STARS of MPEG decoder: a case study in worst-case analysis of discrete-event systems

Felice Balarin
Cadence Berkeley Labs
felice@cadence.com

ABSTRACT

STARS (STatic Analysis of Reactive Systems) is a methodology for worst-case analysis of discrete systems. Theoretical foundations of STARS have been laid down [1, 2, 3], but no implementation has been presented so far. We introduce an implementation of STARS as an extension of YAPI, a programming interface used to model signal processing applications as process networks [7]. We apply STARS to a YAPI model of an MPEG decoder. We show that worst-case bounds computed by STARS are quite close to simulated values (within 15%). We also show that additional effort by the designer required to build STARS models is very small compared to effort of building the YAPI simulation model, and that the run times of STARS are negligible compared to the simulation run times.

KEY WORDS: system verification, worst-case analysis, static analysis

1. INTRODUCTION

STARS (**ST**atic **A**nalysis of **R**eactive **S**ystems) is a methodology for worst-case analysis of discrete systems. Theoretical foundations of STARS have been laid down in [1, 2, 3]. It can be used to verify different properties of systems, such as power consumption, timing performance, or resource utilization. It consists of three main phases:

1. choosing an abstract representation of signals, called a *signature*,
2. building abstractions (called σ-*abstractions*) of system components,
3. analyzing σ-abstractions and interpreting results.

The main results of [1, 2] are the properties that signatures and σ-abstractions must satisfy in order for STARS to produce valid worst-case bounds. An ordering must be defined in the domain of signatures, so that it can be precisely determined if a signal is "worse" than the other one. In addition, signatures need to preserve sufficient information such that the usage of resource of interest (e.g. time, power, memory bandwidth, ...) can be accurately estimated.

The main requirement on σ-abstractions is that they be conservative predictor of the system behavior, i.e. they need to predict a system response that is at least as "bad" as the real response. If σ-abstractions are not conservative, then results of STARS might not be worst-case bounds (in other words, they are useless). Checking whether a σ-abstraction is conservative is an instance of a classic verification problem: "Is every behavior of an implementation (in this case, the system) consistent with the specification (σ-abstraction)?". Therefore, it can be solved by one of the usual approaches: by construction, by formal verification, or by simulation.

Solution by construction means that σ-abstractions are automatically generated from system specifications. While this approach has been pursued for a limited class of systems [3], in general, automatic abstraction is provably untractable. Solution by formal verification has the usual pluses and minuses, the promise of a complete verification on one hand, and very limited capacity in practice on the other hand.

Solution by simulation is never complete, because exhaustive simulation is not feasible. Nevertheless, it is the mainstay of verification, and often the only available option. In this paper, we describe an environment which allows both simulation and STARS, so that the *same* σ-abstraction can be *verified* by simulation and *used* by STARS. More precisely we extend the simulation environment YAPI [7] with notions of *counters*, σ-abstractions, and *monitors*. Counters, both built-in and user-defined, count the number of occurrences of interesting events in the system. The purpose of counters is to define signatures, and as such they are used by σ-abstractions. The purpose of monitors is to check whether the values in a current simulation run satisfy the bounds given by a σ-abstraction. In other words, monitors check if σ-abstractions are indeed conservative.

In YAPI, systems are modeled as concurrent *processes* that communicate by exchanging *tokens*. STARS is used to compute an upper bound on the number of generated tokens in a given time interval. This information can be useful in many ways. It is often not hard to relate the number of tokens to the number of memory accesses, or the energy needed to process the tokens. Therefore, a maximum number of tokens in a given time period can be used to bound power and

memory bandwidth in that period. Another use of STARS results is to check the quality of simulation test-sets. If the number of generated tokens in a simulation is close to the bound computed by STARS, we can be quite confident that those test vectors indeed stretch the system resources to the maximum. If that is not the case, the designer may try using STARS results as a guideline in developing a more challenging test-set.

In principle, our approach could be applied to other discrete event simulators, such as Ptolemy [4] or VCC [5]. However, YAPI does have a couple of advantages. Firstly, it is just a C++ class library, so it is easily extendible. Even more importantly, we have available a YAPI simulation model of a substantial realistic design, namely an MPEG decoder. This enables us to obtain practical information about the quality of STARS results, and the effort of building σ-abstractions.

The rest of this paper is organized as follows: first, in Section 2 we give an overview of YAPI and the MPEG decoder design we use. Then, in Section 3 we propose extending YAPI to facilitate STARS. We present experimental results in Section 4, and give some final conclusions and ideas for future work in Section 5.

2. YAPI AND THE MPEG DECODER

YAPI is a programming interface used to model signal processing applications. In YAPI, systems are represented as networks of communicating *processes*. A process communicates with other processes through input and output *ports*. A process can *read* tokens from an input port, and *write* tokens to an output port. A *channel* connects an output port of some process to an input port of some other process. Each channel is an infinite FIFO queue.

YAPI, as defined so far, is a variant of Kahn process networks. In addition, YAPI allows a process to *select* a port with available tokens from a list of ports. This feature makes YAPI non-deterministic (unlike Kahn process networks). However, it is indispensable for modeling reactiveness, such as user interaction.

A YAPI model of an MPEG decoder is shown in Figure 1. The model was first presented in [8]. It is intended to represent functional decomposition of the decoding process, rather than the architecture of the implementation. This is typical in the design methodology based on YAPI, where applications and architecture are modeled separately, so that multiple architectural mapping can be explored for a given application.

An example of a YAPI process is shown in Figure 2. It is an excerpt of the specification of Tvld module from Figure 1. For brevity, some code has been deleted (comments // ... mark locations of deleted code). The module is represented by the class *Tvld*. Like any module representation in YAPI, *Tvld* is a refinement of class *Process*. Its inputs include *Tvld_bits_In* of type *unsigned char*, and *Tvld_cmd_In* of type *int*. Its outputs include *Thdr_status_Out* of type *unsigned int*. Every refinement of *Process* must have the *main()* function defined. When the module has to be executed, the simulator calls that function. A module can also have some auxiliary functions defined, like *bit_NextStartCode()* in case

```
class Tvld : public Process {
  // ...
  InPort<unsigned char> Tvld_bits_In;
  InPort<int> Tvld_cmd_In;
  OutPort<unsigned int> Thdr_status_Out;

  void main();
  unsigned int bit_NextStartCode();

  void sigmaAbs(double T);  // used
  counter nextCodeResp;     // by
  counterRef nextCodeCmd;   // STARS
};

void Tvld::main() {
  while(1) {
    read(Tvld_cmd_In,cmd);
    switch(cmd) {
      case CMD_NEXTSTARTCODE:
        nextCodeResp++;
        write(Thdr_status_Out, bit_NextStartCode());
      break;
    // ...
} } }
```

Figure 2: A YAPI process.

of *Tvld*. The rest of *Tvld* definition shown in Figure 2 is an extension of YAPI for STARS, and will be explained in the next section.

The *main()* function of *Tvld* is a typical for a server module. In an infinite loop, it reads a token carrying a request (by executing *read(Tvld_cmd_In,cmd)*, where *cmd* is an integer member of *Tvld*), and than performs appropriate action (e.g. *write* a token with value *bit_NextStartCode()* to port *Thdr_status_Out*).

3. YAPI EXTENSIONS FOR STARS

In this section, we present STARS-YAPI, an extension of YAPI that enables verification of σ-abstractions by simulation, as well as performing STARS. The first extension is used to define signatures. In general, signatures are functions of system executions. They have to satisfy two basic requirements:

1. it must be possible to compare them, i.e. a partial order must be defined on their ranges,
2. they must be monotone, in the sense that the signature of a segment must be smaller than a signature of the whole execution.

In STARS-YAPI, signatures are vectors of *counter* values. For every port in the system there is a built-in counter which counts the number of tokens transmitted through that port. The user can also extend the signature by defining additional counters, counting the number of occurrences of some other relevant events (e.g. the number of times certain event carries some particular value).

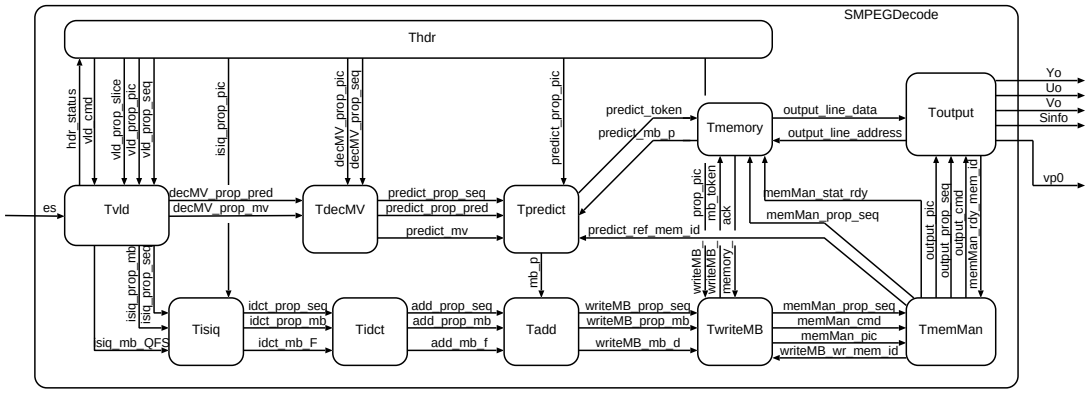

Figure 1: MPEG decoder

It is not hard to see that vectors of counter values satisfy two necessary conditions for a signature:

1. component-wise comparison is a partial order,
2. the values of counters (i.e. the number of produced and consumed tokens) can only increase if an execution segment is extended.

The counters are used to build abstractions of systems. However, the counters provide only a limited information about the system behavior, and thus the resulting system abstraction can be only of limited precision. If a more refined abstraction is desired, the user can define additional counters to keep extra information.

For example, module *Tvld* in Figure 2 has a user-defined counter *nextCodeResp*. It is intended to count the number of times *Tvld* responds to *CMD_NEXTSTARTCODE* command. This information is than used to bound the activity of some other modules, such as *Thdr*. During the simulation, STARS-YAPI automatically updates values of built-in counter, but the user is responsible for updating user-defined counters. For example, in Figure 2, *nextCodeResp* is explicitly incremented just before generating a reaction to command *CMD_NEXTSTARTCODE*.

Defining an additional counter is much like adding an additional output port to the module, except that this port is not used in normal operation, but just provides additional information needed to build an accurate abstraction. Therefore, with only a slight abuse of notation, we use the term *output counters* to denote both built-in counters associated with output ports, as well as user-defined counters.

In our experience, by adding user-defined counters, it was always possible to reach the desired accuracy of abstraction. However, it is not clear that this is true in general. At present, we can neither prove that arbitrarily accurate abstraction can be found if enough counters are added, nor do we have an example where no number of counters can preserve sufficient information.

While system signals are represented with signatures, system components are represented with *σ-abstractions*. In

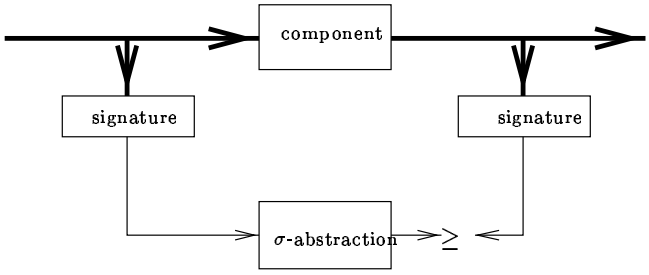

Figure 3: σ-abstraction must be conservative.

STARS-YAPI, each process has an associated σ-abstraction. Roughly, its purpose is to compute bounds of output counters based on the values of the input counters.

A σ-abstraction is valid if it is a conservative predictor of module's behavior. That means that for any sequence of inputs (and any segment of that sequence), the predicted output signature computed by a σ-abstraction must be larger than the signature of actual outputs. This requirement is presented graphically in Figure 3. Equivalently, we may say that the σ-abstraction must be an *abstract interpretation* [6] of the original module.

In some cases, an accurate σ-abstraction requires more information about the module's environment than what is provided by the number of tokens transfered through the input ports. STARS-YAPI provides *counter references* as mechanism to access that additional information. Counter references are like pointers to counters in other components, except that they allow "read-only" access, i.e. access through counter references does not allow changing the value of a counter.

For example, to bound *nextCodeResp* in *Tvld*, it is useful to know a bound on how many *CMD_NEXTSTARTCODE* commands can *Tvld* get at its inputs. If such a counter exists in the module producing commands for *Tvld*, the user can *connect* it to the counter reference *nextCodeCmd*.

Counter references help maintain the modularity of specification. Using *nextCodeCmd* allows the same σ-abstraction

```
void Tvld::sigmaAbs() {
  if(nextCodeCmd.is_connected()) {
    nextCodeResp.set_bound(
      min(Tvld_bits_In/4, nextCodeCmd)
    );
  } else {
    nextCodeResp.set_bound(Tvld_bits_In/4);
  }
  //...
}
```

Figure 4: σ-abstraction of *Tvld* module.

of *Tvld* to be used in different environments. In the typical *intellectual property (IP) assembly* design paradigm, *Tvld* and its σ-abstraction could be developed by the IP provider and then re-used in many different applications. In this scenario, the responsibility of the application designer would just be to connect a proper counter to *nextCodeCmd*. To do this, the application designer must understand what kind of information *Tvld* expects from *nextCodeCmd*, but this is no different than connecting any other input or output port in the assembly process.

In STARS-YAPI, every refinement of a class *Process* may have a member function called *sigmaAbs*. The function *sigmaAbs* returns no value, but its body should contain calls to *set_bound* member function of all output counters. To compute this bounds, *sigmaAbs* has access to input counters and all member counter references. The bound of either inputs or counter references can be accessed simply by putting the name in an integer expression. In addition, counter references have member function *is_connected()* with the obvious meaning. Trying to access a bound of a counter reference that is not connected will cause an exception. Similarly, trying to access a counter bound that has not been previously set will also cause an exception.

For example, a fragment of the *Tvld::sigmaAbs* is shown in Figure 4. There are two bounds on *nextCodeResp*. The first bound states that the number of responses to command *CMD_NEXTSTARTCODE* cannot be larger than the number of commands received. The second bound can be established because every response to *CMD_NEXTSTARTCODE* consumes at least four tokens from the input *Tvld_bits_In* (this fact can be deduced by analyzing the code of function *bit_NextStartCode()*). Therefore, if *nextCodeCmd* is connected, the bound of *nextCodeResp* is set to the tighter of two bounds. Otherwise, it is set to the only available bound.

The σ-abstraction of the system as a whole is a collection of σ-abstractions of all system components. Together, they compute bounds on all output counters. Since the system is closed and every counter is an output counter of some component, the system σ-abstraction (denoted by F) maps a vector of counter values to another vector of counter values.

The basic theorem behind STARS states that if some vector of counter values x is a fix-point of F, and x is larger than the vector of initial counter values, then x is larger than the signature of *any* execution [1]. Based on this result, STARS solves $x = F(x)$ by iteration, using counter values in the initial state as an initial solution. Since F is guaranteed to be monotone, the iteration will either converge or the value of x will grow beyond any bounds. To prevent infinite iteration, the user my specify a boundary value. If x reaches that value, the iteration terminates with failure. However, if the convergence is achieved before that, than x is a valid worst-case bound.

STARS needs only σ-abstractions for its computation, and it never executes modules' *main()* functions. Therefore, STARS cannot check any correspondence between two views of the module (*main()* and *sigmaAbs()*). For this purpose, we extend YAPI with the concept called *monitor*. Each *Process* has a *monitor* member function. Calling *monitor* at any time during simulation creates a monitor object. Any time an input of the module gets a new token, the monitor object executes its *sigmaAbs* function. However, in this context, mentioning an input counter has a slightly different meaning. Rather than evaluating to a previously set bound, a counter name in the integer expression evaluates to the actual increment in counter value from the moment the monitor object was created until the current time. Also, when *set_bound* is executed for some output counter, the monitor checks whether the bound is indeed larger than the actual (incremental) counter value. If not, a warning is reported. Schematically, monitors perform the test as in Figure 3. In addition, monitors report a warning if σ-abstractions exhibit non-monotone behavior.

Monitors are the key tool in developing σ-abstractions. Without them, debugging σ-abstractions would be almost impossible. They are also useful in ensuring that engineering changes made to *main()* later in the design process do not invalidate existing σ-abstractions.

4. RESULTS

We have applied STARS to the MPEG decoder shown in Figure 1. The purpose of that experiment was to test on a realistic example how accurate can STARS be, and how much additional designer effort is needed to create σ-abstractions.

The effort in building σ-abstractions was very small compared to the effort of specifying the design. Many blocks of the design are of data-flow nature, where the number of output tokens is the same as the number of input tokens, and the block implements some complex signal-processing algorithm. σ-abstractions of such blocks are trivial. However, some blocks are more data-dependent and the number of output tokens depend on specific values of input tokens. σ-abstractions of such blocks are more complex, but still significantly simpler than blocks themselves. Overall the code for σ-abstractions was less than 1% of the total code.

To test the accuracy of STARS, we have performed the following two-step experiment:

1. We have simulated the MPEG decoder using a particular video stream as an input. During simulation, we have collected statistics on the relevant parameters of the stream (the number of pictures, the number of macro-blocks, ...).
2. We have used the parameters collected in step 1 as

output	simulation	STARS	difference
Yo	2842560	3248640	14.3%
Uo	1423296	1626624	14.3%
Vo	1423296	1626624	14.3%
Sinfo	1	1	0%
vp0	14	16	14.3%

Table 1: Experimental results.

the worst-case values, i.e. we have built an environment σ-abstraction in which the bounds of the picture, macro-block and other counters are set to be exactly the values collected in the simulation.

Obviously, the results produced by STARS in this experiment are valid only as long as the input stream satisfies the environment σ-abstraction, which any sufficiently long input stream will not. A more useful experiment would let the environment σ-abstraction bound the *rates*, e.g. the maximum number of pictures or macro-blocks per time unit, since such bounds can be valid for *any* input stream. In that case, the STARS computes the maximum output rates, and even more importantly maximum rates of internal signals, which in turn determine performance requirements of system resources (e.g. processor speed and memory bandwidth).

Nevertheless, our experiment is well suited for the purpose of checking STARS accuracy. Since the environment's worst-case behavior is assumed to be exactly the one of the simulated stream, STARS should ideally compute the output bounds that are exactly equal to simulated values.[1] The results of the experiment are summarized in Table 1. As it can be seen from the table, the error is always less than 15%. This reasonable accuracy was accomplished with four orders of magnitude lesser computation effort for simulation (30s for simulation vs. 2ms for STARS). It is interesting that the error is the same (in percents) for all the signals. This suggests that all of the errors have a common root in a non-ideal abstraction of a single internal signal.

It is important to note that the error is due to a particular choice of σ-abstractions and not inherent is STARS. It is quite possible that a more elaborate σ-abstraction (perhaps with addition of some extra counters) would result in more accurate bounds.

5. CONCLUSIONS

We have presented an implementation of STARS as an extension of YAPI, a programming interface used to model signal processing applications. In our approach, σ-abstractions used by STARS can also be exercised during simulation. This facilitate keeping σ-abstractions consistent with the more detailed simulation model of the system. We use an MPEG decoder as a case study to indicate that a reasonable worst-case bound can be obtained with a small increase in designer's effort, and with negligible computation resources.

In this paper, we have used STARS to bound only the number of tokens that are produced and consumed in the system. A more interesting analysis would be to relate these numbers to system resource usage, and compute the bounds on processor time, memory bandwidth and energy consumption, for example. YAPI is not sufficient for this purpose because it contains no notion of architectural resources. However, there are design environments (e.g. VCC [5]) where a functional description like YAPI can be mapped to an architecture description. We plan to extend STARS to such mapping to allow very rapid check if performance requirements are met by a particular mapping.

[1] We cannot make this statement with absolute certainty. In general, some other input stream with the same (or lesser) signature, might produce the output stream with the larger signature. If that is the case, then even the most accurate σ-abstractions should result in bounds larger than the simulated values. However, our informal inspection of the system lead us to believe that such an input stream does not exist.

Acknowledgments

I am grateful to Erwin de Kock and Wido Krujitzer of Philips Research and Jean-Yves Brunel of Cadence for providing access to YAPI and the MPEG decoder, generous help during this work, and helpful comments on early drafts of this paper.

6. REFERENCES

[1] Felice Balarin. Worst-case analysis of discrete systems. In *Digest of Technical Papers of the 1999 IEEE International Conference on CAD*, November 1999.

[2] Felice Balarin. Worst-case analysis of discrete systems based on conditional abstractions. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)*, July 1999.

[3] Felice Balarin. Automatic abstraction for worst-case analysis of discrete systems. In *Proceeding of DATE 2000 Conference*. IEEE Computer Society, 2000.

[4] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[5] Cadence virtual component co-design (VCC) environment . http://www.cadence.com/datasheets/vcc_environment.html.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th Ann. ACM Symp. on Principles of Prog. Lang.* 1977.

[7] E.A. de Kock, G.Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverese, and K.A. Vissers. YAPI: Applocation modeling for signal processing systems. In *Proceedings of the 37th ACM/IEEE Design Automation Conference*, pages 402–405, June 2000.

[8] P. van der Wolf, P. Lieverese, M. Goel, D.L. Hei, and K.A. Vissers. An MPEG-2 decoder case study as a driver for a system level design methodology. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)*, July 1999.