Abstract—In model-based development, executable software (e.g., C or Java code) can be generated from a high-level model using a code generator. However, the execution of the generated software on a target platform remains a challenge due to a mismatch in communication semantics assumed by the model and the platform-dependent software (e.g., sampling/actuation routines). This paper proposes an input/output (I/O) interface module that bridges this semantic gap by means of buffers and interface policies, which explicitly capture the information required to adapt the model’s communication semantics to that of the platform. We present a framework that can be used to systematically synthesize—directly from the model—the I/O interfaces and accompanying APIs that the generated software and the platform-dependent software need to communicate with one another. Our interface policies can also encode relaxations of a model semantics that may not be implementable, thus making derivations of the implemented systems from the model traceable. We illustrate the applicability and the benefits of our framework with a case study of an infusion pump.

I. INTRODUCTION

The rapid growth in size and complexity of software in real-time embedded systems has led to a wide adoption of model-based development to achieve safe and effective software design. In this approach, the software is developed using high-level models that abstract the software behaviors, and these models can be analyzed and verified using toolsets, such as model-checkers and model-based testers. The verified models are then used as inputs to a code generator, which automatically generates the executable code\(^1\) that will be executed on a target hardware platform. This automated methodology not only enables effective software development, it also improves system safety because the information encoded in the model is traceable down to the implementation through tool support.

Despite the available tool support, several challenges remain in the software integration stage, where the generated (platform-independent) code needs to interact with the platform-dependent code, such as platform-specific sensor/actuator processing routines. Typically, the platform is not explicitly captured in the early modeling phase (due to the lack of knowledge about the target platform or to reduce modeling complexity) but instead, the system model is verified under an abstract environment model (see Fig. 1). However, during the software integration, the generated code of the system model is deployed on a real platform, which may handle communication differently from the communication semantics assumed by the model. This mismatch in communication semantics leads to a need for an adaptation of the semantics assumed by the model to fit that of the platform.

The current approach to the above adaptation is to use glue code, which is often written manually without following any safety enforcement. This is a tedious and error-prone process. Further, some model semantics may not be implementable. For instance, transitions on a synchronous channel in an UPPAAL model [4] should be taken at the same time, and they should take zero time. This semantics is difficult to realize in the target platform due to platform-specific factors, such as communication delay or scheduling policy constraints; as a result, it needs to be relaxed. The relaxation of the model semantics needs to be captured precisely, so that any unexpected errors or discrepancies between the implemented system and the model are traceable.

To address the above challenges, we propose a framework for synthesizing I/O interface modules that perform the adaptation between the communication semantics of the platform-independent and platform-dependent code. These two types of code can now communicate with each other using (only) the APIs of the interface modules. Systematically generated from the model, both the interfaces and their APIs are correct-by-construction. Their goals are to establish a clean separation between the platform-independent and platform-dependent code using the interfaces, so that the platform-independent code can be safely ported to different platforms, and to minimize the needs for ad hoc glue code.

Contributions: Towards the above framework, we make the following contributions:

- We propose the use of buffers and the concept of interface policy that explicitly describes the information that is needed to implement the adaptation of the model’s communication semantics to that of the platform;
- Based on the interface policy concept, we present a synthesis method for generating systematically from the model the I/O interfaces and their APIs, which are used for the communication between the two types of code.

II. OVERVIEW OF OUR APPROACH

Fig. 1 shows the mapping between a model and the implemented system in the model-based approach. Our goal is to develop a framework that systematically synthesizes the I/O interfaces between the platform-independent software and the platform-dependent software within the implemented system. This synthesis is illustrated by the dashed arrow (2) in Fig. 1.

In the above mapping, the model is described by a high-level formalism (e.g., UPPAAL, Stateflow, or extended finite state machine model). It is composed of (i) a system model that represents the software system (e.g., infusion pump software),

---

\(^1\)In this paper, we use the terms “code” and “software” interchangeably.
and (ii) an environment model that abstracts the platform on which the system operates. We assume that the model has been formally verified against the desirable properties given by the system specifications under the environment model.

In the implemented system, we distinguish platform-independent software, which is generated from the system model, and platform-dependent software, such as I/O device drivers, which are specific to a particular device. The two are connected by an I/O interface. The entire software stack runs on a platform hardware, such as a Lifecare 4100 PCA infusion pump, and can interact with the environment, e.g., the patient and the operating room.

As is illustrated in Fig. 1, the code generation step (denoted by the dashed arrow (1)) only generates code for the system model and not for the environment model. Instead, the environment model is replaced with a target platform (which contains the platform-dependent software and the platform hardware) in the implemented system. Since the communication semantics assumed by the model can be different from the way the platform handles communication, an adaptation between the two semantics is required. The role of the I/O interface is to perform this adaption.

The I/O interface module generated by our framework provides an API, which contains the signatures of two methods called read() and write(), through which the platform-independent and the platform-dependent software communicate with one another. We refer to this API as interface API, and we assume that the communication between the two types of software is done exclusively via the interface API.

We follow a series of steps to generate the I/O interfaces. First, we classify the different communication semantics of the model and annotate them in a systematic manner (Section III). We then formally define the system boundary (Section IV) and use this specification to define an interface policy that contains the necessary information to perform the adaptation (Section V). Finally, we introduce a method for systematically generating the I/O interfaces and the interface APIs based on the derived interface policy (Section VI).

III. THE MODEL AND PLATFORM

In this section, we use an example of a PCA (Patient Controlled Analgesia) infusion pump to explain the communication semantics that the platform-independent software expects from the target platform. A PCA infusion pump is a special type of infusion devices that delivers a certain amount of drugs, called bolus, when the patient requests it by pressing the patient-pendant. Here, we present part of the system model and the platform that were studied in our previous work [6] using the Generic PCA (GPCA) infusion pump model [1].

A. The GPCA Model

Fig. 2 shows part of the system model of the GPCA infusion pump system. The (partial) model is a network of timed automata [3] that consists of two controllers: the Alarm-Detecting Controller (ADC) and the State Controller (SC). The ADC interacts with the platform (described in Section III-B) to detect any abnormal conditions, such as occlusion or door-open, and informs the SC, which stops the current infusion and raises an audible alarm on such conditions. The environment model that abstracts the platform behavior is not shown in the figure. The code for both controllers is generated automatically using a code generator (c.f. Fig. 1).

Communication semantics: The model supports the synchronous communication over channels and the asynchronous communication over shared variables. For example, the communication over the channel BolusReq in Fig. 2 is a synchronous communication. When the environment model (sender) and the SC (receiver) are ready for the synchronization (i.e., the transitions annotated with BolusReq! and BolusReq? are active), they take the corresponding transitions at the same time, and the transitions take zero time to complete. In contrast, the communication over the shared variable Occlusion_stat is an asynchronous communication. Here, the ADC can take the transition to the state L2Alarm whenever the value of the variable Occlusion_stat is higher than a predefined upper limit, which happens independently of when and how this variable is updated by the environment model. We postpone our discussion on the implication of these semantics on the implemented system to later sections.

We highlight two types of communication, internal and external. Internal communication is the communication within the system model, e.g., the channel LevelTwoAlarm between the ADC and the SC. The code for this type of communication is generated automatically from the model; thus, its semantics is preserved in the platform-independent software. In contrast, external communication is the communication between the system model and the platform, e.g., the channel BolusReq...
between the SC and the (hidden) environment model. Only the system model side of the communication is translated into code.

![Diagram of communication](image)

Fig. 3: Four kinds of external communication.

Fig. 3 illustrates the four categories of external communication. Since internal communication semantics is assumed to be preserved by the code generation, our I/O interface synthesis framework concerns only external communication.

### B. The PCA Infusion Pump Platform

A typical PCA pump platform, such as the ones shown in Fig. 4, has several types of sensors and accompanying software routines to detect abnormal conditions, such as occlusion, door-open, and air-in-line conditions. For example, the condition empty-reservoir can be detected by a switch sensor, which is located at the bottom of the pump (Fig. 4-(4)). The platform also has several actuators, including typically a pump motor (Fig. 4-(2)) that is used to move the syringe-bar up and down to inject fluid in a controlled manner and a buzzer (Fig. 4-(7)) that is used to make alarm sound during abnormal conditions. Different infusion pump platforms have different software routines that interact with the platform’s sensors and actuators.

![Diagram of infusion pump](image)

Fig. 4: Experimental infusion pump platforms.

**The gap in communication semantics:** As we can draw from the above description, there is a gap in the model’s communication semantics and how the platform handles communication. For example, the synchronous communication over the channel BolusReq in the model is performed by taking the two transitions (BolusReq! and BolusReq?) at the same time with zero communication delay. In the platform, however, any communication takes non-zero time; for example, sampling/interrupt routines take a certain time delay to get the status of the bolus request button. The goal of this paper is to identify the information that is necessary to fill this semantic gap, and further, to generate the I/O interfaces based on such information.

We note that in external communication, the platform side (e.g., the one that sends the event BolusReq! or updates the variable occlusion-stat) is abstracted by the environment model, which is not shown in Fig. 2.

### IV. Separation of the System Boundary

This section introduces the relationship between system components and their interactions at the system boundary. We specify the system boundary using the four-variable model [9], a well-known formalism in the requirements engineering domain. In this formalism, the separation among subsystems is explicitly defined using four variables, namely, monitored (m), controlled (c), input (i) and output (o). The interactions among subsystems are specified using the NAT, REQ, IN, OUT and SOF relations on these variables. We will use the IN, OUT and SOF relations to define the system boundary.

![Diagram of system boundary](image)

Fig. 5: System boundary of an infusion pump. This figure is motivated by the work in [11], which conceptualizes the four-variable model.

Fig. 5 shows the system boundary from the four-variable model perspective. The boundary separates the software from the input and output devices. In this context, the Software refers to the platform-independent code that is generated from the model; for instance, the SC and the ADC in Fig. 2 are encoded in the Software. The Input-Device and the Output-Device refer to the platform-dependent code, such as sampling and actuation routines. There are two types of interfaces: the input interface is used for the interaction between the Input-Device and the Software, and the output interface is used for the interaction between the Software and the Output-Device.

We note that the system boundary given in Fig. 5 explicitly represents the mapping in Fig. 1. Here, the input and output of the infusion system to the real environment in Fig. 1 is specified using m-variables and c-variables. The I/O interface in Fig. 1 consists of the input interface and output interface. Finally, the platform-dependent software in Fig. 1 is specified by the Input-Device and the Output-Device. We next introduce an example to explain the mapping in the four-variable model.

**Example 1:** Consider the infusion pump platform shown in Fig. 4. In this platform, the infusion pump’s switch sensor checks the low-reservoir condition, through sampling or interrupt mechanism, and converts it into different levels of alarm that are used by the control software. The control software sends output signals to the pump motor and the buzzer, which then update the variables flow-rate and noise-decibel, respectively.

In this example, the switch sensor is the Input-Device and its sampling routines implement the IN-relation; the control-software is the Software, which implements the SOF-relation; and the pump motor and the buzzers are the Output-Device, and their actuator routines implement the OUT-relation. The physical position of the syringe, the input alarm level to the control software, the output signal from the control software, and the variables flow-rate and noise-decibel are encoded by
\( m, i, o \) and \( c \), respectively. Finally, the Input-Device converts \( m \) into \( i \) using the IN-relation; the Software takes \( i \) as input and updates \( o \) using the SOF-relation; and the Output-Device converts \( o \) to \( c \) using the OUT-relation.

We assume the code that implements the SOF-relation is generated from the model through a code generation process that ensures the conversion from \( i \) to \( o \) is correct by construction. In the coming sections, we will describe the I/O interface that captures the adaptation between the communication semantics of the IN/OUT and SOF relations.

V. I/O Interface and Interface Policy

In this section, we give the definition of a generic I/O interface and describe the concrete interfaces corresponding to the different communication semantics of the model.

A. I/O Interface

An I/O interface is defined based on the input \((i)\) and output variables \((o)\) that connect the Input-Device (Output-Device) and the Software (see Fig. 5). We assume that these variables can be extracted from the external communication of the system model.

Let \( \mathbf{I} \) and \( \mathbf{O} \) be the sets of input and output variables, respectively, of the platform-independent software. For each variable, we create a FIFO buffer that is used to store new values of the variable before they are being consumed (read) by the receiver. Each buffer is associated with an interface policy, which describes the necessary information to implement the communication semantics (assumed by the model) that is associated with the corresponding variable. The exact parameters of an interface policy are described in Section V-B.

**Definition 1 (Input and Output Interface):**

An input [output] interface is a set of pairs, \( \{(B_1, P_1), (B_2, P_2), ..., (B_n, P_n)\} \), where \( B_k \) is the FIFO buffer associated with the \( k \)-th input [output] variable. \( P_k \) is the interface policy imposed on \( B_k \), for all \( 1 \leq k \leq n \), where \( n \) is the number of input [output] variables.

As was mentioned in Section II, the input (output) interface provides an API, which contains the \texttt{read()} and \texttt{write()} method signatures that the Input-Device (Output-Device) uses to communicate with the Software. For example, the Input-Device sends a new value of an input variable \( i_k \) by calling the \texttt{write()} method, and the Software reads \( i_k \) by calling the \texttt{read()} method. In our framework, both the input (output) interface and its API are synthesized from the model. Next, we describe the interface policy.

B. Interface Policy

The communication semantics provided by the model (shown in Fig. 3) does not explicitly specify all necessary information to generate concrete I/O interfaces. For example, for the synchronous communication shown in Fig. 3-(a), the model implicitly assumes that the platform-independent code processes the request \texttt{BolusReq} immediately after it is delivered by the platform-dependent code. This zero time delay (implicitly assumed in the model) needs to be captured in the generated I/O interface to ensure that communication over the variable \texttt{BolusReq} follows the synchronous semantics. We use an interface policy to explicitly specify the information required to systematically implement the communication semantics associated with a variable. Such a policy imposes constraints on the read and write operations of the buffer associated with the variable.

In the following, we define a generic interface policy. We then show how concrete interface policies corresponding to the model’s communication semantics can be instantiated.

**Definition 2 (Interface policy):** An interface policy imposed on a buffer is defined by a tuple \( P = \{\text{Delay}, \text{Bufsize}, \text{Overrun}, \text{Access}\} \), where

- \( \text{Delay} \in \mathbb{N} \) is the maximum delay (in time units) permissible from the instant a value is written into the buffer until it is read.
- \( \text{Bufsize} \in \mathbb{N} \cup \{\infty\} \) is the size of the buffer (i.e., the maximum number of values that can be stored in the buffer), where \( \infty \) denotes an unbounded (infinite) buffer size. We require that \( \text{Bufsize} \geq 1 \).
- \( \text{Overrun} \in \{\text{OVERWRITE, DROP, DISALLOWED}\} \) is the buffer overrun scheme when a component tries to write a new value when the buffer is full. In the OVERWRITE scheme, the new value overwrites the oldest value in the buffer. In the DROP scheme, the new value is dropped and the current values in the buffer are unaffected. In the DISALLOWED scheme, values should not be lost and thus, an exception indicating a violation of the semantics should be raised.
- \( \text{Access} \) is an access control policy that specifies which components among the Input-Device, the Output-Device and the Software can access the buffer, as well as their access types. We consider three types of access permission: Read, Write, and Read/Write. If a component is not specified in \( \text{Access} \), it has no access to the buffer.

Next, we show via an example how the above generic interface policy can be instantiated to specify the information needed to synthesize the I/O interfaces for the synchronous and asynchronous communication semantics.

**Interface policy for synchronous communication:** Consider the synchronous communication over the channel \texttt{BolusReq} in Fig. 2. In this example, the Input-Device and the Software participate in the synchronization. The interface policy of this communication semantics are given by:

1. \( \text{Delay} = 0 \): any synchronization between the Input-Device and the Software takes zero time;
2. \( \text{Bufsize} = \infty \): the buffer size should be infinite \([\infty]\) to ensure no bolus requests from the Input-Device are lost;
3. \( \text{Overrun} = \text{DISALLOWED} \): if the buffer is full, an exception is raised, since requests should not be lost;
4. \( \text{Access} = \{\text{Input-Device, Write, Software, Read}\} \): the Input-Device only needs the Write permission, since it is the sender of the request, whereas the Software only needs the Read permission, since it is the receiver. Since the Output-Device does not participate in the synchro-
nization, it has no access to the buffer of BolusReq.

**Interface policy for asynchronous communication:** Consider the asynchronous communication over the shared variable Occlusion_stat. In this case, the Input-Device and the Software participate in the communication.

1. \( \text{Delay} = \infty \): the semantics does not impose any timing constraints on when the reading and writing of the shared variable happen;
2. \( \text{Bufsize} = 1 \): the buffer size should be 1 according to the semantics of the shared variable itself;
3. \( \text{Overrun} = \text{OVERWRITE} \): in case the buffer is full, the shared variable keeps the newest value;
4. \( \text{Access} = \{ \text{(Input-Device, Write)}, \text{(Software, Read)} \} \): the Input-Device only needs the Write permission, since it only updates the variable whereas the Software only needs the Read permission, since it only reads the variable. Since the Output-Device does not participate in the asynchronous communication, it has no access.

As the above examples show, an interface policy gives the information required to implement the communication semantics; this information was not explicitly specified in the model. Further, it provides customizable parameters that can be used to implement the semantics relaxation in a traceable way, as we explain next.

**Interface policy for customized semantics:** As was discussed in the earlier sections, the interface policy derived directly from the communication semantics may not be implementable. For example, the communication semantics specified by the BolusReq interface policy requires that it should take zero time from the instant the bolus request is produced by the Input-Device until the instant the Software processes the request. This is unrealistic from the implementation perspective: due to limitations of the platform, such as the speed of the communication medium or communication policy, it takes some delay to deliver any event from the Input-Device to the Software across the system boundary. Further, it is not possible to have infinite buffers in practice. As a result, customizations of the original semantics are needed to make the model implementable.

The generic interface policy in Definition 2 can also be instantiated to express customized semantics. For example, consider the following relaxed communication semantics of the synchronous communication over the channel BolusReq:

1. allow up to 100 ms delay from the instant a bolus request is sent by the Input-Device until the instant it is received by the Software, and
2. allow up to 5 pending requests at any instant in time. (The modified parameters (e.g., 100ms and 5) are typically obtained from the platform-dependent analysis, such as Worst-Case Execution Time (WCET) or end-to-end delay analysis, which is beyond the scope of this paper.) The interface policy for this relaxed semantics can be obtained by modifying the original policy by setting \( \text{Delay} = 100 \) (ms) and \( \text{Bufsize} = 5 \), while keeping other parameters unchanged. We will refer to this relaxed policy \( P_{\text{BolusRelax}} \) to demonstrate the interface synthesis in the next section. We note that how the relaxed policy affects the behavior of the platform-independent code is beyond the scope of this paper; we refer readers to related work, such as [13], which considers how a relaxed semantics may affect the safety properties of the model.

**VI. INTERFACE SYNTHESIS**

Fig. 6 shows the workflow for systematically synthesizing the I/O interfaces from the interface policies obtained in Section V. Its building blocks are described below.

![Workflow of the interface synthesis.](image)

**Extracting external I/O:** The model may include both external and internal communication; however, the UPPAAL model does not explicitly distinguish between them. We will show how the different classes of external communication semantics (c.f. Fig. 3) can be extracted from the model in a systematic manner under the following assumption:

**Assumption 1:** The environment model is annotated a priori. Under Assumption 1, all synchronous channels and shared variables used in the environment model are classified as external communication. These are classified into input and output types, depending on whether they are used by the environment model as a sender (writer) or receiver (reader). Using the above classification, we can systematically obtain the four types of communication semantics shown in Fig. 3.

<table>
<thead>
<tr>
<th>Input variable (i)</th>
<th>Output variable (o)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_{\text{BolusReq}} )</td>
<td>( o_{\text{BolusStart}} )</td>
</tr>
<tr>
<td>( i_{\text{BolusFinish}} )</td>
<td>( o_{\text{BolusProcessed}} )</td>
</tr>
<tr>
<td>( i_{\text{Occlusion_stat}} )</td>
<td>( o_{\text{RaiseAlarm}} )</td>
</tr>
<tr>
<td>( i_{\text{ClearAlarm}} )</td>
<td>( o_{\text{SilenceAlarm}} )</td>
</tr>
<tr>
<td>( i_{\text{DoorStat}} )</td>
<td>( o_{\text{InfuProgress}} )</td>
</tr>
</tbody>
</table>

As an example, Table I shows the obtained variables extracted from Fig. 2. The columns \( i \) and \( o \) indicate the input and output variables, annotated with their communication types: \( S \) and \( A \) indicate that communication over the associated variable is synchronous and asynchronous, respectively.

**Input/Output Semantics Customization:** As was mentioned in Section V, the original communication semantics imposed by the model and its derived interface policy may not be implementable. Therefore, the interface policy can be customized to relax such semantics. We assume that the concrete parameters to be relaxed are given.
**Input/Output Interface Synthesis:** From the previous steps, we obtain the (i) input/output variables extracted from the model (e.g., Table I), and (ii) the (customized) interface policy. We will transform this information into code for the interface systematically. Below, we explain the synthesis of the input interface (defined in Definition 1); in particular, we show how to generate the FIFO buffers associated with the input variables and the concrete interface policies via an example of the model in Fig. 2 and the relaxed interface policy $P_{BolusRelax}$ in Section V-B. Synthesis of the output interface is analogous.

![Fig. 7: The input interface generated from the model in Fig. 2.](image)

**TABLE II: The exceptions of the input interface**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Parameter</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request-Timeout</td>
<td>variable-index</td>
<td>A request for synchronization associated with the variable is not read by the Software within the timeout.</td>
</tr>
<tr>
<td>Buffer-Full</td>
<td>variable-index</td>
<td>The input buffer is full in case DISALLOWED policy is used.</td>
</tr>
<tr>
<td>Invalid-Access</td>
<td>expected-operation</td>
<td>A [write] operation by the Software does not conform to the data-flow direction of the interface.</td>
</tr>
</tbody>
</table>

Fig. 7 shows the input interface generated from the model in Fig. 2. The Input-Device (ID) can access the input interface through calling `write(BolusReq)` method; the Software (SOF) can access the input interface through calling `read(BolusReq)` method. The input interface processes these requests as below.

For each input variable extracted from the model, a buffer is created (e.g., the buffer for `i_BolusReq`) with its associated interface policy. The size of the buffer associated with a channel is extracted from the interface policy; for example, the size of the buffer associated with `i_BolusReq` under the relaxed policy in Section V-B is 5 events. Further, if the Input-Device produces a request for synchronization when the buffer is full, the input interface triggers a `buffer-full exception` that should be handled by the Input-Device. On the contrary, the buffers associated with shared variables (e.g., `i_Occlusion_stat`) are of size 1, and they implement OVERWRITE scheme.

A Timeout table (shown in Fig. 7) is created to keep track whether the code conforms to the policy at runtime. When a synchronization request comes from the Input-Device, its arrival time is timestamped and the timeout-counter – which indicates the remaining time of the request to be processed – starts decreasing as time passes. The timeout value is set to be the delay between the Input-Device and the Software that is specified in the interface policy. For example, the interface policy $P_{BolusRelax}$ in Section V allows at most 100 ms delay once a request on `i_BolusReq` occurs. Thus, the Software is expected to read the request within 100ms; otherwise, a `timeout-exception` occurs and should be handled by the Input-Device. The exception handling mechanism is platform-dependent.

The input interface is unidirectional from the Input-Device to the Software according to the policy assigned in Section V: the Input-Device can only `write` requests for synchronization or shared variables, and the Software can only `read` the requests or shared variables from the input interface. Similarly, the output interface is unidirectional from the Software to the Output-Device, where the access permission is specified in an opposite way. Any invalid access to the input (output) interface results in an `invalid-access exception` that should be handled by the component that caused it. For instance, an `invalid-access exception` would occur if the Output-Device performs a write operation on the output interface.

The exceptions explained above are monitored through the Exception table (the right table in Fig. 7) at run time. Any exception that violates the interface policy is tagged in the table and the associated exception handler is triggered by the input interface. Table II summarizes the exceptions that need to be handled.

**VII. DISCUSSION: USAGE AND BENEFITS**

We explain the utility and benefits of the synthesized I/O interface in the GPCA experimental platform [6].

The experimental platform [6] consists of two infusion pump hardware, namely, the LifeCare 4100 PCA Plus II and the Baxter PCA II syringe pump, which are shown in Fig. 4. We compared the internal hardware structures of the two infusion pumps, and identified a set of sensors and actuators typically found in PCA pump systems to mitigate certain hazards, such as overinfusion or underinfusion.

**Usage of the I/O interface:** A platform is assumed to produce input values (i.e., values that are fed into the platform-independent code). For example, the Patient-Pendant is a switch sensor, i.e., a button that the patient can press to request a bolus (Fig. 4-(5)). The Input-Device converts the button activation event into a request for synchronization over the channel, `BolusReq`; the request is enqueued at the input interface by calling the `write(BolusReq)` method. Then, the Software completes the synchronization by calling the `read(BolusReq)` to accept the request. Similarly, the Output-Device can communicate with the Software through the output interface API; for instance, the Software and the Output-Device communicate by calling `write(BolusStart)` and `read(BolusStart)`, respectively.

**Benefits of the I/O interface:** Different platforms may have different mechanisms to prevent or mitigate hazardous situations; for instance, under-infusion hazards can be avoided by detecting the low-reservoir condition [2]. The Baxter infusion pump has the reservoir-low sensor (Fig. 4-(3)) to detect the condition; however, the LifeCare is not equipped with such...
a sensor. Instead, the LifeCare uses a different mechanism to detect the condition low-reservoir – specifically, an optical switch that counts pump strokes (Fig. 4-(6)); when the counter reaches a certain value, it is considered as a low-reservoir condition.\footnote{This is our conclusion from a previous case study [6]; the software specification is not publicly available.} In addition, the types of pump-motor and user-interface (Fig. 4-(1), (2)) are slightly different in the two platforms, even though they serve the same purpose. Therefore, we expect that the platform-independent code needs to interact with very different platform-dependent code whenever its target platform is changed. The systematically generated I/O interface provides a uniform way to integrate the platform-independent code to a particular platform through [\texttt{write}] and [\texttt{read}] methods; this enforcement reduces the errors during the integration stage.

VIII. RELATED WORK

The protection enforced by the I/O interface module in this paper is related to the role of the \textit{temporal firewall} introduced by Kopetz in [7]; our work extends the notion of firewall into the I/O interface synthesis in the model-based development.

Automatic generation of the glue code was also studied in the context of distributed systems. In particular, the \textit{Ocarina} [8] code generation environment for AADL models also considers what AADL properties are needed to generate the glue code between distributed components. The properties that Ocarina uses to generate the code, such as the queue size and queue overflow behaviors, are related to our policy. However, our work focuses on the adaptation of synchronous/asynchronous communication semantics via both timing and buffer constraints. Our goal is also to account for the variability in the I/O device behaviors of uniprocessor platforms, whereas Ocarina is concerned with distributed systems.

At the device-driver level, Chuo et al. [5] presented a method for generating the glue code for the communication link shared among hardware peripherals to preserve its timing constraints. However, the basis of the synthesis is the specifications of the micro-processors and hardware peripherals, which are of a much lower level of abstraction compared to the high-level models (e.g., UPPAAL) considered in this paper.

Interface synthesis has also been considered extensively in the context of hardware/software co-design. For instance, Rajawat et al. [10] introduced a general interface synthesis flow that can be instantiated for different applications. Their workflow also separates software generation from interface generation, and the system partitioning step is similar to the system boundary separation in our paper. However, Rajawat et al. only gave a high-level workflow without considering the issues in implementing the model’s communication semantics.

Separation of concerns between models and platforms has also been studied in the component-based software engineering field. For example, Waignier et al. [12] proposed the CALICO framework that supports correct composition of component-based systems. The specification of components’ assumptions/guarantees and the code instrumentation in CALICO are similar to the interface policy specification and the I/O interface generation. However, Waignier et al. focused only on the correctness of values transferred among different components, rather than timing guarantees.

IX. CONCLUSION

We have proposed a framework for the systematic I/O interface synthesis that contributes to the safe integration of the platform-independent code with the target platform. Automatically generated I/O interface code potentially contains fewer errors than manually written glue code, and it allows the communication semantics to be implemented in a traceable way. The I/O interfaces are correct-by-construction, as they are systematically generated from a concise specification of the communication semantics required by the model. The interface policy concept enables possible relaxations of the original model semantics to be presented and codified in a traceable manner. Our framework also provides well-defined interface APIs through which the platform-independent code and the platform-dependent code communicate, which eliminates possible erroneous interactions. We have also discussed how I/O interfaces can be used in the context of GPCA infusion pump systems. We plan to extend our work to synthesize the I/O interfaces for distributed code generation where a set of models are deployed on physically distributed systems.

ACKNOWLEDGEMENT

This research is supported in part by NSF grants CNS-0834524, CNS-1035715, CNS-1117185 and CNS-1135630.

REFERENCES