Abstracting Distributed, Time-Sensitive Applications

Kyle Liang

November 2023

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Abstract

Distributed and Time-sensitive (DT) cyber-physical systems are challenging to design and develop. When writing systems in conventional languages, programmers struggle to manage the conciseness and complexity of intrusive, cross-cutting concerns caused by a heterogeneous, distributed system with sensing and actuation timing requirements.

This thesis proposes that a system using the dataflow graph as an intermediate representation can reduce barriers-to-entry for programmers and domain experts unfamiliar with designing distributed, time-sensitive applications. We present TTPython: a domain-specific language and runtime system for writing shorter and cleaner code for challenging cyber-physical applications. Its novel timed tagged-token dataflow graph execution model allows programmers to develop applications at a macroprogramming scale while supporting timing specifications such as periodicity and soft deadlines. The programmer uses annotations to guide TTPython in placement of code and system-provided function calls to specify timing requirements while TTPython handles distribution, communication, and coordination between devices. We have evaluated TTPython by comparing it to a best-practice implementation of a 1/10th-scale connected autonomous vehicle application. An in-progress case study on an intercity flooding application will examine how TTPython affects design and system decisions during development. These two case studies will serve as the foundation of a user study in which we ask users to write DT applications for both of these scenarios using either TTPython or Python with a message-broker system.

Dataflow graphs have been applied in various contexts in parallelism and distributed computing, but little work has been focused on token-tagged dataflow graphs. This thesis will show how incorporating time in a token-tagged dataflow graph makes it an effective tool for handling distributed, time-sensitive applications. These advances work towards developing key abstractions that make programming of large arrays of sensors and actuators approachable by non-specialist programmers. The modified dataflow graph can serve as the initial framework towards creating a standard digital distribution service for developing and deploying DT applications across devices shared by many, such as in a smart city.

Contents

Chapter 1

Introduction

Distributed, time-sensitive (DT) applications permeate cyber-physical systems (CPS). Improvements in wireless, embedded, cloud, and networking technologies enable larger, more interconnected applications that measure and control the physical world. For example, autonomous vehicular networks enable signal-free intersections [14] in which vehicles coordinate their trajectories through the intersection to use the space more efficiently and increase throughput. The vehicles must plan these trajectories with millisecond precision to avoid colliding with each other. Large drone swarms must similarly determine their location and plan their path at millisecond or microsecond precision to form spectacular displays like Intel's swarm at the 2018 Winter Olympics in South Korea.

DT applications are difficult to develop and deploy, especially when programmers attempt to reason about synchronized, real-time actions in large-scale distributed systems. Programmers are interfacing with dozens if not hundreds of devices with varying hardware capabilities over large geographic regions. Devices collaborate by providing spatial and temporal data and provide valuable information about the environment they reside in by virtue of their hardware or location, but their physical separation causes many technical challenges. DT applications need to accommodate unexpected runtime behaviors of the code, such as recovery procedures for mitigating hardware, software, or network failure. In general, systems cannot guarantee hard deadlines due to communication failures and delays, so programmers must write plans to recover and adapt when deadlines are missed. Distribution and timing concerns are also cross-cutting and intrusive. Coordinating between multiple devices at the device level is unwieldy. The programmer needs to explicitly call 3rd-party library communication protocols to interface between devices. This leads to potential architectural bugs when the application evolves, as message formats and types are hidden by opaque communication function calls. In traditional languages, programmers are restricted by the language's low-level model of time, which is usually a system call to the UNIX timestamp. This forces programmers to encode the common sense-compute-actuation paradigm in CPSs as an infinite for-loop and compare timing differences on the device's local clock. The application code awkwardly carries time around with data, which is an inherent requirement in DT apps. Writing recovery procedures for timing violations is also difficult as their installation and execution is dependent solely on time: the deadline by which the action must happen by. The separation between time and data in languages designed without DT systems support in mind leads to verbose and convoluted code.

DT applications benefit from abstractions to handle code distribution, communication, and time requirements at scale. Embedded programming solutions offer helpful abstractions at the device level but fail to scale coordination of space and time for DT applications. Fortunately, prior research in programming languages focus on this issue with macroprogramming. This prior research provides a global view of the system and homogenizes the devices in it, allowing programmers to write device-agnostic code. Homogenization is a powerful abstraction when handling distributed hardware and timing concerns. The programmer focuses on designing system-wide behaviors rather than those local to specific devices. These frameworks provide abstractions for selecting sets of devices, efficient in-network aggregation, and interfaces between heterogeneous devices in the system. This dramatically reduces the overhead for programmers to interface between devices, where the programmer writes a single file rather than one for each device and the framework handles the distribution of code.

We are especially interested in expressing timing constraints across the application. Many research macroprogramming frameworks have been developed but fall short in their handling of timing requirements. Kairos ([9]) was the first step in targeting cyber-physical applications at scale with macroprogramming. It offered extra querying constructs as additions to a host language to easily iterate over multiple devices. However, Kairos lacks timing specifications in its coordination language and requires the user to write code to couple data with time to handle time requirements. This causes cognitive overhead for the programmer to manage time and data separately in a context where the two in isolation is unhelpful. Lingua Franca (LF) ([17]) incorporates a tag with data to specify an ordering of actions in logical time. However, the reduction of timestamps to logical time means LF cannot support the fusion of variable-frequency data streams. LF also requires programmers to check within the host language whether deadlines occur during reaction execution, undercutting the advantages of offering deadline timing abstractions in the macroprogramming layer. In general, macroprogramming research frameworks have little support of timing specifications at the system-level design view. Timing is still handled at the device or function level, whereas DT applications have timing requirements at the global level across devices.

In this thesis, we describe and present TTPython, a domain-specific language and runtime embedded in Python, whose design goals are to help programmers write shorter and cleaner DT applications. TTPython provides abstractions for managing large-scale distributed device networks and for expressing timing constraints on sensing and actuation. Programmers can specify within the programming language soft deadlines in the global distributed application space at a macro level. Our system is written in Python as it is a familiar language for many programmers that is widely supported and has many built-in libraries and useful third-party packages. Many DT applications operate in hardware-constrained environments and have limited power and computation resources, so TTPython does not have a large footprint that could affect system performance. It also includes constructs for timed, reactive backup plans to account for unreliable hardware and networks.

To integrate macroprogramming into a conventional host language, we compile code to a dataflow graph architecture, which is composed of nodes of computation and edges for data communication. We chose a dataflow graph as it is more flexible in partitioning and in execution timing than its control flow counterpart. The intermediate graph representation supports our systems-level view of the application. This model allows us to easily abstract and assign computation to devices. TTPython utilizes a modified version of the MIT Tagged-Token Dataflow graph Architecture (TTDA) [21] to represent timing requirements at the macroprogramming level. The TTDA was insufficient to represent time requirements, so we upgraded the architecture by pairing timing information with data and including a control plane to support periodic deadlines. This control plane is composed of special edges in the graph where the data passing through these edges encode timing information. Data is encapsulated in tokens which have a tag to indicate the liveness of the value. The tag includes a time interval to indicate which iteration of periodic execution the data belongs to. The extra control plane added to the TTDA allows us to encode periodic and deadline behaviors at the graph level; thus, the programmer can specify timing at the global system view. To our understanding, we are the first to explore a time-based dataflow architecture.

TTPython removes the overhead of managing multiple devices due to the intrusiveness of handling communication between devices. Programmers do not have to worry about using architectural interfaces (be they 3rd party libraries or Internet protocols) to communicate across different devices. Communication is accomplished by a variable reference between different graph nodes (that we call Scheduling Quanta [SQ]) at the system-level view of the program. TTPython focuses on time as a first-class construct. Programmers can specify timing requirements and safety mechanisms to take if the timing is not satisfactorily met. Time constraints are specified as comparisons between data availability and deadlines in the global specification of the program. Timing is tightly coupled with data at the system-level view of the application, which highlights timing concerns as a primary concern for the programmer. These constructs are tightly coupled with exception handling, as breaking soft deadlines prompts programmer provided intervention. The programmer seamlessly transitions between developing functional components of their code with Python semantics with the function annotations `@SQify` and `@STREAMify` while specifying their data interactions within the annotated `@GRAPHify` function.

1.1 Thesis Statement

The use of a macroprogramming language with a timed tagged-token dataflow graph intermediate representation helps users write cleaner and shorter distributed, time-sensitive applications.

To evalutate this, we examine TTPython through two case studies and a user study.

1.2 Evaluation

We apply TTPython on two distributed, time-sensitive applications: a 1/10th-scale intersection with connected autonomous vehicles (CAV) and intercity flooding sensing. These applications vary in their time and distribution requirements. In terms of development, the intersection application was initially written in Python and rewritten in TTPython, whereas the intercity flooding application is using TTPython during its development cycle before writing coordination code between the different devices in their system. These two case studies can shed light on how TTPython affects design and development and the challenges found in using TTPython.

The case studies will be used as the tasks for a user study focusing on programmers unfamiliar with cyber-physical systems writing these DT applications. The purpose is to highlight the difficulties programmers face when using conventional frameworks to write DT applications. Participants will experience writing these applications with both TTPython and classical Python to identify the advantages gained with the dataflow graph intermediate representation and the disadvantages of having different semantics from Python as a host language. Each application will have basic tasks for users to complete within a few hours, focusing on making incremental changes to the program. We plan to include a simulation for each application for participants to use to emulate the design and deployment process for these applications.

We will assign programming tasks that are small enough to perform in a few hours but are otherwise representative of the kinds of applications and programming constraints that we observed in our field studies. We will collect data on outcomes, which include time spent programming, bugs introduced (including timing and robustness errors), and success in writing working programs.

1.3 Outline

Chapter 2 describes more in-depth description of research that inspired TTPython. We then provide a high-level description of the TTPython system along with its syntax and dataflow graph semantics in Section 3 and Section 3.5. Section 4.1 describes a case-study in which we evaluate the experience of designing a non-trivial DT application of a 1/10th-scale intersection with connected autonomous vehicles (CAV). The future-work Intercity Flooding Application is covered in Section 4.2. The work culminates with a comparative user study in Chapter 5.1 to observe the challenges user may face when using TTPython or with other classical solutions.

Chapter 2

Related Works

Wireless Sensor Networks (WSN) [23] are often viewed as a precursor to modern cyber-physical systems, particularly those that rely on wireless communications. WSN research considered a wide variety of programming paradigms [19]. Node-centric programs may be written in low-level languages like C, in domain-specific variants like NesC[8], or as a set of high-level interpreted instructions as in Maté [16]. Popular embedded system languages and frameworks such as PRET-C [1] allow programmers to carefully manage resources on a device-level basis for small applications, but the mechanisms they provide prove unscalable when applications utilize thousands to millions of devices. The underlying problem is that they lack the abstractions necessary to coordinate CPSs to work at larger scales. Many approaches sought a more holistic view than node-centric programming by introducing abstractions to handle locality via region formation [20, 22], efficient in-network aggregation [10, 18], and shared-memory based on locality [9].

More CPS-focused approaches are cognizant of heterogeneity in the system and the need to clearly define interfaces between elements of the program [3, 4], leading to macroprogramming frameworks. They provide abstractions for selecting sets of devices, efficient in-network aggregation, and interfaces between heterogeneous devices in the system. These often take a host-coordination language approach in which the host language, such as C or Java, is used for platform specific code and the coordination language encodes communication channels, message formats, and code location to hardware. These models almost exclusively use message-passing architectures and encode the macroprogram in a graph-based intermediate representation before mapping chunks of code to devices. The devices typically host middleware to handle communication, interfaces, and other common runtime mechanisms that are non-specific to the application. Their functionality reflect many ideas found in Links, which pioneered multitier/tierless programming [5]. Links has programmers write client/server applications within a single file and designate functions with annotations where code should be placed. This shortens the distance between functional interfaces from different files (one for a client application and one for the server) into a single file, making it easier to reason about the flow of data in communication between client and server. In this way, Links removed interaction mismatches and centered focus around programmer application code.

Macroprogramming frameworks have slowly begun to include time as a central component. Early frameworks such as Kairos [9] did not include timing constructs within the framework and opted to leave timing handled by its host language, leading to intrusive intermingling of application and system behaviors. COSMOS [3] introduced timing at the system-level overview, but lacks time-specification outside of periodicity. Lingua Franca [17] and PTIDES [24] use "logical time" in each node for deciding when to act upon time-sensitive inputs. They both introduce timing constructs to specify periodicity and deadlines at the level of devices or functions. The choice to use logical time at the application level reduces the flexibility of the language when handling variable-frequency data streams. An iteration in logical time assumes that data is synchronized to that logical tick of time, which is incompatible when data streams provide data at different frequencies.

Importantly, distributed systems interacting with the physical world require a timeline not only for ordering events but also coordinating physical I/O and determining concurrency to compare or combine data values. Having a strong time integration with a macroprogramming framework can alleviate the difficulties found in writing DT applications. The TTPython framework aims to enable easier development of nontrivial distributed, time-sensitive applications by combining these ideas of macroprogramming, time-cognizant architectures, and uncertainty-aware timestamping. The combination of these challenges make this system design difficult, and our advancements in compilation techniques and representation enable TTPython to abstract cross-cutting concerns from distribution while incorporating time-sensitivity.

Dataflow graphs were first introduced in the seminal paper by Dennis [7]. The novelty was to expose the most amount of parallelism in the system by its minimal constraint: data. The semantics were later refined with the U-Interpreter paper [2] by assigning labels to data, allowing nodes to work asynchronously from each other. These labels allowed for-loop execution, as the labels identify which data corresponds to a specific iteration. Data must share the label's context before it can be used for execution. These ideas were later refined in the MIT Token-Tagged Dataflow paper [21] to work well within the context of a von Neuman machine. Our work builds on the foundation of token-tagged dataflow and introduces time as a specific "tag," which allows us to extend dataflow to account for DT applications.

Chapter 3

TTPython

TTPython is a domain-specific language embedded in Python. It separates coordination logic from the application code and allows programmers to specify which code requires careful timing consideration. Although TTPython shares the syntax of Python, top-level application functions execute using a dataflow semantics as specified in Section 3.5. The units of computation of TTPython is the node in the dataflow graph and executes in a functional manner such that program flow depends on data availability as opposed to the strict line-by-line imperative execution in Python. The programmer writes functions that are compiled to nodes, but these functions execute with ordinary Python semantics that is familiar to programmers. TTPython is similar to the Tagged-Token Dataflow Architecture [11, 21] (TTDA) with some modifications to accommodate time, which are detailed in Section 3.5. We first motivate the use of dataflow graph architecture before diving into an application to demonstrate TTPython's capabilities.

3.1 Why the Dataflow Graph Architecture?

When designing a language and system for DT applications, we first take into account the scale of the target applications. The number of devices in these applications can be very large, such as in remote sensing where multiple sensors are scattered across a wide area. Coordinating data and timing requirements between all these devices can quickly become unscalable due to complexity in programmer management. On top of this, platform diversity can hinder development as programmers interface with varying frameworks. Thus, the primary decision we start with is for the programmer to write a single, device-agnostic program with abstracted communication and timing requirements. The difficulty now is to partition the program and map the division to the devices.

We utilize the dataflow graph (DFG) as described in the MIT Tagged Token Dataflow Architecture for maximum flexibility during partitioning the program, since data is the minimum requirement for running a block of code. Compiling to an intermediate graph representation also offers great abstractions for cross-cutting concerns such as communication. The dataflow graph matches the macroprogramming approach of declaring system-wide interactions instead of local steps at the device level.

3.2 Introduction of the Timed Tagged-Token Dataflow Architecture

A dataflow graph is comprised of nodes and arcs where a node is a block of computation and an arc is communication. In our approach, nodes are the smallest block of code that can be scheduled, so we call them Scheduling Quanta (SQs) to differentiate from past literature. Data travels on arcs as a token. The connections of arcs to SQs are called ports, and are used to differentiate between different data tokens. A token is composed of its data value and a tag used to identify the downstream SQ as well as the run-time *context*. The DFG can support multiple concurrent computations as tokens can enter and exit freely, but it needs to identify which tokens are part of the same computation. The context is used to identify the set of tokens that go together, for example data produced in a single iteration of a periodic sensing loop.

3.2.1 Firing Rules

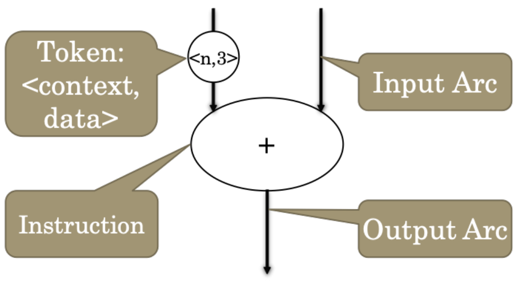


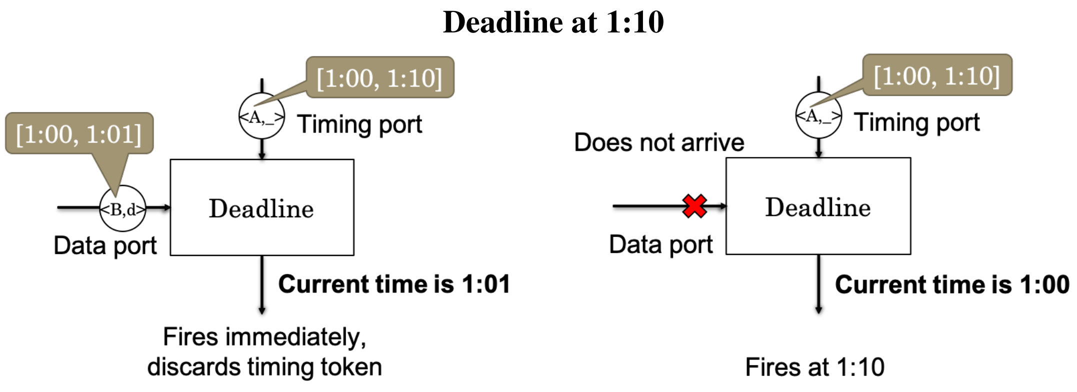
Figure 3.1: A Scheduling Quantum

Firing rules dictate when an SQ is enabled, acting as a synchronization barrier until the required inputs are available. The canonical firing rule is that an SQ can fire once there are tokens on all of its input arcs that have a matching context.

What the time tagged-token DFG differs from its predecessor is the use of a time interval as the context. Tokens are now defined as data with an associated time interval ($[t_s, t_e]$), similar in principle to Spanner [6]. At run-time, these timestamp intervals dictate data validity and concurrency for stream aggregation as well as deadline-driven failure handling for time-sensitive operations like actuation. The canonical firing rule, the Data-Validity Firing Rule, takes this into account: a SQ with this firing rule can fire if there is a token available on every input data port, and the intersection of their time intervals is non-zero (*i.e.*, their time intervals overlap), indicating that they share the same *context*. The intuition behind this is that sampled data can be used with other data if their generation is relatively close to each other in time.

Our data-validity firing rule ensures that we compute with data that are synchronized relative to each other, but we still need a way to make sure that an action will take place before a specified time. To account for this, we introduce control ports. Tokens on these control ports encode a valid time range for associated SQs. We introduce a new firing rule: the Time-Based Trigger: a SQ will fire if a control port has a token for which the end tick of the timestamp is before the current time. The idea is that a timer, represented by the timestamp interval on the control token, has expired, so we need to take some action even though data may not have arrived on the data ports. The Data-Validity Firing Rule can still fire for an SQ with control ports if tokens arrive on all ports (including the control port) and the tokens' timing timing intervals overlap.

To see how this works, say a node with the Time-Based Trigger firing rule receives a data



(a) Normal Execution (b) Plan B Execution

Figure 3.2: Time-Based Trigger Example Firing Semantics

token with the context of [1:00, 1:01], and the current time is 1:00 as shown in Fig 3.2a. The node would immediately fire and pass on the data token as execution has finished on time. This would also invalidate any past or current control tokens as the current iteration of the execution falls within the time constraints set by the programmer. In the extraordinary case, data does not show up, either because the upstream node producing the token is taking too long, or the upstream connection is faulty and fails to send the token. The token on the timing port would sit until 1:10, when the SQ would fire and realize that the data port has not yet supplied a token, as shown in Fig 3.2b.

3.3 Example TTPython Program

We describe the language and its semantics through a sample application adding two simulated sine wave data sources as seen in Figure 3.3. TTPython tracks the addition of these two streams as well as the average of the addition for a period of 30 seconds. The programmer first writes the high-level coordination between functions that will run in parallel in the application. This is done with the function annotation `@GRAPHify`, as seen on line 21. `@GRAPHify` takes the associated function and constructs a dataflow graph from the operations within it. It requires that all operations within the function must either be basic arithmetic or boolean operations or a TTPython annotated function (`@SQify` or `@STREAMify`). The compiler automatically converts basic Python constants, arithmetic, and boolean operators to their SQ counterpart. For example, the constants seen on line 23 are actually converted into SQs that generate said constant as an output token. As the programmer describes the application at the system level in `@GRAPHify`, SQs can be assigned to different devices. TTPython uses message passing semantics, so the data passed between SQs are copied and are not shared in memory.

SQs can be defined by the programmer through the function annotations `@SQify` and `@STREAMify`. `@SQify` takes the associated function and translates it into a node of computation in the dataflow graph intermediate representation. Each parameter in its signature corresponds to an input arc, and the return values to an output arc. In the case of returning a pair, as in the function f from the example, each value in the pair will result in an output arc. The exception

```python
@STREAMify
def sinusoid_sampler(A, f, phi):
    from math import sin, pi    # local import
    global sq_state
    if sq_state.get('count', None) == None:
        sq_state['count'] = 1
    sample = A * sin(sq_state['count'] * 2 * f / pi + phi)
    sq_state['count'] += 1
    return sample

@SQify # user defined SQ
def moving_average(new_input):
    global sq_state             # persistent local state
    count = sq_state.get('count', 0)
    avg = (sq_state.get('avg', 0) * count + new_input) \
        / (count + 1)
    sq_state['count'] = count + 1
    sq_state['avg'] = avg
    return avg

@GRAPHify # main program specifying SQ linking
def add_sine(trigger):
    A_1 = 1; f_1 = 0.25; phi_1 = 0 # constants are SQs
    A_2 = 2; f_2 = 0.25; phi_2 = 0

    with TTClock.root() as root_clock:
        start_time = READ_TTCLOCK(trigger, TTClock=root_clock)
        # periodically generate for 30 seconds
        N = 30
        stop_time = start_time + (1000000 * N)

        # create a sampling interval by setting the
        # start and stop tick for one of the args
        sampling_time = VALUES_TO_TTTIME(start_time, stop_time)
        A1_sample = COPY_TTTIME(A_1, sampling_time)
        A2_sample = COPY_TTTIME(A_2, sampling_time) # copies token time interval
        sine_1 = sinusoid_sampler( # streamify call 1
                    A1_sample, f_1, phi_1,
                    TTClock=root_clock, TTPeriod=500000,
                    TTPhase=0, TTDataIntervalWidth=100000)
        sine_2 = sinusoid_sampler( # streamify call 2
                    A2_sample, f_2, phi_2,
                    TTClock=root_clock, TTPeriod=500000,
                    TTPhase=0, TTDataIntervalWidth=100000)

    output = sine_1 + sine_2
    y = moving_average(output)
    return output
```

Figure 3.3: *A TTPython application adding two sine waves and calculating the average periodically every 0.5 seconds for 30 seconds.*

is for keyword arguments prefixed with TT, which are special arguments used by the runtime. These keyword arguments are not accessible to the annotated function. Line 11 shows @SQify applied to a moving average function. During compilation, this would translate to an SQ with one input and output arc. It weights the existing average by the number of samples, adds the new sample value, and divides by the new number of samples. The implementation of a moving average requires tracking the number of samples taken between iterations. To account for this, TTPython reserves the global variable sq_state to store state between multiple executions of a SQ. Note that this does not share the same semantics as Python's **global** keyword. Instead, the variable is only locally observable by the SQ, so accesses are to the local device's version of the SQ's state.

The @SQify and @GRAPHify constructs we have described so far are insufficient for writing periodic computations. The system needs programmer-specified arguments to describe how long to run the data generation for the sine wave stream. The function annotation @STREAMify describes a function that will generate a continuous data stream when called. @STREAMify operates similarly to @SQify but has added functionality to support periodic execution of the associated function. The function operates similarly to a self-emitting stream in which computation will periodically trigger itself to run. As a reminder, TTPython wraps and unwraps data from arguments and return values respectively through *tokens*, which are values passed along edges in the dataflow graph. Tokens include time intervals with data. Intuitively, these time intervals indicate the time period when the data is valid. Computation involving multiple data values can execute if these data share overlapping time intervals (Data-Validity Firing Rule), indicating that they share a common temporal context as discussed in Section 3.2.1. We modify an input token's time interval with the two provided library SQs VALUES_TO_TTTIME and COPY_TTTIME. VALUES_TO_TTTIME sets the start and stop ticks of the output token's timestamp interval to be the data values from the first and second input tokens respectively. COPY_TTTIME creates a new token with the data value of the first token and the time interval of the second token. In Figure 3.3, by setting N=30, we are creating a sampling interval of 30 seconds given that the periodicity is 1 second. These are some of the API functions we've provided to the programmer to interface with the TTPython architecture.

The invocation of the STREAMified sinusoid_sampler function provides periodicity information via specially named keyword arguments TTClock, TTPeriod, TTPhase, and TTDataIntervalWidth. The programmer can specify when the SQ will trigger. A clock is implemented by a counter that counts subticks before the clock increments logically in time, such as a second. A TTPhase=0 specifies that the sinusoid_sampler will trigger when the counter reads 0 modulo the period; for TTPeriod=500000, a TTPhase=250000 would trigger 250ms after the counter wraps around. When running periodic computation, the function also needs to define the time context of the data it generates. When the runtime initiates an instance of the SQ, the runtime will take the start and stop time for executing this SQ and take the average. The new interval is this timestamp average, plus-minus the TTDataIntervalWidth divided by 2. The width of the resulting interval is thus TTDataIntervalWidth. Currently, this is dictated by the programmer but could later be extended to dynamically estimate the typical clock uncertainty in the network.

3.4 Time and Location Constraints

```python
@SQify
def deadline_check():
    # sample reactive failure: provide a null value or a previously one
    return None

@SQify
def moving_average(new_input):
    ...

@GRAPHify
def add_sine(trigger):
    ...
    with TTClock.root() as root_clock:
        ...
        deadline_time = READ_TTCLOCK(    # deadline generation
            sine_1,
            TTClock=root_clock
        ) + 50000
        ...

        with TTConstraint(name="dev1"): # assign SQs to 'dev1'
            safe_add = TTFinishByOtherwise(
                output,
                TTTimeDeadline=deadline_time,
                TTPlanB=deadline_check(),
                TTWillContinue=True
            )
            # if deadline fails, it produces a separate
            # value (similar to ternary operation:
            # a = y if clock.now < t else None)

        moving_average(safe_add)      # downstream SQ
```

Figure 3.4: *An updated example of the deadline construct with location specification with adding two sine waves.*

TTPython also offers two more constructs at the `@GRAPHify` level to assist programmers in dealing with space and time constraints. We upgrade our sinusoid program to include a deadline constraint on the addition of the two sine wave values and specifying the location of that SQ to a particular device.

3.4.1 Specifying Location

Before graph execution, TTPython needs to decide how to map each SQ in the dataflow graph to its host device. We offer a simple constraint syntax with the keyword `TTConstraint` where the programmer can specify requirements on groups of SQs. These requirements can range from required hardware for an SQ (*e.g.*, a function that takes an image requires a camera on the device) or software (*e.g.*, a specific library for Fast Fourier Transforms). In this program, the programmer has specified that the SQ within the **`with`** block on line 21 will be assigned to the device named "dev1". If multiple devices fulfill a requirement, an arbitrary device that satisfies the requirement is selected. Future work will focus on optimizing this selection with respect to timing and other user-specified parameters.

3.4.2 Deadline Constructs

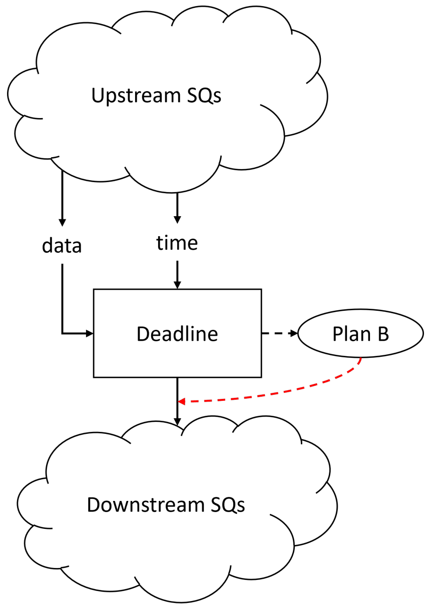


Figure 3.5: Deadline DFG Compilation

Timing requirements in TTPython have two components: the time by which the computation should finish and the backup procedure to run if the execution doesn't finish on time. The former is produced by generating a timestamp from `READ_TTCLOCK` and offsetting it with an addition. `READ_TTCLOCK` accepts two parameters: a trigger as to when to take the timestamp and a clock from which to take the timestamp. TTPython uses a clock to allow programmers to specify the level of synchronization necessary between time-sensitive actions. It requires a root clock so that all timing specifications in the program can be compared to each other even on different devices. The root clock, a proxy for Universal Coordinated Time, is sufficient for the contributions of this proposal. In the updated example in Figure 3.4, `deadline_time` is generated by each iteration of `sinusoid_sampler`. The TTPython SQ `TTFinishByOtherwise` construct ensures that data has been generated on time by triggering a backup procedure (*Plan B* [[15]]) if the data is not received by the specified deadline, which is `deadline_check` in our example. If Plan B is run, the programmer can specify different behaviors regarding the rest of execution. These behaviors are similar to those associated with raising an exception.

The SQ `TTFinishByOtherwise` supports two types of recovery behaviors: replacing a "late" value with a default value provided by Plan B or running a separate backup routine without executing the rest of the data dependencies of that data variable. For example, a backup routine on an autonomous vehicle that instructs the car to apply the brakes may not want downstream SQs to execute after applying the brakes, as they could command the car to continue moving forward. `TTWillContinue` accepts a bool and indicates if the programmer intends to execute the downstream SQs from the data value given when Plan B is executed. If `TTWillContinue` is **`True`**, then the default value passed on will be the value returned by the function call specified with `TTPlanB`. Downstream SQ execution will continue with this default value. This pass-through procedure is visually represented in Fig 3.5 by the red dotted line. Otherwise, if `TTWillContinue` is **`False`**, the compiler will not include the red dotted line. The bool value **`True`** for `TTWillContinue` indicates that the graph will always execute the `moving_average` SQ on line 32 regardless if Plan B occurred. The graph will use the value **`None`** from the output of `deadline_check` as the value for `safe_add` if `sine_1` does not arrive by `deadline_time`.

3.5 TTPython's DFG Semantics

We will now present a formalism of the semantics of the Timed Tagged-Token DFG. A SQ *fires* (executes computation) when certain conditions are satisfied, which is known as its **firing rule**. TTPython has two firing rules: the Data-Validity (DV) and Time-Based Trigger (TBT). The former is the default firing rule for most SQs, as it groups received tokens that match in context to fire. The latter enables the DFG to encode deadlines at the system level, relieving programmers of low-level code required to interface with time checking. If the firing rule is successful, the SQ then applies its functional computation on the data and updates the context of the token for downstream SQ forwarding. Tokens are stored in a *waiting-matching section*, where tokens wait until the SQ accumulates enough tokens with the same context to fire.

The dataflow graph $\mathbb{G}$ is represented by a set of scheduling quanta V. Each SQ has a set of named input ports and a set of output ports. Edges are implicitly defined by the output ports, which are represented as a list of the names $\overline{\mathtt{x}}$ of the input ports they are connected to.

A SQ ($v \in \mathsf{V}$) is defined as $\langle f, \bar{i}, \bar{o}, r, s \rangle$ where f is the function v is encapsulating, $\bar{i}$ is a list of input ports, $\bar{o}$ is a list of output ports, r describes the firing rule for v, and s is the internal state of the SQ.

f is of type $\mathsf{List}\langle\mathsf{Token}\rangle \rightarrow \mathsf{Set}\langle\mathsf{Pair}\langle\mathsf{Token}, \mathsf{Name}\rangle\rangle$, where the name is the name of input ports we are sending the corresponding token to (we assume the implementation of the function does this by looking up what the SQ's output ports are connected to).

Each input port has a waiting-matching section $\mathtt{W}$, which holds tokens received by the port until the SQ fires. The structure of an input port i is composed of $\langle \mathtt{x}, \mathtt{W} \rangle$. s is the internal state of the SQ for firing rule purposes and differs from `sq_state` as described in Section 3. `sq_state` is handled as a closure within the execution of the node and is abstracted from the semantics of the DFG. s instead is used to keep track of particular tokens for the Time-Based Trigger firing rule, which we will discuss later in Section 3.5.2.

Tokens

An execution of the graph involves tokens propagating through $\mathbb{G}$. A token k is defined as follows:

$$k = \langle d, t \rangle$$

where d is data and t is a time interval $[t_s, t_e]$ where $t_s \leq t_e$. For simplicity, both timestamps in t can be represented as elements of $\mathbb{N}$ (*e.g.* a Unix timestamp).

By default, an output token's time interval is the intersection of all the input tokens' time intervals. If the programmer needs to override this default, such as extending the lifetime of a value beyond that of the data it was computed from, we provide APIs for this purpose. These APIs are also used to set the time interval for tokens that specify the length of time to generate a periodic stream, as was shown in Figure 3.3 from lines 33 to 36.

DFG Model

We first describe the system configuration $\mathcal{C}$ of DFG. This is described as the tuple

$\mathcal{C} = \langle \mathbb{G}, \mathcal{N}, t \rangle$

where $\mathbb{G}$ is the dataflow graph, $\mathcal{N}$ is a set to hold onto tokens in transit through the network, and t as the current time for the system. An element of $\mathcal{N}$ is described as a tuple $\langle k, \mathtt{x} \rangle$ where k is a token and $\mathtt{x}$ is the name of the input port it is addressed to.

Sometimes, there are no possible actions for $\mathbb{G}$ because tokens are still in the network $\mathcal{N}$. Similarly, tokens can take time to travel through $\mathcal{N}$. To simulate this, a rule describes the passage of time with no actions taken by the dataflow graph or network. We represent this with the following rule[1]:

$$\frac{}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G}, \mathcal{N}, t+1 \rangle}\ \textit{Inc-Time} \tag{3.1}$$

Tokens can also exit from the network. They are added to the waiting-matching section of its destination input port[2].

$$\frac{\begin{array}{c}\langle k, \mathtt{x} \rangle \in \mathcal{N} \quad v \in \mathbb{G} \wedge i \in v.\bar{i} \wedge \mathtt{x} = i.\mathtt{x} \\ v' = v[i \rightarrow \langle x, i, i.\mathtt{W} \cup \{k\} \rangle]\end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \setminus \{\langle k, \mathtt{x} \rangle\}, t \rangle}\ \textit{Rcv-Tok} \tag{3.2}$$

The notation for line 2 in Rule 3.2 ($v' = v[i \rightarrow \cdots]$) is shorthand for updating the SQ's input port with the new waiting-matching section unioned with the token to be added. To start a nontrivial execution, an initial configuration begins with a nonempty $\mathcal{N}$. The system generates new tokens by consuming tokens from its input ports and sending them to the network. An SQ's firing rule describes this behavior.

3.5.1 Firing Rule: Data-Validity

An SQ v with the data-validity firing rule will fire if there exists a token on each input port of v such that their time intervals overlap. Tokens $k_i = [t_s^i, t_e^i]$ and $k_j = [t_s^j, t_e^j]$ have overlapping time intervals ($k_i \cap k_j \neq \emptyset$) if $t_s^i \leq t_e^j$ and $t_s^j \leq t_e^i$. The firing rule is described by the rule below:

$$\frac{\begin{array}{c} v = \langle f, \bar{i}, \bar{o}, \mathtt{DV}, s \rangle \in \mathbb{G} \wedge \mathtt{len}(\bar{i}) = n \wedge \bar{i} = [i_1, \cdots, i_n] \\ \exists\, k_1, \cdots, k_n.\ \forall\, a, b \in 1..n. \\ a \neq b \implies k_a \in i_a.\mathtt{W} \wedge k_b \in i_b.\mathtt{W} \wedge k_a \cap k_b \neq \emptyset \\ f(k_1, ..., k_n) = \mathcal{N}' \\ \forall j \in 1..n.\ \mathtt{W}_j{}' = i_j.\mathtt{W} \setminus \{k_j\} \\ v' = \langle f, [\langle i_1.\mathtt{x}, \mathtt{W}_1{}' \rangle, \cdots, \langle i_n.\mathtt{x}, \mathtt{W}_n{}' \rangle], \bar{o}, \mathtt{DV}, s \rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t \rangle}\ \textit{DV-FR} \tag{3.3}$$

[1]This rule has lower precedence than the future rules described later in this section.
[2]Set operations are left-to-right associative unless otherwise noted.

3.5.2 Firing Rule: Time-Based Trigger

If a SQ has this firing rule, then it has two input ports, delineated as i_d and i_c. i_d corresponds to a data port, and i_c is the control port. We label tokens k_d and k_c for the i_d and i_c input ports respectively. The list of output ports $\overline{o}$ consists only of downstream SQs that use the data token, so we simplify $\overline{o}$ to a single output port o. The SQ acts as a no-op when the deadline time is satisfactorily met.

The SQ will internally keep track of the last control token it has seen from i_c that caused Plan B to run, which we denote as k'_c. This is tracked in $v.s$, the SQ's internal state. k_c encodes the deadline in its time interval. $k_c.t_s$ denotes when the deadline was initiated and $k_c.t_e$ is the timestamp of the deadline.

TTPython uses a simplifying assumption that all intervals have been widened for uncertainty. Its properties are as follows:

1. Plan B fires at most once per control token. It will either fire if the data token does not arrive on time or be discarded otherwise.
2. If the control token input to a TBT SQ is generated locally, Plan B will run in a timely manner (*i.e.*, there are no network complications in running the SQ).

We describe the rules with success and failure cases. Summarily, success is when both conditions are satisfied:

1. Data is synchronized with the control token (*i.e.*, they overlap).
2. Tokens arrive on time (*i.e.*, they arrive before the deadline specified by the control token).

$$\frac{\begin{array}{c} v = \langle f, [i_d, i_c], o, \texttt{TT}, \langle k'_c \rangle\rangle \in \mathbb{G} \\ k_d \in i_d.\texttt{W} \quad k_c \in i_c.\texttt{W} \quad t \leq k_c.t_e \quad k_d \cap k_c \neq \emptyset \\ \bar{i} = [\langle i_d.\texttt{x}, i_d.\texttt{W} \setminus \{k_d\}\rangle, \langle i_c.\texttt{x}, i_c.\texttt{W} \setminus \{k_c\}\rangle] \\ v' = \langle f, \bar{i}, o, \texttt{TT}, \langle k'_c \rangle\rangle \\ \mathcal{N}' = \{\langle k_d, x\rangle \mid x \in o\} \end{array}}{\langle \mathbb{G}, \mathcal{N}, t\rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t\rangle} \; \textit{TBT-S} \tag{3.4}$$

Exceptional cases occur when either condition is not met.

- Data is late (*i.e.*, time t is greater than the control token k_c's end timestamp).

  Plan B is run (Rule 3.5), and we will discard any data tokens that would be synchronized to this control token (Rule 3.6). The function $\texttt{max}(k, k')$ returns the token with the greater t_e.

$$\frac{\begin{array}{c} v = \langle f, [i_d, i_c], o, \texttt{TT}, \langle k'_c \rangle\rangle \in \mathbb{G} \\ k_c \in i_c.\texttt{W} \quad k_c.t_e < t \\ i^n_c = \langle i_c.\texttt{x}, i_c.\texttt{W} \setminus \{k_c\}\rangle \\ v' = \langle f, [i_d, i^n_c], o, \texttt{TT}, \langle \texttt{max}(k_c, k'_c)\rangle\rangle \\ f(k_1, ..., k_n) = \mathcal{N}' \end{array}}{\langle \mathbb{G}, \mathcal{N}, t\rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N} \cup \mathcal{N}', t\rangle} \; \textit{TBT-F-PBC} \tag{3.5}$$

$$\frac{\begin{array}{c} v = \langle f, [i_d, i_c], o, \texttt{TT}, \langle k'_c \rangle \rangle \in \mathbb{G} \\ k_d \in i_d.\texttt{W} \quad k_d.t_s \leq k'_c.t_e \\ i^n_d = \langle i_d.\texttt{x}, i_d.\texttt{W} \setminus \{k_d\} \rangle \\ v' = \langle f, [i^n_d, i_c], o, \texttt{TT}, \langle k'_c \rangle \rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N}, t \rangle} \textit{ TBT-F-PBD} \tag{3.6}$$

- Data has arrived but is unsynchronized.

  This failed condition has three cases:

  - **Data token has arrived but there is no overlapping control token.**

    The SQ will wait for a control token to come.

  - **Control token arrives after its specified deadline.**

    As the deadline has already passed, the SQ must run Plan B immediately. This is also already encoded with Rule 3.5.

  - **Control and Data token do not match.**

    This rule depends on the order of the time intervals between the control and data tokens. We assume here that $k_d \cap k_c = \emptyset$.

    If k_d's time interval is before k_c, the SQ will wait until time is greater than $k_d.t_e + \texttt{Exp}$, where $\texttt{Exp}$ is a programmer-defined time extension to wait for the control token. $\texttt{Exp}$ by default is upper-bounded by the periodicity of the stream that k_d has been generated by. The default ensures in-order operation for periodic streams. This implies that if iteration n of the control token in the stream arrives, any tokens from iteration $n-1$ are automatically late (unless if $\texttt{Exp}$ is set explicitly longer by the programmer).

$$\frac{\begin{array}{c} v = \langle f, [i_d, i_c], o, \texttt{TT}, \langle k'_c \rangle \rangle \in \mathbb{G} \\ k_d \in i_d.\texttt{W} \quad k_c \in i_c.\texttt{W} \\ k_d.t_e < k_c.t_s \quad k_d.t_e + \texttt{Exp} < t \\ i^n_d = \langle i_d.\texttt{x}, i_d.\texttt{W} \setminus \{k_d\} \rangle \\ v' = \langle f, [i^n_d, i_c], o, \texttt{TT}, \langle k'_c \rangle \rangle \end{array}}{\langle \mathbb{G}, \mathcal{N}, t \rangle \longrightarrow \langle \mathbb{G} \setminus \{v\} \cup \{v'\}, \mathcal{N}, t \rangle} \textit{ TBT-F-US} \tag{3.7}$$

    If k_c's time interval is before k_d, the SQ waits for the corresponding data and control tokens for each as the data for the next iteration has arrived earlier than expected.

Chapter 4

Case Studies

4.1 Smart Intersection

Cooperative autonomous vehicle intersections present a unique problem in terms of timing. The goal of an autonomous vehicle intersection is to allow the Connected Autonomous Vehicles (CAVs) to drive through the intersection as close to the speed limit as possible without colliding [14]. Often this will result in vehicles traveling at high velocity in close proximity to each other with little margin for error. Accomplishing this feat requires precise timing of sensing, sensor fusion, communications, and execution of the planned path. Any deviation or failure could be catastrophic. This case study application was created using 1/10-scale autonomous vehicle models with scale accurate LIDAR and camera sensors shown in figure 4.1 as well as 1/10-scale connected infrastructure sensor (CIS) which is a stationary camera that is also sharing information with the intersection, shown in figure 4.2. The 1/10-scale vehicles drive a figure eight loop with an intersection in the middle, which can be seen in Figure 4.3. This intersection is controlled using an autonomous vehicle intersection controller that is run on a Jetson TX2. The intersection has four scale CAVs to control, 2 scale CISs, and a road side unit (RSU) that is running the intersection controller.

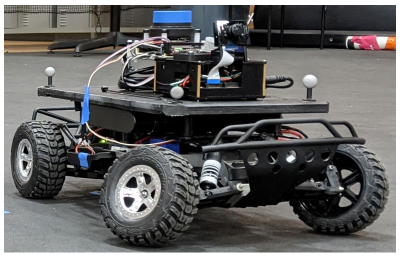


Figure 4.1: One tenth scale CAV with camera, LIDAR, and Nvidia Jetson Nano for on-board processing.

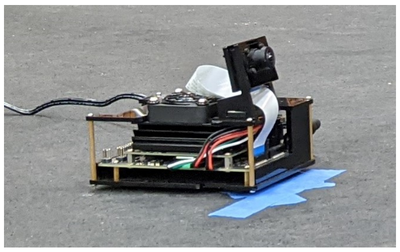


Figure 4.2: One tenth scale CIS with camera using Nvidia Jetson Nano for on-board processing.

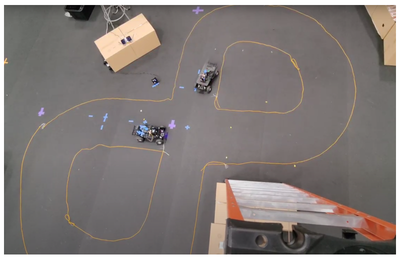


Figure 4.3: Overhead of one tenth scale CAVs shown driving within the figure 8 intersection.

The CAV intersection application consists of a multitude of sensor streams that must be combined into a single worldview, which is then used to determine the paths each CAV will carry out in the physical world. Figure 4.4 depicts a CAV or CIS data-flow diagram. All processes of the CIS are depicted in blue while those of the CAV are in blue and gray. A CAV has an extra LIDAR sensor and can actuate a steering and drive motor to move, whereas the CIS sensor is just for sensing alone and has no actuation ability. All CAVs and CIS sensors in the network must synchronise their sensor frequency to within a tight margin so that the sensor fusion operates correctly. All CAVs and CISs must process and locally fuse their data and have it sent to the RSU so that there is enough time for the RSU to calculate the global fusion and intersection control [13] (see Figure 4.5). Then that data is in turn sent out to all the CAVs so they actuate their steering and motors on time. This entire process must happen within 0.125ms so that timing error does not cause a crash of the CAVs. This timing and communication is very sensitive to problems and thus makes an excellent case study for TTPython. Even slight timing issues can cause the perceived positions and trajectories of the CAVs to be estimated incorrectly which can result in a crash as the CAV is in actuality at a different position. Additionally any late or out of order communications can result in a crash because these type of errors may also affect the perceived positions.

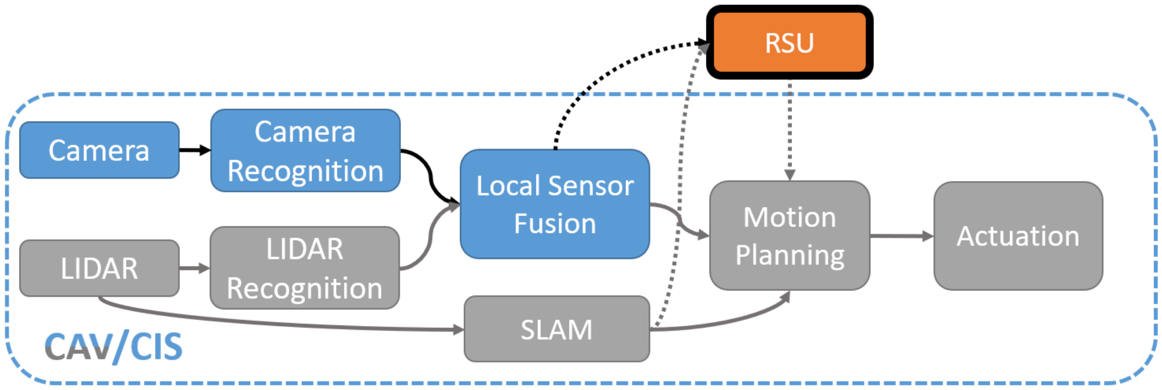


Figure 4.4: Data-flow of a 1/10-scale Connected Autonomous Vehicle (CAV) shown in blue and gray and a 1/10-scale Connected Infrastructure Sensor (CIS) shown in blue. CAVs and CISs send their locally fused sensor data to the RSU and CAVs receive intersection control back which is used to actuate the steering and motors.

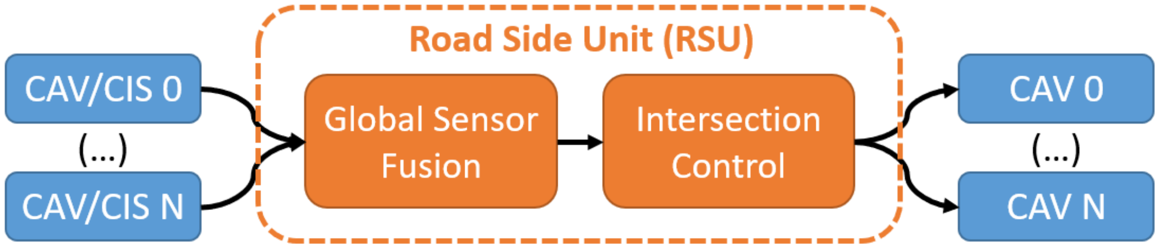


Figure 4.5: Data-flow of a 1/10-scale Road Side Unit (RSU) that gathers sensing data from the CAVs and CISs in the area, processes the global sensor fusion, and calculates the intersection controls to send back out to the CAVs in the area.

(LoC)	Original App	TTPython Variant
Timing	255	39
Threading	29	X
Networking	340	X

Table 4.1: Breakdown of non-application specific lines of code. TTPython gets threading and networking code for free due to the nature of the DFG.

4.1.1 Code Comparison

We compare the original code for the aforementioned smart intersection application to a newly converted TTPython version. The original application has 6415 lines of code, 624 of which is non-application specific code. The functionality of both code bases is identical from the application output perspective. However, many of the internal structures were changed in migrating the original Python code base to TTPython. These changes fall into three main categories: 1) timing management code ensuring that sensors and devices sense and actuate at the proper time, 2) threading and thread management, including pipelines for thread-to-thread communication, and 3) networking code allowing all the different devices to communicate. TTPython reduces the code needed for all three of these categories and is able to eliminate 93% of the code in these three categories combined, as seen in Table 4.1.

Timing Management Code is Drastically Reduced

Timing management code manages the flow of a program in time. This includes making sure that information sources such as the cameras and LIDARs stay synchronized at the desired frequency and within a certain time interval. It also includes checking for missing data and running backup routines (PlanB). TTPython provides direct support for timing management, so there is marked improvement in which 255 lines of Python code were removed and replaced by 39 lines of TTPython. For example, in TTPython a single line of code such as `cam_sample = camera_sampler(cav_0, sample_window, TTClock=root_clock, TTPeriod=125000, TTPhase=0, TTDataIntervalWidth=62000)` can replace around 30 lines of manually written Python code to directly manage the timing and synchronization. Thus, as shown in Table 4.1, our case study reduced this category of code by 85%, while also making the code more declarative and easier to understand.

Examining Figure 4.6, the original CAV application made direct use of the Python time library to implement waiting for the right time to start computation and a `while` loop to implement periodicity with a `time.sleep()` to run every millisecond. As seen in Figure 4.7, TTPython separates these concerns from the programmer into 4 steps: setting a start time, stop time, periodicity, and data interval validity for downstream tokens. In the interest of space, we have not shown the timestamp checking code for the original application. This code is built into the TTPython runtime system and thus does not have to be written by the applications programmer; eliminating it is one of the reasons for the reduction in code size mentioned above.

```
# Start after 10 seconds
start_time = 10000000
interval = 125000 # Interval is 125ms

# Sleep until test start time
wait_until_start = (
  start_time - time.time() -.01
)
if wait_until_start > 0:
  time.sleep(wait_until_start)

# Sleep thread until target time
target = start_time
while 1:
  if (fetch_time(simulation_time) >=
      target):
    now = time.time()

    camcoordinates = camera_sampler()

    # Prep value to be sent
    # Clear the queue of old data
    while not out_queue.empty():
        out_queue.get()
        out_queue.put(
            [camcoordinates,
             now,
             time.time()]
        )

    target = target + interval
  time.sleep(.001)
```

Figure 4.6: *The original CAV application using Python time and user-written communication library.*

```
N = 10
# starts N seconds later
start_time = READ_TTCLOCK(
    trigger, TTClock=root_clock
) + 1000000 * N
stop_time = GET_INFINITY(
    trigger, TTClock=root_clock
)
sampling_time = VALUES_TO_TTTIME(
    start_time, stop_time
)
sample_window = COPY_TTTIME(
    A_1, sampling_time
)

with TTConstraint(name="cav0"):
    cam_sample = camera_sampler(
        cav_0, sample_window,
        TTClock=root_clock,
        TTPeriod=125000,
        TTPhase=0,
        TTDataIntervalWidth=62000
    )
```

Figure 4.7: *The CAV application rewritten using TTPython.*

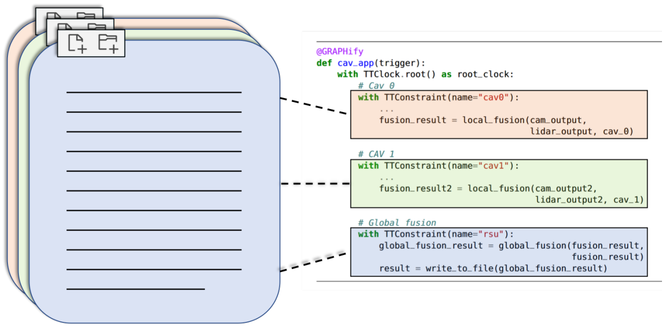
Thread Management and Networking Management Code is Eliminated

In TTPython, threading is managed by the data flow runtime. From the programmer's perspective, each SQ can be considered its own thread, and thus the need to write extra code to manage each thread (beyond `SQify` annotations) is not necessary. This completely eliminates 29 lines of threading and thread management from the original application. Communication is also built into the TTPython data-flow architecture. In our case study the user had to select a communication stack and chose Flask, a Python library. Though a popular choice for networking, it was still not trivial to implement the communication API and call it from each device. Additionally, timing data structures, fallback routines, and routing tables had to be created manually, whereas TTPython generates those automatically. TTPython thus resulted in a savings of 340 lines of code, as well as the complete removal of the Flask library.

Referring back to Fig 4.6, there are explicit calls to the network queues used for connections to other devices. In the old application, each ensemble required a separate file responsible for handling communication. TTPython provides this to the programmer for free. In TTPython, communication across devices is handled via variables and thus a line of code like `local_fusion_result = local_fusion(cam_output, lidar_output` followed by `global_fusion_result = global_fusion(local_fusion_result)` specifies that the local fusion result from up to n CAVs will be sent via TTPython's networking stack, accumulated and fed to the function `global_fusion()` with no further networking or synchronization code necessary from the programmer side. This further simplifies the project structure as shown in Fig 4.8. In the original application, there are separate folders for the CAV and RSU code. Any shared communication would first need to interface with the networking library before other devices could receive it. Sharing is simple in TTPython, as **`with`** blocks with `TTConstraint` do not limit variable scoping. Referencing and updating what data needs to be shared across devices is more explicit and easier to verify within a file as a shared variable name.

4.1.2 TTPython Best Practices

The process of converting to TTPython is trivial at times and at other times required a complete re-arrangement of how a traditional programmer may write code. TTPython effectively requires the programmer to consider the distributed data flow nature of their code at all points. For instance, reading from the camera and LIDAR sensors using the `@STREAMify` construct happens in TTPython simultaneously despite being written in a sequential fashion whereas in traditional Python the functions would be executed in-order. This allows inherent parallel execution by simply setting a timing parameter in `@STREAMify` rather than needing to explicitly split code into threads or use the thread pooling construct. This mindset is very useful, because as is well documented in the literature, distribution is not a feature that can be hidden but rather should



```
@GRAPHify
def cav_app(trigger):
    with TTClock.root() as root_clock:
        # Cav 0
        with TTConstraint(name="cav0"):
            ...
            fusion_result = local_fusion(cam_output,
                                         lidar_output, cav_0)

        # CAV 1
        with TTConstraint(name="cav1"):
            ...
            fusion_result2 = local_fusion(cam_output2,
                                          lidar_output2, cav_1)

        # Global fusion
        with TTConstraint(name="rsu"):
            global_fusion_result = global_fusion(fusion_result,
                                                 fusion_result)
            result = write_to_file(global_fusion_result)
```

Figure 4.8: Macroprogramming framework TTPython Code. The original implementation had separate files designated for each ensemble, while TTPython supports development of all constituent devices seamlessly in one file.

be at the forefront of the programmer's mind [12]. TTPython is not designed for all distributed systems; for example, if a programmer wants to write a single threaded sequential type of code that connects to a server and uploads some data, TTPython will not be useful for that programmer. However a programmer writing highly distributed and multi-threaded code executing on heterogeneous devices would benefit from TTPython's abstractions for managing distribution, concurrency, and mapping to devices. In our Intelligent Intersection use case, the programmer gains a lot for free in terms of timing and sequentiality of the firing order of processes as well as many benefits from the communication protocols that are built into TTPython functions.

Built-in Communication, Timing, and Synchronization Saves Time

The second source of improvement from TTPython is that communication, timing, and synchronization are built-in. Thus, the programmer does not need to worry about using libraries or implementing these manually. This is a huge benefit in terms of development-time savings. By simply calling an SQ and using the resulting variable in the next SQ, TTPython will automatically handle that communication, including determination of the downstream code's location on the network and correctly transferring data to said downstream code, be it on another device or the current one. There is no need to implement APIs for networking or to understand exactly where the SQ is mapped—if it is on the same device, TTPython will use short-circuit the networking stack to save time.

By using the TTPython `@STREAMify` decorator, almost all aspects of timing for a sensor stream can be initialized using a single line of code. In fact, that line of code does even more, incorporating validity intervals that will support semantic matching with other similarly produced tokens as they arrive to in downstream SQs. This saves a significant amount of coding effort: the programmer need not recreate the timing structures manually nor concern themselves with time-stamping data and matching the validity intervals. Finally, the backup routines present in the Plan B structure within TTPython allows for the easy addition of any backup function required with only two extra line of code and no additional timestamp comparisons.

4.2 Intercity Flooding

The second case study will focus on wide-area sensing for detecting flooding of intercity sewer systems. Elizabeth Carter from Syracuse University is working closely with the United States Geological Survey (USGS) and the city of Syracuse to develop this application. Uneven water levels in a city sewage system can be caused by numerous external factors, such as impeding foliage or focused rain fall. Identifying sewage blocking is imperative as treatment prevents the backflow of sewage violently expelling from house plumbing. The application combines thermal and optical imaging with location sensors (GPS and IMU) to identify where flooding occurs within the sensor network deployed across the city. If flooding is detected, the app will notify personnel to address the flooding. Furthermore, the device that detected the flooding will notify surrounding devices to increase their sampling rate to better gauge the water levels as the situation unfolds. The city of Syracuse plans to integrate this application into its city infrastructure, which can provide a more stable environment in terms of power and internet capabilities, while the USGS is interested in more remote locations where cell reception can be difficult to find.

We are already in an active collaboration with Dr. Carter to author their Intercity Flooding application. The application is an ideal real-world testbed for TTPython, as it is a distributed, cyber-physical system including timing constraints which vary adaptively, failure scenarios that motivate backup routine use, and different modalities of execution based off of environmental factors. It is motivated by these research questions.

RQ 1: **How do domain-scientists work with experts in cyber-physical systems to create large-scale hydraulic sensor networks?**

RQ 2: **How does TTPython's abstraction of timing and distribution help programmers write application-specific code?**

RQ 3: **What are the limitations and barriers-to-entry of TTPython?**

4.2.1 Initial Findings

There have been initial observations with Dr. Carter's team both at the development and planning level to better understand the scope of the flooding application. The observation on coding development focused on a team member using TTPython to handle timing requirements between different data streams. The participant had written the code interface with the hardware system calls, but had not yet considered the timing implications between different data streams during that period. We found difficulties with understanding the interface of timing in TTPython and hardware application issues. A problem we noticed was a mismatch of timing intention to graph compilation. TTPython implements periodicity through a modified Data-Validity firing rule, so periodicity is linked to an SQ. When writing the code, it was more natural to specify the periodic parameters for each data stream. However, it is more likely the case that the programmer intended that all data streams are synchronized by a single top level periodic SQ that generates triggers for each data stream to fire. This differs as the earlier design does not require the streams synchronize their start together. Another interesting aspect was learning about different use cases for Plan B The camera hardware used in the application is unreliable, in the sense that it can become unresponsive and requires a reboot. This led to questions on how to use TTPython to address hardware flakiness and difficulties to debug the hardware while using TTPython. User defined functions in SQs are encoded as anonymous functions, so debugging becomes opaque as the programmer loses the line number where the bug occurred.

We have been in a considerable amount of group meetings with the Syracuse team starting from late summer of 2023. These were in part to both understand better the Intercity Flooding application and to flesh out how TTPython can support their needs. We recently had a meeting discussing the actual realization of the software and hardware stack for the application. Although we envisioned TTPython to be an inclusive tool in development, we realized that TTPython does not yet support LoRaWAN (Low Power Wide Area Networking), a popular networking technology in sensor networks. The programmer needs finer tune on the type of communication that devices will use, as LoRaWAN does not setup endpoint addresses for edge devices. TTPython would need to interface with LoRaWAN frameworks (such as Chirpstack) or operate on LoRa gateways to allow addressible edge devices.

4.2.2 Methodology

To collect data for these research questions, we will conduct a series of interviews, observations of the team, and code analysis after the application has been completed. These will be semi-structured interviews of the team members in Dr. Carter's group asking about their experiences before and after using TTPython on the project. The observations will include notes taken during check-ins and meetings with the team. Once the final application is finished, we will identify patterns and compare these with the ongoing observational notes throughout development. We will conduct a retrospective with the team to understand the challenges faced during development and how TTPython interacted with these difficulties.

4.2.3 Expected Results

For RQ 1, we expect to see that domain-scientists have a strong understanding of the model of the environment where they want to deploy their application, but struggle on realizing limitations in the system. For example, Dr. Carter is an expert in combining imaging and satellite data to identify flooding. However, we have noticed that their understanding of realizing this application onto actual hardware is quite different. For example, the USGS have expressed interest in shifting towards LoRaWAN due to their deployments being in locations where power is not guaranteed. This limits networking capabilities due to LoRaWAN's low data transfer speeds. LoRaWAN makes it difficult to implement features such as uploading a photo of the detected flooded area due to the size of transferring an image over the network. To remedy this, the application needs to support both LoRa and internet protocols, showcasing a trade off between time and power used to transmit an image. We expect to see more evidence of this through the semi-structured interview and through future meeting observations.

We expect to see that at the programming level TTPython helps abstract timing and distribution so users can focus more attention on developing their applications. TTPython requires programmers to first consider the high-level timing and distribution requirements before writing code while abstracting necessary boilerplate code required to coordinate and parallelize the execution. Evidence will come from interviews, observations of the team during meetings and programming tasks, and an analysis of the application once finished.

We have already seen some preliminary difficulties in using TTPython with our initial findings of specifying synchronized, periodic data streams. Other observations have also shown that although the Data-Validity Firing rule is important for the correctness of DT applications, the semantics are not intuitive. This could be caused by the fact that programmers are unfamiliar with a dataflow graph architecture.

Chapter 5

Normative User Study

5.1 Comparative User Study between TTPython and Classical Python Solutions

To validate TTPython's efficacy in abstracting timing and distribution concerns in DTS applications, we will conduct a user study with participants developing two sample programs. These two sample programs will have a corresponding solution in TTPython and Python with library support for timing and distribution, namely the Python `time` library and a message broker (RabbitMQ). We believe this to be worthwhile comparison as both case studies use Python and its basic time library to handle their DT application. Furthermore, many popular embedded frameworks use a publish-subscribe model to handle communication between distributed embedded devices.

TTPython's target audience includes domain experts unfamiliar with CPS-specific concerns found in DTS applications. The difficulty in this user study is to introduce participants to non-trivial DTS applications while abstracting most of the domain knowledge required to develop these programs. We expect that TTPython will show advantages both in describing timing and distribution requirements and in software evolution concerning these requirements. This is due to the fact that time is inherently coupled with data in tokens and communication is implicitly specified by variable call across SQs in the dataflow graph. Our study aims to answer the research questions listed below.

RQ 1: **Does TTPython increase the productivity of programmers when writing timing and distribution?**

RQ 2: **Do programmers using a message broker to handle communication between devices include more bugs where TTPython would catch at compile time?**

5.1.1 Methodology

We will solicit participants from CMU and acquaintances and select experienced Python programmers. Each participant will self-report their level of expertise in Python.

Each participant will use TTPython and Python to implement one application. Thus, each participant will first take a tutorial with periodic quiz questions to foster active learning. Participants will be randomly assigned both the order and which solution to use per application. Both applications contain two parts: the first in which the participant will be handed a finished application without timing or distribution requirements. This scaffolding comes from the idea that the application was designed and tested on a single device before wide-scale distribution, which should ease the burden of understanding domain-specific knowledge of each program. The programmer will then take this single application and appropriately add timing and distribution requirements through TTPython or Python, the details of which are specified in later subsections.

To make the development process as realistic as possible, we plan to provide simulated data streams and output monitors for participants to use to run and debug their applications. Both applications will use a simulated data source to provide a deterministic output. For the 1/10-scale CAV Intersection, we will have a 2D-graph plotting the movement of the cars through the intersection. Its data will be sourced from a sample data source provided by our collaborators at Arizona State University. The intercity flooding detection will have a command-line interface printing a log describing the output of the sensors. We will base our simulated data on prior flooding data taken from the USGS monitoring website. We will be measuring the time taken to complete the program, bugs encountered during development and in the solution, and lines of code for solution.

Task 1: 1/10th-scale Connected Autonomous Vehicle Intersection

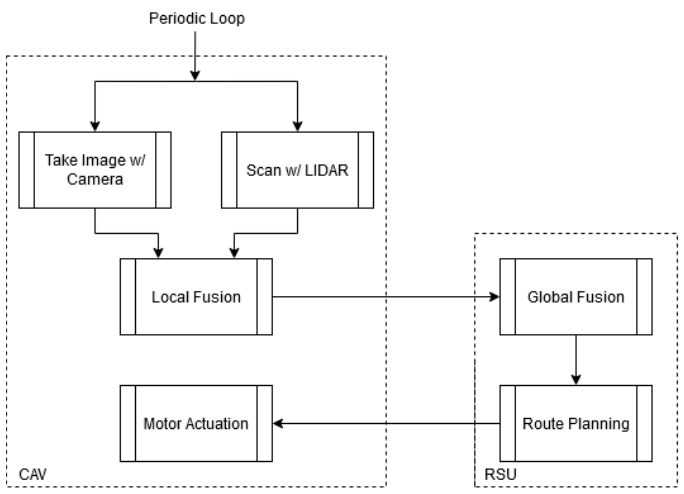


Figure 5.1: A dataflow graph of the solution to the CAV application.

Participants will write a 1/10-scale Connected Autonomous Vehicle Intersection application. The application consists of three types of devices: connected autonomous vehicles, infrastructure sensors, and road-side units. Each model autonomous car is equipped with camera and LIDAR sensors and performs local fusion to detect nearby objects. They then communicate with each other combined with an infrastructure sensor for global fusion to collaborate on the objects they detect. Global fusion occurs on the road-side unit, the host for this global state of the environment of the intersection. If there are disruptions in the periodic execution of 125 milliseconds, the cars will take cautionary procedures such as braking.

For the task, participants will be given this application designed to run for one device: namely an untimed for-loop containing local fusion and global fusion. Participants will have to both set timing requirements (of periodicity and matching-in-time data values) and partition the program to two devices: the car itself and a road-side unit. The dataflow graph of this program is seen in Figure 5.1

Task 2: Intercity Flooding Detection

Participants will develop a wide-area sensor application to detect flooding of intercity sewer systems. The application combines thermal and optical imaging with location sensors (GPS and IMU) to identify where flooding occurs within the sensor network deployed across the city. The GPS and IMU data will be coupled with the optical data to identify the location the image. If flooding is detected, the app will notify a server as a proxy to alert city officials to address the flooding.

Participants will initially write a sample program with Figure 5.2 dataflow graph in mind. Users will create three separate data streams: two for imaging and one for coordinate collection. The data streams will be used in a machine learning model as a function call to determine if flooding is detected. If it is, the device will then upload an image and coordinates to a remote server to simulate alerting city officials.

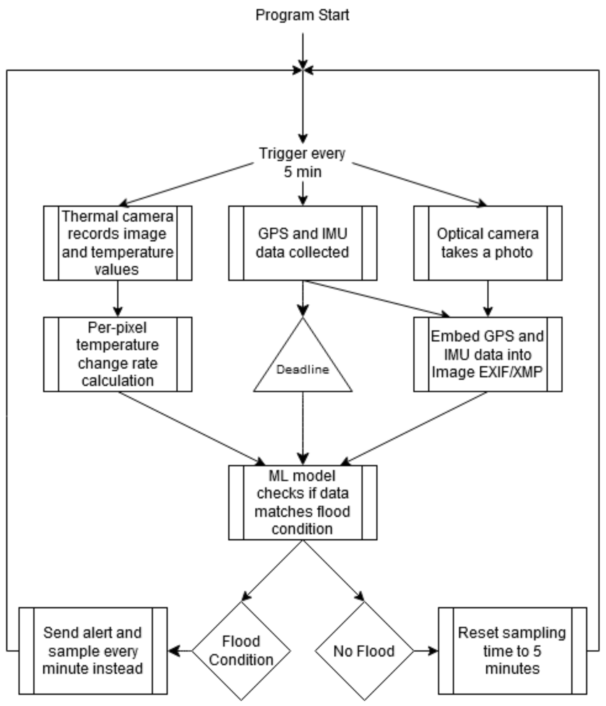

Program Start
Trigger every 5 min
Thermal camera records image and temperature values
GPS and IMU data collected
Optical camera takes a photo
Per-pixel temperature change rate calculation
Deadline
Embed GPS and IMU data into Image EXIF/XMP
ML model checks if data matches flood condition
Send alert and sample every minute instead
Flood Condition
No Flood
Reset sampling time to 5 minutes

Figure 5.2: A sample dataflow graph of the solution to the Flooding Detection application.

5.2 Expected Findings

We expect to see that TTPython will have fewer errors and faster completion. This is due to the abstractions that TTPython provides with distribution and timing. In DT applications, it is essential that data to be used together in computation share a temporal context (i.e. their timestamps are close together in time). The code to do so is difficult to write and manage, as this must be done consistently throughout the program and is required when sending data between devices.

We also expect programmers to have challenges with the dataflow semantics in Python. Classical Python is imperative, and our use of macroprogramming changes the logic of some keywords in Python such as `global`. The tutorial may mitigate some of these challenges.

Chapter 6

Proposed Contributions

My thesis will have made the following contributions:

- A defined syntax and semantics for a language and framework (TTPython) for an intermediate dataflow graph execution.
- Evidence of advantages of TTPython for abstracting timing and distribution concerns through a case study and user study.
- Evidence showcasing challenges domain-experts face when working in a cyber-physical environment.
- Evidence of how dataflow graphs can manage code evolution in DT applications.

Chapter 7

Proposed Timeline

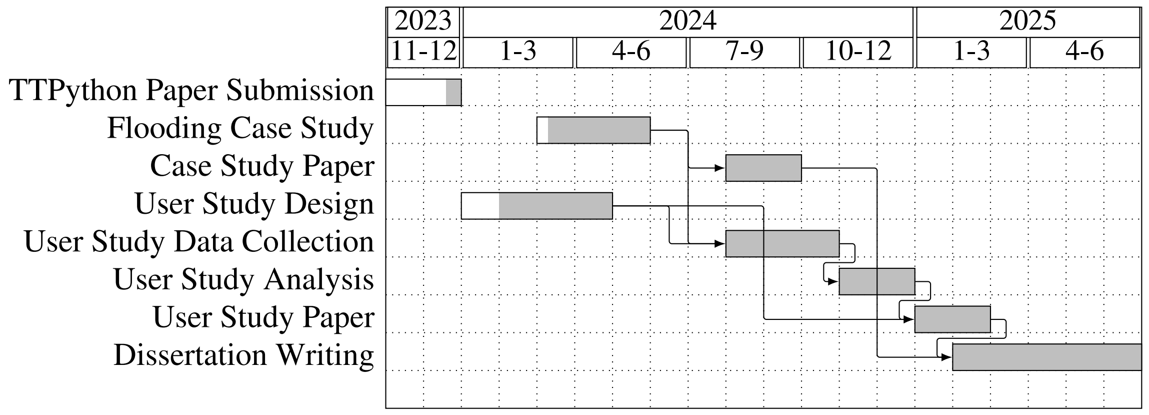


Figure 7.1: A breakdown of deliverables for the thesis. White is the estimated amount of work finished for the task, while gray is unfinished.

I plan to finish up and submit the TTPython paper to TOPLAS within this year. I am waiting on some new results on the implementation of Plan B. The case study and the user study can be done in parallel as they are not dependent on each other: the user study is simply using the intercity flooding case study as a practical, non-trivial example for a task. I have left some time in the summer on the case I take an internship. I expect that the user study will take 3 months to complete.

Bibliography

[1] Sidharta Andalam, Partha Roop, Alain Girault, and Claus Traulsen. Pret-c: A new language for programming precision timed architectures. 2009. 2

[2] A. Arvind and K. Gostelow. The u-interpreter. *Computer*, 15(02):42–49, feb 1982. ISSN 1558-0814. doi: 10.1109/MC.1982.1653940. 2

[3] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using cosmos. *ACM SIGOPS Operating Systems Review*, 41(3):159–172, 2007. 2

[4] Amol Bakshi, Viktor K Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. *Proceedings of the 2005 workshop on Endtoend senseandrespond systems applications and services*, (Eesr 05):19–24, 2005. 2

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 266–296, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74792-5. 2

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3): 1–22, 2013. 3.2.1

[7] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ISCA '75, page 126–132, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450373661. doi: 10.1145/642089.642111. URL `https://doi.org/10.1145/642089.642111`. 2

[8] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices*, 38(5):1–11, 2003. 2

[9] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, 2005. 1, 2

[10] Timothy W Hnat, Tamim I Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 225–238, 2008. 2

[11] Robert A Iannucci. Toward a dataflow/von neumann hybrid architecture. *ACM SIGARCH Computer Architecture News*, 16(2):131–140, 1988. 3

[12] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, USA, 1994. 4.1.2

[13] Mohammad Khayatian, Rachel Dedinsky, Sarthake Choudhary, Mohammadreza Mehrabian, and Aviral Shrivastava. R2im – robust and resilient intersection management of connected autonomous vehicles. In *proceedings of The 23rd IEEE International Conference on Intelligent Transportation Systems*, 2020. 4.1

[14] Mohammad Khayatian, Mohammadreza Mehrabian, Edward Andert, Rachel Dedinsky, Sarthake Choudhary, Yingyan Lou, and Aviral Shirvastava. A survey on intersection management of connected autonomous vehicles. *ACM Transactions on Cyber-Physical Systems*, 4(4):1–27, 2020. 1, 4.1

[15] Mohammad Khayatian, Mohammadreza Mehrabian, Edward Andert, Reese Grimsley, Kyle Liang, Yi Hu, Ian McCormack, Carlee Joe-Wong, Jonathan Aldrich, Bob Iannucci, and Aviral Shrivastava. Plan b - design methodology for cyber-physical systems robust to timing failures. *ACM Trans. Cyber-Phys. Syst.*, jan 2022. ISSN 2378-962X. doi: 10.1145/3516449. URL `https://doi.org/10.1145/3516449`. Just Accepted. 3.4.2

[16] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Operating Systems Review (ACM)*, volume 36, pages 85–94. ACM, 2002. doi: 10.1145/635508.605407. 2

[17] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20 (4):1–27, 2021. 1, 2

[18] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005. 2

[19] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):1–51, 2011. 2

[20] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 489–498. IEEE, 2007. 2

[21] Rishiyur S Nikhil et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1990. 1, 2, 3

[22] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, 2004. 2

[23] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer networks*, 52(12):2292–2330, 2008. 2

[24] Jia Zou, Slobodan Matic, Edward A Lee, Thomas Huining Feng, and Patricia Derler. Execution strategies for ptides, a programming model for distributed embedded systems. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 77–86. IEEE, 2009. 2