2014 43rd International Conference on Parallel Processing Workshops

Hierarchical Programming Language for Modal Multi-Rate Real-Time Stream Processing Applications

Stefan J. Geuns
University of Twente
stefan.geuns@utwente.nl

Joost P.H.M. Hausmans
University of Twente
joost.hausmans@utwente.nl

Marco J.G. Bekooij
NXP Semiconductors/University of Twente
marco.bekooij@nxp.com

***Abstract*—Modal multi-rate stream processing applications with real-time constraints which are executed on multi-core embedded systems often cannot be conveniently specified using current programming languages. An important issue is that sequential programming languages do not allow for convenient programming of multi-rate behavior, whereas parallel programming languages are insufficiently analyzable such that deadlock-freedom and a sufficient throughput cannot be guaranteed.**

In this paper a programming language is proposed by which a sequential specification of the behavior of an application can be nested in a concurrent specification. Multi-rate behavior can be conveniently expressed using concurrent modules which have well-defined, but restricted interfaces. Complex control behavior can be expressed in the sequential specification of the body of a module. The language is not Turing complete such that a Compositional Temporal Analysis (CTA) model can be derived. It is shown that the CTA model can be used despite the presence of control statements and that the composition of black-box components is possible. Algorithms with a polynomial time complexity can be used to verify whether throughput and latency constraints are met and to determine sufficient buffer capacities.

A Phase Alternating Line (PAL) video decoder application is used to demonstrate the applicability of the presented language and analysis approach.

I. Introduction

Embedded real-time stream processing applications often process data at different rates. For example, a PAL decoder simultaneously processes a video and an audio stream. The video decoder processes data with a different rate than the audio decoder. The processing of both streams is preferably specified in a single description of the application. Such a description can be specified in a sequential language, which requires parallelization to fully exploit multi-core systems, or in a parallel language. However, neither of these programming languages allows for a convenient specification of multi-rate stream processing applications, while maintaining analyzability of real-time constraints.

A sequential programming language is not sufficiently expressive to specify multi-rate behavior elegantly. This can result in the programmer being forced to include the schedule of functions in the program specification. However, it can be cumbersome to find this schedule and the schedule can be very long. In contrast, parallel languages allow a very convenient specification of multi-rate behavior. However, there is a trade-off between the expressivity and analyzability of a parallel language. In parallel languages which are Turing complete [1], [2], properties such as deadlock-freedom are undecidable in general. Analysis for parallel languages in which the expressivity is restricted [3], [4] is decidable. However, it must be verified whether a uniquely defined behavior is specified, which requires algorithms with an exponential time complexity. Furthermore, it can be cumbersome to mold applications such that they can be specified in the language.

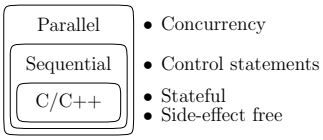


Fig. 1: Hierarchy in the OIL language

A real-time application has temporal constraints and it should be verified whether these constraints are satisfied. This requires the existence of a corresponding concurrent temporal analysis model. Because it is hard to guarantee that such a model is correct, approaches exist in which a temporal analysis model is derived automatically [5], [6]. However, these approaches do not allow for a convenient description of multi-rate behavior because they are based on sequential languages. The approach from [7] is based on a parallel language but specifying an application such that it fits in the language is challenging and the verification of real-time constraints has an exponential time complexity. The limitations of existing approaches justify the definition of a language in which both control and multi-rate behavior can be conveniently specified without compromising on analyzability.

In this paper we introduce a hierarchical variant of the experimental programming language OIL, in which multi-rate behavior can be conveniently specified and from which a corresponding conservative Compositional Temporal Analysis (CTA) model [8] can always be automatically derived. Using the derived CTA model buffer sizes can be determined such that throughput and latency constraints can be met. Furthermore, it allows for the inclusion of black-box components which have interfaces that define maximum rates and delays.

Hierarchy in OIL is achieved by having a parallel specification on top of a sequential specification, as is illustrated by Figure 1. The unit of concurrency is called a module (outer layer in the figure). The body of each module is described as a sequential specification (middle layer) which can be automatically parallelized. Control statements, such as *if*-statements and *while*-loops, are allowed in the sequential specification such that modes of an application can be specified. Pointers, dynamic memory allocation and recursion are not allowed, making the language not Turing complete. Because of the existence of a sequential schedule, the satisfaction of properties, such as throughput and deadlock-freedom, is decidable. Control statements are not allowed in the parallel specification because verifying the satisfaction of these properties would become undecidable in general. OIL is a coordination language in which C/C++ functions can be included such that existing code can be reused (inner layer). These functions can have state, but must be side-effect free and are not parallelized.

From each valid OIL program a corresponding CTA model can be derived, despite the presence of *while*-loops with unknown iteration bounds. From each module in an OIL

1530-2016/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPP.Workshops.2014.66

program a corresponding CTA component can be derived. These CTA components can be composed to derive a model of the complete application. A distinctive feature of the CTA model is that it allows for independent implementability and associative composition. This enables the incremental design of parts of an application, for example libraries. Throughput analysis and buffer sizing algorithms on the CTA model have a polynomial time complexity, allowing for efficient analysis of throughput and latency constraints and for determining sufficient buffer sizes for a given throughput.

The remainder of this paper is organized as follows. In Section II related work is discussed. In Section III we present the basic idea behind our approach. Section IV discusses the proposed hierarchical programming language. Section V introduces the CTA model and the derivation of a CTA model from an OIL program. Section VI illustrates the applicability of the presented approach using a PAL decoder. Finally, Section VII concludes this paper.

II. Related Work

Several programming languages have been proposed for the programming of multi-core systems. They can be classified as either parallel languages or sequential languages, or a mix of both.

The main advantage of parallel languages, or a mix of a sequential and parallel language, is that they allow for explicit parallelism and for a convenient specification of multi-rate behavior. However, for parallel languages which are Turing complete, such as Communicating Sequential Processes (CSP) [1], G for LabVIEW [9] and the Discrete-Event language for PTides [10], it is undecidable in general to verify whether an application is deadlock-free and whether it can execute in bounded memory.

Functional languages such as Haskell [11] are implicitly concurrent and require lazy-evaluation. Expressions are only evaluated when their value is required to complete another expression. Instead of loop structures they employ recursion to denote repetition. However, whether a program with unrestricted recursion satisfies temporal constraints cannot be verified in general.

Restricted parallel languages have been introduced to overcome undecidable properties being present. For example the synchronous languages [3], [4] limit expressivity by having the synchrony hypothesis which states that in the language every function takes zero time and all updates are only visible at global ticks. Next to decidability of the analysis, this also ensures a deterministic behavior of an application. The main disadvantage is that it must be determined whether a unique functional behavior is defined by a program, which requires an algorithm with a exponential time complexity.

The Giotto approach [12] is related to the synchronous languages, but instead of a zero-delay time step, the logic execution time of each component and a valid concurrent schedule are specified by the user. This simplifies analyzing whether a Giotto program has a unique functional behavior, but it can be cumbersome to specify a schedule.

The StreamIt language [13] is based on the Synchronous Dataflow (SDF) model [14]. Temporal analysis for SDF models is decidable. However, exact analysis algorithms to verify the satisfaction of temporal constraints have an exponential time complexity. Furthermore, arbitrary conditions cannot be specified and therefore StreamIt is in general unsuitable for the specification of modal multi-rate systems.

A restricted variant of the CAL actor language [15] is used to generate a Scenario Aware Dataflow (SADF) model in [7]. This model allows for the description of switching behavior in a Finite State Machine (FSM). However, temporal analysis has an exponential time complexity and it can be hard to refactor an application such that it can be described in the CAL language.

In contrast to parallel languages, it is relatively easy to allow control behavior in a sequential language because by definition a sequential language specifies a valid static order execution schedule. Furthermore, if pointers, dynamic memory allocation and recursion are excluded, an analysis model can be derived where deadlock-freedom is decidable and satisfaction of temporal constraints can be verified. However, specifying multi-rate behavior can be very cumbersome, as will be shown in Section III, and parallelization has to be done to efficiently utilize multi-core systems. Also deriving an analysis model automatically for all input programs is difficult in general. For languages which are not restricted, such as OpenMP [16] and Pthreads extensions to C++ [2], deriving an analysis model is even impossible in general.

In the approach from [6] Static Affine Nested Loop Programs (SANLPs) are automatically parallelized and in [17] it is shown that a corresponding Cyclo-Static Dataflow (CSDF) model can be derived. However, arbitrary conditional behavior cannot be expressed in SANLPs.

The P^3L approach [18], [19] allows a hierarchical mixture of a sequential and parallel specification similar to our approach. The sequential specification is written in a host language, such as C. The parallel specification is restricted to a set of templates. A disadvantage is that parallelism cannot be extracted from parts with control behavior. Furthermore, analyzing temporal properties is restricted to applications where control behavior is hidden.

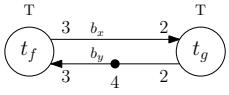
The Hume [20] language also consists of a hierarchical specification where each hierarchical level extends the expressivity of lower levels. The language also allows mixing a parallel and sequential specification. Increasing the expressiveness however makes it impossible to analyze temporal properties in general.

The OIL language presented in this paper also has hierarchy in the form of a parallel specification nesting a sequential specification. What makes the approach unique is that the sequential specification allows for control statements whereas the parallel specification does not allow for control statements, such that the derived temporal analysis model is decidable. The presented approach extends the approach in [5] with the parallel specification of modules to enable a convenient specification of multi-rate behavior. Instead of generating a dataflow model, as done in [5], a corresponding CTA model [8] is derived in which the CTA components correspond elegantly with the modules introduced in OIL. The interface of a component in the CTA model is based on maximum rates and latencies, which allows for composition of such CTA components. The CTA model is used to verify deadlock-freedom and for the computation of sufficient buffer capacities with an algorithm with a polynomial time complexity.

As with the CTA model, the Real-Time Calculus (RTC) [21] method allows for the compositional analysis of real-time systems. However, the use of buffers with a finite capacity results in cyclic dependencies. Such cyclic dependencies result in an exponential worst-case time complexity of the analysis algorithms while analysis algorithms for the CTA model have a polynomial time complexity. Furthermore, buffer sizing for applications with loops which result in inter-iteration dependencies has not been addressed.

III. Basic Idea

In this section we first discuss the issues with supporting multi-rate behavior in a sequential programming language.



(a) Task graph

```
int x[6], y[6];

init(out y[0:3]);
loop{
  f(out x[0:2], y[0:2]);
  g(out y[4:5], x[0:1]);
  f(out x[3:5], y[3:5]);
  g(out y[0:1], x[2:3]);
  g(out y[2:3], x[4:5]);
} while(1);
```

(b) Sequential program

```
mod_seq A(out int a, int b){
  loop{
    f(out a:3, b:3);
  } while(1);
}

mod_seq B(out int c, int d){
  init(out c:4);
  loop{
    g(out c:2, d:2);
  } while(1);
}

mod_par C(){
  fifo int x, y;

  A(out x, y) || B(out y, x)
}
```

(c) Parallel program

Fig. 2: Rate conversion in a cyclic task graph

Next to that, we show the basic idea behind our temporal analysis approach in the presence of multi-rate behavior.

A. Multi-Rate Specification

A program contains multi-rate behavior if the sample rate of data is changed. The task graph in Figure 2a shows an example of such multi-rate behavior. Task t_f first reads three values and T time later writes three values. Task t_g reads only two values and writes two values T time later. Both tasks execute data-driven, meaning they execute when sufficient data is available at their inputs. Because both tasks read a different number of values, task t_g must execute $\frac{3}{2}$ as often as task t_f. The dot labeled 4 indicates that four initial values are available for task t_f to read.

Writing such a cyclic application as a sequential program can be difficult as often the only option is to specify the complete schedule until the initial state is reached again. This is illustrated by the sequential program in Figure 2b where a schedule is shown for the task graph in Figure 2a. The notation *x[0:2]* denotes that locations 0, 1 and 2 of array *x* are read. Analogously, *out x[0:2]* denotes that locations 0, 1 and 2 are written. The four initially available values are written by the *init* function. The *loop-while* statement in this example indicates that the execution of all functions inside is repeated indefinitely. Finding such a schedule can be difficult and the schedule can be very large. Therefore, expressing multi-rate behavior in sequential programs can be very inconvenient.

If the same application is specified as a concurrent program it becomes a lot simpler to express multi-rate behavior, as is shown in Figure 2c. Every module specified by a *mod_seq* block specifies a block which can execute concurrently with other modules. The module *C* specifies the parallel modules *A* and *B*. Communication between these sequential modules is performed via First-In First-Out (FIFO) buffers *x* and *y*. Writing three values to *x* is notated as *out x:3*, reading three values as *x:3*. As can be seen in the figure, only one function call is made to both functions *f* and *g* and thus the schedule of these functions does not have to be encoded in the program.

B. Real-Time Analysis Model Derivation

The CTA model is used to verify whether the real-time constraints of an OIL program are met and to determine sufficient buffer capacities. A CTA model consists of components, depicted as rectangles on the right in Figure 3 and connections, depicted as arrows. Data is transferred periodically between components over connections at a given rate. A connection can delay a transfer by a pre-defined amount of time.

On the left in Figure 3 a graphical sketch of an OIL program is shown. Data is produced by a source *src* at a rate f_1,

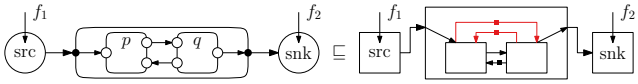


Fig. 3: Refinement of temporal analysis

is processed by a module, depicted by the outer rectangle, and transferred to a sink *snk*, which consumes data at a rate f_2. Processing is done in two *while*-loops, represented by the inner rectangles. The number of iterations of these loops is given by the parameters p and q respectively.

On the right in Figure 3 the corresponding CTA model is shown. This model is constructed from an OIL program such that for every module and every *while*-loop a CTA component is extracted. CTA components extracted from OIL modules nest CTA components corresponding to *while*-loops. Thus the topology of a CTA model is equivalent to an OIL program. However, a CTA component is not parameterized and is always active at a given periodic rate. Therefore, *while*-loops cannot be directly modeled as CTA components. In the remainder of this section we present the intuition behind the abstraction made to model a *while*-loop as a CTA component.

To show that the abstraction of a parameterized *while*-loop to a CTA component with periodic rates is allowed, it must be guaranteed that every source and sink can execute strictly periodic. To ensure a bounded time between accesses to a source or sink, they must be accessed in every *while*-loop iteration [5], [22]. In the CTA model this implies that every component corresponding with a *while*-loop has an access to every source and sink. Thus on the right in Figure 3, the two nested components access both *src* and *snk* as illustrated by the connections.

Because we only consider temporal constraints and not the functional behavior, it is irrelevant which component is active. We now illustrate that independent of which component is active, the temporal constraints imposed by periodic sources and sinks are met. Two cases can be considered: a component is repeatedly active or a transition occurs to the next component. For both cases it must hold that the time between source or sink accesses is within the period of the source or sink.

If a component is repeatedly active, periodic accesses of sources and sinks can be verified by imposing a periodic rate on this component. Periodic rates are inherent in the CTA model and algorithms exist to verify whether they can be met [8]. An over-approximation is made in this step because it is assumed in the model that a component is always active.

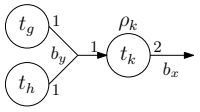
Also for a transition between two components it must hold that sources and sinks are accessed within their period. This is enforced by additional connections between components accessing sources and sinks. In Figure 3 the two connections at the top of the figure enforce this period. The delay on these connections is chosen such that the transition time is one period. These connections model the worst-case behavior, which is that a transition occurs after every execution of all statements in a *while*-loop.

IV. Hierarchical Language Specification

In this section we present our hierarchical programming language OIL. The hierarchical structure of the language is shown in Figure 1. OIL is a coordination language, meaning that it coordinates computation implemented in functions. These functions can be implemented in a different language such that existing single-core compiler infrastructure can be used. A similar approach is taken by for example LabVIEW [9], CSP [1] and StreamIt [13]. As illustrated in Figure 1, OIL coordinates C or C++ functions. We restrict C/C++ functions to

```
mod_seq M(out int x){
  if(...){ y = g(); }
  else{ y = h(); }
  k(y, out x:2);
}
```

(a) OIL module



(b) Task graph

Fig. 4: Parallelization of a sequential OIL module

the class of side-effect free functions in which state is allowed. A side-effect free function can be reordered with respect to other functions, if there are no data-dependencies between these functions to prevent this. Various tools exist to verify whether a function is side-effect free [23], [24], [25].

Next to coordinating C/C++ functions, hierarchy is also present in OIL itself. At the parallel level modules are specified concurrently and control behavior is not allowed such that analysis is possible. Modules contain either other concurrent modules or a sequential specification. In a sequential specification control statements are allowed, but opposed to C, pointers, dynamic memory allocation and recursion are not allowed.

From an application specified in OIL, parallelism is extracted in the form of a task graph with the method of [5]. A task is created for every function and assignment statement. If a function or assignment is guarded by an *if*-statement, the corresponding task is executed unconditionally, but the function or assignment in the task remain guarded. An example OIL program is shown in Figure 4a. Functions g and h are executed conditionally, but their corresponding tasks t_g and t_h are executed unconditionally. For every variable a Circular Buffer (CB) is created. Such a CB is a generalization of a FIFO buffer where multiple producers and consumers are allowed [26]. In the example variable y is converted to buffer b_y. Every assignment statement assigning a value to this variable is converted to a producer in the corresponding CB. Every function or assignment statement reading from this variable is transformed to a consumer in the CB. For every task the OIL compiler generates a sequential code fragment which can be compiled using an appropriate single core compiler. By default C++ code is generated for every task. This C++ code can be compiled with a C++ compiler, such as GCC [27].

Figure 5 shows the syntax of the core of the OIL language. Parallel statements are delimited by the $\|$-symbol and sequential statements by a semi-colon. Currently, the language does not include its own type system. Because OIL code is compiled into C or C++ code, verification of types is left to the C/C++ compiler. To simplify discussions in this paper we only consider scalar variables and not arrays. Note that the OIL language is not Turing complete because dynamic memory allocation and recursion are not supported such that temporal analysis remains decidable and a corresponding CTA model can be derived. Furthermore, OIL is functionally deterministic, meaning that executing an OIL program twice on the same input trace results in the same output trace. In the sections below we discuss the features and requirements of the language in more detail.

A. Modules

Modules describe the concurrent structure of an application. A module is denoted by either *mod_par* or *mod_seq*. A module denoted by *mod_par* contains instantiations of other modules, which execute concurrently. A module denoted by *mod_seq* contains a sequential specification, which can be parallelized. In this sequential specification, control statements such as *if*-statements and *while*-loops coordinate data exchanged between functions.

Program	$\mathcal{P}$	::=	$\mathcal{M}^\star$
Modules	$\mathcal{M}$	::=	mod_par $A(\mathcal{R}^\star)\{\ \mathcal{G}^\star\mathcal{L}^\star\mathcal{N}\ \}$ | mod_seq $A(\mathcal{R}^\star)\ \{\ \mathcal{V}^\star\mathcal{S}^\star\ \}$
Buffers	$\mathcal{G}$	::=	fifo $T\ x$; | source $T\ x = F()$ @ n Hz; | sink $T\ x = F()$ @ n Hz;
Latency	$\mathcal{L}$	::=	start x n ms after y; | start x n ms before y;
Streams	$\mathcal{R}$	::=	out $T\ r$ | $T\ r$
Module calls	$\mathcal{N}$	::=	$A(\mathcal{B}^\star)$ | $\mathcal{N} \parallel \mathcal{N}$
Parameters	$\mathcal{B}$	::=	out r | r
Variables	$\mathcal{V}$	::=	$T\ x$
Statements	$\mathcal{S}$	::=	x = e; | $F(\mathcal{A}^\star)$; | if(e)$\{\ \mathcal{S}^\star\ \}$ else $\{\ \mathcal{S}^\star\ \}$ | if(e)$\{\ \mathcal{S}^\star\ \}$ | switch(e) $\mathcal{C}^\star$ default $\{\ \mathcal{S}^\star\ \}$ | loop $\{\ \mathcal{S}^\star\ \}$ while(e)
Cases	$\mathcal{C}$	::=	case $n\ \{\ \mathcal{S}^\star\ \}$
Arguments	$\mathcal{A}$	::=	e | out x | out r | out r:n
Expression	e	::=	m | x | r | r:n | $F(e^\star)$ | $e\ \mathcal{O}\ e$
Operator	$\mathcal{O}$	::=	* | \ | + | -

Fig. 5: Core syntax of the OIL language, T represents a type name, m a number, n a positive number, A a module name, F a function, x and y variables and r a stream

Modules denoted by *mod_par* can communicate using FIFO buffers, which periodically transfer data between modules. Only one module can write to a FIFO, but multiple modules can read from it. All modules reading from a FIFO read the same data. FIFOs can be passed as arguments to modules. These arguments are called streams to distinguish them from function arguments, which have a constant value during the execution of the function.

Because values are read and written concurrently from and to streams, it must be defined when new values are made available to other modules and when values are no longer required. A new value is made available from an input stream at the end of every *while*-loop iteration. In module A in Figure 2c a function *f* is reading a new value from stream *b* every execution of *f*. Analogously, values written to output streams are made available to other modules at the end of every *while*-loop. Output streams have to be written every loop iteration. If they are written by multiple statements during one loop iteration, only the last value written by the last statement will become visible to other modules. Input streams do not have to be read or can be read multiple times during a loop iteration, in which case the same value is read repeatedly.

Multiple values can be written to an output stream or read from an input stream simultaneously using the colon notation. All of the written values become visible to other modules, but they can be observed individually. Multi-rate behavior can be easily described using this colon operator. An example of such multi-rate behavior is shown in Figure 2c. Module A writes three values per *while*-loop iteration whereas module B only reads two values. To ensure consistent behavior, meaning the same number of values are produced and consumed, module B will execute with a 1.5x higher rate than module A.

B. Sources and Sinks

Sources and sinks provide means to communicate with the environment of an OIL program. Sources sample the environment and produce samples to a program, sinks transfer samples from a program and to the environment. Since the environment is by definition in parallel with a program, sources and sinks also execute in parallel with our program. They execute time-triggered, and as a consequence execute time-triggered with an interval defined by the programmer.

Sources and sinks are a special case of modules because they are specified using a single function which implements the low-level details of the communication with the environment.

```
mod_par A(int a,
          out int b){
  fifo int z;

  B(a, out z) ||
  C(a, z, out b)
}
```

```
mod_par D(){
  source int x = src() @ 1 kHz;
  sink int y = snk() @ 1 kHz;
  start x 5 ms before y;

  A(x, out y)
}
```

Fig. 6: Example of a program with a source and a sink

Communication with modules however, also occurs via CBs with FIFO semantics to preserve the ordering of written values. Periodicity of sources and sinks also imposes temporal constraints on any module communicating with them. For example in Figure 6 the source src and the sink snk execute with a rate of 1 kHz, thus also module A must on average be able to execute at 1 kHz.

Between sources and sinks latency constraints can be specified using the *start. . .after* and *start. . .before* constructs. They enforce that a source or sink has to be started a pre-defined amount of time after respectively before another source or sink. Thus they represent latency constraints between sources and sinks. An example of such a latency constraint is given in Figure 6 where sink snk must start before $5ms$ have passed since the start of source src. Because src and snk communicate via module A, $5ms$ after a sample enters the system at src the processed sample is visible at snk.

V. Temporal Analysis

In this section we first describe the CTA model which is used to verify whether the real-time throughput and latency constraints are met. Next, we show that a CTA model can be derived from a sequential OIL module and then we describe the derivation of a CTA model for parallel OIL modules.

A. The CTA Model

A CTA model is a graph of components and directed connections. A component w in the CTA model can be defined as $w = (P, \hat{r}, C, \gamma, \epsilon, \phi)$. Here P is the set of ports of the component. Each port can transfer data at a maximum rate $\hat{r}$, with $\hat{r} : P \rightarrow \mathbb{R}^+$. The actual transfer rate of a port is dependent on ports connected to it and is defined as $r : P \rightarrow \mathbb{R}^+$. For every port $p \in P$ it must hold that the actual transfer rate is at most the maximum transfer rate: $r(p) \leq \hat{r}(p)$.

Connections between ports of a CTA component are defined by the set C, with $C \subseteq P \times P$. A connection directed from port p to port q is denoted as $(p, q) = c_{pq}$, with $c_{pq} \in C$. In the CTA model periodic event sequences are used to express constraints. These periodic event sequences are specified using an offset and a distance between events. CTA connections can delay streams and thus change the offset. A delay is either constant or rate dependent, meaning the delay depends on the distance between events. The constant delay is defined as $\epsilon : C \rightarrow \mathbb{R}$, and the rate dependent delay as $\phi : C \rightarrow \mathbb{R}$. A connection can have a different rate on its input and output ports. This transfer rate ratio is specified by $\gamma : C \rightarrow \mathbb{R}^+$. The time that data is delayed over a connection c_{pq} is $\Delta(c_{pq}) = \epsilon(c_{pq}) + \frac{\phi(c_{pq})}{r(p)}$.

CTA components can be composed and a composition of CTA components and CTA connections is again a CTA component. A composition must be consistent, meaning that all transfer rates are below the corresponding maximum transfer rates and that data arrives in time on every port. Data can be delayed over a connection. If a sequence of connections forms a cycle, data can be delayed a positive amount of time, meaning it arrives too late at the ports on the cycle. An algorithm to verify whether a CTA composition is consistent is given in [8]. This algorithm has a polynomial time complexity. Next to a binary answer whether a model is consistent, the consistency algorithm also returns the maximal achievable transfer rates for every port.

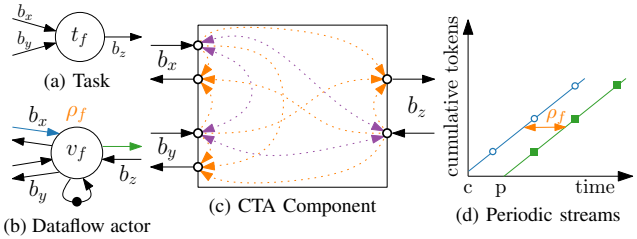


Fig. 7: Construction of a single-rate CTA component

B. Temporal Analysis of a Sequential Module

In this section we describe the derivation of a CTA model from a sequential module. First the modeling of functions and assignments as a CTA component is described. Then we describe the modeling of *while*-loops and finally streams are modeled using a set of connections.

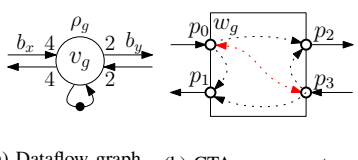
1) Functions and Assignments: When parallelizing a sequential OIL module, a task is created from every function and assignment. Before such a task can be modeled as a CTA component, an intermediate abstraction is made in the form of a dataflow actor, as described in detail in [8]. This abstraction is repeated here for convenience. We first show the derivation of such an actor from a task. Then we show how a corresponding CTA component can be derived from this actor. We first show this for a single-rate actor and then for a multi-rate actor. Finally, we describe the modeling of buffer capacities in the CTA model.

Every task is modeled as an SDF actor. An example task is shown in Figure 7a. This task reads from buffers b_x and b_y and writes to buffer b_z. The corresponding dataflow actor is shown in Figure 7b. The firing duration of this actor equals the response time of the task. For every buffer two oppositely directed edges are connected to the dataflow actor. The first edge represents reading data from or writing data to the buffer. The second edge represents releasing space back to the buffer. An actor can only fire if sufficient tokens are available on all incoming edges and consumption of these tokens is atomic.

From the derived dataflow actor a corresponding CTA model can be created. The CTA component in Figure 7c corresponds with the actor in Figure 7b. A port is added to this component for every incoming and outgoing edge at the actor. Every edge is modeled as a connection in the CTA model. Because consumption of tokens is atomic in an actor, there is no delay between the consumption of tokens on different edges. Therefore, a connection with a delay of zero is added between all input ports such that all input ports start at the same time. In the example CTA component these connections are shown in purple.

The firing duration of an actor represents the time between consumption and production of tokens and thus enforces a maximum rate. Figure 7d shows a schedule of the consumption and production of tokens by actor v_f. The blue dots indicate the consumption of tokens on the blue incoming edge denoted b_x of actors v_f. The green squares represent the production of tokens on the green edge. The time difference between these dots is the firing duration of the actor.

As presented in [8], the arrival of tokens represent events and the cumulative number of produced/consumed tokens per



(a) Dataflow graph (b) CTA component

	ϵ	ϕ	γ
(p_0, p_1)	ρ_g	3	1
(p_0, p_2)	ρ_g	2	2/4
(p_0, p_3)	0	0	2/4
(p_3, p_0)	0	0	4/2
(p_3, p_1)	ρ_g	3/2	4/2
(p_3, p_2)	ρ_g	1	1

(c) Delays and transfer rate ratios

Fig. 8: Construction of a multi-rate CTA component

time unit can be bounded by a strictly periodic sequence of events. This event sequence is characterized by its rate. In the CTA model a maximum rate is defined for every sequence. The maximum rates for the sequences in Figure 7d are $\frac{1}{\rho}$. This response time also imposes a delay between the consumption and production sequences. This delay is enforced by adding a connection between every input and output port of the CTA component. In the example CTA component in Figure 7c these connections are in orange.

If an actor has multi-rate behavior it consumes a different number of tokens than it produces. This change in tokens is enforced by the transfer rate ratio of a CTA connection. The transfer rate ratio is defined as $\gamma = \frac{\pi}{\psi}$, where π represents the number of produced tokens and ψ the number of consumed tokens. In Figure 8a an example is shown of an actor with such multi-rate behavior. Figure 8b shows the corresponding CTA component and Figure 8c the transfer rate ratios corresponding with the connections in the CTA component. Because actor v_g consumes four tokens and produces two tokens the transfer rate ratio on connections (p_0, p_2) and (p_0, p_3) is $\frac{2}{4}$. On the oppositely directed connections the production and consumption are also reversed, thus the transfer rate ratio is $\frac{4}{2}$.

Because the periodic event sequences are defined in the number of producer or consumed tokens, there is a difference between the production and consumption sequences. This difference causes the time between the sequences to be dependent on the required rate and can be defined as $\phi(c) = \psi - \frac{\psi}{\pi}$ for a connection c. In Figure 8c also the delays on the connections in the CTA component from Figure 8b are shown. For example on the connection (p_0, p_2) the rate-dependent delay is $4 - \frac{4}{2} = 2$.

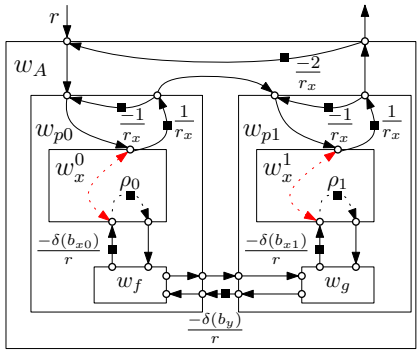
The throughput of an application is affected by the capacity of buffers. This buffer capacity is modeled in a dataflow graph as the number of initial tokens on an edge representing a buffer. Initial tokens allow the consuming actor to start earlier. If there are δ initial tokens the actor can start $\frac{\delta}{r}$ earlier. Therefore, on the corresponding CTA connection there is a delay of $\frac{-\delta}{r}$.

2) While-Loops: Because every *while*-loop induces a different execution rate on the statements located in its body, they are modeled as separate CTA components. Every component of which the corresponding function or assignment is placed in the body of a *while*-loop are made sub-components of the component corresponding with the *while*-loop. For example in Figure 9a function f is in the body of the first *while*-loop. In Figure 9b the corresponding component w_f is a sub-component of component w_{p0}.

For every variable which is written or read in a different *while*-loop, two ports are added to the component corresponding with this loop. Connections expressing the buffer constraints are added similarly as above, except there are intermediate ports. An example is again shown in Figure 9. Variable y is written by an assignment in the first *while*-loop and read in the second loop. To components w_{p0} and w_{p1} two intermediate ports and connections are added such that buffer b_y between is modeled. These are shown at the bottom.

```
mod_seq A(int x){
  loop{
    y = f(x);
  } while(...);
  loop{
    g(x, y);
  } while(...);
}
```

(a) OIL program



(b) CTA model

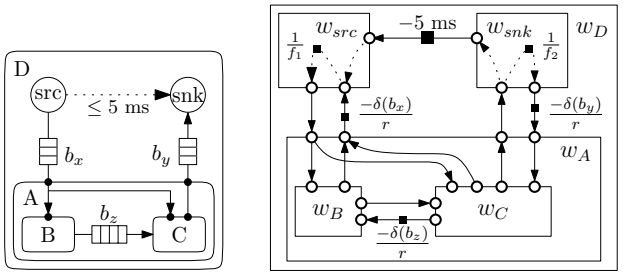
Fig. 9: OIL program with multiple stream accesses and the corresponding CTA model

3) Streams: A stream can be read or written by statements in multiple *while*-loops. Each of these functions can execute a different number of times and therefore read or write a different number of values, depending on the termination condition of the loop. An input stream is implemented using a task which distributes values read from a stream to these functions using separate buffers. For an output stream a task combines values from separate buffers. Each such task is modeled using a component for every buffer. In Figure 9 there is a stream access in two *while*-loops and therefore two components, w_x^0 and w_x^1 are added to model this distribution of combining of values.

In the model, every stream s must be read or written periodically with a period P_s. Consequently, a stream must have a rate of $r_s = 1/P_s$. Because multiple components model one stream, this periodicity constraint must be distributed over all these components. Additional ports and connections are now added such that this periodicity constraint is enforced.

On every component representing a *while*-loop or a module, an input and output port are added for each stream s. The input port receives the constraint on the rate from less deeply nested components via a connection. The output port is used to distribute the rate to other components such that components can be linked together. A connection is added from the input port to the component where the corresponding statement is the first to access s in the order defined by the sequential program. From the component where the corresponding statement is the last to access s a connection is added to the output port. This is illustrated in Figure 9b where two ports are added to w_{p0} and w_{p1} for stream x. In the OIL program the first statement to access x is function f. Therefore, component w_x^0, which corresponds with the *while*-loop around f, is connected to the input port. Function g is the last statement to access x and therefore a connection is added from w_x^1 to the output port.

Statements in *while*-loops have to execute with a rate r_s to keep up with the corresponding stream. If such statements are executed, there are two options. Either the same statements in the same *while*-loop are repeated, or the loop condition is false and statements in the next *while*-loop are executed. In both of these cases the next stream access must be in $P_s = 1/r_s$ time. Therefore, a connection with a delay of $1/r_s$ is added from a component representing a stream access to the next component. In Figure 9b this results in a connection from w_x^0 to w_x^1. Because w_x^1 has a different parent component than w_x^0, this connection is split in two connections via the output port of w_{p0}. The delay of $1/r_x$ is on the first connection from w_x^1 to this output port.



(a) Graphical view (b) CTA model

Fig. 10: Graphical view of the program in Figure 6 and the corresponding CTA model

This last connection, from w_x^1 to the output port of w_{p1}, also imposes a delay between iterations of the same *while*-loop. However, because a delay only specifies a minimum delay, a maximum delay and thus a strictly periodic execution, must be enforced separately. This maximum delay can be enforced by adding a connection from the output port back to the input port. On this connection the delay is equal to the negated sum of all delays from the input to the output port. For w_{p0} and w_{p1} in Figure 9b this negated sum is $-1/r_x$. In w_A there are two delays on the path from input to output, thus the total delay is $2/r_x$. This results in a negated delay of $-2/r_x$ on the connection from output to input.

C. Temporal Analysis of Parallel Modules

In this section we show that a CTA component can be derived from parallel OIL modules. Communication between modules in OIL is performed via FIFO buffers. Also these buffers affect the minimum throughput of an application, ie. the inverse of the maximum rate, and must therefore be of a sufficient capacity.

Every instantiation of a module is converted to a CTA component. If the instantiated module is a sequential module, the derivation from the previous section is used. Otherwise, the following derivation is used. For every stream into or out of a module two ports are added to the component. The maximum rate of these ports is infinite as they are modeling artifacts and are not not present in the implementation. The actual rate of these ports can be determined with the consistency algorithm for CTA. For every input stream a connection is added from the first of these ports to all components which have the same stream as input of their corresponding module. Also a reverse connection is added from the sub-component to the second port. Module A in the example in Figure 6 has an input stream x and output stream y. This is shown graphically in Figure 10a. In the corresponding CTA model in Figure 10b two pairs of connections are added to model the input stream x to w_B and w_C and the connections are reversed for the output stream y.

Modules can communicate via FIFO buffers passed as arguments to module instantiations. For each FIFO two oppositely directed connections are added which link the ports corresponding with the read and write accesses to this FIFO. A rate dependent delay $\frac{-\delta}{r}$ is added on either the first or second connection, depending on whether the stream is an input respectively output stream. The value of δ corresponds with the capacity of the FIFO the connections represent. In Figure 10a module B writes to FIFO b_z and module C reads from b_z. In the corresponding CTA model in Figure 10b components w_B and w_C both have two ports modeling accesses to b_z and two connections between these ports.

Periodic sources and sinks are also converted to CTA components. Such a component has two ports modeling data output and input respectively. A connection is added between these two ports with a constant delay set to the inverse of the frequency of the source or sink. In Figure 6 components are added for a source defined by function *src* and a sink, defined by function *snk*. Their frequency is f_1 and f_2, therefore the delay is $\frac{1}{f_1}$ and $\frac{1}{f_2}$ respectively. Communication between a source or a sink and a module is modeled similarly as communication between modules with FIFO buffers. In Figure 10 the communication between source src and module A and between A and sink snk is modeled by four connections. The buffer capacity is modeled with a delay of $\frac{-\delta(b_x)}{r}$ and $\frac{-\delta(b_y)}{r}$ respectively.

Latency constraints between sources and sink can also be specified in an OIL program and should therefore be included in the CTA model. For a latency constraint, a single connection is added between the two components corresponding to the sources and sinks between which the constraint should hold. The delay on this connection corresponds with the time of the latency constraint. In the example from Figure 6, a latency constraint of 5 ms is added between a source *E* and a sink *F*. A connection is added in Figure 10b between components *src* and *snk* with a delay of 5 ms, which corresponds to the latency constraint.

VI. Case-Study

In this section we describe a PAL video decoder as an OIL program with modules that express the inherent multi-rate behavior of the application. Furthermore, we describe the derivation of the corresponding CTA model. The OIL language is implemented as an input for our experimental multi-processor compiler. The PAL decoder is implemented on a multi-core system, which is described in [28].

Figure 11 shows the for this paper relevant implementation of such a PAL decoder as a hierarchical OIL program. In a PAL decoder the broadcasted RF signal is received by an analog RF front-end, where it is sampled periodically at a rate of 6.4 MS/s such that both the video and audio bands are received. This signal is split by the *Splitter* module into video and audio bands. The audio signal is first mixed to zero (module *Mix_A*) and then the video signal is removed by a low-pass filter. Simultaneously, this signal is downsampled with a factor 25 to preserve only the audio band (module *SRC_A*). From the video signal the audio signal is removed by a low-pass filter (module *LPF_A*) and resampled with a factor $\frac{10}{16}$ (module *SRC_V*). This factor is required by the black-box *Video* module which processes samples at a rate of 4 MS/s. Also the *Audio* module is a black-box module and downsamples the audio signal again by a factor 8 such that samples can be sent to a sink of 32 kHz. The audio module internally has control behavior, for example to mute the audio output in case of a bad reception. Because video images and audio signal must be in sync, the latency difference between both sinks is defined as zero.

Figure 12 shows the CTA model corresponding with the PAL application. As shown in the OIL program, the $w_{Splitter}$ component contains the two parallel rate conversion modules. All components modeling functions are constructed as discussed in Section V-B and internal connections are omitted for clarity. Components modeling *while*-loops, as well as components modeling streams are hidden. Hiding is a feature of the CTA model where internal ports are removed and only the constraints are preserved.

The latency constraint between both sinks is modeled as a cycle with zero delay. The required rate conversion is left implicit in this figure.

```
mod_seq SRC_A(sample si, out sample so){
  loop{
    LPF(si:25, out so);
  } while(1);
}

mod_seq SRC_V(sample si, out sample so){
  loop{
    resamp(si:16, out so:10);
  } while(1);
}

mod_par Splitter(sample rf, out sample v, out sample a){
  fifo sample mas, mvs;

  Mix_A(rf, out mas) || SRC_A(mas, out a) ||
  LPF_V(rf, out mvs) || SRC_V(mvs, out v)
}

mod_par{
  fifo sample vid, aud;
  source sample rf  = receiveRF()@ 6.4 MHz;
  sink sample screen = display() @ 4 MHz;
  sink sample speakers = sound() @ 32 kHz;
  start screen 0 ms after speakers;
  start screen 0 ms before speakers;

  Splitter(rf, out vid, out aud) ||
  Video(vid, out screen) || Audio(aud, out speakers)
}
```

Fig. 11: PAL decoder fragment

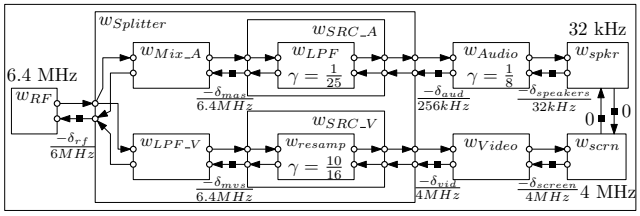


Fig. 12: Fragment of the CTA model of the PAL decoder

VII. Conclusion

In this paper a hierarchical programming language was introduced for the specification of modal multi-rate real-time stream processing applications. In this language a sequential specification of the module behavior is nested in a concurrent specification of communicating modules. Modules enable an intuitive description of multi-rate behavior, whereas the sequential specification allows for a description of control behavior. The proposed language is not Turing complete which enables temporal analysis, despite that concurrency and control behavior can be specified.

A CTA model can always be derived from an OIL program despite the presence of *if*-statements and *while*-loops with unknown iteration bounds. Existing analysis methods for the CTA model can be used despite this control behavior. CTA modeling supports composition of black-box modules such that modules with temporal interfaces can be specified in libraries. The CTA model can be used to determine buffer sizes such that throughput and latency constraints can be met.

The presented OIL programming language is the input language of an experimental multiprocessor compiler. A PAL decoder, in which a video and audio stream are processed at different rates, is used to demonstrate that modal multi-rate behavior can be conveniently specified in OIL and analyzed using a CTA model.

References

[1] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[2] B. Nichols *et al.*, *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[3] G. Berry and E. Sentovich, "Multiclock Esterel," *Correct Hardware Design and Verification Methods*, pp. 110–125, 2001.

[4] N. Halbwachs *et al.*, "Programming and verifying real-time systems by means of the synchronous data-flow language lustre," *IEEE Trans. on Software Engineering*, vol. 18, no. 9, pp. 785–793, 1992.

[5] S. Geuns *et al.*, "Automatic dataflow model extraction from modal real-time stream processing applications," in *Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2013, pp. 143–152.

[6] D. Nadezhkin and T. Stefanov, "Automatic derivation of polyhedral process networks from while-loop affine programs," in *Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. IEEE, 2011, pp. 102–111.

[7] F. Siyoum *et al.*, "Automated extraction of scenario sequences from disciplined dataflow networks," in *Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, 2013, pp. 47–56.

[8] J. Hausmans *et al.*, "Compositional temporal analysis model for incremental hard real-time system design," in *Conf. on Embedded Software (EMSOFT)*. ACM, 2012, pp. 185–194.

[9] H. Andrade and S. Kovner, "Software synthesis from dataflow models for G and LabVIEW," in *Conf. on Signals, Systems & Computers*, vol. 2. IEEE, 1998, pp. 1705–1709.

[10] Y. Zhao *et al.*, "A programming model for time-synchronized distributed real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, pp. 259–268.

[11] S. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[12] T. Henzinger *et al.*, "Embedded control systems development with giotto," in *ACM SIGPLAN Notices*, vol. 36, no. 8. ACM, 2001, pp. 64–72.

[13] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Conf. on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010, pp. 365–376.

[14] E. Lee and T. Parks, "Dataflow process networks," in *IEEE*, May 1995.

[15] J. Eker and J. Janneck, "Cal language report," *University of California, Berkeley, Tech. Rep.*, vol. 3, 2003, http://ptolemy.eecs.berkeley.edu/papers/03/Cal/.

[16] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[17] M. Bamakhrama *et al.*, "A methodology for automated design of hard-real-time embedded streaming systems," in *Design, Automation and Test in Europe (DATE)*, 2012, pp. 941–946.

[18] B. Bacci *et al.*, "P3L: A structured high-level parallel language, and its structured support," *Concurrency: practice and experience*, vol. 7, no. 3, pp. 225–255, 1995.

[19] S. Ciarpaglini *et al.*, "Anacleto: a template-based p3l compiler," in *Parallel Computing Workshop (PCW)*, vol. 97, 1997.

[20] K. Hammond and G. Michaelson, "Hume: a domain-specific language for real-time embedded systems," in *Generative Programming and Component Engineering*. Springer, 2003, pp. 37–56.

[21] K. Lampka *et al.*, "Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems," in *Conf. on Embedded Software (EMSOFT)*. ACM, 2009, pp. 107–116.

[22] S. Geuns *et al.*, "Temporal analysis model extraction for optimizing modal multi-rate stream processing applications," in *Workshop on Software & Compilers for Embedded Systems (SCOPES)*, 2014, to appear.

[23] L. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Copenhagen, 1994.

[24] E. Clarke *et al.*, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[25] A. Rountev and B. Ryder, "Points-to and side-effect analyses for programs built with precompiled libraries," in *Compiler Construction*. Springer, 2001, pp. 20–36.

[26] T. Bijlsma *et al.*, "Circular Buffers with Multiple Overlapping Windows for Cyclic Task Graphs," *Conf. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, vol. 5, no. 3, 2011.

[27] "GNU Compiler Collection," http://gcc.gnu.org, Apr. 2014.

[28] B. Dekens *et al.*, "Low-cost guaranteed-throughput communication ring for real-time streaming mpsocs," in *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2013, pp. 239–246.