The 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE'11). June 21-23, 2011. Boulder, Colorado, USA. Copyright ACM

Towards Verified Synthesis of ProCom, a Component Model for Real-Time Embedded Systems

Etienne Borde
Institut TELECOM, TELECOM ParisTech, LTCI
Paris, F-75634 CEDEX 13, France
etienne.borde@telecom-paristech.fr

Jan Carlson
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
jan.carlson@mdh.se

ABSTRACT

To take advantage of component-based software engineering, software designers need a component framework that automates the assemblage and integration of developed components. It is then of prime importance to ensure that the synthesized code respects the definition of the component model's semantics. This is all the more difficult in the domain of embedded systems since the considered semantics usually aims at characterizing both functional properties (*e.g.* data and control dependencies) and non-functional properties such as timing and memory consumption.

The component model considered in this paper, called ProCom, relies on an asynchronous operational semantics and a formal hypothesis of atomic and instantaneous interactions between components. The asynchronous approach targets higher flexibility in the deployment and analysis process, while the formal hypothesis helps in reducing the combinatory problems of formal verification.

In this paper, we present a code generation strategy to synthesize ProCom components, and a formalization of this generated code. This formalization extends the verification possibilities of ProCom architectures, and constitutes a step toward the verification that the produced code respects the operational semantics of ProCom.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*validation*; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Theory, Verification

Keywords

Component Based Model, Synthesis, Verification, Real-Time, Embedded Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.
Copyright 2011 ACM 978-1-4503-0723-9/11/06 ...$10.00.

1. INTRODUCTION

Embedded systems developers are facing the challenging task to provide fancier and fancier functionalities while reducing the time-to-market of the production. When embedded in, for example, avionic or automotive systems, another important challenge is to be able to assess (for obvious safety reasons), the correction of the developed software.

In order to accelerate the design process, component-based software engineering (CBSE) advocates the reuse of already developed functionalities [6]. CBSE also helps in managing software complexity by dividing its design into modules with different granularity levels. However, CBSE for embedded systems still raises an important set of challenges such as predictability [7].

To tackle this issue, formal methods and more particularly model checking, help in strengthen the verification that a developed application respects predefined safety properties. However, modular model checking is a difficult problem [21] and the verification of a complete system suffers from serious scalability problems.

In answer, synchronous languages [4, 8] rely on a formal semantics, *i.e.* a set of mathematical hypothesis that eases the verification of the system; synchronous languages assume that all the actions undertaken by the application are instantaneous. This approach and the associated tools (verification and code generation) have been used industrially in the domain of avionic systems.

More generally, synchronous approaches constrain the design process to limit strictly the software architecture to well-known and analyzable patterns. The price of this strategy is a degradation of the flexibility in the deployment process. For instance, synchronous languages made possible the verification of a complete system, to the price of the modularity of the generated code [13].

By considering in this paper a component model relying on an asynchronous semantics we give more flexibility (in terms of design process) to a software architect. At the same time, we consider a component model that provides a formal semantics in order to help in reducing the state space explosion problems raised by model-checking. The problem we address can be formulated as follows: "how to synthesize the assembly code of such a component model in order to respect both the asynchronous and formal aspects of its semantics?"

As an answer to this problem, we present in this paper the code generation dedicated to *ProCom*, a component model targeting control-intensive embedded systems (*e.g.* automotive systems) [17]. This component model relaxes the synchronous languages hypothesis by specifying that only local interactions between components are instantaneous. Thus, this component model relies on an asynchronous but formal semantics [20] that helps in preserving the modularity of a component-based approach.

As for synchronous languages, generating code for a formally defined component model like ProCom requires to ensure that the generated code respects the formal hypothesis. Note that although the formal semantics of ProCom defines instantaneous interactions, no physical implementation can of course fully satisfy this. However, it should be possible to ensure that "semantically relevant" characteristics of instantaneous interactions are preserved. An important issue is to guarantee that the interaction patterns (emission/reception of trigger and data) is the same as the one that would have been produced in case interactions were instantaneous. This is a difficult problem because of interferences that might occur *during* interactions that are supposed to be instantaneous, thus modifying the interactions order that was expected from the semantics. As we will show in this paper, using simple locking mechanisms is not enough to ensure the preservation of interactions order. Regarding this problem, we present in this paper two contributions:

- the code generation strategy we adopted to produce the glue code wrapping ProCom components, which aims at ensuring that both the asynchronous and the formal aspect of the semantics are preserved *by construction*; a code generation tool has been integrated in the ProCom editor (PRIDE[1]) to automate the production of the components glue code.

- a formalization of this generated code that extends the verification of ProCom architectures and constitutes a step towards the verification of the consistency between the formal semantics of ProCom component model and its implementation.

To show that the proposed code generation patterns respect the semantics of ProCom components, we focus on a meaningful characteristic which is the atomic data broadcast: if one output data port writes to several input data ports, then all input data ports receive the same value (the last data written by the writter). We provide a formal verification of this property and show that ensuring such property in the generated code is not trivial.

The paper is organised as follows: we present in section 2 the ProCom component model and its operational semantics. In section 3, we propose a code generation strategy that aims at implementing the formal hypothesis of the ProCom component model. To illustrate this contribution, we present selected parts of the generated code (in section 4), obtained on an example illustrating an important characteristic of the formalization of ProCom: the atomic data broadcast. We then propose a formal model of this generated code, and verify on this model that the atomic broadcast characteristic is ensured (section 5). In sections 6 and 7, we respectively present related works and conclude the paper.

2. PROCOM – THE COMPONENT MODEL

The ProCom component model [17] is specifically developed to address the particularities of the embedded systems

[1]http://www.idt.mdh.se/pride/?id=home

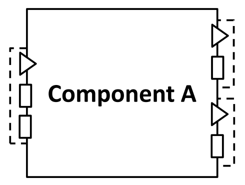


Figure 1: Example of a ProCom component.

domain, including resource limitations and requirements on safety and timeliness. Key features include support for various analysis tecniques related to timing [12], resource usage [15, 19] and dependability; and reuse of design artifacts (e.g., extra-functional properties, analysis results, and behavioral models) [16] as well as code reuse. To address resource limitations, the ProCom approach view components as design-time entities rather than concrete parts of the final system. An application is built as a collection of interconnected components, and in the later stages of development this component-based design is transformed into executable units, such as tasks that can be handled by traditional real-time operating systems.

2.1 General Overview

ProCom is organized in two distinct – but related – layers, addressing the different concerns on different levels of granularity. The layers differ in terms of architectural style and communication paradigm. In the the top layer, a system is modeled as a collection of active, interconnected *subsystems* that execute concurrently and communicate by asynchronous message passing. The lower layer, which is the focus of this paper, consists of smaller, passive *components*. It is based on a pipes-and-filters architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components. Data ports always appear in a group together with a single trigger port, and the ports in the same group are read or written together in a single atomic action. Together, an input group and its associated output groups are called a *service*.

Figure 1 shows a simple ProCom component with one input port group and two output port groups (denoted by dashed lines). Triangles and boxes denote trigger- and data ports, respectively.

Components interact through connections from output- to input ports. In addition, *connectors* provide detailed control over the data- and control flow. These include constructs for forking and joining data and trigger paths, and for selecting at runtime an execution path.

Both layers are hierarchical, meaning that subsystems as well as components can be nested. The way in which the two layers are linked together is that a primitive top layer subsystem can be further decomposed into components at the lower layer. At the bottom of the hierarchical nesting, the behavior of a primitive ProCom component is implemented as a C function.

2.2 Operational Semantics

Next, we give an overview of the operational semantics of ProCom, focusing on aspects that are particularly relevant for the code generation presented in this paper, such as the behaviour of individual components. The semantics has been formalized in a high-level finite state machine notation [20]. This notation can, in turn, be translated into timed automata [1] which provides the formal foundation.

One particular characteristic of ProCom, compared to most general purpose component models, is the restricted component behaviour. Components can not freely communicate through input and output ports, but must follow a read-execute-write cycle, where the activation is always initiated externally and the dependencies between input and output are explicitly modeled in the component interface. The motivation for these – and other – restrictions, is that they permit various types of detailed analysis on the component-model level, i.e., based on component interconnections and extra-functional properties associated with the individual components but without knowledge of component implementations.

Contrasting synchronous languages, ProCom components generally execute asynchronously, which allows for more flexibility, *e.g.*, when designing multi-rate applications. However, the interaction (data and triggering) between components is governed by assumptions of atomicity and instantaneousness. Together with the restricted component semantics, this also facilitates analysis by reducing the number of semantically distinguishable interleaving possibilities that must be considered.

Next, we describe the operational semantics of a ProCom component. Each service is initially in the idle state, just receiveing data on its input data ports. When a service is triggered, *i.e.*, when the input trigger port is activated, it switches from idle to active state in which it ignores any further incoming triggers. The active phase consists of the following four steps:

- *Step 1:* The data at the input data ports of the service are atomically copied to an internal representation which remains unchanged until the end of the service execution.
- *Step 2:* The service functionality is executed. For primitive components, the functionality is given by a C function, while the internal computation of a composite component is defined by the subcomponents and their interconnections.
- *Step 3:* The computation can update the output data ports of the service, but the values are only made visible externally when one of the output port groups is triggered (from inside the service). When this happens, the values at the data ports of that group are forwarded along the outgoing connections, and then the triggering is propagated from the trigger output port. The transfer of data and triggering from a group is performed as an atomic action, and thus may not be interleaved by other concurrent activities.
- *Step 4:* When all output port groups have been triggered once, the service immediately returns to the idle state. Multiple triggering of one output group during a single activation is not permitted.

In addition to the operational behaviour of individual components, the semantics defines the behaviour of connectors, communication between subsystems in the top layer, the semantical link between the two layers, etc. Those aspects, however, are outside the scope of this paper.

As discussed previously, one aspect of the component semantics that is of particular importance for the synthesis is the atomic and instantanous transfer of data from ports in the same port group, both when input data are copied into the component (in *Step 1*) and when output is forwarded to other components (in *Step 3*). Ensuring this atomicity is a key challenge of the synthesis, and the proposed way to address this is elaborated in section 3.

3. SYNTHESIS PRINCIPLES

Synthesis of a component model consists of generating glue code that performs interactions of components with other components and with their execution environment. An important requirement of a component synthesis process is to enforce the respect of the component model semantics.

In the synthesis solution we present in this paper, we take advantage of the asynchronous hypothesis to implement a ProCom system as a set of concurrent tasks, dispatched by the reception of periodic or aperiodic events. In the remainder of this paper, we assume that

- tasks are executed by a real-time operating system with a preemptive scheduler that ensures tasks are dispatched FIFO[2] when dispatched with the same priority; and
- the components-to-tasks allocation has been defined.

In the considered component model, data flows can be joined; meaning that several components can write data to the same recipient. The recipient reads the last produced data when it is activated. This modeling pattern might induce concurrent data accesses when the producers are dispatched in different tasks.

As stated in the introduction of this paper (section 1), the synthesis process we propose must guarantee that the order of interactions is consistent with the interaction semantics defined by the ProCom component model. As a consequence, we need to use data access protocols that guarantee the integrity of data while preserving interactions order. For the same reason, the generated code must enforce that activation order of components is preserved; not only when activation is transferred by another component but also when activation comes from an event source (clock timeout, or message reception).

In order to enforce the respect of these different characteristics, we propose in this section:

- a method to identify sets of port groups that can interact concurrently and thus risk to violate the ordering requirement;
- a technical solution that preserves the interaction order among such ports whether they belong to event sources (clocks, messages) or components.

In the following subsections, we explain how interactions among ProCom components are configured in order to preserve their formal semantics in multi-task applications.

3.1 Ensuring Port Group Atomicity

As described in section 2, all data of a port group should be transferred in a single atomic operation. In most cases,

[2]First-In/First-Out

though, interleaving the transfer of data belonging to different port groups has no semantically relevant impact, so enforcing full atomicity is unnecessarily restrictive.

Thus, to combine consistency and efficiency, it is important to identify port groups for which a concurrent execution might violate the characteristics of the component semantics. Indeed, the implementation of interactions involving such port groups needs to be protected by a locking mechanism that ensures the order of component interactions in a multi-task application. Before discussing further the principle of such a mechanism, we present how to compute the sets of port groups that are vulnerable to concurrent executions.

To begin with, we assume (for the sake of clarity of the presentation) that the considered ProCom model is flattened: port groups of composite components are transformed into simple primitive components operating as interactions proxies.

To compute the sets of port groups that are vulnerable to concurrent executions, we focus on the concurrency among output port groups. Since output trigger ports are the initiators of components interactions, and since data production (respectively consumption) is done when the corresponding output (resp. input) trigger port is activated, we propose to implement each interaction from one output trigger port towards one (or several) input trigger port(s) within one critical section. We will show in section 5 that this mechanism, under certain conditions described in the remainder of the paper, ensures that data are atomically broadcasted: if one output data port writes to several input data ports, then all input data ports receive the same value. This property might seem trivial first, but verifying formally that the implementation patterns we propose ensure its validity required to go into the details we present in the remainder of this section.

Next, we present formally how to compute the scope of the critical sections, i.e., the set of output port groups that need to lock the same resource. To compute this, we first define pairwise potentially conflicting output port groups, and then group them into sets.

Definition 1 (Potentially conflicting groups). *For two output port groups o_1 and o_2, we define the binary relation $PC(o_1, o_2)$ to hold if there exists two ports, in o_1 and o_2 respectively, that are connected to ports (trigger or data) in the same input port group.*

Definition 2 (Potentially conflicting sets). *Let OG denote the set of all output groups and PC^* the transitive closure of PC. The potentially conflicting sets (PCS) is defined as follows:*

$$\begin{aligned} PCS = \{S \mid\ & S \subseteq OG\ \wedge \\ & \forall_{o_1 \in S}\ \forall_{o_2 \in S}\ PC^*(o_1, o_2)\ \wedge \\ & \forall_{o_1 \in S}\ \forall_{o_2 \in OG-S}\ \neg PC(o_1, o_2)\ \} \end{aligned}$$

Finally, the actually conflicting port group sets are computed by checking if the potentially conflicting port groups can be accessed concurrently (*i.e.*, by several tasks). The following equation gives the formal definition of the conflicting sets:

Definition 3 (Conflicting sets). *Let $T(o)$ denote the set of tasks that can execute the component that the output group o belongs to. The set of conflicting sets CS is defined as follows:*

$$CS = \{S \mid S \in PCS\ \wedge\ \left|\bigcup_{o \in S} T(o)\right| > 1\}$$

For each set in CS, a lock is initialized and the corresponding interactions between ProCom components are implemented within critical sections using this lock. This is described further in section 3.3.

3.2 Example

Figure 2 provides an example of connected port groups, and their association to tasks (T_1 and T_2 in this example). In this figure, $o_1 \dots o_6$ represent output port groups while $i_1 \dots i_4$ represent input port groups. Edges linking port groups represent connections between some ports in them.

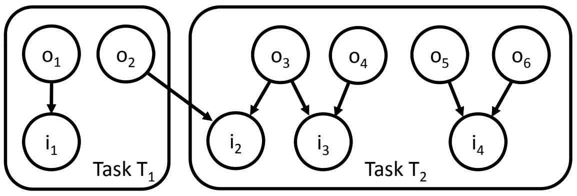


Figure 2: Example: Group dependencies and task allocation

In this example,

- PC is the reflexive and symmetric closure of the set $\{\langle o_2, o_3\rangle, \langle o_3, o_4\rangle, \langle o_5, o_6\rangle\}$.
- $PCS = \{\{o_1\},\ \{o_2, o_3, o_4\},\ \{o_5, o_6\}\}$
- $CS = \{\{o_2, o_3, o_4\}\}$; the other sets in PCS are eliminated since they are executed in a single task.

Note here that although o_2 and o_4 are not interacting with the same input port, the processing of o_2 by T_1 will be executed in a critical section sharing the same lock as the processing of o_4 by T_2. This aims at preserving the order of the computed conflicting set $\{o_2, o_3, o_4\} \in CS$.

3.3 Preserving Data Integrity and Component Interactions Order

Thanks to the method presented above, we identify sets of output port groups that might interact concurrently. To ensure the atomicity property, we propose to instantiate a lock for each conflicting set of port groups, and to execute the whole interaction (transfer of trigger and data) inside a critical section protected with this lock. This technical solution ensures the atomicity of the interactions among components.

To preserve the order of interactions, we propose to use a FIFO locking mechanism. Note that a simple locking mechanism (like POSIX mutexes) does not serve blocked tasks in a FIFO manner: Consider three tasks T_0, T_1 and T_2 with decreasing priorities, sharing the same lock. Assume that T_2 is in the critical section when T_1 is dispatched. T_1 reaches the critical section, then T_0 is dispatched and reaches also the critical section. When T_2 releases the lock, T_0 enters the critical section because it has a higher priority than T_1 (although T_1 reached the critical section first).

To tackle this issue, we propose to use a specific locking policy that ensures that locking actions are served in a FIFO manner. To implement this, we rely on the immediate ceiling priority protocol (ICPP) in order to benefit from existing results related to this locking policy [18, 2]. When locking a resource configured with this protocol, the corresponding task inherits a priority higher than all the tasks that may access the resource, thus impeding other tasks sharing this resource from interfering and causing an interaction order different from FIFO.

Considering again the previous scenario with tasks T_0, T_1 and T_2. When T_2 enters the critical section, it becomes the highest priority tasks. When T_1 and then T_0 are dispatched, none of them is scheduled until T_2 releases the resource. Thus none of them can reach the critical section and thus the locking trivially satisfies the FIFO requirement.

ICPP is a very well known solution in real-time systems since it avoids deadlocks while minimizing the blocking time of tasks du to shared resources and interferences (it thus minimizes the number of context switches). It is implemented by POSIX-compliant operating systems and natively supported by the Ada language. From a schedulability analysis perspective, using ICPP eases the schedulability analysis (the blocking time of tasks due to shared resources is easy to compute from the WCET[3] of the critical sections).

In our solution, the task holding a lock is in fact assigned a priority one level higher than the highest priority of tasks that may request the lock. This is consistent with the ICPP description, and means that performing an interaction is only delayed by other interactions, not by internal computations executed by tasks that might enter the critical section.

The generated code implementing the control- and data transfer of an output port group, performs the following steps:

1. If the group belongs to a *conflict set*, request the lock associated with this set.
2. For each data port of the group, transfer the value to the connected (if any) input data ports.
3. Trigger the connected (if any) input trigger ports.
4. Release the lock (if locking was performed in step 1).

Note that triggering a component inside the critical section does not imply to execute its functionalities in this critical section. On the contrary, a clear separation between the components' interfaces and the components' internal implementations leads to execute only the connected interface in the critical section. This separation will be illustrated further in section 4.

3.4 Preserving Task Activation Order

In this section, we tackle the problem of guaranteeing that the order of triggering is preserved for releases of tasks corresponding to event sources (*e.g.* a clock timeout or the arrival of a message). When scheduling tasks, an operating system usually queues dispatched threads according to their priority. Thus the order in which releases of tasks are treated depends not only on their timing properties, but also on their priority. According to the considered semantics, the instant when a task is released should by itself determine the order in which the trigger of connected components should be performed.

In answer to this problem, we define two priority levels for each task: an *activation priority* and a an *functional priority*. In case the considered task triggers a port group belonging to a conflicting set, we define the *activation priority* to be one level higher than the highest priority of tasks that can execute one of the port group of the set (the same way we compute the ceiling priority for a conflicting set configured with ICPP). Then, we define the *functional priority* as the priority expected based on the information in the design. In case the considered task triggers a port group that does not belong to any conflicting set, then the *activation priority* equals the *functional priority.*

To sum up, this execution pattern consists of dispatching a task in a state that corresponds to being in a critical section configured with the ICPP protocol. This solution assumes that tasks with the same priority are dispatched in a FIFO manner. If this assumption is implemented by the underlying platform, using the proposed priority management pattern preserves the order of activation among port groups in a same conflict set, when those port groups are triggered first in a task.

From a schedulability analysis perspective, this priority management pattern can be analysed by considering each task that has an activation priority different from the functional priority as if it was replaced by two separate tasks with these two (fixed) priorities, released at the same time.

Note also that triggering a component at the *activation priority* does not imply to execute its functionalities at this priority, nor the interactions leading out of the component.

4. SYNTHESIS IN PRACTICE

To illustrate in practice the synthesis principles presented in the previous section, we rely on the example given in Figure 3, consisting of three components: Producer, Consumer1 and Consumer2. The input ports of these components are triggered by three independent external events, corresponding to three different tasks (T0, T1 and T2).

This example was chosen since it exhibits an important characteristic of atomic/instantaneous interactions: the atomicity of data broadcast. Indeed, in this example, the atomic broadcast characteristic can be verified by checking that when both Consumer1 and Consumer2 are triggered without interleaving of the activation of t_out, they have the same input value (the last value produced by Producer). This property might seem straightforward to ensure, but the formal verification result we present in section 5 show that it was necessary to implement the solution presented in section 3.

Using this example, we present in this section the glue code that connects and activates ProCom components. Note that this example presents one conflict set consisting of the three output port groups of Producer, T1 and T2.

4.1 Generation of Interfaces

The first step of the glue code generation consists of producing the components interfaces. For each port of a component, a specific operation is generated. Listings 1 and 2 present the generated code for input and output port groups, exemplified by Consumer1 and Producer, respectively.

Listing 1 presents the generated code for the input port group of Consumer1. The operation Consumer1_t_in1 implements the first interaction step presented in section 2.2: if the service is in idle state, the received data are transferred to the service's internal state (*cf.* Consumer1_transfer_d_in1 in listing 1) and the component's implementation is scheduled (return 1).

Listing 2 presents the generated code for the output port

[3]Worst-Case Execution Time

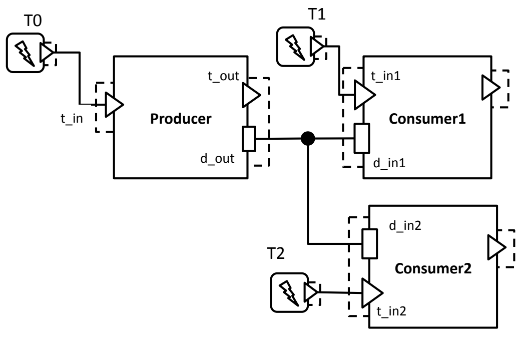


Figure 3: Example illustrating the atomic broadcast property

```
int Consumer1_t_in1 (Consumer1_svc * svc){
  // Ignore trigger if already active
  if(svc->active) return 0;
  else {
    Consumer1_transfer_d_in1(svc);
    return 1; // Schedule the component's execution
  }
}

void Consumer1_transfer_d_in1(Consumer1_svc * svc) {
  // Copy external->internal view (cnx_d_in1->d_in1)
  svc->d_in1=svc_h->cnx_d_in1;
}
```

Listing 1: Component interface implementation: Inputs

group of the component Producer. The operation corresponding to the output trigger port (Producer_t_out) returns the identifier of the considered port (STATE_Producer_t_out here). This value is used to schedule components connected to the output trigger port and transfer data out from the corresponding port group.

The operation transferring data (Producer_transfer_d_out) is also illustrated in listing 2. If the corresponding output data has been updated, the data transfer operation writes it to a given memory location (represented by the dest parameter). This data transfer operation will be called in the generated code of the task that executes the output port group (described in the next subsection). It is then called for each connected input data port; the destination (dest) being the "external view" (as opposed to the stable internal copy) of the connected input data port. In the current case, Producer_transfer_d_out will be called twice: first with dest=cnx_d_in1 and then with dest=cnx_d_in2.

4.2 Tasks Body Implementation

When generating the code of a task, the synthesis process uses the connections expressed in the design to produce the control and data flow of the task. In this subsection we present two important parts of the synthesized code for the internal implementation of a task:

1. the scheduling of subcomponents according to the connections expressed in the design;
2. the scope of critical sections that ensures the atomic data broadcast property.

```
STATE_Producer Producer_t_out (Producer_svc * svc) {
  return STATE_Producer_t_out;
}

void Producer_transfer_d_out(Producer_svc * svc,
                             int ** dest) {
  // Check if data was updated
  if (svc->d_out_updated) {
    dest = &(svc_h->d_out);
    svc->d_out_updated = 0; // Reset update flag
  }
}
```

Listing 2: Component interface implementation: Outputs

```
1   void T0_schedule(Task * tsk) {
2     if(Producer_trigger_in(tsk->Producer))
3       T0_todo_list_push(tsk, STATE_Producer_t_in);
4     while(T0_todo_list_not_empty(tsk))
5       STATE_T0 current_state=T0_todo_list_pop(tsk);
6         switch(current_state) {
7         case STATE_Producer_t_in :
8           subservice_Producer(task);
9         break;
10    }
11  }
12
13  void subservice_Producer(Task * tsk) {
14    STATE_Producer state=entry_Producer(tsk->Producer);
15    switch(state) {
16    case STATE_Producer_t_out:
17      lock(tsk->port_group1);
18      Producer_transfer_d_out(tsk->Producer,
19        &(tsk->Consumer1->cnx_d_in1));
20      Producer_transfer_d_out(tsk->Producer,
21        &(tsk->Consumer2->cnx_d_in2));
22      // No input trigger port connected
23      unlock(tsk->port_group1);
24      if(Producer_store(tsk->Producer,
25                         STATE_Producer_t_out)) {
26        T0_todo_list_push(tsk, STATE_Producer_t_in);
27      }
28    break;
29    }
30  }
```

Listing 3: Subcomponent scheduling implementation

Listing 3 presents the function that implements the scheduling of the subcomponents executed in task T0. This implementation relies on operations that wrap the execution of those subcomponents. In our example, T0 consists of a single subcomponent, and the entry function of this component (entry_Producer) is wrapped by the function subservice_Producer. As shown in listing 3, this function first executes the subcomponent's entry point (entry_Producer). Then, depending on the outcome of this execution (*i.e.* which output port group was triggered), the wrapping function enters the critical section of the corresponding *conflicting set* (requesting the lock associated with port_group1). Then, output data are transferred to the external view of the connected input data ports (cnx_d_in1 and cnx_d_in2), and triggers to the connected input ports; in this example, there are no connected input trigger ports. If some input trigger ports had been connected, the commentary line 22 would have been replaced by a code snippet equivalent to lines 2 and 3, for each connected port.

Finally, the lock is released and then an operation Producer_store is called to keep track of already triggered outputs, and if not all output ports have been triggered, Producer_t_in is put back in the task's todo list in order to be resumed (line 25).

4.3 Preserve the Triggering Order

As mentioned in the introduction of this section, the considered example was chosen in order to illustrate the atomic/instantaneous broadcast of data from the producer to both consumers. In addition, to enforce the respect of triggering order for components that are connected to clocks or events, we have proposed in section 3.4 a specific priority management policy. The usage of this policy is illustrated in the implementation of T1_schedule in listing 4. The corresponding task (T1) is activated with the ceiling priority of the lock corresponding to the conflicting set of the port group containing t_in1. T1_schedule first enters the critical section in which the input interface code (Consumer1_t_in1, see listing 1) is executed. Of course, in case ICPP is used to configure the lock, this locking action is useless and removed. We kept it here since the formal verification we propose in section 5 only relies on a FIFO locking pattern, which is more general than ICPP. When the operation is terminated, the resource is released and the priority is set to the expected priority according to the scheduling policy (see line 6). The scheduling of subcomponents is executed in a similar way to the one described above for T0_schedule. Before exiting, T1_schedule resets the priority of the task to the ceiling priority of the lock.

```
1   void T1_schedule(Task * tsk) {
2     lock(tsk->port_group1);
3     if(Consumer1_t_in1(tsk->Consumer))
4       T1_todo_list_push(tsk, STATE_Consumer1_t_in1);
5     unlock(tsk->port_group1);
6     set_functional_priority (tsk);
7     while(T1_todo_list_not_empty(tsk)) {
8       STATE_T1 current_state=T1_todo_list_pop(tsk);
9           switch(current_state) {
10          case STATE_Consumer1_t_in1 :
11            subservice_Consumer1(task);
12          break;
13    }
14    reset_priority (tsk);
15  }
```

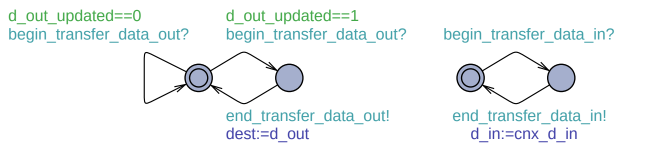
Listing 4: Task with activation priority different from functional priority

5. GENERATED CODE FORMALIZATION

In order to extend the verification capabilities presented in [20], we propose in this section a formalization of the generated code. Beyond the verification of particular examples, this formalization aims at proving that the proposed implementation respects the formal definition of the component model. As a first step, we verify that the generated code we presented in previous section ensures the atomic broadcast characteristic. In this section, we present the corresponding formal models and the properties we verified. For this verification process, we rely on UPPAAL [10] models since an existing formalization of the ProCom semantics was already specified using this formalism [20].

5.1 Interfaces Models

Figures 4 (a) and (b) represent generic models of interfaces dedicated to data transfer, corresponding to the operation implementations described in listings 1 and 2, respectively. These are generic models that contain parameters (begin_transfer_data_out, d_out, d_out_updated, dest and end_transfer_data_out in (a), and begin_transfer_data_in, d_in, cnx_d_in and end_transfer_data_in in (b)), that can be instantiated for individual component ports. In our example, the transfer data out model is thus instantiated once (corresponding to port d_out), while the transfer data in model is instantiated twice (for d_in1 and d_in2).



(a) Transfer Data Out (b) Transfer Data In

Figure 4: Formalization of component interfaces

5.2 Tasks Body Model

Figure 5 illustrates the formal model corresponding to the operation subservice_Producer in listing 3. When the operation is called (modeled by the transition Producer_t_in?), the model simulates the call to the component entry point (entry_Producer!). When the output trigger is reached (modeled as a synchronization STATE_Producer_t_out? for simplicity), the lock of the conflicting set is requested (lock_group1!), data is transferred to Consumer1 (begin_transfer_to_d_in1 is instantiated to match the instantiation of begin_transfer_data_out in the corresponding instance of the model in figure 4 (a)), and to Consumer2. Finally, the lock is released (unlock_group1!).

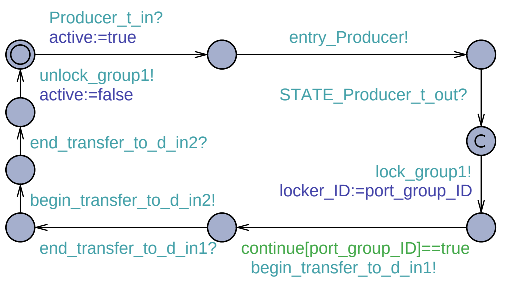


Figure 5: Formalization of the Producer wrapper

The models for the corresponding operations wrapping Consumer1 (shown in figure 6) and Consumer2 follow the same principles, but focus on input aspects.

5.3 First-in/First-out Lock

Figure 7 represents the generic model of the first-in/first-out locking mechanism. The initial state, called Free, is left when receiving a locking command (lock?). The identifier of the task requiring the lock (stored in locker_ID) is then copied to a cyclic buffer (buf). The task can then continue its execution (represented in the model by continue[locker_ID]:=true) and the automaton reaches the Locked state. From this state, if another task reaches the critical section (lock? synchronization), it is then blocked (continue[locker_ID]:=false) and the

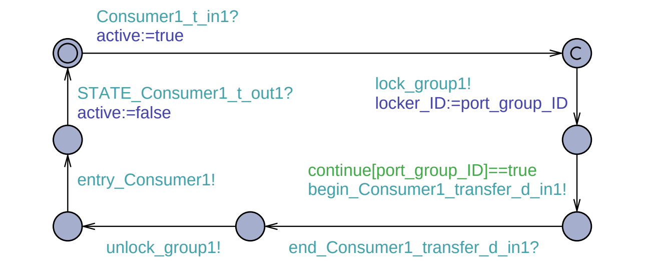


Figure 6: Formalization of the Consumer1 wrapper

corresponding identifier is stored in the buffer. When unlocked (unlock? synchronization), either there are elements in the buffer (current!=storage), in which case the corresponding task is resumed (represented by continue[buf[current]]:=true), or the buffer is empty (current==storage) and in this case the lock is freed and the internal variables are reset.

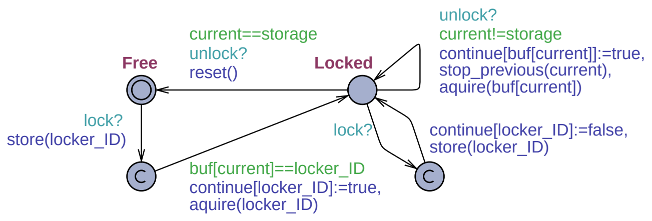


Figure 7: Formalization of the FIFO locks

5.4 Closing the Model

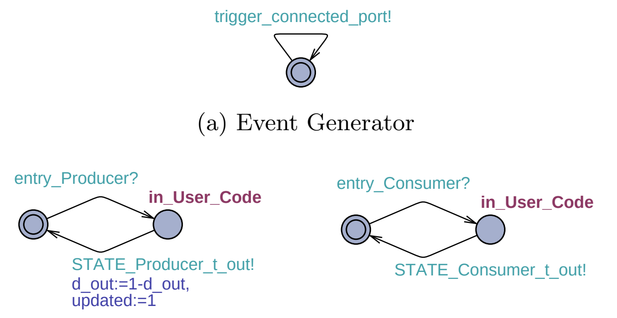
Figure 8 presents the modeling elements that enable to close the model. The generic model in (a) is instantiated three times to trigger the execution of T0, T1, and T2. The automata in (b) and (c) represent the internal implementation of the components (producer and consumers respectively).

Figure 8 (b) shows how the produced value (d_out) is updated each time Producer is triggered (the value alternates between 0 and 1). Figure 8 (c) is the generic model for the two consumers implementation. This model exhibits an interesting state regarding the property we intend to verify: in_User_Code is the state in which the corresponding instance of consumer has been triggered and performs internal computations.

5.5 Verification

In order to verify the instantaneous characteristic of ProCom interactions, we include in the model an observer automaton, shown in Figure 9, that follows the actions of interest, and thus simplifies the formulation of correctness properties. The two states sync1 and sync2 represent that Consumer1 and Consumer2 were triggered (in the figure represented by t_in1? and t_in2?, respectively) without new data being produced by Producer in between (t_out? in the figure).

In order to validate the formalization, we have verified a set of properties that show the reachability of the different relevant states, and the absence of deadlock in the model. In addition, we have verified the following property that expresses the atomic/instantaneous interactions between the components:



(a) Event Generator

(b) Producer Entry Point (c) Consumer Entry Point

Figure 8: Formalization of event generator and entry points

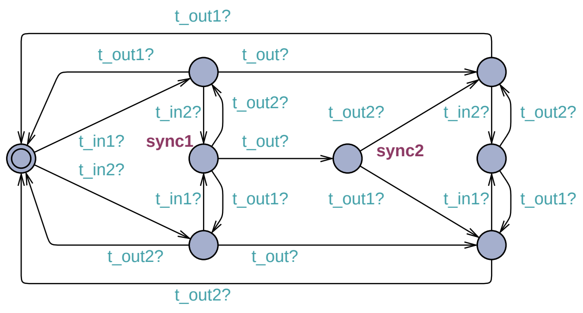


Figure 9: Verification of the instantaneous broadcast: Observer automaton

Property_1 = **A**[] ((OBS.sync1 || OBS.sync2)
&& C1.in_User_Code
&& C2.in_User_Code)
imply d_in_1 == d_in_2

where C1, C2, and OBS are the final instances of Consumer1, Consumer2, and the observer in the UPPAAL model. The verification required the exploration of 25837 states. Of course, removing the proposed locking scheme leads to a violation of the considered property.

In order to verify that the model we proposed enables interleaving of consumers and producer, we also verified the following property:

Property_2 = **E**⟨⟩ d_in_1 != d_in_2
&& C1.in_User_Code
&& C2.in_User_Code

This property represents the possibility that Consumer1 and Consumer2 have different input values while they are both executing their respective functionality. In our model, this situation can only occur when t_out has been triggered in between triggers of t_in1 and t_in2. Indeed, the opposite situation has been covered by the verification of Property_1 and shows both consumers receive the same value. Thus, Property_2 shows that we did not verify with Property_1 a trivial model in which consumers always receive the same input data independently of the interleaving scheme.

5.6 Discussion

As one can notice in figures 5 and 6, these two UPPAAL models contain "committed locations". This modeling pattern is an approximation that consists of neglecting the possibility of preemption by another task during the time spent in a rather small section of the generated code. The general output code pattern, when the component has several output port groups, is illustrated in listing 5 and the corresponding model in figure 10. The committed state corresponds to the execution from line 2 to line 5.

The motivation for this simplification is that the observer automaton synchronizes with the `STATE_t_out1` action, but the semantically significant action is actually the lock request. The correctness of the simplification can be argued by noting that no interaction with the rest of the system happens between the triggering and the lock request, which means that a preemption in this interval is equivalent to a preemption immediately before the triggering.

```
1   ...
2   STATE current_state=Component_trigger_out(...)
3   switch(current_state){
4     case STATE_t_out1:
5     lock(lock_group1);
6      ...
7     case STATE_t_out2:
8     lock(lock_group2);
9      ...
10  }
11   ...
```

Listing 5: General output code pattern

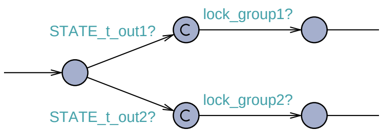


Figure 10: Formalization of the output code pattern

6. RELATED WORK

In this paper, we have presented the synthesis process of a component model that relies on an asynchronous semantics and a formal hypothesis of atomic and instantaneous interaction. We have also verified that the generated code respects these aspects of the semantics. In this section, we briefly compare this contribution to related works, on the one hand considering the synthesis of formally defined component models, and on the other hand considering the synthesis of component models relying on an asynchronous semantics.

In the domain of embedded real-time systems, when it comes to component models with a formally defined semantics, synchronous languages constitute an important category. Compared to synchronous languages, the main semantic difference is that execution of functionalities of ProCom components is not considered as instantaneous. Considering the execution of the components functionality as instantaneous limits the deployment flexibility, *e.g.* the modularity and reusability of existing components because of possible instantaneous cyclic dependencies [13]. Relaxing the synchronous hypothesis, ProCom answers, by definition, to this cyclic instantaneous dependency. To verify the system's behavior despite of the relaxation of the synchronous hypothesis, it is necessary to model in more details than with a synchronous language the internal behavior of a component. As a benefit, the formal verification can be extended to more features of the system's behavior.

Compared to other component models for real-time embedded systems (MyCCM-HI [5], for instance), ProCom imposes a number of behavioral restrictions that eases analysis of the software application. Applying similar restrictions to an existing component model should make the solution presented in this paper beneficial in that context too.

Rubus [9] is a commercial real-time component model originating from the same ancestor as ProCom. Since the two share many characteristics, parts of the presented synthesis scheme should be applicable to Rubus as well. However, since Rubus lacks formal operational semantics, it is not obvious what semantical properties we would have to guarantee.

If the verification of a ProCom component model requires to model the internal behavior of a component, this model may abstract away the details of the component's implementation. This concept of gray-box component contrasts with approaches that require to model in details the internal behavior of the components (such as the BIP framework [3]).

On the other hand, code generation and verification based on models that provide no formal hypothesis on the component model (such as Ocarina/AADL [11]) can require the exploration of a huge state space [14]. The specification of instantaneous/atomic interactions enable to model a complex interaction as a reduced set of states and transitions.

7. CONCLUSION AND FUTURE WORKS

The advantages associated with component-based development, including improved modularity and ease of reuse, are clearly desirable in the domain of embedded systems. However, characteristics such as resource and real-time constraints, mean that traditional CBSE approaches intended for desktop or enterprise applications, must be adapted to fit this domain. ProCom is a component model specifically targeting critical, distributed embedded systems with strong requirements in terms of resource usage and timeliness. It is based on a rich design-time component concept, facilitating reuse of extra-functional properties as well as code, and a formal operational semantics providing support for formal analysis through, *e.g.*, model checking. To address resource limitations, ProCom envisions a synthesis phase in the development process, where the rich component-based design is translated into an efficient runtime system consisting of asynchronous tasks executed in a lightweight real-time operating system.

In this paper, we have presented the basic code generation strategy for ProCom, focusing on the generated code wrapping the internal component computations to handle the component interactions in terms of data and control transfer. We also described the locking scheme developed to guarantee that interactions occur in accordance with the formal semantics (*e.g.*, respecting the atomic and instantaneous properties of data transfer) without introducing unnecessary overhead. We also showed how the generated code can be formally modelled using timed automata, and described the first step in proving conformance to the formal

ProCom semantics, by verifying that the integrity and order of interactions is preserved by the locking scheme.

Our current and future work includes extending the code generation to cover also the component interaction used in the top layer of ProCom, *i.e.*, asynchronous message passing instead of data- and trigger transfer. At this level, the generated code should also be designed for hierarchical scheduling. We will also complete the conformance validation initiated in this paper.

Moreover, different aspects of the current code synthesis can be optimized, *e.g.*, targeting increased efficiency by identifying cases where the general, safe implementation can be replaced by efficient alternatives without changing the observable result. Other optimizations, *e.g.*, simplifying the control flow of the generated component code and making it more static, would aim at increasing the code-level analysability. Finally, we want to investigate the possibility of partial flattening, which would allow more flexibility in the tradeoff between efficiency and reusability.

Acknowledgements

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and by the Swedish Research Council project CONTESSE (2010-4276).

8. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, Apr. 1994.

[2] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3:67–99, April 1991.

[3] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[4] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.

[5] E. Borde, P. H. Feiler, G. Haïk, and L. Pautet. Model driven code generation for critical and adaptative embedded systems. *SIGBED Rev.*, 6:10:1–10:5, October 2009.

[6] A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[7] I. Crnkovic. Component-based software engineering for embedded systems. In *Proceedings of the 27th International Conference on Software engineering*, ICSE'05, pages 712–713. ACM, 2005.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[9] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck. The Rubus component model for resource constrained real-time systems. In *3rd International Symposium on Industrial Embedded Systems*, pages 177–183. IEEE, June 2008.

[10] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[11] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. In *Reliable Software Technologies'09 - Ada Europe*, Brest, France, jun 2009.

[12] T. Leveque, E. Borde, A. Marref, and J. Carlson. Hierarchical composition of parametric WCET in a component based approach. In *14th IEEE International Symposium on Object/Component/ Service-oriented Real-time Distributed Computing (ISORC'11)*. IEEE, March 2011. To appear.

[13] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 78–89, New York, NY, USA, 2009. ACM.

[14] X. Renault, F. Kordon, and J. Hugues. From AADL architectural models to Petri nets: Checking model viability. In *Proceedings of the 12th IEEE International Symposium on Object/Component/ Service-Oriented Real-Time Distributed Computing*, ISORC'09, pages 313–320. IEEE, 2009.

[15] C. Seceleanu, A. Vulgarakis, and P. Pettersson. REMES: A resource model for embedded systems. In *14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09)*. IEEE, June 2009.

[16] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković. Integration of Extra-Functional Properties in Component Models. In *12th International Symposium on Component Based Software Engineering (CBSE 2009)*. Springer Berlin, LNCS 5582, June 2009.

[17] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A component model for control-intensive distributed embedded systems. In *Component-Based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 310–317. Springer Berlin / Heidelberg, 2008.

[18] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[19] A. Vulgarakis, S. Sentilles, J. Carlson, and C. Seceleanu. Integrating behavioral descriptions into a component model for embedded systems. In *36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 113–118. IEEE, September 2010.

[20] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson. Formal semantics of the ProCom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA'09, pages 478–485. IEEE Computer Society, 2009.

[21] H. Zheng, H. Yao, and T. Yoneda. Modular model checking of large asynchronous designs with efficient abstraction refinement. *IEEE Transactions on Computers*, 59(4), April 2010.