Utilising Assumptions to Determine the WCET of Multi-component Classification Systems

Alan Burns[1(✉)] and Sanjoy Baruah[2]

[1] University of York, York, UK
alan.burns@york.ac.uk
[2] Washington University in St. Louis, St. Louis, USA

Abstract. Cyber-Physical Systems (CPS) are being used increasingly in safety-critical settings. These are often in complex and dynamic environments that require a range of sensors and AI-enabled classifiers to be employed to monitor and understand the challenges posed by such environments. For real-time CPS it is necessary to execute a range of classifiers so that: (i) the final output is delivered on time, (ii) the required level of confidence in the correctness of this output is beyond a safety-derived threshold, and (iii) the minimum level of resources is consumed in the process of meeting these timing and confidence constraints. To undertake timing analysis the Worst-Case Execution Time (WCET) of these multi-component classification systems must be estimated; doing so requires the maximum input load imposed by any potential environment to be determined. In this paper we show how this can be done by exploiting well-defined assumptions about the environment via a Dynamic Programming algorithm. Assumptions can take many forms; here we demonstrate the expressive power of the approach by including a range of illustrative examples.

1 Introduction

A key element of Cliff Jones' work [7,13,14,16–22] on the verification of concurrent software is that the assumptions that define and constrain the relationships between concurrent threads, and between these threads and the environment in which they function, are crucial and must be articulated, challenged and ultimately agreed upon and documented. This is especially true for Cyber-Physical Systems (CPS) where there is a close and mutually dependent relationship between the physical and software elements. For those CPS that are also 'Real-Time,' assumptions that limit the work to be undertaken by the software are essential if the load on the system is to be bounded and timing constraints, such as meeting deadlines, are to be satisfied. In this paper we focus on Learning Enabled multi-component Classifiers and illustrate how a range of assumptions can be accommodated within the same framework. This framework allows tight bounds on Worst-Case Execution Time (WCET) to be derived using a relatively standard Dynamic Programming technique.

The recent rapid development of Machine Learning (ML) and Deep Learning (DL) techniques has led to the widespread use of Deep Neural Nets (DNNs) within autonomous resource-constrained CPS. One of their primary applications is to undertake classification exercises. Here a complex chain of DNNs is used to 'understand' the

A. Cavalcanti and J. Baxter (Eds.): *The Practice of Formal Methods*, LNCS 14780, pp. 46–64, 2024.
https://doi.org/10.1007/978-3-031-66676-6_3

dynamic environment within which the CPS is operating [24]. Many of these systems are employed (or are been considered for employment) in safety-critical applications and require accurate predictions to be delivered in real time using limited computing resources (this is sometimes called "edge AI" where the efficient execution of machine intelligence algorithms on embedded edge devices is required [8,28]). For example, an autonomous mobile systems (AMS) such as a vehicle or robot must repeatedly check that its designated path is free of hazards. Each such check must be completed by a deadline (that may be a function of the current speed of the AMS). For each check, the AMS may apply one or several available classifiers that read in input from different sources (e.g., cameras, lasers, LiDAR, radars, microphones and even satellite feeds) to evaluate the designated path.

To deploy such multi-component classifiers within time-critical CPS it is necessary to model the worst-case load being placed on the system by the environment in which it is being deployed. In this paper we model such workloads and resource-usage specifications as a contract between *assumptions* ($\mathcal{A}$) (or *rely conditions*) and *obligations* ($\mathcal{O}$) (or *commitments* and *gurantees*) [5,9,15,16,23]: if the system behaves according to the assumptions then the obligations shall be delivered. For example, if each classifier executes for no more than a computed Worst-Case Execution Time (WCET) and there are no more than a specified number of objects to classify then the classification of all objects will be completed by the specified deadlines.

The rest of this paper is organised as follows. The following section provides straightforward definitions of Class and Classifiers. It also introduces invariant assumptions. The analysis framework is presented via an extended example which is introduced in Sect. 3. Section 4 develops the Dynamic Program that is used to determine the WCET of a complete set of classifiers; the code implementing this analysis tool is outlined in the Appendix. Section 5 considers a collection of more expressive assumptions and demonstrates how the Dynamic Program is adapted. One issue that the analysis framework can address is how to optimise the order in which the classifiers are executed, this is discussed in Sect. 6. Section 7 considers some open issues and Sect. 8 presents some conclusions.

The material presented in the current paper is a significant extension of an earlier conference paper [5].

2 Classes, Classifiers and Basic Invariant Assumptions

We assume that a single input I to the classification system consists of N RoIs (Regions of Interest), each of which contains an object $(\delta_1, \ldots, \delta_N)$ which is a member of one or more class hierarchies (Ωs). For example

$$\Omega_{Pet} \stackrel{\text{def}}{=} \{Dog, Cat\}$$

$$\Omega_{Colour} \stackrel{\text{def}}{=} \{Black, White, Ginger, \ldots\}$$

$$\Omega_{Dog} \stackrel{\text{def}}{=} \{Poodle, Dalmatian, Labrador, \ldots\}$$

$$\Omega_{Cat} \stackrel{\text{def}}{=} \{Persian, Siamese, Sphynx, Burmese, \ldots\}$$

With these definitions, Pet is an example of a binary class, and Dog a multi-class; with Dog being a subclass of Pet (and Pet a superclass of Dog). An object of class Dog is also of class Pet. An object that is a member of more than one class hierarchy is usually referred to as being multi-labelled [25]; e.g. an object that is classified to be a $Ginger\ Labrador$.

The number of objects in I with class X ($\in \Omega_X$) is denoted by N_X; for example, $N_{Pet}, N_{Dog}, N_{Poodle}$. Clearly there are some obvious relations between these values, e.g. $N_{Pet} = N$; $N_{Poodle} \leq N_{Dog} \leq N$; $N_{Dog} + N_{Cat} = N$.

The work that the system must be capable of managing is defined by the predicate $\mathcal{A}$ that captures the assumptions made about the environment in which the system is deployed. At a minimum, $\mathcal{A}$ must bound N (i.e. $N \leq N^{\max}$). It may also define bounds for individual classes ($\forall X \in \Omega_X : \ N_X \leq N_X^{\max}$). For example, the assumptions that there are at most ten pets with a maximum of eight Cats and seven Dogs[1]:

$$\mathcal{A} \stackrel{\text{def}}{=} N \leq 10 \wedge N_c \leq 8 \wedge N_d \leq 7 \tag{1}$$

This is an example of a simple invariant assumption. The assumption is true before classification commences, and it remains true after each RoI is processed.

As noted in the introduction, a complete classification system may involve a collection (or cascade) of individual classifiers. Each classifier aims to output a single object that is a subclass of the input's class. However, some classifiers may output an ordered list of the most likely classes, and this list may have probabilities assigned.

Where a classifier is unable to output a subclass with sufficient confidence, the output class may be augmented by the addition of classes IDK or U. A classifier that is unable to determine the right subclass of the input may output IDK to imply "I Don't Know" [1,11,27]. Alternatively, a classifier that is confident that the input is not of the right class (and hence an appropriate subclass cannot be derived) may output U to signal 'unclassified' or 'unclassifiable.'

A complete classification system incorporates a set of M class specific classifiers, $K_1, K_2, \ldots, K_M$. Each K takes as input a RoI that is either unclassified (of class U) or partially classified (is of class Ω_{in}). It outputs a member of class Ω_{out}. Where there is an input class, Ω_{out} will be a subclass of Ω_{in}. Classifiers are linked so that the output of one classifier may be used as input to another (this is illustrated with the example in Sect. 3 below). We assume that each operation of the classifier K_i has a worst-case execution time of E_i.

As classifiers based on DNNs are not 100% accurate a classifier may classify an object incorrectly. If this incorrect classification is an output of the full system then this will impact on the effectiveness, and even safety, of the process, but it will not effect the worst-case execution time of the actual classification system. However, if this misclassification impacts on the internal flow of objects between the individual classifiers then it is likely to lead to extended paths through the system and hence higher worst-case execution times. In order to bound this worst-case behaviour it is necessary to define a *fault model* that explicitly states the maximum number of misclassifications

[1] In the following, for ease of presentation, N_{Cat} is abbreviated to N_c and N_{Dog} is abbreviated to N_d.

that can be experienced in the processing of a single input. This is discussed further, and examples given, in Sect. 5.4.

3 An Example Classifier

We illustrative our framework with a toy example[2] that concerns a *CADIS* (for *Cat And Dog Identification System*), a software component that is tasked with identifying the breeds of all the Cats and Dogs that appear in an input image – see Fig. 1.

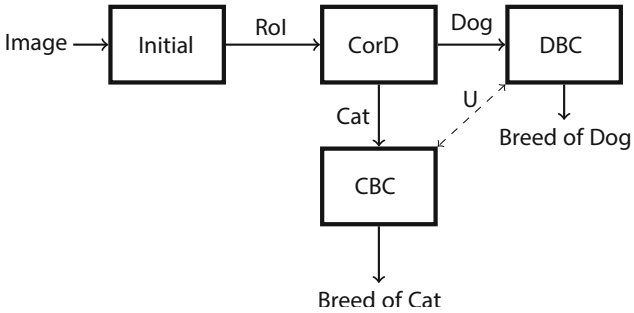

Fig. 1. CADIS – A *Cat And Dog Identification System*

Given an image, an **Initial** component first breaks it down into a number of "*boxes*" (RoIs – Regions of Interest), each of which contains an image of interest (i.e., an image of either a Cat or a Dog). Each RoI is then passed on to a binary classifier component (**CorD**) that determines if the input is a Cat or a Dog. Components (**CBC** – Cat Breed Classifier) and (**DBC** – Dog Breed Classifier) then determine the breed of each RoI.

To make the example more realistic we assume that **CorD** can fail by passing on to **DBC** (to **CBC**, respectively) a RoI that actually contains an image of a cat (dog, resp). This input will cause **DBC** (**CBC**, resp.) to output 'unclassified' (U). This output is then passed to **CBC** (to **DBC**, resp) to enable the breed of this Cat (this Dog, resp.) to be determined. We assume the Assumptions bound the maximum number of faults that need be accommodated; for example $\mathcal{A}$ might specify that $F_{CorD} \leq 2$.

The WCET for each component in our example is shown in Table 1 – these values are part of the system's Assumptions.

We initially assume that these values are constant; extensions in which they are a function of the input RoI or recent history or stakeholder in a multi-model specification are considered in Sects. 5.2 and 5.3.

From these values it follows that to classify the breed of a Cat takes 16 time units (1+5+10) and for a Dog 18 (1+5+12). If there is a fault then this increases to 28

[2] This toy example is very loosely based on an *Identify Friend or Foe* (IFF) [3] application system that uses DNN-based image processing (for example [12]) to distinguish between friendly and hostile aircraft, and may further classify each kind.

Table 1. WCET for each Component

K_i	E_i
Initial	1
CorD	5
DBC	12
CBC	10

(1+5+12+10). However the values reduce if previous RoIs have exhausted one of the top level classes. So if the maximum number of Cats has been observed then the next RoI must be a Dog; **CorD** need not be executed and the cost is now 13 (1+12). Similarly for a RoI that must be a Cat its cost is 11 (1+10).

To illustrate the steps that are undertaken to compute the total cost of an input sequence, let us suppose that our Assumptions assert that there will be at most two Dogs and at most two Cats in the sequence (and no faults in the one classifier that impacts on the flow of objects through the classification system):

$$\mathcal{A} \stackrel{\text{def}}{=} N \leq 4 \wedge N_c \leq 2 \wedge N_d \leq 2 \wedge F_{CorD} = 0$$

If the four animal images were to appear in the order ⟨D, D, C, C⟩ then the first two would each result in a WCET of 18. However, it will subsequently follow (from the Assumption predicate) that there are no more Dog images in the sequence, and hence the remaining animal images do not need to be pre-processed in the **CorD**; each would consequently only experience a WCET of 11. Summing over all four RoI's we have a duration equal to $\mathbf{58}$.

If, however, the images were to appear in the order ⟨C, D, C, D⟩ within the sequence one can easily show that only the last pet-image RoI (the last "D") would skip the first step in the **CorD**, for a total duration bound of $\mathbf{63}$.

It may be verified by exhaustive enumeration (in Sect. 4 below, we obtain a more efficient means of doing so) over all possible orderings of the two Dogs and the two Cats in the RoI sequence that ⟨C, D, C, D⟩ represents the worst case and that $\mathbf{63}$ is consequently the duration bound under the assumption that there are at most two Cat images and at most two Dog images (and no faults) in the sequence of 4 RoI's.

In the earlier example (1), which also only contains simple invariant assumptions, there are a maximum of 10 RoIs, 8 Cats and 7 Dogs (and again no faults). Clearly as Dogs are more expensive to process than Cats the worst-case is when there are 7 Dogs and hence 3 Cats. The total cost is 7*18 + 3*16 (= $\mathbf{174}$). This is delivered by a sequence such as ⟨D, D, D, D, D, D, C, C, C, D⟩.

These two simple examples show that for basic invariant assumptions (and a relatively small number of classifier components) the worst-case path through the classifiers can be determined by exhaustive enumeration. However for more complicated sets of assumptions this may not the case. In the next section we show how the use of Dynamic Programming can furnish a method of solving these WCET problems for a wide range of useful types of Assumptions.

4 Determining WCET via a Dynamic Program

In this section we will use the following slightly more complicated example to illustrate how a Dynamic Program can be used to determine the WCET of the collection of CADIS classifiers introduced above (and in Fig. 1) under the following assumptions:

$$\mathcal{A} \stackrel{\text{def}}{=} N \leq 8 \wedge N_c \leq 5 \wedge N_d \leq 6 \wedge \big(N_d > 3 \Rightarrow N_c \leq 1\big) \wedge F_{CorD} = 0$$

Now there are at most 8 Pets, of which at most 5 may be Cats and at most 6 may be Dogs; in addition, if there are more than three Dogs then there may be at most one Cat.

The first step is to derive a general recurrent function for determining the maximum duration needed to fully process an image given Assumption $\mathcal{A}$. We will do this with a bottom-up definition of the function $FC(N_c, N_d)$ that will deliver the maximum cost of the complete classification starting from the position of having already classified N_c Cats and N_d Dogs. Clearly the complete problem is solved by computing $FC(0, 0)$. The general recurrent function is simply

$$FC(N_c, N_d) = \max\Big(Cc + FC\big(N_c + 1, N_d\big),\ Dc + FC\big(N_c, N_d + 1\big)\Big)$$

where Cc is the cost of dealing with one further Cat (i.e., 16) and Dc is the cost accrued by one further Dog (i.e., 18).

At each step a comparison made between the cost of one more Cat being classified against one more Dog. The path with the highest cost is always taken. The recurrence is terminated when it is no longer possible (because of the Assumptions) to process one more Cat, or one more Dog. Let $V(N_c, N_d)$ denote a predicate that returns true if the Assumptions allow these values for the number of Cats and Dogs. We now give, in Fig. 2, the full recurrent definition of FC.

$$FC(N_c, N_d) = \begin{cases} \max\begin{pmatrix} Cc + FC(N_c + 1, N_d) \\ Cd + FC(N_c, N_d + 1) \end{pmatrix} & \text{if } V(N_c + 1, N_d) \wedge V(N_c, N_d + 1) \\ RCc + FC(N_c + 1, N_d), & \text{if } V(N_c + 1, N_d) \wedge \neg V(N_c, N_d + 1) \\ RDc + FC(N_c, N_d + 1), & \text{if } V(N_c, N_d + 1) \wedge \neg V(N_c + 1, N_d) \\ 0 & \text{otherwise} \end{cases}$$

Fig. 2. Recurrent Definition of FC

The constant RCc is the reduced cost of classifying a Cat when it is known that the RoI is not a Dog (i.e. $\neg V(N_c, N_d+1)$). This allows the component **CorD** to be skipped with the result that the cost of classification is reduced form 16 to 11. Similarly RDc has the value 13, having been reduced from 18.

The recurrence in Fig. 2 clearly demonstrates that the problem of computing $FC(0, 0)$ possesses the *optimal substructure* property (see, e.g., [10, p. 379]), and is hence amenable to solution as a Dynamic Program [2]. Notice that the recursive calls made in computing $FC(N_c, N_d)$ are to $FC(N_c+1, N_d)$ and $FC(N_c, N_d+1)$ – in both cases, one of the arguments is strictly increasing and getting closer to the upper bounds

of $N^{\max}$, $N_c^{\max}$ and $N_d^{\max}$. Clearly the resulting Dynamic Program has running time no worse that $O(N_c^{\max} \times N_d^{\max})$, implying an asymptotic complexity no worse than $\boldsymbol{O((N^{\max})^2)}$.

For the simple example introduced at the beginning of this section the V (for 'valid') predicate is easily derived from the defined Assumptions:

$$V(N_c, N_d) = N_c + N_d \leq 8 \wedge N_c \leq 5 \wedge N_d \leq 6 \wedge \neg(N_d > 3 \wedge N_c > 1)$$

In the Appendix we outline the Dynamic Program that implements this recurrence. When executed with this $V(\cdot,\cdot)$ predicate it returns the WCET calculation of $\mathbf{129}$ – this is realised upon the input sequence ⟨C, C, C, C, D, D, C, D⟩. Note that if the associated clause of the $\mathcal{A}$ is changed from $N_d > 3 \Rightarrow N_c \leq 1$ to $N_d \geq 3 \Rightarrow N_c \leq 1$ then the WCET value would reduce to $\mathbf{119}$ – from the sequence ⟨D, D, D, D, D, C, D⟩.

In the following section we consider a range of more expressive assumptions which together demonstrate the effectiveness of the proposed approach. In all of these examples the Assumptions have the dual role of constraining the size of the problem (and hence terminating the recurrence) and sanctioning optimisations that lead to reduced WCET estimates. Assumptions restrict the set of legal paths through the components, and hence enable more efficient paths to be chosen. As 'assumptions' the program does not need to check their authenticity: they are a fundamental aspect of the (ideally formal) specification of the complete classifier.

5 Extending the Model – More Expressive Assumptions

In this section we are going to look at a number of ways that the semantics of Assumptions can be extended. While individual examples are used to illustrate these extensions, we note that an arbitrary number of these can be combined within the context of a single application.

The examples considered so far employ a model defined by the costs of each operation and a function that checks for a valid operation. The costs reflect assumptions made about the RoI that is to be processed. Typically, if something about the class of the RoI is known then the cost of the operation can be reduced. In the simple example above if the RoI is known to not be of a Cat then it may be passed directly to the Dog classifier and its execution time reflects the fact that the input is definitely a Dog (unless this is an error). We re-emphesize that the assumptions are, in effect, *axioms* – they are true if the environment in which the system operates behaves correctly (as specified); and if the environment does not behave correctly then nothing need be (or indeed can be) guaranteed. More resilient systems can be built by including degraded models that have weaker assumptions and obligations. This is a form of multi-model often explored in the Mixed-Criticality literature [6,26] – we discuss this further in Sect. 5.3.

In this section we focus on the constraints and provide a number of more expressive examples.

- Incorporating arbitrary constraints on the relationships between the object classes.
- Incorporating constraints that relate to recent history.
- Incorporating non-constant execution time parameters.
- Incorporating a Fault Model.

5.1 Arbitrary Post-condition Constraints

In the examples given so far the Assumptions are simple invariants – they remain true throughout the processing of all the RoIs. Other assumptions may take the form of post-conditions: they will be true at the end of the processing but are not necessarily true during processing.

In addition to constraints concerning the number of RoIs, the maximum number of each class of RoI, and the maximum number of faults, it is possible to add further constraints that can help reduce the solution space for the algorithm. So, for example, if it is known (i.e. this is part of the specification) that if there is more than one Dog then there are always more Dogs than Cats then the Assumptions can reflect this:

$$\mathcal{A} \stackrel{\text{def}}{=} N \leq 7 \ \wedge N_c \leq 5 \ \wedge N_d \leq 3 \ \wedge \big(N_d > 1 \Rightarrow N_d > N_c\big)$$

This assumption must be true once the entire input sequence has been processed. It can be false 'during classification,' as long as correctness is delivered as an overall post-condition. The Assumptions leads to the following definition of the necessary V predicate that determines if specific numbers of Cats and Dog can lead to a correct outcome:

$$\begin{aligned} V(N_c, N_d) = {} & N_c + N_d \leq 7 \wedge N_c \leq 5 \wedge N_d \leq 3 \wedge \\ & (N_d \leq 1 \vee N_d + \min\{7 - (N_c + N_d), 3 - N_d)\} > N_c) \end{aligned}$$

The predicate is true if N_c and N_d are not too large and either there is at most one Dog or if the number of Dogs so far identified (N_d) plus the minimum that could still be in the input image is greater then the number of Cats so far identified (N_c). If this is true then there is a possible future that will satisfy the constraint and hence this is a Valid step: the assumptions remain satisfiable.

As the assumptions define the sanctioned behaviour of the outside world, they define a semi-cooperative environment. This is in contrast to an analysis in which the environment is deemed to be an adversary that will always behave in the worst possible way in order to break the behaviour of the system. Here the environment always exhibits behaviour that is sanctioned by the specified assumptions – it is nevertheless necessary to identify the worst possible behaviour (in terms of time taken to complete the classification) that is compatible with the assumptions.

For this example the Dynamic Program given in the Appendix is unchanged other than having its 'Valid' function reflect the V predicate given above. The program outputs a worst-case cost of $\mathbf{93}$ with the sequence ⟨C, C, C, C, C, D⟩. But with one less Cat (i.e. $N_c \leq 4$) in $\mathcal{A}$, the worst-case drops to $\mathbf{81}$ from the sequence ⟨C, C, D, D, D⟩. With this last sequence after the second Dog there is an equal number of Cats and Dogs. This temporarily invalidates $\mathcal{A}$; however, a fifth RoI is feasible, it can be a Dog and then the post-condition is delivered. As this final RoI cannot be a Cat if the assumptions are to be obeyed then the cost of classification is at the lower level of 13 rather than 18: a useful reduction in the WCET.

To illustrate a final really arbitrary example, consider a somewhat strange requirement that the number of Cats must be a prime number! The code for the Valid predicate is somewhat more complicated (see Appendix 2.2.1), but the Dynamic Program

is unchanged. Example runs of the program with N=100 and various values of $N_d^{\max}$ generate the outputs given in Table 2. Note the number of Cats is always a prime, but its value is reduced as more Dogs are allowed.

Table 2. Example Outputs for the 'prime number of Cats' Assumption

$N_d^{\max}$	WCET	N_C	N_D
10	1601	97	3
20	1629	83	17
30	1653	71	29
40	1673	61	39
50	1689	53	47

In Sect. 7 we give further consideration to the potential expressive power of the form of Assumptions used in this WCET analysis framework.

5.2 Assumptions Related to Recent History

The above examples demonstrate the expressive power of the modelling technique being proposed. In this section we introduce assumptions that relate to the order in which RoIs are presented to the classifiers. We will consider two examples where this is the case.

In order to deal with assumptions that relate to the past it is necessary for the recursively called function to have an additional parameter that carries the relevant history of that sequence of calls. At the extreme, the entire sequence from first RoI to the current assignment must be accommodated. But this would add significantly to the memory requirements of the Dynamic Program; a more efficient representation is used in the following examples.

Restricted Neighbours. First we will consider a constraint that will limit the number of objects of the same class that can appear in adjacent RoIs. To illustrate this we will add the assumption that no more than three Dogs can appear in sequence. So if there have been three Dogs in a row then the next entry must be a Cat - and hence this RoI will take less time to process as the **CorD** classifier can be omitted.

To cater for such constraints in the Dynamic Program (see Appendix, A.2.1) the recursive function, `Fc` has an additional parameter added, `Count`, that holds a count of the number of Dogs in the immediate history. This value is either 0, 1, 2 or 3. If it is 3 then the next element cannot be a Cat.

An execution of this program with the basic assumptions of no more than 10 Pets, 5 Cats and 7 Dogs produces a WCET estimate of **174** from the sequence ⟨C, D, D, C, D, D, C, D, D, D⟩. Initially the Dogs are in pairs; this maximises the execution time as once there are three in a row then the next one must be a Cat. An assumption such as 10 Pets, 1 Cats and 10 Dogs of course has the worst-case output sequence of ⟨D, D, D, C, D, D, D⟩.

Cache Example. In the above example a history parameter is used to expand the expressive power of the Assumptions predicate; it broadens what is considered to be valid allocations of Cats and Dogs. In this next example history is used not to define what is valid, but to alter the execution time required by each classifier as it executes.

To support what might be called a cache effect we modify the model as follows. The operation of **DBC** which works on a RoI that has been categorised as a Dog, and which usually takes eighteen time units to execute, is reduced by one unit if the previous RoI was also a Dog. Similarly, two or more Cats in sequence reduces the execution time of **CBC** by one for all but the first RoI in this sequence.

The code for this cache example is contained in Appendix, A.2.2. The execution of this code, for the assumption that there are at most six pets, two Cats and five Dogs results in a worst-case cost of **103** for the sequence ⟨D, C, D, D, D, D⟩ with individual costs of 18, 16, 18, 17, 17, 17. Although only one Cat is contained in this sequence, if the predicate is changed to have $N_c \leq 1$ then the worst-case cost is reduced to **98** which comes from the sequence ⟨D, D, D, D, C, D⟩ (with individual costs of 18, 17, 17, 17, 16, 13). Note, without the cache effect the maximum cost would be **101**.

5.3 Assumptions Related to Multi-models

One of the motivations for developing an approach to timing analysis based on assumptions is to support Multi-Model specifications [4,5]. This is an extension to Mixed-Criticality [6,26] thinking and allows a specification to cater for more than one model of the load on the system. This is motivated by either there being more than one definition of the expected worst-case environment for the deployment of the system, or there being distinct Stakeholders that have different classification requirements.

For example, with CADIS, assume there are two modes for the environment, One (D) that mainly contains Dogs:

$$\mathcal{A}^D \stackrel{\text{def}}{=} N \leq 8 \wedge N_c \leq 2 \wedge N_d \leq 5$$

and another (C) that is mainly cats:

$$\mathcal{A}^C \stackrel{\text{def}}{=} N \leq 7 \wedge N_c \leq 5 \wedge N_d \leq 3$$

The models could be united into one:

$$\mathcal{A} \stackrel{\text{def}}{=} N \leq 8 \wedge N_c \leq 5 \wedge N_d \leq 5$$

but this now incorporates scenarios that cannot occur (lots of both Dogs and Cats). The classification framework developed above can easily be adapted to support integrated multi-models [5]. Each model has its own Valid predicate (VD and VC) and the composite is simply

$$V(N_c, N_d) = VD(N_c, N_d) \vee VC(N_c, N_d)$$

For a step to be valid at least one of the models must deem it to be valid – but of course both can. So, for example, $V(1,1)$ is valid as $VD(1,1)$ is valid (and also $VC(1,1)$);

$V(3,1)$ is also valid but now only one model (the model C) sanctions the step; and for completion $V(4,4)$ is invalid since neither models considers the step to be valid.

With the above example the multi-model delivers a worst-case cost of **118**; for the single collapsed model this rises to **138**, a significantly more pessimistic estimate.

This example can be extended. Assume each of the models has a distinct stakeholder (SD and SC). Now SD is only interested in the breed of the Dogs (not Cats), and SC requires the breed of the Cats to be output but not that of the Dogs. The existence of two such stakeholders now effects the cost of classification. Consider the assignment to `X` in the Dynamic Program code (see Appendix) for a call of the recursive function `Fc` (but reversing the terms for readability):

```
X := Fc(TC, TD+1) + 18
```

where 18 is the full cost of classifying the breed of the potential further dog. But this full cost need only be spent if SD is still interested, i.e. when model D is still valid and therefore boolean variable `VD` is true. If SD is not interested than the cost of classifying the breed (i.e., 12) can be saved. Hence the above line becomes;

```
X := Fc(TC, TD+1) + (if VD then 18 else 6)
```

Similar changes are made to the other timing parameters.

5.4 Assumptions Related to Fault Tolerance

A robust classifier must be able to tolerate a level of misclassification. In our example **CorD** can, incorrectly, pass a Cat image to **DBC** (or a Dog image to **CBC**). This will result in an unclassified output from the two Breed classifiers. These outputs are then passed to the other classifier where they are (in our example) correctly processed. There is no output error, but misclassifications will increase the system's maximum execution time. To bound this value there must be an upper limit on the number of faults to be expected (and hence tolerated). This fault model becomes part of the Assumptions predicate; for example

$$\mathcal{A} \stackrel{\text{def}}{=} N \leq 10 \wedge N_c \leq 8 \wedge N_d \leq 8 \wedge F_{CorD} \leq 2$$

Now the recurrence has two extra alternatives. If the maximum number of faults has not yet been reached then there can be, in addition to a further Cat or Dog, a Cat that is initially misclassified or a Dog that is misclassified.

The recurrence function now has a third parameter (see Appendix 2.3) – the number of faults that have been experienced. And, in general there are four alternatives at each recurrence: a further Dog, ⟨D⟩, or Cat, ⟨C⟩, or Dog that is initially misclassified as a Cat, ⟨D*⟩, or a Cat misclassified as a Dog, ⟨C*⟩.

With the Assumptions specified above the worst-case cost is **200** which comes from the sequence ⟨D, D, D, D, D, D, D, C*, C*, D⟩.

6 Optimising the Internal Order of the Execution of the Classifiers

With the fault tolerance example employed in the previous section it is reasonable to ask if it is actually worth executing the **CorD** classifier. For example, would the simpler arrangement depicted in Fig. 3 lead to a lower worst-case cost? The recurrence for this arrangement is straightforward and leads to a worst-case cost of **210**. Reversing **DBC** and **CBC** is an improvement and delivers a worst-case cost of **206**. Nevertheless, it follows that the inclusion of **CorD**, which as shown above leads to a cost of **200**, is worthwhile.

However if the fault model was $F_{CorD} \leq 3$ then this worst-case cost rises to **210** and it would not be beneficial to include this (faulty) classifier; the sequence **Initial** → **DBC**→ **CBC** with its worst-case cost of **206** would now be the optimal arrangement.

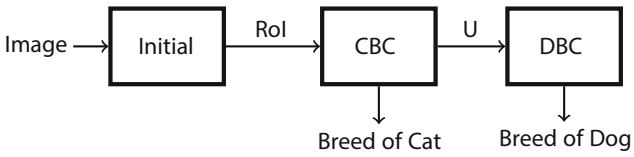

Fig. 3. CADIS2 – A *Simplified Cat And Dog Identification System*

This example points to an important application of the proposed framework: it supports design exploration. With the objective of minimising worst-case execution time, alternative designs can be compared. And even with a fixed set of classifiers it is possible to examine the order of their executions to again minimise the cost of undertaking the complete classification.

7 Open Issues

The above discussions of the framework has illustrated a number of different examples of the kinds of assumptions that can be accommodated. But what is the expressive power of the approach: what is the full range of assumptions that can be included?

A starting point for such a discussion is to note that it is clearly possible to define assumptions that are unsatisfiable. Folding into an assumption the need to solve the Halting Problem, or an encoding of some NP-hard problem will of course rule out the possibility of obtaining an efficiently computable 'Valid' predicate. For other, more reasonable, assumptions the ability to construct the Valid predicate is itself proof of satisfiability. If checking the satisfiability of assumptions is computationally non-trivial, then efficiency considerations must take the computational complexity of doing so into account; it may be computationally more efficient to simply assume that some or all of the assumptions hold and thereby take on the responsibility of satisfying more obligations than may be strictly necessary.

In terms of implementation, an assumption needs to be converted into a Valid predicate that checks that the addition of a further object (of a defined class) is Valid: it moves the system from one Valid state to another (and hence closer to termination).

Examples have shown that the Valid function:

- Can, for simple assumptions, be computed immediately from the current state.
- Needs to 'simulate' potential futures to check that there is a path from the proposed next step to a final state in which the Assumptions are upheld.
- Needs to retain knowledge of earlier steps if the sequence (and not just the final state) is constrained, or if the cost of a classification is dependent on recent history.
- Can incorporate a Fault Model to cater for faulty classifiers.
- Needs to indicate which model or models (in a multi-model application) reports a valid step.

8 Conclusion

It is clear that AI-enabled components will be increasingly deployed in future Real-Time Cyber-Physical Systems (RTCPS), and many of these will take the form of Classifiers. Autonomous behaviour requires the external world to be understood at a level of detail far beyond what is required for the current class of control systems. It is likely that banks of classifiers will be employed to identify the key objects in the environment. But to build a system with the right functionality and scale arguably requires a comprehensive model of this environment. Unfortunately in many scenarios such models are impossible (or infeasible) to derive. For this reason Jones and others have concentrated more on capturing the assumptions that define, and therefore constrain, the dynamic characteristics of the domains in which the RTCPS is to be deployed.

In this paper we have shown how assumptions can be used to constrain the work that needs to be undertaken by a cascade of classifiers. The assumptions not only specify how big the input space can be but they also critically indicate where particular classifiers need not be executed, and hence the computation load be reduced. By employing a relatively straightforward Dynamic Program it is possible to compute the worst-case execution time of the complete cascade. A correct, but tight, bound for this parameter is crucial if the schedulability of the RTCPS is to be achieved.

Assumptions can take many forms. A number of such have been illustrated in this paper; many others are feasible. To be practical it must be possible for an assumption to be validated – it must be possible in a reasonable time to determine if a particular input sequence can ultimately satisfy the assumption. Of course, this timing analysis is undertaken pre-run-time, and although a variety of sensors and banks of classifiers are likely to be employed in the future it is unlikely that the number of scheduled components will be more than perhaps 20 to 30. And hence the effort necessary to undertake the timing analysis is manageable.

As future work we will look at multi-labelling; where an input RoI has more than one class (a *Ginger Labrador* was the example given in Sect. 2). Now there is the potential for concurrent execution of some classifiers. As Edge computing is typically multi-core then real parallel execution is possible. This will speed up the classifying operations, but determining the worst-case execution time is not straightforward.

Appendix – The Dynamic Program

An outline of the (Ada) code for the Dynamic Program is given in below. The function returns one of four values: (i) the maximum of the two valid paths, or (ii) the value of taking a Cat when only a Cat is valid, or (iii) the value of taking a Dog when only a Dog is valid, or (iv) the value 0 as neither a Cat nor a Dog can be taken. Initially ignore the code in red.

```
KD = 13 (1+12)
KC = 11 (1+10)
UKD = 18 (1+5+12)
UKC = 16 (1+5+10)

type Pet is (Cat, Dog);
P : Pet;

function Fc(TC, TD : integer) return integer is
   X,Y : integer := 0;
   VD, VC : boolean;
begin
   if S(TC,TD) > -1 then return S(TC,TD); end if;
   VD := Valid(TC, TD+1);
   VC := Valid(TC+1, TD);
   if VD and VC then
     X := UKD + Fc(TC, TD+1);
     Y := UKC + Fc(TC+1, TD);
     X := max(X,Y);
     if X=Y then P:=Cat; else P:=Dog; end if;
     S(TC,TD) := X;
     return X;
   end if;
   if VC then P:=Cat; return KC + Fc(TC+1, TD); end if;
   if VD then P:=Dog; return KD + Fc(TC, TD+1); end if;
   return 0;
end Fc;
```

The array `S` holds previously computed values – that can be used to reduce the computational load. All elements in this array are initialised to -1. If the appropriate element of `S` holds a value strictly greater than -1 then it must have been previously computed and can be returned immediately.

Since the recurrence is bottom up, the initial call of the function is:

```
Cost := Fc(0, 0);
```

The call terminates and returns when a recursive call is made that has no valid successor (and hence returns 0 without making a further call on `Fc`).

The code implementing the function `Valid` is written according to the assumptions, and is therefore application-specific. The parameters are valid if they satisfy the assumptions, $\mathcal{A}$, or if there exists a sequence of future RoIs that will satisfy $\mathcal{A}$. As an example of the first type of definition: if there is a maximum of 8 pets, 4 Cats and 4 Dogs then the `Valid` function is simply:

```
function Valid(TC, TD : integer) is
begin
     return TC+TD <= 8 and TC <= 4 and TD <= 4;
end
```

This gives a result of **131**.

To output the sequence that gives rise to this maximum result a minor modification can be made to the code (see elements highlighted in red) and rather than calling the function `Fc` just once a series of calls are make:

```
SC := 0;  SD := 0;  -- count of Cats and Dogs so far
Cost := Fc(SC, SD);
while Cost > 0 loop
       -- re-initialise S
       case P is
       when Cat =>
            Put("C, "); SC := SC + 1;
       when Dog =>
            Put("D, "); SD := SD + 1;
       end case;
       Cost := Fc(SC, SD);
end loop;
```

A call of `Cost := Fc(0, 0)` now also returns, via the `P` parameter, the first entry in the worst-case sequence. If `P=Cat` then the `Fc` function is called again with `Cost := Fc(1, 0)`. This returns the next entry in the sequence. Calls are repeated with the number of Cats and Dogs so far identified used to define the parameters. Eventually, `Cost := Fc(4, 4)` is called and the, terminating, response of zero is returned. With this example the sequence ⟨C, C, C, D, D, D, C, D, ⟩ is produced for the worst-case execution time of **131**.

To give an indication of the efficiency of this code, an assumption of a maximum of 1000 Pets, 500 Cats and 700 Dogs returns the value **17400** immediately on a standard laptop. Indeed the production of the full sequence that returns this value, and which involves 1000 runs of the program, is only slowed down by the print to screen of the Cs and Ds. The initial execution, that just produced the WCET estimate, involved 830,601 recursive calls to the `Fc` function. And on 328,901 occasions (40%) the result was returned immediately by the `S` matrix.

A.2.1 Code for Prime Number of Cats Example

The $Valid$ predicate leads to the following code. To check that the number of Cats can be a prime number a loop runs from the current proposed value of `TC` to the maximum value it is allowed to take. If any of these values is a prime the loop is exited and `Pr` records `true`. If the loop terminates with no prime found then `Pr` retains its initiated value of `false`. For the function to return `true` the usual tests must comply and `Pr` must be `true`.

```
function Valid(TC, TD : integer) return Boolean is
  Pr : boolean := false;
begin
  for I in TC .. TC + (min(N-(TD+TC),Nc-TC)) loop
    if I in primes then Pr := true; exit; end if;
  end loop;
  return TC+TD <= N and TC <= Nc and TD <= Nd and Pr;
end;
```

A.2.1 Code for Restricted Neighbour

The code for the `Fc` function now has a `Count` parameter that holds the number of consecutive Dogs in the recent history. If `Count` is 0, 1 or 2 then a further Dog is allowed (and the `Count` value is increased by 1). If `Count` is equal to 3 then it is not valid for the current RoI to be a Dog. The basic code for this version of `Fc` is below.

```
function Fc(TC, TD, Count : integer) return integer is
    X,Y : integer := 0;
    VD, VC : boolean;
begin
    VD := Count < 3 and then Valid(TC, TD+1);
    VC := Valid(TC+1, TD);
    if VD and VC then
       X := UKD + Fc(TC, TD+1, Count+1);
       Y := UKC + Fc(TC+1, TD, 0);
       X := max(X,Y);
       return X;
    end if;
    if VC then return KC + Fc(TC+1, TD, 0); end if;
    if VD then return KD + Fc(TC, TD+1, Count+1); end if;
    return 0;
end Fc;
```

A.2.2 Code for Cache Example

The code for the `Fc` now has a very simple history parameter that gives the type of the last RoI to be classified. If this history flag has the same pet type as the current allocation then a unit of computation time is removed from the calculation.

```
type Pet is (Cat, Dog, Void);

function Fc(TC, TD : integer; H : Pet) return integer is
    X,Y : integer := 0;
    VD, VC : boolean;
begin
    VD := Valid(TC, TD+1);
    VC := Valid(TC+1, TD);
    if VD and VC then
      X := UKD + Fc(TC, TD+1, Dog) - (if H = Dog then 1 else 0);
      Y := UKC + Fc(TC+1, TD, Cat) - (if H = Cat then 1 else 0);
      X := max(X,Y);
      return X;
   end if;
   if VC then return KC + Fc(TC+1, TD, Cat)
           - (if H = Cat then 1 else 0); end if;
   if VD then return KD + Fc(TC, TD+1, Dog)
           - (if H = Dog then 1 else 0); end if;
   return 0;
end Fc;
```

The initial call on this function is now `Fc(0,0,Void)`.

A.2.3 Code for Fault Model Example As described in the main text, again a third parameter to `Fc` is added, this time to count the number of faults so far experienced. The following shows the code for the general four-way comparison.

```
function Fc(TC,TD,F : integer) return integer is
   X,Y,Z,W : integer := 0;
   VD, VC : boolean;
begin
   VD := Valid(TC, TD+1);
   VC := Valid(TC+1, TD);
   if VD and VC  and F < Fmax then
     X := UKD + Fc(TC, TD+1,F);
     Y := UKC + Fc(TC+1, TD,F);
     W := UKF + Fc(TC, TD+1, F+1); -- where UKF=28 (1+5+12+10)
     Z  := UKF + Fc(TC+1, TD, F+1);
     Z := max(X,Y,W,Z);
     return Z;
  end if;
   ...
end Fc;
```

References

1. Baruah, S.K. , Burns, A., Wu, Y.: Optimal synthesis of IDK-cascades. In: Proceedings of the 29th International Conference on Real Time Networks and Systems, RTNS. ACM (2021)
2. Bellman, R.: Dynamic Programming, 1st edn. Princeton University Press, Princeton (1957)
3. Bowden, L.: The story of IFF (Identification Friend or Foe). IEE Proc. A Phys. Sci. Measur. Instrument. Manag. Educ. Rev. **132**(6), 435 (1985). https://doi.org/10.1049/ip-a-1.1985.0079
4. Burns, A.: Multi-model systems—an MCS by any other name. In: Proceedings of the WMC Workshop, IEEE Real-Time Systems Symposium (RTSS) (2020)
5. Burns, A., Baruah, S.: Multi-model specifications and their application to classification systems. In: Proceedings of the 31th International Conference on Real-Time Networks and Systems, RTNS 2023. ACM (2023)
6. Burns, A., Davis, R.I.: A survey of research into mixed criticality systems. ACM Comput. Surv. **50**(6), 1–37 (2017)
7. Burns, A., Hayes, I.J., Jones, C.B.: Deriving specifications of control programs for cyber physical systems. Comput. J. **63**(5), 774–790 (2020)
8. Chen, J., Ran, X.: Deep learning with edge computing: a review. Proc. IEEE **107**(8), 1655–1674 (2019)
9. Chen, Y.-F., et al.: Automated assume-guarantee reasoning through implicit learning. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification, pp. 511–526. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_44
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
11. Davis, R., Baruah, S., Burns, A., Wu, Y.: Optimally ordering IDK classifiers subject to deadlines. Real-Time Systems (2022)
12. Gupta, P., Pooja, J., Kakde, O.G.: Deep learning techniques in radar emitter identification. Def. Sci. J. **73**, 551 (2023)
13. Hayes, I.J., Jackson, M.A., Jones, C.B.: Determining the specification of a control system from that of its environment. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 154–169. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_10
14. Hayes, I.J., Jones, C.B.: A guide to rely/guarantee thinking. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) SETSS 2017. LNCS, vol. 11174, pp. 1–38. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02928-9_1
15. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: methodology and case studies. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, pp. 440–451. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028765
16. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University (1981). Printed as: Programming Research Group, Technical Monograph 25
17. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP, pp. 321–332. North-Holland (1983)
18. Jones, C.B.: Tentative steps toward a development method for interfering programs. Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
19. Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal Methods Syst. Design **8**(2), 105–122 (1996)
20. Jones, C.B., Hayes, I.J.: Possible values: exploring a concept for concurrency. J. Logic. Algeb. Methods Program. **85**(5), 972–984 (2016)

21. Jones, C.B., Hayes, I.J., Jackson, M.A.: Deriving specifications for systems that are connected to the physical world. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 364–390. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75221-9_16
22. Jones, C.B., Pierce, K.G.: Elucidating concurrent algorithms via layers of abstraction and reification. Formal Aspects Comput. **23**(3), 289–306 (2011)
23. Powell, D.: Failure mode assumptions and assumption coverage. In: Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 386–395. IEEE Computer Society Press (1992)
24. Razavi, K., Luthra, M., Koldehofe, B., Muhlhauser, M.M., Wang, L.: FA2: fast, accurate autoscaling for serving deep learning inference with SLA guarantees. In: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2022)
25. Sokolova, M., Lapalme, G.: A systematic analysis of performance measures for classification tasks. Inf. Process. Manage. **45**(4), 427–437 (2009)
26. Vestal, S.: Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: Proceedings of the Real-Time Systems Symposium (RTSS), pp. 239–243 (2007)
27. Wang, X., Luo, Y., Crankshaw, D., Tumanov, A., Yu, F., Gonzalez, J.E.: IDK cascades: fast deep learning by learning not to overthink. arXiv preprint arXiv:1706.00885 (2018)
28. Yao, S., Hu, S., Zhao, Y., Zhang, A., Abdelzaher, T.: Deepsense: a unified deep learning framework for time-series mobile sensing data processing. In: Proceedings of the 26th International Conference on World Wide Web, pp. 351–360 (2017)