checking results we introduce live synthesis, the major contribution of this paper.

6.2 Live Synthesis

In this section, we introduce the problem of live synthesis and show the complexity of synthezising live systems. Synthesis of live updates during the runtime of the initial system promises correct-by-definition updates that can substitute the executed system instantaneously. In contrast to model checking, the synthesis procedure returns an implementation or *unrealizable*, proving that the finite trace or initial system and initial specification are incompatible with the update specification. We begin with live updates for an explicit finite trace of the initial system – the update system needs to react to the explicit context and open obligation. The definition follows the model checking problem, but searches for a transition system satisfying the live update.

Definition 10 (Finite Trace Live Synthesis). *Let φ be an initial specification, ψ be an update specification, and η be a finite trace. The finite trace live synthesis problem is the computation of a transition system TS s.t. $\eta \cdot \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi, \psi, \eta)$.*

We additionally call a live update *realizable* if there exists a transition system that satisfies the finite trace live update. The complexity of the update synthesis is expressed w.r.t. φ and ψ and aligns to existing LTL synthesis bounds.

Theorem 4 (Complexity in φ and ψ). *The finite trace live synthesis problem is* 2EXPTIME*-complete in $|\varphi|$ and $|\psi|$.*

The proof is subsumed by the proof of Theorem 5. The universal update is again of interest if the context of the live update is unknown. Synthesizing a transition system that satisfies the universal live update enables the user to plug-in the new system at any time-step without further analysis.

Definition 11 (Universal Live Synthesis). *Let φ be an initial specification, TS_I be an initial system, and ψ be an update specification. The universal live synthesis problem is the computation of a transition system TS s.t. $\forall \eta \in \mathit{FinTraces}(TS_I) : \eta \cdot \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi, \psi, \eta)$.*

Again, we call the problem of the existence of a solution realizability. In general, the universal update obtains a conjunction of double exponentially many conjuncted obligations. To avoid the expansion of the update system, we combine the parity games of the initial and update system. Again, the initial formula conducts the impact on the update system and provides the complexity results.

Theorem 5 (Complexity in φ and ψ). *The universal update synthesis problem is* 2EXPTIME-*complete in $|\varphi|$ and $|\psi|$.*

Hardness follows from Theorem 1, the completeness proof is a reduction from LiveLTL to LTL. Therefore, a parity game is built for the combination of the formulas, where the environment controls all edges before the update, thereby choosing the update context. The full proof is given in [6].

7 Case Study

We explore the live update problems on benchmarks from the reactive synthesis competition [13] and robot control communities [16]. Our goal is a qualitative analysis of pairs of specifications that can be updated live according to the finite trace live update and the universal live update. In more detail, we aim to answer the following questions: For specifications that can potentially be updated to each other, does the LiveLTL semantics state universally updatable obligations? And if not, in how many obligation states is a finite trace update possible?

A prototype for the live synthesis procedure is implemented on top of BoSy [5], a tool that synthesizes implementations for LTL formulas[1]. We use Spot [3] for LTL formula manipulation and implemented the obligation monitor construction for arbitrary LTL formulas. For our experiments, the following structure is used: BoSy synthesizes a system for the initial specification which is used to build the obligation monitor. Therefore, the result of the synthesis query, i.e., a transition system satisfying the formula, is parsed and cut with the obligation monitor to eliminate unreachable states. Since the result of BoSy may differ per execution, we may obtain different sizes of the obligation monitor for different benchmark runs. Based on the obligation monitor, we perform explicit trace live synthesis for every monitor state label and universal live synthesis for

[1] The prototype and experiments are available online at https://github.com/reactive-systems/LiveSynthesisArtifact.

all monitor states combined. Therefore, we build the conjunction of obligation formula and update formula and execute BoSy to check realizability.

For the benchmarks in Section 7.1, Table 1 shows multiple results: The number of obligation monitor states built by the initial system and specification, the number of finite trace updates that are realizable, and the result of the universal update. Despite the finite trace live update stating updates from every possible finite execution of the initial system, we use the state representation of the obligation monitor to symbolically represent every execution. The runtime in seconds for the update specification without update constraints and the universal update conclude the table. All experiments were executed on an Intel i7 processor with 2,8 GHz and 16 GB RAM.

7.1 Benchmark Families

The upper part of Table 1 shows the results for live updates from specification patterns introduced by Menghi et. al. [16], where **Reactivity** implements additional interaction with the environment. The specifications define the behavior of a robot that is able to travel between n different locations and needs to satisfy different specifications on the way. Our second set of benchmarks is taken from the annual synthesis competition SYNTCOMP [13]. The results for live updates in the reactive synthesis setting are shown in the lower part of Table 1.

- **Visit**, **Seq. Visit**, and **Patrolling** enforce the robot to visit every location once, in a sequence, and infinitely often respectively.
- **Reactivity**. The reactivity specification forces the robot to react to an event after two steps at latest by driving to a delineated location, e.g., for refueling. The Reactivity specification can be added to arbitrary specifications.
- **Relay Station**. The running example of this paper. The relay station communicates with n satellites and forwards the message if clients acknowledged.
- **Arbiter**. An arbiter controls the access of multiple clients to a shared resource. It ensures that every request to the resource is eventually granted. We consider three variants of arbiter, a simple arbiter (**s**) only iterating over grants, a full arbiter (**f**) only granting access if requested beforehand, and a prioritized arbiter (**p**) that prioritizes the requests of client 0.
- **ABP**. The alternating bit protocol consists of a receiver **ABPReceiver** and a transmitter **ABPTransmitter** specifying the data link layer in the OSI communication network.
- **Load Balancer**. The load balancer distributes workload over n worker.

In addition to the specifications, we denote updates with an increased parameter with $n \to n+1$. This property is of interest if the parameter may change during the execution, e.g., increasing the number of clients of an arbiter.

7.2 Observations

Throughout all experiments, the minor runtime overhead of the universal update synthesis shows that the additional cost for live update correctness is feasible.

Robot Specification Patterns						
Ben.	*Update*	*#OM-States*	*#Fin. Trace*	*Universal*	*Time ψ*	*Time Univ.*
Visit	Seq. Visit	4	4	*real.*	0.75	0.75
	Patrolling	6	6	*real.*	0.68	0.68
	Seq. Patrolling	6	6	*real.*	0.64	0.72
	Reactivity	7	7	*real.*	0.49	0.49
Seq. Visit	Patrolling	14	14	*real.*	0.56	0.59
	Seq. Patrolling	16	16	*real.*	0.57	0.59
	Reactivity	5	5	*real.*	0.44	0.44
Patrolling	Ord. Visit	6	6	*real.*	0.61	0.67
	Reactivity	7	7	*real.*	0.49	0.52
SYNTCOMP						
Relay Station	$1 \to 2$	4	4	*real.*	16.26	17.23
	$2 \to 1$	19	19	*real.*	0.61	0.61
Arbiter	2f $\to$ 3f	11	6	*unreal.*	5.30	-
	2s $\to$ 2f	4	2	*unreal.*	0.56	-
	2s $\to$ 4s	4	4	*real.*	0.69	0.79
	2s $\to$ 2p	13	13	*real.*	0.46	0.48
	2f $\to$ 2p	10	10	*real.*	0.45	0.52
	2p $\to$ 3p	6	6	*real.*	0.65	0.74
ABPReceiver	$1 \to 2$	5	4	*unreal.*	0.55	-
	$2 \to 3$	9	3	*unreal.*	0.43	-
ABPTransmitter	$1 \to 2$	5	5	*real.*	2.70	2.82
Load Balancer	$2 \to 4$	7	7	*real.*	0.72	0.75

Table 1: Results of Live Updates for Robot and SYNTCOMP specifications.

The robot specifications provide insight of obligations raised during execution. Since most of the benchmarks obtain the same structural behavior, i.e., the robot visits the locations under some restrictions, the universal live updates are realizable. Even when adding requests, e.g., the robot has to refuel in two steps after requested, the live update is realizable by satisfying the open obligations after the update. Changes to the visiting sequence or infinitely often reaching a location with patrolling increases the size of the obligation monitor (*#OM-States*) but does not lead to unrealizability. Nevertheless, the sizes of the obligation monitors indicate that tracking the behavior of the system is necessary to obtain the correct obligation. Altogether, our results show that although robot specifications raise obligations, synthesizing correct live updates is often feasible due to the absence of conflicts between the specifications. Most interestingly for the reactive systems benchmarks are arbiter live updates. Changing a specification to a simple arbiter is realizable since the arbiter does not additionally restrict the behavior. However, live updates to full arbiter are only possible from some obligation monitor states, shown by the difference of *#OM-States* and *#finite trace updates*. Unrealizability follows from obligation states forcing a grant - an unrequested grant of the update system would be spurious. Since the prioritized arbiter does not include non-spuriousness, a live update from and to this arbiter is realizable. The relay station can be universally updated to the one more and one less base stations. Once computed, the obligations can be satisfied in finite time-steps and synthesizing a solution that reacts to all obligations is possible.

The experiments answer the questions stated at the beginning of this section: Specifications that are meaningful live updates state obligations for the update system, shown by the large number of states of the obligation monitors. Realizability of the update system depends on the restrictiveness of the specification, even if the universal update is unrealizable, our results show that in all benchmarks some finite trace live updates are realizable.

8 Related Work

The necessity of live updates in always-on systems is long known and was introduced as [4,7]. Dynamic updates for programming languages, e.g., in C++ [12] and Java [10], enable developers to update *dynamic classes* during runtime and are called *dynamic software updates* (DSU). The proposed frameworks implement functionality and are unable to ensure temporal correctness of the updates. Live kernel patches received huge attention in the operating system community [1,9], where bug-fixes and features of the kernel can be deployed without reboot. Recent work in live updates for operating systems achieved real-life implementations, e.g. for Linux [14] and Android [2] kernels. Implementations of dynamic updates raised the need for verification: Following the idea of observability by the user, Hayden et. al. [11] introduce *client-oriented specifications* (CO-specs) to define and verify against client-visible behavior. Closest to our work are dynamic updates in controller verification and synthesis. Ghezzi et.al. [8] introduce a controller synthesis approach based on Modal Sequence Diagrams (MSD). The update is a synthesized MSD that takes over the execution when a safe state is reached. While reaching a safe state is also necessary in [15], the authors omit the obligations of the previous system. Where [8] also relies on the existence of a safe state for the live update, [18] also proves the reachability of the update state. Therefore, the condition of the handover between the systems is defined as LTL specification. The main difference is stating the correctness as LTL formula and not observing the update condition semantically from the initial formula.

9 Conclusion

We introduced live synthesis, a synthesis framework for dynamic updates in reactive systems. We identified *obligations* of a running system as the currently open *co-safety* formulas and defined LiveLTL to specify the correct handover between two systems. The presented obligation monitor enables tracking of obligations during system execution and continuously shows the open obligations. We explored synthesis and model-checking for two update problems, *finite trace live updates* and *universal update*, which consider full information and zero information of the currently open obligations respectively. Our case study on robot specifications and reactive synthesis benchmarks show that it is necessary to verify live updates in *always-on* systems and *live synthesis* is able to automatically generate correct update systems if realizable. We believe that live updates play a crucial role in *high-availability* system verification and can benefit from existing techniques for reactive systems.

References

1. Baumann, A., Heiser, G., Appavoo, J., Silva, D.D., Krieger, O., Wisniewski, R.W., Kerr, J.: Providing dynamic update in an operating system. In: USENIX (2005)
2. Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., Wei, T.: Adaptive android kernel live patching. In: Kirda, E., Ristenpart, T. (eds.) USENIX Security 2017
3. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and \omega -automata manipulation. In: ATVA 2016. LNCS. https://doi.org/10.1007/978-3-319-46520-3_8
4. Fabry, R.S.: How to design a system in which modules can be changed on the fly. In: ICSE 1976. IEEE Computer Society
5. Faymonville, P., Finkbeiner, B., Tentrup, L.: Bosy: An experimentation framework for bounded synthesis. In: CAV 2017. pp. 470–476. https://doi.org/10.1007/978-3-319-63390-9_17
6. Finkbeiner, B., Klein, F., Metzger, N.: Live Synthesis (Full Version) (2021), http://arxiv.org/abs/2107.01136
7. Frieder, O., Segal, M.E.: On dynamically updating a computer program: From concept to prototype. JSS 1991 . https://doi.org/10.1016/0164-1212(91)90096-O
8. Ghezzi, C., Greenyer, J., Manna, V.P.L.: Synthesizing dynamically updating controllers from changes in scenario-based specifications. In: SEAMS 2012. IEEE Computer Society. https://doi.org/10.1109/SEAMS.2012.6224401
9. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Safe and automatic live update for operating systems. SIGPLAN Not. **48**. https://doi.org/10.1145/2499368.2451147
10. Gregersen, A.R., Jørgensen, B.N.: Dynamic update of java applications - balancing change flexibility vs programming transparency. JSWM (2009). https://doi.org/10.1002/smr.406
11. Hayden, C.M., Magill, S., Hicks, M., Foster, N., Foster, J.S.: Specifying and verifying the correctness of dynamic software updates. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS. https://doi.org/10.1007/978-3-642-27705-4_22
12. Hjalmtysson, G., Gray, R.: Dynamic C++ classes - A lightweight mechanism to update code in a running program. In: USENIX 1998. USENIX Association
13. Jacobs, S., Basset, N., Bloem, R., Brenguier, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Michaud, T., Pérez, G.A., Raskin, J., Sankur, O., Tentrup, L.: SYNTCOMP 2017. In: SYNT@CAV 2017. https://doi.org/10.4204/EPTCS.260.10
14. Makris, K., Ryu, K.D.: Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In: EuroSys 2007. ACM. https://doi.org/10.1145/1272996.1273031
15. Manna, V.P.L., Greenyer, J., Ghezzi, C., Brenner, C.: Formalizing correctness criteria of dynamic updates derived from specification changes. In: SEAMS 2013. IEEE Computer Society. https://doi.org/10.1109/SEAMS.2013.6595493
16. Menghi, C., Tsigkanos, C., Berger, T., Pelliccione, P., Ghezzi, C.: Property specification patterns for robotic missions. In: ICSE 2018. ACM (2018)
17. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS. https://doi.org/10.1007/978-3-319-96145-3_31
18. Nahabedian, L., Braberman, V.A., D'Ippolito, N., Honiden, S., Kramer, J., Tei, K., Uchitel, S.: Assured and correct dynamic update of controllers. In: SEAMS@ICSE 2016. ACM. https://doi.org/10.1145/2897053.2897056
19. Pnueli, A.: The temporal logic of programs. In: SFCS 1977) (Oct). https://doi.org/10.1109/SFCS.1977.32