Modelling Evolvable Systems: A Temporal Logic View

HOWARD BARRINGER AND DAVID E. RYDEHEARD

1 Motivation

There is a growing trend to develop and deploy computing systems that adapt or evolve themselves dynamically according to use and observed changes in their environment. Evolution may be triggered either from within the system or be forced by some external, and not necessarily human, agent. The traditional view of a non-trivial and meaningful system specification being invariant for the duration of a system run will no longer hold for such an evolvable system; it will be rare to be able to predict all possible ways in which a system may evolve.

In this contribution, we consider one approach to how temporal logic, and specifications based on temporal logic, can be used to support formal reasoning about evolvable systems in a natural way. Our approach builds on our past work in temporal logic-based specification, the imperative view of temporal logic as in our work on executable temporal logic, METATEM, developed jointly with Dov Gabbay in the late 1980s [3], [2], and the more recent work on the EAGLE logic for run-time verification [4], [1].

A rather crude view of formal systems development has basic steps comprising

1. formalisation of requirements as a set of system properties
2. definition/construction of a formal specification
3. revision of the specification
4. development of an implementation that satisfies the specification
    - via refinement and/or decomposition of specification, together with theorem proving
    - via checking individual properties against a purported model
5. revision of the specification and/or implementation

6. test analysis
7. deployment of system

Although this is well recognized as being overly simplistic and generally unrealizable, a key point is that once the system is deployed, its specification remains invariant during its deployed lifetime. Of course, later releases of the system, including bug fixes, new features and general updates, are quite usual. However, one again regards the specifications of these new releases as invariant during their (often too short a) lifetime.

This static view of specification and system is unsatisfactory from many points of view. In some highly specialized application areas, for example deep space mission software, control software for future unmanned exploration of planets, e.g. NASA's Martian rovers, a certain degree of adaptive software has already been prototyped. In Earth-based deployment, software systems are beginning to be constructed that feature in limited ways *autonomy*, *adaptation* and *evolution*. Of course, there are many applications which include algorithms that adapt, or attempt to optimize, their performance to the current situation, e.g. network routing, dynamic load-balancing of work for processors, etc. In these latter scenarios, however, the overall system specification remains invariant, e.g. correct delivery of packets or the timely execution of work. At a quite different organisational level, the evolutionary behaviour of businesses and business processes, and the formal descriptions of such behaviour, has attracted considerable interest (see, for example, [10], [9] or [6]).

The question then is, as one changes the nature of systems so that adaptation or evolution becomes a dominant feature and to be present at significantly higher levels of system organisation, how does one specify and reason about systems with such features?

1.1 Behavioural specification

The possible worlds model, which is usually used to provide meaning for temporal and modal logics might at first sight appear appropriate for capturing the behaviour of evolving systems. The accessibility relation captures the evolution between the system in one world and another. But in most uses the notion of evolution of the system is at a very low computational level. For example, typically, in temporal logic-based specification, the world represents the state of a system, providing for example values to variables, and the accessibility relation captures the next-state relation of the "computation" of the system being specified. The dynamic behaviour of a system, the specification of which remains invariant, can thus be modelled.

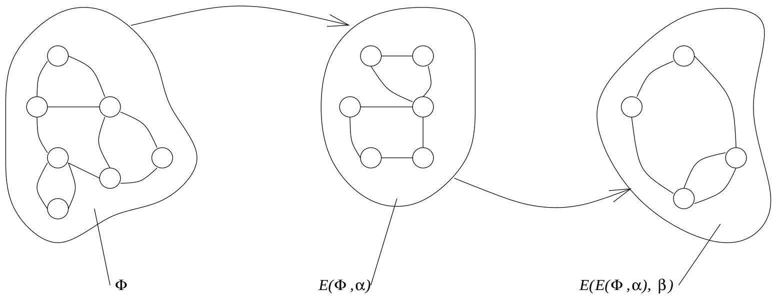

Figure 1. Evolving Worlds

Now consider the following scenario. A system is running according to some specified behaviour, say Φ. At some point, the system is interrupted (the apparatus for doing that is not important here), it is then modified and re-started but now running according to a modified specification, say $E(\Phi, \alpha)$. Let us view the entirety of the interrupt, modify and re-start process as an evolutionary step of the original system. Indeed, the transformer $E : \mathit{Spec} \times Modification \rightarrow \mathit{Spec}$ performs the evolution.

It appears sensible to separate the system level evolution from the computational state evolution of the system. Thus, in a possible worlds setting, at the system evolution level, let the individual worlds be associated with the possible evolved system specifications; indeed, one might take the worlds as Kripke structures capturing the computational evolution of the specified system. The relation between the Kripke structures is then the system evolution relation. We thus have a two-level model description of the overall system.

As presented above in pictorial form, the set-up all seems rather straightforward! Perhaps one reason for being fooled is that we are reading the diagram operationally: execute the system according to the specification Φ, then evolve at some point with modification α, and then continue executing $E(\Phi, \alpha)$, etc.. But what is α? Where did it come from? If we know what α is at the time the original system Φ is developed then indeed one can reason about the future evolution. This, however, is rather unlikely. Indeed, in the absence of any constraints on what the modifications α and β may be, one would need to reason about the system Φ and all its possible evolutions — clearly a challenging if not impossible task.

As an aside, it is worth noting that one could construct a temporal logic capable of expressing the situation outlined above. It would need to be a multi-modal temporal logic containing in addition to the basic next step temporal operator, a system evolution operator (each characterising the respective underlying evolution relation). However, the logic would also require some means to embed meta-level action, i.e. changing behaviour, at the object level. Rather simplistically, the following (heavily extended) fixpoint interval temporal logic could represent an evolving system.

$$\nu B(F).(F; \exists \alpha.(input(\alpha) \wedge \bigcirc B(E(F, \alpha))))$$

Read the above formula as follows: F is a specification of the "current" system. At some point F stops and is followed by an input of the modification α which is then applied to the specification F to yield a new specification for execution. This is a very powerful logic, undoubtedly far too powerful, and one has to question its virtue here, over and above just formal description. We will later propose a different approach to the use of temporal logic which appears to provide a restricted but flexible and sufficiently general framework to modelling and reasoning about evolvable systems.

1.2 Modifying a specification

Let us briefly comment on some crucial and difficult issues associated with modifying specifications, in particular, behavioural specifications given purely logically. Such a specification is essentially a logical theory given as a set of constraints over a set of observables which are expressed as predicates and functions. As an example, imagine the behaviour of a buffer is specified in terms of observations of input to and output from the buffer; indeed we assume predicates $in(v)$ and $out(v)$ hold if value v passes in, respectively out, from the buffer. Without going into the details of what the specification is (just imagine it!), name it Φ and let us assume that at some stage during the buffer's operation something causes the system to evolve through the ADDITION of a new constraint α on the buffer. If we further assume that this new constraint is not in conflict with, i.e. is consistent with, the old specification Φ, then the new specification can be given by $\Phi \wedge \alpha$. Further restricting behaviour is, in effect, relatively straightforward and can be embedded easily as a logical operation. If, however, the required evolution required REMOVAL (or replacement) of some behaviour, characterised by a logical constraint, we are in deep water. Deletion is a very difficult issue and has been stretching some computer scientists and logicians for many years. Of course, physical deletion of some given constraints from a known specification is clearly possible - this is a meta-level operation. However, physical deletion doesn't usually yield the desired effect of removing unwanted behaviours. The deletion we most probably want is a logical deletion, i.e. to ensure that α is not amongst the consequences of the modified specification. How this is achieved, or whether this can be achieved, appears to be a fundamental issue for giving a logical characterisation of an evolving system. There are interesting ideas based on the notion of Gabbay's anti-formulae [8] - which effectively annihilate the use of a particular formula during proof, thus disabling its use. However, at present, such ideas won't actually achieve what is required here, for some other formula is sure to characterise the undesirable behaviour, which then requires addition of all such anti-formulae! This is clearly an avenue to be explored but it is one that is certainly strewn with numerous pitfalls.

1.3 Checking and monitoring for safe behaviour

Assume that we have a solution to creating formal logic-based specification for an evolving or adaptive software system. What can one do with it? The ultimate challenge is to prove, preferably in an automated fashion, that certain properties are invariant over evolution or adaptation. The properties may be both safety and liveness. How meaningful the properties are depends crucially on what constraints are placed on how a system evolves. As indicated above, if one allows complete freedom in evolution then there is probably very little that can be achieved. On the other hand, one can turn this around to try to answer synthesis questions such as what constraints must be placed on the evolution of this system in order to preserve a desired property.

The automated checking and/or synthesis of properties is an intractable, if not impossible, problem and even with today's best technology it will be of extremely limited capability for these sorts of evolvable systems, i.e. size of system being analysed. This argument should not be used to rule out research into the development and application of techniques to combat these problems for specialized situations. We're just pointing out there'll be no silver bullets for full-scale formal verification.

An alternative to static verification, as achieved by model-checking, automated theorem proving and their combination, is dynamic formal analysis, or run-time verification. In run-time analysis, the system under scrutiny is observed during execution and properties checked either in an on-line (potentially interacting with the system) or off-line (a post hoc analysis of some execution). Clearly this can be significantly less costly than attempting to analyse the complete behaviour of a system, which is what static analyses, such as model-checking, will attempt to achieve. On-line analyses have the benefit of being able to take action on the system as soon as it is discovered that some property fails. Integration of observer and analyst with the system under scrutiny must ensure, however, that there is no unacceptable degradation in overall system performance. Furthermore, integration of an observer shouldn't change the desired observational behaviour of the system. And most interesting and challenging of all is the integration of action in both the logical specification, analysis and execution frameworks. Is adaptation and evolution treated at a meta-level or embedded with the logical specification itself? To return to the rather expressive fixpoint formula given above, one can certainly see how it might be used in a run-time analysis. However, apart from the semantic cleanliness, it is not clear what else may be gained.

1.4 Towards a logic for evolving systems

In this article, we present our initial steps at using linear-time temporal logic for the specification, and the static and dynamic logical analysis, of evolvable systems. We use as a motivational example a model of an evolvable bounded buffer. A process, which we refer to as an *evolver* process, monitors the dynamic behaviour of a bounded buffer. If the evolver finds that the buffer needs to store more items than the buffer's current capacity, the evolver triggers an evolution of the buffer to increase its size. On the other hand, if the evolver finds that the buffer is nearly empty most of the time, the evolver triggers an evolution that contracts the capacity of the buffer. Of course, there are different ways in which a buffer may expand or indeed contract. For example, it may expand by first creating a new empty buffer, of the same size and type, and then re-directing the output of the original buffer to the input of the new one and then taking the new output and directing at the target of the original buffer. To put this another way, the two buffers are serially piped together. Or an expansion could have been achieved by putting two buffers in parallel. Or perhaps even simply, the buffer may have had a specific capacity that can directly be modified. To make this all quite concrete, we present one possible implementation of the example in Java. A property that we'd like to establish of the subsequent *evolvable* buffer – the buffer and its evolver– is that, the composite system has effectively unbounded capacity.

Rather than collapse the reasoning all down to the object level, we present a two-level temporal logic that can be applied, quite elegantly, to capture both the base-level behaviour and meta-level evolutionary behaviour of the buffer. We require a model structure, which we call an *E-structure*, that captures the two-level structure suggested by Figure 1. That particular figure really presents too simplistic a view. Indeed, the set-up has to be more flexible and hence a little more complex. For instance, in order to able to reason about the (meta-level) evolution of a system, predicate abstraction is used to abstract away from lower-level detail. Indeed, it is precisely that abstraction process that simplifies the reasoning over different Kripke-like structures. Sections 3 and 4 present this model-theoretic framework for a two-level approach to evolvable systems.

We then introduce an appropriate fix-point temporal logic over our E-structures that supports our specification and reasoning style. For example, there are three types of *next* modalities: (i) the familiar $\bigcirc$ modality for moving to the next state of an existing system; (ii) a family of $\mathbf{X}_e$ action modalities that are used to indicate an evolutionary transition e of the system; (iii) the anonymous $\mathbf{X}$ modality to denote that some arbitrary evolutionary step occurs.

Sufficient technical machinery is then in place for us to present some examples of evolvable buffering systems.

2 An Evolvable Bounded Buffer

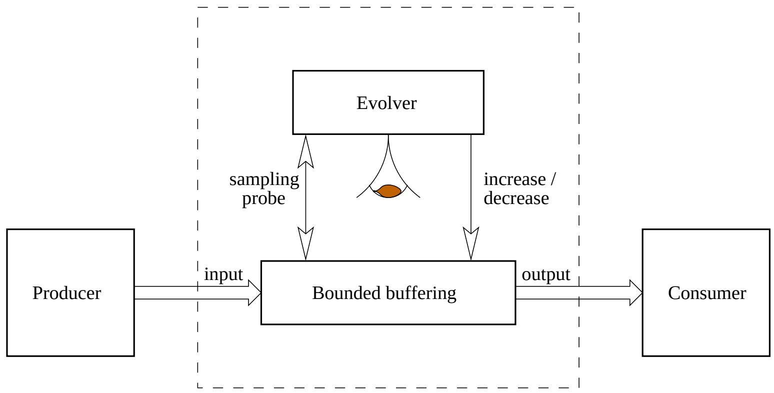

Figure 2. Communication via an Evolvable Buffer

Figure 2 depicts a typical scenario for an application of a bounded buffer. A producer process passes items to a consumer process. The producer and consumer processes are assumed to be independent, asynchronous, processes and hence typically have different rates of production and consumption of items. A bounded buffer is used to smooth the transmission between the producer and consumer and avoid unnecessary blocking that may occur through the buffer being full and thus blocking further input and slowing the producer process. If we further assume that the rates of production and consumption of items are highly variable, it is then difficult to predict an optimal size for the bounded buffer. The buffering system therefore needs to be adaptive. The buffer capacity must adapt, or evolve, dynamically to suit the varying rates of input from the producer process and output demanded by the consumer process. An evolver process samples the performance of the bounded buffer via probes and determines when to increase, or decrease, the buffer's capacity.

We provide a simple model of this example in Java. The class definition below describes a bounded buffer (that holds strings) implemented as a passive object, indeed in concurrency terminology as a monitor. The class has two synchronized methods for input to and output from the buffer. The methods are synchronized to ensure exclusive access to the shared instance variables of class. The constructor of the class creates buffer objects of fixed size. If an input method is called from a thread `t` when the buffer is full, the calling thread `t` is suspended on a wait list until there is notification that buffer has space for more items. Similarly, if a thread attempts to obtain an item from the buffer when there are none, the calling thread is also suspended until the buffer has some actual content to pass out.

```
public class Buffer {

  protected int max;
  protected String [] content;
  protected int head;

  Buffer(int rqdMax){
    max = rqdMax;
    head = 0;
    content = new String [max];
  }

  public synchronized void input(String item)
          throws InterruptedException
  {
    while (!(head < max) ) { wait();}
    content[head++] = item;
    notifyAll();
  }

  public synchronized String output()
          throws InterruptedException
  {
    while ((head == 0) ) { wait(); }
    String value = content[--head];
    notifyAll();
    return value;
  }
```

```
} //class Buffer
```

The next class definition extends the Buffer class with features to support the notion of an evolvable buffer. The possible evolutions of the buffer are kept very simple. A method is provided to increase the size of the buffer and a method to decrease its size. Naturally, both methods are synchronized methods to ensure exclusive access to the shared information. The class also provides two probe methods, `shouldExpand` and `shouldContract`, which can be called to determine whether the buffer evolve, i.e. its capacity either expanded or contracted.

```
public class EvolvableBuffer extends Buffer
{
  private final int initialSize;
  private double limit;
  private int factor;

  EvolvableBuffer(int size, double rqdLimit, int rqdFactor){
    super(size);
    initialSize = size;
    limit = rqdLimit;
    factor = rqdFactor;
  }

  public boolean shouldExpand(){
    return (head >= limit*max);
  }

  public boolean shouldContract(){
    return (max > initialSize && head <= (1-limit)*max);
  }

  public synchronized void expand()
  {
    reSize(max*factor);
    max = max*factor;
    limit += (1-limit)/factor*(factor-1);
    notifyAll();
  }

  public synchronized void contract()
  {
    reSize(max/factor);
    max = max/factor;
    limit = 1-((1-limit)/(factor-1)*factor);
    notifyAll();
  }
```

```
  private void reSize(int newSize)
  {
    String [] copy = new String [newSize];
    for (int i = 0; i < head; i++){
      copy[i] = content[i];
    }
    content = copy;
  }

} //class EvolvableBuffer
```

Next, in the code below, we provide a very simple evolver process. An evolver is an active object that samples the state of a bounded buffer using the probe methods `shouldExpand` and `shouldContract` of the `Buffer` object with which it is associated. If either method returns true, the evolver invokes the appropriate evolutionary action via `expand` or `contract`. We also give a suitable class definition for the example main program that sets up a producer, consumer, evolvable buffer and evolver process, as depicted in Figure 2.

```
public class Evolver extends Thread {
  EvolvableBuffer x;

  Monitor(EvolvableBuffer givenBuffer){
    this.x = givenBuffer;
  }

  public void run()
  {
    while (true) {
      try {
        if (x.shouldExpand()) { x.expand(); }
        else if (x.shouldContract()) { x.contract(); }
        Thread.sleep(500);
      }
      catch (InterruptedException e) {}
    }
  }

} //class Evolver

public class Example
{
  public static void main (String [] args) {
```

```
    EvolvableBuffer x = new EvolvableBuffer(8, 0.666667, 2);
    Producer p = new Producer("A", x);
    Consumer c = new Consumer("B", x);
    Evolver o = new Evolver(x);
    p.start();
    c.start();
    o.start();
    }
} //class Example
```

When this particular program is run, the producer and consumer processes are always eventually able to input or output. In fact, this would also be the case for a producer, consumer and basic, non-evolvable, buffer. The key difference, however, is that the evolvable buffer will not block on input nearly as often as the basic buffer. Indeed, the evolver process will (crudely) adjust the size of the evolvable buffer to match the varying rates of producer and consumer.

If, on the other hand, the consumer process were to stop, the evolver process would ensure that the evolvable buffer kept increasing in size and never permanently blocking the producer. This is a typical property that we'd like to establish of the system.

3 Models for Evolvable Systems

We now begin a mathematical analysis of the structure and behaviour of evolvable systems such as the system of buffers given in the previous section.

Here are some of the key ideas introduced in this analysis:

- The computational structure of evolvable systems is modelled using transition systems. This is one of several possibilities for computational models - one which gives us sufficient generality of the purposes of this paper.

  We introduce the notion of an *E-structure*, a two-level structure consisting of a Kripke-like structure whose states (or worlds) are themselves Kripke structures. Evolutionary steps are labelled and provide a transition relation connecting Kripke structures. In fact, they link states in one structure with states in a possibly different structure.

- In the example of the buffers above, we model the individual buffers, and the evolutionary steps between buffers as an E-structure. However, the evolver and its interaction with the buffer system is not included in the model. To specify the behaviour of the evolver and reason about and verify properties of the overall evolvable system we introduce a temporal logic interpreted in E-structures. To do so we need to adapt the usual presentation of temporal logic expanding the range of modal operators so that we can describe both the behaviour of local systems and the behaviour of the evolutionary steps. Similar forms of two-level modal and temporal logics have been proposed and investigated elsewhere, see for example [11].

A range of temporal logics is available and most could be adapted to this purpose. In fact, we present a Linear Time Logic where propositions are interpreted along computational paths in the model. Paths include both steps within local systems and evolutionary steps. The logic we present is based on EAGLE-like primitives, as one key motivation for this work lies in the automated verification and run-time monitoring of evolvable systems, an application for which EAGLE was designed [4], [1].

- In this description of models, evolutionary steps may take place between arbitrary and unrelated source and target systems. However, inherent in the notion of 'evolution' is that the source and target systems share common structure, so that the target is indeed a modification or evolution of the source. To reason about the overall behaviour of an evolvable system we need to be able to identify certain common features occurring across all the local systems. Note that, in general, such common structure may be represented in very different ways in different local systems. Indeed one reason for an evolutionary step might be to re-represent some internal structure for reasons of capacity (as in the buffer example), or efficiency, or otherwise.

  Again, it appears that there are several possible approaches to identifying and reasoning about common features of the local systems. One possibility, adopted here, is to use *predicate abstraction*. Thus properties of states in local systems, which are common to all systems, are abstracted to form a new transition system on which the evolutionary steps act at an abstracted level. We may then reason about the underlying evolvable system using the structure of the abstracted system. Later, we will define the relevant form of predicate abstraction and show how it can be used to specify and reason about various systems of buffers.

We begin with the definition of a model as a two-level Kripke structure.

DEFINITION 1. An *E-structure*, E, is a 5-tuple

$$E = (K, K_0, \Lambda, R, I)$$

where

- K is a set of labels of Kripke structures, i.e. for each $k \in K$ there is a Kripke structure $(S(k), S_0(k) \subseteq S(k), R(k) \subseteq S(k) \times S(k))$, where $S(k)$ is the set of states, $S_0(k) \subseteq S(k)$ the set of start states, and $R(k) \subseteq S(k) \times S(k)$ the transition relation,
- $K_0 \subseteq K$ is the set of start states,
- Λ is a set of labels (of evolutionary steps), and
- $R \subseteq (k : K)S(k) \times \Lambda \times (k : K)S(k)$, where (borrowing notation from type theory) $(k : K)S(K) = \{(k, s)|k \in K, s \in S(k)\}$ is the set of all (dependent) pairs of k in K and states s in $S(k)$.
- I is an interpretation of propositional symbols. Propositional symbols are of two kinds, the global symbols in a set P describing properties of members of K, and a K-indexed collection of local symbols $P_k, k \in K$, where $p \in P_k$ is a propositional symbol describing properties of states in the Kripke structure k.

  An *interpretation* of propositional symbols $(P, \{P_k|k \in K\})$ consists of

  - a function $I : K \to \mathcal{P}(P)$, where $\mathcal{P}(P)$ is the powerset of P, and
  - for each $k \in K$ a function $I_k : S(k) \to \mathcal{P}(P_k)$.

  This definition extends to an interpretation of n-place predicate symbols as n-place functions from states to truthvalues.

As defined, E-structures have just two levels of structure, but we envisage systems where more levels are natural. For example, we may observe not only the behaviour of the local systems but also the computational history of the evolutionary steps. An evolver may then change the entire system if the pattern of evolutionary steps is not as desired - a circumstance that we may model as evolutionary steps between E-structures of the above kind.

3.1 Example: Modelling an Evolvable Buffer

As an example of an E-structure, consider a case of the evolvable buffer described in the previous section, for simplicity expanding only when full and contracting only when empty.

Firstly, we model a buffer of capacity $N > 1$ as a vector of values from set V of length at most N. We denote the collection of these vectors as $V^{[N]}$. For each $N > 1$ this defines a Kripke structure $B_N = (S_N, S_N^0, R_N)$, where

- $S_N = V^{[N]}$,

- the initial states $S_N^0 = \{[]\}$ consist of the empty vector [] only, and
- R_N is the transition relation defined by $\forall s \in S_N, v \in V.\ |s| < N \implies (s, s : v) \in R_N$ and $\forall s \in S_N, v \in V.\ (s : v, s) \in R_N$. Here $s : v$ is the vector s extended with value v. The first clause corresponds to input of v to the buffer with contents s, the second clause to output value v from buffer with contents $s : v$.

The only predicate symbol on this buffer is $\mathsf{contents}_N : V^{[N]}$, interpreted in state $s \in V^{[N]}$ as $\mathsf{contents}_N(s')$ iff $s = s'$.

We now introduce evolutionary steps of `expand` and `contract` as follows, for each $N > 1$:

$$\begin{aligned}&\texttt{expand} : B_N, s \to B_{2N}, s \text{ where } |s| = N\\ &\texttt{contract} : B_{2N}, [] \to B_N, []\end{aligned}$$

Thus there is an evolutionary step of expansion when a full state is encountered and then the buffer capacity is doubled but the contents left unchanged. Likewise, when the empty state is encountered, there is a contraction step to a buffer of half the capacity.

With these evolutionary steps, the system of buffers B_N for $N > 1$ defines an E-structure B.

We illustrate part of this system in Figure 3. The left-hand structure k_0 of represents a Kripke structure corresponding to the behaviour for a buffer of size $N = 2$, the right-hand structure is a buffer of size $N = 4$, and some evolutionary steps have been illustrated.

This example is too simple to illustrate all aspects of evolvable systems. In particular, the same notion of contents applies across all the local systems. In general, internal structure like this will appear differently in different local systems and evolutionary steps will allow such modification. As a very simple example, consider Figure 4 which illustrates an evolutionary step in a buffer system in which instead of an expansion of a sequential buffer, a parallel buffer is introduced. Then the notion of contents in the source system is spread over two structures, each of which has contents, in the target system.

As mentioned previously, we handle this changing representation of internal structure using predicate abstraction. We now present the relevant notion of predicate abstraction and show how to define it for evolvable systems.

4 Predicate abstraction

We first consider individual Kripke structures and describe the relation of predicate abstraction between two structures.

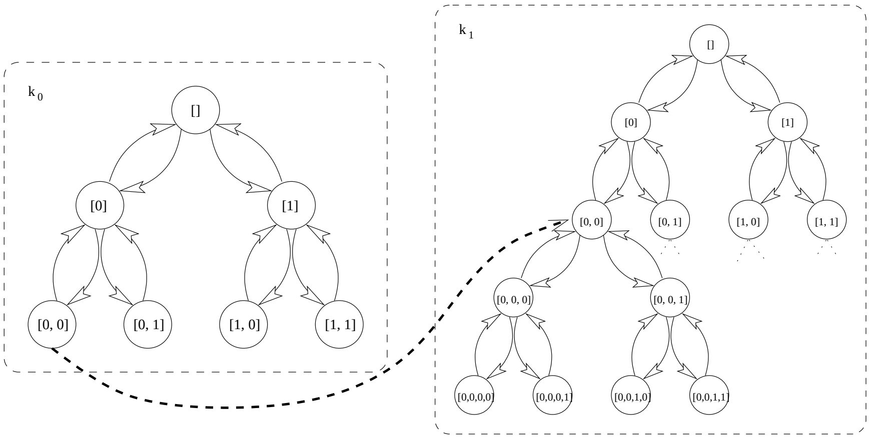

Figure 3. Part of an E-structure for an Evolvable Buffer model

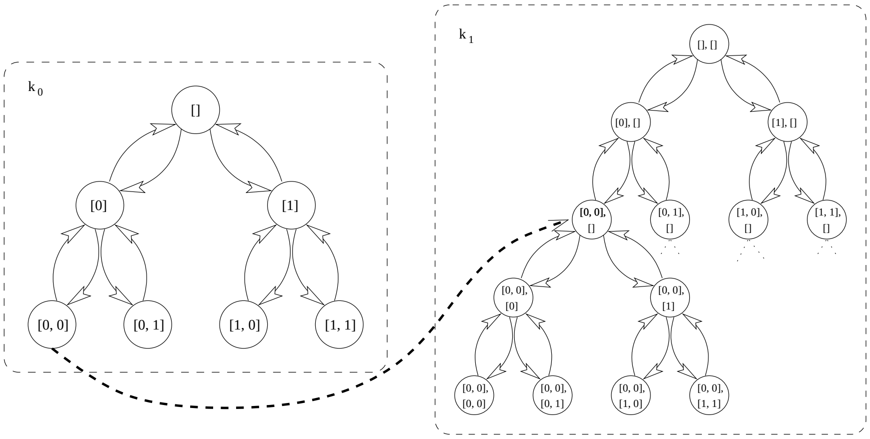

Figure 4. Part of an E-structure for a modified Evolvable Buffer

Consider two Kripke structures, the 'abstract' structure $k_A = (S_A, S_A^o, R_A)$ over predicate symbols P_A and interpretation I_A, and the 'concrete' structure $k_C = (S_C, S_C^o, R_C)$ over predicate symbols P_C and interpretation I_C. Predicate abstraction is defined in terms of extension of each of the interpretations to satisfiability relations of the form:

$$k_A, s \models_{I_A} \phi, \text{ where } s \in S_A, \text{ and } \phi \text{ is a formula in predicate symbols } P_A,$$

$$k_C, s \models_{I_C} \phi, \text{ where } s \in S_C, \text{ and } \phi \text{ is a formula in predicate symbols } P_C.$$

The relationship of predicate abstraction arises from a collection Δ of definitions of the form $p \triangleq \phi$, one for each $p \in P_A$ with ϕ a formula built from the predicate symbols in P_C:

DEFINITION 2. We say k_A is a *predicate abstraction* of k_C via Δ iff

$$\begin{aligned}\forall s, t \in S_A.\ \ & R_A(s,t) \text{ iff} \\ & \quad \exists s', t' \in S_C.\ R_C(s', t') \text{ and for each } p \triangleq \phi \in \Delta. \\ & \quad\quad k_A, s \models_{I_A} p \text{ iff } k_C, s' \models_{I_C} \phi \text{ and} \\ & \quad\quad k_A, t \models_{I_A} p \text{ iff } k_C, t' \models_{I_C} \phi.\end{aligned}$$

The idea is that the definitions in Δ allow us to consider states in k_C as equivalent according to whether or not they are distinguished under the predicate symbols in P_A. Two states are related in the abstraction if any of the corresponding concrete states are related.

Given an abstraction k_A of k_C via Δ, we say k_A is a minimal abstraction iff there is no k'_A which is an abstraction of k_C via Δ such that $|S'_A| \leqslant |S_A|$, where S'_A is the set of states of k'_A and likewise S_A of k_A.

Note that a minimal abstraction over predicates P_A has no more than $2^{|P_A|}$ states.

4.1 Predicate abstraction and E-structures

We now describe how predicate abstraction interacts with evolutionary structure.

To simplify notation, we consider the case where there is a one-one correspondence between the abstract and concrete Kripke structures, and use the notation k_A, k_C to denote corresponding structures, with predicate symbols P_A and P_C and interpretations I_A and I_C. To define the notion of predicate abstraction over models we need an indexed collection of definitions Δ_{k_A,k_C}, one set of definitions for each pair of corresponding structures k_A, k_C, which define the predicate symbols $p \in P_A$, in terms of formulas ϕ built from the predicate symbols in P_C.

Given two corresponding E-structures and an indexed collection of definitions, we define when the E-structures are related by predicate abstraction:

DEFINITION 3. Consider E-structures $E_A = (K_A, K_A^0, \Lambda, R_A)$ and $E_C = (K_C, K_C^0, \Lambda, R_C)$ in one-one correspondence with predicate symbols P_A and

P_C and interpretations I_A and I_C, extending to satisfiability relations $\models_{I_A}$ and $\models_{I_C}$. Let Δ_{k_A,k_C} be an indexed collection of definitions for corresponding $k_A \in K_A$ and $k_C \in K_C$.

We say E_A is a *predicate abstraction* of E_C iff

1. k_A is a predicate abstraction of k_C via Δ_{k_A,k_C} for all corresponding $k_A \in K_A$ and $k_C \in K_C$,

2. We say $s \in k_C$ represents $s_A \in k_A$ iff for all $p \triangleq \phi$ in Δ_{k_A,k_C}

$$k_C, s \models_{I_C} \phi \text{ iff } k_A, s_A \models_{I_A} p.$$

   Then we require that for each $(k_A, s_A, e, k'_A, s'_A) \in R_A$, for all s representing s_A, and all s' such that $(k_C, s, e, k'_C, s') \in R_C$, s' represents s'_A.

4.2 Example: A system of evolvable buffers and predicate abstraction

As a simple example of predicate abstraction, we consider the above system B of evolvable buffers $B_N = (S_N, S_N^0, R_N)$, where S_N is defined as the collection of vectors $V^{[N]}$. The evolutionary steps between these systems consist of those of expand and contract.

A predicate abstraction from buffers in this form is illustrated in Figure 5 where k_0 and k_1 are in fact the same Kripke structure k.

Formally, this E-structure is defined as follows.

$$\begin{aligned} E = (\{k_0, k_1\}, \{k_0\}, \{ \; & ((k_0, s_2), \texttt{expand}, (k_1, s_1)), \\ & ((k_1, s_0), \texttt{contract}, (k_0, s_0)), \\ & ((k_1, s_0), \texttt{contract}, (k_1, s_0)), \\ & ((k_1, s_2), \texttt{expand}, (k_1, s_1))\} \end{aligned}$$

where

$$\begin{aligned} k_0 = (\{s_0, s_1, s_2\}, \{s_0\}, \{ \; & (s_0, s_1), \\ & (s_1, s_0), (s_1, s_1), (s_1, s_2), \\ & (s_2, s_1)\} \\ k_1 = (\{s_0, s_1, s_2\}, \{s_0, s_1\}, \{ \; & (s_0, s_1), \\ & (s_1, s_0), (s_1, s_1), (s_1, s_2), \\ & (s_2, s_1)\} \end{aligned}$$

We now show that this system is in fact a predicate abstraction of the system of buffers B.

Each of k_0 and k_1 (which as they are the same we denote by k) has three states s_0 = 'empty', s_2 = 'full' and s_1 = 'neither (empty nor full)'. The predicates on these are empty and full with the obvious interpretations.

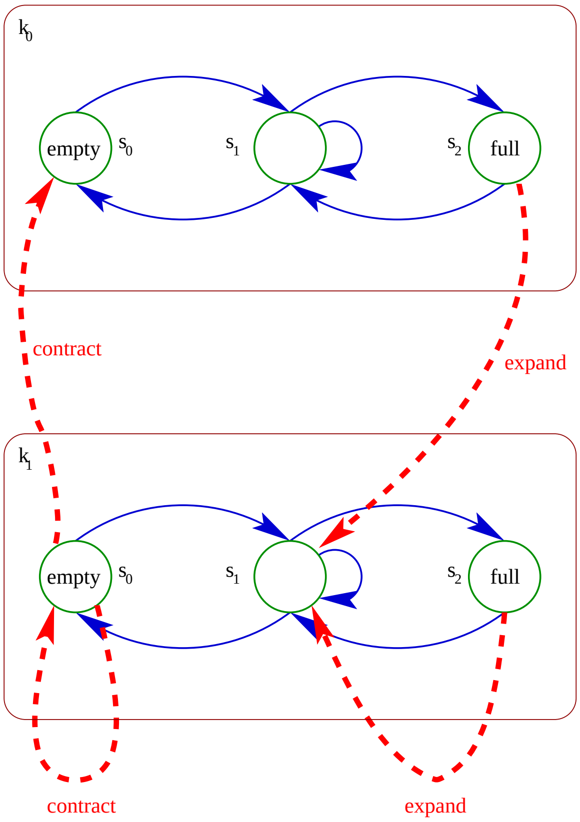

Figure 5. Abstraction of an Evolvable Buffer

Linking these systems with the buffers are the collection of definitions of the predicates, which in this case are:

$$\mathsf{empty} \triangleq \mathsf{contents}_N([])$$
$$\mathsf{full} \triangleq \exists s \in V^{[N]}.\ \mathsf{contents}_N(s) \wedge |s| = N$$

To verify that k is a predicate abstraction of B_N for each $N > 1$ we need to show that for each transition in k there is a transition in B_N such that, at both the source states and the target states of these transitions, the interpretation of the left-hand sides of each of the definitions coincides with the interpretations of the right-hand sides.

For example, for the transition from 'neither' to 'full' in k, there is a transition in B_N from $s \in S_N$ with $|s| = N - 1$ to $s : v$ for any $v \in V$. For the definition of empty, its interpretation at 'neither' is false as is the interpretation of the right-hand side at any s with $|s| = N - 1$. The same interpretation applies to the definition of full at state 'neither' and the right-hand side at each s with $|s| = N - 1$. For the target states, again the interpretations of the definitions of empty coincide. For the case of the definition of full this is interpreted as true in k and also $s : v$ is full in B_N (i.e. satisfies the right-hand side of the definition), as required.

A similar verification for each transition in k establishes that k is a predicate abstraction of B_N for each $N > 1$.

Now let us extend this abstraction to the E-structure B of evolvable buffers.

We now show that the E-structure in Figure 5 is a predicate abstraction of this E-structure B. We have already verified that k is a predicate abstraction for B_N for each $N > 1$. All that remains is to show that this abstraction respects the evolutionary steps.

It may help to have an illustration of this situation. In Figure 6 we depict an evolutionary step of `expand` and how it acts both on the buffers and on the abstracted systems.

The evolutionary step of `expand` is a step from 'full' (i.e. state s_2) to 'neither' (i.e. state s_1). Now, full states s in buffer B_N are those that satisfy $|s| = N$, and for each such state the evolutionary step `expand` maps s to s as defined above, and s is neither empty nor full in B_{2N} hence represents this abstract state. A similar argument applies to the `contract` step.

This establishes that the E-structure in Figure 5 is indeed a predicate abstraction of the simple system of evolvable buffers.

5 The temporal logic $\mathcal{E}$-tl

We introduce a two-level temporal logic, $\mathcal{E}$-tl, that will be interpreted over E-structures. $\mathcal{E}$-tl is essentially a linear time temporal logic, although it has a branching capability, and possesses temporal modalities for reasoning (i) purely locally within a system, (ii) locally and globally within and across systems and (iii) purely globally considering just the evolutionary steps being made. To reason directly about possible evolutionary choice, the logic is equipped with a (branching) modality $<e>$ (for each label e of an evolutionary step) such that $<e>\, f$ holds when there is a possible evolution e enabled after which f would hold.

We have based the logic $\mathcal{E}$-tl on the style and ideas behind the rather expressive temporal logic EAGLE. EAGLE was designed specifically to support the run-time monitoring of complex temporal patterns of observable

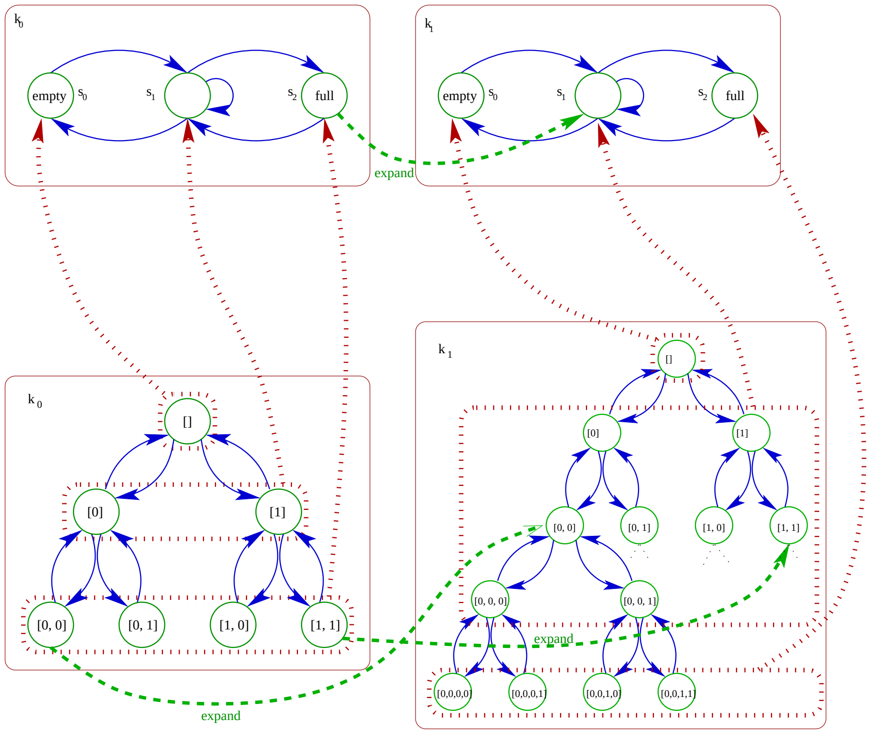

Figure 6. An evolutionary step and its predicate abstraction

system behaviour. In essence, it can be characterized as fixpoint temporal logic over finite (linear) traces, where, additionally, the fixpoints can be parameterized by data values. However, rather than have the user write formulas in fixpoint form, EAGLE supports user-defined recursive rule schema and, formally, has only a primitive next, previous, concatenation and chop temporal operators. We chose EAGLE as a basis for $\mathcal{E}$-tl because of its richness and intimate connection with run-time monitoring, which is central to the notion of an evolver monitoring the behaviour, or aspects thereof, some system component. On the other hand, $\mathcal{E}$-tl differs from EAGLE in that we require the logic to reason about infinite traces (and/or trees) in addition to finite ones.

5.1 Syntax

Figure 7 provides a concise syntactic definition of $\mathcal{E}$-tl.

$$
\begin{array}{lcl}
D & ::= & R^* \\
R & ::= & \{\textbf{max} \mid \textbf{min}\}\ N(x_1, \ldots, x_n) = F \\
P & ::= & \textit{atomic proposition} \\
F & ::= & \textbf{true} \mid \textbf{false} \mid P \mid x_i \mid \\
 & & \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid F_1 \leftrightarrow F_2 \mid \\
 & & \bigcirc F \mid \bigodot F \mid \\
 & & \mathbf{X}_e F \mid \mathbf{Y}_e F \mid \mathbf{X} F \mid \mathbf{Y} F \mid \\
 & & \langle e \rangle\ F \mid \\
 & & N(F_1, \ldots, F_n)
\end{array}
$$

Figure 7. Syntax of $\mathcal{E}$-tl.

Formulas of $\mathcal{E}$-tl are constructed in the context of rule schemata declarations. The rule schemata provide the capability to introduce user-defined temporal modalities. The body of a rule definition is an $\mathcal{E}$-tl formula. Rules can be passed formulas as arguments and may be recursively defined. For example, one might define the rule schema

$$\textbf{max}\ \Box(\varphi) = \varphi \wedge \bigcirc\Box(\varphi)$$

and then, informally speaking, for proposition p, the formula $\Box(p)$ will hold if and only if p holds now and in every next local computation step.

$\mathcal{E}$-tl also allows mutually recursive sets of rule schemata definitions, e.g.

$$
\begin{array}{l}
\textbf{max}\ \mathtt{A}(f, g) = f \wedge \bigcirc\mathtt{A}(f, g) \vee g \wedge \bigcirc\mathtt{B}(f, g) \\
\textbf{min}\ \mathtt{B}(f, g) = g \wedge \bigcirc\mathtt{B}(f, g) \vee f \wedge \bigcirc\mathtt{A}(f, g)
\end{array}
$$

The arguments of each rule in the set of mutually recursive definitions must be identical.

An atomic formula is either one of the propositional constants TRUE and FALSE, or it is an *atomic proposition*, i.e. a propositional variables p, q, r, etc., or a predicate formulas, $P(d_1, \ldots, d_n)$, over expressions d_1 to d_n, etc., or a rule argument x_i.

Propositional combinations of formulas are constructed in the usual way using the standard propositional connectives.

Formulas can be built using the local temporal modality, $\bigcirc f$, informally meaning that f holds in the next local state of the computation. The past time mirror of next, i.e. $\bigodot f$, means that in the immediately preceding local state of the computation f held. EAGLE also has primitive modalities for the "concatenation" and "sequential composition" (overlapping concatenation) of two formulas, as in $f_1 \cdot f_2$ and $f_1;\ f_2$; for the present we omit such intervaloriented primitives from $\mathcal{E}$-tl more for simplicity of presentation than for any other reason.

Using the global temporal modality, the formula $\mathbf{X}_e f$ holds true if the system evolves under event e to a new system for which f holds true. The logic is equipped with the past time mirror, i.e. $\mathbf{Y}_e f$ holds true if the system has just evolved under e and previously f held.

Formulas can also be built using the choice modality. For example, the formula $<e>$ TRUE signifies, if true, that there was a possible evolution labelled by event e.

Finally, formulas can be built using the anonymous evolution modality, $\mathbf{X}f$, meaning that the system performs some arbitrary evolution after which the formula f holds. Similarly, we have the past time mirror formula $\mathbf{Y}f$ meaning that the system has just evolved, in some arbitrary way, with the formula f holding immediately prior to the evolution step. Note that for known, finite, sets of possible types of evolution, $\mathbf{X}$ and $\mathbf{Y}$ are definable using $\mathbf{X}_e$ and $\mathbf{Y}_e$, respectively.

5.2 Semantics

To simplify the presentation of the semantics of $\mathcal{E}$-tl, we define a core language, $\mathcal{E}$-tl$^{\texttt{min}}$, from which the full logic can defined. The principle simplification is the removal of rule schema and replacement of freely occurring applications of user-defined rule schema by a more compact anonymous fixpoint form, $\mu x.\chi(x)$ denoting the minimal fixpoint solution to the equation $x = \chi(x)$ for temporal formula χ free in x. The maximal solution can be denoted by $\nu x.\chi(x)$, however, we will rely on the duality between the minimal and maximal forms, namely

$$\nu x.\chi(x) = \neg\mu x.\neg\chi(\neg x)$$

and, formally, only use the minimal form in $\mathcal{E}$-tl$^{\texttt{min}}$.

As an example, in the context of the two rule schemata:

$$\begin{array}{l}\mathbf{min}\ \texttt{Eventually}(f) = f \vee \bigcirc\texttt{Eventually}(f) \vee \mathbf{X}\texttt{Eventually}(f)\\ \mathbf{max}\ \texttt{Always}(f) = f \wedge \bigcirc\texttt{Always}(f) \wedge \mathbf{X}\texttt{Always}(f)\end{array}$$

a free application of $\texttt{Eventually}(p)$ would be replaced by the instantiated anonymous fixpoint representation

$$\mu y.p \vee \bigcirc y \vee \mathbf{X}y$$

whereas a free application of $\texttt{Always}(p)$ would be replaced by the instantiated anonymous fixpoint representation

$$\nu x.p \wedge \bigcirc x \wedge \mathbf{X}x$$

which, when written using the dual (minimal) fixpoint form, becomes

$$\neg\mu x.\neg p \vee \neg \bigcirc \neg x \vee \neg\mathbf{X}\neg x$$

Again, for compactness in presentation of these fixpoint formula, we will assume both maximal and minimal forms, even though the $\mathcal{E}$-tl$^{\mathtt{min}}$ has only the minimal form. Also remember that when interpreting next-time modalities over both finite traces, the usual equivalence

$$\bigcirc\phi \Leftrightarrow \neg \bigcirc \neg\phi$$

is no longer valid, i.e. it fails to hold in the last element of the trace. The mirror situation holds for the previous modality, $\odot$.

Now consider a pair of mutually recursive rule definitions, such as the ones given earlier, namely:

$$\begin{array}{l}\textbf{max}\ \mathtt{A}(f,g) = f \wedge \bigcirc\mathtt{A}(f,g) \vee g \wedge \bigcirc\mathtt{B}(f,g)\\ \textbf{min}\ \mathtt{B}(f,g) = g \wedge \bigcirc\mathtt{B}(f,g) \vee f \wedge \bigcirc\mathtt{A}(f,g).\end{array}$$

A free application of $\mathtt{A}(p,q)$ would be replaced by the fixpoint form

$$\nu x.p \wedge \bigcirc x \vee q \wedge \bigcirc(\mu y.q \wedge \bigcirc y \vee p \wedge \bigcirc x)$$

A free application of $\mathtt{B}(p,q)$ would be replaced by the fixpoint form

$$\mu y.q \wedge \bigcirc y \vee p \wedge \bigcirc(\nu x.p \wedge \bigcirc x \vee q \wedge \bigcirc y)$$

which, when written using just minimal fixpoint forms, becomes

$$\mu y.q \wedge \bigcirc y \vee p \wedge \bigcirc\neg(\mu x.(\neg p \vee \neg \bigcirc \neg x) \wedge (\neg q \vee \neg \bigcirc y)).$$

DEFINITION 4. An *anonymous* $\mathcal{E}$-tl formula φ' of φ is the formula obtained from φ by replacing all free applications to rule definitions by their equivalent anonymous fixpoint representations.

Next, we provide a few necessary preliminary definitions on finite and infinite sequences, define the linear paths on which $\mathcal{E}$-tl$^{\mathtt{min}}$ formula are interpreted, then provide an inductively defined semantic interpretation.

DEFINITION 5. For any set H, let $H^\dagger$ denote the set of finite and infinite sequences of elements of H, i.e. $H^\dagger = H^* \cup H^\omega$. Indexing of sequences starts from 0. Given a sequence ρ, $\rho(i)$ denotes the i-th element of ρ, $\rho(0..i)$ denotes the initial prefix of ρ up to and including the element at index i. For finite sequences ρ, i.e. $\rho \in H^*$ for some H, $|\rho|$ denotes the length of the sequence.

DEFINITION 6. Assume $\sigma : S^\dagger$. A *local path* of a Kripke structure $K = (S, S_0, T)$ is a sequence of states

$$\sigma = s_0, s_1, \ldots, s_i, s_{i+1}, \ldots$$

such that $(s_i, s_{i+1}) \in T$ with $i \in \mathbb{N}$ for infinite sequences and $0 \leqslant i < |\sigma| - 1$ for finite sequences.

DEFINITION 7. A local path σ of a Kripke structure $k = (S, S_0, T)$ is said to be *rooted* in k if $\sigma(0) \in S_0$.

DEFINITION 8. Assume $\pi : (K \times S^\dagger \times \Lambda)^\dagger$. A *global path* in an E-structure $E = (K, K_0, R)$ from $k \in K$ is a sequence

$$\pi = (k_0, \sigma_0, e_0), (k_1, \sigma_1, e_1), \ldots, (k_i, \sigma_i, e_i), \ldots$$

such that: (i) $k = k_0$; (ii) for each index i of π, σ_i is a rooted local path of k_i; and (iii) for all pairs of indices, i and $i+1$, $((k_i, \sigma_i), e_i, (k_{i+1}, \sigma_{i+1})) \in R$.

If π is of infinite length then each σ_i is an element of S^*, i.e. of finite length. If π is of finite length then for each i, $0 \leqslant i < |\pi| - 1$, $\sigma_i \in S^*$, i.e. is of finite length, but we have $\sigma_{|\pi|-1} \in S^\dagger$ with $e_{|\pi|-1}$ being empty.

For notational convenience, we assume that the index i of a global path π can be applied componentwise.

DEFINITION 9. A global path π of E-structure E, which starts with (k_0, σ_0, e_0), is said to be *rooted* in E if and only if $k_0 \in K_0$.

DEFINITION 10. Given an E-structure E and a rooted global path π, we say that an $\mathcal{E}$-tl$^{\texttt{min}}$ formula ϕ holds on π if and only if $E, \pi, 0, 0 \models \phi$, where, for global and local path indices i and j, $E, \pi, i, j \models \phi$ is defined (inductively) as in Figure 8.

DEFINITION 11. An $\mathcal{E}$-tl$^{\texttt{min}}$ formula ϕ is said to be true in an E-structure E if and only ϕ holds for all rooted global paths of E.

5.3 Examples

We now provide a few examples of property specification using $\mathcal{E}$-tl. We continue with the abstracted evolvable buffer example of an E-structure

Given E-structure E and global path π, indices $i \in ind(\pi)$ and $j \in ind(\sigma_i)$, and an $\mathcal{E}$-tl$^{\texttt{min}}$ formula ϕ we have:-

$$
\begin{array}{lll}
E,\pi,i,j \models \textsc{true} & & \\
E,\pi,i,j \not\models \textsc{false} & & \\
E,\pi,i,j \models P & \textbf{iff} & I(\pi_i, P) \text{ — for global symbol } P \\
E,\pi,i,j \models p & \textbf{iff} & I_{\pi_i}(\sigma_i(j), p) \text{ — for local symbol } p \\
E,\pi,i,j \models \neg\phi & \textbf{iff} & E,\pi,i,j \not\models \phi \\
E,\pi,i,j \models \phi \wedge \psi & \textbf{iff} & E,\pi,i,j \models \phi \text{ and } E,\pi,i,j \models \psi \\
E,\pi,i,j \models \phi \vee \psi & \textbf{iff} & E,\pi,i,j \models \phi \text{ or } E,\pi,i,j \models \psi \\
E,\pi,i,j \models \bigcirc\phi & \textbf{iff} & j+1 \in ind(\sigma_i) \text{ and } E,\pi,i,j+1 \models \phi \\
E,\pi,i,j \models \mathbf{X}_e\phi & \textbf{iff} & e = e_i \text{ and } |\sigma_i| = j+1 \text{ and } E,\pi,i+1,0 \models \phi \\
E,\pi,i,j \models <e>\ \phi & \textbf{iff} & \text{for some path } \pi' \text{ s.t. } ((k_i,\sigma_i(j)), e, (k'_0, \sigma'_0(0))) \in R \\
 & & \text{and } E, \pi(0..i) \frown \pi', i+1, 0 \models \phi \\
E,\pi,i,j \models \mathbf{X}\phi & \textbf{iff} & E,\pi,i,j \models \mathbf{X}_e \text{ for any } e \\
E,\pi,i,j \models \bigodot\phi & \textbf{iff} & j-1 \in ind(\sigma_i) \text{ and } E,\pi,i,j-1 \models \phi \\
E,\pi,i,j \models \mathbf{Y}_e\phi & \textbf{iff} & i > 0 \text{ and } e = e_{i-1} \text{ and } j = 0 \text{ and } E,\pi,i-1,|\sigma_{i-1}| \models \phi \\
E,\pi,i,j \models \mathbf{Y}\phi & \textbf{iff} & E,\pi,i,j \models \mathbf{Y}_e \text{ for some } e \\
E,\pi,i,j \models \mu x.\chi(x) & \textbf{iff} & E,\pi,i,j \models \chi^{\alpha}_{\vee}(\textsc{false}) \text{ for some ordinal } \alpha \\
E,\pi,i,j \models \chi^{\alpha}_{\vee}(x) & \textbf{iff} & E,\pi,i,j \models \chi^{\beta}_{\vee}(x) \text{ for some } \beta < \alpha, \text{ for limiting ordinal } \alpha \\
E,\pi,i,j \models \chi^{\alpha}_{\vee}(x) & \textbf{iff} & E,\pi,i,j \models \chi(\chi^{\alpha-1}_{\vee}(x)), \text{ for non-limiting ordinal } \alpha
\end{array}
$$

Figure 8. $\mathcal{E}$-tl$^{\texttt{min}}$ interpretation on a global path

presented earlier in section 4.2. For convenience, we repeat Figure 5 as Figure 9 below, presenting a graphical representation of an E-structure and an interpretation of abstract propositions. As explained previously, this system has been abstracted from the following evolvable system. We assume there is an initial buffer of fixed, finite, size. A evolver process observes the buffer behaviour and if it sees it nearly full, or indeed full, too often, it triggers an evolution. The evolved buffer is also finite, however, it may be triggered either to evolve to a buffer of larger capacity, or, when empty, to contract in capacity. The initial buffer and all its evolutions are abstracted by the predicates $full$ and $empty$. The buffer is abstracted to be in one of three states: empty; neither empty nor full; full. The initial buffer is represented by the E-structure element k_0. All evolved buffers of greater fixed capacity are then represented by k_1. The thick dashed directed edges,

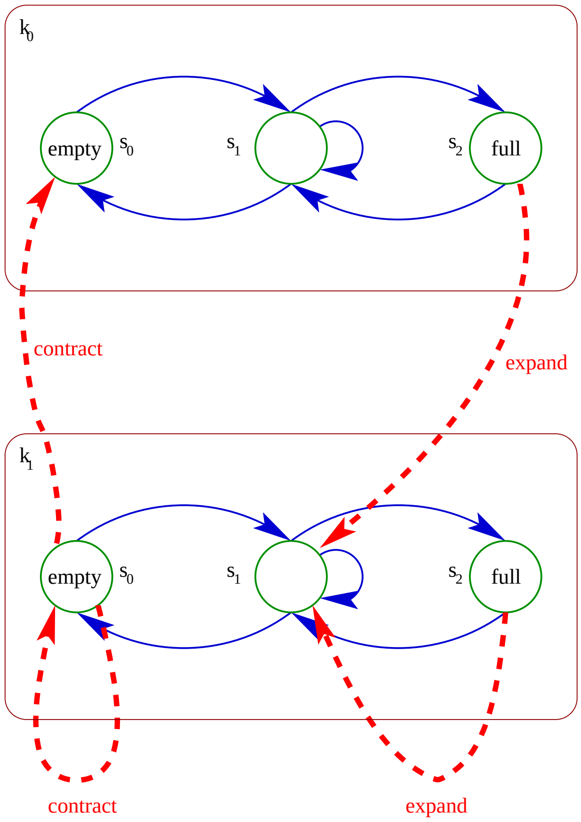

Figure 9. Abstraction of an Evolvable Buffer

labelled by either *contract* or *expand* denote evolution steps of the system. The thinner solid edges denote local transitions of the buffer between the three abstracted states.
There are two local system propositions that denote whether the buffer is full or whether it is empty. The interpretation of these propositions is given formally be the following mapping.

$$I_{k_i} = \{s_0 \mapsto \{empty\}, s_1 \mapsto \{\}, s_2 \mapsto \{full\}\} \qquad \text{for } i = 0, 1$$

A finite local path of the Kripke structure k_0 might be

$$\sigma = s_0, s_1, s_1, s_1, s_0, s_2, s_1, s_2.$$

but since the Kripke structure k_1 has a different initial state set, the following is a finite local path of k_1 but not for k_0

$$\sigma = s_1, s_1, s_1, s_0, s_2, s_1, s_2.$$

A finite global path of the E-structure E might be

$$\pi_1 = ((k_0, (s_0, s_1, s_2), \texttt{expand}), (k_1, (s_1, s_1, s_1, s_1, s_2, s_1, s_1, s_1, s_1), \texttt{contract})),$$

whereas an infinite global path of E might be

$$\pi_2 = ((k_0, (s_0, s_1, s_2), \texttt{expand}), ((k_1, (s_1, s_2), \texttt{expand}), ((k_1, (s_1, s_2), \texttt{expand}), \ldots$$

Let us now consider some properties of this system. First assume the following rule schema definitions for a localised linear time temporal modalities, unless, until, always and eventually in the future.

$$\begin{array}{l}
\textbf{max } \texttt{Unless}(f,g) = g \vee (f \wedge \bigcirc\texttt{Unless}(f,g)) \\
\textbf{min } \texttt{Until}(f,g) = g \vee (f \wedge \bigcirc\texttt{Until}(f,g)) \\
\textbf{max } \Box(f) = f \wedge \bigcirc\Box(f) \\
\textbf{min } \Diamond(f) = f \vee \bigcirc\Diamond(f)
\end{array}$$

We also the rule schema definitions for a global path always and eventually in the future, and for all evolutions, respectively some evolution.

$$\begin{array}{l}
\textbf{max } \texttt{Always}(f) = f \wedge (\bigcirc\texttt{Always}(f) \vee \mathbf{X}\texttt{Always}(f)) \\
\textbf{min } \texttt{Eventually}(f) = f \vee (\bigcirc\texttt{Eventually}(f) \vee \mathbf{X}\texttt{Eventually}(f)) \\
\textbf{max } \texttt{AllEvolutions}(f) = f \wedge (\texttt{Unless}(\textbf{true}, \mathbf{X}\texttt{AllEvolutions}(f))) \\
\textbf{min } \texttt{SomeEvolution}(f) = f \vee (\texttt{Until}(\textbf{true}, \mathbf{X}\texttt{SomeEvolution}(f)))
\end{array}$$

The formula $\texttt{AllEvolutions}(f)$ thus holds at some state s when the argument formula f holds on s and at the start of all subsequent evolutions (of the system).

Property Example 1: As a first, somewhat trivial, property, let us formulate that whenever the buffer is empty, its next local state, if one exists, must be neither empty nor full empty.

$$f_1 = \Box((\mathit{empty} \wedge \bigcirc\text{TRUE}) \Rightarrow \bigcirc(\neg \mathit{empty} \wedge \neg \mathit{full}))$$

It is straightforward to determine that this property is true for the given example E-structure. Consider any rooted global path π. $\Box(\phi)$ will be true on π if ϕ is true for each of the local states of the first trace σ_0 of π. By inspection of k_0 from which σ_0 is derived, a state satisfying the *empty* proposition (i.e. s_0) will either be locally terminal or followed by a state of k_0 in which neither *empty* nor *full* propositions hold, i.e. s_1.

Note that because local paths may be finite and the semantics of $\bigcirc$ is defined to be existential, i.e. there is a next local state, our formulation had to be guarded by the existence of a local next state. An alternative formulation can be given by noting that $\neg \bigcirc \neg\phi$ will evaluate to true at an end state of some finite local state trace. Thus f_1 could be re-written as

$$f_1 = \Box(\mathit{empty} \Rightarrow \neg \bigcirc \neg(\neg \mathit{empty} \wedge \neg \mathit{full}))$$

If we wish to express the fact that f_1 should be true for all evolutions of the buffer system, then we could write any of the following.

$$\begin{aligned}
f_{1a} &= \texttt{Always}(\mathit{empty} \Rightarrow \neg \bigcirc \neg(\neg \mathit{empty} \wedge \neg \mathit{full}))\\
f_{1b} &= \texttt{Always}(\Box(\mathit{empty} \Rightarrow \neg \bigcirc \neg(\neg \mathit{empty} \wedge \neg \mathit{full})))\\
f_{1c} &= \texttt{AllEvolutions}(\Box(\mathit{empty} \Rightarrow \neg \bigcirc \neg(\neg \mathit{empty} \wedge \neg \mathit{full})))
\end{aligned}$$

Property Example 2: Consider the property that whenever the buffer is full, an evolutionary expansion must occur and leave the buffer in a non-full state.

$$f_2 = \texttt{Always}(\mathit{full} \Rightarrow \mathbf{X}_{\texttt{expand}} \neg \mathit{full})$$

Whilst there are global paths derivable from the given E-structure that satisfy the formula f_2, this property clearly doesn't hold for all rooted global paths. Both Kripke structures k_0 and k_1 contain local transitions from a full state to a non-full state, corresponding to the consumption of an item from the buffer.

Consider, however, the following formula.

$$f_{2a} = \texttt{Always}(\mathit{full} \Rightarrow <\texttt{expand}> \neg \mathit{full})$$

This formula specifies that whenever the system is in a full state there is a choice of undertaking an `expand` evolution to a non-full state in the evolved system. There is no guarantee that the expansion will occur, it is just a possibility.

Property Example 3: As a further straightforward constraint, the following formula characterizes the property that a buffer will always make room for further input items.

$$f_3 = \texttt{Always}(\mathit{full} \Rightarrow (\bigcirc \neg \mathit{full} \vee \mathbf{X}_{\texttt{expand}} \neg \mathit{full}))$$

In other words, the buffer will never stop in a full state.

Property Example 4: As a form of fairness constraint on the buffering system, we might wish to express that a buffer can't locally cycle infinitely through its full state. Under such a constraint, a buffer will locally (i) operate only finitely and then terminate, or (ii) evolve after some finite duration, or (iii) cycle infinitely but not through a full state.

$$f_4 = \texttt{Always}(\neg\Box(\bigcirc\Diamond(full)))$$

Again, note that this property is not true for the given E-structure, however, the formula might be used as a fairness constraint on the given model behaviour.

Property Example 5: Let us now try to specify logically when it is possible for a buffer to contract in size. Our model, as depicted in the E-structure of Figure 9 shows that this is possible when the buffer is empty, but only when the buffer is in some expanded state. The following formula

$$\texttt{Always}(empty \Rightarrow <\texttt{contract}> empty)$$

clearly doesn't hold; no contraction is possible in the initial kripke structure k_0. The premise *empty* is too weak. How can this be strengthened to uniquely characterize the empty state of k_0? For our particular concrete model, there are two ways ahead. The easiest is to introduce a global proposition, say *initialbuffer*, which is an abstraction of the size of the buffer at the concrete model level, and is true only in k_0.

$$\texttt{Always}(empty \wedge \neg initialbuffer \Rightarrow <\texttt{contract}> empty)$$

An alternative approach is to write a formula that characterises whether the numbers of expansion and contraction evolutions the system has undertaken so far are equal, which, because of the concrete interpretations of expand and contract, will mean that the buffer is in its initial size configuration. $\mathcal{E}$-tl as defined above is not expressive enough to do this, although the inclusion of the cut (interval) modality, or data parametrization of rule names, will provide sufficient expressivity, as in the EAGLE logic. However, this is not a general solution.

Property Example 6: Although we have, so far, specified properties of the evolvable buffer at an abstract level, $\mathcal{E}$-tl can equally be used for property specification at the concrete level. For example, one would wish to state that after an evolution expansion or contraction of the buffer, the contents remained the same. Clearly, we must assume a standard first order extension of $\mathcal{E}$-tl to handle predicates and quantification over data.

$$f_{6a} = \texttt{Always}(\forall s.contents(s) \wedge \mathbf{X}\textsc{true} \Rightarrow \mathbf{X}contents(s))$$

Furthermore, if we assume a predicate $size(m)$ that is true if and only the buffer is of size m, then we can now specify that an `expand` evolution will indeed expand the buffer, and similarly for `contract`..

$$\begin{aligned} f_{6b} = \texttt{Always}(\forall m.\exists n.size(m) \wedge \mathbf{X}_{\texttt{expand}}\mathrm{TRUE} \\ \Rightarrow \mathbf{X}_{\texttt{expand}}(size(n) \wedge n < m)) \\ f_{6c} = \texttt{Always}(\forall m.\exists n.size(m) \wedge \mathbf{X}_{\texttt{contract}}\mathrm{TRUE} \\ \Rightarrow \mathbf{X}_{\texttt{contract}}(size(n) \wedge n > m)) \end{aligned}$$

5.4 Specifying an Evolver

Our motivational Java implementation of an evolvable buffer used the idea of an "evolver" thread of control that monitored, on-line, the behaviour of the buffer and forced an evolution, either an expansion or contraction, when particular conditions were fulfilled (determined via probe methods). The examples given in the above subsection are of *declarative* specifications that can be checked against complete system behaviours, for example the E-structure of Figure 9. Here we very briefly indicate how our logic $\mathcal{E}$-tl can be used directly for the executable specification of an evolver process. Our approach adopts the imperative view of temporal logic that we developed jointly with Gabbay in MetateM and, in a sense, extends the run-time monitoring approach of Eagle in the obvious way with the inclusion of action.

An evolver is specified by a set of temporal conditional evolution rules of the form

$$\texttt{Always}(\textit{condition} \Rightarrow \textit{evolution})$$

where the *condition* is treated *declaratively* but the *evolution* is treated *imperatively*. The *condition* should be viewed as a property that is to be monitored on the system. Whenever it holds, the *evolution* formula determines how the system must evolve, in other words, the system under the watchful eye of the evolver is stopped, evolved in the way specified and then re-started.

Here are some simple examples of rules for specifying evolvable buffering.

Example 1: Whenever the buffer is full, it must be expanded to allow for more input. The following formula achieves this.

$$\texttt{Always}(full \Rightarrow \mathbf{X}_{\texttt{expand}}\neg full)$$

Let's analyze what happens in the imperative, or executable, view of the logic. At each monitoring step the premises of the rules are checked to determine which evolution actions are to be undertaken. Thus, when the proposition $full$ is observed to be true, the execution engine for the logic constructs a future to satisfy the consequents of the triggered rules. In this case, an `expand` action must be undertaken. If one assumes that this rule is the only rule for determining when `expand` occurs, the imperative view ensures that expansion can only occur under these conditions; this is, of course, rather different to the declarative interpretation of temporal logic that would allow models where $\mathbf{X}_{\texttt{expand}}\neg full$ to be true with $full$ true prior to the evolution.

Example 2: Of course, the rule above is undoubtedly too strong. A weaker version would be to require (i.e. force) an expansion as soon as the buffer has been full twice.

$$\texttt{Always}(full \wedge \texttt{HasBeen}(full) \Rightarrow \mathbf{X}_{\texttt{expand}}\neg full)$$

in the context of the $\mathcal{E}$-tl rule definition

$$\textbf{min}\ \texttt{HasBeen}(f) = \odot f \vee \odot\texttt{HasBeen}(f)$$

Thus, as soon as the premise is observed to be true, the right hand side of the rule is fired. The premise will be first observed to be true the second time that $full$ is seen to be true and so an expansion is executed there and then. Again, this is very different to the declarative view of the temporal rule that would accept models where the evolution occurred even when there had been no full states!

Example 3: The EAGLE logic allows rule schema names to be parameterized by data as well as by EAGLE formulas. Assume such an extension to the $\mathcal{E}$-tl logic. One could then define a rule for counting occurrences of formulas holding over the past.

$$\begin{aligned}\textbf{min}\ \texttt{Count}(f,n) = \ & (\neg\odot\text{TRUE} \wedge n == 0 \wedge \neg f) \vee \\ & (\neg\odot\text{TRUE} \wedge n == 1 \wedge f) \vee \\ & (\odot\text{TRUE} \wedge \neg f \wedge \odot\texttt{Count}(f,n)) \vee \\ & (\odot\text{TRUE} \wedge f \wedge \odot\texttt{Count}(f,n-1))\end{aligned}$$

The formula $\texttt{Count}(full,n)$ will be true at some state if there have been precisely n occurrences of the buffer in the full state of its current size or configuration. This could therefore be used to trigger when an expansion occurs, for example:

$$\texttt{Always}(full \wedge \odot\texttt{Count}(full,10) \Rightarrow \mathbf{X}_{\texttt{expand}}\neg full)$$

6 Conclusions

This is very much a preliminary foray into the area, opening up a wide range of possibilities and applications, and raising many issues relating to the description and analysis of adaptive and evolutionary computational systems.

The aspect that distinguishes evolvable systems from more general transition systems is that of 'constancy through change'. It is not sufficient that at each evolutionary step part of the structure (albeit possibly re-represented) is preserved. As in the parlour game of 'Chinese whispers', this may lead to little or nothing being preserved overall. Inherent in the notion of evolution is the idea of constancy and the more that is preserved globally the more we can say about the overall behaviour of the system. In keeping with the logical approach of this paper, we have used predicate abstraction to identify some of the common structure preserved through evolutionary steps. It is not clear that this is the final answer, and perhaps a more algebraic approach to this aspect of evolution may serve equally well or better in identifying common structure and allowing us to model the overall behaviour of evolvable systems in terms of their components.

It is the role of the evolver process, not only to monitor the behaviour of local systems, and to evoke and compute evolutionary steps, but also to maintain invariant properties. The use of temporal logic to describe evolver processes opens up the possibility, through executable temporal specifications, of not only verifying existing evolver processes or guiding the design of such processes, but also in actually generating prototype evolvers. This aspect of the treatment above requires further understanding, especially in the relationship of logical description to execution and the models we have adopted.

We have used a simple illustrative example of a buffering system in this paper. Adaptive and evolutionary systems are of widespread occurrence. For example, hybrid systems (incorporating both discrete and continuous effects) are often evolvable, where the physical environment provides a changing context which may require adaptive and evolutionary behaviour of a system. Again, businesses and business processes are naturally adaptive evolvable systems changing according to external factors or internal drivers for change. We intent to investigate to what extent the techniques of this paper capture the behaviour of these more complex systems.

Finally, one of the key motivations for this work is to investigate runtime monitoring and model checking for evolvable systems. To this end, we have used an Eagle-like approach to temporal logic so that we may, in future work, implement and run temporal descriptions to both statically check evolvable systems and to provide run-time monitors.

BIBLIOGRAPHY

[1] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Michael R. Lowry, Corina S. Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser

and Richard Washington. Combining tset case generation and runtime verification. *Theor. Comput. Sci.* 336(2–3): 209–234. 2005.

[2] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. MetateM: An introduction. *Formal Aspects of Computing*, 7(5): 533–549, 1995.

[3] H. Barringer, M. Fisher, D. Gabbay, R. Owens and M. Reynolds. *The Imperative Future: Principles of Executable Temporal Logic*. Research Studies Press. 1996.

[4] Howard Barringer, Allen Goldberg, Klaus Havelund, Koushik Sen. Rule-Based Runtime Verification. Proceedings of the *VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract interpretation*, Venice. Volume 2937, Lecture Notes in Computer Science, Springer-Verlag. 2004.

[5] Carlos Caleiro, Amílcar Sernadas and Christina Sernadas. Fibring Logics: Past, Present and Future. *This volume.*

[6] P.Y. Cunin, R.M. Greenwood, L. Francou, I. Roberston and B.C. Warboys. The PIE Methodology - Concept and Application. *EWSPT 2001 Proceedings 8th European Software Process Technology Workshop*, LNCS 2077, 3–26. Springer Verlag. 2001.

[7] Dov. M. Gabbay. *Fibring Logics*. Oxford University Press. 1999.

[8] Dov. M. Gabbay, Odinaldo Rodrigues, and John Woods. Belief Contraction, Antiformulae and Resource Overdraft: Part I Deletion in Resource Bounded Logics. *Logic Journal of the IGPL*, 10(6): 601–652. Oxford Univ. Press. 2002

[9] R.M. Greenwood, I. Robertson, B.C. Warboys, and B.S. Yeomans. An Evolutionary Approach to Process System Development. Proceedings of the *International Process Technology Workshop*, Villard de Lans (Grenoble). 1999.

[10] R.M. Greenwood, B.C. Warboys, R. Harrison and P. Henderson. An Empirical Study of the Evolution of a Software System. Proceedings of the *13th IEEE Conference on Automated Software Engineering*, Honolulu. IEEE Computer Society Press. 1998.

[11] Amílcar Sernadas, Christina Sernadas and José Félix Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5): 603–630. 1995.

[12] Amílcar Sernadas, Christina Sernadas and Carlos Caleiro. Fibring of logics as a categorial construction. *Journal of Logic and Computation*, 9(2): 149–179. 1999.