Model-driven Specification of Component-based Distributed Real-time and Embedded Systems for Verification of Systemic QoS Properties

James H. Hill
Vanderbilt University
Nashville, TN, USA
j.hill@vanderbilt.edu

Aniruddha Gokhale
Vanderbilt University
Nashville, TN, USA
a.gokhale@vanderbilt.edu

Abstract

The adage "the whole is not equal to the sum of its parts" is very appropriate in the context of verifying a range of systemic properties, such as deadlocks, correctness, and conformance to quality of service (QoS) requirements, for component-based distributed real-time and embedded (DRE) systems. For example, end-to-end worst case response time (WCRT) in component-based DRE systems is not as simple as accumulating WCRT for each individual component in the system because of inherent complexities introduced by the large solution space of possible deployment and configurations. This paper describes a novel process and tool-based artifacts that simplify the formal specification of component-based DRE systems for verification of systemic QoS properties. Our approach is based on the mathematical formalism of Timed Input/Output Automata and uses generative programming techniques for automating the verification of systemic QoS properties for component-based DRE systems.

Keywords component-based distributed real-time and embedded systems, formal specification, generative programming, model-driven engineering, Timed I/O Automata, system verification

1 Introduction

Complexities in verifying systemic properties of component-based DRE systems. Component-based middleware technologies such as the Lightweight CORBA Component Model (LwCCM) [15] are becoming the de facto choice for implementing distributed real-time and embedded (DRE) systems. One benefit of using component-based technologies is it allows developers to capture the application's "business-logic" into components that can be assembled (*i.e.*, wired together) to realize a deployed system. Moreover, individual components or partial assemblies can be reused across multiple application domains thereby preventing reinvention of core intellectual property.

Although component technologies raise the level of abstraction to improve software development productivity and reuse [7], system developers who use such technologies possess reduced knowledge of complete system behavior. Existing techniques for verifying systemic properties [3,8,19], such as deadlock free execution or correctness, rely heavily on formal methods, such as automata-based languages [3, 5, 12, 14, 20]. In order to formally specify a component-based DRE system, however, developers must possess complete knowledge of the deployed system's behavior.

In general, formal specification and verification of component-based DRE systems require system developers to understand the behavior of the system's artifacts, such as the intercommunication channels between components, threads scheduled to handle events received by components, and individual components of the system. The behavior of the individual components, however, is the only aspect of the entire system within system developer's knowledge domain. To verify systemic QoS properties, such as worst-case response time (WCRT), is even "harder" for system developers because it also requires them to understand and formally specify the intricate timing aspects of all artifacts of the system in insolation and in conjunction with one another.

Consequently, it is inherently difficult for component-based system developers to formally specify the deployed system's complete behavior because both the details of the system and—more so—the use of formal methods is outside their knowledge domain. System developers, therefore, need new techniques and tools that operate within their knowledge domain, *i.e*, at the application level, and simplify—as much as possible—the verification process of component-based DRE systems.

Solution approach → Model-driven verification of component-based DRE systems. Ideally, to simplify component-based DRE system verification, developers need tools and techniques that automatically transform high-level behavior specifications (*i.e.*, specifications within their knowledge domain) into formal specifications thereby alleviating many of the accidental complexities of component-based system verification. Model-driven engineering (MDE) [17] is showing promise in addressing many complexities within component-based system devel-

opment [18, 21, 22]. MDE raises the level of abstraction via domain-specific modeling languages (DSMLs) [13] so system developers construct models using artifacts that are related to their domain.

This paper describes our approach of formally specifying component-based DRE systems for verifying QoS properties, such as WCRT, which is realized within an MDE tool called *CUTS Uses TIOA for System Verification (CUTS$_V$)*. CUTS$_V$ uses a standalone behavioral DSML called the Component Behavior Modeling Language (CBML) [9]. CBML is based on the mathematical formalism of Timed Input/Output Automata (TIOA) [12] and designed to integrate with DSMLs that model only system structure, *e.g.*, the Platform Independent Component Modeling Language (PICML) [2]. System developers use CBML to model the behavior of individual components and use the hosting structural DSML to model the system's composition. Model interpreters then auto-generate TIOA configuration files that can be used in model checkers and simulators to verify systemic QoS properties. The approach can also be generalized to verify other properties, such as state reachability, safety, and deadlocks.

Our approach is designed so developers (1) do not have to possess complete system knowledge to model the behavior of their system, and (2) work within their knowledge domain since the formal specifications, such as TIOA files, are automatically generated from the CBML models. System developers, therefore, can focus more on verifying systemic properties as opposed to manually constructing formal models of their deployed system.

Paper Organization. The remainder of this paper is organized as follows: Section 2 presents a case study in which we highlight several challenges for verifying component-based systems; Section 3 describes our preliminary work CUTS$_V$ for verifying QoS properties of component-based systems; Section 4 discusses related work; and Section 5 provides concluding remarks and future research directions.

2 Challenges in Verifying Systemic QoS Properties of Component-based DRE Systems

This section describes the challenges in developing a framework for verifying QoS properties of component-based DRE systems. We use a case study to highlight these challenges.

2.1 A Shipboard Computing Environment Case Study

We use a representative example drawn from the shipboard computing domain called the SLICE scenario [10] as a case study to illustrate challenges of verifying QoS properties of component-based DRE systems. We also use the example to show how our research artifacts described in this paper enable us to construct a framework for verifying QoS properties of component-based DRE systems.

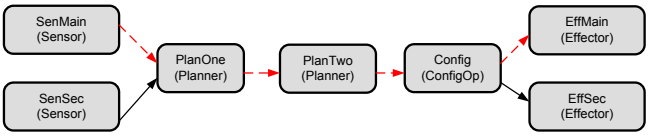

Figure 1. High-level structural composition of the SLICE scenario.

Figure 1 shows the high-level structural composition of the SLICE scenario. The rectangular objects in Figure 1 represent the individual components that are assembled to create the SLICE scenario. The directed lines between the components represent the communication channel for inter-component communication, such as an event channel. As illustrated in Figure 1, the SLICE scenario consists of 7 different component implementations: *SenMain*, *SenSec*, *PlanOne*, *PlanTwo*, *Config*, *EffMain*, and *EffSec*. Although the SLICE scenario is assembled from 7 different component implementations, it only has 4 different component types: *Sensor*, *Planner*, *ConfigOp*, and *Effector*.

The component implementations in the SLICE scenario are deployed across 3 computing nodes because the workload generated by the deployed system cannot be handled by a single host. Moreover, `SenMain` and `EffMain` are deployed on separate nodes to reflect the placement of physical equipment in the production shipboard environment. Finally, events that propagate along the inter-component communication path marked by the dashed (red) lines in Figure 1 have an end-to-end WCRT of 350 msec.

2.2 Impediments to Verifying Systemic QoS Properties of Component-based DRE Systems

There are many ways to deploy (*i.e.*, place components on nodes) and configure (*i.e.*, set component properties) the SLICE scenario. Only a subset of possible deployment and configurations (D&Cs), however, will achieve the desired QoS specification stated in Section 2.1, *e.g.*, the 350 msec WCRT. Below we outline the challenges in verifying the systemic QoS properties of component-based DRE systems, which stem from the fact that verifying systemic properties does not boil down to simply aggregating verified properties for individual components.

• **Limitations of emulation-based verification:** In previous work [10], we used emulation and instrumentation to search the D&C solution space and find possible D&Cs that met the QoS requirements. Although emulation and instrumentation techniques yield realistic results, it is by no means any form of system verification since it is not based on any formal methods. Moreover, it is "hard" to cover

the entire solution space of a system under verification (*i.e.*, evaluate every possible state of the system) using emulation and instrumentation as it is with model-checking tools and simulation.

• **Limitations of isolated component verification:** In previous work [9], we showed how we can convert CBML models into TIOA configuration files via semantic anchoring [4] for verification purposes. In that work, however, we only permitted verification of components in isolation and assumed only one event is active in a component at any given time to contain the problem of formally describing a component. We, therefore, could not verify a deployed system because we did not take into account the target environment's behavior or the interaction between components. Verifying components in isolation and aggregating their verified properties does not illustrate the properties of the whole system since effects of communication and the hosting environment are not taken into account.

To facilitate complete system verification, we have to solve the challenges of (1) formally specifying the assembled components that represent the deployed system, and (2) relaxing the assumption that one event is active in a component (and system) at any given point in time. Moreover, developers who want to specify and verify QoS properties of their system, such as WCRT, should be (3) required to only model at the same level of abstraction as their development process, *i.e.*, at the application level. The remainder of this paper discusses our MDE-based approach that addresses these key challenges when verifying QoS properties of component-based DRE systems.

3 Model-driven Verification of QoS Properties for Component-based DRE Systems

This section describes our design of CUTS$_V$, a MDE tool for formally specifying component-based DRE systems to verify QoS properties.

3.1 Overview of the CUTS$_V$

The Component Behavior Modeling Language (CBML) is the foundation of CUTS$_V$ for verifying QoS properties of component-based DRE systems. CBML is a DSML implemented in the Generic Modeling Environment (GME) [13] and based on the mathematical formalisms of Timed Input/Output Automata (TIOA). System developers who use CBML do not need prior knowledge of either TIOA semantics or syntax. Moreover, system developers only focus on modeling behavior at the application level, which is within their knowledge domain (*i.e.*, Challenge 3 in Section 2.2).

Formally, we define a behavior model $BM = (V, S, \Theta, I, O, A, T, E)$ of a component in CBML as:

- A set V of *internal variables*.
- A set $S \subseteq val(V)$ of *states*.

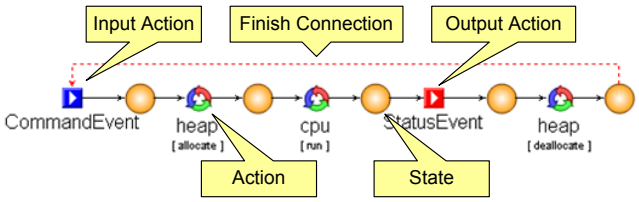

Figure 2. CBML model for the Effector component type in the SLICE scenario.

- A nonempty set $\Theta \subseteq S$ of *start states*.
- A set I of *input actions*, which are events received from an external source, *e.g.*, a connected component.
- A set O of *output actions*, which are events sent to an external destination, *e.g.*, a connected component.
- A set A of *actions*, which are events (or actions) visible only to the component hosting the behavior, *i.e.*, internal operations.
- A set T of *transitions* such that given $\Delta \in T$ and $s \in S$:

$$\Delta(s) \rightarrow \alpha, \tag{1}$$

  where $\alpha \in (A \cup O)$.
- A set E of *effects* such that given $\Gamma \in E$ and $a \in (I \cup O \cup A)$:

$$\Gamma(a) \rightarrow s, \tag{2}$$

  where $s \in S$.

Figure 2 highlights an example CBML model for the `Effector` component type in the SLICE scenario. As illustrated in Figure 2, CBML models are a sequence of action and state elements—similar to a process model [16]. Each series begins with an input action and terminates with a finish connection from the final state to the initial input action. CBML models alone, however, cannot verify systemic QoS properties of the realized system since it does not contain any information about either the system's composition or its D&C.

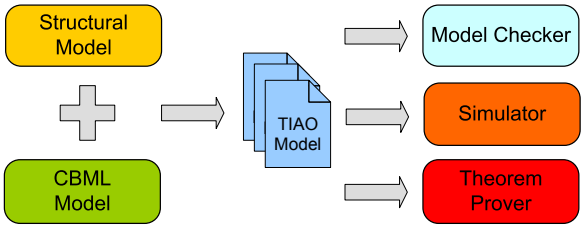

Figure 3. Conceptual workflow of CUTS$_V$.

As illustrated in Figure 3, CBML models are integrated with structural models to define the deployed system's behavior and composition. Such models include the behavior of the individual components, their intercommunication channels for passing messages (or events) between components, and their D&C within the realized system. The CBML and structural models are then transformed automatically into TIOA models via model interpreters thereby simplifying the formal specification of the system.

We selected TIOA as the target formal language because (1) CBML is based on the mathematical formalisms of TIOA, (2) input and output actions in TIOA are considered first-class entities similar to components, and (3) TIOA is better suited for component-based systems since components are reactive and inputs can occur at any given point in time (*i.e.*, input actions are always enabled). Lastly, the auto-generated TIOA files are read by model checkers, theorem provers, or simulators to verify QoS properties. The remainder of this section discusses workings on transforming CBML and structural models into TIOA models to facilitate verification via model checking using the Tempo Toolkit (`www.veromodo.com`) and UPPAAL (`www.uppaal.com`) model checker.

3.2 Modeling Multi-event Component Behavior

Component-based systems are characterized as reactive systems where any number of events can be active within a component, or the entire system, at any given point in time. The number of events active in a component, or the system, however, depends on the configuration of the underlying component technology. For example, the event handler for a component's input port may be configured to handle at most N active events simultaneously, which equates to at most N active threads. Likewise, a component has $|I|$ input ports and thus each input port can be configured to handle N_i events simultaneously, where $i \in I$ and N_i is the max number of events active on port i. To simplify our solution, however, we make the assumption that at most one event is active per port in a component (*i.e*, $N_i = 1$). Each individual component, therefore, can have at most $|I|$ active events at any given point in time (*i.e.*, Challenge 2 in Section 2.2).

Listing 1 highlights a snippet of the TIOA specification for `CommandEvent` port of the `Effector` component type as illustrated in Figure 2. As showcased in Listing 1, each component has $|I|$ number of `recv_event(i)` input actions where $i \in I$ and allows selection of the appropriate event queue (`evq`) for the incoming event (line 16). Likewise, each component has one `send_event(o)` where $o \in O$ selects the output port for the new event (line 26).

```
1  % legend:
2  % CE = CommandEvent; SE = StatusEvent
3  automaton Effector(myid,ch_CE,ch_SE:Int)
4    signature
5      input  recv_event (i : Int)
6      output send_event (o : Int)
7      internal handle_CE
8      ...
9      internal i_heap_deallocate
10   states
11     time : Real;
12     mode : [Port,Location];
13     thr_state : Array[Port,Thr_State];
14     evq : Array[Port,Int];
15   transitions
16     input recv_event (i)
17       eff if i = ch_CE
18           then evq[CE] := succ (evq[CE]); fi;
19     internal handle_CE
20       pre thr_state[CE] = ready ∧
21           mode[CE] = nil ∧ evq[CE] > 0;
22       eff evq[CE] := pred (evq[CE]);
23           thr_state[CE] := running;
24           mode[CE] := state_1;
25     ...
26     output send_event (o)
27       pre mode[CE] = state_3 ∧ o = ch_SE;
28       eff mode[CE] := state_4;
29     internal i_heap_deallocate
30       pre mode[CE] = state_4;
31       eff time := time + 0; mode[CE] := nil;
32           thr_state[CE] := complete;
33   trajectories
34     trajdef default
35       evolve d (time) = 1;
```

Table 1. Select Keywords in TIOA

Keyword	Description
automaton	Beginning of a TIOA definition
eff	Effect of a transition
evolve	Continuous variable evolution over time
input	Input transition from external automaton
internal	Transition visible only to self
output	Output transition to external automaton
pre	Precondition (guard) of a transition
trajdef	Define a new trajectory
trajectories	Trajectory section of automaton
transitions	Implementation section for transitions
signature	Definition section for transitions
states	Variable section of automaton

Listing 1. Code snippet of Effector component's TIOA model.

To support the assumption that each port has only one event active at a time, each port is assigned at most one thread (line 13). When the current thread on the corresponding port is ready (*i.e.*, not handling another event) and there are events waiting on the queue, the thread is allowed to process the next event (line 19). The thread then executes the sequence of internal actions as specified by the original CBML model. When the last action in the sequence is executed (line 29), the thread repeats the process if any events are remaining in the port's event queue.

The TIOA specification in Listing 1 is automatically generated from a CBML model constructed by system developers of a single port in one component. In general, system

developers construct high level CBML models of all their component's behavior and CUTS$_V$ automatically generates a formal specification similar to Listing 1. System developers, therefore, remain within their knowledge domain and do not have to focus as much on formally specifying the behavior of each component type in the system (*i.e.*, Challenge 3 in Section 2.2). Likewise, the TIOA specification supports individual components to handling multiple events simultaneously, which implicitly scales to the system handling multiple events simultaneously (*i.e.*, Challenge 2 in Section 2.2).

3.3 Modeling the Hosting Environment

The hosting environment for component-based systems comprises both hardware artifacts (*e.g.*, servers and communication links) and software artifacts (*e.g.*, component containers[1]). In the case of software artifacts, and more so component containers, system developers can create D&Cs such that all components—or a disjoint subset of components—deployed on the same host are executed in the same container, or within their own container. Each of these resulting configurations defines a different processing model, which impacts the system's overall behavior and QoS properties, such as end-to-end WCRT.

To simplify our solution, however, we make the assumption that all components deployed on the same host are executed within the same process. Moreover, instead of requiring developers to formally specify the host's behavior, we provide it for them so they remain within their knowledge domain (*i.e.*, Challenge 3 in Section 2.2). System developers, therefore, only have to select which host's behavior to use in the deployed system (see Section 3.4).

```
1  automaton Host (hid,thr_count : Int)
2    signature
3      input thr_request (host,cid,chid : Int)
4      input thr_release (host : Int)
5      output thr_assign (cid,chid : Int)
6      internal i_block, i_unblock
7    states
8      mode : Location := waiting;
9      time : Real;
10     thr_inuse : Int := 0;
11     target_cid, target_chid : Int;
12   transitions
13     input thr_request (host,cid,chid)
14       eff if host = hid ∧ mode = waiting ∧
15           thr_inuse < thr_count
16           then mode := dispatching;
17                target_cid := cid;
18                target_chid := chid; fi;
19     input thr_release (host)
20       eff if host = hid ∧ thr_inuse > 0
21           then thr_inuse := pred(thr_inuse); fi;
22     output thr_assign (cid,chid)
23       pre cid = target_cid ∧ chid = target_chid ∧
24           mode = dispatching ∧ thr_inuse < thr_count;
25       eff thr_inuse := succ(thr_inuse);
26           mode := waiting;
27     internal i_unblock
28       pre mode = blocked ∧ thr_inuse < thr_count;
29       eff mode := waiting;
30     internal i_block
31       pre mode = waiting ∧ thr_inuse = thr_count;
32       eff mode := blocked;
33   trajectories
34     trajdef traj
35       evolve d(time) = 1;
```

Listing 2. Code snippet of TIOA model for a component's hosting environment.

Listing 2 contains an example TIOA specification of a host provided by CUTS$_V$ to illustrate the effort that would have otherwise been required by system developers to define the behavior of a simple host (or component container). As illustrated in Listing 2, each `Host` is configured to have a unique id for identification purposes and specify the number of threads available for use by hosted components (line 1). Components request threads via the `thr_reqeust` input action (line 13). If threads are available, then the requesting port of the requesting component is assigned a thread to process its event (line 22). If there are no threads available, then a thread is not assigned until one is available via the `thr_release` input action (line 19).

```
1  % legend:
2  % CE = CommandEvent, SE = StatusEvent
3  automaton Effector (myid,host,ch_CE,ch_SE : Int)
4    signature
5      input  thr_assign (cid,chid : Int)
6      output thr_request (host,cid,chid : Int)
7      output thr_release (host : Int)
8    transitions
9      output thr_release (hid)
10       pre hid = host ∧ thr_state[CE] = complete;
11       eff thr_state[CE] := nil;
12     output thr_request (hid,cid,chid)
13       pre hid = host ∧ cid = myid ∧
14           chid = ch_CE ∧ evq[CE] > 0 ∧ mode = nil;
15     input thr_assign (cid,chid)
16       eff if cid = myid ∧ chid = ch_CE
17           then thr_state[CE] := ready; fi;
```

Listing 3. Code snippet of the Effector component's TIOA model with support for hosting environment interaction.

Likewise, Listing 3 contains a snippet of the TIOA specification, which is an extension of the `Effector` component from Listing 1 in Section 3.2, that system developers would have to write to support components requesting threads from their host to process events. As highlighted in Listing 3, when there are events in the corresponding port's queue and no events are being processed, the component can request a thread from its host (line 12). If there are threads available, then the requesting port of the requesting component is assigned a thread (line 15). After the component's port is finished processing the event, it releases the thread so the host can assign it to another port of the same, or a different component (line 9).

[1] A component container is an isolated environment that can be configured to meet the execution requirements of the hosted component(s).

Because CUTS$_V$ provides TIOA models of the hosting environments, system developers do not have to worry about formally specifying low-level details of the system's behavior. Likewise, since the formal specification of a component's behavior is autonomously generated by CUTS$_V$, this simplifies formally specifying how components interacting with their target environment. System developers, therefore, are still able to remain within knowledge domain when using CUTS$_V$ by focusing mainly on modeling system behavior and composition (*i.e.*, Challenge 3 in Section 2.2).

3.4 Modeling System Composition

To model system composition of a component-based system, it is necessary to construct formal models that reflect instances of component types (see Section 3.2) being deployed on hosts in the target environment (see Section 3.3), and intercommunication channels that components use to communicate with each other. Since system developers use CBML to model the behavior of individual components and the hosting structural language, such as PICML, to model the system's composition, we have enough information to formally specify the deployed component-based system autonomously (*i.e.*, Challenge 1 in Section 2.2)—thereby alleviating the complexity of formally specifying the realized system.

Before we can formally specify a deployed (or wired) system, we must first describe the behavior of communication channels (or connections) between components. As illustrated in Figure 1 of Section 2, component instances communicate with each other via connections. These connections can be configured such that there may or may not be a limit on the number of events allowed on an connection before the underlying component middleware begins to drop events. Likewise, to verify systemic QoS properties, it may be feasible to allow system developers to specify network properties, such as delayed delivery based on event size. To simplify our solution, however, we currently assume that the connection between events is unbounded and there are no network properties associated with connections. Although we make these assumptions, our approach does allow system models to incorporate network properties if we (*i.e.*, CUTS$_V$) provide the necessary TIOA connection models that allow developers to set different network configuration parameters, such as network delay.

Listing 4 highlights the TIOA model for an unbounded event connection with no network properties, which is provided by CUTS$_V$. As illustrated in Listing 4, each connection is given a unique id so components can select which communication channel to use for transmitting an event. Connections receive events via the `send_event(chid)` input action (line 11), which corresponds to the `send_event(o)` output action of a component (see line 26 in Listing 1) such that $chid = o$.

```
1  automaton Connection (ecid : Int)
2    signature
3      input  send_event (chid : Int)
4      output recv_event (chid : Int)
5      internal i_set_empty, i_set_nempty
6    states
7      mode : Location := empty;
8      time : Real;
9      count : Int := 0;
10   transitions
11     input send_event (chid)
12       eff if chid = ecid
13           then count := succ (count); fi;
14     output recv_event (chid)
15       pre mode = not_empty ∧
16           count > 0 ∧ chid = ecid;
17       eff count := pred (count);
18     internal i_set_empty
19       pre mode = not_empty ∧ count = 0;
20       eff mode := empty;
21     internal i_set_nempty
22       pre mode = empty ∧ count > 0;
23       eff mode := not_empty;
24   trajectories
25     trajdef traj
26       evolve d(time) = 1;
```

Listing 4. Code snippet of a TIOA model for an unbounded event connection between two or more components.

Likewise, connections send events to the correct component port via the `recv_event(chid)` output action (line 14), which corresponds to the `recv_event(i)` input action of a component (see line 16 in Listing 1) such that $chid = i$. This allows more than one component to receive an output event from a single source component (*i.e.*, simulate publishing an event to multiple destinations), or multiple components to publish events to the same input port of a component. Finally, since this is an unbounded connection, the `recv_event(chid)` output action can fire as long as there is an event on the connection, *i.e.*, $count > 0$.

```
1  automaton SLICE (hid_SenMain, hid_SenSec,
2   hid_PlanOne, hid_PlanTwo, hid_Config,
3   hid_EffMain, hid_EffSec : Int)
4    components
5      c0 : Connection (0);
6      c1 : Connection (1);
7      c2 : Connection (2);
8      c3 : Connection (3);
9      c4 : Connection (4);
10     c5 : Connection (5);
11     SenMain : Sensor (1,hid_SenMain,1);
12     SenSec : Sensor (2,hid_SenSecondary,1);
13     PlanOne : Planner (3,hid_PlanOne,1,2);
14     PlanTwo : Planner (4,hid_PlanTwo,2,3);
15     Config : ConfigOp (5,hid_Config,3,4,5);
16     EffMain : Effector (6,hid_EffMain,4,0);
17     EffSec : Effector (6,hid_EffSec,5,0);
```

Listing 5. TIOA model of the SLICE scenario assembly.

Listing 5 shows the TIOA model for the SLICE scenario assembly, which is autonomously generated by CUTS$_V$.

To formally specify the assembled system, we leverage the composition features of TIOA. As illustrated in Listing 5, the assembled system is composed of artifacts autonomously generated from CBML models (*e.g.*, the component behavior model) or provided by CUTS$_V$, (*e.g.*, the `Connection` model). Each component assembly is derived from the structural model of the system created by system developers and parameterized by the host id of each component instance (line 1). This allows CUTS$_V$ to simplify the specification of different D&Cs, or systems based on partial assemblies, by reusing the appropriate TIOA composition model. The assembly also contains instances of connections for each separate communication channel in the deployed assembly (line 5– 10). Finally, each component instance is declared in the composition with a unique id, assigned to a host via its corresponding host parameter, and associated with the appropriate communication channels (line 11– 17).

To formally specify the D&C of an assembled system, such as SLICE scenario exemplified in Figure 1 and Listing 5, for verification, CUTS$_V$ provides a D&C composition of the system. The deployment composition is also autonomously derived from the structural model created by system developers. Listing 6 shows a example deployment of the SLICE scenario, which is ready to undergo system verification.

```
1  automaton ExampleDeployment
2    components
3      h1 : Host (1);
4      h2 : Host (2);
5      h3 : Host (3);
6      system : SLICE (1,2,3,1,3,2,3);
```

Listing 6. TIOA model of an example deployment of the SLICE scenario.

As illustrated in Listing 6, each deployment has a collection of `Host` instances—each having a unique id for identification purposes—for deploying components (line 3– 5). The main assembly is also declared and parameterized with the appropriate host id (line 6) for each component instance (see Listing 5). This will force each component to request threads of execution from the correct host when attempting to process events. Once the TIOA model for the given D&C is autonomously generated by CUTS$_V$, system developers use the Tempo Toolkit to convert the TIOA models into Timed Automata [1] models (see Figure 3 in Section 3.1). The Timed Automata models can then be imported into UPPAAL to verify systemic properties.

4 Related Work

Gössler et al [6] present a framework for modeling and verifying properties of component-based systems. Their approach separates component behavior from component interaction, and makes component artifacts, such as connectors, first-class entities in their formal language. Our approach is similar because we separate component behavior from component interaction, but we do so at the DSML level. This simplifies the formal specification of component-based systems for system developers because they operate within their knowledge domain as opposed to manually writing formal specifications—the equivalent of manually writing TIOA specifications. We also make component artifacts first-class entities, but we do so at the a higher level of abstraction, *i.e.*, the DSML level, and implement the first-class entities as reusable artifacts in the target formal language, *i.e.*, TIOA. This prevents the creation of a new formal language, which must be theoretically proven valid.

Zschaler [23] presents initial workings on a framework for specifying and verifying non-functional requirements, such as WCET, for component-based systems. Our approach is similar to Zschaler in that we are not trying to create a new formal language to verify non-functional properties, but define a framework that leverages existing formalisms to define the component-based system interactions. The resultant is the ability to "plugin" reusable artifacts, such as connectors, containers and components, which implicitly map to existing theories that can be proven, or disproved. Our approach differs because we operate within the knowledge domain of component-based system developers (i.e., the application level) when formally specifying the system's behavior and composition, and leverage DSMLs to simplify specification. Finally, the verification of non-functional properties in both Zschaler's and our approach is still a work in progress.

5 Concluding Remarks

In this paper, we presented our preliminary ideas on a framework called CUTS$_V$ for automating the formal specification of component-based DRE systems to verify systemic QoS properties. We also showed how CUTS$_V$ is designed to work at the same level of abstraction as component-based system developers are used to, *i.e.*, at the application level. System developers use CBML and structural models to capture the behavior and composition of the system, respectively. Model interpreters then automatically transform the CBML and structural model into Timed Input/Ouput Automata (TIOA) configuration files for system verification. We believe CUTS$_V$ will help simplify verifying multiple QoS properties of component-based DRE systems because the system developers operate within their knowledge domain and the formal specification process is automated.

The following list summarizes the future research directions of CUTS$_V$ for verifying systematic QoS properties of component-based DRE systems:

- Ideally, system developers should be required to specify their system at a high level of abstraction, including the QoS requirements, and let the tool autonomously search the solution space to locate and verify candidate D&Cs that meet QoS requirements. Future work, therefore, includes fully automating the verification process to reach the ideal tooling environment.
- The verification process is not complete without validation, *i.e.*, ensuring candidate D&Cs meet their QoS requirements in the target environment. Future work, therefore, includes using the emulation and instrumentation features of CUTS to validate if candidate D&Cs located during the verification process also meet their QoS requirements in the target domain.
- As systems grow larger and more complex, the D&C solution space inadvertently grows larger. Using TIOA to formal specify such a system may not be a feasible solution. Future work, therefore, includes understanding how our solution approach scales to larger and more complex systems, *e.g.*, ultra-large-scale systems [11].

CUTS_V is available in open-source format for download at `www.dre.vanderbilt.edu/CUTS`.

References

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Los Alamitos, CA, USA, 2005.

[3] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-interaction Automata as a Verification-oriented Component-based System Specification. In *Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems (SAVCBS '05)*, 2005.

[4] K. Chen, J. Sztipanovits, S. Abdelwahed, and E. K. Jackson. Semantic anchoring with model transformations. In *ECMDA-FA*, pages 115–129, 2005.

[5] L. de Alfaro and T. A. Henzinger. Interface Automata. *SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[6] G. Gössler, S. Graf, M. Majster-Cederbaum, M. Martens, and J. Sifakis. An Approach to Modeling and Verification of Component Based Systems. In *Proceedings of Current Trends in Theory and Practice of Computer Science (SOFSEM '07)*, Harrachov, Czech Republic, 2007.

[7] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[8] A. Guerrouat and H. Richter. A Component-based Specification Approach for Embedded Systems using FDTs. *SIGSOFT Softw. Eng. Notes*, 31(2), 2006.

[9] J. H. Hill and A. Gokhale. Model-driven Engineering for Early QoS Validation of Component-based Software Systems. *Journal of Software*, 2, 2007 (to appear).

[10] J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[11] S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jun 2006.

[12] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata, Synthesis Lectures in Computer Science*. Morgan and Claypool Publishers, San Rafael, CA, Apr. 2006.

[13] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[14] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[15] Object Management Group. *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 edition, June 2004.

[16] Object Management Group. BPMN Information Home. www.bpmn.org, 2005.

[17] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[18] Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/gray/Research/C-SAW, University of Alabama at Birmingham, Birmingham, AL.

[19] V. Subramonian. *Timed Automata Models for Principled Composition of Middleware*. PhD thesis, Washington University in St. Louis, Computer Science and Engineering Department Technical Report WUCSE-2006-23, May 2006.

[20] M. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. PhD thesis, Leiden University, Leiden, Netherlands, 2003.

[21] J. White, K. Czarnecki, D. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege. Automated Model-based Configuration of Enterprise Java Applications. In *Proceedings of the 11th IEEE International EDOC Conference (EDOC 2007)*, Annapolis, MD, Oct. 2007.

[22] J. White and D. C. Schmidt. Automated Configuration of Component-based Distributed Real-time and Embedded Systems from Feature Models. In *Proceedings of the 17th Annual Conference of the International Federation of Automatic Control*, Seoul, Korea, July 6-11 2008.

[23] S. Zschaler. Towards a Semantic Framework for Non-functional Specifications of Component-Based Systems. pages 92–99, Rennes, France, Sept 2004.