

# Analysis of the SEI PACC Starter Kit

---

## Team – VdashNeg

Berin Babcock-McConnell

Saurabh Gupta

Jonathan Hartje

Shigeru Sasao

Sidharth Surana

## Tool Description

Our studio project is based upon the PACC Starter Kit (PSK) from the SEI. PACC stands for Predictable Assembly from Certifiable Code. This technology is composed of Construction and Composition Language (CCL), Component Formal Reasoning Technology (ComFoRT), Copper, and the performance reasoning framework (Lambda-\*). CCL is an architecture description language and is used to specify components and their assemblies. ComFoRT is a reasoning framework for predicting whether a system will satisfy its safety, reliability, and security requirements. Copper is a core component of ComFoRT and is a model checker for C programs. To succinctly describe the relationship of the above entities, CCL is used to describe a program. ComFoRT then translates this description into C. Finally, Copper is used to analyze the resultant program for its conformance to safety, reliability, and security requirements. Lambda-\* is the performance reasoning framework that analyses component assemblies written in CCL for deadline conformance and whether tasks are schedulable.

## Behavioral Analysis

By way of ComFoRT, Copper is used to perform a behavioral analysis of a program created with CCL. Finite state machines, also known as Labeled Transition Systems (LTSs), are used to describe the behavior of the program to be analyzed. An FSP-like notation is used to specify the behavior of LTSs and an extension to the FSP syntax allows C functions to be linked to LTSs. The C program source is also annotated to indicate information, such as error states, to Copper.

Using simple LTSs claims can be made regarding state reachability. In this matter it can be determined whether a program behaves in a manner consistent with the LTS or not. The following example is taken from the tutorial included in the PACC Starter Kit and describes a claim that ensures that a “... philosopher always performs the actions pick\_left and put\_left alternately, starting with pick\_left.” [SEI2008]

```
PhilSpec1 = ( pick_left -> put_left -> PhilSpec1 ).
```

Copper also supports temporal logic claims expressed in State/Event Linear Temporal Logic (SE-LTL). The syntax of SE-LTL is similar to LTL with the addition of the following logical and temporal operators:

&	Conjunction
	Disjunction
!	Negation
#X	Next time
#U	Until
#G	Globally
#F	Eventually
#R	Release

These operators can be combined to make assertions about events that will, or will not, happen. The following example is taken from the tutorial included in the PACC Starter Kit and says that "... whenever [a] philosopher is eating, it eventually always releases its left fork." [SEI2008]

```
ltl PhilSpec4 { #G ( [P0::eating == 1] => #F put_left ); }
```

LTS and SE-LTL claims can be associated with the C program source using program blocks. The following example, based upon the tutorial included in the PACC Starter Kit, describes how to do this.

```
program philosopher {  
    specification abs_1, {1}, PhilSpec1;  
    specification abs_4, {1}, PhilSpec4;  
}
```

This example links the claims described above with a function named "philosopher" in the C program source. Provided this linkage to the C program source Copper can examine the source for its conformance to the LTS and SE-LTL claims.

## Performance Analysis

PACC can analyze the performance of systems that are constructed in CCL. It uses the principles of General Rate Monotonic Analysis (GRMA). In the basic analysis described by [LIU73], tasks are determined to be schedulable if it satisfies the following equation:

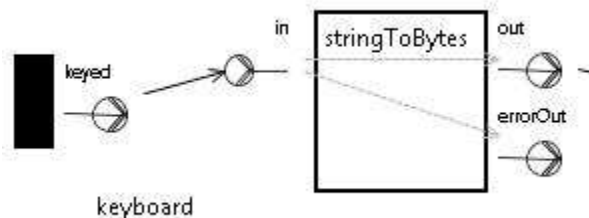
$$\sum_{i=1}^n \frac{C_i}{T_i} \leq U(n) = n \left( 2^{\frac{1}{n}} - 1 \right)$$

$C_i$  and  $T_i$  are execution time and period of all tasks. However, this equation assumes periodic tasks that are prioritized according to rate monotonic scheduling (shortest period is highest priority). The performance analyzers in PACC extend this model, to support aperiodic tasks and preemption of tasks through varying priorities. [MORENO2009]

In order to define a task, CCL uses the notion of sink and source pins to connect components to one another. Each sink pin is considered a task, which will be used as input to the analyzer. For example, the following notation,

```
keyboard:keyed ~> stringToBytes:in;
```

is used to connect the source pin of the keyboard to the sink pin of the string-to-bytes converter. Visually, it would look as follows:



**Figure 1.** Source pin to sink pin connection in PACC/CCL

As described earlier, the input to the rate monotonic analysis is the priority, execution time and period of each task. In CCL, each sink pin is annotated with its priority and execution time. For example, for the string-to-bytes converter, we will use the following annotations:

```
annotate stringToBytes:Work {"Pin", const int priority = 195}
annotate stringToBytes:in {"lambda*", const string execTime =
"G(0.107700, 0.125083, 0.170000)" }
```

The period of the task may depend on its pin construct. For example, the period of the string-to-bytes converter depends on the period of the keyboard. Thus, we define the period of the keyboard as follows:

```
annotate keyboard:keyed {"lambda*", const string
eventDistribution
= "C(10)" }
```

The event distribution can be annotated with minimum, average, and maximum values to simulate an aperiodic stochastic running time.

From the information of the architectural construction provided by CCL and the inputs as defined by annotations (priority, execution time, period), PACC allows many different implementations of performance analyzers to provide information on deadline conformance and tasks schedulability. The PACC Starter Kit (PSK) provides four different implementations of performance analyzers such as closed form worst case analysis (MAST) and simulation based worst case analysis (SIM-MAST).

## Experimental Setup and Results

The application that we are building using PACC is an intelligent controller for a commercial off the shelf robot called “SRV-1 Surveyor Robot”. The controller must have the following capabilities:

1. Run the robot using the basic movement commands
2. Capture image feedback from the camera mounted on the robot to perform image processing
3. Identify path markers at runtime and instruct the robot to follow it.
4. Identify any enemy targets along the way (colored balloons) and eliminate them by “firing” lasers at them.

For carrying out the experimentation for the tool’s analysis capabilities, we used our prototype code base. The prototype is capable of performing the first two functionalities listed above. In order for the prototype to function properly, we need to ensure the correctness of its behavior and ensure that it adheres to certain real time performance benchmarks. Thus, in the following sections we will provide details about the analysis frameworks we used to ensure this.

## Behavioral Analysis

In order for the robot to display correct behavior, we had to ensure that the components composing the system adhere to certain behaviors. Some of them include:

1. The “Gateway” component responsible for communication between the robot and the controller should ensure:
  - a) For every command that is received from within the controller it will be eventually sent out to the robot.
  - b) There is nothing sent to the robot unless a command is received from within the controller. This is important to ensure that there is no garbage value sent to the robot that might affect the robot activity.
  - c) For every response that it receives from the robot it will pass that response to another component in the controller.
  - d) There is nothing sent to another component in the controller unless a response is received from the robot. This is important to ensure that there is no garbage value fed into the controller that might affect the subsequent decision the controller makes.
2. The “Response Router” component responsible to separate the image data from the command response received from the robot to appropriate components in the controller system should ensure:
  - a) If the image data is received then it is sent to the imageBuffer sink pin for aggregating and displaying the image.
  - b) If a command response is received then it is sent to the pin connected to the text console.
3. Once the ImageBuffer component aggregates the image, it will send it to another component to display on the screen. The component should ensure:
  - a) The image data is aggregated till the time the end of image is detected.

- b) The image is sent only when aggregation is completed.

The behavioral claims in PSK are written using the Linear Temporal Logic (LTL) syntax. Some of the claims in the CCL language are as follows:

```
// Good Claim - should hold
// For every command that is receives from within the controller
// it will be eventually sent out to the robot
    annotate ProcessIncomingDataFromRobot{"comfort",
    const string ResponseAlwaysPassed = "G([ResponseRecieved]) =>
        F([ResponseSentToBrain])"}

//Bad Claim - should not hold
// Response is sent to the brain even when nothing is received from
the robot.
    annotate ProcessIncomingDataFromRobot{"comfort",
    const string ResponsePassedWithoutReceiving =
    "F([ResponseSentToBrain &&
        !ResponseRecieved])"}

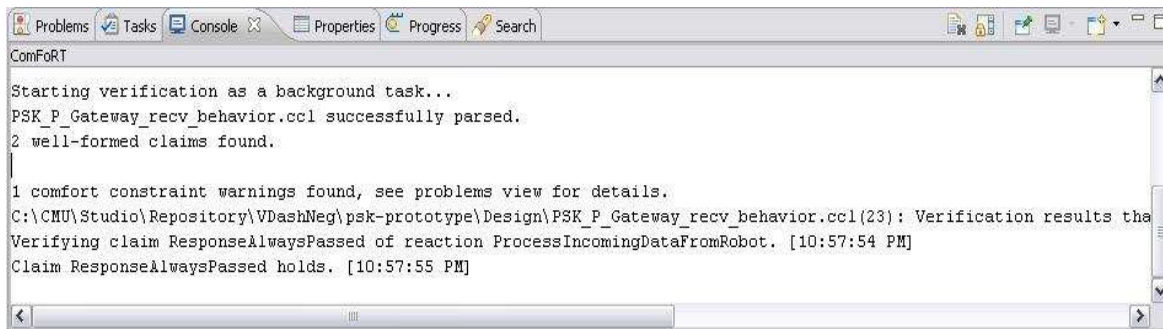
//if the response is a command then the contents will be sent to the
// print_image pin
    annotate ProcessReceivedRobotResponse{"comfort",
    const string ResponseSentToPrint = "G(!wait
    && !isImage)=>F([sent2print])"}

//if the response is image then the contents will be sent to the
send_image pin
    annotate ProcessReceivedRobotResponse{"comfort",
    const string ImageSentForDisplay = "G(!wait &&
        isImage)=>F([sent2display])"}

```

In the last claim above “annotate” is the keyword. “ProcessReceivedRobotResponse” is the name of a threaded reaction in the CCL code. “comfort” is the analysis framework class to be used for evaluating the claim, in this case we are using the ComFoRT reasoning framework. “ImageSentForDisplay” is the name identifier for the claim. The expression following the identifier is the LTL code representing the claim.

Below is the sample output of the reasoning framework after verifying a claim.



The screenshot shows a console window titled 'ComFoRT' with the following text:

```
Starting verification as a background task...
PSK_P_Gateway_recv_behavior.ccl successfully parsed.
2 well-formed claims found.

1 comfort constraint warnings found, see problems view for details.
C:\CMU\Studio\Repository\VDashNeg\psk-prototype\Design\PSK_P_Gateway_recv_behavior.ccl(23): Verification results the
Verifying claim ResponseAlwaysPassed of reaction ProcessIncomingDataFromRobot. [10:57:54 PM]
Claim ResponseAlwaysPassed holds. [10:57:55 PM]
```

## Performance Analysis

Our prototype interfaces with an external entity (the robot) outside of the software system. The software system must ensure that it meets the performance quality attributes so that the robot will not be impacted by delayed instructions (jitter movements, missed responses). Thus, we would like to reason about the performance of the system, to ensure that the tasks are schedulable and deadlines will be met for aperiodic tasks. We will analyze the message sending part of our system for deadline conformance. The following pin construct is defined in our prototype:

```
// sending message to robot
keyboard:keyed ~> stringToBytes:in;
stringToBytes:out ~> bytesToNetBytes:in;
bytesToNetBytes:out ~> gatewaySend:netBytes_in;
gatewaySend:send_robot ~> netGateway:netWrite;
```

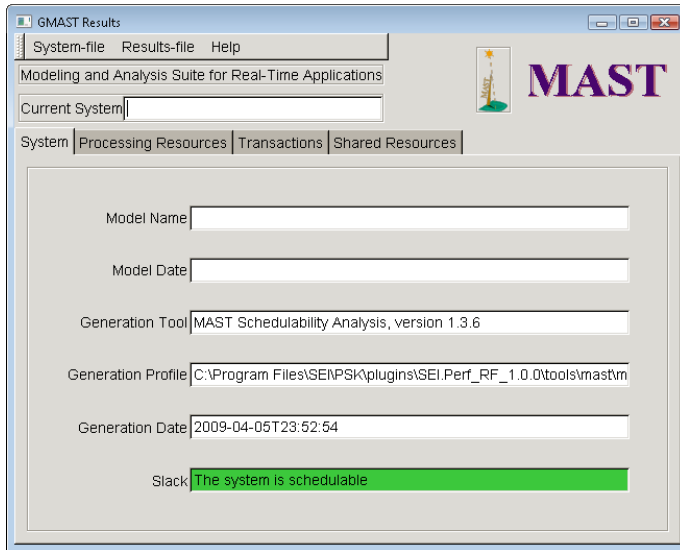
This shows the internal structure of the system, where it receives keyboard input from the user, converts the data type, and sends the command to the robot. For the purpose of this analysis, we have imposed a hard deadline of 0.1 milliseconds from the time when the keyboard is pressed, and the message is sent to the robot.

As mentioned earlier, the General Rate Monotonic Analysis conducted by PACC requires the priority, period and execution time of each component. Thus, our prototype was annotated with this information. For example, the string-to-bytes data converter was annotated as follows:

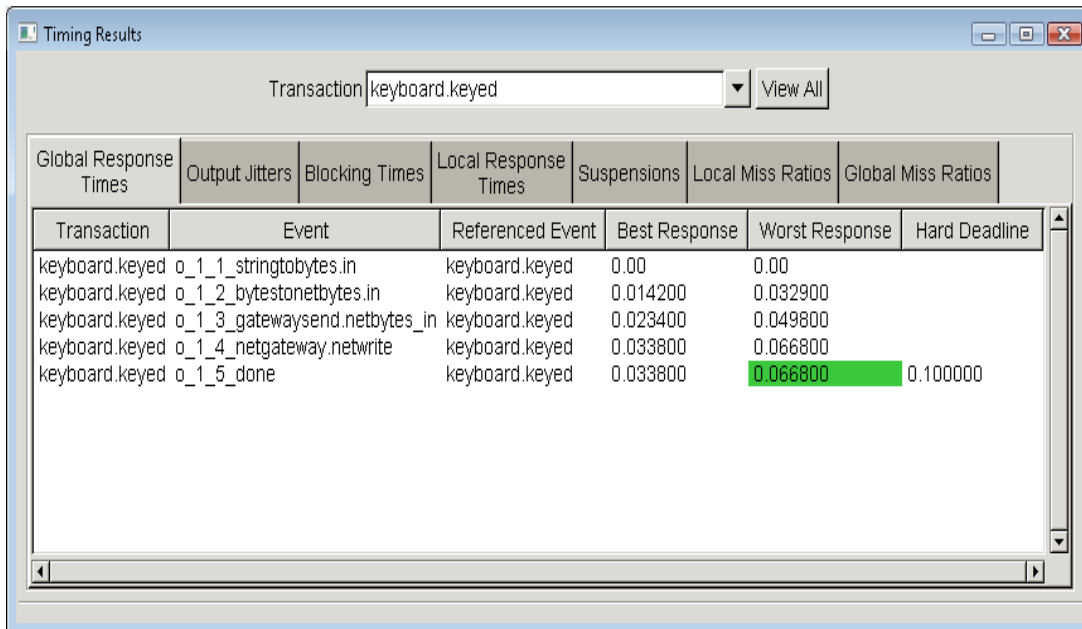
```
annotate stringToBytes:Work {"Pin", const int priority = 195}
annotate stringToBytes:in {"lambda*", const string execTime =
"G(0.107700,0.125083, 0.170000)"} }
```

Note that the period of the task depends on the initial input from the keyboard, so it was not explicitly annotated for the string-to-byte converter.

Once all necessary information was annotated, we ran the analyzer to check the result. The following output was produced indicating that the tasks above are schedulable for the 0.1 millisecond deadline:



We are also able to see the details of the analysis for individual components:



As you can see, the predicted results show that the worst response by the components was 0.0668 milliseconds. PACC also has a function to take actual runtime measurements. So, we can compare the predicted values to the actual runtime.

	Predicted Best Response	Actual Best Response	Predicted Worst Response	Actual Worst Response
stringToBytes	0	0	0	0
bytesToNetBytes	0.0142	0.0142	0.0329	0.0329
gatewaySend	0.0234	0.0234	0.0498	0.047
netGateway	0.0338	0.0346	0.0668	0.0627

**Table 1.** Predicted vs. Actual Comparison

As seen from the above table, the predicted values and the actual values are almost identical. Thus, the output by the performance analysis holds, and we are able to guarantee that the components are schedulable for the performance deadline of 0.1 milliseconds.

## Conclusion

### False/True Positives

Unlike some analytic tools which check the entire source code for errors, the behavioral and performance reasoning frameworks allow the developers to specify precisely what they want to check. Hence, no false positives are reported, and no irrelevant true positives are reported. As discussed in the experiment results, we were able to guarantee our prototype code with 8 behavioral claims, and 4 tasks were confirmed to be schedulable. We also confirmed predicted values for all 4 tasks used in the performance analysis to match the actual values from measurement. On the other hand, the analysis relies upon the annotations in the source code. If the source code's behavior is not correctly specified the analysis may fail to reveal an undesired behavior or incorrectly demonstrate a valid behavior incorrect.

### Benefits/Drawback

The formal specification of the state machine and architectural construction through CCL allows many analyses to be run against the software. Behavior and performance analyses are only two of many possibilities. Security, memory footprint, and other analyses are also possible through the formalized specification. Thus, components in the architecture can be certified to behave according to certain quality attributes that have been predicted through the reasoning frameworks. For example, embedded systems can be guaranteed to perform within deadline conformance, and safety critical systems can be guaranteed to maintain certain invariants.

The drawback to using PACC is that it restricts the power and flexibility of the programming language so that analyses can be performed against it. Although CCL is similar to C, it has no pointers, no bit shifting, and dynamic data structures are not allowed. This is because COPPER, which is the model checker, cannot properly check the behavioral claims if CCL has too much flexibility. CCL has a feature called verbatim where C code can be embedded into CCL with the full feature of the C library, but the use of verbatim but the use of verbatim will not allow COPPER to properly check behavioral claims.



The concepts behind PACC are very powerful. Predictable assembly of software components would be a new paradigm to quality assurance, but further advancements would be required in model checkers and other analysis techniques to lessen the restrictions imposed by the analyzable language.

## References

[SEI2008] PACC Starter Kit, Software Engineering Institute, <http://www.sei.cmu.edu/pacc/starter-kit.html>, 2008

[LIU73] Liu, C. L., Layland, James W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Association for Computing Machinery, Vol. 20, 1973

[MORENO2009] Moreno, Gabriel A., Hansen, Jeffrey, *Overview of the Lambda-\* Performance Reasoning Frameworks*, Software Engineering Institute, Carnegie Mellon University, 2009