Expressing structural temporal properties of safety critical hierarchical systems

Massimo Benerecetti$^{1[0000-0003-4664-6061]}$, Fabio Mogavero$^{1[0000-0002-5140-5783]}$, Adriano Peron$^{1[0000-0002-7111-3171]}$, and Luigi Libero Lucio Starace$^{1\star[0000-0001-7945-9014]}$

University of Naples Federico II, Naples, Italy
{massimo.benecetti,fabio.mogavero,adrperon,luigiliberolucio.starace}@unina.it

Abstract. Software-intensive safety critical systems are becoming more and more widespread and are involved in many aspects of our daily lives. Since a failure of these systems could lead to unacceptable consequences, it is imperative to guarantee high safety standards. In practice, as a way to handle their increasing complexity, these systems are often modelled as hierarchical systems.

To date, a good deal of work has focused on the definition and analysis of hierarchical modelling languages and on their integration within model-driven development frameworks. Less work, however, has been directed towards formalisms to effectively express, in a precise and rigorous way, relevant behavioural properties of such systems (e.g.: safety requirements).

In this work, we propose a novel extension of classic Linear Temporal Logic (LTL) called Hierarchical Linear Temporal Logic (HLTL), designed to express, in a natural yet rigorous way, behavioural properties of hierarhical systems. The formalism we propose does not commit to any specific modelling language, and can be used to predicate over a large variety of hierarchical systems.

Keywords: Formal Specification · Safety-critical Software Systems · Formal Verification · Temporal Logics.

1 Introduction and related works

In today's world, computer and software systems are ubiquitous and involved in almost every aspect of daily life. From railway-traffic control systems to smartphones, from medical appliances to the stock exchange market, from power plants to communication networks, society relies on such systems to an ever-growing extent, making their reliability an issue of great social importance. Furthermore, it is increasingly rarer to find isolated computer systems, as they are typically embedded in larger contexts, interacting with several other concurrently-executing systems over wired and wireless networks. Due to this interconnection trend and to the increasing variety and complexity of the performed tasks, the complexity

* Corresponding author. Email: luigiliberolucio.starace@unina.it of computer systems is growing apace, along with the difficulty in their design, specification, implementation and verification.

Of greatest concern, in particular, are the so-called *safety-critical systems*, i.e., those systems whose failure could lead to consequences that are determined to be unacceptable. Typical examples of safety-critical systems include medical care devices, aircraft and railway traffic controllers, nuclear power plants, and so on. However, a way broader class of systems has the potential for very high consequences of failure, and these systems should be considered to be safety-critical as well. For example, it is obvious and sadly known [10] that a malfunctioning in commercial aircraft could lead to loss of lives. It is not obvious, on the other hand, that also a malfunction in a telephone exchange system could kill people. Indeed, a protracted loss of 911 service has very high potential of resulting in serious consequences [12].

When dealing with the specification and design of large and complex software-intensive reactive systems, the notion of hierarchy arises quite naturally as witnessed by the establishment of hierarchical specification languages such as Statechart [9] as a standard in the Software Engineering community. Such systems can be indeed described as collections of nested components, or modules, organized in a tree-like structures, evolving concurrently and interacting with each other in some meaningful way.

Quite a lot of work has been directed towards the definition of hierarchical models [3, 13, 7, 15], towards the study of the complexity of basic decision problems, such as reachability and model checking of classic temporal logic properties [2], and towards their integration within model-driven development frameworks [14, 5, 6]. Less work, however, has been devoted to languages to express relevant behavioural properties of such models, taking into account also their intrinsic hierarchical structure.

A notable exception is [8], which proposes a formal model for recursive concurrent programs, namely Communicating Recursive State Machines (CRMS). In that work, the authors also propose a logic called ConCaRet, extending with parallel operators the linear logic for Call and Return CaRet [1]. ConCaRet is designed to specify behavioural properties of recursive sequential systems such as CRMS, in which a computation can be seen as a sequence of (possibly unbounded) ranked trees to model recursive calls. ConCaRet introduces operators to reason about tree paths corresponding to computation treads. Temporal operators moves along a thread, intertwining temporal displacements and moves within the recursive structure.

In this paper we propose a different extension of classic Linear Temporal Logic (LTL [17]) called Hierarchical Linear Temporal Logic (HLTL). Among temporal logics, Linear-time Temporal Logic (LTL) has been widely used as a specification language to formalize behavioural properties of systems [18, 16]. LTL allows for reasoning about behaviours represented as sequences of unstructured (flat) system states. When dealing with hierarchical models, however, the ability of naturally contextualizing behavioural properties with respect to the horizontal and vertical modular structure of the specification could prove to be very useful. To this end, HLTL combines the analysis of the standard temporal dimension of LTL with two additional orthogonal dimensions that account for the hierarchical and concurrent nature that characterizes the computations of hierarchical systems. This allows HLTL to naturally express behavioural local properties of modules instead of being limited to reasoning about global system states. Unlike ConCaRet, HLTL syntactically separates temporal moves along a computation from the moves along its vertical and horizontal dimensions. The clear separation between the temporal and the structural operators in HLTL gives a clear way to precisely contextualize properties in a module of the system, as opposed to ConCaRet which lacks this specification ability.

This paper is structured as follows. In Section 2 we provide some basic preliminary notions on Dynamic State Machines (DSTM), a recently-proposed hierarchical modelling language explicitly devised to meet industrial requirements in design, verification and validation of complex, multi-process, control systems. In Section 3 we introduce the logical formalism we propose, and show how it can be applied to express properties of hierarchical systems, and in particular of DSTMs. In Section 4, we hint at possible direct applications of HLTL in different phases of the system development lifecycle. At last, in Section 5, we draw some closing remarks and discuss future research directions.

2 Dynamic State Machines and Hierarchical Computations

In this section, we provide some preliminary notions on hierarchical modelling languages and on the formalization of their behaviours by giving an overview of one of such formalisms, namely Dynamic State Machines (DSTM). For a complete account of the formal syntax and semantics of DSTM we refer to [5].

2.1 Dynamic State Machines

The Dynamic STate Machine (DSTM) formalism is a recently-developed modelling language originally proposed in [15] and explicitly devised to meet industrial requirements in design, verification and validation of complex control systems. DSTM features both complex control flow constructs such as asynchronous forks, preemptive termination, recursive execution and complex data flow constructs such as custom complex type definition, parametric machines, and inter-process communication through global channels and variables.

DSTM takes many syntactic elements from UML Statecharts, and extends them with the notion of re-usable module and with the possibility of recursion and dynamic instantiation. A machine is capable of dynamically instantiating one of its modules when the number of concurrently-executing instances of said module is decided at run-time.

In more detail, a Dynamic STate Machine (DSTM) model is a sequence of machines $\langle M_1, M_2, \dots, M_n \rangle$ communicating over a set X of global variables and a set C of global communication channels. The first of such machines M_1 is the so-called *initial machine*, i.e., the highest level of the hierarchical system. A machine M_i represents a module in the hierarchical specification and is defined as a state-transition diagram composed by vertices connected by transitions. The following kinds of vertices are defined:

node basic stable control state of a machine;
entering node initial pseudo-node of a machine. A machine may have multiple entering nodes, corresponding to different initial conditions;
initial node default entering pseudo-node of a machine, to be used when no entering node is explicitly specified. There must be exactly one for each machine;
exit node final (or exiting) node of a machine. A machine may specify multiple exiting nodes, corresponding to different termination conditions;
box node modelling the parallel activation of machines associated with the box itself. A transition entering a box represents the parallel activation of the corresponding machines, while a transition exiting a box corresponds to a return from said activation.
fork control pseudo-node modelling the activation of new processes. Such activation may be either *synchronous* (the forking process is suspended and waits for the activated processes to terminate) or *asynchronous* (the forking process continues its activity along the newly-activated processes).
join control pseudo-node used to synchronize the termination of concurrently executing processes or to force their termination when necessary (*preemptive* join).

In the description above the vertices corresponding to stable, meaningful control points are called *nodes*. On the other hand, *pseudo-nodes* represent transient points.

Transitions represent changes in the control state of a machine. A transition is labelled with a name and decorated with a *trigger* (an input event originating from the external environment or from other concurrent machines), a *guard* (a Boolean condition on the current contents of variables and channels) and an *action* (one or more statements on variables and channels). For a transition to be fired it is necessary that its trigger is fulfilled and that its guard is satisfied. When a transition fires, its action is executed with possible side-effects.

Example 1 (The Counting *DSTM).* As an example, consider the *Counting* DSTM, consisting of the machines *Main*, *Counter* and *Incrementer* represented in Figure 1. In the proposed graphical formalism, default entering pseudo-nodes are depicted as black circles, entering pseudo-nodes as white circles, final nodes as crossed-out white circles. Boxes are represented by rectangles and decorated with a comma-separated list of associated machines enclosed in square brackets. Nodes are drawn as rounded rectangles and fork and join pseudo-nodes are represented by black bars. Each node and pseudo-node is decorated with its name. Transitions are, as usual, drawn as directed edges between the source and the target vertices. The transitions for these machines are detailed in Table 1, in which we denote the trivial trigger and the trivial guard, which are always

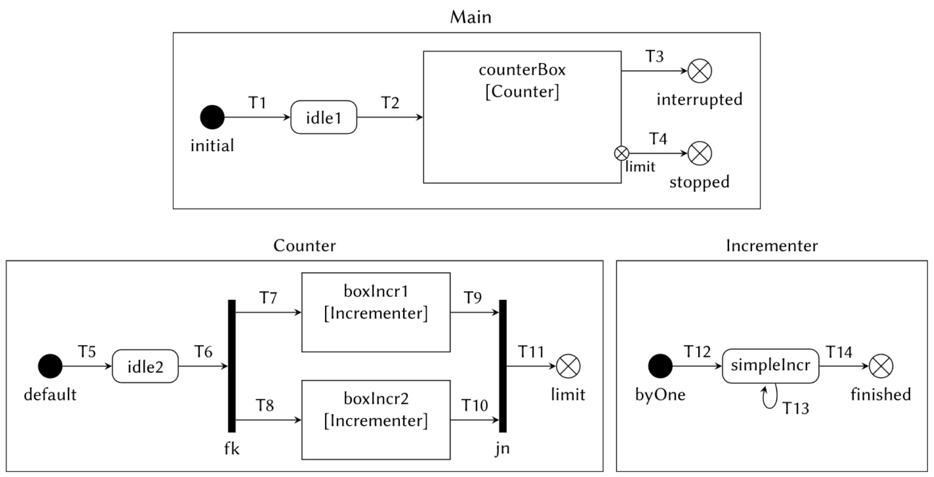

Fig. 1. The *Counting* DSTM specification

satisfied, respectively with τ and *True*. The empty list of actions, which produces no side-effects, is represented with ε. As for the semantics of the *Counting* DSTM, we describe it informally as follows. The initial machine, and highest module in the hierarchy, is the *Main* DSTM, and its initial stable state is idle1. Subsequently, by executing transition T2 (which has trivial trigger, guard, and actions, and thus is always executable and has no side effects), *Main* performs a call operation instantiating a *Counter* module by entering the *counterBox* box. After instantiation, the *Counter* machine will be in its local stable state idle2. After performing the fork operation consisting in transitions T6,T7,T8, the counter *Counter* module can instantiate two instances of the *Incrementer* module, which will run in parallel at the same level of the hierarchy. Back to the *Main* module, notice that transition T3 can be executed at anytime when the `signal?` trigger, which requires that a message is present on the communication channel named `signal`, is fulfilled. If T3 is executed, the *Counter* module (and eventually the modules the latter instantiated) will be pre-emptively terminated and deallocated. Transition T4, on the other hand, can be executed only if the *counter* module has reached its limit termination state, which can happen only once both the *Incrementer* modules have terminated and the join operation represented by transitions T9,T10,T11 is executed. As for the *Incrementer* module, its initial stable state is simpleIncr. Then, until the global variable x is incremented to 10 and transition T14 becomes executable, only transition T13 can be executed. This transition does not lead to a different local state, but is associated with an action having the side-effect of incrementing x by 1. Summarizing, the two *Incrementer* modules instantiated in parallel by Counter will, at each computation step, increment each the global variable x by 1. After 5 computation steps, x will be incremented up to 10, and the *Incrementer* modules will execute transition T4, reaching their exiting `finished` states. Subsequently,

Table 1. Transition structure for the DSTM model *Counting*

T_1	*Source*	*Target*	*Trigger*	*Guard*	*Action*
T1	initial	idle1	τ	*True*	ε
T2	idle1	counterBox	τ	*True*	ε
T3	counterBox	interrupted	`signal?`	*True*	ε
T4	(counterBox, limit)	stopped	τ	*True*	ε
T5	default	idle2	τ	*True*	ε
T6	idle2	fk	τ	*True*	ε
T7	fk	boxIncr1	τ	*True*	ε
T8	fk	boxIncr2	τ	*True*	ε
T9	boxIncr1	jn	τ	*True*	ε
T10	boxIncr2	jn	τ	*True*	ε
T11	jn	limit	τ	*True*	ε
T12	byOne	simpleIncr	τ	*True*	ε
T13	simpleIncr	simpleIncr	τ	x<10	x++
T14	simpleIncr	finished	τ	x$\geq$10	ε

the *Counter* module will perform the join operation we previously mentioned, reaching its final `limit` state, and the *Main* module will be able to terminate, reaching its `stopped` final state.

2.2 Hierarchical Computations

The evolution of a dynamic system can, in general, be seen as a sequence of stable system states. When dealing with hierarchical systems such as DSTM, each state has an intrinsic hierarchical structure, and thus can be represented using a tree-like structure reflecting the internal organization of modules. In such a tree-like representation, which we call *configuration tree*, each node represents a currently active module, and a module is a child of another if the latter instantiated the former. Hence, sibling modules execute concurrently at the same level of the hierarchy and are instantiated by the same parent module. A behaviour of a hierarchical system, i.e., a *hierarchical computation*, is thus a sequence of configuration trees, each representing a state of the entire system. Adjacent configuration trees in a hierarchical computation are the source and the target of a single computation step of the system.

Along a hierarchical computation, adjacent configuration trees are not structurally unrelated, but must reflect possible structural changes induced by computational steps of the underlying system. To capture this intuition, the concept of *frontier* is introduced. A frontier is a possibly empty subset of nodes of a configuration tree marking the modules that take part in the current computation step. All the points that are not descendant of the frontier remain unchanged during the step, since they are not performing any action. In more detail, if a point belongs to the frontier, we assume that the module instantiated in that point is performing an action. Otherwise, if an ancestor of a point belongs to the frontier, then the module instantiated in that point is deallocated. Finally, if a module is an ancestor of a point belonging to the frontier, then the current computation step affects a module internally invoked by the considered module.

As an example of hierarchical computation, consider the partial behaviour of the *Counting* DSTM depicted in Figure 2.

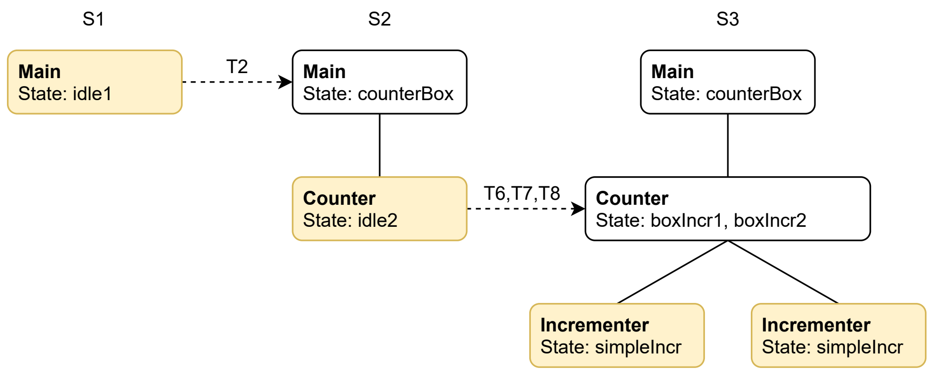

Fig. 2. A partial computation of the *Counting* DSTM

In the initial state $S1$, only the *Main* module is active, and is in its local *idle1* state. Then, after performing transition *T2*, the system reaches state $S2$, in which, in addition to the *Main* module, also a *Counter* module is active. In particular, the *Counter* module is instantiated by the *Main* one via the box *counterBox*, and hence is its child in the hierarchy. The *Counter* module will be in its *idle2* local state. Subsequently, in our example, the *Counter* module performs a fork operation instantiating two *Incrementer* modules. As a result of this operation (which corresponds to transitions T6, T7 and T8 in the DSTM model), the overall state of the systems is $S3$, in which the two *Incrementer* modules are added as children of *Counter*. In our representation, modules belonging to the frontier of each state are highlighted with a yellow background.

3 Hierarchical Linear-time Temporal Logic

Temporal logic is a widely-used formalism for describing behaviours of dynamic systems, which can be seen as a sequence of system states representing the system evolution over time. Temporal logic extends propositional or predicate logic with modalities that permit to refer to the the temporal dimension of behaviours. They provide a very intuitive but mathematically precise notation for expressing properties about the relations between states in a behaviour [4].

Among temporal logics, Linear-time Temporal Logic (LTL), originally proposed in [17], has been widely used as a specification language to formalize behavioural properties of systems [18, 16]. LTL allows for reasoning about sequences of unstructured (flat) states. However, as discussed in the previous section, hierarhical models such as DSTM represent systems whose global state in a given instant can be described by tree-like structures. As a consequence, a behaviour of such system can be represented as sequence of tree-structured states. As a consequence, LTL cannot be used to express, in a natural way, system properties that take into account the intrinsic structure of hierarchical systems.

To address this issue, we propose an extension of LTL, which we name Hierarchical Linear-time Temporal Logic (HLTL), which is designed to express properties of hierarchical computations, i.e., of sequences of tree-structured states. HLTL is able to explicitly reference the tree-like structure of each state. The main intuition behind HLTL is to use classic LTL operators to reason about the evolution of a given module, while additional operators are used to contextualize formulae in the hierarchy of activated modules. A HLTL formula is locally evaluated with regard to a context (i.e. a given module, corresponding to a vertex in the tree-like hierarchical structure of the current state), and the context can change during the evaluation of a formula by moving along both the vertical and the horizontal dimension. The vertical dimension is related to the hierarchy (caller/called relations), while the horizontal one is related to concurrency (left/right sibling in the tree).

Suppose that the current context is fixed in a given point of a configuration tree, corresponding to a module committed to a call, which is to say that that point has children (the modules it invoked) in the current state. It is possible to express the fact that the formula ϕ is required to hold in the i-th child of that module by means of the formula $\downarrow_i(\phi)$. Notice that in HLTL it is possible to navigate the vertical dimension only downwards. If the context has siblings in the current state (i.e. is executing concurrently with other modules as a result of a call operation performed by its parent), then it is possible to express the fact that the formula ϕ is required to hold in its left (resp. right) sibling with the formula $\leftarrow(\phi)$ (resp. $\rightarrow(\phi)$).

These vertical and horizontal displacement operators can be freely combined with linear temporal operators that allow for expressing behavioural properties of a module with regard to the temporal dimension.

Before going into details about the syntax and semantics of HLTL, we firstly formalize the notion of hierarchical computation, which was informally introduced in Section 2.

Definition 1 (Hierarchical computation). *A hierarchical computation over a set of atomic propositions $\mathcal{P}$ is a sequence of the form*

$$\langle(T_0, v_0), Fr_0\rangle, \langle(T_1, v_1), Fr_1\rangle, \ldots, \langle(T_i, v_i), Fr_i\rangle, \ldots$$

such that, for all $i \geq 0$:

1. *(T_i, v_i) is a labelled tree representing, with $v_i : T_i \rightarrow 2^{\mathcal{P}}$;*
2. *the frontier Fr_i is a subset of the T_i;*
3. *all the nodes in T_i that are not descendant of the nodes in the frontier Fr_i cannot be deallocated, i.e., they belong to T_{i+1};*
4. *for all the nodes $t \in T_i \setminus Fr_i$ it holds that $v_i(t) = v_{i+1}(t)$.*

In the above definition, Item 1 captures the idea that each state in a hierarchical computation has a tree structure in which each node is a currently active module. The labelling function $v_i : T_i \rightarrow 2^{\mathcal{P}}$ is used to assign, to each module in the configuration tree, a set of atomic propositions that are satisfied in that module. Item 2 requires that the set of nodes belonging to the frontier is well-formed, i.e., it is a subset of the modules that are currently active. Of course, since the frontier represents nodes that are involved in the current computation step, only currently active modules can belong to it. Items 3 and 4 capture the fact that adjacent configuration trees in a hierarchical computation are not structurally unrelated, but must reflect changes induced by the computation step the system performs. In particular, Item 3 guarantees that only nodes that are descendant of the frontier can be deallocated (i.e., can be removed in the subsequent configuration tree) as a result of a preemptive termination performed by their ancestor in the frontier. Item 4, on the other hand, requires that the labelling remains unchanged for all the modules that do not belong to the frontier, capturing the intuition that modules that do not take part in the current computation step must remain unchanged, and thus must satisfy the same atomic propositions.

When dealing with sequences of states, the standard interpretation of LTL has a *global* character, i.e. the concept of *next* state is relative to the overall system state. With flat, unstructured states, this classic interpretation is perfectly adequate, but it falls short when dealing with sequences of structured states of concurrent hierarchical modular systems, as it is not able to fully capture the concurrent nature of computations. To better capture the concurrent and hierarchical nature of these computations, HLTL considers a *local* interpretation of next, capturing a local notion of successor, which requires that the context is directly interested by a local change. As follows, we formalize the notion of local next we considered.

Definition 2 (Local interpretation of *next*). *Given an interrupting hierarchical word*

$$\xi = \xi_0, \xi_1, \ldots, \xi_k, \ldots,$$

with $\xi_i = \langle (T_i, v_i), Fr_i \rangle$, for all $i \geq 0$ and $t \in T_i$ (t is a node in the tree T_i) the local *next of a given module t in ξ_i, in symbols* $\mathrm{Next}(\xi_i, t)$*, is the hierarchical computation symbol ξ_j (if any) such that $j > i$ is the least index such that $t \in Fr_{j-1}$ and, for each $i < \ell < j$ there is no prefix t' of t with $t' \in Fr_\ell$.*

As usual, $Next^*(\xi_i, t)$, denotes the reflexive and transitive closure of $Next^*(\xi_i, t)$ and is defined inductively as the set of hierarchical symbols such that:

1. $\xi_i \in Next^*(\xi_i, t)$;
2. $\xi_j \in Next^*(\xi_i, t)$ iff $\xi_j \in Next(\xi_k, t)$ and $\xi_k \in Next^*(\xi_i, t)$.

As an example, consider the hierarchical computation $\xi = \xi_0, \xi_1, \xi_2, \xi_3, \xi_4$ in Figure 3, in which each node is represented by a circle and decorated with the atomic proposition it satisfies, and nodes belonging to the frontier are depicted with a double circle. In the figure, the local successor relation is depicted as a dashed arrow connecting a module with its local next. Consider the only module

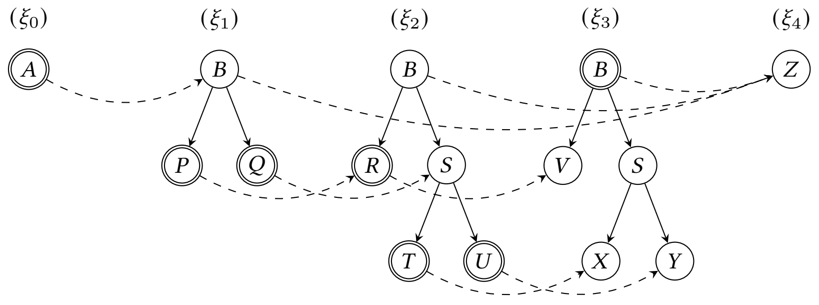

Fig. 3. An hierarchical computation decorated with the *local* next relation

in the first state ξ_0 of ξ. Its local next is the root in the subsequent state ξ_1, since 1 is the least index greater than 0 such that the root module belongs to the frontier Fr_0. When considering the root module in the state ξ_1, on the other hand, the local next is the corresponding module in state ξ_4.

With the definitions of hierarchical computation and local next in place, we can introduce the syntax and semantics of HLTL.

Definition 3 (HLTL syntax). *HLTL formulae are inductively defined as follows:*

$$\phi ::= \top \mid p \in \mathcal{P} \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \downarrow_n(\phi) \mid \leftarrow(\phi) \mid \rightarrow(\phi) \mid \mathtt{X}\,\phi \mid \phi\,\mathtt{U}\,\phi \mid \phi\,\mathtt{R}\,\phi,$$

where $\mathcal{P}$ is a set of atomic propositions.

A HLTL formula is interpreted over interrupting hierarchical computations according to the following semantics.

Definition 4 (HLTL semantics). *The satisfaction of an HLTL formula ϕ in node $t \in T_i$ at the i-th symbol of a hierarchical computation $\xi = \xi_0, \xi_1, \ldots, \xi_k, \ldots$, with $\xi_i = \langle (T_i, v_i), Fr_i \rangle$, is defined recursively as follows:*

- $\langle \xi_i, t \rangle \vDash p$ *iff* $p \in v_i(t)$, *with* $p \in \mathcal{P}$*;*
- *Boolean connectives are defined as usual;*
- $\langle \xi_i, t \rangle \vDash \downarrow_j \phi$ *iff there exists a node* $t' \in T_i$ *being the* j*-th child of* t *and* $\langle \xi_i, t' \rangle \vDash \phi$*;*
- $\langle \xi_i, t \rangle \vDash \leftarrow(\phi)$ *iff there exists a node* $t' \in T_i$ *that is the left sibling of* t *and* $\langle \xi_i, t' \rangle \vDash \phi$*;*
- $\langle \xi_i, t \rangle \vDash \rightarrow(\phi)$ *iff there exists a node* $t' \in T_i$ *that is the right sibling of* t *and and* $\langle \xi_i, t' \rangle \vDash \phi$*;*
- $\langle \xi_i, t \rangle \vDash \mathtt{X}(\phi)$*, iff there exists* ξ_j *such that* $\xi_j = \mathrm{Next}(\xi_i, t)$ *and* $\langle \xi_j, t \rangle \vDash \phi$*.*
- $\langle \xi_i, t \rangle \vDash \phi\,\mathtt{U}\,\psi$ *iff there exists* $\xi_j \in \mathrm{Next}^*$ *such that* $\langle \xi_j, t \rangle \vDash \psi$ *and, for all* $\xi_k \in \mathrm{Next}^*$*, with* $i \leq k < j$*,* $\langle \xi_k, t \rangle \vDash \phi$*.*
- $\langle \xi_i, t \rangle \vDash \phi\,\mathtt{R}\,\psi$ *iff, for all* $\xi_j \in \mathrm{Next}^*(\xi_i, t)$*, if* $\langle \xi_j, t \rangle \nvDash \psi$*, then there exists* $\xi_k \in \mathrm{Next}^*$*, with* $i \leq k < j$*, such that* $\langle \xi_k, t \rangle \vDash \phi$*.*

A hierarchical computation ξ satisfies a HLTL$^{\not{t}}$ formula ϕ, in symbols $\xi \vDash \phi$, if ϕ holds in the root of the first configuration tree.

The following abbreviations will be used hereafter: $\bot$ for $\neg\top$; $\phi \Rightarrow \psi$ for $\neg\phi \vee \psi$; Stop_x for $\neg\,\mathtt{X}\,\top$, expressing the fact that the current context has no local future. Derived temporal operators *eventually* F and *globally* G can be defined as follows. $\mathtt{F}\,\phi$, requiring that, in some future point in Next^*, ϕ holds, is equivalent to $\top\,\mathtt{U}\,\phi$. Similarly, $\mathtt{G}\,\phi$, requiring for ϕ to hold in all points in Next^*, is equivalent to $\bot\,\mathtt{R}\,\phi$.

Example 2 (HLTL formulae). In this example, we hint at the expressive power of HLTL through some examples which are then evaluated against the hierarchical computation ξ depicted in Figure 3.

1. Let ε denote the root node of each configuration tree in xi. It holds that $\langle \xi_{i\in\{1,2,3\}}, \varepsilon\rangle \vDash \mathtt{X}\,Z$, since in each of these contexts, the local next $\langle \xi_4, \varepsilon\rangle \vDash Z$. The HLTL formula $\mathtt{X}\,Z$, on the other hand, is not satisfied in the root of ξ_0, since its local next is the root of ξ_1, and $\langle \xi_1, \varepsilon\rangle \nvDash Z$.
2. The formula $\phi = (A \vee B)\,\mathtt{U}(\mathtt{X}\,Z)$ is satisfied by ξ, i.e., $\langle \xi_0, \varepsilon\rangle \vDash \phi$, since there exists the symbol ξ_1 in $\mathit{Next}^*(\xi_0, \varepsilon)$ such that $\langle \xi_1, \varepsilon\rangle \vDash \mathtt{X}\,Z$ and $\langle \xi_0, \varepsilon\rangle \vDash (A \vee B)$.
3. The formula $\psi = \mathtt{G}\,((\downarrow_1(R)) \Rightarrow (\downarrow_2(\mathit{Stop})))$ requires that, in each state of the computation, if the first child of the primary module satisfies R, then the second child has no local next. It is easy to see that it holds that $\xi \vDash \psi$.
4. The formula $\mu = \mathtt{X}\,(\downarrow_1 (P \Rightarrow \rightarrow(\mathtt{X}(S))))$ requires that, in the local next relative to its context, if the first child satisfies P, then its right sibling has a local next satisfying S. This formula is satisfied in ξ.
5. The formula $\lambda = \mathtt{G}\,(\downarrow_2(S \Rightarrow \downarrow_1(T \vee \rightarrow(Y))))$ requires that, at each step of the computation, if the second module instantiated by the primary module satisfies S, then its first child must either satisfy T, or have a right sibling satisfying Y. It is easy to see that $\xi \vDash \lambda$, as well.

4 Towards automatic verification of HLTL properties for hierarchical systems

We envision that HLTL could prove to be a valuable tool in the specification of behavioural properties of hierarchical systems, supporting different phases of the system development lifecycle. For instance, HLTL could be used to formalize system requirements in a more natural yet rigorous way, assisting requirement engineers in the initial phases of system development. Moreover, by integrating HLTL within existing verification frameworks, it is also possible to allow for the fully-automatic verification (*model checking*) of HLTL properties against hierarchical models.

As an example of this, let us consider again the existing work on DSTM presented in [5, 6]. These works define a toolchain to '*translate*' a DSTM model into a semantically equivalent Promela specification for the well-known SPIN model checker [11], enabling simulation and test case generation for DSTM models. This translation was designed in such a way that the concurrent and hierarchical nature of DSTM modules is preserved in the Promela encoding. That toolchain could be extended to support automatic verification of HLTL properties by including an additional module that translates a HLTL specification into a semantically-equivalent, although possibly way more complex and less natural, LTL one. With the LTL equivalent in place, the SPIN Model Checker, which natively supports LTL specifications, could then be used to perform automatic verification. An overview of the resulting DSTM verification framework is shown in Figure 4.

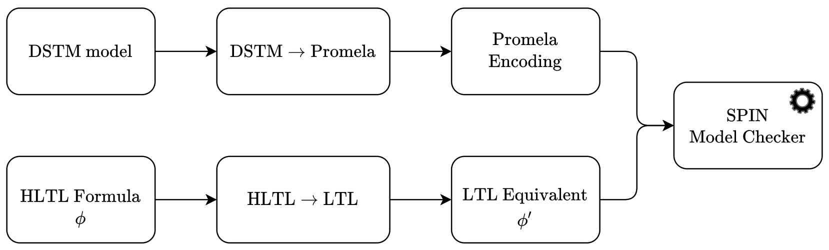

Fig. 4. Automatic HLTL verification framework for DSTM models

5 Conclusions and future works

Driven by the increasing need of methodologies to support the design and verification of safety critical systems, quite a lot of work has been directed towards the definition of hierarchical models such as StateChart, Simulink, or Dynamic State Machines. Less work, however, has been devoted to formal languages to express in a precise, non-ambiguous and practical way properties of the computations of such systems.

In this work, we present a novel formalism named Hierarchical Linear-time Temporal Logic (HLTL), an extension of the well-known Linear-time Temporal Logic (LTL) designed to express linear properties of hierarchical systems. Thanks to the introduction of specific operators, our approach allows practitioners to concisely express linear-time properties of hierarchical system that take into account the intrinsic hierarchical structure of the states of such systems.

In future works, as discussed in Section 4, we plan to integrate HLTL within the modelling framework for Dynamic State Machines (DSTM) proposed in [5, 6], allowing for fully-automated verification of properties, expressed as HLTL formulae, of DSTM systems.

References

1. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 467–481. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
2. Alur, R., Henzinger, T.A., Mang, F.Y., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: Modularity in model checking. In: International Conference on Computer Aided Verification. pp. 521–525. Springer (1998)
3. Alur, R., Kannan, S., Yannakakis, M.: Communicating hierarchical state machines. In: International Colloquium on Automata, Languages, and Programming. pp. 169–178. Springer (1999)
4. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
5. Benerecetti, M., De Guglielmo, R., Gentile, U., Marrone, S., Mazzocca, N., Nardone, R., Peron, A., Velardi, L., Vittorini, V.: Dynamic state machines for modelling railway control systems. Science of Computer Programming **133**, 116–153 (2017). https://doi.org/https://doi.org/10.1016/j.scico.2016.09.002, https://www.sciencedirect.com/science/article/pii/S0167642316301332, formal Techniques for Safety-Critical Systems (FTSCS 2014)
6. Benerecetti, M., Gentile, U., Marrone, S., Nardone, R., Peron, A., Starace, L.L.L., Vittorini, V.: From dynamic state machines to promela. In: Biondi, F., Given-Wilson, T., Legay, A. (eds.) Model Checking Software. pp. 56–73. Springer International Publishing, Cham (2019)
7. Benerecetti, M., Peron, A.: Timed recursive state machines: Expressiveness and complexity. Theor. Comput. Sci. **625**, 85–124 (2016). https://doi.org/10.1016/j.tcs.2016.02.021, https://doi.org/10.1016/j.tcs.2016.02.021
8. Bozzelli, L., Torre, S.L., Peron, A.: Verification of Well-Formed Communicating Recursive State Machines. Theor. Comput. Sci. **403**(2-3), 382–405 (2008)
9. Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming **8**(3), 231–274 (1987)
10. Herkert, J., Borenstein, J., Miller, K.: The boeing 737 max: lessons for engineering ethics. Science and engineering ethics **26**(6), 2957–2974 (2020)
11. Holzmann, G.J.: The model checker spin. IEEE Transactions on software engineering **23**(5), 279–295 (1997)
12. Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th international conference on software engineering. pp. 547–550 (2002)
13. Lanotte, R., Maggiolo-Schettini, A., Peron, A., Tini, S.: Dynamic hierarchical machines. Fundamenta Informaticae **54**(2-3), 237–252 (2003)
14. Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: Test generation and test prioritization for simulink models with dynamic behavior. IEEE Transactions on Software Engineering **45**(9), 919–944 (2018)
15. Nardone, R., Gentile, U., Peron, A., Benerecetti, M., Vittorini, V., Marrone, S., De Guglielmo, R., Mazzocca, N., Velardi, L.: Dynamic state machines for formalizing railway control system specifications. In: International Workshop on Formal Techniques for Safety-Critical Systems. pp. 93–109. Springer (2014)
16. Oyeleke, R.O., Chang, C.K., Margrett, J.: Situation-driven context-aware safety model for risk mitigation using ltl in a smart home environment. In: 2020 IEEE World Congress on Services (SERVICES). pp. 22–24. IEEE (2020)

17. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (Oct 1977). https://doi.org/10.1109/SFCS.1977.32
18. Zhang, S., Zhai, J., Bu, L., Chen, M., Wang, L., Li, X.: Automated generation of ltl specifications for smart home iot using natural language. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 622–625. IEEE (2020)