A Formal Design Technique for Real-Time Embedded Systems Development using Duration Calculus

François Siewe, Dang Van Hung, Hussein Zedan and Antonio Cau

Abstract— **In this paper we present a syntactical approach for the design of real-time embedded systems. The requirement of the system is specified as Duration Calculus formula over continuous state variables. We model discretization at the state level and approximate continuous state variables by discrete ones. The discrete design is formulated as Duration Calculus formula over discrete state variables. The correctness of the design can be established using compositional proof rules. A real-time program is then derived from the discrete design using an extension of the assumption-commitment paradigm to real-time. We illustrate our approach using a simple water tank control system.**

Keywords— **Continuous specification, discrete design, real-time control program, compositional verification.**

I. Introduction

REAL-TIME control systems usually consist of some physical plant, in permanent interaction with its environment, for which a suitable controller has to be constructed such that the controlled plant exhibits the desired time dependent behaviour. They are safety-critical systems as a slight failure in the behaviour of the plant might cause damages to people and the environment. The challenge is to provide suitable technique for deriving the control program to be executed by the embedded computer.

A model of real-time control systems is depicted in figure 1. The *plant* denotes the continuous components of the system, in permanent interaction with the environment. The states of the environment and those of the plant can change at any time according to the laws of physics. Therefore the continuous time model (real numbers) is suitable for specifying their behaviour. The *controller* is a discrete component denoting a program executed by a computer. However, the state of a digital program changes only at discrete time points at ticks of a computer clock. The discrete time model (natural numbers) should then be considered for the implementation of the system.

The *sensors* sample the states of the plant for them to be observable by the controller. The *actuators* receive commands from the controller and change the state of the plant accordingly. The sensors constitute the *continuous-to-discrete* interfaces whereas the actuators implement the *discrete-to-continuous* interfaces. A question raises as how the two models of time can be combined into the same formalism such that the design and its correctness can accurately be reasoned about in a uniform manner.

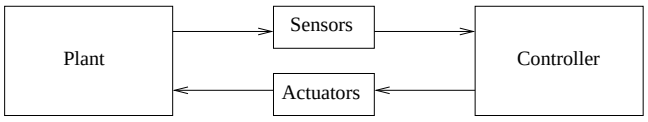

Fig. 1. A model of control system

This paper presents a syntactical approach for deriving real-time programs from a formal specification of the requirements of real-time systems. The approach provides a formal framework that can handle both continuous time and discrete time models in a uniform manner. We consider Duration Calculus (DC for short) [16, 17] as specification and top level design language, for its effectiveness in reasoning about real-time properties of systems. DC is a variant of Interval Temporal Logic [4] based on a continuous time model (i.e. times are real numbers). We denote by DC^* an extension of DC with iteration [8]. Our design technique is formulated as follows. At the first step of the design process a states model of the system is defined. It is a set of relevant plant state variables. Then the requirement of the system is formalised as a DC formula Req constraining the plant state variables. A design decision must be taken as how the requirements of the system will be met and refined into a detailed design Des such that $Des \Rightarrow Req$. Then the discretization step follows.

We approximate continuous state variables by discrete ones and formalise the relationship between them based on the general behaviour of the sensors and the actuators. Then the control requirement is derived from the detailed design and refined into a DC^* formula $Cont$ over discrete state variables such that $\mathcal{A} \vdash Cont \Rightarrow Des$, for some assumptions $\mathcal{A}$ about the behaviour of the environment (e.g. stability of states) and the relationship between continuous state variables and discrete state variables. The discrete formula $Cont$ is the formal specification of the controller. The last step of the design process consists to refine the discrete specification $Cont$ into a real-time control program using our extended assumption-commitment proof rules. Here, the implementation language is Occam used in ProCos project [6] for implementing real-time and concurrent systems. However, our approach can also be used with other real-time programming languages such as Signal or

F. Siewe, H. Zedan and A. Cau are with the Software Technology Research Laboratory (STRL), De Montfort University, Gateway House G4.61, The Gateway, LE1 9BH, Leicester, United Kingdom. E-mails: {fsiewe,hzedan,acau}@dmu.ac.uk

D.V. Hung is with the United Nations University / International Institute for Software Technology (UNU/IIST), P.O. Box 3058 Macau. E-mail: dvh@iist.unu.edu

Esterel.

The remainder of the paper is organized as follows. We give a summary of DC in Section II. Section III details our design technique. The program construction technique is presented in Section IV. Section V describes a simple example. We conclude the paper by a discussion in Section VI.

II. Duration Calculus with Iteration

In this section we give a brief summary of DC^*. The readers are referred to [8,16,17] for more details on the calculus. We are given the following sets of symbols: the set $GVar$ of global variables $x, y, z, \ldots$, independent of *time*; the set $Pvar$ of state variables $P, Q, \ldots$; the set $Tvar$ of temporal variables $v_1, v_2, v_3, \ldots$ interpreted as functions of time intervals; the set $FSymb$ of global function symbols $f, g, \ldots$ (such as *constants* and $+, -, *, \ldots$), and the set $RSymb$ of relation symbols $G, H, \ldots$ (such as *true* and *false*, and $=, \geq, \ldots$) which have standard meaning; and the set $PLetter$ of temporal propositional letters $X, Y, \ldots$ which will be interpreted as truth-valued functions of time intervals.

State expressions S, terms θ and formulas φ are built using the following grammars

$$\begin{array}{lcl} S & \widehat{=} & \mathbf{0} \mid P \mid \neg S \mid S \vee S \\ \theta & \widehat{=} & x \mid v \mid \int S \mid f(\theta, \ldots, \theta) \\ \varphi & \widehat{=} & X \mid G(\theta, \ldots, \theta) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi^\frown\varphi \mid \varphi^* \mid \exists x \varphi \end{array}$$

In addition to standard abbreviations, we denote

$$\begin{array}{lcllcl} [\![S]\!] & \widehat{=} & (\int S = \ell) \wedge (\ell > 0). & \ell & \widehat{=} & \int 1. \\ \Diamond\varphi & \widehat{=} & true^\frown(\varphi^\frown true). & \Box\varphi & \widehat{=} & \neg\Diamond\neg\varphi. \end{array}$$

The semantics of terms and formulas are defined as follows. Let a value assignment ν ($\nu : GVar \to I\!R$) for global variables, and an interpretation $\mathcal{I}$ for the symbols be given.

Let $\mathbf{Intv} \widehat{=} \{[b, e] \mid b, e \in I\!R \wedge b \leq e\}$ denote the set of time intervals. State expressions are interpreted as binary functions of time, i.e. $\mathcal{I}(S) : I\!R^+ \to \{0, 1\}$, such that $\mathcal{I}(S)(t) = 1$ means that state S is present at time t, and $\mathcal{I}(S)(t) = 0$ means that S is not present at time t. We assume that a state has finite variability in any finite time interval.

The semantics of a term θ is a mapping

$$\theta_{\mathcal{I}} : \mathbf{Intv} \to I\!R.$$

The semantics of terms of the kinds x or v, is exactly the one given by ν and $\mathcal{I}$. For a term $\theta = \int S$, $\theta_{\mathcal{I}}([b, e])$ is defined as $\int_b^e \mathcal{I}(S)(t)dt$. For a term $\theta = f(\theta_1, \ldots, \theta_n)$, $\theta_{\mathcal{I}}([b, e])$ is defined as $f((\theta_1)_{\mathcal{I}}([b, e]), \ldots, (\theta_n)_{\mathcal{I}}([b, e]))$. The semantics of a formula φ is a mapping

$$\mathcal{I}[\![\varphi]\!] : \mathbf{Intv} \to \{true, false\}$$

defined as follows.

- $\mathcal{I}[\![X]\!]([b, e]) = true$ iff $X_{\mathcal{I}}([b, e]) = true$;
- $\mathcal{I}[\![G(\theta_1, \ldots, \theta_n)]\!]([b, e]) = true$ iff $G((\theta_1)_{\mathcal{I}}([b, e]), \ldots, (\theta_n)_{\mathcal{I}}([b, e])) = true$;
- $\mathcal{I}[\![\varphi_1{}^\frown\varphi_2]\!]([b, e]) = true$ iff $\mathcal{I}[\![\varphi_1]\!]([b, m]) = true$ and $\mathcal{I}[\![\varphi_2]\!]([m, e]) = true$ for some $m \in [b, e]$;
- $\mathcal{I}[\![\varphi^*]\!]([b, e]) = true$ iff $b = e$ or $(\exists m_1, \ldots, m_n : b = m_1 < m_2 < \ldots < m_n = e$ and $\mathcal{I}[\![\varphi]\!]([m_i, m_{i+1}]) = true$ for $i = 1, \ldots, n-1)$;
- For other cases of φ the definition is just the same as in predicate calculus.

Readers are referred to [17] for the proof system of Duration Calculus. The following section presents our technique for deriving a discrete design from a continuous time specification.

III. From Continuous Specification to Discrete Design

This section presents a syntactical approach for designing real-time hybrid systems that can handle both continuous time and discrete time models in a uniform formal framework. The main idea of our approach is to model the discretization at the state level and to approximate continuous state variables by discrete ones based on the general behaviour of the sensors and actuators. Then the control requirement is formulated as a Duration Calculus over discrete state variables. We provide rules useful for refining and verifying the correctness of a design syntactically.

A. Discrete State variable

We take discrete time points to be natural numbers. A discrete state variable s is interpreted as a function from $I\!N$ (set of natural numbers) to $\{0, 1\}$, i.e. $\mathcal{I}(s) : I\!N \to \{0, 1\}$, for any interpretation $\mathcal{I}$. Therefore a discrete state variable can only be evaluated at discrete time points. In order to deal with continuous time model we suppose that the value of a discrete state variable does not change in between two consecutive discrete time points. So we can accurately extend to the continuous time model any discrete state s by a continuous state u_s defined by:

$$\mathcal{I}(u_s) : I\!R^+ \to \{0, 1\}$$

such that $\mathcal{I}(u_s)(t) \widehat{=} \mathcal{I}(s)(\lfloor t \rfloor)$ for any interpretation $\mathcal{I}$, where $\lfloor t \rfloor$ stands for the greatest discrete time point less than or equal to t. In the following, by discrete state variable we mean its extension to continuous time. So in our model a discrete state is just a particular case of continuous state that can change only at discrete time points at ticks of a given clock. In general, continuous state variables might change independently at any time.

B. Formalising the Discrete Interface

As depicted in Figure 1, the interface between continuous components and discrete ones in hybrid systems is essentially implemented by sensors and actuators. We formalise the behaviour of sensors and actuators as relationship between continuous state variables and discrete ones. In this respect, we consider the following definitions.

Definition III.1 (Stability) Given a state variable s and a positive real number δ, we say s is δ-*stable* iff the following formula is satisfied by any interval

δ-*stable*$(s) \widehat{=}$

$$\Box([\![\neg s]\!]^\frown[\![s]\!]^\frown[\![\neg s]\!] \Rightarrow [\![\neg s]\!]^\frown([\![s]\!] \wedge \ell > \delta)^\frown[\![\neg s]\!]).$$

The formula δ-*stable*(s) means that s lasts for at least δ time units whenever it becomes true.

Definition III.2 (Control relation) Given two state variables r and s, and a non-negative real number δ, we say r *δ-controls* s iff the following formula is satisfied by any interval

$$r \rhd_\delta s \mathrel{\widehat{=}} \Box(\lceil r \rceil \wedge \ell > \delta \Rightarrow (\ell \leq \delta)^\frown \lceil s \rceil).$$

The formula $r \rhd_\delta s$ states that when r lasts for a period longer than δ time units, s must hold and last until at least r becomes false. The *control relation* can be used for formalising the behaviour of actuators. Let r be a state variable modelling a program command (output port), and s a state of the plant. Then the relation $r \rhd_\delta s$ means that whenever the controller issues the command r, the plant must get into state s within at most δ time units, where δ stands for the *latency* of the actuators.

Definition III.3 (Observation relation) Given two state variables r and s, and a non-negative real number δ, we say r *δ-observes* s iff the following formula is satisfied by any interval

$$r \overrightarrow{\approx}_\delta s \mathrel{\widehat{=}} (s \rhd_\delta r) \wedge (\neg s \rhd_\delta \neg r).$$

The formula $r \overrightarrow{\approx}_\delta s$ states that any change in s that is stable for at least δ time units is observable in r. The *observation relation* can be used for formalising the behaviour of sensors. Let r be a state variable (input port) used by the controller to observe the environment state s through sensors. Then the relation $r \overrightarrow{\approx}_\delta s$ means that any change (stable enough) in s is observed by the controller within δ time units. So the sampling step is δ time units.

C. Some Discretization Rules

The design process starts from the specification of the requirements of the system as a formula over continuous time state variables. Then a number of design decisions are taken successively to refine the system further towards an implementation. At certain stage of the design process, discrete time state variables are gradually introduced to approximate continuous time ones. This leads to a *hybrid* specification in which continuous time state variables and discrete time state variables are intermixed. Establishing the correctness of such a specification is not a trivial task. In [12], we proposed a rich set of verification rules useful for establishing formally the correctness of a discrete design w.r.t. a continuous specification. Here are some samples of those rules; interested readers are referred to the above reference for further details.

Let $F(X)$ be a DC^* formula that may contain occurrences of (a sub-formula) X. The rule

$$\frac{s \rhd_\delta r}{F(\lceil s \rceil \wedge \ell > \delta) \Rightarrow F((\ell \leq \delta)^\frown \lceil r \rceil)}$$

states that a design of the form $F(\lceil s \rceil \wedge \ell > \delta)$ refines a specification of the form $F((\ell \leq \delta)^\frown \lceil r \rceil)$ provided that the (discrete time) state variable s is used to control the (possibly continuous time) state variable r.

If a continuous time state variable is stable enough then its sampling lasts for about as long as it does, with possibly a delay not longer than the sampling step. This property is expressed by the following rule.

$$\frac{r \overrightarrow{\approx}_\delta s \quad \delta\text{-stable}(s)}{\Box(\lceil s \rceil \Rightarrow \ell \leq \tau) \Rightarrow \Box(\lceil r \rceil \Rightarrow \ell \leq \tau + \delta)}$$

Furthermore, the delay induced by a cascaded sampling of a state variable can be reasoned about using rules such as

$$\frac{r \overrightarrow{\approx}_\delta s \quad s \overrightarrow{\approx}_\tau t}{r \overrightarrow{\approx}_{(\delta+\tau)} t} \quad .$$

The following section presents a compositional approach for verifying that a real-time program satisfies a discrete design.

IV. Real-Time Programs Construction

Our design technique for real-time control systems aims at developing a control program that allows the system to meet its real-time requirement. The Occam programming language is considered in ProCos project [6] as a suitable programming language for implementing real-time systems. In this section we give the syntax of Occam and investigate assumption-commitment style rules for real-time programs verification.

A. Syntax and Informal Semantics

The following BNFs describe an abstract syntax of Occam 3. We denote boolean expressions by b, arithmetic expressions by e, natural numbers by n, program variables by x, channels by c, and processes by P.

$$\begin{aligned} P &::= \texttt{skip} \mid \texttt{stop} \mid x := e \mid c?x \mid c!e \mid \Delta\ n \mid P_1; P_2 \mid \\ &\quad P_1 \parallel P_2 \mid \texttt{if}\ [b_i\ P_i]_{i=1}^n \mid \texttt{alt}\ [G_i\ P_i]_{i=1}^n \mid \texttt{while}\ b\ P \\ G &::= b\&g \mid g \\ g &::= c?x \mid c!e \mid \texttt{skip} \end{aligned}$$

The `skip` statement preserves its usual semantics. It does not change the state nor it takes time. The `stop` statement does not change the state but it does not terminate. However it lets time advancing. The input statement $c?x$ receives a value on channel c and assigns this value to the variable x. The ouput statement $c!e$ evaluates the expression e and sends its value along channel c. The delay statement $\Delta\ n$ holds the execution for n time units, then terminates. A conditional $\texttt{if}\ [b_i\ P_i]_{i=1}^n$ combines a number of processes each of which is guarded by a boolean expression. The conditional evaluates each boolean expression in sequence; if a boolean expression is found to be true the associated process is performed, and the conditional terminates. If none of the boolean expressions is true the conditional behaves like the process `stop`.

An alternation $\texttt{alt}\ [G_i\ P_i]_{i=1}^n$ combines a number of processes guarded by inputs. The alternation perfoms the process associated with a guard which is ready. If no channel is ready then the alternation waits until an input becomes ready. If many inputs are ready, only one of the inputs and its associated process are performed. A boolean expression may be included (the guard has the form $b\&g$) in an alternation to selectively exclude inputs (if b is evaluated to false in the guard) from being considered ready. A parallel $P_1 \parallel P_2$ combines a number of processes which are performed concurrently. The processes start together and

the parallel terminates when all the processes have terminated. Variables and channels in a parallel are subject to disjointness rules which prevent them from being accidentally shared between processes. The remaining statements have their usual meanings.

B. Compositional Verification of Programs

In this section we investigate how to develop an Occam program that satisfies a real-time property. The approach must be compositional and syntax-based to be easy to use in practice. The assumption-commitment paradigm (also known as *rely-guarantee*) is intensively studied [2,9,10,13,14] for reasoning about the behaviour of concurrent processes. The specification of a program has the general form (p, A, C, q), where p and q are predicates representing the precondition and the postcondition respectively, A specifies the assumption about the behaviour of the environment of the program and C expresses the commitment of the program.

Such a specification is interpreted as follows. If the program starts its execution in a state satisfying the precondition p and any transition of the environment satisfies the assumption A and the program terminates then any transition of the program satisfies the commitment C and the final state satisfies the postcondition q.

We extend the assumption-commitment paradigm to real-time by embedding a duration calculus formula φ within the specification. The formula φ specifies the temopral behaviour of the program. Thus we can develop compositional rules for reasoning about both functional and real-time behaviour of processes in a uniform manner. In general real-time control programs run forever. Our extension has two forms: the first form is (p, A, φ, C, q) for terminating behaviour and the second form is (p, A, φ, C) for non-terminating behaviour with the following meaning.

- **Termination:** A correctness formula

$$P \ \underline{\mathbf{Sat}}_{fin} \ (p, A, \varphi, C, q)$$

means that if p holds in the initial state, and any transition of the environment satisfies A, and P terminates then any transition of P satisfies C, the execution period of P satisfies φ and q holds in the final state.

- **Non-termination:** $P \ \underline{\mathbf{Sat}}_{inf} \ (p, A, \varphi, C)$, means that if p holds in the initial state, and any transition of the environment satisfies A, and P does not terminate then any transition of P satisfies C and any prefix of the execution period of P satisfies φ.

Note that it can be recognised syntactically if a tuple is expressing properties of finite behaviour or is expressing properties of infinite behaviour for a process. Like in Duration Calculus, *false* and *true* are overloaded to be both predicate and DC formula with the obvious meaning. According to our interpretation, a correctness formula $P \ \underline{\mathbf{Sat}}_{inf} \ (p, A, false, C)$ means that the process P under assumption A should terminate when started from a state satisfying the predicate p. Similarly, any correctness formula from the forms

- $P \ \underline{\mathbf{Sat}}_{fin} \ (p, A, false, C, true)$,
- P $\underline{\mathbf{Sat}}_{fin}$ (p, A, false, C, false),
- and $P \ \underline{\mathbf{Sat}}_{fin} \ (p, A, true, C, false)$

means that the process P under assumption A should not terminate when started from a state satisfying the predicate p. Due to space limit the readers are referred to [11] for a complete presentation of the verification rules. Here are some samples of the rules. The notation $PREF(\varphi)$ stands for a formula that holds for all prefixes of an interval that satisfies the formula φ.

Rule 1 states the semantics of the sequential composition of processes. It highlights the similarity between the sequential composition of processes and the *chop* operator of duration formulas. The assumption is taken to be a fix-point of the operator $\Box$ (always) and the commitment the fix-point of the operator "*" (chopstar). Rule 1-a says that a sequence $P_1; P_2$ terminates if both P_1 and P_2 terminate. Rule 1-b states that a sequence $P_1; P_2$ fails to terminate if P_1 or P_2 fails to terminate.

Rule 1 (**Sequence**)

$$\text{(a)} \ \frac{P_1 \ \underline{\mathbf{Sat}}_{fin} \ (p, A, \varphi_1, C, m) \quad C \equiv C^* \\ P_2 \ \underline{\mathbf{Sat}}_{fin} \ (m, A, \varphi_2, C, q) \quad A \equiv \Box A}{P_1; P_2 \ \underline{\mathbf{Sat}}_{fin} \ (p, A, \varphi_1 \frown \varphi_2, C, q)}$$

$$\text{(b)} \ \frac{P_1 \ \underline{\mathbf{Sat}}_{fin} \ (p, A, \varphi, C, m) \quad P_2 \ \underline{\mathbf{Sat}}_{inf} \ (m, A, \alpha_2, C) \\ P_1 \ \underline{\mathbf{Sat}}_{inf} \ (p, A, \alpha_1, C) \qquad A \equiv \Box A \quad C \equiv C^*}{P_1; P_2 \ \underline{\mathbf{Sat}}_{inf} \ (p, A, \alpha_1 \vee PREF(\varphi \frown \alpha_2), C)}$$

The alternation allows a process to wait for input on several channels at the same time. All the boolean expressions are evaluated. The input guards and processes corresponding to the boolean expressions which are evaluated to false are discarded. The alternation executes in parallel the input guards associated with the boolean expressions which are evaluated to true. The first guard triggered and the associated process are executed and the alternation terminates (Rule 2-a and Rule 2-b). At any execution only one branch is chosen, the one that the guard is triggered first. However we do not know beforehand which guard will be triggered first. That is why in our rule the assumption for alternation is the conjunction of the assumptions about the execution of the different branches. For the same reason the commitment and the real-time behaviour of the alternation are disjunctions. Besides, if none of the boolean expressions is evaluated to true, the alternation behaves like `stop` (Rule 2-c and Rule 2-d).

Rule 2 (**Alternation**)

$$\text{(a)} \ \frac{\forall i \in J \bullet \ p \Rightarrow b_i \\ \forall i \notin J \bullet \ p \Rightarrow \neg b_i \\ \forall i \in J \bullet \ g_i; P_i \ \underline{\mathbf{Sat}}_{fin} \ (p \wedge b_i, A_i, \varphi_i, C_i, q_i) \quad J \neq \emptyset}{\texttt{alt} \ [b_i \& g_i \ P_i]_{i=1}^n \ \underline{\mathbf{Sat}}_{fin} \ (p, \underset{i\in J}{\wedge} A_i, \underset{i\in J}{\vee} \varphi_i, \underset{i\in J}{\vee} C_i, \underset{i\in J}{\vee} q_i)}$$

$$\text{(b)} \ \frac{\forall i \in J \bullet \ p \Rightarrow b_i \\ \forall i \notin J \bullet \ p \Rightarrow \neg b_i \\ \forall i \in J \bullet \ g_i; P_i \ \underline{\mathbf{Sat}}_{inf} \ (p \wedge b_i, A_i, \varphi_i, C_i) \quad J \neq \emptyset}{\texttt{alt} \ [b_i \& g_i \ P_i]_{i=1}^n \ \underline{\mathbf{Sat}}_{inf} \ (p, \underset{i\in J}{\wedge} A_i, \underset{i\in J}{\vee} \varphi_i, \underset{i\in J}{\vee} C_i)}$$

$$\text{(c)} \ \frac{p \Rightarrow \wedge_{i=1}^n \neg b_i}{\texttt{alt} \ [b_i \& g_i \ P_i]_{i=1}^n \ \underline{\mathbf{Sat}}_{fin} \ (p, true, false, true, false)}$$

$$\text{(d)} \ \frac{p \Rightarrow \wedge_{i=1}^n \neg b_i}{\texttt{alt} \ [b_i \& g_i \ P_i]_{i=1}^n \ \underline{\mathbf{Sat}}_{inf} \ (p, true, [\![p]\!]^*, true)}$$

Rule 3 states the semantics of the parallel composition of processes. Let A_i be the assumption about the environment of the process P_i, $i = 1, 2$. The interactions between concurrent processes are specified by the formulas $A \wedge C_2 \Rightarrow A_1$ and $A \wedge C_1 \Rightarrow A_2$ meaning that the commitment of process P_2 (P_1) is an assumption for P_1 (P_2 respectively) as each of the processes is part of the environment of the other. In Occam, processes involved in a parallel do not share variables, but can exchange data through channels. The real-time behaviour of a parallel can be expressed as the conjunction of the behaviours of its processes as they are performed concurrently. The fact that some processes might complete their execution before others is expressed by adding *true* as suffix to those formulas corresponding to shorter processes. However the parallel terminates when all the processes have terminated (Rule 3-a). The parallel fails to terminate if at least one of the processes does (Rule 3-b).

Rule 3 (**Parallel**)

(a) $$\frac{P_1 \ \underline{\mathbf{Sat}}_{fin}\ (p, A_1, \varphi_1, C_1, q_1) \qquad A \wedge C_2 \Rightarrow A_1 \\ P_2 \ \underline{\mathbf{Sat}}_{fin}\ (p, A_2, \varphi_2, C_2, q_2) \qquad A \wedge C_1 \Rightarrow A_2}{P_1 \parallel P_2 \ \underline{\mathbf{Sat}}_{fin}\ (p, A, \varphi, C_1 \wedge C_2, q_1 \wedge q_2)}$$

(b) $$\frac{A \wedge C_2 \Rightarrow A_1 \qquad A \wedge C_1 \Rightarrow A_2 \\ P_1 \ \underline{\mathbf{Sat}}_{inf}\ (p, A_1, \alpha_1, C_1) \qquad P_2 \ \underline{\mathbf{Sat}}_{inf}\ (p, A_2, \alpha_2, C_2) \\ P_1 \ \underline{\mathbf{Sat}}_{fin}\ (p, A_1, \varphi_1, C_1, q_1) \qquad P_2 \ \underline{\mathbf{Sat}}_{fin}\ (p, A_2, \varphi_2, C_2, q_2)}{P_1 \parallel P_2 \ \underline{\mathbf{Sat}}_{inf}\ (p, A, (\alpha_1 \wedge \alpha_2) \vee \alpha, C_1 \wedge C_2)}$$

where $\alpha \mathrel{\widehat{=}} (\alpha_1 \wedge \mathit{PREF}(\varphi_2{}^\frown \mathit{true})) \vee (\alpha_2 \wedge \mathit{PREF}(\varphi_1{}^\frown \mathit{true}))$, and $\varphi \mathrel{\widehat{=}} ((\varphi_1{}^\frown \mathit{true}) \wedge \varphi_2) \vee (\varphi_1 \wedge (\varphi_2{}^\frown \mathit{true}))$. We illustrate our approach in the following section for the design of a simple water tank controller.

V. A Simple Example: Water Tank Controller

We consider a water level controller that opens and closes a valve regulating the outflow of water from a container. This example is inspired from [7]. The container has an input vent through which water flows at unknown rate but not greater than λ_{in}. The valve with a larger capacity, λ_{out}, allows to remove water from the container. The requirement of the system is that the water level must be kept in between a safety low level SL and a safety high level SH units. Let ω denoting the water level at any time. We consider the following continuous state variables $s \mathrel{\widehat{=}} \omega \geq SL$ which holds when the water level is above the safety low level, and $r \mathrel{\widehat{=}} \omega \leq SH$ which holds when the water level is below the safety high level. Then the requirement of the water level monitoring system is formalised by

$$\mathit{Req} \mathrel{\widehat{=}} \Box(\ell > 0 \Rightarrow [\![s \wedge r]\!])$$

We assume that w is a continuous function of time. A device (sensors) for measuring the water level is installed. A critical low level, CL, and a critical high level, CH, are selected. Initially the water level is ω_0, and it must be in between the critical low level and the critical high level. Then the response time θ for the control system is calculated from the parameters λ_{in}, λ_{out}, CL, CH of the system such that

(i) $CL \leq w_0 \leq CH$

(ii) $SH \geq CH + \lambda_{in}\theta$

(iii) $SL \leq CL - \lambda_{out}\theta$

We suppose that the valve is either opened or closed. We model the state of the valve by a continuous state variable *Open* which is true when the valve is opened and false when it is closed. Let $\tilde{s} \mathrel{\widehat{=}} \omega < CL$ and $\tilde{r} \mathrel{\widehat{=}} \omega > CH$ be two continuous state variables. The state variable $\tilde{r}$ holds if the water level is above the critical high level. The state variable $\tilde{s}$ holds if the water level is below the critical low level.

It follows that (i) implies $[\![\neg\tilde{r} \wedge \neg\tilde{s}]\!]{}^\frown \mathit{true}$, (ii) implies $O_1 \rhd_\theta \neg\tilde{r}$ and (iii) implies $O_2 \rhd_\theta \neg\tilde{s}$, where $O_1 \mathrel{\widehat{=}} \mathit{Open} \wedge r$ and $O_2 \mathrel{\widehat{=}} \neg\mathit{Open} \wedge s$. So the assumptions about the behaviour of the environment can be specified as

$$\mathcal{B} \mathrel{\widehat{=}} ([\![\neg\tilde{r} \wedge \neg\tilde{s}]\!]{}^\frown \mathit{true}) \wedge (O_1 \rhd_\theta \neg\tilde{r}) \wedge (O_2 \rhd_\theta \neg\tilde{s}).$$

Then we consider the control requirement that the valve should not be kept closed (opened) for more than θ time units whenever the water level is above the critical high level (below the critical low level respectively). Thus the design decision is formalised by

$$\mathit{Des} \mathrel{\widehat{=}} (\tilde{r} \rhd_\theta \mathit{Open}) \wedge (\tilde{s} \rhd_\theta \neg\mathit{Open}) \wedge \mathcal{B}.$$

The correctness of the design is established by the following theorem.

Theorem 1: $\vdash \mathit{Des} \Rightarrow \mathit{Req}$.

Due the space limit, we refer the readers to [11] for the details of the proof of the theorem. For the sake of simplicity we assume that the controller can observe $\tilde{r}$ and $\tilde{s}$ through sensors. Let the state variables $\tilde{r}_c$ and $\tilde{s}_c$ be their samplings. The relationships between them are $\tilde{r}_c \overrightarrow{\approx}_\delta \tilde{r}$ and $\tilde{s}_c \overrightarrow{\approx}_\delta \tilde{s}$, where δ is the sampling step. In order to model the behaviour of the actuators, we introduce a program variable *valve* ranging over $\{0, 1\}$ which is 1 when the controller requests the actuators to open the valve and 0 when the controller requests the actuators to close the valve. Let

- $\varphi \mathrel{\widehat{=}} [\![\neg\tilde{r}_c \wedge \neg\tilde{s}_c]\!]^*{}^\frown([\![\mathit{valve} \wedge \tilde{r}_c]\!]^* \vee [\![\neg\mathit{valve} \wedge \tilde{s}_c]\!]^*)$.
- $\mathit{Cont} \mathrel{\widehat{=}} \varphi^*{}^\frown \mathit{PREF}(\varphi)$.
- $$\mathcal{A} \mathrel{\widehat{=}} \left(\begin{array}{c} (\tilde{r}_c \overrightarrow{\approx}_\delta \tilde{r}) \wedge (\tilde{s}_c \overrightarrow{\approx}_\delta \tilde{s}) \wedge (\mathit{valve} \rhd_\tau \mathit{Open}) \\ \wedge \\ (\neg\mathit{valve} \rhd_\tau \neg\mathit{Open}) \wedge (\delta + \tau \leq \theta) \wedge \mathcal{B} \end{array} \right).$$

The following Theorem holds (see [11] for the proof).

Theorem 2: $\mathcal{A} \vdash \mathit{Cont} \Rightarrow \mathit{Des}$.

The formula $\mathcal{A}$ represents the assumption about the behaviour of the environment (i.e. $\mathcal{B}$) and the relationship between continuous state variables and discrete state variables. *Cont* is a discrete specification of the control program. The control program is depicted in Table I. The correctness of the control program w.r.t. the specification *Cont* is stated by the following theorem.

Theorem 3:

$\mathit{Controller}\ \underline{\mathbf{Sat}}_{inf}\ (\neg\tilde{r}_c \wedge \neg\tilde{r}_c, \mathcal{A}, \mathit{Cont}, \mathit{true})$.

The theorem says that if initially the water level is between the critical low level and the critical high level, and the

```
Controller ≙  seq
                r̃_c := 0
                s̃_c := 0
                while true
                  alt
                    sensor_1?r̃_c   seq
                                     if
                                       r̃_c
                                         valve!1
                                       ¬r̃_c
                                         skip
                    sensor_2?s̃_c   seq
                                     if
                                       s̃_c
                                         valve!0
                                       ¬s̃_c
                                         skip
```

TABLE I
THE WATER LEVEL CONTROL PROGRAM

environment (e.g. sensors and actuators) satisfies the assumption $\mathcal{A}$ then the behaviour of the program *controller* satisfies the specification *Cont*.

VI. Discussion

In this paper we have presented a technique for the design of real-time hybrid systems using Duration Calculus. Our technique aims to derive an Occam program that controls a physical plant to meet a real-time requirement. We provide rules for refining a continuous requirement into a discrete design which is a Duration Calculus formula over discrete state variables. The correctness of the discrete design w.r.t. the continuous requirement can be proved under assumptions about the behaviour of the environment (e.g. the stability of the plant state variables) and the relationship between continuous state variables and discrete ones. The discrete design represents in fact a specification of the control program. We have extended the assumption-commitment paradigm to capture the real-time behaviour of programs, and provided compositional rules for verifying the correctness of Occam programs w.r.t. both their functional and real-time behaviour. The termination and non-termination of programs can also be reasoned about in a natural way.

In the literature, some works have addressed the problem. Fränzle has developed in [5] a technique for synthesizing controllers from Duration Calculus specifications. His approach is semantical, and leads to an automaton as controller. Similar works can be found in [1, 3]. Our aim is to provide the designers with syntax-based compositional interface for the design and verification of real-time programs, hiding semantic details. Another advantage of a syntactical approach is that the design process can be assisted by proof tools. In [7], Hooman extended the Hoare triples by adding timing primitives in the assertion language. This has constituted an interesting extension of Hoare triples for reasoning about temporal properties of programs. However the model is not suitable enough for reasoning about concurrent programs. Xu and Mohalik in [15] presented a technique for reasoning about the behaviour of real-time concurrent programs, based on assumption-commitment paradigm. Their approach can be used to reason about the functional behaviour of real-time programs. Our approach extends theirs by introducing in an assumption-commitment specification a formula representing the temporal specification of the program.

References

[1] F. Ajayi and D. V. Hung. Translating DC Designs into Occam. *Technical Report 268*, UNU/IIST, P.O. Box 3058, Macau, December 2002.

[2] A. Cau and P. Collette. Parallel composition of assumption-commitment specifications, a unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica,* 33:153-176, 1996.

[3] H. Dierks. Synthesizing controller from real-time specifications. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 18(1), 1999.

[4] B. Dutertre. On First Order Interval Temporal Logic. *Technical Report CSD-TR94-3*, Department of Computer science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, England, 1995.

[5] M. Fränzle. Synthesizing Controllers from Duration Calculus Specifications. Proceedings of *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRFT'96)*, LNCS 1135, Springer-Verlag, 1996.

[6] Jifeng He, C. A. R. Hoare, E.-R. Olderog Markus Muller-Olm, M. Schenke, M. R. Hansen, Anders P. Ravn, and H. Rischel. The ProCoS approach to the design of real-time systems: Linking different formalisms. *Tutorial Material for FME'96*, March 1996.

[7] Jozef Hooman. Extending Hoare Logic to Real-Time. *Formal Aspects of Computing*, 6A:801-825, 1994.

[8] D. V. Hung and Dimitar P. Guelev. Completeness of a fragment of Duration Calculus with Iteration. *Proceedings of Asian Computing Science Conference (ASIAN'99)*, Phuket, Thailand, December 10-12, 1999, P.S. Thiagarajan and R. Yap (eds), *Advances in Computing Science*, LNCS 1742, Springer-Verlag, pp. 139–150, 1999.

[9] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596-619, 1983.

[10] J. Misra and K.M. Chandy. Proofs of Network of Processes. *IEEE Transactions on Software Engineering*, 7(7):417-426, 1981.

[11] F. Siewe and D. V. Hung. Deriving Real-time Programs from Duration Calculus Specifications. *Technical Report 222*, UNU/IIST, P.O. Box 3058, Macau, December 2000.

[12] F. Siewe and D. V. Hung. From Continuous Specification to Discrete Design. In the *proceedings of the International Conference on Software: Theory and Practice (ICS2000)*, Yulin Feng, david Notkin and Marie-Claude Gaudel (eds), Beijing, August 21-24, pp. 407–414, 2000.

[13] C. Stirling. A Generalization of Owicki-Gries's Hoare Logic for a Concurrent While Language. *Theoretical Computer science*, North-Holland, 58:801-825, 1988.

[14] X. Qiwen, W.-P. de Roever and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing,* 9:2:149-174, 1997.

[15] X. Qiwen and M. Swarup. Compositional Reasoning using Assumption-Commitment Paradigm. *Technical Report 136*, UNU/IIST, P.O. Box 3058, Macau, February 1998.

[16] Z. Chaochen, C.A.R. Hoare, and A. P. Ravn. A calculus of duration. *Information Processing Letters*, 40(5):269–276, 1991.

[17] Z. Chaochen and M. R. Hansen. *A Formal Approach to Real-Time Systems.* Monograph in Theoretical Computer Science, an EATCS Series, Springer, 2004.