Hierarchical Multi-Formalism Proofs of Cyber-Physical Systems

Michael W. Whalen[†], Sanjai Rayadurgam[†], Elaheh Ghassabani[†],
Anitha Murugesan[†], Oleg Sokolsky[*], Mats P.E. Heimdahl[†], Insup Lee[*]
[†]Department of Computer Science and Engineering,University of Minnesota
{whalen, rsanjai, ghassaba, anitha, heimdhal}@cs.umn.edu
[*]Department of Computer and Information Science, University of Pennsylvania,
{sokolsky, lee}@cis.upenn.edu

***Abstract*—To manage design complexity and provide verification tractability, models of complex cyber-physical systems are typically hierarchically organized into multiple abstraction layers. High-level analysis explores interactions of the system with its physical environment, while embedded software is developed separately based on derived requirements. This separation of low-level and high-level analysis also gives hope to scalability, because we are able to use tools that are appropriate for each level. When attempting to perform compositional reasoning in such an environment, care must be taken to ensure that results from one tool can be used in another to avoid errors due to "mismatches" in the semantics of the underlying formalisms. This paper proposes a formal approach for linking high-level continuous time models and lower-level discrete time models. Specifically, we lift a discrete-time controller specified using synchronous observer properties into continuous time for proof using timed automata (UPPAAL). To define semantic compatibility between the models, we propose a direct semantics for a network of timed automata with a discrete-time component called *Contract-Extended Network of Timed Automata* (CENTA) and examine semantic issues involving timing and events with the combination. We then propose a translation of the discrete-time controller into a timed automata state machine and show the equivalence of the translation with the CENTA formulation. We demonstrate the usefulness of the approach by proving that a complex medical infusion pump controller is safe with respect to a continuous time clinical scenario.**

I. Introduction

Modern cyber-physical systems are extraordinarily complex, involving interactions between multiple distributed software controllers, sensors, actuators, and the physical environment. In order to cope with this complexity and understand how systems will work in their intended environment, system behavioral models are constructed and analyzed. These models are typically hierarchically organized into multiple abstraction layers: high-level models explore interactions of the system with its physical environment, while detailed models of embedded software are developed separately based on derived requirements. This separation of low-level and high-level models also gives hope to scalability, because we are able to use tools that are appropriate for each level.

This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715 and the DARPA HACMS program (contract FA8750-12-9-0179).

For example, consider next-generation medical systems, in which several medical devices communicate in order to provide optimal patient care. To model a clinical scenario, one must monitor the behavior of the patient (physical environment), a variety patient sensors and actuators, and control and coordination software. Researchers and developers often construct continuous time models of system interactions to perform verification (e.g., [1], [2]). While such models are very useful, the software controller models usually describe "toy" controllers that contain very little modal behavior and error handling found in realistic medical devices.

On the other hand, it is possible to construct and verify models of realistic software controllers using discrete time analysis tools. This is the reasoning approach used by commercial model-checking tools for SCADE [3] and Simulink [4], as well as code-level model checkers for C and Java ([5], [6]). To perform reasoning, the user usually defines an assertion (equivalently a *synchronous observer* [7] for SCADE or Simulink) in the same design notation as the modeled system. In this analysis, real time is abstracted away and the behavior over a number of discrete computational "steps" is examined.

At issue is the notion of time: At the system level, time is often best represented as a continuous quantity to accurately describe the inherently continuous, controlled physical environment. Timed Automata [8] are a standard approach for modeling and reasoning about these systems, and tools such as UPPAAL [9] allow scalable analysis of networks of timed automata. However, rewriting existing complex software controllers into timed automata is impractical; there is no straightforward mapping between implementation notations such as Simulink or C code to timed automata. Alternately, one could derive software controllers from UPPAAL controller models. Unfortunately, while tools such as UPPAAL are scalable, they are not nearly as scalable for complex software controllers as discrete time analysis tools.

What is necessary is a principled way to split the analysis so that we can prove the requirements of a complex discrete-time controller using scalable (and compositional) discrete time tools, then lift these requirements into continuous time to be the representation of the controller component when proving continuous time system-level properties.

In this paper, we describe an approach for lifting discrete time symbolic transition systems into timed automata. The discrete time systems are specified using *contracts* containing *assumptions* that the component expects of the environment and *guarantees* that the component will fulfill if the assumptions are true. The contract notation is called AGREE (Assume-Guarantee REasoning Environment) [10], and both the assumptions and guarantees are specified as synchronous observers. We have used this notation to reason about complex avionics and medical controllers.

To describe the composition, we first propose a direct semantics for a network of timed automata extended with a discrete-time contract called *Contract-Extended Network of Timed Automata* (CENTA) and examine semantic issues involving timing and events with such a combination. We then define a scheme to translate a discrete-time contract into a timed automata state machine and show that the translation is equivalent to the CENTA formulation.

Previously, we have explored semi-formal approaches for determining whether results in different notations could be composed [11]. In this paper, we both generalize the work from [11] and place it on a proper foundation. Thus, the contributions of this paper are:

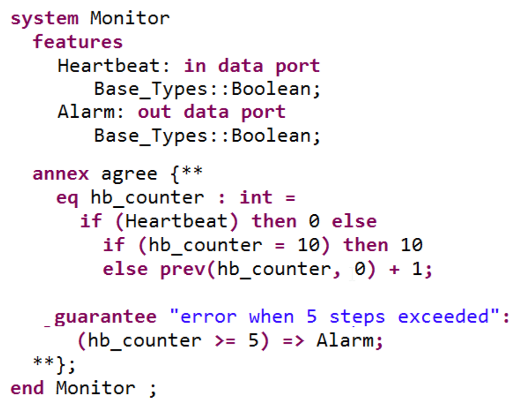
- an approach for embedding discrete-time contracts in timed automata,
- a provably correct translation of discrete time components into timed automata, and
- a demonstration of this approach's practical value in reasoning about case example involving a complex and clinically interesting medical device scenario.

The paper is organized as follows. Section II describes the motivation for this work using a simple medical CPS as an example. Section III provides the necessary formal background. Section IV presents the main contributions of this paper; we formalize the semantics of *Contract Extended Network of Timed Automata* (CENTA) which enables embedding a discrete-time contract in a continuous time model, describe a scheme to translate a contract directly into timed automata, and then sketch a proof to show that the resulting automata behaviour is equivalent to the CENTA formulation. Section VI briefly discusses the verification of a medical CPS clinical scenario example using translated timed automata obtained as defined by our approach. Section V discusses the limitations of our approach and provides an informal explanation for generalizing timing models. Section VII discusses the relevant related work and finally Section VIII concludes the paper.

II. Motivation and Simple Example

In previous work in both medical and avionics domains, we have analyzed complex software controllers implemented via periodic polling systems, where the software periodically reads inputs, calculates its current state and emits outputs. In these systems, real time can be abstracted into discrete time and it is therefore possible to demonstrate that software implementations meet their requirements for industrial-scale models [12], [13]. For these systems, requirements are often specified as *shall* statements that define expected behaviors under different combinations of system-state and input configurations. When analyzing the system, these requirements take the form of expressions over sequences of time steps, and can be formalized in, for example, LTL [14] or Synchronous Observers [7]. In order to scale discrete-time analysis, Rockwell Collins and the University of Minnesota have developed the AGREE (Assume Guarantee Reasoning Environment) tool suite [15] that allows reasoning about software architectures specified in AADL [16]. This tool suite allows us to compositionally reason about *assume-guarantee contracts*, composing results from leaf-level components within the software architecture to prove properties of the software as a whole. Figure 1 shows a simple example of an AGREE contract specification.

While AGREE enables reasoning about the software controller, we would like to know that the software interacting with its physical environment will achieve the intended effect – i.e., we would like to examine *system-level* properties rather than *software-level* properties. For this, we need to account for the continuous nature of the physical environment, either by direct representation of time as a continuous quantity (such as Timed Automata [8]) or through a sound over-approximation of continuous behaviors in discrete time — ultimately, we want to ascertain whether the software, as specified by its requirements, is adequate to control the physical system as desired. What we propose in this paper is a (formally justified) mechanism for lifting a set of discrete-time requirements in the controller domain into a Timed Automata representation in the system domain, to perform such system-level analysis.



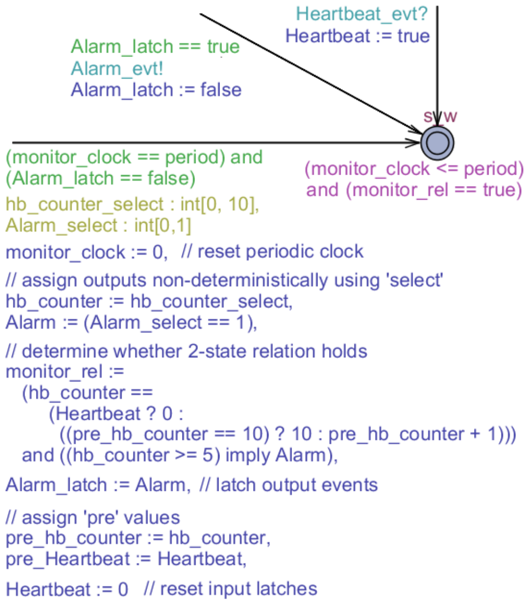
```
system Monitor
  features
    Heartbeat: in data port
       Base_Types::Boolean;
    Alarm: out data port
       Base_Types::Boolean;

  annex agree {**
    eq hb_counter : int =
      if (Heartbeat) then 0 else
        if (hb_counter = 10) then 10
        else prev(hb_counter, 0) + 1;

     guarantee "error when 5 steps exceeded":
       (hb_counter >= 5) => Alarm;
  **};
end Monitor ;
```

Fig. 1. Toy monitor contract in AGREE

We present a toy example in order to motivate the approach and explain it informally.[1] Consider a hypothetical monitoring device for a process control system that—among other functions—periodically checks to see whether it has received a "heartbeat" signal from another (monitored) device. Figure 1 shows the model of this controller in AGREE. One of the properties of the system is as follows: *If the monitor does not receive a heartbeat signal within 0.5 seconds, the monitor shall signal an alarm until the heartbeat signal is received.* Assuming that the monitor runs at $10Hz$, this translates into a simple discrete time property that references a heartbeat counter; the counter resets when a heartbeat is received. If the counter reaches the value 5 ($0.5s = 5 * 100ms$, where $100ms$ is the sample rate), an alarm will be raised. Note that we are

[1]The formal semantics of timed automata and AGREE will be explained in Section III.

not *assigning* the `Alarm` variable, we are simply checking whether it behaves as intended.



```
(monitor_clock == period) and
(Alarm_latch == false)
hb_counter_select : int[0, 10],
Alarm_select : int[0,1]
monitor_clock := 0,  // reset periodic clock
// assign outputs non-deterministically using 'select'
hb_counter := hb_counter_select,
Alarm := (Alarm_select == 1),
// determine whether 2-state relation holds
monitor_rel :=
  (hb_counter ==
     (Heartbeat ? 0 :
       ((pre_hb_counter == 10) ? 10 : pre_hb_counter + 1)))
  and ((hb_counter >= 5) imply Alarm),
Alarm_latch := Alarm,  // latch output events
// assign 'pre' values
pre_hb_counter := hb_counter,
pre_Heartbeat := Heartbeat,
Heartbeat := 0   // reset input latches
```

Fig. 2. Toy monitor contract in UPPAAL

We can represent the same contract in UPPAAL as shown in Figure 2. Informally, in order to represent the discrete time contract in UPPAAL, we must (1) define a period for the contract and a clock to cause it to periodically evaluate, (2) latch UPPAAL input events as they occur into corresponding Boolean variables in the contract, (3) "execute" the contract by non-deterministically assigning outputs in such a way that the contract is satisfied, and (4) emit output events for each Boolean variable in the contract corresponding to an event. In the translated UPPAAL machine, we use *select* statements to make the non-deterministic assignments and make the contract relation *monitor_rel* part of the state invariant for the UPPAAL state machine. This invariant ensures that the non-deterministic assignment to outputs satisfies the contract. Intuitively, this timed automaton represents a component that upholds the requirement (raise alarm if no heartbeat for 0.5 sec), in much the same way as the AGREE specification. In the sequel, this notion of equivalence will be formalized.

III. Background

In this section, we provide a formal overview of two topics that merit coverage as background for the work presented in this paper - Timed Automata and AGREE Automata.

A. Timed Automata with Data Variables

A timed automaton [17] is a finite-state machine extended with a notion of dense-time modeled as a finite collection of synchronously progressing real-valued *clock* variables. Transitions between states (called *locations*), may be constrained using simple conditions over the clock variables. Often, the automaton is also extended with bounded discrete-valued data variables, which can also be used to constrain transitions between locations. Clock variables may be reset and data variables may be assigned values on transitions (called *firing of edges*). A system can be modeled as a network of timed automata (NTA) that may synchronize transitions through rendezvous channels and communicate through data variables. The state of the system modeled by such a network of timed automata is defined by the locations, the clock constraints and the values of the data variables. We first provide a formal definition of a network of timed automata with data variables.

Notation. Let C denote the set of all clocks. A clock valuation is a function $u : C \to \mathbb{R}^+$; we use $\mathbb{R}^C$ to denote the set of all clock valuations. A simple clock valuation is the function $u_0(x) = 0$, for all $x \in C$. For clock valuation u, $u+d$ denotes the valuation where $(u + d)(x) = u(x) + d$, for $x \in C$. Furthermore, let $B(C)$ denote the set of conjunctions over simple clock conditions of the form $x \bowtie n$ or $x - y \bowtie n$, where $n \in \mathbb{N}_0$, $x, y \in C$, and $\bowtie \in \{\leq, \geq, =, <, >\}$. We use K to denote the set of all channels and $A = \{\alpha? \mid \alpha \in K\} \cup \{\alpha! \mid \alpha \in K\} \cup \{\tau\}$ the set of all actions. Let V denote the set of variables, T denote the set of types, Val_t be the set of values for type t, $Val = \bigcup\{Val_t \mid t \in T\}$ denote the universe of values for variables, and Φ be the set of expressions over variables:

$$\Phi = \{\phi \mid \phi ::= c \mid v \mid \text{ ite } \phi \; \phi \; \phi \mid \text{ uop } \phi \mid \phi \text{ bop } \phi\}$$

where c and v denote constants and variables, respectively; ite is an if/then/else expression; uop $\in \{\neg, -\}$ and bop $\in \{+, -, \times, /, =, \neq, <, \leq, >, \geq, \wedge, \vee, \implies\}$ define unary and binary operators with the usual semantics. A function $\nu : V \to Val$ maps variables to values; we use Val^V to denote the set of all variable valuations. A function $ty : V \to T$ maps variables to types. We assume that models are well-typed.

Definition 1 (Timed Automaton [17]). *An automaton $\mathcal{A}$ is a tuple (L, l_0, A, C, V, E, I), where L denotes the set of locations in the timed automaton, l_0 is the initial location, A is a set of actions, C a set of clocks, V the set of variables, and $E \subseteq L \times A \times G \times Y \times 2^C \times L$ denotes the set of edges (between locations, with an action $a \in A$, a guard $g = (\phi_c, \phi_d) \in G : B(C) \times \Phi$, a sequence of variable updates $y = (v, \phi) \in Y : V \times \Phi$, and a subset of clocks to be reset), while $I : L \to B(C) \times \Phi$ assigns invariants to locations.*

For notational convenience, we write $l_1 \xrightarrow{a,g,y,r} l_2$ whenever $(l_1, a, g, y, r, l_2) \in E$.

A network of n timed automata is obtained by composing automata $\mathcal{A}_i = (L_i, l_i^0, A, C, V, E_i, I_i)$, $i \in \{1, ..., n\}$. In this case, a location vector is defined as $\bar{l} = (l_1, l_2, ..., l_n)$. In addition, an invariant for location vector $\bar{l}$ is defined as $I(\bar{l}) = \wedge_i I_i(l_i)$. To denote the vector where i^{th} element of vector $\bar{l}$ (i.e., l_i) is substituted with l_i' we use notation $\bar{l}[l_i'/l_i]$. If clock valuation u and variable valuation ν satisfy the invariant at location l, we abuse the notation and write $(u, \nu) \in I(l)$. Similarly, if valuation (u, ν) satisfies condition $g = (\phi_c, \phi_d)$ where $g \in (B(C), \Phi)$, and satisfaction means u satisfies ϕ_c and ν satisfies ϕ_d, we write $(u, \nu) \in g$. $r(u)$ denotes the clock valuation obtained from u when all clocks from the set $r \subseteq C$ are reset to zero, and $y(\nu)$ denotes the variable valuation obtained from ν from sequentially evaluating the assignment list from head to tail in y. $\text{dom}(y)$ defines the set of variables to be assigned in the sequence y.

Definition 2 (Semantics of NTA [17]). *Let $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2..., \mathcal{A}_n\}$ be a network of n timed automata, and let $\bar{l}^0 = (l_1^0, l_2^0, ..., l_n^0)$ be the initial location vector, and v_0 be the initial variable state. The semantics is defined as a transition system $\langle \mathcal{S}, s_0, \rightarrow \rangle$, where $\mathcal{S} = (L_1 \times L_2 \times ... \times L_n) \times \mathbb{R}^C \times Val^V$ is the set of states, $s_0 = (\bar{l}_0, u_0, \nu_0)$ is the initial state, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation defined by:*

1) $(\bar{l}, u, \nu) \rightarrow (\bar{l}, u + d, \nu)$ *if* $\forall d', 0 \le d' \le d \Rightarrow (u + d', \nu) \in I(\bar{l})$.
2) $(\bar{l}, u, \nu) \rightarrow (\bar{l}[l_i'/l_i], u', \nu')$ *if there exists* $l_i \xrightarrow{\tau, g, y, r} l_i'$ *s.t.* $(u, \nu) \in g$, $u' = r(u)$, $\nu' = y(\nu)$ *and* $(u', \nu') \in I(\bar{l}[l_i'/l_i])$.
3) $(\bar{l}, u, \nu) \rightarrow (\bar{l}[l_j'/l_j, l_i'/l_i], u', \nu')$ *if there exists* $l_i \xrightarrow{c?, g_i, y_i, r_i} l_i'$ *and* $l_j \xrightarrow{c!, g_j, y_j, r_j} l_j'$, *s.t.* $i \neq j$, $(u, \nu) \in g_i$, $(u, \nu) \in g_j$, $(y_j(y_i(\nu))) = (y_i(y_j(\nu)))$, $u' = (r_i \cup r_j)(u)$, $\nu' = (y_j(y_i(\nu)))$, *and* $(u', \nu') \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

Given a transition system $\langle \mathcal{S}, s_0, \rightarrow \rangle$, a sequence $\mathcal{R} := (\bar{l}_0, u_0, \nu_0) \rightarrow (\bar{l}_1, u_1, \nu_1) \rightarrow ... \rightarrow (\bar{l}_i, u_i, \nu_i) \rightarrow ...$, is called a *run*. The set of all runs for a transition system TS is designated $\mathcal{R}^{TS}$.

B. AGREE Automata

The Assume Guarantee Reasoning Environment (AGREE) [10] is a discrete compositional reasoning framework based on the notion of assume-guarantee contracts [10], where guarantees correspond to component requirements, and assumptions correspond to the environmental constraints that are used in verifying the component requirements. A contract (assume-guarantee pair) specifies precisely the information that is needed to reason about the component's interaction with other parts of the system.

An AGREE contract Λ is a 4-tuple (V_A, V_{A0}, A_A, P_A) containing the variables, initial variable valuation, assumption, and promise (guarantee) of the contract, respectively. Variables are partitioned into input and output variables: $V_A = I \cup O$ that are disjoint sets: $(I \cap O) = \emptyset$. Each variable has an initial value that is assigned by $V_{A0} : V_A \rightarrow Val$. After removing several layers of syntactic sugar, assumptions and promises are defined by Boolean expressions drawn from a slight extension to the expression grammar in the previous section:

$$\Phi' = \{\phi' \mid \phi' ::= c \mid v \mid \text{ ite } \phi' \ \phi' \ \phi' \mid \text{ uop } \phi' \mid \phi' \text{ bop } \phi' \mid \text{pre}(v)\}$$

Pre-expressions allow state to be introduced; in the initial execution step, $\text{pre}(v)$ has the value $V_{A0}(v)$ and thereafter it has the previous value of variable v. When writing concrete expressions, we write them within double braces: $[\![a + b]\!]$ to distinguish the expression language from the metalanguage.

As with timed automata, a *variable valuation* ν_A is a mapping from $V_A \rightarrow Val$, and a *trace* is a mapping from time instants to valuations: $\sigma : \mathbb{N} \rightarrow \nu_A$. The contract is *satisfied* for a trace σ if the following past time LTL (PLTL) property is true of the assumptions and guarantees:

$$\sigma \vdash G(H(A_A) \implies P_A);$$

The past time *Historically* operator $H(\alpha)$ states that α has been true from the initial instant up until the current instant in the trace. In order to construct a two-state relation from the contract, we add fresh output variable v_{histA} which is initialized to true in V_{A0}. The contract formula is already a two-state invariant other than the "historically" operator, which we can define by $[\![v_{histA} = A_A \wedge \text{pre}(v_{histA})]\!]$. The two-state expression defining contract satisfaction $C2S$ then becomes:

$$[\![(v_{histA} = (A_A \ \wedge \ \text{pre}(v_{histA}))) \ \wedge \ (v_{histA} \implies P_A)]\!]$$

A trace σ satisfies a contract when $\forall i \in \mathbb{N}$, $C2S(\sigma_i, \sigma_{i+1})$. The set of all traces satisfying the contract is Σ_A.

IV. Embedding AGREE in UPPAAL

This section describes how we map a discrete-time model (AGREE) to a continuous-time model (timed automata). Although we use timed automata as the target semantics, similar translations could be performed to richer notations, such as hybrid automata. Section IV-A first presents a direct formalization of an AGREE contract into a network of timed automata (NTA). The formalization can be considered as a way of embedding AGREE into UPPAAL, where the discrete controller can be viewed as a state machine in UPPAAL that runs at a fixed rate, latches all inputs prior to beginning computation, computes its next state, and then emits outputs. Then, it is quiescent until its period elapses. Also, transitions into a "new state" can be represented by the valuations of the variables in the AGREE model. A direct semantics provides the most direct way to specify the properties that should hold of the combined model.

Nevertheless, as there is no tool that can analyze the embedding semantics, we provide a translation from AGREE contracts directly into a timed automaton. This allows analysis of the combined model. Section IV-B describes a translation of an AGREE contract into a timed automata, whereby a pure NTA is obtained and can be verified using existing tools like UPPAAL. This AGREE contract represents a set of "software level" properties that are true of a complex controller. In Section IV-C, we prove that two approaches are equivalent, and that the translation matches the embedding semantics.

The primary complexity in translation comes from differences in time representation and synchronization. There are several possible semantic choices related to the *responsiveness* of the environment to output events generated by the AGREE component: when should synchronization be enforced. In the presented semantics (and translation) the environment must respond to output events (via synchronization) before the AGREE component can execute its next cycle. UPPAAL uses events to synchronize parallel state machines, while AGREE uses only variables. Much of the translation involves mediating this boundary.

A. A Direct Semantics of AGREE Contracts in UPPAAL

In order to create a direct semantics of AGREE contracts in timed automata, we introduce some notation. We must define the period of the AGREE model, κ. To be compatible with UPPAAL, κ must be a positive integer; it is assumed to be scaled appropriately for the model. We split the variables in the AGREE contract $V_A \subseteq V$ into input and output *event* and *data* variables $V_A = I_E \cup I_D \cup O_E \cup O_D$, where each of I_E, I_D, O_E, O_D is pairwise disjoint, $I = I_E \cup I_D$ and $O = O_E \cup O_D$. The *event* variables are used to record input or output events from/to the timed automata network and the *data* variables are used for communicating data between the AGREE model and in the network. We assume user-provided partial one-to-one functions $\iota : K \to I_E$ and $\rho : K \to O_E$ to map the timed automata events to variables in AGREE.

The state of a contract embedded in a timed automata is a pair $\Psi : Val^V \times 2^A$, written $\psi = (\nu_p, o)$. The ν_p element is the previous state valuations of variables and o is the set of output events generated by the contract at the current instant. To express non-deterministic assignment of outputs, we define the set of all possible AGREE output valuations given an existing valuation ν as follows: $Val^O(\nu) = \{ \nu' \mid (x \notin O \implies \nu'(x) = \nu(x)) \wedge (x \in O \implies \nu'(x) \in ty(x))\}$. We also define the set of outputs corresponding to 'true' output variables: $\theta(\nu) = \{ c! \mid x \in \operatorname{dom} \rho \wedge \rho(x) = c \wedge \nu(x) = \text{true}\}$. Finally, we introduce notation for functions: the notation $x \mapsto y$ defines a function with a single element, and $\oplus$ defines function override: $F \oplus G = G \cup \{(x, y) \mid (x, y) \in F \wedge x \notin \operatorname{dom} G\}$.

Given the definitions above, a contract-extended network of n timed automata (CENTA) is obtained by composing a network of automata where $\mathcal{A}_i = (L_i, l_i^0, A, C, V, E_i, I_i)$, $i \in \{1, ..., n\}$ with a timed AGREE contract (Λ, κ). All definitions and substitutions over automata are defined as in Section III-A. We assume the existence of a clock $c_\kappa \in C$ that is not assigned or referenced by the automata. The semantics of the CENTA $\mathcal{A}_C(\mathcal{A}, \nu_0, \Lambda)$ are as follows:

Definition 3 (CENTA). *Let*

- $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2 ..., \mathcal{A}_n\}$ *be a network of* n *timed automata,*
- $\Lambda = (V_A, V_{A0}, A, P)$ *be an AGREE contract,*
- $\bar{l}^0 = (l_1^0, l_2^0, ..., l_n^0)$ *be the initial location vector.*
- $\nu_0 = \nu_a \cup V_{A0}$ *be the initial variable valuation. An initial state is* valid *if* ν_0 *is a function.*

The semantics is defined as a transition system $\langle \mathcal{S}_C, s_0, \to \rangle$*, where* $\mathcal{S}_C = (L_1 \times L_2 \times ... \times L_n) \times \mathbb{R}^C \times Val^V \times \Psi$ *is the set of states,* $s_0 = (\bar{l}_0, u_0, \nu_0, (\nu_0, \emptyset))$ *is the initial state, and* $\to \subseteq \mathcal{S}_C \times \mathcal{S}_C$ *is the transition relation defined by:*

1) $(\bar{l}, u, \nu, \psi) \to (\bar{l}, u + d, \nu, \psi)$ *if* $\forall d', 0 \leq d' \leq d \Rightarrow (u + d', \nu) \in I(\bar{l})$ *and* $u(c_\kappa) + d \leq \kappa$.
2) $(\bar{l}, u, \nu, \psi) \to (\bar{l}[l_i'/l_i], u', \nu', \psi)$ *if there exists* $l_i \xrightarrow{\tau, g, y, r} l_i'$ *s.t.* $(u, \nu) \in g$, $u' = r(u)$, $\nu' = y(\nu)$ *and* $(u', \nu') \in I(\bar{l}[l_i'/l_i])$.
3) $(\bar{l}, u, \nu, \psi) \to (\bar{l}[l_j'/l_j, l_i'/l_i], u', \nu', \psi)$ *if there exists* $l_i \xrightarrow{c?, g_i, y_i, r_i} l_i'$ *and* $l_j \xrightarrow{c!, g_j, y_j, r_j} l_j'$*, s.t.* $i \neq j$, $(u, \nu) \in g_i$, $(u, \nu) \in g_j$, $(y_j(y_i(\nu))) = (y_i(y_j(\nu)))$, $u' = (r_i \cup r_j)(u)$, $\nu' = (y_j(y_i(\nu)))$, *and* $(u', \nu') \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.
4) $(\bar{l}, u, \nu, \psi) \to (\bar{l}[l_i'/l_i], u', \nu'', \psi)$ *if there exists* $l_i \xrightarrow{c!, g, y, r} l_i'$ *s.t.* $(u, \nu) \in g$, $u' = r(u)$, $\nu' = y(\nu)$, $c \in \operatorname{dom}(\iota)$, $y(\nu) \oplus (v_{ie} \mapsto \text{true}) = y(\nu \oplus (v_{ie} \mapsto \text{true}))$, $\iota(c) = v_{ie}$, $\nu'' = \nu' \oplus (v_{ie} \mapsto \text{true})$, *and* $(u', \nu'') \in I(\bar{l}[l_i'/l_i])$.
5) $(\bar{l}, u, \nu, (\nu_p, \emptyset)) \to (\bar{l}, u', \nu'', (\nu', o'))$ *if* $\nu' \in Val^O(\nu)$, $u(c_\kappa) = \kappa$, $C2S(\nu_p, \nu')$, $\nu'' = \nu' \oplus \{(v, false) \mid v \in V_{IE}\}$, $u' = u \oplus (c_\kappa \mapsto 0)$, $o' = \theta(\nu')$, *and* $(u', \nu'') \in I(\bar{l})$.
6) $(\bar{l}, u, \nu, (\nu_p, o)) \to (\bar{l}[l_i'/l_i], u', \nu', (\nu_p, o'))$ *if there exists* $l_i \xrightarrow{c?, g, y, r} l_i'$ *s.t.* $(u, \nu) \in g$, $u' = r(u)$, $\nu' = y(\nu)$, $c! \in o$, $o' = o - \{c!\}$, *and* $(u', \nu'') \in I(\bar{l}[l_i'/l_i])$.

The first three rules are nearly unchanged from the Definition 1. The only change is that these rules have an additional state component ψ for the AGREE contract state. Rules (4),(5), and (6) define the *input, compute*, and *output* rules for the contract. Rule 4 latches an input event for the contract into a variable in I_E. Rule 5 non-deterministically assigns the output variables from the AGREE component such that the resulting state satisfies the AGREE two-state relation, resets the clock c_κ, creates a set of pending output events that will be emitted by the component based on the assignments to O_E variables, and resets the input event latches. This rule can only fire when the period has elapsed and there are no pending outputs (all outputs from the previous step have been consumed). Rule 6 emits any one of a set of pending output events from the AGREE contract.

Given a transition system $TS_C = \langle \mathcal{S}_C, s_0, \to \rangle$, a list $s := (\bar{l}_0, u_0, \nu_0, \psi_0) \to (\bar{l}_1, u_1, \nu_1, \psi_1) \to \ldots \to (\bar{l}_i, u_i, \nu_i, \psi_i) \to \ldots$, is called a *run*. We refer to states in the run using subscripts: s_0, s_1. The set of all runs for a transition system TS_C is designated $\mathcal{R}^{TS_C}$.

1) Properties of Extended Model: There are some desirable properties that are straightforward to establish to show that the embedding of the AGREE contract is consistent with its discrete-time semantics.

To do so, we define notation for lists. Lists can be formed using the cons operator ::, or defined using bracket notation: $[a, b, c]$ is the list $a :: b :: c :: nil$. List concatenation is notated $l_0 \frown l_1$. Lists can be constructed from other lists using list comprehensions: $[f(x) \mid x \text{ in } L]$, which map the elements of the list x to a value $f(x)$. We can also define lists through filter comprehensions $[f(x) \mid x \text{ in } L \text{ and } P(x)]$ in which only elements satisfying $P(x)$ are stored in the resulting list. Finally, sets can be converted to lists using a list-choice operator Υ which provides an (arbitrary) ordering of the set elements. We also introduce domain restriction of functions: $F \upharpoonright D : \{(x, y) \mid (x, y) \in F \wedge x \in D\}$.

To describe equivalence, we extend the idea of a run to store the rule used to generate the next state: $\zeta : (r_0, s_0) \to (r_1, s_1) \to \ldots$, where the rule r is 0 in the first step and thereafter 1..6 depending on the rule used. The set of all extended runs of transition system TS_C is designated $\mathcal{RE}^{TS_C}$.

Given an extended run ζ, we can define a function *compress* that extracts a discrete time AGREE trace from its embedding in UPPAAL as follows:

$$compress(\zeta) = [(\nu \upharpoonright V_A) \mid (r, (\bar{l}, u, \nu, \psi)) \text{ in } \zeta \text{ and } r = 5]$$

and we can define the set of compressed runs Σ^{TS_C} as $\Sigma^{TS_C} = \{compress(\zeta) \mid \zeta \in \mathcal{RE}^{TS_C}\}$.

We can now describe a handful of properties about the model:

- **Timeliness:** By construction, the AGREE model must update at the moment of its period κ. Rule (1) will not allow time to advance if the AGREE period clock c_κ exceeds its period. In addition, Rule (5), which evaluates the AGREE contract, can only occur when $c_\kappa = \kappa$. Finally, by assumption, no other automata can reset or reference clock c_κ.
- **Soundness:** The set of compressed runs Σ^{TS_C} is a subset of the contract-satisfying traces: $\Sigma^{TS_C} \subseteq \Sigma_A$. We first note that, for any extended trace, the compressed initial state matches the initial state of the AGREE contract. Subsequently, by construction, the states in the compressed trace are produced by rule (5). Rule 5 requires that the pre- and post-state must satisfy the same predicate $C2S$ that is used to determine acceptability of pre- and post-state for traces for AGREE. Therefore any valuation produced by rule 5 is acceptable for Σ_A. Note that soundness does not depend on other processes modifying outputs of the machine (these are latched into the AGREE state for the pre-state, and for the post-state, they will be recomputed).
- **Relative Completeness:** It is *not* usually the case that the CENTA embedding of the AGREE contract exhibits all behaviors of Σ_A, because the embedding restricts the valuation of contract inputs. In addition, some traces of the CENTA are *finite*, because of possible deadlocks. If the CENTA is deadlock-free and non-Zeno, then we can talk of *relative completeness*, in which we restrict the traces of Σ_A to only those that share an input sequence with a trace in Σ^{TS_C}. (call it Σ^I_A). For this set, $\Sigma^I_A = \Sigma^{TS_C}$, by a similar argument as for soundness; Rule (5) allows any output value that satisfies $C2S$, and deadlock freedom and non-Zeno restrictions ensure that the guard for rule (5) is eventually satisfied.
- **Assumption Consistency:** When embedding AGREE contracts within an environment, we want to ensure that the *assumptions* of the contract are true whenever the contract is evaluated. Otherwise, the AGREE contract can produce any output value and is essentially meaningless. This can be checked: $\forall \sigma \in \Sigma^{TS_C} . \forall i \in \mathbb{N} . A_A(\sigma_i, \sigma_{i+1})$. Operationally, once the assumption is translated into UPPAAL (call it A^*), this property can be checked using the invariant formula A[] (A^*). The assumption translation is described in the following section.

For formal arguments, see [18].

B. Translation of AGREE Contracts into Timed Automata

To perform analysis, we want to translate AGREE contracts directly into timed automata. This can be accomplished schematically as defined in Figure 3. In Figure 3, given AGREE contract (Λ, κ), we define the components of automaton $A_a = (L, l^0, A, C, V, E, I)$ based on the structure of the AGREE contract. Our model has only one state: l_w: the 'wait' state in which the machine waits for the AGREE period to elapse. The actions performed are A_{in} and A_{out}, which are the actions corresponding to the input and output event variables, respectively. The variables V consist of the variables in the original AGREE contract (V_A, including the v_{histA} used in the definition of the $C2S$ relation), the set of previous values of these variables V_P, the 'latched output' variables V_{OL}, which we use for tracking pending output events, and a variable v_{sat}, which, we will see, will be equated to the satisfaction implication $v_{histA} \implies v_P$. All of the sets of variables are assumed disjoint. To construct the sets V_P and V_OL, we create fresh variables in correspondence with the original variables: for example, the notation $\{v_{ol} \mid v_o \in (\text{range } \rho)\}$ means that for each variable $v_o \in (\text{range } \rho)$ we construct a corresponding fresh variable v_{ol}.

The edge set E contains edges for input assignments E_I, output assignments E_O, and AGREE contract next-state computation E_T. The computation E_T is mapped to a set of transitions, one for each valuation of the set of variable valuations for variables in V_O. Each of these transitions in E_T perform a sequence of assignments Y_T. In Y_T, we first assign values to the 'current' output variables in Y_{TA}. Next,

$$\begin{aligned}
\mathcal{A}_a &= (L, l_0, A, C, V, E, I)\\
L &= \{l_w\}\\
l_0 &= l_w\\
A &= A_{in} \cup A_{out}\\
C &= \{c_\kappa\}\\
V &= V_A \cup V_P \ \cup V_{OL} \cup \{v_{sat}\}\\
E &= E_I \cup E_T \cup E_O\\
I &= \{(l_w, (\llbracket c_\kappa \leq \kappa \rrbracket, \llbracket v_{sat} = \text{true} \rrbracket))\}
\end{aligned}$$

where:

$$\begin{aligned}
&A_{in} = \{\alpha? \mid x \in I_E \wedge (\alpha, x) \in \iota\}\\
&A_{out} = \{\alpha! \mid x \in O_E \wedge (\alpha, x) \in \rho\}\\
&V_P = \{v_{pre} \mid v \in V_A\}\\
&V_{OL} = \{v_{ol} \mid v_o \in (\text{range } \ \rho)\}\\
&C2S^* = \llbracket (v_{histA} = (A^*_A \wedge v_{pre\_histA})) \wedge (v_{histA} \implies P^*_A) \rrbracket\\
&Y_{TP} = [(v_{pre}, \llbracket v \rrbracket) \mid v \text{ in } \Upsilon(V_A)]\\
&Y_{TA} = \{[(v_0, c_0), (v_1, c_1), \ldots, (v_k, c_k)] \mid v_i \in O \ \wedge\\
&\qquad c_0 \in ty(v_0) \ \wedge c_1 \in ty(v_1) \ \wedge \ldots \wedge c_k \in ty(v_k)\}\\
&Y_{TS} = [(v_{sat}, C2S^*)]\\
&Y_{TO} = [(v_{ol}, \llbracket v_o \rrbracket) \mid v_o \text{ in } \Upsilon(\text{range } \ \rho)]\\
&Y_{TI} = [(v, \llbracket \text{false} \rrbracket) \mid v \ \in V_{IE}]\\
&Y_T = \{y_{TA} \frown Y_{TS} \frown Y_{TO} \frown Y_{TP} \frown Y_{TI} \mid y_{TA} \in Y_{TA}\}\\
&E_I = \{(l_w, \alpha_i?, (\llbracket \text{true} \rrbracket, \llbracket \text{true} \rrbracket), \{(v_i, \llbracket \text{true} \rrbracket)\}, \emptyset, l_w) \mid\\
&\qquad (\alpha_i, v_i) \in \iota\}\\
&E_T = \{(l_w, \tau, (\llbracket c_\kappa = \kappa \rrbracket, \bigwedge \{\llbracket v_{ol} = \text{false} \rrbracket \mid v_o \in \text{range } \ \rho\}),\\
&\qquad y, \{c_\kappa\}, l_w) \mid y \in Y_T\}\\
&E_O = \{(l_w, \alpha_o!, (\llbracket true \rrbracket, \llbracket v_{ol} \rrbracket), \{(v_{ol}, \llbracket \text{false} \rrbracket)\}, \emptyset, l_w) \mid\\
&\qquad (\alpha_o, v_o) \in \rho\}
\end{aligned}$$

Fig. 3. Translation Rules for AGREE Contracts

we assign the variables associated with the transition system in Y_{TS}. Finally, we latch the 'current' value of variables into the 'pre' variables (Y_{TP}) for use in the next computation. We define A^*_A and P^*_A to be the result of replacing every expression of the form $\mathrm{pre}(v)$ with v_{pre} in A_A and P_A, respectively. We can take any transition from E_T (i.e., any valuation of variables in V_A) that satisfies v_{sat} (and therefore the invariant for state l_w).

The initial variable state for the automaton is as follows:

$$\nu_{a0} = \{(v, V_{A0}(v)) \mid v \in V_A\} \;\cup\; \{(v_{pre}, V_{A0}(v)) \mid v \in V_A\} \\ \cup \{(v_{sat}, true)\} \;\cup\; \{(v, false) \mid v \in V_{OL}\}$$

Given a network of n timed automata with initial variable state ν_0: $(\mathcal{A}, \nu_0)$, where $\mathcal{A}_i = (L_i, l_i^0, A, C, V, E_i, I_i)$, $i \in \{1, ..., n\}$ and the translation $(\mathcal{A}_a, \nu_{a0})$ of an AGREE contract (Λ, κ), the result is a network of $n+1$ timed automata with initial state ν_{t0}: $(\mathcal{A}_T, \nu_{t0})$, where the $n+1th$ automata is A_a, and $\nu_{t0} = \nu_0 \cup \nu_{a0}$.

In order to be a correct embedding, ν_{t0} must be a function, and we make the restriction that only the AGREE automaton can assign variables in the set of output events O_E and pre-variables V_P. We assume the existence of a predicate $assigns : \mathcal{A} \to 2^V$ that given an automaton $\mathcal{A}_i$ returns the set of variables that are assigned by the edges (transitions) of the automaton. Given the definition of $assigns$, the restriction on output assignment is designated by $assigns\_ok$ as follows:

$$assigns\_ok(\mathcal{A}) \equiv (\forall \mathcal{A}_i \in \mathcal{A} \,.\, ((A_i \neq A_a) \implies \\ (assigns(A_i) \;\cap\; (O_E \cup V_P \cup V_{OL} \cup \{v_{sat}\}) = \emptyset)))$$

C. Proof of Equivalence

To prove equivalence, we define state equivalence $\equiv_s$ between a CENTA state $s_C = (\bar{l}_c, u_c, \nu_c, (\nu_{pc}, o_c))$ and a translated state $s_T = (\bar{l}_t, u_t, \nu_t)$ as follows:

$$\begin{array}{ll} & (\bar{l}_c,\ u_c, \nu_c, (\nu_{pc}, o_c)) \ \equiv_s \ (\bar{l}_t, u_t, \nu_t) \ \text{ iff:} \\ (a.) & \bar{l}_c = \bar{l}_{t-a} \ \wedge \\ (b.) & u_c = u_t \ \wedge \\ (c.) & (\forall x \in \mathrm{dom}\ \nu_c \,.\, \nu_c(x) = \nu_t(x)) \ \wedge \\ (d.) & (\forall x \in V_A \,.\, \nu_{pc}(x) = \nu_t(x_{pre})) \ \wedge \\ (e.) & (\forall (\alpha_o, v_o) \in \rho \,.\, (\alpha_o! \in o_c \iff \nu_t(v_{ol}) = true)) \ \wedge \\ (f.) & \nu_t(v_{sat}) = true \end{array}$$

where $\bar{l}_{t-a}$ is the vector of locations of l_t without l_a: $l_t = (l_1, \ldots, l_k, l_a) \iff l_{t-a} = (l_1, \ldots, l_k)$.

One can immediately lift state equivalence to trace equivalence $\equiv_\sigma$ by requiring state equivalence for all states in an infinite trace. We also define prefix equivalence $\equiv_k$ as state equivalence for the first k states of a path: $(\sigma_c \equiv_k \sigma_t) \iff (\forall x : 0 \leq x \leq k \,.\, \sigma_{cx} \equiv_s \sigma_{tx})$.

Theorem 1 (Trace Equivalence). *Given*

- *an AGREE contract* Λ,
- *an original network of timed automata with initial variable state* ν_0: *TA* $= (\mathcal{A}, \nu_0)$ *where* $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$
- *a contract-extended network of timed automata* $TA_C = (\mathcal{A}, \nu_{c0}, \Lambda)$, *with* ν_{c0} *a valid initial state, and associated transition system* TS_C.
- *From* Λ, *a translated automata* $\mathcal{A}_a$ *with initial variable state* ν_{a0}.
- *a network of timed automata with initial variable state* ν_{t0}: $TA_T = (\mathcal{A}_T, \nu_{t0})$, *where* $\mathcal{A}_T = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n, \mathcal{A}_a\}$, $\nu_{t0} = \nu_{a0} \cup \nu_0$, *with* ν_{t0} *a function, and* $assigns\_ok(\mathcal{A}_T)$, *yielding an associated transition system* TS_T.

Then (1) for every trace t_c *in* $\mathcal{R}^{TS_C}$ *there exists a trace* t_t *in* $\mathcal{R}^{TS_T}$ *such that* $t_c \equiv_\sigma t_t$, *and (2) for every trace* t_t *in* $\mathcal{R}^{TS_T}$, *there exists a trace* t_c *in* $\mathcal{R}^{TS_C}$ *such that* $t_c \equiv_\sigma t_t$.

Proof: (1) by construction. Given an arbitrary trace σ_c we construct an equivalent trace σ_t. We construct σ_t using induction over the natural numbers, assuming that we have constructed $\sigma_{t1} \ldots \sigma_{tk}$ and then extending the trace to σ_{tk+1}.

The proof decomposes into two cases. First, we must show that the initial states match (base case). By the initial state construction $l_c = l_{t-a} = (l_1^0, l_2^0, ..., l_n^0)$ and $u_c = u_t = u_0$. By construction, the domain of ν_c consists of $\mathrm{dom}(V_A) \cup \mathrm{dom}(v_0)$, and for $x \in \nu_c \,.\, \nu_c(x) = \nu_t(x) = \nu_0(x)$, and for $x \in V_A$, we have $\nu_c(x) = \nu_t(x) = \nu_{pc}(x) = \nu_t(x_{pre}) = V_{A0}(x)$. We initialize $\nu_t(v_{sat})$ to true. Finally, $o_c = \emptyset$ and $\{(v, false) \mid v \in V_{OL}\}$, so $(\forall (o, v_o) \in \rho \,.\, (o \in o_c \iff \nu_t(v_{ol}) = true))$.

Suppose alternately that we have $k > 0$. We show that we can extend the trace σ_t to match σ_c at step $k+1$.

Given state σ_{ck}, σ_{ck+1} must be reached by one of the six transition rules in Definition 3. Suppose CENTA rule (1) is used. In this case, time advances by d. But in this case, we can apply NTA rule 1 for σ_{tk} for the same value of d. This is immediate for any of the invariants for machines $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$ because from the pre-state equivalence $\equiv_s$ the states have the same valuations for locations, variables, and clocks. For the valuation of $\mathcal{A}_a$, there is only one state (l_w) with invariant ($[\![c_\kappa \leq \kappa]\!], [\![v_{sat} = \text{true}]\!]$). v_{sat} is true in state k and remains true in $k+1$ since no variables change value during a time update. It remains to show that $u_t(c_\kappa) + d \leq \kappa$, which is straightforward since $u_t(c_\kappa) = u_c(c_\kappa)$ and (1) contains a constraint: $u(c_\kappa) + d \leq \kappa$. Therefore $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$.

Suppose CENTA rule (2) is used. In this case, a τ transition occurs in one of the machines $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$. In this case, same transition can occur in the translated model using rule (2), yielding the same destination state, clock resets and variable valuations, so (a), (b), (c) are immediately satisfied. Furthermore, ψ is not modified by rule (2), so the definitions v_{pc} and o_c remain the same. By $assigns\_ok$, it is also the case that no variables in the sets V_P and O_E or variable v_{sat} will be modified, so (d), (e), and (f) are maintained, and $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$.

Suppose CENTA rule (3) is used. In this case, the reasoning is very similar to rule (2).

Suppose CENTA rule (4) is used. In this case, we are latching an input signal into an input variable related to the AGREE contract. By rule (4), there exists an event $(\alpha_i, v_{ie}) \in \iota$. Therefore, we can apply rule (3) with the E_I transition:

$(l_w, \alpha_i?, (\llbracket\text{true}\rrbracket, \llbracket\text{true}\rrbracket), \emptyset, \{(v_i, \llbracket\text{true}\rrbracket)\}, l_w)$. The result of the application of rule (3) to the translation and rule (4) to the AGREE model performs the same variable modifications and clock resets, satisfying (a), (b), (c). By $assigns\_ok$, no variables in the sets V_P and O_E or variable v_{sat} will be modified, so (d), (e), and (f) are maintained and the state invariant for l_w is maintained, and $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$.

Suppose CENTA rule (5) is used. This rule has the form: $(\bar{l}_c, u_c, \nu_c, (\nu_{pc}, \emptyset)) \rightarrow (\bar{l}_c, u'_c, \nu''_c, (\nu'_c, o'_c))$ if $\nu'_c \in Val^O(\nu_c)$, $u_c(c_\kappa) = \kappa$, $C2S(\nu_c, \nu'_c)$, $u'_c = u_c \oplus (c_\kappa \mapsto 0)$, and $o'_c = \theta(\nu'_c)$. We note that ν'_c is constructed from ν_c by nondeterministically assigning a value to each of the m output variables from their types $Val^O(\nu_c)$. For the moment, we will call these additional assignments $\nu_O = \{(v_0, c_0), (v_1, c_1), \ldots, (v_m, c_m)\}$, and note that $\nu'_c = \nu_c \oplus \nu_O$. In the construction of the translated automata, we create an assignment for *every* such valuation of outputs in the Y_{TA} rule. We choose the edge e_{to} that has the matching assignment ν'_c from Y_{TA}: Y_{tao}. This edge is defined in the translation as: $(l_w, \tau, (\llbracket c_\kappa = \kappa \rrbracket, \bigwedge\{\llbracket v_o = \text{false}\rrbracket \mid v_o \in \text{range } \rho\}), \{c_\kappa\}, y, l_w)$, where $y = Y_{tao} \frown Y_{TS} \frown Y_{TO} \frown Y_{TP} \frown Y_{TI}$.

We first note that the guard for e_{to} is satisfied due to state equivalence on pre-states s_C and s_T (b) and (e). We then examine transition post-states. First, the valuations of l_c and l_{t-a} are unchanged in both rules and that the reset clocks are the same, satisfying equivalence parts (a) and (b) on the post-states. To determine equivalence of variable maps, we first describe intermediate variable maps during evaluation of y, noting that $\nu'_t = y(\nu_t)$ is equivalent to $\nu^1_t = Y_{tao}(\nu)$, $\nu^2_t = Y_{TS}(\nu^1)$, $\nu^3_t = Y_{TO}(\nu^2_t)$, $\nu^4_t = Y_{TP}(\nu^3_t)$, and $\nu'_t = Y_{TI}(\nu^4_t)$, and that each of the lists assign a disjoint set of variables. Because $\nu^1_t = \nu_t \oplus \nu_O$, $(\forall x \in \text{dom } \nu'_c \,.\, \nu'_c(x) = \nu^1_t(x))$, satisfying (c). By disjointness of assignments, (c) is also satisfied for ν'_t. Since Y_{tao} does not assign any 'pre' variables, $(\forall x \in V_A \,.\, \nu_{pc}(x) = \nu^1_t(x_{pre}))$ holds. From these equivalences of valuations of current and pre variables ν'_c, ν_{pc} with ν^1_t, we claim[2] that $C2S(\nu_{pc}, \nu'_c) = C2S^*(\nu^1_t)$. Therefore, $\nu^2_t(v_{sat}) = \nu'_t(v_{sat}) = true$, so we satisfy (f) and the state invariant of l_w. Next, we assign output latch variables to match outputs in ν^3 (satisfying (e)), and finally assign 'pre' variables based on current valuations in ν^4 (satisfying (d)). Finally, to satisfy (c) for ν'_t and ν''_c, we reset all latched input variables to false using Y_{TI}. Since variables other than latched inputs are unchanged, the properties (d) (e) (f) still hold.

Suppose finally that CENTA rule (6) is used. The proof here is very similar (and symmetric) to the proof of rule (4).

Since σ_{ck+1} must be derived from σ_{ck} through one of the six CENTA rules, and we demonstrate that any rule CENTA application has an analogous NTA rule for the translated AGREE model, it is possible to extend σ_{tk} to σ_{tk+1} such that $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$.

Proof: (2) By construction. The proof is similar to the proof of (1). Given an arbitrary trace σ_t, an equivalent trace σ_c is constructed by induction over the natural numbers. That is to say, we assume that $\sigma_{c1} \ldots \sigma_{ck}$ has been constructed, and then we are extending it to σ_{ck+1}. The base case is established in a similar way described for 1. Given state σ_{tk}, state σ_{tk+1} must be reached by one of the three rules in the definition of NTA. We show that, for each rule whereby σ_{tk+1} is reached, we can construct σ_{ck+1} using CENTA rules such that $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$. Suppose that we have reached σ_{tk+1} using NTA rule (1); using this rule, $(\bar{l}_t, u_t, \nu_t) \rightarrow (\bar{l}_t, u_t + d, \nu_t)$ such that, $I(\bar{l}_t)$ is satisfied after adding d to u_t. Therefore, we know that, in σ_{tk+1}, for every $\mathcal{A}_i \in \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n, \mathcal{A}_a\}$, invariants are satisfied. Since the invariant of $\mathcal{A}_a$ is $I = \{(l_w, (\llbracket c_\kappa \leq \kappa\rrbracket, \llbracket \nu_{sat} = true\rrbracket))\}$, we have $u_t(c_\kappa) + d \leq \kappa$. Then, here, with the same value of d, we can apply CENTA rule (1) to σ_{ck}. Due to pre-state equivalence for every $\mathcal{A}_i \in \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$, the states have the same valuation for locations, variables, and clocks. For $\mathcal{A}_a$, we have only one state l_w, where ν_{sat} remains true because, during the time update, no variables have changed. As $u_t(c_\kappa) = u_c(c_\kappa)$, the clocks are the same in state $k+1$ after adding the same amount of d to c_κ. Therefore, $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$.

Suppose we construct σ_{tk+1} using NTA rule (2); hence, there is an action τ by which a transition occurs in one of the automata of $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n, \mathcal{A}_a\}$. The proof is decomposed into two parts:

(2A.) Suppose τ occurs in one of the machines in $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$. In this case, the same transition can occur in the CENTA by applying CENTA rule (2). So, we will get the same destination states, clock resets, and variable valuations. Therefore, (a), (b), (c) are immediately established. As ψ is not modified by applying τ, valuation of ν_{pc} and o_c remain unchanged. By $assigns\_ok$, v_{sat} and variables of V_P and O_E will be unmodified so (d), (e), (f) are satisfied, and $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$.

(2B.) Suppose alternately that a τ transition occurs in $\mathcal{A}_a$. Such a transition must be derived using rule E_T. Suppose it is the (arbitrary) edge $e_t \in E_T = \{(l_w, \tau, (\llbracket c_\kappa = \kappa\rrbracket, \bigwedge\{\llbracket v_{ol} = false\rrbracket \mid v_{ol} \in \text{range } \rho\}), \{c_\kappa\}, y, l_w) \mid y \in Y_T\}$, where $y = Y_{tao} \frown Y_{TS} \frown Y_{TO} \frown Y_{TP} \frown Y_{TI}$ for some assignment sequence Y_{tao} that assigns all output variables (V_O) in the AGREE model. Since all the assignments in the sequence Y_{tao} are disjoint, we can view them as a set $\nu_O = \{(v_0, c_0), (v_1, c_1), \ldots, (v_m, c_m)\}$. We choose the assignment in $Val^O(\nu)$ matching the assignments to be ν'_c, i.e., $\nu'_c = \nu_c \oplus \nu_O$.

Because this transition occurred, the state invariant v_{sat} must be true in v'_t, so (f) holds. Now we will match the post-state by applying CENTA rule (5). To execute this rule, we first demonstrate the precondition $o_c = \emptyset$ holds in state k. To do so, we note that the guard of this transition states that all variables v_{ol} are false; by equivalence of pre-states (e), this means that the set $o_c = \emptyset$. We also must demonstrate $C2S(\nu_c, \nu'_c)$ is true. For this, we again describe intermediate variable maps during evaluation of y, noting that $\nu'_t = y(\nu_t)$ is equivalent to $\nu^1_t = Y_{tao}(\nu)$, $\nu^2_t = Y_{TS}(\nu^1)$, $\nu^3_t = Y_{TO}(\nu^2_t)$, $\nu^4_t = Y_{TP}(\nu^3_t)$, and $\nu'_t = Y_{TI}(\nu^4_t)$, and that each of the lists assign a disjoint set of variables. We (again) note that $\nu^1_t = \nu_t \oplus \nu_O$, so part (c) of the state equivalence holds between variable states ν'_c and ν^1_t, and also, by disjointness of assignments, that part (d) of the state equivalence holds between ν_{pc} and ν^1_t. From these

[2] A complete argument would require translation rules for replacing 'pre' expressions and expression evaluation semantics; this is a lengthy but not difficult argument.

equivalences and the assignment v_{sat} to true by evaluating $C2S^*$ in ν_t^1, we claim that $C2S(\nu_c, \nu_c')$.

We then examine transition post-states. First, the valuations of l_c and l_{t-a} are unchanged in both rules and that the reset clocks are the same, satisfying equivalence parts (a) and (b) on the post-states. By disjointness of assignments, (c) continues to hold up between ν_t^4 and ν_c'. Next, we assign output latch variables to match outputs in ν^3 (satisfying (e)), and finally assign 'pre' variables based on current valuations in ν^4 (satisfying (d)). Finally, to satisfy (c) for ν_t' and ν_c'', we reset all latched input variables to false using Y_{TI}. Since variables other than latched inputs are unchanged, the properties (d) (e) (f) still hold.

Suppose NTA rule (3) is used for the construction of σ_{tk+1}; therefore, there exists a transition from $(\bar{l_t}, u_t, \nu_t)$ to $(\bar{l_t}[l'_{tj}/l_{tj}, l'_{ti}/l_{ti}], u'_t, \nu'_t)$ s.t. $l_{ti} \xrightarrow{c?, g_i, y_i, r_i} l'_{ti}$ and $l_{tj} \xrightarrow{c!, g_j, y_j, r_j} l'_{tj}$, $i \neq j$, $(u_t, \nu_t) \in g_i$, $(u_t, \nu_t) \in g_j$, $(y_j(y_i(\nu_t))) = (y_i(y_j(\nu_t)))$, $u'_t = (r_i \cup r_j)(u_t)$, $\nu'_t = (y_j(y_i(\nu_t)))$, and $(u'_t, \nu'_t) \in I(\bar{l_t}[l'_{tj}/l_{tj}, l'_{ti}/l_{ti}])$. In this case, there are three possibilities. First, both sender and receiver belong to $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$. Here, we can apply CENTA rule (3) to construct σ_{ck+1}. As ψ does not change, rule (3) of both CENTA and NTA work in the same way, which immediately implies $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$. Second, the receiver is $\mathcal{A}_a$; then we can construct σ_{ck+1} using CENTA rule (4). By NTA rule (3), there is a transition in E_I of the form $(l_w, \alpha_i?, (\llbracket true \rrbracket, \llbracket true \rrbracket), \emptyset, \{(\nu_i, \llbracket true \rrbracket) l_w\})$, for which there is an equivalent event (α_i, ν_{ie}) in CENTA rule (4). These two perform the same variable modifications and clock resets. Therefore, (a), (b), (c) hold. In light of *assigns_ok*, (d), (e), (f) also hold, consequently $\sigma_{ck+1} \equiv_s \sigma_{tk+1}$. Finally, suppose the sender is $\mathcal{A}_a$; in this case, we can apply CENTA rule (6) to construct σ_{ck+1}. The proof here is analogous to the second case.

V. Discussion

Most of the interesting aspects of translation (unsurprisingly) have to do with timing. In the current translation the environment must respond to output events (via synchronization) before the AGREE component can execute its next cycle. We also considered requiring *immediate* synchronization (the environment must respond to all output events before time can advance) and removing the requirements on output event synchronization entirely. The translation can be easily adapted to these semantics.

For the modeling and proofs that are performed in this paper, the behavior of the controller (specified by its contract) is *timely* and *synchronous*: the controller executes exactly when its period elapses and requires no computation time. When reasoning about controller implementations, each of these timing assumptions is faulty (though often reasonable, as in our GPCA example, given the relative time scales of the controller vs. its environment).

To account for drift, rather than defining a definite instant at which the controller must execute, we could instead define a time range that includes the maximum amount of per-cycle drift, and modify the translation rules accordingly. This change (1) adds non-determinism to the model, significantly increasing the model state space, and (2) may require a re-basing of the UPPAAL unit time value: since UPPAAL only allows integer time constants, depending on the accuracy bounds on the drift, the mapping from the UPPAAL unit time scale to real time may need to be made significantly more dense.

Similarly, computation time within the implementation can be modeled using an additional "computing" state to describe the computation time of the implementation. The contract outputs are stored in local variables and "released" to the outputs when the computation is complete. Again, this poses no theoretical difficulty but makes the analysis considerably more expensive.

VI. Case Example

In this section we illustrate our approach using a Patient Controlled Analgesic (PCA) infusion pump, a medical device that is used to infuse liquid drug into a patient's bloodstream. PCA is an approach used to accurately infuse liquid pain medication, typically an opiate such as morphine, in post-operative patients.

A. *Closed Loop System and Infusion Pump Controller*

Consider a clinical scenario for critical care patients where PCA infusion pump is used for infusing pain medication. A serious side effect of opiate is that an overdose can lead to respiratory failure in patients. Hence a sensor, such as a pulse oximeter, continuously monitors the patient and raises an alarm if it senses signs of respiratory problems. Caregivers manually control the infusion of drug in response to such alarms. However, to improve the quality of care, a safety interlock device could be used in the system, as shown in Figure 4, that continuously monitors the pulse oximeter readings and stops the pump once a pre-set threshold is crossed. This system was modeled using Timed Automata and was verified for safety properties using UPPAAL model checker [19]. In an independent effort, we modeled an elaborate software controller of a generic PCA (GPCA) infusion pump using discrete notations such as AADL, AGREE and Simulink/Stateflow. Using AGREE tools we also demonstrated that the controller satisfies critical safety properties [13].

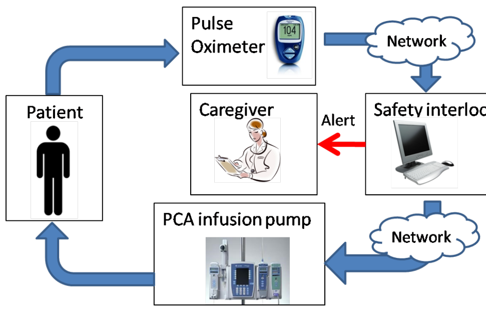


Fig. 4. Overview of the closed-loop system

B. *Linking Abstraction*

In a previous effort [20], we demonstrated a semi-formal approach to show that the particular infusion pump controller when used as part of closed-loop system can be guaranteed to uphold critical safety properties. In this work, we have defined an embedding that is proven to preserve the semantics of the discrete time contract in order to place the linking on a formal foundation.

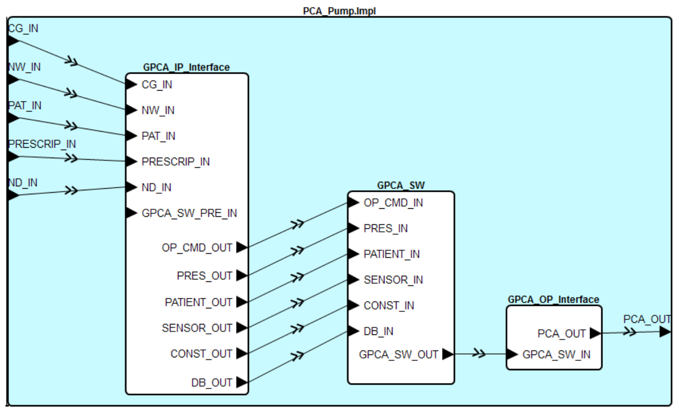
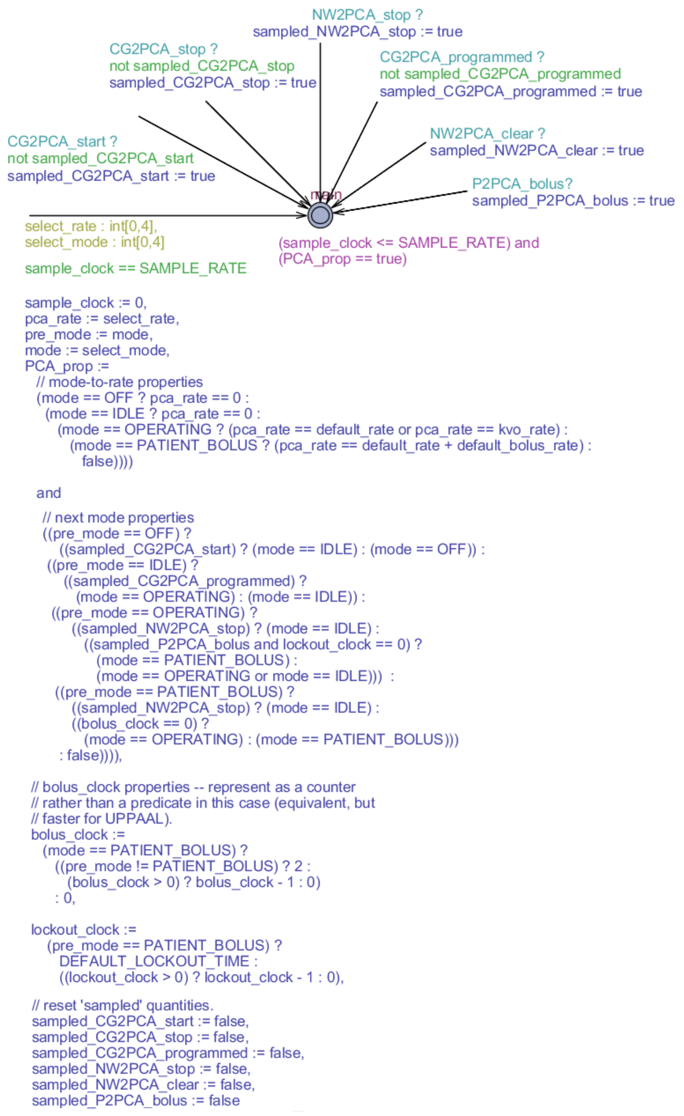


Fig. 5. PCA Architecture in AADL

In order to embed the GPCA controller into UPPAAL, we first had to unify the interfaces between the GPCA controller and the PCA UPPAAL model that was used for the closed-loop scenario. In order to adapt the properties that were proven over the original GPCA model to the new interface, we constructed an AADL model, shown in Figure 5, that normalized the interface between the two systems. We then used the properties proved about the GPCA model to prove new "architecture level" properties over the PCA interface using AGREE. Because the size of the generated UPPAAL model is directly related to the number of AADL properties, we included only properties that were relevant to the clinical scenario; otherwise we encountered scalability limits when using UPPAAL.

Using the rules defined in our approach and the contracts of GPCA we created the timed automata of the GPCA system as shown in Figure 6. This model was derived from four AGREE properties: (1) a mode-to-rate property that describes the infusion rate for different modes of operation, (2) a mode transition property that describes how the system mode is updated by different environmental stimuli, (3) and (4) properties that define the counters used for patient bolus length and bolus lockout. To verify this model, we replaced this automata in the place of the original PCA automata in the closed loop system and using the UPPAAL model checker, we verified all the system level properties (15 properties) of the closed loop system. This way, we were able to successfully demonstrate that the verified GPCA controller can be used in the closed loop system safely.

The translation of the AGREE contract was performed by hand using the rules in Section IV. Two fairly trivial optimizations were made after the schematic translation: First we did not define 'pre_' variables for variables that are not used in pre-expressions (these needlessly expand the UPPAAL state space). Second, we defined the behavior of the counters as UPPAAL assignments rather than as part of the next-state predicate (PCA_prop) to avoid the computational cost of additional *select* statements. This is sound if (a) a predicate for the variable is an equality with one side being the variable and the other side being an expression and (b) the assignment can be placed in dataflow order either before or after the evaluation of the next-state predicate. It is possible to provably establish the soundness of the optimizations by using (for example) simulation relation checking between the optimized and unoptimized models, but we did not do this.


NW2PCA_stop ?
sampled_NW2PCA_stop := true
CG2PCA_stop ?
not sampled_CG2PCA_stop
sampled_CG2PCA_stop := true
CG2PCA_programmed ?
not sampled_CG2PCA_programmed
sampled_CG2PCA_programmed := true
CG2PCA_start ?
not sampled_CG2PCA_start
sampled_CG2PCA_start := true
NW2PCA_clear ?
sampled_NW2PCA_clear := true
P2PCA_bolus?
sampled_P2PCA_bolus := true
main
(sample_clock <= SAMPLE_RATE) and
(PCA_prop == true)
select_rate : int[0,4],
select_mode : int[0,4]
sample_clock == SAMPLE_RATE

```
sample_clock := 0,
pca_rate := select_rate,
pre_mode := mode,
mode := select_mode,
PCA_prop :=
  // mode-to-rate properties
  (mode == OFF ? pca_rate == 0 :
   (mode == IDLE ? pca_rate == 0 :
     (mode == OPERATING ? (pca_rate == default_rate or pca_rate == kvo_rate) :
       (mode == PATIENT_BOLUS ? (pca_rate == default_rate + default_bolus_rate) :
         false))))

  and

   // next mode properties
   ((pre_mode == OFF) ?
      ((sampled_CG2PCA_start) ? (mode == IDLE) : (mode == OFF)) :
    ((pre_mode == IDLE) ?
       ((sampled_CG2PCA_programmed) ?
         (mode == OPERATING) : (mode == IDLE)) :
     ((pre_mode == OPERATING) ?
        ((sampled_NW2PCA_stop) ? (mode == IDLE) :
          ((sampled_P2PCA_bolus and lockout_clock == 0) ?
            (mode == PATIENT_BOLUS) :
            (mode == OPERATING or mode == IDLE)))  :
      ((pre_mode == PATIENT_BOLUS) ?
        ((sampled_NW2PCA_stop) ? (mode == IDLE) :
        ((bolus_clock == 0) ?
          (mode == OPERATING) : (mode == PATIENT_BOLUS)))
      : false)))),

// bolus_clock properties -- represent as a counter
// rather than a predicate in this case (equivalent, but
// faster for UPPAAL).
bolus_clock :=
  (mode == PATIENT_BOLUS) ?
    ((pre_mode != PATIENT_BOLUS) ? 2 :
      (bolus_clock > 0) ? bolus_clock - 1 : 0)
    : 0,

lockout_clock :=
   (pre_mode == PATIENT_BOLUS) ?
     DEFAULT_LOCKOUT_TIME :
     ((lockout_clock > 0) ? lockout_clock - 1 : 0),

// reset 'sampled' quantities.
sampled_CG2PCA_start := false,
sampled_CG2PCA_stop := false,
sampled_CG2PCA_programmed := false,
sampled_NW2PCA_stop := false,
sampled_NW2PCA_clear := false,
sampled_P2PCA_bolus := false
```

Fig. 6. AGREE Properties modeled as a Timed Automata

VII. Related work

In the context of multi-formalism modeling, frameworks have been developed to integrate models specified using different formalisms. The Möbius approach [21], provides a comprehensive infrastructure to support interacting formalisms and solvers. Metropolis [22] provides a meta-model with formal semantics to capture and analyze electronic system designs at multiple levels of abstractions. The Ptolemy project [23] seeks to address the problem of heterogeneous mixtures of models of computation for modeling and analyzing CPS. Synchronous language principles are exploited for hierarchical composition [24], a strategy also adopted here. Similarly, the OsMoSys approach [25] and AToM^3 [26], require the formalism (graph based) to be defined in terms of an XML based meta-language. The main drawback of these approaches is requiring the formalism to be defined using common semantics which may restrict the freedom of the designer to models using only those notations the framework.

Two-level approaches to the verification of embedded software, similar to the one considered in this paper, are commonly used; for example [27]. However, most of this work takes an informal approach to matching the semantics of the two levels. Our approach can serve as the formal foundation for most of those efforts as well. While the mapping depends on the formalisms used in each case, it can be constructed for a variety of formalisms, including hybrid systems specifications. Rajhans and Krogh [28], on the other hand, use behavioral semantics to derive a set of general conditions for compositionality across heterogeneous abstractions. In contrast, the present work provides a complete approach for the two specific formalisms considered (timed automata and AGREE automata).

An approach for the analysis of globally-asynchronous, locally synchronous (GALS) systems is proposed in [29]. The specification logic used in this work is similar to AGREE. However, they are focused only on the problem of asynchronous composition of synchronous systems without an asynchronous "environment", whereas in our work, the environment is a first-class consideration. Also, no formal argument as to the correctness of their translation is provided.

There are several streams of research on translating portions of AADL into model checkers, e.g., [30]–[33]. One immediate difference between all of these works and our work is the property notation used for specifying behavior; primarily these papers target subsets of the AADL *behavioral annex* which allows specification of partial implementations. We have instead focused on a declarative formalism (AGREE) which we feel provides a more natural formalism for "shall" requirements, as commonly used in avionics and medical device systems. For example, [31] provides a translation of AADL into FIACRE although, because of the conceptual gap between two languages, only a common subset of ADDL and FIACRE was considered. In [30], a technique for verifying the consistency/conformity of AADL specifications with the end product has been proposed. This work is primarily focused on flow and deadlock analysis rather than behaviorial specification. Chkouri *et al.*, in [33], proposed a method for translating AADL models into BIP, which makes it possible to make use of the BIP toolset for verification. Although this work supports behavioral reasoning over AADL models, no arguments are provided as to the correctness of their translation and there is no discussion about the scalability of the approach. In [32], authors provided a framework which verifies the correctness of an integrated model obtained from independent AADL specifications. Although they also considered assumptions and guarantees, they reason only in discrete time, and do not consider real-time issues.

Modeling and analysis of closed-loop medical systems have been primarily studied in the context of diabetes care. Much attention is given to modeling patient physiology and design of algorithms for glucose control; see, for example, [34], [35]. A closed-loop safety interlock for PCA infusion, similar to the one studied in this paper, has been proposed in [36], however, the authors do not show any analysis results. Similarly, the GPCA pump has been used in a number of case studies that involved a variety of formal methods to model different aspects of the pump behavior. In [37], code for a simple infusion controller has been generated from UPPAAL-verified code.

VIII. CONCLUSION

When systems become complex, multiple analyses must be utilized to demonstrate their correct behavior. One natural division is across multiple notions of time: software controllers can often be analyzed in discrete time, while determining their effect on the physical world requires continuous time. In this paper, we introduced a formal approach for embedding discrete time contracts into continuous timed models via a direct embedding semantics (CENTA). We then defined rules to translate a discrete-time contract into a timed automata state machine, and proved the two descriptions equivalent. The utility of the approach has been demonstrated on a clinical scenario involving a large-scale discrete time medical device controller.

While techniques used in this paper are specific to the two formalisms considered, we believe that this work can form the basis for a general, scalable and practical approach to layered verification of properties in complex cyber-physical systems.

Acknowledgments: We thank A. Gacek for his critiques of the formalism and proofs, and D. Cofer and J. Backes at Rockwell Collins for developing JKind and AGREE.

REFERENCES

[1] M. Pajic, R. Mangharam, O. Sokolsky, D. Arney, J. Goldman, and I. Lee, "Model-driven safety analysis of closed-loop medical systems," *Industrial Informatics, IEEE Transactions on*, vol. PP, pp. 1–12, 2012, in early online access.

[2] Z. Jiang, M. Pajic, and R. Mangharam, "Cyber-physical modeling of implantable cardiac medical devices," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 122–137, 2012.

[3] A. Joshi and M. Heimdahl, "Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier," in *SAFECOMP*, ser. LNCS. Springer-Verlag, 2005.

[4] J.-F. Etienne, S. Fechter, and E. Juppeaux, "Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain," in *Complex Systems Design & Management*, M. Aiguier, F. Bretaudeau, and D. Krob, Eds. Springer Berlin Heidelberg, 2010, pp. 61–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15654-0_4

[5] "Cbmc home page," http://www.cprover.org/cbmc/.

[6] "Java path finder home page," http://babelfish.arc.nasa.gov/trac/jpf/.

[7] P. R. Nicolas Halbwachs, Fabienne Lagnier, "Synchronous Observers and the Verification of Reactive Systems," in *Third Int'l Conf. on Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing*, June 1993.

[8] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[9] G. Behrmann, A. David, and K. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems (revised lectures)*, ser. LNCS, vol. 3185, 2004, pp. 200–237.

[10] D. D. Cofer *et al.*, "Compositional verification of architectural models," in *NFM*, vol. 7226, 2012, pp. 126–140.

[11] A. Murugesan *et al.*, "Linking abstract analysis to concrete design: A hierarchical approach to verify medical cps safety," *ICCPS*, pp. 139–150, 2014.

[12] S. P. Miller, M. W. Whalen, and D. D. Cofer, "Software model checking takes off," *Commun. ACM*, vol. 53, no. 2, pp. 58–64, 2010.

[13] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl, "Compositional verification of a medical device system," in *ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013*. ACM, November 2013.

[14] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[15] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, A. E. Goodloe and S. Person, Eds., vol. 7226. Berlin, Heidelberg: Springer-Verlag, April 2012, pp. 126–140.

[16] SAE-AS5506, *Architecture Analysis and Design Language*. SAE, Nov 2004.

[17] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[18] M. Whalen *et al.*, "Hierarchical multi-formalism proofs," University of Minnesota Sofware Engineering Center, Tech. Rep., 2014. [Online]. Available: www.umsec.umn.edu/publications/Hierarchical-Multi-Formalism-Proofs-Cyber-Physical

[19] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, , and R. Mangharam, "From verification to implementation: A model translation tool and a pacemaker case study," in *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012)*, Apr. 2012.

[20] A. Murugesan, O. Sokolsky, S. Rayadurgam, M. Whalen, M. Heimdahl, and I. Lee, "Linking abstract analysis to concrete design: A hierarchical approach to verify medical cps safety," *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, vol. 0, pp. 139–150, 2014.

[21] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *Software Engineering, IEEE Transactions on*, vol. 28, no. 10, pp. 956–969, 2002.

[22] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, April 2003.

[23] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Readings in hardware/software co-design," G. De Micheli, R. Ernst, and W. Wolf, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 2002, ch. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, pp. 527–543. [Online]. Available: http://dl.acm.org/citation.cfm?id=567003.567050

[24] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *Proceedings of the 7th ACM &Amp; IEEE Int'l Conf. on Embedded Software*, ser. EMSOFT '07. New York, NY, USA: ACM, 2007, pp. 114–123. [Online]. Available: http://doi.acm.org/10.1145/1289927.1289949

[25] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis, "The OsMoSys approach to multi-formalism modeling of systems," *Software and Systems Modeling*, vol. 3, no. 1, pp. 68–81, 2004.

[26] J. De Lara and H. Vangheluwe, "AToM^3: A tool for multi-formalism and meta-modelling," in *Fundamental approaches to software engineering*. Springer, 2002, pp. 174–188.

[27] R. Doornbos *et al.*, "Complementary verification of embedded software using ASD and UPPAAL," in *IIT*, 2012, pp. 60–65.

[28] A. Rajhans and B. H. Krogh, "Compositional heterogeneous abstraction," in *HSCC '13*, 2013, pp. 253–262.

[29] H. Günther *et al.*, "On the formal verification of systems of synchronous software components," in *SAFECOMP'12*, 2012, pp. 291–304.

[30] A. Johnsen *et al.*, "Automated verification of AADL-specifications using UPPAAL," in *HASE '12*, 2012, pp. 130–138.

[31] J.-P. Bodeveix *et al.*, "Towards a verified transformation from AADL to the formal component-based language FIACRE," *Sci. Comput. Program.*, vol. 106, no. C, pp. 30–53, 2015.

[32] I. Ruchkin *et al.*, "Active: A tool for integrating analysis contracts," in *The 5th Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2014.

[33] M. Y. Chkouri *et al.*, "Models in software engineering." Springer-Verlag, 2009, ch. Translating AADL into BIP - Application to the Verification of Real-Time Systems, pp. 5–19.

[34] S. Weinzimer, G. Steil, K. Swan, J. Dziura, N. Kurtz, and W. Tamborlane, "Fully automated closed-loop insulin delivery versus semiautomated hybrid control in pediatric patients with type 1 diabetes using an artificial pancreas," *Diabetes Care*, vol. 31, no. 5, pp. 934–939, May 2008.

[35] C. Cobelli, C. D. Man, G. Sparacino, L. Magni, G. D. Nicolao, and B. P. Kovatchev, "Diabetes: Models, signals, and control," *IEEE Reviews in Biomedical Engineering*, vol. 2, pp. 54–96, 2009.

[36] P.-A. Cortes, S. M. Krishnan, I. Lee, and J. M. Goldman, "Improving the safety of patient-controlled analgesia infusions with safety interlocks and closed-loop control," in *Proceedings of the 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability*, 2007, pp. 149–150.

[37] B. Kim, A. Ayoub, O. Sokolsky, I. Lee, P. Jones, Y. Zhang, and R. Jetley, "Safety-assured development of the GPCA infusion pump software," in *Proceedings of EMSOFT*, 2011.