STARS in VCC: complementing simulation with worst-case analysis

Felice Balarin
Cadence Berkeley Labs
Berkeley, CA
felice@cadence.com

Abstract

STARS is a methodology for worst-case analysis of embedded systems. STARS manipulates abstract representations of system components to obtain upper bounds on the number of various events in the system, as well as a bound on the response time. VCC is a commercial discrete event simulator, that can be used both for functional and performance verification. We describe an extension of VCC to facilitate STARS. The extension allows the user to specify abstract representations of VCC modules. These abstractions are used by STARS, but their validity can also be checked by VCC simulation. We also propose a mostly automatic procedure to generate these abstractions. Finally, we illustrate on an example how STARS can be combined with simulation to find bugs that would be hard to find by simulation alone.

1 Introduction

STARS (**ST**atic **A**nalysis of **R**eactive **S**ystems) is a methodology for worst-case analysis of embedded systems [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 signatures and σ-abstractions must satisfy certain properties, in order for STARS to produce valid worst-case bounds [2]. An ordering must be defined in the domain of signatures, so that it can be precisely determined if a signal, represented by one signature, is “worse” than another signal, represented by a different signature. 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 simulation is never complete, because exhaustive simulation is not feasible. Nevertheless, it is the mainstay of verification, and often the only available option. At the high-level of abstraction, embedded systems are often modeled (and simulated) as networks of concurrent *processes* that communicate through *events*. There are a number of public (e.g. Ptolemy [6]), proprietary (e.g. YAPI [10]), and commercial (e.g. VCC [7]) simulators that operate at this level.

Typically, system components are modeled in a standard procedural language, like C++, enhanced with a library defining *ports*, through which events can be *emitted* and *detected*. For example, in VCC, such an extension of C++ is called *Black-Box C++*. VCC also allows the components to be specified in *White-Box C*, a dialect of C.

In addition to functional simulation, VCC also allows performance analysis. The design is first *mapped* to an implementation architecture, with typical architectural elements being processors, buses, ASICs, and memories. Once the design is mapped, it is possible to *estimate* performance of the given implementation. These estimates can either be provided by the user through so-called *delay models*, or, in case of White-Box C models mapped to a processor, VCC can generate them automatically. Given a White-Box C model, VCC can produce a Black-Box C++ model that is equivalent to the original one, except that it is also annotated with timing estimates. It is then possible to simulate the design with these timing information.

There are two basic contributions of this paper. First, we describe an extension of VCC (called *VCC-STARS*) that allows both simulation and STARS, so that the *same* abstractions can be *verified* by simulation and *used* by STARS.[1] More precisely, we extend Black-Box C++ 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. Their purpose is to define signatures, and as such they are used by σ-abstractions. The purpose of monitors is to check whether the values in the current simulation run satisfy the bounds given by a σ-abstraction. In other words, monitors check (by simulation) if σ-abstractions are indeed conservative.

The second contribution of this paper is a procedure to automatically generate σ-abstractions for White-Box C models. The procedure may not always succeed, and it is not hard to show that no procedure that always succeeds exists, because the problem it attempts to solve is undecidable. Therefore, we have designed the procedure to accept a partial solution from the user. In this way, the procedure is reducing (and in some cases eliminating) the designer's effort in building a σ-abstraction.

STARS can be used to compute an upper bound on the number of events generated in a given time interval. This information can be useful in many ways. It is often not hard to relate the number of events to the number of memory accesses, or the energy needed to process the events. Therefore, a maximum number of events 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 events in 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 use STARS results as a guideline in developing a more challenging

[1]While VCC is a commercial product, at this point, VCC-STARS is a research project, and it is not commercially available.

test-set.

Finally, for mapped and estimated designs, STARS can also be used to analyze processor utilization. More precisely, STARS uses estimated run-times to compute a bound on the *maximum busy-period*, i.e. the longest interval of time the processor can continually be not idle. If the processor cannot be idle while there are pending service requests, then a bound on the busy-period is also a bound on the response time.

1.1 Related Work

STARS is based on the analysis of a system abstraction. Many researchers have suggested using abstractions to simplify formal verification of systems (e.g. [8, 5, 11]). In these approaches it is shown that abstraction preserves some properties of systems, so to verify a property of the detailed system, it is enough to verify it for the abstract one. In contrast, the theory behind STARS relates results of certain analysis of the abstract system to the worst-case behavior of the detailed system. Thus, it can be seen as an instance in the abstract interpretation framework [9].

The original motivation for STARS was timing analysis of real-time software. This problem has been addressed before, starting with Liu and Layland [13]. They give an exact solution, but only for a very restricted model. To fit a realistic system into this model, many conservative simplifications are required. The restrictions on the model have been somewhat relaxed later [1], however several significant limitations are present in all the previous approaches, but not in STARS. For example, the processing time requirements of a component were assumed to be constant. In contrast, STARS allows them to be a function of the inputs and internal states.

Integrating STARS into a simulation framework was also proposed in [4]. In that case, the simulator was YAPI, but the basic approach is quite similar (counters, references, σ-abstraction, monitors). We take these similarities as an indication that these concepts are valid for any discrete-event simulator, and not particular to either YAPI or VCC. The VCC integration proposal goes significantly beyond YAPI proposal in connecting STARS to timing information and using VCC produced performance estimates for automatic generation of σ-abstractions. (YAPI has no notion of either architecture or time.)

The rest of this paper is organized as follows. In Section 2, we describe extensions to Black-Box C++ models necessary to specify signatures and σ-abstractions, and then in Section 3, we explain how to run STARS and how to validate σ-abstractions by simulation. In Section 4, we propose a procedure to automatically generate σ-abstractions from a White-Box C model, and explain how the user may provide hints to help the automatic procedure. In Section 5, we describe an example, and finally we give conclusions in Section 6.

2 Black-Box C++ Extensions

In Black-Box C++, each system component is represented by a separate class. Each one of these classes must be derived from a base class called `CPPBlackBoxModel`. These derived classes usually have some members of classes `InputPort` and `OutputPort`. The class `InputPort` has a member function `Enabled` used to check if there is a token at that port, and a member function `Value` which return the value of the token (if any is present). The class `OutputPort` has a member function `Post` which emits a token on that port. Each derivative of the class `CPPBlackBoxModel` must contain a definition of the `Run` member function. This function models the behavior of the component. It is called during the simulation. The precise time when it is called depends on the scheduling algorithm, which is a property of the implementation architecture.

In this section we describe new classes and extensions to existing classes used to define signatures and σ-abstractions. (A word of caution: examples in this paper are derived from those in VCC, but, for brevity, they have been modified somewhat: some names have been shortened, some macros expanded, some levels in the class hierarchy skipped. Thus, they are no longer valid VCC examples. Nevertheless, they highlight all the key concepts, and the parts related to STARS have not been altered at all.)

2.1 Counters

The first step of STARS is signature definition. In general, signatures are functions on 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 an execution can only increase as the execution extends in time.

In VCC-STARS, 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. For example, for the input port `QueueIn` in Figure 1, the associated counter is `QueueIn.count`. The user may also extend the signature by defining additional counters as members of a Black-Box C++ model, e.g. `numExecutions` in Figure 1.

We represent counters by objects of the class `starsCounter` which maintain the value and the bound for a counter. The value can be changed by the increment operator ++. The bound can be changed using the member function `setBound`. The state of an object of the class `starsCounter` can be accessed by casting the object to the integer type. The value of the integer cast is different in simulation and STARS. In STARS, it evaluates to the current bound of the counter, while in simulation it roughly corresponds to the value of the counter (see Section 3.1 for more details).

During simulation, built-in counters are incremented automatically each time a token is transmitted through a port. However, the user is responsible for incrementing user defined counters, in order for them to represent the number of occurrences of the event of interest. For example, `numExecutions` in Figure 1 is incremented each time the function `Run` is executed. Therefore, its value is equal to the number of executions of `Run`.

Vectors of counter values satisfy the two necessary conditions for a signature stated above, because component-wise comparison is a partial order, and 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. 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.

2.2 Counter References

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. VCC-STARS pro-

```
class fifo:
  public CPPBlackBoxModel
{
  InputPort<int> QueueIn;
  InputPort<int> ClearToSend;
  OutputPort<int> QueueOut;
  bool cts_;
  void FIFOPush(int);
  int  FIFOPop()
  starsCounter numExecutions;
  starsCounterRef maxExecutions;
  ...
};
```

```
void fifo::Run()
{
  numExecutions++;
  if (ClearToSend.Enabled()) {
    if (!FIFOempty()) {
      QueueOut.Post(FIFOPop());
      cts_ = false;
    } else {
      cts_ = true;
  } }

  if (QueueIn.Enabled()) {
    FIFOPush(QueueIn.Value());
    if (cts_) {
      QueueOut.Post(FIFOPop());
      cts_ = false;
} } }
```

```
void fifo::starsAbstraction()
{
  QueueOut.count.setBound(
    min(QueueIn.count,
ClearToSend.count));

  if(maxExecutions.isConnected()){
    numExecutions.setBound(
      maxExecutions);
  } else {
    numExecutions.setBound(
      QueueIn.count +
      ClearToSend.count);
} }
```

Figure 1: A Black-Box model of a FIFO Queue

vides *counter references* as mechanism to access this 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. More precisely, it is not possible to increment or call `setBound` on an abject of class `starsCounterRef`. However, it is possible to cast it to the integer type, which has the same value as casting the referenced counter to the integer type. Whether a counter references is valid, i.e. whether it indeed references a counter, is checked by the member function `isConnected`. Trying to access the value of a counter reference that is not connected will cause an exception.

For example, the bound on `numExecutions` in Figure 1 depends on the module's environment, i.e. on the number of times the scheduler calls `Run`. If the environment maintains a counter with such information, it could be *connected* to `maxExecutions` counter reference (perhaps at the time `fifo` is instantiated). Then, the σ-abstraction of `fifo` could use `maxExecutions` to bound `numExecutions`.

Counter references help maintain the modularity of specification. They enable the same σ-abstractions to be used in different environments. In a typical *intellectual property (IP) assembly* design paradigm, a module 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 proper counters to module's counter references. To do this, the application designer must understand what kind of information the module expects from the counter reference, but this is no different than connecting any other input or output port in the assembly process.

Counter references are similar to input ports, in the same way that user-defined counters are similar to output ports. Therefore we use the term *input counters* to refer to both input ports counters and counter references.

2.3 σ-abstractions

In STARS, σ-abstractions are used to represent system components. The purpose of a σ-abstraction is to compute bounds on output counters based on the values of 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 given by a σ-abstraction must be larger than the signature of actual outputs. This requirement is presented graphically in Figure 2.

In VCC-STARS, σ-abstraction are represented by a member function of Black-Box C++ objects called `starsAbstraction`. This function returns no value, but its body should contain calls to

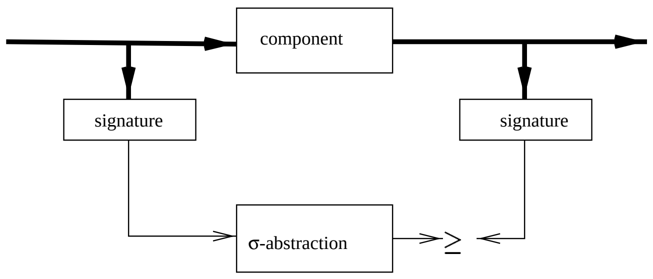

Figure 2: σ-abstraction must be conservative

`setBound` member function of all output counters. To compute these bounds, it has access to input counters.

For example, the `fifo` module in Figure 1 has two output counters `numExecutions` and `QueueOut.count`. In the function `starsAbstraction`, the bound on `QueueOut.count` is set to the smaller bound of two counters: `QueueIn.count` and `ClearToSend.count`. This bound can be easily justified by analyzing the code of `Run`. If `maxExecutions` is connected to some counter, the bound on `numExecutions` is set to the value of that counter. Otherwise, it is set to the sum of the counters `QueueIn.count` and `ClearToSend.count`. (This alternative bound is valid for all scheduling algorithms that are currently available in VCC, but it may not be valid for an arbitrary scheduling algorithm.)

Occasionally, a σ-abstraction needs to know the duration of an execution whose signature should be bounded. This is particularly true for σ-abstractions modeling the environment of the device being designed. This information is kept by the object of the class called `starsManager`. At any time, there is a unique object of this class, and a pointer to that object is returned by `CPPBlackBoxModel` member function `mgr`. The manager has a member function `timeWindow` which returns the length of the time window relevant in the current context. As we will describe later, this value may differ during simulation and during STARS.

For example, a module that generates a signal called `Burst` every 625*ms* could have the following σ-abstraction:

```
void env::starsAbstraction() {
  Burst.count.setBound(
    mgr()->timeWindow()/625000+1);
}
```

This σ-abstraction assumes that the time unit used by STARS is 1*ns*. The size of the time unit is also held by `starsManager`, and it can be changed from session to session.

```
//// file: .../perf.dsl ///////
delay_model() {
  input(VoiceFrameIn);
  run();
  delay('0.000001');
  output(PcmSamplesOut);
}

//// file: .../black.cpp //////

...

void QCELP::starsAbstraction() {

  PcmSamplesOut.count.setBound(
    VoiceFrameIn.count);

  starsCounter * processorTime =
    processorTimeCounter();

  if(processorTime) {
    processorTime->setBound(
      1000*VoiceFrameIn.count);
  }
}
```

Figure 3: Bounding processor time based on a delay model

The σ-abstraction of the system 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.

2.4 STARS and timing

VCC-STARS maintains a counter for each Black-Box C++ module mapped to a CPU. The purpose of this counter is to measure CPU time used by that module. STARS increments this counter during simulation, but it is the user's responsibility to set its bound, to indicate worst-case CPU time requirement.

A Block-Box C++ module can access its associated CPU-time counter by a member function `processorTimeCounter`. If the module is mapped to a CPU, the function returns a pointer to the CPU-time counter for that module. Otherwise, it returns 0.

For example, a delay model and a σ-abstraction of a module called QCELP are shown in Figure 3. It indicates that QCELP emits its output exactly $1\mu s$ after reading its input. Since the module uses $1\mu s$ of CPU time each time it is executed, its worst-case usage is set to `1000*VoiceFrameIn.count`. This bound assumes that the module is not executed unless there is a fresh input, which is true of all the currently available scheduling policies in VCC.

In principle, a user could also write σ-abstractions for CPU-time counters of estimated White-Box C modules. However, this would require the user to analyze delay-annotated code generated by VCC. Since the generated code is quite unreadable and bears little resemblance to the original C code, this process is very tedious and error-prone. Instead, we have modified the procedure to generate delay-annotated code. The modified procedure generates a σ-abstraction as a side product. In this way, the details of delay-annotated code may remain hidden from the user.

VCC-STARS also maintains a counter for each CPU to measure its total usage. STARS automatically increments this counter during simulation, and it also generates its σ-abstraction. In the generated abstraction, VCC-STARS sets the bound on the usage counter of a CPU c to the sum of bounds of processor time counters of all the modules mapped to c.

3 Running STARS

The basic theorem behind STARS states that if some vector of counter values x is a fix-point of the system σ-abstraction F, and x is larger than the vector of initial counter values, then x is larger than the signature of *any* execution [2]. 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 at least one counter value in x will grow beyond any bounds. To prevent infinite iteration, the user may specify a boundary value for each counter. If a counter in x reaches its boundary value, the iteration terminates with failure. However, if the convergence is achieved before that, than x is a valid worst-case bound.

STARS is performed by calling `runStars` member function of the `starsManager` object. This function comes in two version. The first version has a single argument `T` of type `double`. In this version, `timeWindow` always returns `T`. Consequently, the results of STARS should be interpreted as the worst-case signature of all execution windows of length `T`. In the second version of `runStars`, the argument is the counter `Cnt` associated with CPU for which we want to perform busy-period analysis. In this case `timeWindow` returns the current bound of `Cnt`, which of course may grow from iteration to iteration. If the iteration converges, the final bound on `Cnt` is also the bound on the duration of a busy period of the associated CPU.

3.1 Monitors

STARS needs only σ-abstractions for its computation, and it never executes modules' `Run` functions. Therefore, STARS cannot check any correspondence between two views of the module (given by `Run` and `starsAbstraction`). For this purpose, we extend VCC with *monitors*. Each Black-Box C++ module 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 `starsAbstraction` function.

In this context, casting an object of class `starsCounter` to integer type evaluates to the difference between the current counter value and the value at the time the monitor object was created. Similarly, `timeWindow` evaluates in this context to the difference between the current time and the time the monitor was created. When `setBound` 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 2.

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 `Run` later in the design process do not invalidate existing σ-abstractions.

4 White-Box C Extensions

The simulation kernel of VCC can simulate only Black-Box C++ models. Therefore, for simulation, White-Box C models need to be converted into Black-Box C++ models. This can be done in two ways. For functional simulation, the translation is mostly syntactic to account for differences between C and C++. Table 1 shows corresponding Black-Box C++ and White-Box C constructs, both for basic VCC and STARS extensions.

For performance simulation, VCC estimates execution times of each basic block of the White-Box C code. Then, VCC generates

Table 1: Black-Box C++ vs. White-Box C

Black-Box C++	White-Box C
`CPPBlackBoxModel` object	.c file
`object::Run()`	`Run()`
`Input.Enabled()`	`Input_Enabled()`
`Input.Value()`	`Input_Value()`
`Output.Post(val)`	`Output_Post(val)`
`starsCounter`	`starsCounter`
`cnt++`	`starsIncrement(cnt)`
`cnt.setBound(expr)`	`cnt_SetBound(expr)`

equivalent Black-Box C++, and adds to every basic block a statement that increments simulation time by its estimated execution time. In VCC-STARS, we have modified this procedure to also generate a σ-abstraction as a side-product. The σ-abstraction generation procedure consists of the following two phases:

Bound propagation In this phase we try to find a bound for each basic block. We start with a small set of *seed bounds*, and then deduce from these additional bounds, until hopefully we get a bound for each output counter and each basic block. The seed bounds consist of bounds on input counters, and also any user-given bounds on output counters. We describe the bound propagation procedure in details in Section 4.1.

Code generation In this phase, we use the bounds from the first phase to generate the code of `starsAbstraction`. The code sets bounds on all output counters, as well as on the `processorTimeCounter`. We describe this procedure in Section 4.2.

4.1 Bound Propagation

To build a σ-abstraction for a White-Box C module, we need to find a bound for each output counter (as a function of input counters). Furthermore, to bound required processor time, we need to find a bound for each basic block of the `Run` function, or any function called by `Run`. More precisely, the bound propagation procedure may generate three types of bounds:

1. For every counter (both built-in and user-defined), we try to find a bound on its value. The value of a counter c is denoted by $V(c)$.

2. For every statement, we try to find a bound on the total number of times that statement is executed. We use $N(s)$ to denote this number for some statement s. Clearly, a bound on $N(s)$ is also a bound on the number of executions of the basic block to which s belongs.

3. For every expression appearing in a conditional statement (and its sub-expressions), we try to find a bound on the number of times that expressions can evaluate to TRUE (i.e. to some value different from 0). We use $T(e)$ to denote this number for some expression e.

Because we are looking for a bound on each $N(s)$, $T(e)$, and $V(c)$, we jointly call them *targets*.

The bound propagation procedure takes seed bounds and derives from them as many bounds on targets as possible. The bound propagation procedure is rule-based. Each rule specifies how to generate new bounds from existing ones. Rules are iteratively applied, until no further bounds can be derived.

The rules that we propose next are valid for a subset of White-Box C that conforms to rules of structure programming, i.e. no

```
/****     file: .../white.c   *****/
...
starsCounter c1,c2,c3;

void Run() {
  starsIncrement(c1);
  if (PcmSamplesIn_Enabled()) {
    samplesin = PcmSamplesIn_Value();
    Length = samplesin -> length;
    lengthReceived_ = 1;
    Data = samplesin -> data;
    dataReceived_ = 1;
  }
  if (lengthReceived_ && dataReceived_) {
    starsIncrement(c2);
    for (i = 0; i < Length; i++) {
      starsIncrement(c3);
      data[i] = *Data++;
    }
    PCMSamplesFromDECODER(data, Length);
    lengthReceived_ = 0;
    dataReceived_ = 0;
  }
  ...
}

void starsSetBounds() {
  c3_SetBound(160*PcmSamplesIn_Count());
  c2_SetBound(PcmSamplesIn_Count());
  c1_SetBound( ... );
  ...
}
```

Figure 4: A White-Box C model of an audio buffer

`goto`, `break`, or `continue` statements, and no `return` statements except as the very last statement in a function. Furthermore, we do not allow `switch` statements, nor recursive calls.

Consider the statement `starsIncrement(c1)` in Figure 4, and assume b is some known bound for $V(\texttt{c1})$. We can conclude that the statement cannot be executed more than b times. Otherwise, the bound on $V(\texttt{c1})$ would be violated. This reasoning can be represented by the rule:

$$N(\texttt{starsIncrement(c1);}) \leq V(\texttt{c1}) \ .$$

Once we have a bound on `starsIncrement(c1);`, we can extend it to `if(PcmSamplesIn_Enabled())`, because these two statements are in the same basic block. In other words, the following rule holds:

$$N(\texttt{if(PcmSamplesIn\_Enabled())} = N(\texttt{starsIncrement(c1);}) \ .$$

As a final example, consider the statement:

`samplesin = PcmSamplesIn_Value();`.

It is executed if `PcmSamplesIn_Enabled` evaluates to TRUE. In VCC, executions of the `Run` function "consume" all input tokens. In other words, in a single execution of `Run`, the value of `PcmSamplesIn_Enabled` will always be the same, but it can be TRUE only if a new `PcmSamplesIn` has been generated between the current execution and the previous one. It follows that any bound on the counter of `PcmSamplesIn` can be used to bound the number of times `PcmSamplesIn_Enabled` evaluates to TRUE, and thus also the number of times the statement above is executed.

Table 2: Bound propagation rules for statements

statement s	rules
$p;q$	$N(s) = N(p) = N(q)$
`if(`e`) then` p `else` q	$N(s) \leq N(p) + N(q)$ $N(p) \leq N(s)$ $N(p) = T(e)$ $N(q) \leq N(s)$
`for(`$e;f;g$`)` p	$N(s) = N(e)$ $N(g) = N(p) = T(f)$
`while(`e`)` p	$N(p) = T(e)$
`do` p `while(`e`)`	$N(s) \leq N(p)$ $N(p) \leq T(e) + N(s)$
`starsIncrement(c);`	$N(s) \leq V(\texttt{c})$
`out_Post(token);`	$N(s) \leq V(\texttt{out.count})$

Table 3: Bound propagation rules for expressions

expression e	rules
`in_Enabled()`	$T(e) \leq M(e) * V(\texttt{in.count})$
f && g	$T(e) \leq T(f)$ $T(e) \leq T(g)$
$f \| g$	$T(e) \leq T(f) + T(g)$

This can be derived from the following rules:

$$N(\texttt{samplesin=PcmSamplesIn\_Value();}) = T(\texttt{PcmSamplesIn\_Enabled()})$$

$$T(\texttt{PcmSamplesIn\_Enabled()}) = V(\texttt{PcmSamplesIn.count}) * M(\texttt{PcmSamplesIn\_Enabled()}) ,$$

where $M(e)$ represents a bound on the number of times e can be executed during a single execution of the `Run` function. In our example $M(e)$ is one, but in general it can be conservatively computed by standard flow analysis techniques (e.g. [12]). In particular, if e appears inside a loop, we assume $M(e)$ is infinity, and ignore the corresponding rule. Notice that $M(e)$ is an integer constant (if finite) that can be determined at compile time, while $N(s)$, $T(e)$ and $V(c)$ are functions of input counter values. Thus, they can be symbolically manipulated at compile time, but can be evaluated only at run-time (i.e. during simulation or STARS).

The rules used by the bound propagation procedure are shown in Tables 2 and 3. Table 2 shows rules associated with statements, while Table 3 shows rules associated with expressions. There are additional rules related to output counters and function calls. They are:

$$\begin{aligned} V(\texttt{cnt}) &\leq \sum_{s \in Inc(\texttt{cnt})} N(s) \\ V(\texttt{out.count}) &\leq \sum_{s \in Post(\texttt{out})} N(s) \\ N(p) &\leq \sum_{s \in Call(p)} N(s) \ , \end{aligned}$$

where:

- `cnt` is a user-defined counter, and $Inc(\texttt{cnt})$ is the set of statements of the form `starsIncrement(cnt);`,
- `out` is an output port, and $Post(\texttt{out})$ is the set of statements of the form `out_Post(val);`,
- p is the first statement in the body of a function, and $Call(p)$ is the set of all statements that contain a call to that function.

The bound propagation procedure maintains a set of bounds $B(X)$ for each target X. Initially, target statements and expressions have no bounds, input counters have a single bound (themselves), and output counters have any bounds specified by the user in the function called `starsSetBounds`. The bound propagation procedure iteratively applies one of the propagation rules until all sets of target bounds stabilize. There are two types of rules:[2] one is of the form $X \leq c * Y$, and the other is of the form $X \leq Y + Z$, where X, Y, and Z are targets, and c is an integer constant (usually 1). To apply the rule of the first type, the bound propagation procedure performs the following operation:

$$B(X) \ := \ B(X) \cup \{c * y | y \in B(Y)\} \ .$$

To apply the rule of the second type, the bound propagation procedure performs the following operation:

$$B(X) \ := \ B(X) \cup \{y + z | y \in B(Y), z \in B(Z)\} \ .$$

It can be shown that the bound propagation procedure will eventually converge, as long as no redundant bounds are kept. The procedure is successful if at least one bound is derived for each target. If that is not the case, `starsAbstraction` cannot be generated, and a warning is reported instead.

Consider the example in Figure 4. Without user given bounds, the bound propagation procedure can only find bounds on the basic block inside the first `if` statement. So, to make the procedure successful, the user needs to define some counters and provide their bounds. For example, counter `c1` is used to bound the total number of executions of `Run`. Similarly, `c2` is used to bound the number of executions of the basic block inside the second `if`. Note that the bound on `c1` propagates also inside the second `if`, so the bound propagation procedure would be successful even without a bound on `c2`. However, overall precision of the σ-abstraction can be improved, if `c2` is given a stronger bound than `c1`. Finally, the bound propagation procedure cannot be successful without a bound on the basic block inside the loop. This bound is derived from the bound on `c3` (this bound is not apparent from the code in Figure 4, but it is asserted elsewhere that a packet cannot contain more that 160 samples).

For the example in Figure 4, a more sophisticated procedure could find a bound on all targets without any user given bounds. A bound for `c1` could be derived by analyzing the scheduling algorithm. This information is readily available in VCC, but no such analysis have been developed yet. A bound on `c2` could be obtained by finding an invariant stating that the two `if`'s are always satisfied together. Finally, a loop analysis technique could be used to find a bound for `c3`. However, there is no hope of developing a procedure so sophisticated, that it totally eliminates a need for user given bounds. It is not hard to see that such a procedure would have to be able to solve the halting problem, which is known to be undecidable.

4.2 Code Generation

The generated `starsAbstraction` consists of two parts. In the first part we evaluate bounds on output counters, and basic blocks of code. For example, to evaluate a bound on some counter c, we evaluate all bound expression for $V(c)$, and find the smallest. Bounds for $N(s)$ and $T(e)$ for a statement s and a conditional expressions e are evaluated in the same way. Finally, the bound on the number of executions of some basic block b (denoted by $N(b)$) is set to the bound on $N(s)$ for some statement s in b. Let

[2] A rule with equality can be converted into two rules with inequalities.

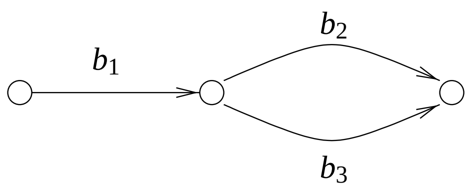

Figure 5: A control-flow graph.

$bound(N(s))$, $bound(N(b))$, $bound(V(c))$, and $bound(T(e))$ denote these bounds.

In the second part of `starsAbstraction` we compute a bound on the required processor time. For this purpose we need a *control-flow graph* of the `Run` function, and all functions called by `Run`. Formally, we represent a control-flow graph with a 4-tuple $(Nodes, Blocks, In, Out)$ where $Nodes$ is the set of nodes in the graph, the set of basic blocks $Blocks$ correspond to edges of the graph, and functions $In : Nodes \mapsto 2^{Blocks}$, and $Out : Nodes \mapsto 2^{Blocks}$ specify for each node its fan-in and fan-out, respectively. We also make use of estimated execution time of basis blocks, as computed by VCC estimation package. We use $est(b)$ to denote that time for a basic block b.

In the second part of `starsAbstraction`, we set the bound on the required processor time to:

$$\sum_{b \in Blocks} bound(b) * est(b) \quad . \tag{1}$$

Expression (1) is a valid, but not the best possible bound on the required processor time. For example, consider the control-flow graph in figure 5 and assume:

$$bound(N(b_1)) = bound(N(b_2)) = bound(N(b_3)) = 2 \quad .$$

Expression (1) would bound the processor time requirement of this graph to:

$$2 * (est(b_1) + est(b_2) + est(b_3)) \quad .$$

However, from the bound on b_1, we know that b_2 and b_3 cannot both be executed two times, and the bound above can be strengthen to:

$$2 * (est(b_1) + \max(est(b_2), est(b_3))) \quad .$$

To capture such constraints in general, instead of evaluating (1), we could solve the following linear programming problem for a vector of variables x indexed by the basic blocks:

$$\begin{array}{llll} \max_x & \sum_{b \in Blocks} est(b) * x[b] & & \\ \text{subject to} & x[b] \leq bound(b) & & \forall b \in Blocks \\ & \sum_{b \in In(n)} x[b] = \sum_{b \in Out(n)} x[b] & & \forall n \in Nodes' \quad , \end{array}$$

where $Nodes'$ is the set of all nodes except for the unique source and sink nodes. This approach is a generalization of the *program path analysis* presented in [12]. Intuitively, $x[b]$ represents the number of times b is executed in the worst case. It has to respect both the derived bounds and the flow constraints in the graph.

5 Examples

We have applied the `starsAbstraction` generation procedure to the White-Box C model of *audio buffer* component of the voice-mail pager in the VCC library. To make the bound propagation procedure successful, we had to define four additional counters. Three of these counters are shown in Figure 4, and the fourth one is used to bound the number of passes through a `for` loop appearing elsewhere in the code. The function specifying the seed bounds has only ten lines. It required a minor additional designer's effort, compared to almost three hundred lines specifying the functionality of the audio buffer.

To measure the level of automation that our procedure provides, it is reasonable to compare the sizes of the seed bounds and the generated `starsAbstraction`. The comparison is favorable as the generated `starsAbstraction` is two orders of magnitude larger than the the function specifying seed bounds. The size difference drops to (still significant) one order of magnitude when the comparison is made to a hand-crafted `starsAbstraction` instead of the generated one. The order of magnitude difference between hand-crafted and generated versions can be attributed partly to the more comprehensive set of bounds expressed in the generated version, and partly to sub-optimal code generation.

We have also applied STARS busy-period analysis to the complete voice-mail pager in the VCC library. The design has a total of 13 modules, 4 of which are intended to model the environment. We have studied a case where 9 other modules where all implemented in software running on a single processor. The total size of the design was approximately 2500 lines of C code. The pager needs to service several periodic and aperiodic requests. The most frequent one of these repeats every $125\mu s$, hence the requirement that the maximum busy-period be less than $125\mu s$.

The original simulation test-bench for the design tested the scenario where a single message was received and then played. To compare the results of the busy-period analysis to those of simulation, we have developed a σ-abstraction of the environment that is valid for that case (single message only). The longest busy-period observed in the simulation trace was $82\mu s$, while the busy-period analysis provided an upper bound of $83\mu s$. These results differ by less than 2%, and they were both well within the $125\mu s$ requirement.

In the second experiment, we have developed a σ-abstraction of the environment that was no longer limited to a single message. In that case the busy-period analysis gives a bound of $148\mu s$ (violating the $125\mu s$ requirement). Careful analysis of the results indicated that this worst case can appear only if one message is received and then played, while another message is being played. Based on this information, we were able to construct a simulation trace which contains a $146\mu s$ busy-period.

The overhead of STARS was very low in this case too. The size of σ-abstraction was less than 200 lines of code, indicating that additional designer's effort is minimal. We cannot compare the efforts more precisely, because the original design was developed a long time ago, by a different designer, requiring an unknown effort. CPU time required by STARS was negligible compared to simulation times (fractions of a second vs. minutes).

6 Conclusions

STARS can help find bugs and increase confidence in a design by complementing simulation with worst-case analysis. We have demonstrated that STARS and simulation can be integrated in a single environment, and that additional effort of developing abstract models used by STARS is minor compared to the development of simulation models, and pays off in better verification. The designer's effort is further decreased if automatic abstraction generation is used. In this case, the designer still has to specify a small number of seed constraints, but since the problem is undecidable in general, this is unavoidable.

References

[1] Neil C. Audsley, Alan Burns, M. Richardson, Ken W. Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, September 1993.

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

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

[4] Felice Balarin. STARS of MPEG decoder: a case study in worst-case analysis of discrete-event systems. In *Proceeding of CODES-01 Symposium*, 2001.

[5] S. Bensalem, A. Boujjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. v. Bochmann and D.K. Probst, editors, *Proceedings of Computer Aided Verification: 4th International Workshop, CAV '92, Montreal, Canada, June 29-July 1, 1992*. Springer-Verlag, 1993. LNCS vol. 663.

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

[7] Cadence Virtual Component Co-design (VCC) . http://www.cadence.com/datasheets/vcc.html.

[8] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proc. Principles of Programming Languages*, January 1992.

[9] 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.

[10] 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.

[11] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[12] Y-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[13] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-realtime environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.