

BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

**THE SENSOR NETWORK WORKBENCH:
TOWARDS FUNCTIONAL SPECIFICATION, VERIFICATION AND
DEPLOYMENT OF CONSTRAINED DISTRIBUTED SYSTEMS**

by

MICHAEL JAMES OCEAN

B.S., Rutgers University, 1998

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

2009

© Copyright by
MICHAEL JAMES OCEAN
2009

Approved by

First Reader

Azer Bestavros, Ph.D
Professor of Computer Science

Second Reader

Assaf Kfoury, Ph.D
Professor of Computer Science

Third Reader

Ibrahim Matta, Ph.D
Associate Professor of Computer Science

Acknowledgments

While at Boston University, I have had the great fortune of working with some of the best colleagues and faculty members a graduate student could hope for.

I thank Professor Bestavros, without whom, this document and work would not exist. People occasionally ask, “Why did you choose the graduate program at Boston University?” and my answer is always, without hesitation, that I selected this program based on my interactions with faculty members when visiting the campus; I had invited myself to attend Sensorium meetings at BU and, at these meetings, I was immediately drawn to Professor Bestavros’ vision and enthusiasm. He has been the most amazing advisor, teacher, and advocate. My experience as his student has lead me to pity my friends and colleagues at other schools who do not have Azer in their corner.

I must also thank Professor Kfoury, for his help and patience with my foray into programming language theory. Ever since attending Professor Kfoury’s graduate PL course, I have been enamored with the elegance and grandeur of type theory. This affinity for PL is clearly reflected in this thesis work. Assaf has been like a second advisor to me, his contribution to this document, my love of PL, and my education overall, cannot be overstated.

My interest in Networks and Programming Languages formalism naturally drew me to the iBench group (*i.e.*, “Networks and Programming Languages can be friends”), where I had the pleasure of regular meetings with Professor Matta. Ibrahim, despite being typecast as a Networks faculty member, has been an advocate for the causes of the iBench group, of which my work is a subset.

My passion for Systems debates (of all kinds) were often satisfied by Professor Richard West, who has been a great colleague –not only as a source of criticism for my love affair with static verification techniques and the trappings of cushy, high-level programming languages– but also as a source of pointers to excellent British Science Fiction.

I also must acknowledge Professor Betke who has been a constant supporter of this work, and allowed me to develop the very first prototype of (what would eventually evolve into) SNBENCH as a project in her Image and Video Computing class.

Very special thanks go out to Adam Bradley who, while working as postdoctoral research associate at BU, seemingly conspired with Azer to get me back to working on SNBENCH. Much of the initial thrust of the re-design of the SNBENCH (from my first initial prototype) and my own regenerated excitement for the project can be credited directly to Adam’s involvement early on. Without a doubt, Adam also deserves acknowledgment for paving the way for the tag-team co-advising pair of Azer and Assaf.

This work (and indeed the preservation of my own sanity) would not have been possible without the help of my fellow students and very good friends. Specifically: Gabe Parmer for being a very knowledgeable sounding board for my various rants and always being willing to go for a cup of coffee, Kyle Burke for keeping things light and helping me re-align my brain to the more theoretical aspects of CS that I temporarily misplaced in my time in industry, and Ben Hescott for conveniently being one year ahead of me and always willing to impart advise based on his experiences.

Last, but certainly not least, I owe a great deal of gratitude to my family, and especially my wife, Elizabeth, for her constant support throughout this crazy, yet wonderful, ordeal.

**THE SENSOR NETWORK WORKBENCH:
TOWARDS FUNCTIONAL SPECIFICATION, VERIFICATION AND
DEPLOYMENT OF CONSTRAINED DISTRIBUTED SYSTEMS**

(Order No.)

MICHAEL JAMES OCEAN

Boston University, Graduate School of Arts and Sciences, 2009

Major Professor: Azer Bestavros, Professor of Computer Science

ABSTRACT

As the commoditization of sensing, actuation and communication hardware increases, so does the potential for dynamically tasked sense and respond networked systems (*i.e.*, Sensor Networks or SNs) to replace existing disjoint and inflexible special-purpose deployments (closed-circuit security video, anti-theft sensors, *etc.*). While various solutions have emerged to many individual SN-centric challenges (*e.g.*, power management, communication protocols, role assignment), perhaps the largest remaining obstacle to widespread SN deployment is that those who wish to deploy, utilize, and maintain a programmable Sensor Network lack the programming and systems expertise to do so.

The contributions of this thesis centers on the design, development and deployment of the SN Workbench (SNBENCH). SNBENCH embodies an accessible, modular programming platform coupled with a flexible and extensible run-time system that, together, support the entire life-cycle of distributed sensory services. As it is impossible to find a one-size-fits-all programming interface, this work advocates the use of tiered layers of abstraction that enable a *variety* of high-level, domain specific languages to be compiled to a common (thin-waist) tasking language; this common tasking language is statically verified and can be subsequently re-translated, if needed, for execution on a wide variety of hardware platforms.

SNBENCH provides: (1) a common sensory tasking language (Instruction Set Architecture) powerful enough to express complex SN services, yet simple enough to be executed by highly constrained resources with soft, real-time constraints, (2) a prototype high-level language (and corresponding compiler) to illustrate the utility of the common tasking language

and the tiered programming approach in this domain, (3) an execution environment and a run-time support infrastructure that abstract a collection of heterogeneous resources into a single *virtual* Sensor Network, tasked via this common tasking language, and (4) novel formal methods (*i.e.*, static analysis techniques) that verify safety properties and infer implicit resource constraints to facilitate resource allocation for new services. This thesis presents these components in detail, as well as two specific case-studies: the use of SNBENCH to integrate physical and wireless network security, and the use of SNBENCH as the foundation for semester-long student projects in a graduate-level Software Engineering course.

Contents

Acknowledgments	iv
Contents	viii
List of Tables	xii
List of Figures	xiii
List of Source Code Listings	xiv
List of Abbreviations	xv
1 Introduction and Motivation	1
1.1 Motivation	1
1.2 The Sensor Network WorkBench	5
1.3 Organization of this Document	8
2 Related Work	11
2.1 Classification By Abstraction	13
2.1.1 Complete Transparency	13
2.1.2 SN Programming “In the large”	15
2.1.3 Individual Sensor Programming	23
2.2 Research Opportunities in SNCPs	26
2.3 Conclusion	28
3 Functional Tasking Languages For Service Composition	29
3.1 The SNAFU Programming Language	30
3.2 Sensorium Task Execution Plan (STEP)	39
4 The snBench Architecture and Infrastructure	45
4.1 The Sensorium Service Dispatcher (SSD)	46

4.1.1	Resource Management	47
4.1.2	Scheduling and Dispatch of STEP Programs	48
4.2	Sensorium Execution Environments (SXE)	51
4.3	Implementation Overview:	57
4.4	Putting it all together	59
4.5	Conclusions	63
4.5.1	Catalytic Work	64
5	Safety Verification Through Programming Language Formalisms	66
5.1	Introduction	67
5.2	Formalism for “Core” STEP	72
5.2.1	Syntax of Expressions	73
5.2.2	Preliminary Definitions	73
5.2.3	Dynamic Semantics	75
5.2.4	Syntax of Types	79
5.2.5	Static Semantics and Sizing of Types	79
5.2.6	Soundness of Core STEP	85
5.3	Applications of the formalism	97
5.3.1	Additional Syntax	97
5.3.2	Typing and Evaluation Rules for Pairs	97
5.3.3	Additional Typing Rules for Image Manipulation	98
5.3.4	Flowtypes	100
5.3.5	In Practice	100
5.3.6	Implementation Details	105
5.4	Future Work	107
5.4.1	Additional Type Annotations	107
5.4.2	Applications to Image Pyramids	108
5.4.3	Cost and Size Signatures For New Opcodes	109
5.4.4	Different Models of Cost	109

5.4.5	Expanding to “Complete” STEP	110
5.5	Conclusions	110
6	A Native Execution Environment for Embedded Devices	111
6.1	Overview	112
6.2	Related Work	112
6.3	The STEP Virtual Machine	114
6.4	STEP Compilation	116
6.5	Benchmarks	129
6.6	Conclusions and Future Work	131
7	Case Study: Combined Physical and Wireless Network Security	134
7.1	Motivation	135
7.2	Related Work	137
7.3	Enabling Wireless Monitoring	141
7.4	Deployment Environment	145
7.5	Wireless Security Services	147
7.6	Future Work	150
7.7	Conclusions	153
8	Case Study: snBench as a Teaching Aide	162
8.1	Software Engineering Goals	164
8.2	Projects	165
8.2.1	STEP Programming GUI	166
8.2.2	Expanding SXE Capabilities	169
8.2.3	STEP Compilation	171
8.2.4	SXE Scheduler Modification	171
8.3	Conclusions	174
9	Conclusion and Future Work	178
9.1	Summary of Contributions	178
9.2	Future Directions	179

9.2.1	Resource Allocation	179
9.2.2	Extending SNBENCH to Other Domains	182
9.2.3	Security and Privacy	184
9.2.4	Safety and Verification	184
	Bibliography	186
	Curriculum Vitae	199

List of Tables

6.1	Opcodes for snCODE	117
6.2	Evaluation benchmarks comparing snCode and STEP	130
7.1	Alert events detected by Kismet and reported to snBENCH	143

List of Figures

1.1	The SN program life-cycle as enabled by <code>SNBENCH</code>	5
1.2	The thin-waist (hour glass) model applied to SN Programming Languages .	7
2.1	Organizing SN Construction Platforms by abstraction granularity	13
4.1	The SSD's Scheduling and Dispatch process	48
4.2	Partitioning STEP programs for dispatch	50
4.3	The SXE as a Virtual Machine	52
4.4	State Transition for the Evaluation of STEP Nodes	54
4.5	The SNAFU Integrated Development Environment	60
5.1	The SNAFU Compiler type checking a STEP program	106
5.2	The SNAFU Compiler size checking a STEP program	107
6.1	Evaluation benchmarks comparing <code>snCode</code> and STEP	130
6.2	Memory usage comparison between <code>snCode</code> and STEP	131
6.3	Garbage Collector invocations, comparing <code>snCode</code> and STEP	132
7.1	The graphical representation of STEP Program 7.7	156
7.2	The graphical representation of STEP Program 7.8	159
7.3	The graphical representation of STEP Program 7.9	161

List of Source Code Listings

7.1	Add an entry to a central log when a wireless alert is detected.	147
7.2	E-mail an administrator whenever a specific wireless alert is detected. . . .	148
7.3	Point a PTZ camera and return an image when a wireless alert is detected.	149
7.4	Functionally equivalent to Program 7.3, but uses black-box opcodes.	150
7.5	Get an image from the camera with the best coverage of a wireless alert. . .	151
7.6	Enable only the MAC addresses of user's whose faces have been recognized.	152
7.7	The STEP representation of SNAFU Program 7.1.	154
7.8	The STEP representation of SNAFU Program 7.2.	157
7.9	The STEP representation of SNAFU Program 7.3.	160

List of Abbreviations

SNBENCH	Sensor Network WorkBench
AP	Wireless Access Point
BNF	BackusNaur Form
HTTP	Hyper-Text Transfer Protocol
JavaCC	Java Compiler Compiler
nSXE	Native Sensor Execution Environment
SN	Sensor Network
SNAFU	Sensor Network Applications As Functional Units
snCODE	Sensor Network Byte-code
SNCP	Sensor Network Construction Platform
SRM	Sensorium Resource Manager
SSD	Sensorium Service Dispatcher
STEP	Sensorium Task Execution Plan
STEP _{vm}	Sensorium Task Execution Plan, Virtual Machine
SXE	Sensor Execution Environment
VM	Virtual Machine
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction and Motivation

1.1 Motivation

Traditionally Sensor Networks are thought of as a collection of programmable, yet impoverished resources that are tasked to complete relatively simple data acquisition activities (*e.g.*, passive habitat monitoring). Indeed, Sensor Networks are generally characterized equally by both their nodes' ability to observe (sense) their immediate surroundings (*e.g.*, light, temperature) and their relative lack of resources (battery power, CPU speed, memory, storage and bandwidth, *etc.*). A new view of SNs is emerging, in which SNs consist of a heterogeneous collection of sensing, actuation, and computing systems; the increased processing capability of (some) of the constituent nodes within the system enable a much more complex sense and respond activities including image processing in tandem with more traditional (*e.g.*, temperature and light monitoring) capabilities.

We envision the emergence of general-purpose, well-provisioned sensor networks—which we call “Sensoria”—that are embedded in (or overlaid atop) physical spaces, and whose use is *shared* amongst autonomous users of that space for independent and possibly conflicting missions. Our conception of a Sensorium stands in sharp contrast to the commonly adopted view of an embedded sensor network as a special-purpose infrastructure that serves a well-defined, fixed mission.

Our work (and indeed interest) is focused on the goal of enabling shared, dynamically tasked sensory infrastructures: physical environments augmented with a rich set of sense and respond capabilities that may be tasked simultaneously by both the owners and occupants of the space.

Many shared physical spaces (*e.g.*, Shopping Malls, Airports, Elder Care facilities) already enjoy some of the “low-hanging” benefits of sensor deployments, achieved via separate, disjoint, non-programmable sensing “networks.” In these environments most processing and response is performed by simple dedicated circuitry or humans (*e.g.*, a security guard watching a closed-circuit video feed). The goal of our work is to provide an acquisition, response, and computation infrastructure that completely replaces, and ultimately supersedes, these existing closed deployments.

To motivate this alternative “open system” view of a SN, consider a public space such as an airport, shopping mall, museum, parking garage, subway transit system, among many other such examples. Clearly, there are many different constituents who may have an interest in monitoring these public spaces, using a variety of sensing modalities (video cameras, motion sensors, temperature sensors, smoke detectors, *etc.*). In the case of a shopping mall, these constituents may include mall tenants, customers, mall security, local police and fire departments, *etc.* One alternative is for all these different constituents to overlay the public space they share with independent SN infrastructures – comprising separate sensors, actuators, processing and storage server nodes, and networks – each of which catering to the specific mission or application of interest to each respective constituent. Clearly, this is neither efficient nor practical, and for many constituents may not be even feasible. Rather, it is reasonable to expect that such a shared SN infrastructure will be an integral part of the public space in which it is embedded, operated and managed by an entity different from the entities interested in “programming” it for their own use.

By example, we consider a parking garage which already contains a closed-circuit video surveillance system that is monitored by the on-site security personnel to detect suspicious activity. The easiest way to transition this environment to a programmable infrastruc-

ture would be to connect a computer with digital video capture capability directly to the existing camera network; doing so essentially turns the existing closed-circuit camera deployment into an IP camera network. Once the video frames are available digitally, they can be processed by any number of image manipulation/recognition operations. A simple monitoring process might try to detect someone breaking a car window on the premises by computing motion vectors across the images and producing warning whenever a short, fast movement is detected. Audio sensors would also be effective in this environment, as loud noises would indicate that there is an event that requires attention (especially when coupled with the image processing data to further increase confidence). What makes the use of the programmable network so appealing, is the ease with which new services could be easily added, at least “in theory.”

Consider placing a new camera in full view of the license plates of cars as they enter and leave the parking garage. These data sources can be used to count the number of cars that have entered/exited the garage. This data can be used to estimate the number of remaining open spots in the garage (possibly updating some display to indicate when the garage is full), offer a pricing discount if the garage is far under capacity, allow police officials to occasionally query for stolen vehicles, or enable real-time queries in the event of a highly time sensitive crime (*e.g.*, an Amber alert). These example uses allude to some of the inherent needs of such an embedded sensory environment, including (but not limited to): (1) the ability to perform processing on multiple data sources, on more than one computer, with processing done potentially off site, (2) a notion of different task priorities (the tasks of the police take priority over the simple car counting example), (3) different tiers (or groups) of users who utilize the system, each with different device access, and (4) a notion of different time constraints (counting cars would likely have a strict (measurable) time latency bound, while querying for the entrance of stolen cars can probably be dealt with as time permits). A myriad of other potential examples are also possible (*e.g.*, allowing mobile users to check the number of remaining spaces, which floors have open parking spaces, or to query where they parked their car). We cannot underscore enough the need that arises

from the need to close the gap between *imagining* new potential, sensory applications and actually *implementing* and *deploying* these new applications.

Harnessing the power of such shared SN infrastructures will hinge on our ability to streamline the process whereby relatively unsophisticated “third parties” are able to rapidly develop and deploy their applications without having to understand or worry about the underlying, possibly complex “plumbing” in the SN infrastructure supporting their applications.

Low-level Sensor Network development requires the specification of highly synchronized and concurrent source code for each of the constituent nodes/roles in the larger SN application. This task is challenging for well-trained software developers and is cruelly out-of-reach for those who would see the most benefit from a SN deployment; security, civil engineers, elder care professionals, *etc.* In an attempt to lower the complexity of service composition and SN construction, a variety of works have emerged that explore extending language and common run-time support services to SNs as an alternative to low-level, device and role-specific code.

Existing SN research in this domain underscores the inherent challenges in making a system accessible to novices, while not simultaneously rendering the system impotent by over-simplification. We observe that the current offerings make many trade-offs, have tangible disadvantages, and ultimately bring about a strong sense of *deja-vu*.

Today, programming SNs suffers from the same lack of organizing principles as did programming of stand-alone computers some forty years ago. Primeval programming languages were expressive but unwieldy; software engineering technology improved with the development of new high-level programming languages that support more expressive and useful abstraction mechanisms for controlling computational processes, as well as through the wide adoption of common software development platforms that leverage these abstractions. For SNs, we believe that the same evolutionary path need not be (painfully) retraced, if proper abstractions, expressive languages, and software engineering platforms are developed in tandem with advances in lower-level SN technologies.

1.2 The Sensor Network WorkBench

To this end, this thesis presents the Sensor Network WorkBench (or SNBENCH). SNBENCH leverages type-theoretic concepts from the Programming Language (PL) community in tandem with lessons learned from traditional Operating and Distributed Systems work. SNBENCH embodies an accessible, flexible, and extensible programming platform and run-time system that support the entire life-cycle of distributed sensory applications. Our primary focus is *not* on optimizing the various algorithms or protocols supporting an *a-priori* specified application or mission, but rather on providing *flexibility* and *extensibility* for the rapid development and deployment a of wide range of applications.

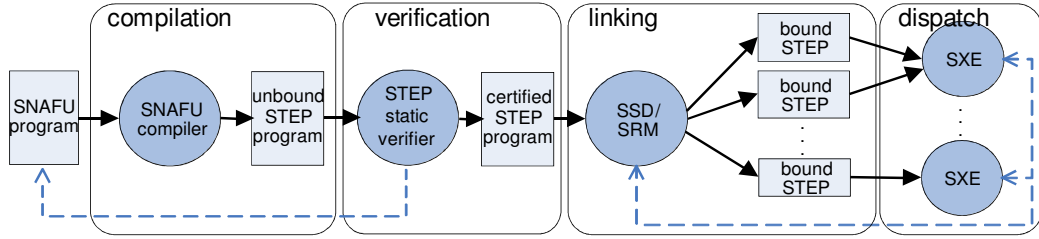


Figure 1.1: The SN program life-cycle as enabled by SNBENCH. Rectangles represent data, circles represent tasks/processes, and the dashed lines represent control communication (*i.e.*, dependency).

The SNBENCH framework allows Sensorium users to easily program, deploy, and monitor the services that run in this space while insulating the user from the complexity of the physical resources therein. SNBENCH provides a high-level programming languages with which to specify programs (services) that are submitted to the resource management component, which in turn disseminates program fragments to the run-time infrastructure for execution. SNBENCH is extensible by design insofar as new hardware and software capabilities may be painlessly folded into the infrastructure by its advanced users and those new capabilities easily leveraged by its novice users.

The components of SNBENCH are analogous to those commonly found in traditional, stand-alone general-purpose computing environments (language safety, APIs, virtualization

of resources, scheduling, resource management, *etc.*). High-level languages that support stateful, temporal, and persistent computation are *compiled* into an intermediate, abstract representation of the processing graph, called a STEP (Sensorium Task Execution Plan). STEP is a strongly-typed functional language that is *verified* for safety and resource requirements, then *linked* to available Sensorium eXecution Environments (SXE). A Sensorium Service Dispatcher (SSD) decomposes the STEP graph into a linked execution plan, *loading* STEP sub-graphs to appropriate individual SXEs and binding those loaded sub-graphs together with appropriate network protocols. The SSD may load many such programs onto a Sensorium simultaneously, taking advantage of programs’ shared computation and dependencies to make more efficient use of sensing, computation, network, and storage resources. SNBENCH provides each user access to his or her own “private” SN, easily tasked with new programs and handling the scheduling, deployment, and management concerns transparently (*i.e.*, offering each user an abstract, Virtual SN).

The architecture of SNBENCH has grown naturally from the properties we desire from the systems/services that we target with it. Foremost, the SNBENCH infrastructure aims to provide a modular sense and response platform with which SN services can be effortlessly composed and on which multiple, potentially competing, SN services can be executed simultaneously on heterogeneous resources. The central organizing principle of our system is the use of a common, data-centric tasking language that provides the “thin-waist” of a layered programming approach, wherein multiple programming abstractions may be layered atop each other to offer multiple abstraction granularities to users of varying sophistication simultaneously (Figure 1.2).

We conceive of a SN service as a flow of data through a series of atomic functions. As such, we are able to share data when multiple services require the same sensing resources simultaneously. We provide an “interactive” functional language (*i.e.*, a functional language with inherent temporal dependency) as our tasking language; the functional style fits data-centric programming well and dovetails naturally with the desire for formal methods to statically verify safety and resource requirements.

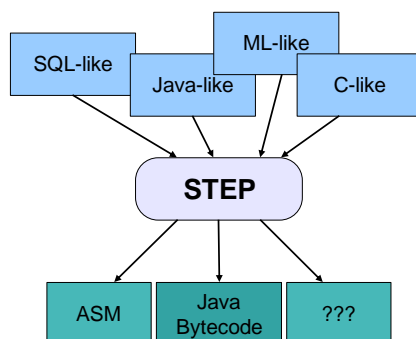


Figure 1.2: The thin-waist (hour glass) model applied to SN Programming Languages. We provide higher level languages that are compiled down to our common ISA, STEP. If a node cannot execute STEP directly, it can be recompiled into an alternate representation.

Distributing a SN service across a set of nodes requires partitioning the service into components to be placed on available resources (which is made easier by the functional nature of the tasking language), however the desire to support multiple SN services running simultaneously drives the need to also partition the SN resources to support multiple SN service components simultaneously. When a SN service is distributed across a resource set, the service programmer may not care which specific resource runs what computation in every case, so the resource allocation component must support services compositions are potentially both resource aware and resource agnostic (*i.e.*, if the programmer does not specify where a computation is to occur, it will be assigned to any resource that exists and is able to run his or her computation). These needs motivate explicit resource annotation in our tasking language as well as extensions to our type checker to statically determine an expression’s resource demands and execution delays for resource allocation.

Finally, as the SNs targeted by SNBENCH are not limited to collections of homogeneous resources, SNBENCH must provide a common tasking abstraction and execution environment that abstracts away the unimportant nuances of different hardware components while exposing the unique capabilities of these devices to the programmer (and infrastructure, at large). Our type system must not only infer data *types* but also must extract the relationship between data *sizes* and resource availability, potentially suggesting changes to the program to improve performance or enable execution, in general, given resource availability

(*e.g.*, suggest a specific image resolution or the use of a specific sensor that supports that resolution) based on contextual use or timing constraints).

The work in this thesis has been developed in response to the needs and requirements detailed above.

1.3 Organization of this Document

What follows is an overview of the individual chapters of this document.

Chapter 2: Related Work

This chapter presents a survey of other works that are related to SNBENCH *as a whole* (*i.e.*, other Sensor Network Construction Platforms). Some individual chapters of this thesis also contain their own, individual related work sections to specifically consider works that are related to the contributions described therein.

Chapter 3: Functional Tasking Languages For Service Composition

This chapter presents the two tasking languages provided within SNBENCH.

We present the functional-style common tasking language (*i.e.*, *virtual* Instruction Set Architecture), Sensorium Task Execution Plan, or STEP. STEP is the unifying, thin-waist language in our tiered tasking model (Figure 1.2). STEP is used to task both the larger SN and the individual execution environments within the SN. STEPs chain together core capabilities supported by the runtime environment of the SN to form a logical path of execution. Conceptually, a STEP instance is viewed as a directed graph of the computations requested of a single computing element. Given a program’s representation as a graph, detecting parallelization and other optimizations become tractable.

We also detail the high-level, SN-scoped programming language (SNAFU) and the compiler that produces STEP code from SNAFU programs. SNAFU exists to ease development and allow the developer a clear shift toward resource agnostic programming (*i.e.*, program-

ming the network rather than the nodes). SNAFU is merely one instance of a higher level, domain specific language that can target compilation to STEP. It is a layer of syntactic sugar and provides significant convenience over programming in STEP alone.

Chapter 4: The snBench Architecture and Infrastructure

This chapter describes the run time components of the architecture that provide resource management, task assignment, and the execution environment for participant nodes.

The Sensor eXecution Environment (SXE) is the run-time virtualization environment for heterogeneous computing and sensing resources; it is a common runtime that provides sensing and/or computing elements of the SN the ability to interpret and execute dynamically assigned task execution plans. In part, the execution environment hides irrelevant differences between various physical components and exposes a unified virtualized interface to their unique capabilities.

There are two key run-time support components that monitor SN resources and schedule newly submitted STEP programs to available SN resources. The Sensorium Resource Manager (SRM) is responsible for maintaining a current snapshot of the available resources in the SN, while the Sensorium Service Dispatcher (SSD) is responsible for accepting new STEP programs for the SN and scheduling the tasks of the STEP to available resources. Optimally, the scheduler must identify tasks within the new STEP that match currently deployed tasks and reuse them as appropriate and find a partitioning of the STEP program that can be deployed to physical resources.

Chapter 5: Safety Verification Through Programming Language Formalisms

This chapter describes a static-analysis methodology that provides type safety checks as well as data size bounding and resource requirement estimation on STEP instances. The multi-dimensional sized type checker tracks and constrains the upper and lower size bounds for data and, in doing so, can recommend resolution ranges for image processing data, check if a feasible data size exists to satisfy the temporal requirements of a program, and carry

size constraints from the sized data to the sensors that generate it. The ability to infer constraints on data and the sensors that produce it is essential to facilitating the resource agnostic programming model.

Chapter 6: A Native Execution Environment for Embedded Devices

Resources that are highly constrained and have (soft) real-time constraints (*e.g.*, robotics platforms) may require finer grained control over their resource utilizations. To that end we provide an alternate execution environment and instruction set architecture for these devices. This work consists of a small, virtual machine (the STEPvm) and a compiler that takes STEP instances and produces functionally equivalent representations in our lightweight ISA (snCode).

Chapter 7: Case Study: Combined Physical and Wireless Network Security

This chapter presents an instance of how SNBENCH can empower its users to capitalize on the SN resources they have, in order to develop novel, previously hard to conceive of, sense and respond application. In particular, specific details are given as to how SNBENCH can be used to bridge the gap between physical security (*i.e.*, CCTV surveillance) and wireless network security. We argue (by example) that security on both planes (the physical and the virtual) benefit from the data provided by the other.

Chapter 8: Case Study: snBench as a Teaching Aide

This chapter details experiences over the course of three years using SNBENCH as the basis for semester-long group projects in a mixed graduate and undergraduate software engineering class. Much of the work submitted as part of these student projects has been adopted back into the SNBENCH code base, in some form or another, including a graphical programming development environment, alternative scheduling policies for the sensor execution environment, and a myriad of image manipulation functions (opcodes).

Chapter 2

Related Work

In much the same way that a Distributed Operating System provides scheduling and management services to collect disparate and disjoint hardware into a single programmatic interface, SNBENCH aims to provide the same functionalities for sensing and responding environments. In our consideration of related work, we consider other works that ease the development and deployment of Sensor (and Responder) Networks. We label such works as Sensor Network Construction Platforms (or SNCs); all of which provide mechanisms that ease SN construction and deployment (*e.g.*, high(er)-level languages, abstractions, APIs, run-time support services, libraries). SNCs allow SN services (or SN instances) to be constructed at a higher-level by leveraging an underlying run-time or library to provide functionalities commonly required of SN nodes/services (*e.g.*, network topology discovery and maintenance, routing, naming, sensory data acquisition and manipulation, task propagation and scheduling). Unlike the previous generation of SNs that were hand coded to optimally achieve a single predetermined goal, SNCs enable Sensor Networks that can be shared and are dynamic in function (*i.e.*, executing multiple tasks simultaneously, and capable of re-tasking after the initial deployment).

Sensor Network Construction Platforms provide the means to describe new sensory services and the runtime components that enable the individual sensing/computing hardware (or hardware abstraction layer) to execute these sensory services (*i.e.*, analogous to middle-

ware in a software stack). What differentiates an SNCP from a rich operating system for sensors is that (1) rather than providing a single, fixed solution/implementation for common SN tasks, the implementations may be adjusted (*i.e.*, are modular) to suit the requirements of the SN services they execute and (2) the composition and orchestration of the SN service is supported from a perspective that is higher-level than that of the single node.

For example, TinyOS [HSW⁺00], is a small efficient event driven “operating system” that allows programs written in the nesC programming language to access the hardware of the device (*e.g.*, radio, timers, sensory devices). Programming in nesC for TinyOS is naturally low-level and application/role specific. As such, it is not considered an SNCP in and of itself, however several systems constructed *on top* of TinyOS are considered as SNCPs.

We organize our discussion of these related works by their particular abstraction granularity; ranging from those that provide services to ease programming at individual nodes (*e.g.*, libraries for nesC) to those that abstract away the SN entirely (*e.g.*, represent the SN as Database). Figure 2.1 illustrates the complete range of abstraction granularities and classifies various Sensor Network Construction Platforms in that spectrum. The reader should note that not all of the works presented were designed to be SNCPs as their primary goal (*i.e.*, many have been implemented to highlight and address other specific research issues) and as such criticism of the adequacy and/or completeness of these works as SNCPs is unfair, and is not our intent. Instead we present these works as the state-of-the-art in their progress and contributions toward the individual goals required to provide a complete Sensor Network Construction Platform.

Relation to Distributed and Grid Computing

Providing a unified interface and infrastructure to allocate, task and maintain distributed Sensor Networks is a goal similar to that of Distributed Operating Systems and Distributed Computing (*e.g.*, Grid computing), however there are significant differences in the tasks targeted by emergent Sensor Networks. SNs (unlike Grid environments) consist of heterogeneous devices with non-uniform constraints and capabilities; these capabilities (*i.e.*,

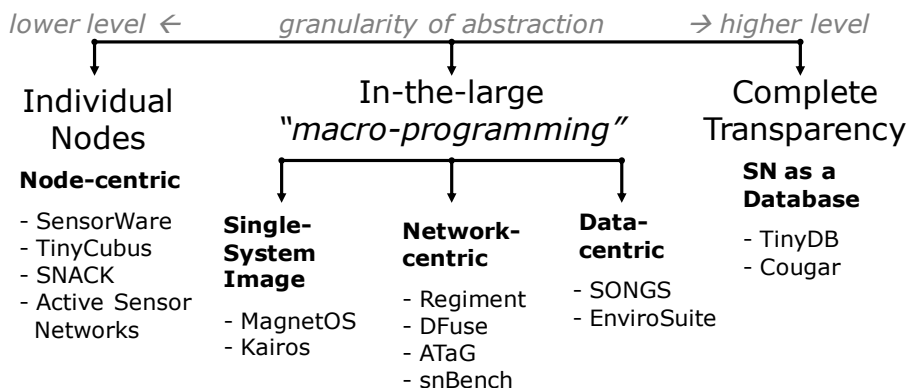


Figure 2.1: Organizing Sensor Network Construction Platforms by their abstraction granularity.

Sensing and Actuating) are themselves novel to the SN environment. Additionally, in traditional Distributed Computing scenarios, submitted jobs tend to be batch jobs (*i.e.*, CPU intensive) while those tasks targeted by Sensor Networks are diverse, and may consist of highly I/O intensive tasks (*e.g.*, streams of video). Distributed Computing jobs tend to be long-running while SN tasks may be persistent or potentially quick, transient requests. Related to this, in a traditional Distributed Computing environment with long running batch jobs, there is not a focus on minimizing the latency between new job submission and execution, whereas in a SN environment, short, event-responsive tasks must be deployed as quickly as possible.

2.1 Classification By Abstraction

2.1.1 Complete Transparency

SN as a Database

The works in this section are at the extreme end of the transparency spectrum. Just as many Distributed OSes strive for *complete* transparency (*i.e.*, the user is unaware that the underlying system is actually composed of several systems) the Sensor Network Construction

Platforms presented here provide data acquisition SNs that abstract the network as a single Database containing the data available at the sensors. At this level of abstraction, “tasking” the SN is limited to inserting new queries for existing data and simple in network processing (aggregation) on that data.

TinyDB

TinyDB [MFHH05] provides sensor nodes with TinyOS/nesC code that (1) participates in the routing and processing of SQL style queries and responses and (2) performs simple in network data processing and aggregation. Each node exposes sensory data via a table (named sensors) that contains a column for each type (*i.e.*, modality) of sensory data (light, temperature, *etc.*) available in the network. This table must be configured prior to deployment and the width of the table cannot increase during runtime resulting in an inflexible SN with respect to adding new sensors or sensory data types after deployment. To the credit of TinyDB, this work does abstract away the SN while still providing SN-centric controls, *e.g.*, augmenting the SQL-like query interface with the ability to specify persistent queries with explicit periodicity (*e.g.*, `SELECT .. SAMPLE PERIOD 30s`), event based queries, and adaptive node behavior and message routing aimed to help reduce battery consumption. The Tiny Application Sensor Kit (TASK) [BHH⁺05] is built on TinyDB and provides a “turn key” sensor network distribution suite including device monitoring and reporting tools for novices wishing to deploy SNs.

Cougar

Cougar [YG02] is similar to TinyDB in that it also abstracts the SN into a single SQL style database. Rather than exposing tables at the nodes, Cougar stores sensor information at the nodes in a custom abstract data type (ADT). Cougar uses a centralized task dissemination strategy that sends queries from and returns ADTs to a central tasking entity which ultimately exposes the database-style interface. Unlike TinyDB, Cougar lacks in-network aggregation. Cougar shares many criticisms in common with TinyDB; particularly difficult

sensor adaptability (evolution) without re-deployment and a fixed notion of sensor node data types, however the use of a custom ADT would theoretically make it easier to support new sensors, data and types after deployment (*i.e.*, localizing most of the required changes to the central tasking entity).

2.1.2 SN Programming “In the large”

At this granularity of abstraction, Sensor Network Construction Platform provide the capability to task the SN with expressive programs in the scope of the SN as a whole, describing the program in terms that span/address the SN rather than the individual nodes. As opposed to “micro-programming” the individual nodes (*i.e.*, providing device and role specific micro code), this approach has been referred to as “macro-programming.” These works differ from the DB approaches discussed previously in that they go beyond the data-acquisition functionality to offer arbitrary programmability.

SN as a VM

MagnetOS

MagnetOS [LRW⁺05] provides a thin layer Operating System to each node of the SN to pool the resources into a power-aware, adaptive *single* Java Virtual Machine (JVM) image. Magnet raises the bar for SN programmability beyond simple SQL queries by enabling dynamic programming of the SN through single-system, event-centric Java programs. A static partitioner (injector) divides the Java program among the nodes via binary rewriting using boundaries determined by object instances or method calls. Magnet allows for explicit annotation within the Java program to allow programmers to define boundaries and object placement. To increase network longevity MagnetOS provides algorithms that migrate object hosting one hop at a time (NetPull) or multi-hop (NetCenter) to minimize network transmission. Run-time components (LinkPull, PeerPull) monitor network traffic to determine candidate nodes for object migration while network structure components (NetCluster, TopoCenter) determine the connectivity and structure of the network.

While the use of the standard Java programming language as a programmatic interface to a SN implies no special considerations for SN programming, the authors do provide several sensor-level abstractions that expose SN-specific run-time state (*e.g.*, Node, Link, NeighborSet, Energy) and provide alternate SN-centric versions of existing abstractions (*e.g.*, Lock and Thread). Requiring Java interpretation at every sensor node introduces runtime overhead and additionally it is unclear how sensor-specific operations are provided (device drivers, for example). Ultimately one may argue that a JVM is not an accessible programming abstraction for a SN and, given some examples of program size from the paper in which MagnetOS is described [LRW⁺05] (*e.g.*, Audio Event Detection is 868 lines, Video Multicast at 222 lines), it seems unlikely that novices would be able to pick up Magnet programming with great ease.

Data-centric

In this section we consider works that task the SN in terms of the sensory data to be tracked, either directly or via abstract objects defined from composite sensory data.

SINA

Sensor Information Networking Architecture and Applications (SINA) [SSJ01] treats the nodes of its SN as taskable entities using a the Sensor Query and Tasking Language [JSS00]. SQTL is an sensor tasking protocol that packages execution directives along with a declarative style program for the individual participant sensors. In SINA a front end node provides an SQL-like user interface augmented with some tasking capabilities, produces SQTL messages to task the SN, performs data aggregation, and presents data to the end user. Sensors host the SINA execution module (also called Sensor Execution Environment or SEE) that runs SQTL and maintains an associative spreadsheet containing information about the local node and programmatic data storage. To utilize the SN in tasks beyond simple responses to data acquisition, the programmer must specify node-centric programs embedded into SQTL. SINA's SEE provides message routing and neighbor discovery on the nodes that,

when coupled with the SCTL programming abstractions provided for nodes, sensors and neighborhood, eases node-centric development on SINA. Ultimately SINA is a sensor-centric programming environment that has been augmented to support simple query-response tasks.

EnviroSuite

EnviroSuite [LAHS06] is the current evolution of EnviroTrack [ABC⁺04], an object-based development platform geared towards tracking objects in the physical environment. Entities and locations in the physical environment are defined by their observable (sense-able) characteristics. Programmers define objects by building a meta-specification including name (*type*), the conditions that define when the object is found (*context*), the measurable properties of the tracked object (*attributes*) and *methods* that may be performed on the object. EnviroSuite contains the environmentally immersive programming library for nesC (EIPLib) that provides common sensing, processing and communication functionalities while the EIP compiler (EIPLC) produces nesC code for the individual sensors given object definitions. Naturally, not all conditions and functionalities exist in the libraries, such that high-level specifications may also require nesC code to introduce new functionalities. The object abstraction of EnviroSuite is actually quite similar to the Semantic Streams notion in SONGS which is discussed below.

SONGs/Semantic Streams

In the SONGS framework (Service-Oriented Network proGramming of Sensors) [LZ05] “programs” are specified as logic based queries that define abstract entities defined by sensible features. Given these so-called ontological definitions of the entities of interest, an entity query will be translated by the service planning component of the infrastructure to generate a Semantic Stream [WZL06]. The semantic stream is a directed acyclic graph (DAG) that describes the flow of relevant data and any manipulations that are needed on the data path. A semantic stream is composed of SONGS service components (implemented in .NET) that specify their preconditions (input types) and post-conditions (output types). Service em-

bedding tries to fit the semantic stream on to the current state of the system, deploying parts of the computation to available resources. As the semantic stream for one entity may share common observable characteristics across other entities (or queries), the service embedding stage attempts to re-use existing deployed sensing operations or manipulations across semantic streams. Service planning uses a logic inference engine to meet the requirements of the high level semantic service. The SONGs architecture assumes a heterogeneous SN architecture, using sensors purely for data acquisition while more powerful nodes perform the roles of data processing (field servers) and service planning and embedding (gateway servers). While the SONGs approach does seem rather accessible, it shares the same de-traction of all the works surveyed in this category, that the programming model is more data-acquisition and simple response oriented and it does not seem to support expressive programming of the SN without delving into the source code at the nodes (in this case, the service components).

Network-centric

In this section we consider works that task the Sensor Network in terms of the desired behavior of the SN as a whole. Unlike the previous classification, these works enable the SN to be programmed with tasks that go beyond simple data-acquisition, while maintaining the benefit of not programming individual nodes. While we have placed our own `snBENCH` framework in this classification (Figure 2.1), `snBENCH` ultimately transcends a rigid classification by abstraction insofar as it enables development to occur at any point in the spectrum (*e.g.*, at any height of abstraction).

snBench

`snBENCH`[OBK06] provides a high-level functional-style programming language to describe SN behavior that is compiled into an intermediate data flow task graph and partitioned across the shared physical resources of the SN infrastructure. The `snBENCH` programming language, `SNAFU` (SN Applications as Functional Units), is a high-level strongly-typed

functional language that supports stateful, temporal, and persistent computation, a cycle-safe recursion construct and annotation of run-time properties (called flow types) SNAFU is *compiled* into an intermediate, abstract representation of the processing graph, called a STEP (*Sensorium* Task Execution Plan). A STEP, serialized as XML, contains nodes that express the resources, values, and sampling and computation operations required to execute the program. The STEP graph is then *linked* to available *Sensorium* eXecution Environments (SXE) possibly taking into account user requested bindings, where appropriate. A *Sensorium* Service Dispatcher (SSD) decomposes the STEP graph into a linked execution plan, *loading* STEP sub-graphs to appropriate individual SXEs and binding those loaded sub-graphs together with appropriate network protocols. The SSD may load many such programs onto a *Sensorium* simultaneously, taking advantage of programs' shared computation and dependencies to make more efficient use of sensing, computation, network, and storage resources.

SNBENCH has been designed with extensibility in mind and, as such, provides a clear path (interface) to extend the computations supported within STEP, the sensory data types (*i.e.*, modalities) manipulated by STEP, and the computation and sensory hardware that comprise the larger system. To facilitate this flexibility, the run-time components (*e.g.*, SXEs) are implemented in Java, yet this comes at the cost of less-predictable processing overhead that may be inappropriate for some time sensitive operations. As a solution to this potential fault, compilation to a lightweight VM for embedded devices or time sensitive tasks is offered (and is described in Chapter (6)).

Unlike other works considered, SNBENCH advocates a tiered programming model, such that multiple high-level languages can be developed and compiled into the common tasking language (*i.e.*, there is not one single interface to task the network). At the narrow-waist, the common tasking language (STEP) can be used to task either individual nodes or a set of nodes; SNBENCH enables development at whichever height of abstraction is suitable for the target audience.

SNBENCH is the only work to employ static verification of sensory services and provides a formal type theory to perform static analysis for safety *and* resource requirements analysis given a STEP instance. Not only is this the only SNCP to employ static verification of Sensory tasks, doing so on the common target (tasking) language enables static verification to occur on the generated tasks regardless of the particular source (composition) language or level of abstraction.

Regiment

Regiment [NW04] describes a functional language for SNs that treats the Sensor Network as a data structure. The authors cite the benefits of a functional approach including that straightforward manner in which parallelism can be extracted, that computation can be migrated/replicated without effecting evaluation and that hiding state manipulations from the programmer allowing the compiler to decide how state should be stored. Regiment is a fully expressive programming model and allows the composition of functions that pass functions in as arguments (an unrestricted functional language). Functional programs composed in Regiment may compute upon and derive events from the abstractions of the SN's elements, space and time. Each sensor is represented as a typed data object (Node) that exposes state, a Space is a collection of Nodes that can be assembled based on common characteristics of Nodes and a Region is a stream of Spaces. Several useful constructs are provided in the Regiment language to manipulate this data including aggregation (fold), a map operation, filter, and several conditional event handlers (when, whenAny, whenPercent).

Regiment programs are executed via compilation into a Distributed Token Machine (DTM) model [NAW05] in which data, synchronization and instructions are passed around the system in tokens (*a la* Active Networking [SN04]). In Regiment's DTM, the entire program (DAG) is placed in the token (like a stack) and the nodes pass the remaining parts of the computation along with any required results computed in a token to the node(s) that should take the next action from the stack. At present only a subset of the language can be processed by the compiler and the output runs in a simulator rather than on a live system,

however the larger concern is that this framework assumes that the SN runs a single service and does not seem to accommodate multi-tasking the SN.

Kairos

Kairos [GGG] provides a compiler and run-time support to provide macro-programming of SNs in the same spirit of distributed memory. In Kairos the programmer specifies a single high-level program in the extended C language Pleiades [KGMG07] that contains code for logical groupings of sensors that may read/write variables at other nodes, iterate through a nodes immediate neighbors and address arbitrary nodes. To specify behavior at multiple nodes, the procedural single-image Pleiades program includes a concurrent-for (`cfor`) construct that is executed simultaneously on all elements within the for loop.

To facilitate remote variable reads and writes Kairos implements a loosely consistent distributed memory model that is handled by a run-time library that transparently manage message passing, internal threads and object bookkeeping. The single “high-level” program contains abstractions to iterate over nodes and the code within the iteration body is the node specific actions to be taken on those nodes (potentially including iteration through neighbors). A preprocessor generates the node specific code in the native language of the individual node, relying on the Kairos run-time libraries to fill in data exchange and neighbor location. Although [GGG] provides several examples of single image programs, the presence of code for individual nodes within this global code is somewhat awkward and seems to defy the expectation of ease in Macro-programming (namely that node specific code need not be provided). On the other hand, the authors newly published Pleiades language seems to simplify code specification and readability significantly.

DFuse

DFuse [KWA⁺03] is an architecture for programming streaming data applications focused on optimizing function placement (*i.e.*, node role assignment) against several different cost functions (*e.g.*, bandwidth, network longevity). A DFuse program consists of a collection

of fusion functions and a data flow graph that indicates how data flows through the fusion functions. Fusion functions may concurrently manipulate multiple streams of different data types simultaneously (hence “fusion”) and each non-sink function will produce one or more streams of data (*e.g.*, a traditional aggregation would produce a single value stream from several streams). Fusion function placement is a heuristic based approach where in nodes decide whether or not to migrate to neighbor nodes given local information about node “health” (where health is a particular feature of interest).

Fusion function implementation is C-based, but eased by the Fusion API, a library of common capabilities including structure management (*i.e.*, fusion communication channel establishment and access), correlation control (*i.e.*, filters to modify how data arrives in the channel), computation management (*i.e.*, the specification, application and migration of a fusion function), memory management, failure and latency handling and a status and feedback channel for inter-function or function-to-device control. Although fusion function implementation is low-level and role specific the infrastructure satisfies node role assignment with respect to a requested optimization such that, assuming a library of fusion function implementations, DFuse *could* provide a macro-programming interface in its application task graph.

DART/Abstract Task Graph

The Abstract Task Graph [BPRL05] is a macro-programming model that is a mix of imperative-declarative programming style and data driven program flow. ATaG is a graph in which the nodes are tasks that produce or consume some data and the edges are data dependency (as in DFuse and SNBENCH). Abstract tasks are implemented by programmers in lower-level languages and otherwise programming is done graphically in the Generic Modeling Environment by connecting and annotating tasks in ATaG.

Execution of an Abstract Task Graph is supported by the Data-driven ATaG Run-Time, DART [BPP05], and the ATaG system features compiler support to produce node specific tasking. DART consists of several modules; NetworkStack handles inter-sensor commu-

nication, NetworkArchitecture performs neighbor/location to node ID resolution, ATaG-Manager maintains the Task Graph, data channels and scheduling information, DataPool provides access to shared variables, Dispatcher exchanges data across DataPools, and finally UserTask is the actual code (instantiation) of a particular abstract task. Unlike DFuse and SNBENCH in which the respective connected functions communicate values directly between each other, DART uses a local DataPool at each sensor run-time to exchange values between functions on the local node and the Dispatcher sends data across remote DataPools. The DataPool concept is present in the ATaG as well, which makes programs appear somewhat complicated compared to a straight-forward data flow.

While at present DART runs on top of Java, there is reference to a μ C/OS-II implementation that is in progress. Details are relatively slim on where ATaG compilation occurs, whether it is performed on-line or offline and if multitasking of the SN is supported. As no details are given regarding automated task assignment we assume the service programmer manually assigns tasks to nodes by annotation.

2.1.3 Individual Sensor Programming

The following works provide libraries and run-time support to ease tasking at the granularity of individual nodes/sensors.

TinyCubus

TinyCubus [MLM⁺05b], is a SN support infrastructure that provides data management services, a cross-layer framework and configuration engine that, when given the specification of a SN program will produce a tailored library configuration including implementations of common SN tasks, a data-sharing platform and task assignment. Although the overall desired program behavior is defined at a high level including optimization parameters (*e.g.*, energy, latency, bandwidth), application requirements (*e.g.*, reliability), and system parameters (*e.g.*, mobility, node density) the development of the SN application still requires node-centric programming. More details about TinyCubus may be found in [MLM⁺05a].

SNACK

The Sensor Network Application Construction Kit (SNACK) [GKE04] aims to promote code reuse, both within SN deployments (SNACK service composition allows annotations for event path merging) and to establish programmer libraries of stock implementations. SNACK does not deal with task assignment or resource management issues, but rather aims to aide in the construction of a single SN application. SNACK provides a library of common SN tasks, linked together with a high-level role specific (*e.g.*, for individual motes) composition language. The language provided allows developers to describes components and their corresponding interfaces and is ultimately compiled into nesC. SNACK does not aim to lower the bar for SN composition to novice users; the service composer must still understand the event-driven programming paradigm of nesC as well as concurrency and synchronization issues. The authors do allude to the fact that some higher-level languages/interface could be created to interface with SNACK.

Active Sensor Networks

The Application Specific Virtual Machines (ASVM) approach presented in [LGC05] eases Sensor Network programming by leveraging the authors prior work on Mate [LC02] which provides Virtual Machines for sensing elements (motes). The ASVM run-time runs on TinyOS and provides a core VM (static, middle-ware part) that can accept, schedule and execute “capsules” that contain specific VM instances (a nesC bytecode translation for the program and a support library implementing any functionality beyond the basic VM that is needed by the bytecode). Programs are event driven, subscribing to events generated by the hardware (including timers) to perform their computation. The compiler (*i.e.*, tool-chain) produces the bytecode and specific VM capsule generated from three provided higher-level node-centric languages (*i.e.*, higher than nesC). The core VM accepts ASVM capsules over the network and schedules them for execution. In the distribution mechanism, all nodes run the same code and messages contain both the data communication as well as the computations required to perform the task (*a la* Active Networks). Because scheduling of programs

by the basic VM supports preemption it is theoretically possible to have concurrency on these devices such that it would be possible for an infrastructure to run multiple simultaneous sensing tasks (see Melete, below). This work does not address issues of resource management, fine grain code deployment, or computational reuse.

Melete

Melete [YRBL06] is similar to the Active Sensor Networks work described in the previous section in that it is based on Mate, however this work adds fine grade task assignment and concurrent execution. The authors extensions to Mate add multi-program support (concurrency) and limited program isolation. Code dissemination is achieved using Trickle [LPCS04] to perform selective flooding and programs are composed in TinyScript one of the node-centric high-level languages supported in Active Sensor Networks.

SensorWare

SensorWare [BHS03] provides a dynamically tasked SN that is programmed at the granularity of sensors using an event-based scripting language and run-time support. SensorWare scripts are written in TcL and provides APIs to access sensing, communication and code propagation (replication) services. While [BHS03] also discusses resource management issues, such as computational reuse and automated task assignment, these issues are presented as questions without answers. It is unclear to what extent this resource management work is complete.

Squawk JVM

The Squawk JVM [SCC⁺06] is a Java Virtual Machine that runs on the bare hardware of the sensing device without an intermediate OS. Squawk allows the programming of the individual nodes of a SN using the familiar Java programming language but, of course, offers nothing in terms of automated resource management. While the Squawk JVM has

a lot of promise in lowering the bar for device specific tasking, Squawk only supports the Sun Spot [Smi07] device at this time severely limiting the applicability of this work.

2.2 Research Opportunities in SNCPs

While many works exist in this realm, many issues remain open in this domain. In this section we enumerate a few of these research opportunities.

Program/Resource Fragmentation

Unlike Grid computing environments (*e.g.*, Emulab) a SN user may have a short, one-time SN application that, given some external deadline, requires immediate deployment and thus cannot wait for an optimal resource allocation (*e.g.*, observing an event that is going to occur imminently, but only once). Such quick-and-dirty resource allocation algorithms may exhibit extreme program fragmentation (excessive communication due to low availability on every node) or resource affinity (most programs wind up on a few nodes). By providing resource allocation as a modular component (*e.g.*, as done in SNBENCH) or allowing optimization along multiple configurable axes (*e.g.*, as done in TinyCubus) some SNCPs can allow the composer to select an assignment scheme that best suit the needs of the particular SN instance. Yet, it remains to be seen how various embedding solutions balance the trade-off between response time and fragmentation.

Authorization and Security

In any remotely programmable system it is imperative that only authorized users task resources and that data is not intercepted by unauthorized third-parties. Similarly there is a need to ensure that the SN computation correctness is not compromised by malicious entities spoofing legitimate SN components. Traditional security approaches may prove too heavy weight for the highly-constrained SN domain. At present, most works underscore the importance of security concerns yet fail to address them in any concrete way.

Fairness and Resource Cost

With multiple users tasking SN resources, some notion of fairness must be provided to prevent any user(s) from monopolizing the system. While several approaches exist in the Distributed Computing domain (*e.g.*, fair share, virtual currency, auctions) the solution and infrastructure to provide fairness must minimize bandwidth, CPU overhead and the latency between job submission and execution.

Wide-area SNs

It is conceivable that future SN applications may span multiple independently administered SNs (*i.e.*, composing a larger wide-area SN). Such “*elastic* SNs” require potentially specialized addressing, resource allocation, and program composition approaches.

Comparing Programming Granularities

A multitude of programming languages exist today with different aims (*e.g.*, scripting, printing and reporting) and all of which leverage the API provided by the Operating System. Similarly we expect multiple, high-level SN languages to emerge targeting different types of applications that are all compiled to the SNCP’s API. As defining a “best” API is impossible given the inherent qualitative nature of such a discussion, some metrics must be defined to quantitatively compare SN APIs.

Further Work on Safety and Verification

Determining SN program faults statically (*i.e.*, static type-checking) has established value in the domain of desktop machines, yet there is *very* little work on extending either static or run-time verification techniques to SN programs. In addition to the static type-checking provided by SNBENCH, one could easily imagine extending further safety and analysis techniques into this domain (*e.g.*, deadlock detection/prevention, race condition detection on shared state).

2.3 Conclusion

While many works may be considered SN Construction Platforms, very few of them expressly address the need to make SN development accessible to a wider programmer population. Many of these works are proofs of concept which, while useful to further discourse, fails to provide the stable platform on which real SNs may be deployed and research pursued. The SN community sorely needs an extensible and cross-platform SNCP to emerge as a catalyst, not unlike UNIX in the mid to late 70's.

As with Operating Systems, SNCP offerings include run-time services, APIs and programming languages that are ultimately compared by qualitative means, yet desperately need a quantitative discussion. Unfortunately, real analysis of architecture (and performance) is on hold until a standard way to acquire and compare benchmarks emerges.

We have seen Sense and Respond systems emerge from the traditional data-acquisition SN approaches and similarly SN composition is diverging into three distinct camps; less constrained devices (*e.g.*, SunSpot), more constrained devices (*e.g.*, RFID, smartdust) and heterogeneous or hybrid SNs (*i.e.*, both types of sensors in the same SN). While it remains to be seen what will be considered "Sensor Network research" in the coming years, it is clear that SNCPs will play an increasingly important role.

Chapter 3

Functional Tasking Languages For Service

Composition

As per the thin-waist development paradigm (Figure 1.2) of `SNBENCH`, in this chapter we present the high-level and common tasking interfaces of the infrastructure, and the programming abstractions they provide. This chapter details the languages and abstractions of `SNBENCH`. These languages have been designed to well encapsulate the needs of SN programming while (1) easing the resource assignment challenges we face in Chapter 4 and (2) enabling the static verification techniques described in Chapter 5.

Programmatic access to the Sensorium is provided via high-level task-centric programming languages that are compiled, distributed, scheduled, and monitored by `SNBENCH`. The prototype high-level `SNBENCH` programming language is `SNAFU` (SN Applications as `FUnctions`). `SNAFU` is a strongly-typed functional-style programming language which serves as an accessible, interface for developers to glue together the functionalities of sensors, actuators, processors, storage devices, and networking units to create stateful, temporal, and persistent programs. In effect, `SNBENCH` presents programmers with an abstract, “single-system” view of a SN, in the sense that `SNAFU` code is written for the Sensorium as a whole, and not separately for each of its various subsystems.

A SNAFU program (or any high-level language in `SNBENCH`) must be compiled into the common tasking language of STEP (Sensorium Task Execution Plan); Conceptually, a STEP program takes the form of a directed acyclic graph (DAG), in which the nodes are the sampling and computation operations (functions) through which data flows in order to execute the program. STEP graphs have a straightforward serialized XML representation. An execution plan may consist of nodes that are either explicitly bound (*i.e.* must be deployed on a specific resource) or unbound (*i.e.* free to be placed wherever sufficient resources may be found). A STEP program instance is analogous to a program that has not been linked (*i.e.*, not connected to the resources that will execute it), an operation that is described in Chapter 4.

SNAFU is provided as a convenient and accessible interface to the STEP tasking language. Aside from syntactic sugar for commonly required STEP graph constructions and the production of complete STEP syntax, SNAFU is a direct sibling of STEP in that they are conceptually interchangeable. Indeed, SNAFU is not particularly unique as a high-level language (aside from its proximity to STEP). We hope to eventually develop other domain-specific languages that may also be compiled to STEP. In Chapter 8 we briefly discuss the Graphical STEP programming environment, which is essentially another high-level interface by which to task `SNBENCH`.

3.1 The SNAFU Programming Language

The current high-level `SNBENCH` programming interface is a high-level, behavior description language called SNAFU. SNAFU is a type safe programming language designed to specify the interdependence of sensors, computational resources, and persistent state that comprise a Sensorium application. A novice user will specify a SNAFU program written for the Sensorium (not the individual nodes), and the infrastructure will transparently handle translation, resource allocation, and dissemination of the program to the nodes of the SN.

SNAFU programs are written in a simple functional style; all SNAFU program terms are expressions, the evaluation of which produces values. The language is implicitly typed (*i.e.*, data types are not explicitly annotated in the language). Type checking is done on the target language (STEP) with hooks back to SNAFU in order to report errors that are meaningful to the programmer; by type checking STEP, we relieve new high-level language implementations from the burden of providing their own type checking. The connection between STEP and the source language is maintained such that end-users remain blissfully unaware of so much as the existence of the intermediate STEP representation during type checking.

Despite this apparent gap between the languages (from the end user’s perspective), to say that STEP and SNAFU are siblings is an understatement. Operationally, SNAFU is little more than a convince wrapper (containing syntactic sugar) that is a more accessible interface to service composition in STEP. In future iterations of SNAFU we plan for STEP and SNAFU to diverge to a greater extent, as we wish to use the SNAFU interface to restrict the programmer from the full potential (and pitfalls) of STEP ([GOKL06] shows that STEP/SNAFU is Turing complete).

Major Features of the SNAFU/STEP languages

Functional Style: Both SNAFU and STEP are functional-style languages; we say functional style, as there is both manipulated state and, as a result of manipulating sensors and actuators in a physical environment, there is are temporal interactions within the language. A programmer does not directly manipulate sensors or actuators, rather he or she specifies a program that describes data flowing through a series of functions.

Cycle-Safe Iteration: Explicit recursion (*e.g.*, by reference, fixed-point, or otherwise) is disallowed in both languages. We provide an iterative model of computation with limited state information such that the behavior of tail-form recursion can be emulated. Presumably, a modified version of SNAFU could be developed that compiles recursion by reference into a STEP-compatible, tail-recursion.

Parallel Execution: Owing to its functional style and lack of recursion SNAFU programs are easily represented as a directed acyclic graph (DAG) derived from the abstract syntax tree. In this DAG potential parallel execution and computational dependency is obvious. This graphical representation *is* STEP.

Explicit Run-Time Constraints: SNAFU and STEP both have a notion of functions that constrain the run-time behavior of the program, without having an impact on the computational value of the program. We call such functions *Flow-Types*. Example *Flow-Types* include functions to annotate a particular block of code with an explicit deadline, physical resource (or resource set) where the computation should occur, or a resolution for an image.

Type Safe: SNAFU and STEP are implicitly typed (no explicit type annotation), yet the type checking engine ensures that instances are type safe. Additionally our type system provides multi-dimensional sized type checking, which can indicate if resolution bounds or run-time constraints cannot be satisfied (Chapter 5 contains details of the type system).

Event Driven: As SNAFU/STEP are intended for a Sense and Respond programming environment, we include event driven programming constructs (*e.g.*, the trigger construct). These constructs allow the programmer to specify the predicate expression to test for that, when satisfied *triggers* the execution of some response expression.

Persistent, Streaming Constructs: The event driven constructs described above may take place over long periods of time; put differently, computations, events and data may *persist* in the system. STEP (and thus SNAFU) directly supports the specification of event driven constructs that persist in the system. Their repeated evaluation ultimately produces a *stream* of values, rather than a single value. We provide constructs to access a data stream as well as to produce it.

Temporally Aware: As our languages interact with a physical environment (*i.e.*, *interactive* functional language), the time at which an expression is evaluated may impact the result of that expression. To that end in SNAFU we support multiple “let” constructs that each have their own explicit temporal evaluation semantic (let’s are “compiled away” into lower level constructs in STEP).

Extensible: Programs written in SNAFU manipulate data through a series of functions. These functions include two forms of “user specified” functions: those that are provided by other SNAFU code, and those that are implemented at a lower level (*e.g.*, library functions) which we call *Opcodes*. Should the lower-level implementation of the Opcodes change, the higher-level SNAFU service logic would not need to change.

We drive the discussion of the constructs and abstractions of SNAFU through several simple examples.

Simple Sensor (and Actuator) Manipulation:

To understand the composition of the functional form of the SNAFU language, consider the following example SNAFU program that inspects a single video frame and returns the number of faces detected in that frame.

```
(* count the number of faces found in an image taken from camera 1 *)
      facecount(get(sensor(IMAGE,"CAMERA1"))))
```

The function `sensor()` takes two arguments and returns a sensor (or actuator). The first argument is the type of sensor and the second is the ID of the sensor to be returned. In the event the caller does not know (or does not care) which sensor sensor to use, the keyword `ANY` may be specified to indicate to the run-time that any sensor of this type is sufficient (if `sensor` is specified with only a single argument, `ANY` is assumed).

The function `get()` is a function which takes a sensor (of any type) and extracts a data sample from that sensor. In the example given above, `get(sensor(IMAGE,"CAMERA1"))`, an image is returned from the named camera, "CAMERA1".

The final function that appears in this example is `facecount()`, which, as is probably clear from context and its name, takes an image as input and returns the number of faces detected in that image. This function is an *Opcode* in our nomenclature (*i.e.*, a primitive operator to others). An *Opcode* is a library function that is defined by an “advanced” user (engineer) of the SNBENCH. The list of the available Opcodes, their descriptions and their type signatures is provided as an eXtensible Markup Language (XML) file to both the graphical STEP editor (IDE) and the SNAFU IDE alike.

Conditionals, Event monitoring and Repetition: SNAFU provides a typical if-then-else conditional construct, `cond(x,y,z)` that evaluates a predicate (`x`) and will evaluate either clause branch depending on the result of the predicate.

Unlike the conditional, the “`trigger`” construct provides repeated evaluation to provide event monitoring; the `trigger` associates a standing predicate (`P`) and clause or result (`Q`) that will be evaluated when the predicate is satisfied. Thus the predicate event *triggers* the evaluation of the clause. There are multiple `trigger` repetition semantics supported in SNAFU, which can be divided into two larger classes: *terminating* and *streaming*.

Terminating Triggers: A terminating `trigger` evaluates the predicate until some point at which it return a single value. Thus we say the evaluation of the `trigger` terminates.

The default terminating `trigger(P,Q)` will repeatedly evaluate `P` until it has evaluated to true, at which point `Q` will be evaluated and the value of `Q` is returned as the value of the trigger expression. The trigger will not execute `P` again after it has evaluated to be true (*i.e.*, it will terminate).

The `while_trigger` is similar to the previous trigger, except that it evaluates `Q` *every time* `P` evaluates to true and when `P` eventually evaluates to false it returns the last value of `Q` (or `NIL` if `P` was initially false).¹ The `while_trigger` construct is required to act as an iterator function (*i.e.*, make progress over a list) when implementing functionalities such as “map” or “reduce” that apply an operation to all entries in a list.

¹The `while_trigger` cannot be built up from the default `trigger` construct as there is internal state to maintain the prior value of `Q`.

Streaming, Persistent Triggers: Persistent triggers extend the basic triggers in that they continue to re-evaluate their predicate “indefinitely” and they return a stream of values from their clause over their persistent evaluation.

We provide two types of persistent triggers. The `level_trigger(P,Q)` will continually evaluate `P` and every time `P` evaluates to true, `Q` is re-evaluated. The `edge_trigger(P,Q)` will continually evaluate `P` and will only evaluate and return the consequent `Q` when `P` *transitions* to true (*i.e.*, on the edge of the signal `P`). The trigger’s value is initially `NIL` and is updated to the value of `Q` every time `Q` is evaluated. The SNAFU trigger example below runs “indefinitely”, repeatedly counting the number of faces found at the specified sensor.

```
(* produce stream of data: the number of faces found at camera1 *)
  level_trigger(TRUE,
                facecount(get(sensor(IMAGE,"CAMERA1"))))
)
```

In fact, persistent triggers typically live for a configuration-specific period of time (*e.g.*, one hour). To terminate a persistent trigger based on some run-time predication, the programmer may wrap the persistent trigger within a terminating trigger. Alternatively a persistent trigger may be wrapped in a *Flow-Type* function allowing the programmer to specify a particular temporal persistence policy.

Iterators and State: SNAFU provides symbolic assignment and function definition, however it forbids explicit recursion by reference. Yet, for some tasks, the body of a `trigger` expression may need to make use of its own prior evaluation (*i.e.*, utilizing prior state to manipulate a list of entities iteratively). This functionality is supported within STEP via the token `LAST_TRIGGER_EVAL`, which can be used within the body of the predicate or response clauses to refer to the previous evaluation (value) of the trigger response. `LAST_TRIGGER_EVAL` can be used in persistent triggers as well as in the terminating `while_trigger`. It does not make sense to support the `LAST_TRIGGER_EVAL` in a simple `trigger`, as in that context the response is only evaluated once, at the conclusion of the `trigger`’s evaluation.

```
(* an example of last_trigger_eval in use: when the trigger first runs, LTE
   is NIL so the stream returns I for latter iterations we return P applied
   to the previous value *)
   level_trigger(TRUE,
                 cond((LTE==NIL),I,P(LTE))
                 )
```

Access to a Trigger’s Value Stream: Naturally the results of persistent triggers may be used by other expressions as data sources for some other task. As the values of persistent triggers are transient and the temporal needs of the dependent expression may vary, we provide four different `read` functions that allow the programmer to specify a synchronization rule for the read of the trigger with respect to the trigger’s own evaluation.²

Specifying a “`non-blocking`” read to a trigger requests that the read immediately return the result of the last completed evaluation of the trigger’s target expression, (or `nil` if the response has not yet returned a new value). A “`blocking`” read waits until the response has produced a value before returning. A “`fresh`” read waits for a complete re-evaluation of the trigger’s predicate and target expressions before returning a value (*i.e.*, it clears the outstanding values and waits for the next value). Finally the use of a “`buffered`” read will buffer results to ensure transient values are not lost by the consumer. If the user desires a lock-step semantic (*i.e.*, do not evaluate the producer until the consumer has completed) then the persistent trigger is the wrong construct to be using on the producer side and the code should be reorganized.

Let Semantics and Temporal Effects: SNAFU provides the ability to bind a value to a recurring symbol to either some fixed value (constant) or commonly occurring sub-expression (macro). Functional languages typically adopt a single semantic to deal with *when* a `let`-bound sub-expression is evaluated (*e.g.*, eager, lazy). Given the implicit effects that time has on value production (*i.e.*, the image retrieved from a camera changes de-

²The value of a persistent trigger should always be read using one of these primitives. A program term containing an expression that directly accesses the value of a persistent trigger will be rejected by the SNAFU type engine. Terminating triggers, on the other hand, have an implicit blocking semantic and should not be wrapped by read primitives.

pending on *when* it is captured) SNAFU offers multiple `let` semantics to cover the various possible temporal semantics desired the programmer.

The `letconst` directive assigns a constant term or expression to a symbol, such that the value of an expression is evaluated only once (when first encountered). All further occurrences of the symbol are assigned the previously computed value and will not be evaluated again. This behavior is analogous to eager evaluation. In the following example, the allocation of the sensor `cam1` will be performed exactly once, at the first instance of `cam1` in `Z`, and all other instances of `cam1` in `Z` will refer to this same sensor.

```
(* bind cam1, once, to any image sensor *)
    letconst cam1 = sensor(IMAGE, ANY) in Z
```

Alternatively symbols may also be used as a shorthand to represent longer (sub)-expressions, each instance of which is to be independently evaluated (*i.e.*, macros). The `leteach` binding, “`leteach x = y in z`”, replaces every occurrence of `x` in `z` with an independently evaluated instance of the expression `y`. This behavior is analogous to lazy evaluation. Notice in the example below every instance of `cam2` in `Z` may refer to a different sensor.

```
(* (re)bind cam2 to a an image sensor on every occurrence *)
    leteach cam2 = sensor(IMAGE, ANY) in Z
```

Finally, SNAFU allows a symbol to be assigned within the scope of a trigger such that the symbol obtains a new value once per each evaluation of the trigger (*i.e.*, once per iteration). The `letonce` bindings, for use with trigger contexts, have the form “`letonce x = y in z`” and allows the expression `y` to be evaluated for the symbol `x` once per iteration of the containing trigger defined in `z`. Consider the use of the `letonce` binding in the following program fragment that continues from the previous example, the intent of which is to take an image sample from each camera once per iteration and then return the image sample that has the most faces in it in a given iteration.

```

(* repeatedly take two images and returns a stream of images, where each
   instantaneous value is the image in which more faces were detected *)
    letonce x = get(cam1) in
    letonce y = get(cam2) in
    level_trigger(TRUE,
        cond((facecount(x)>facecount(y)),x,y)
    )

```

Flow Types: SNAFU allows program terms to be wrapped by “flow type” functions. Flow types provide explicit constraints for program deployment and/or execution in the Sensorium, providing type information for the control flow as well as the data flow. As examples, the programmer may require a particular periodicity for a trigger term’s evaluation, or may wish to ensure that some computations are only assigned to a trusted set of resources. We provide some concrete Flow types within the current `SNBENCH` however, much as the opcode library is extensible, we imagine the palette of Flow types will grow organically as further deployments occur.

The example below includes several useful Flowtypes. The Flowtype function `resolution` specifies a restriction on the resolution of the image sensor (camera). The `bindto` Flowtype specifies that the `facecount` computation should occur on the specified SXE and finally the Flowtype `deadline` indicates that the computation must be completed within the (abstract and discretized) time allowed. All of these Flowtypes have one thing in common; they impact the decisions made by the service dispatcher when it assigns resources to execute this service. Details on how this data is extracted and used in static analysis techniques is provided in Chapter 5.

```

(* An example including several Flowtype functions *)
    bindto("SXE40.BU.EDU",
        deadline(30,
            facecount(get(
                resolution(320,1024,
                    sensor(IMAGE,ANY))))))

```

SNAFU Compilation: SNAFU has been designed to ensure that the abstract syntax tree (AST) of a SNAFU program maps to a task dependency diagram in the form of a directed acyclic graph (DAG) with a single root.³ Nodes in the DAG represent values, sensors or tasks while edges represent data communication/dependency between nodes. The graph is evaluated by lower nodes executing (using their children as input) and producing values to their parents such that values percolate up toward the root of the graph from the leaves. The SNAFU compiler transforms the AST of the SNAFU program into such a representation, which we call a “Sensorium Task Execution Plan” or STEP. The terms and expressions of SNAFU have analogous constructs (nodes) in STEP or clear encoding in the structure of the STEP graph. For example, the single-evaluation `letconst` construct is a directive to link a single subtree onto several parents, and the `if-then-else` function refers to the placement of a conditional (`cond`) STEP node.

3.2 Sensorium Task Execution Plan (STEP)

A Sensorium Task Execution Plan (STEP) is a specification of a Sensorium program in terms of its fundamental sensing, computing and communication requirements. A STEP is serialized as an XML document that encodes the directed acyclic graph (DAG) of the explicit task dependency (evaluation strategy) of a Sensorium program. The STEP language is used to describe (1) whole programs written in the scope of the entire Sensorium (*i.e.*, programs compiled from SNAFU that are either largely or entirely agnostic as to the specific resources on which their constituent operations are hosted) and (2) (sub)programs to be executed by specific individual sensor execution environments to achieve some larger programmatic task (*i.e.*, to task the individual Sensorium resources in support of (1)).

STEP is the preferred target language for the compilation of SNAFU (and other future languages) and as such we refer to STEP as the Sensorium “assembly language” (*i.e.*, STEP is our “lowest-level” Sensorium programming language). That said, STEP is a relatively

³Although the SNAFU AST is a tree, the execution semantic of SNAFU is actually a graph. Consider the `letconst` binding that allows a single node to have multiple parents.

high-level interpreted language. Although STEP programs are technically human-readable, their XML representation (including attributes which the user may have no interest or business assigning) make direct program composition in the STEP language inadvisable at best. We note that while STEP is the preferred target language for SNAFU compilation, for some constrained resources, running a STEP interpreter may not be desirable. In such situations we have two options: (1) Rely on gateway nodes to interpret STEP and relay requests on the nodes behalf (we use this approach within our current Sensorium to integrate Berkley Motes for temperature sensing), or (2) Our STEP re-compiler that produces a lighter weight ISA to run in the “native” SXE which is more suitable for highly constrained, time-sensitive devices (detailed in Chapter 6).

Individual tasks within a STEP (*i.e.*, nodes within the STEP graph) may be “bound” to a particular SN resource (*e.g.*, some sensor sampling operation that must be performed at a specific location) while others are “unbound” and thus free to be placed anywhere in the SN where requisite resources are available. In general, SNAFU compilation results in the creation of an “unbound” STEP – a STEP graph containing one or more “unbound” nodes.

Unbound STEPs are analogous to unlinked binaries insofar as they cannot be executed until required resources are resolved. Unbound STEPs are posted to a Sensorium Service Dispatcher (SSD), the entity responsible for allocating resources and dispatching STEP programs. Given the state of the available system resources and the resources required by the nodes comprising this graph, the SSD fragmenting the unbound STEP graph into several smaller bound STEP subgraphs.

In the remainder of this section we describe the “classes” of tasks (nodes) that are supported by the STEP programming language and convey with broad strokes the runtime semantic they convey. The reader should note a correlation between the node classes presented in this section and the presentation of the SNAFU semantic. Indeed, these constructs are a direct encoding of the functionalities presented in that section.

We frame this discussion within the context of an example for which we provide both the SNAFU and the STEP program below. The program returns the number of faces detected/observed from any camera (with a resolution ranging from 320 to 1024) that is mounted on `s05(.sensorium.bu.edu)`.

```
(* count the number of faces from any camera connected to s05 *)
facecount(
  get(
    resolution(320,1024,
      sensor(IMAGE,any@s05))
  )
)
```

The STEP program that results from compiling this SNAFU snippet is given below.

```
<stepgraph id="SAMPLEPROGRAM">
  <exp id="FACECOUNT" opcode="SXE.CORE.IMAGE.FACECOUNT">
    <exp id="GET" opcode="SXE.CORE.SENSOR.GET">
      <sensor id="IMAGE-ANY@s05">
        <device id="ANY" type="IMAGE" uri="s05" />
        <flowtype label="RESOLUTION">
          <param name="MIN" value="320" />
          <param name="MAX" value="1024" />
        </flowtype>
      </sensor>
    </exp>
  </exp>
</stepgraph>
```

step node: The `step` node is the root node of a STEP and contains the entire STEP program. The node has an `id` attribute that is a globally uniquely identifier (GUID) generated by the SNAFU compiler, uniquely identifying this program (and all nodes of this program) from other programs. The immediate child of the `step` node is the true root node of the program.

exp nodes: An `exp` (*expression*) node conveys a single computing, sensing, storage, or actuator function to be performed by some Sensor eXecution Environment (SXE). An expression node has an `opcode` attribute that identifies which function should be performed,

and the immediate children of the expression node are the arguments to the function. Example opcodes include addition, string concatenation, image capture, image manipulation, detecting motion, *etc.* Opcodes are core library’ operations distributed with the SXE. If an SXE does not have the opcode required, the `jar` attribute may specify a URL where a Java bytecode implementation of this opcode can be found. Similarly, the `source` attribute may be used to specify the location of the Java source code for this opcode.⁴

cond nodes: A `cond` (*conditional*) node has three children: an expression that evaluates to a boolean value (*i.e.*, a condition), an expression that will be evaluated if the condition is true, and an expression that will be evaluated if the condition is false. The *conditional* node has an evaluation semantic that ensures the first child (sub-tree) is evaluated first and, depending on the result, only the second or the third child will be evaluated.

sensor nodes: A `sensor` node conveys a specific physical device (*i.e.*, sensor) within the infrastructure, and is used to provide that device as an argument to some expression node. In the example given the `get` expression node has a `sensor` node as a child to specify on which particular image sensor it will operate. A sensor node has a `device` node as its child. A `device` node may have a `uri` attribute to indicate where the device can be found, and will have a `type` attribute to indicate the type of input that this device provides (*e.g.*, `image`, `video`, `temperature`). Sensor/device nodes only appear as leaves in a STEP graph. A sensor node requires additional processing on the SSD to resolve and reserve “wildcard” sensors (*i.e.*, when the `uri` of the sensor contains the keywords ANY or ALL).

trigger, while_trigger, edge_trigger, and level_trigger nodes: All `trigger` nodes specify that their descendants are subject to iteration as indicated by the corresponding *trigger* construct (explained in Section 3.1). Trigger nodes have two children: the predicate and the body. A trigger may also have zero or more `flowtype` nodes to convey the runtime/deployment QoS constraints of this trigger. Related to trigger functions, **read nodes** may appear as the parent of a `trigger` node to explicitly request specific temporal access to the values produced by that trigger. The `read` node’s `opcode` attribute deter-

⁴The dynamic migration of opcode implementations raises clear security/trust concerns. We expect the SXE owner will maintain a white-list of trusted hosts from which opcodes can be safely retrieved.

mines whether the trigger will be read via a “**blocking**”, “**non-blocking**”, “**fresh**”, or “**buffered**” semantic (described in Section 3.1).

flowtype nodes: The **flowtype** nodes are used to encode run-time security, performance, and persistence constraints. These nodes appear as children of the nodes that they constrain.

socket nodes: **socket** nodes may be inserted into unbound STEP DAGs by the SSD during the binding (scheduling) process to allow distribution of a program’s evaluation across SXEs. The **socket** node connects the computation graph from one SXE to another SXE across the network. A socket node has a **role** attribute which is set to either **sender** or **receiver**. A **sender** node takes the value passed up by a child node and sends it across the network to another SXE specified by the node’s **peeruri** attribute. Assuming the peer SXE is hosting a corresponding **receiver** node, that **receiver** node sends this value along to its parent node allowing a STEP “edge” to span SXEs. A **protocol** attribute specifies which specific communication protocol should be used for data transfer (*e.g.*, HTTP/1.1 pull, HTTP/1.1 push).

splice nodes: A **splice** node is used as a pointer to another node, allowing the encoding of graphs within the tree-centric XML. The **splice** node indicates that the parent of the **splice** node should have an edge that is connected to the “target” of the **splice** node (the **splice** node has a **target** attribute specifies an id of another existing node). The **splice** node only exists when a STEP is serialized as XML, when deserialized, the edge is connected to the target node. These nodes may occur within a compiled STEP graph if some node/subgraph has multiple parents (*e.g.*, the “let” binding provided in SNAFU) or a **splice** may occur as a result of computational reuse allowing one STEP program to be grafted on to another. The **splice** node is also used by the SSD to specify the reuse of previously deployed STEP graph components within a newly deployed STEP graph. If the SSD deems that a sub-graph of a newly submitted task is functionally identical to a previously deployed expression, that subgraph will be replaced with a **splice** node that points to the preexisting computation.

const nodes: The `const` node class is used to block the propagation of “clear” events during evaluation, in effect preventing the re-evaluation of its descendants (*e.g.*, to support the `letonce` and `letconst` bindings in SNAFU). A `const` node will have exactly one child, namely the subgraph that we wish to limit to a single evaluation.

In the next Chapter, we provide the reader with details of how these constructs and abstractions are executed by the runtime components of the SNBENCH infrastructure.

Chapter 4

The `SNBENCH` Architecture and Infrastructure

`SNBENCH` abstracts a collection of dissimilar and disjoint resources into a shared virtual SN (*i.e.*, Sensorium). Programmatic access to the sensory infrastructure occurs via a high-level task-centric programming languages that are compiled into the common tasking language, STEP (Sensorium Task Execution Plan). These programs are written at the scope of the SN rather than its individual components and thus specific details of the components or deployment need not be specified by the developer. To execute STEP tasks, `SNBENCH` provides a run-time support infrastructure that provides efficient (and extensible) automated program decomposition, deployment, and resource management services.

In this Chapter we detail the run-time components provided by `SNBENCH`, specifically the Service Dispatcher, Resource Manager, and Sensor Execution Environment. These components provide the back-bone of a Sensorium infrastructure, converting a heterogeneous collection of nodes into a single, STEP task-able interface. Unlike other work in this domain, these components have been designed to be as modular and and extensible as possible; as a result the functionalities, policies, hardware, and even types of manipulated data can be updated through provided interfaces with nominal changes. We frame our presentation of these components within motivating examples in the domain of networked computer vision applications.

A Sensorium Task Execution Plan (STEP), is logically a directed acyclic graph (DAG), in which the nodes are the sampling and computation operations required to execute the program. An execution plan may consist of nodes that are either bound (*i.e.*, must be deployed on a specific resource) or unbound (*i.e.*, free to be placed wherever sufficient resources may be found). An unbound STEP is analogous to a program that has not been linked. When transferred within the system, a STEP instance has a straightforward serialized representation in XML (the eXtensible Markup Language).

The Sensorium Service Dispatcher (SSD) is responsible for the linking and scheduling of a STEP onto the SN infrastructure. In general, the SSD solves a restricted form of a graph embedding problem, finding resources capable of supporting sub-graphs of the STEP graph and allocating them as appropriate. The SSD optimizes the use of resources and identifies common subexpressions across already deployed execution plans such that computation resources may be shared and/or reused. The SSD relies heavily on the Sensorium Resource Manager (SRM), a registrar of computing and sensing resources available in the system at present. The SSD decomposes a single “unbound” STEP graph into several smaller “bound” STEP graphs and dispatches those graphs onto available Sensorium functional units.

Each Sensorium functional unit features a Sensorium eXecution Environment (SXE), which is a run-time system that realizes the abstract functionalities presented to SNAFU programmers as basic building blocks. Such realizations may rely on native code (*e.g.*, device drivers, or local libraries) or may entail the retrieval of programmer-supplied code from remote repositories. An SXE interprets and executes partial STEP graphs that have been delegated to it by the SSD. Such a partial STEP graph may involve computing, sensing, storage, or communication with other SXEs.

4.1 The Sensorium Service Dispatcher (SSD)

The Sensorium Service Dispatcher (SSD) is the administrative authority of and single interface to each “local-area” Sensorium. The SSD is responsible for allocating available con-

crete Sensorium resources to process STEP (sub)programs (*i.e.*, scheduling) and dispatching STEP (sub)programs to those resources. Each SSD is tightly coupled with a Sensorium Resource Manager (SRM) that maintains the state and availability of the resources within the Sensorium.

The current SSD/SRM implementation is Java based and utilizes HTTP as its primary communication model; an HTTP server provides an interface to managed resources and end users alike. Communications are XML formatted and, for end users, responses are transformed into viewable interactive web page by XSLT. The HTTP namespace is leveraged to provide a natural interface to the hierarchal data and functionality offered by the SSD.

The SSD/SRM has two primary directives: Resource Management and STEP Scheduling and Dispatch. Both are described below.

4.1.1 Resource Management

The Sensorium Resource Manager monitors the state of the resources of its local Sensorium and reports changes to the SSD. Each computing or sensing component in the managed domain that hosts an SXE sends a heartbeat to its SRM, the result of which is used to populate a directory (hashtable) of all known SXEs and their attached sensing resources. The heartbeat includes the SXE's uptime, sensing capabilities, and a scaled score indicating available computing capacity. Should an SXE miss a configurable number of heartbeats, or the SXE report an unexpected computing capacity change without notification of a "STEP complete" the SRM assumes the SXE has failed or restarted and informs the SSD of the change. The SRM's knowledge of the state of the managed Sensorium is essential to the SSD's correct operation in deploying and maintaining STEP programs.

When an SXE leaves the Sensorium (*e.g.*, SXE shutdown, reboot, or graceful exit) there may be impact to one or more running STEP programs as multiple STEP applications may be dependent on a single STEP node or resource. When the SRM detects that an SXE has left the Sensorium, the SSD will treat all STEP tasks deployed on that SXE resource as a new STEP program submission and try to reassign it (in part or in whole) to other

available resources.¹ Updated socket nodes are sent to redirect those SXEs hosting sockets connected to the exiting SXE.² If no sufficient resources can be found to consume the STEP nodes that had been hosted by the old resource, we must traverse the dependency graph and remove all impacted STEP nodes (*i.e.*, programs).

4.1.2 Scheduling and Dispatch of STEP Programs

The SSD maintains a master, non-executable STEP graph consisting of all STEP programs currently being executed by the local Sensorium. Each STEP node in this graph is tagged with the GUID of the SXE on which that STEP node is deployed, such that this master STEP graph indicates what tasks are deployed in the local Sensorium and on to what resources.

When a STEP program is submitted to the SSD, the SSD must locate available resources for the newly submitted STEP graph, fragmenting the newly submitted STEP into several smaller STEPs that can be accommodated by available resources. We approach this task as a series of modules that process the unbound STEP program (Figure 4.1). We present our approach for each of these modules, yet emphasize the benefit of this modular approach is that any module may be replaced with a different or improved algorithm to achieve the same goal.

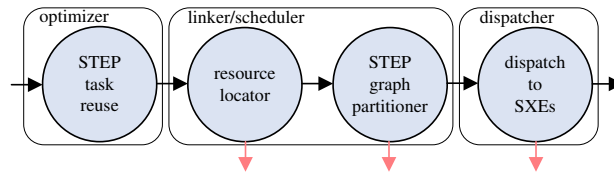


Figure 4.1: The SSD’s Scheduling and Dispatch process for new STEP programs. Circles represent modules, while downward arrows indicate these modules may reject a STEP program due to insufficient resources.

¹STEP nodes that are submitted to the SSD pre-bound to some specific resource cannot be migrated to another SXE and therefore must be terminated

²Synchronization issues abound when this occurs so a “reset” is sent to all nodes involved to restart communication in this event

Code Reuse: In the version of SNBENCH described in this thesis, we assume that the reuse of computation is paramount. When a new STEP graph is posted, the SSD first tries to ensure that the scheduling of unbound expression nodes does not result in unnecessary instances of new tasks. Unfortunately checking for all instances of “functionally equivalent” tasks in a language as expressive as STEP is NP-hard.

Indeed, as many sensing functionalities will be dependent on the use of fresh data, our current code reuse algorithm is intentionally conservative to avoid the reuse of stale data. Unless explicitly specified with flowtypes, the SSD will only reuse nodes that are “temporally compatible”.³ Thus all new nodes that match already deployed nodes are replaced with splices (*i.e.*, pointers) to the previously dispatched nodes.⁴ Regardless of how the code-reuse module is implemented, after it is complete the “new” computation cost of the submitted STEP graph should be reduced.

Admission Control: The SSD must deny admission if any bound STEP node refers an unavailable resource or if the remaining resource cost exceeds total available resources. The SSD first iterates over all bound nodes of the STEP graph to ensure that requested SXEs are known by the SRM and have available computing resources to consume these computations (including available sensors where sensors are prerequisites for a computation). Once complete, the SSD ensures that the total free resources in the Sensorium are sufficient to accommodate the total cost of the remaining unbound nodes.

Graph Partitioning: The SSD must bind all unbound nodes in the STEP graph to specific resources, a task analogous to a graph partitioning where each partition represents deployment on some SXE (*i.e.*, physical resource) with the goal of minimizing the total cost of edges between partitions (*i.e.*, minimize induced communication cost between SXEs). Fortunately, the computations represented at each node have associated datatypes and that type information yields a bound on the “cost” of each edge. For example, if a STEP

³At present, temporal compatibility is ensured by reusing only identical, trigger- rooted subexpressions in the *local* Sensorium (giving us a tractable problem).

⁴There are certainly instances in which such blind bias toward computational reuse will result in a communication penalty that outweighs the benefit of code reuse, however we have not accounted for this in the implementation of the SSD presented in this thesis.

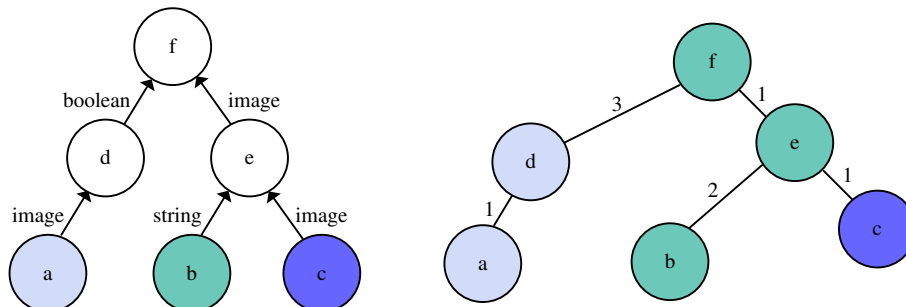


Figure 4.2: Generating colored partitions in a STEP graph. Coloring nodes is analogous to assigning a task to a particular SXE. Uncolored nodes should be colored to minimize communication between SXEs (colors) – there is no communication cost when adjacent nodes are the same color.

node returns an Image, the communication cost of spanning this edge across two different physical resources (*i.e.*, adding this edge to the cut) will be greater than cutting an edge of a node that produces an Integer value (Figure 4.2).

Our initial graph partitioning algorithm makes only a nominal attempt to reduce communication cost. The procedure tries to assign the entire unbound region of the graph to any single available resource. Failing that, the unbound region of the graph is split into smaller subgraphs and we recurse, trying to find a resource large enough to consume the “whole” parts.

Our next generation partitioning algorithm uses a relaxed form of spreading metrics [ENRS95] to produce partitions. A spreading metric defines a geometric embedding of a graph where a length is assigned to every edge in the graph such that nodes connected via inexpensive edges are mapped to be geometrically close, while nodes across expensive edges are physically “spread apart” from each other.

The optimization detailed in [ENRS95] relies on a linear program to assign lengths to edges. Instead, we will use a “quick-and-dirty” approximation of the spreading metric, in which weights and distances for edges are derived entirely from the type information of the nodes (Figure 4.2). Although this approximation will not yield partitions with the same bounds on minimizing the cut, our approach is favorable in running time and we

can compute, off-line, the minimum cut of the spreading metric to use as benchmark for comparison against our approximation algorithm. Again we point out that any graph partitioning solution may replace our existing partitioning logic, and are investigating some “off the shelf” solutions.

Dispatch: Once all STEP nodes are annotated with bindings the SSD must generate the STEP sub-graphs to dispatch to each individual resource. During this phase the SSD inserts socket nodes to maintain the data flow of the original STEP graph after the partitioning. As each SXE receives only a part of the larger computation (and sockets to SXEs with which it shares an edge) each is unaware of the larger task it helps to achieve.

To dispatch the STEP sub-graphs, the SSD performs an HTTP post of the STEP to the SXE’s web server. If all SXEs respond to the dispatch with success, the SSD’s dispatch is complete and the STEP program is live. If not, all partial STEPs of the larger STEP that had been posted to SXEs before this failed partial STEP are deleted from those SXEs and the user must resubmit.⁵

4.2 Sensorium Execution Environments (SXE)

The Sensor eXecution Environment (SXE) is a runtime execution environment that provides its clients remote access to a participating host’s processing and sensing resources. An SXE receives XML formatted Sensor Typed Execution Plans (STEPs) and the SXE schedules and executes the tasks described. Indeed an SXE is a virtual machine (Figure 4.3) providing multiple users remote access to virtualized resources including sensing, processing, storage, and actuators via the STEP program abstraction.

The SXE communicates its capabilities and instantaneous resource availability to its local Sensorium Resource Manager (SRM), allowing the Sensorium Service Dispatcher (SSD)

⁵We do not attempt to re-optimize the Sensorium’s global STEP graph (*i.e.*, all computations on the current Sensorium) when a new STEP is submitted. It is possible that a better, globally optimal assignment may exist by reassigning nodes across the global STEP graph however we expect the computational cost will far outweigh the benefit. In the implementation of SNBENCH presented in this thesis we do not move computations once they have been initially assigned unless absolutely necessary (*e.g.*, in the event of resource failures).

to best utilize the each SXE. Each SXE maintains a local STEP graph containing a composite of all the STEP graphs (and subgraphs) tasked to this node by the SSD. In this section we describe the essential functionalities of the SXE and our current implementation of these functionalities. As is the case with the SSD, the SXE is also implemented with extensibility as a chief goal. We describe the SXE in terms of its necessary actions in support of the larger Sensorium via the STEP interface: STEP Program Admission, STEP Program Interpretation, STEP Node Evaluation and STEP Program Removal.

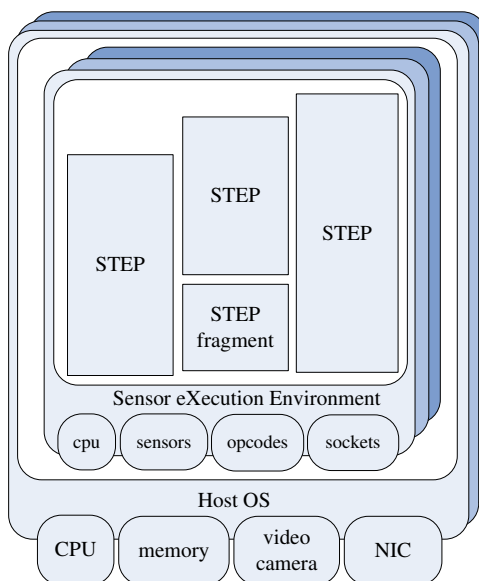


Figure 4.3: The SXE abstracts the devices provided by the OS, allowing clients to task these devices through the Sensorium Task Execution Plan (STEP) abstraction.

Sensorium Task Execution Plan (STEP) Admission: When a new STEP graph is posted to an SXE via the SSD, the new tasks to be executed may be independent of or dependent on previously deployed tasks. Within a newly posted STEP graph, the SSD may embed “splice nodes” in the new STEP graph specifying edges that are to be spliced onto previously deployed STEP nodes (*i.e.*, for task reuse) or nodes that should replace previously deployed STEP nodes with new nodes (task replacement).

1. *Task Reuse:* A newly posted STEP graph may contain one or more “splice” nodes with target ids that point to previously deployed STEP nodes, indicating some new computations will reuse the results computed by existing tasks. Although the splice is specified by the SSD through its seemingly omniscient view of the local Sensorium, each SXE maintains local scheduling control to avoid race/starvation issues.⁶

2. *Task Replacement:* If a new STEP graph includes non-splice STEP nodes with the same (unique) IDs as nodes already deployed, this indicates these new nodes should replace the existing nodes of the same ID. The replacement operation may result in either removal or preservation of children (dependencies) of the original node, while parent nodes are unaffected (although those may be modified through iterative replacement)⁷.

STEP Interpretation: Recall a STEP is a directed acyclic graph in which data propagates up, through the edges from the leaves toward the root. Tasks appearing higher in the STEP graph are not able to be executed until their children have been evaluated (*i.e.*, their arguments are available). Likewise, the need for a node to be executed is sent down from a root (parents need their children before they can execute however once executed they don’t necessarily need to be executed again).

The SXE’s local STEP graph may have several roots, as its graph may be the confluence of several independent STEP subprograms, however there is not necessarily a one-to-one mapping between the number of STEP graphs posted and the number of roots in the local STEP graph.

Each STEP node may be in one of four possible states: ready, running, evaluated, and blocked (Figure 4.4, left). The SXE’s role in interpreting a STEP program consists of (1) maintaining and updating the control flow of a STEP graph, advancing them through their state transitions (described in this section) and (2) the actual execution of STEP nodes that are in the “running” state to enable the data flow of a STEP graph.

⁶It may also be interesting to consider admission-control algorithms for determining when a new partial STEP DAG is eligible for splicing onto an existing STEP DAG. At present, the SXE takes a “the customer (SSD) is always right” policy toward admission control.

⁷Notice that such node replacements may cause synchronization difficulties when replacements involve nodes that communicate across multiple SXEs. In general, we limit our use of replacement to redirect communication nodes to a replacement SXE when another SXE has left the Sensorium.

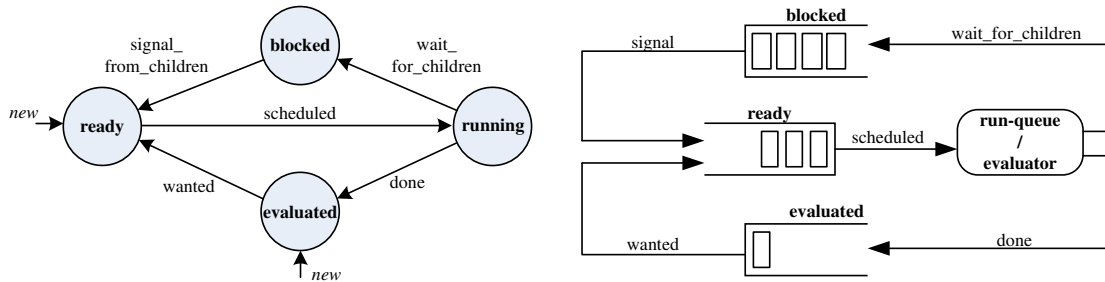


Figure 4.4: The SXE’s STEP node evaluation state transition diagram. During evaluation, STEP expression nodes move between three buffers on the SXE

The SXE interprets its current STEP nodes by continually iterating over all nodes and checking if they are “ready” to be evaluated. A generic node is determined to be ready to be evaluated if it (1) is wanted by a parent node, (2) has fresh input from all immediate children and (3) has not been already executed already (this can be reset by a parent node to enable a node to run more than once as in the case of a node with a parent trigger).

Within our present implementation, all nodes are iterated over in a round-robin fashion to determine if they are “ready”. When non-expression nodes are ready they are evaluated immediately while expression nodes are placed in a separately serviced FIFO run-queue. This approach to evaluating STEP nodes is not unique. Indeed, the selection of which nodes to consider next amounts to a scheduling decision which may be constrained by QoS requirements, or other considerations (frame rates, *etc.*). In fact any scheduling algorithm may be swapped in to service the run queue (Figure 4.4, right) without adverse effect on graph evaluation or the data flow.

A node in the ready queue is evaluated by the evaluation function that corresponds with its node class. When no STEP nodes are ready (*i.e.*, the ready queue is empty), the evaluator sleeps until either a new STEP graph is admitted or another relevant event occurs (*e.g.*, timer, socket node data arrival, SXE shutdown). Once a node has been evaluated, it produces a value that is pushed up the graph to its immediate parents. Providing values father up the graph may, in turn, enable parent nodes who were previously blocked while waiting for fresh input from their children. For some node classes, the ready function

is overridden to accommodate the node’s nonstandard execution semantic. For example, persistent triggers always return to the ready queue (*i.e.*, are always wanted), while a transient trigger only returns to the ready queue if a parent node explicitly wants it.

STEP Node Evaluation: Each STEP node class specifies its own evaluation function. The evaluation function of most node types maintains the runtime semantic of the STEP graph by updating any needed internal (including the execution state flag) and passing up values received from children. The exception to this trivial evaluation model is the evaluation of STEP expression (**exp**) nodes. In all cases, the expectation is that the evaluation function for the node will produce a value to its parents.

The evaluation of **trigger** nodes requires updating the trigger’s internal state, ensuring that first the predicate is evaluated and that the post condition will be evaluated (and re-evaluated) as per the trigger type and result of the predicate. Similarly evaluation of a conditional (**cond**) node maintains state and determines whether the second or third branch should be evaluated and returned depending on the evaluation of the first branch. A **socket** node’s evaluation sends or receives data along the socket, a **value** merely passes a serialized value up the tree and similarly **sensor** nodes are like **value** nodes in that they merely act as an argument to the immediate parent (**exp** node).

The evaluation of an expression **exp** node may take some time and as such the evaluation function for an expression node merely schedules the later execution of the expression node by a separate scheduler and execution thread (**exp** node evaluation should not block the entire resource). Expression nodes are tasks, analogous to the opcodes of the STEP programming language. These nodes are calls to fundamental operations supported by the SXEs (*e.g.*, addition, string concatenation, image manipulation), yet the opcode implementations themselves may be dynamically migrated to the SXE at runtime as needed.

Sensor Management and Datatype Extensibility: To provide access to the sensing resources of an SXE, all physical sensors are abstracted as generic sensors (data sources) with specific functionalities implemented via classes on top of the generic sensor (*e.g.*, ImageSensor, TemperatureSensor).

Opcodes do not directly manipulate sensors, but rather manipulate SNBENCH typed data. Specific details of the sensor hardware of the SXE are abstracted away by a `SensorHandler` module that is capable of communicating with and reformatting the data from a specific sensor to produce to SNBENCH typed data; support for new sensor device types require the addition of new `SensorHandler` modules⁸.

SNBENCH is extensible by design insofar as support for new sensing devices may be easily added. The Sensor eXecution Environment (SXE) requires the implementation of two relatively small interfaces to support new sensing devices; the *SensorHandler* enables the SXE to communicate with the new device type, while the *SensorDetector* module provides facility to detect new devices of this type, initialize them and inspect their state. The `SensorHandler` is akin to a device driver, abstracting away the specific idiosyncrasies of the particular device’s interface, and enabling the device to be accessed by higher-level programming constructs. As far as the SNBENCH framework is concerned, the abstracted device becomes just another managed input device/event generator only different from a video camera or motion sensor insofar as the datatype of its output.

In the SNBENCH framework there is a distinction between a SN Service Developer who uses high-level programming languages to compose Services by gluing together Opcodes and sensors (generally without regard for how the Opcodes are actually implemented beyond their type signature) and the SNBENCH “engineers” who are responsible for expanding the Opcode and `SensorHandler` libraries to enable new functionalities. SNBENCH’s modular approach allows for an advanced user (*i.e.*, engineer) to extend not only the available opcode library (by providing new Java classes that support the Opcode interface) but also to provide new data types (*i.e.*, new `snObjects`) which is analogous to providing new *modalities* to the sensing and actuation capabilities of the system. As a concrete example, if an engineer wished to add audio detection/manipulation, doing so would require the addition of a new `snObject`, new Opcodes to manipulate the data, and a new sensor to communicate with

⁸SXEs can retrieve Opcode implementations at run-time, however support for loading new sensing devices at run-time is not currently supported. Such functionality is not difficult to support, and is analogous to dynamically loading device drivers to support new hardware.

the hardware and extract a sensor reading as a value; we give specific details of providing new sensors, by example, in Chapter 7, which describes adding wireless network intrusion sensors into `SNBENCH`.

STEP Program/Node Removal: The removal of STEP nodes from the SXE may occur due to local or external events. When the evaluation of a STEP graph completes (either successfully or in error) the SXE reports the completion event with the STEP program ID to the SSD. The SXE may mark the local nodes for deletion if no other programs depend on these nodes (*i.e.*, If any ancestor node attached to these nodes has a different program ID than that of this program the SXE knows other programs are dependent on this computation). Externally requested node removal may be signaled by the SSD (for operational reasons or by request of an end user). Removal may be specified at the granularity of single nodes, however removal of a node signals removal of any parent nodes (dependent tasks) including those from different programs (assuming the SSD knows best).

In either case, the SXE does not immediately delete the nodes from its URI namespace, rather deletion is a two-phase operation, consisting of garbage marking followed by a later physical deletion by the NodeJanitor process. The garbage marking algorithm is a straightforward postfix DAG ascent, while the cleanup algorithm simply iterates over all nodes, removing those which have expired.

4.3 Implementation Overview:

Our current implementation of the SXE, SSD, and SRM each leverage Java technologies; each component is written in Java 1.6 (*a.k.a.* Java 6), utilizing the Java Media Framework (JMF) for sensor interaction and the Java based NanoHTTPD [Elo] as a lightweight HTTP server. As a mature, strongly-typed language with exception handling, the use of Java provides programming convenience as well as safety benefits. Java also provides straightforward mechanisms for dynamically loading new functionality, at run-time, over the network (via jar files or dynamically compiled source code), which is a key requirement for our SXE

component. Similarly, Java also provides isolation and run-time inspection properties via its sand-boxed runtime environment and virtual machine profiling API.

The size the jar file for the execution environment (SXE) containing all basic functionality including execution plan interpretation, evaluation, web server and client is about 200kB uncompressed. We expect the SXE to it to be deployed on low-end desktop machines (Pentium Pro) and have not attempted to port to micro-devices at present. Instead, we have approached this challenge in two ways: (1) we have implemented opcodes that act as gateways to communicate with restricted devices and (2) we have developed the “native SXE” a lightweight, stack-based virtual machine that executes a small bytecode that is generated from re-compiling STEP programs (the native SXE and its compiler are described in Chapter 6).

The SXE’s primary mode of communication is via HTTP, acting as both a server and client, at various times. The SSD communicates with constituent SXEs via their HTTP interfaces and each SXE utilizes an HTTP client to communicate with the SSD, SRM, and other SXEs. Each SXE may also utilize other communication protocols to communicate with non-standard SXEs or non-SXE Sensorium participants (motes, IP video cameras, *etc.*). Data transfer between SNBENCH components is almost exclusively XML formatted, including Base64/MIME encoding of binary data. The SXE sends an XML structured heartbeat to the SRM via an HTTP post from the SXE to the SSD. STEP graphs are uploaded to an SXE via HTTP POST of an XML object.

The SXE is distributed with a core library of basic “opcodes” implemented in the Java programming language known as `sxe.core`. For example, there is a class `/sxe/core/math/add.java` corresponding to the opcode `sxe.core.math.add` as there is for each opcode known to the SXE. We implement a custom Java ClassLoader to support the dynamic loading of new opcodes from trusted remote sources (*i.e.*, JAR files over HTTP)⁹.

Internally, all opcode methods manipulate `snObjects`, a first-class Java representation of various STEP datatypes. The `snObject` itself is a helper class that provides common

⁹We imagine that applets could be used to allow opcodes from untrusted remote sources, and new instances of VMs created to ensure complete protection from this untrusted code.

methods that allow data to be easily serialized as XML for transmission between SXEs and for viewing results via a standard web browser (using standard mime- type appropriate content). Similarly, `snObjects` implement a method to parse an object from its XML representation. Specific `snObjects` exist including `snInteger`, `snString`, `snImage`, `snBoolean`, `snCommand`, *etc.* Opcode implementations are responsible for accepting `snObjects`, and returning `snObjects` such that the result may be passed further up the STEP graph. A sample example opcode is given below for illustrative purposes.

```

/* The addition Opcode */
snObject Call(snObjectArgList argv)
                throws CastFailure , InvalidArgumentCount
{
    return (snInteger)(argv.popInt() + argv.popInt());
}

```

While the implementation of an Opcode is invoked via a Java class, within the body of the Opcode, computations are not limited to Java calls. For example, the opcode may communicate with remote hosts, execute C++ code (or other languages) via the Java Native Interface, or generate machine code on-the-fly and transmit it to another host for remote execution.

4.4 Putting it all together

We now illustrate the operation of the various `SNBENCH` components by following an example sensing application through-out its operational life cycle. We assume that a Sensorium has been deployed, with an SSD and SRM hosted by `ssd(.sensorium.bu.edu)` with several participant SXEs. In particular, the SXE deployed on host `labeast(.sensorium.bu.edu)` is on-line and with an attached image sensor. `labeast` advertises its computational and sensing resources via periodic heartbeats to `ssd`.

Suppose an end-user, Joe, would like to see the names and faces of people in that lab. Perhaps Joe does not know everyone's name yet, such that an image of the people currently in the lab, with the faces of people detected superimposed on the image would be useful

to our user. As opcodes that support these functionalities (*e.g.*, grabbing frames, finding faces) are available to the SXEs, this program is easily composed in SNAFU.¹⁰

Our user would generate a SNAFU program to accomplish this goal using the SNAFU integrated development environment (IDE). The SNAFU program (and the development environment) are each shown below:

```
letconst x =
  get(sensor(image,cam0@labeast)) in
  drawstring(identify(facedetect(x)),x)
```

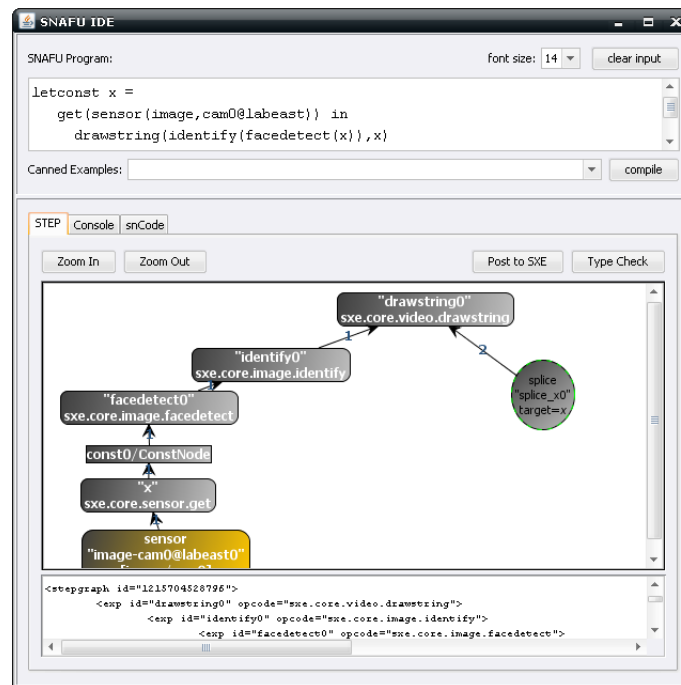


Figure 4.5: A screen-shot from the SNAFU Integrated Development Environment.

The SNAFU compiler generates an unbound STEP graph, stored as XML. A shorthand of the STEP XML graph is shown below (some attributes have been removed for clarity). Notice that the usage of the `let` binding in the SNAFU program results in the second instance of “x” in the STEP program being stored as a splice onto the same node.

¹⁰Recall that the SXE is extensible such that, if the computation that Joe requires from an SXE is not defined in the core SXE opcode library, Joe may develop his own opcode implementations.

```

<?xml version="1.0"?>
<stepgraph id="JOES-PROGRAM">
  <exp id="ROOT1" opcode="SXE.CORE.VIDEO.DRAWSTRING">
    <exp id="IDENTIFY0" opcode="SXE.CORE.IMAGE.IDENTIFY">
      <exp id="FACEDETECT0" opcode="SXE.CORE.IMAGE.FACEDETECT">
        <const id="CONST0">
          <exp id="x" opcode="SXE.CORE.SENSOR.GET">
            <sensor id="IMAGE-CAM0@LABEAST0">
              <device id="CAM0" type="IMAGE" uri="LABEAST/CAM0"/>
            </sensor>
          </exp>
        </const>
      </exp>
    </exp>
  <splice id="SPLICE_X0" target="x"/>
</exp>
</stepgraph>

```

To submit the STEP program to the infrastructure, the user can either do so directly from the SNAFU IDE or may manually launch a web browser and navigate to the SSD that administers the Sensorium deployed in the lab in this case: <http://ssd.sensorium.bu.edu:8079/snbench/ssd/>. An XSLT rendered HTML interface is presented by the SSD, one option of which allows Joe to upload the STEP program (XML file) to the SSD using a standard HTTP POST.

The SSD then parses the posted STEP graph looking for reusable STEP components (*i.e.*, nodes). Assuming no STEP programs are deployed elsewhere in the Sensorium, the SSD proceeds to try and satisfy pre-bound computations. In our example, the sensor node is “bound” to `labeast.sensorium.bu.edu` and the SSD’s scheduler requires that any opcode immediately dependent on a sensor node should be dispatched to that same resource as the sensor. In practice, this is a reasonable restriction, as it ensures that the SXE hosting the sensing device will be responsible for getting data from the sensor. This will not create a bottleneck as additional STEP programs needing data from this sensor will share the need for the same opcode and reuse will occur attaching new computations to that opcode. In

our example, the `get` opcode will be bound to `lab-east` while the other opcodes are free to be scheduled to any available SXE resource (potentially including `lab-east`).

To make things more interesting we assume `lab-east` is only able to accommodate the `get` opcode, so the STEP graph must be split across multiple SXEs. Fortunately, another SXE host on `c02` has available resources for the remaining opcodes. Notice that if we were to split across three SXEs for these computations the Sensorium would pay the communication penalty for transferring the image twice (despite the splice). In this case we only transfer the image once and the socket is reused as a splice target. This is illustrated in the shorthand STEP sub graphs given below:

```
<stepgraph id="JOES-PROGRAM:A" bindTo:"LAB-EAST.SENSORIUM.BU.EDU">
  <socket role="SENDER" id="A" peer="B">
    <exp id="x" opcode="SXE.CORE.SENSOR.GET">
      <sensor id="IMAGE-CAM0@LABEAST0">
        <device id="CAM0" type="IMAGE" uri="LABEAST/CAM0" />
      </sensor>
    </exp>
  </socket>
</stepgraph>

<stepgraph id="JOES-PROGRAM:B" bindTo:"C02.SENSORIUM.BU.EDU">
  <exp id="ROOT1" opcode="SXE.CORE.VIDEO.DRAWSTRING">
    <exp id="IDENTIFY0" opcode="SXE.CORE.IMAGE.IDENTIFY">
      <exp id="FACEDETECT0" opcode="SXE.CORE.IMAGE.FACEDETECT">
        <const id="CONST0">
          <socket id="x" role="RECEIVER" id="B" peer="A" />
        </const>
      </exp>
    </exp>
  </exp>
  <splice id="SPLICE_X0" target="x" />
</exp>
</stepgraph>
```

The SSD dispatches each STEP sub graph to the appropriate SXE (via HTTP/1.1 POST). If the POST at either SXE fails (*e.g.*, the SXE does not respond, fails to accept the STEP), the SSD deletes the graph posted at the other SXE by sending a DELETE of the

STEP graph’s program ID. If both SXEs respond with success codes (200 OK), the SSD and SRM commit their changes and are updated to maintain this new program. The SSD presents Joe with a web page containing a successful POST result and an HTTP link to the SXE node where he may (eventually) find the result of the computation:

`http://c02.sensorium.bu.edu/snbench/sxe/node/root1`. Optionally, as a security measure, the SSD may be used as a relay to prevent end users from directly connecting to SXEs. Joe may now navigate to that link or other presented links to node in the original STEP program tree and will see the current value or runtime state of each of the STEP sub computations.

As soon as the SXE has accepted the posted STEP program, its own web namespace will be updated to include the posted nodes and their current execution state and values. The SXE on `lab-east` has the lower part of the STEP graph (and no external dependency) such that it can immediately start executing its portion of the STEP graph. When the socket node is encountered, `lab-east` tries to contact `c02` and in doing so, provides `c02` with the data it needs to begin its execution. When `c02` computes a result for the orphan node “root1” it will contact the SSD informing it that the program ”joes-program” is complete.

As a single-run (non-trigger) program, the STEP evaluators on each SXE will only run the computation nodes through once and after a configurable amount of time both the nodes and the result are expunged from the SXEs.

4.5 Conclusions

SNBENCH provides a foundation for research that occurs both on-top of and within the SNBENCH platform. Users of the SNBENCH framework may develop distributed sensing applications that run on the provided infrastructure. Researchers developing new sensing or distributed computation methodologies (*e.g.*, the development of distributed vision algorithms, distributed hash tables) may take for granted the communication, scheduling, and dispatch services provided by the SNBENCH freeing them to spend their energy inves-

tigating their area of interest and expertise. These modules can be provided as opcode implementations and plugged into the architecture with ease. Instead, in this section, we focus on the research taking place within the components of the SNBENCH itself; that is, the development and research that extends the SNBENCH to improve Sensorium functionalities and meet the unique challenges of this environment.

4.5.1 Catalytic Work

We envision SNBENCH as a catalyzing agent for a number of interesting research directions both intrinsic (*i.e.*, research that aims to improve future generations of SNBENCH) as well as extrinsic (*i.e.*, research that advances the state-of-the-art in other areas). The following are examples, inspired by the projected trajectories of active research projects currently being pursued within our department.

Extrinsically, SNBENCH abstracts out the details of the SN infrastructure allowing researchers to work on the problems they are best suited to deal with. For example, vision researchers don't need to understand communication protocols, real-time schedulers, or network resource reservation to research HCI approaches for assistive environments [MSWB04]. Similarly, SNBENCH provides researchers in motion mining [TLS05] and in stream database applications [CLKB04] with a unique opportunity to implement and test proposed approaches and algorithms in a real setting. The functional, and strongly-typed nature of SNBENCH programs may inspire the development of SNBENCH (domain-specific) programming languages that are more expressive than SNAFU. In particular, SNAFU maps to STEP in a fairly straightforward way; additional more expressive front-end languages with less intuitive mappings could also be developed.

Intrinsically, the ability of the SSD to guarantee system performance could leverage advances in overlay network QoS management [GBM⁺04], distributed scheduling [Bes97], and on-line measurement, inference and characterization of networked system performance [BBH05]. Moreover, the algorithmic efficiency of the SSD will depend upon finding efficient solutions to labeled graph embedding problems [CBMP03], where those labels will have

interesting interactions with the scheduling and performance issues already raised. SXEs ought to be high-performance runtime systems, and thus can benefit significantly from operating systems virtualization [ZBG⁺05] and optimization techniques [WZSP04].

Chapter 5

Safety Verification Through Programming

Language Formalisms

SNBENCH is a platform on which novice users compose and deploy distributed Sense and Respond programs for simultaneous execution on a shared, distributed infrastructure. It is a natural imperative that we have the ability to (1) verify the safety/correctness of newly submitted tasks and (2) derive the resource requirements for these tasks such that correct allocation may occur. To achieve these goals we have established a *multi-dimensional* sized (*i.e.*, dependent) type system for our functional-style Domain Specific Language (DSL) called Sensor Task Execution Plan (STEP). In our type system, data types are annotated with a vector of size attributes (*e.g.*, upper and lower size bounds). Tracking multiple size aspects proves essential in a system in which Images are manipulated as a first class data type, as image manipulation functions may have specific minimum and/or maximum resolution restrictions on the input they can correctly process.

Through static analysis of STEP instances we not only verify basic type safety and establish upper computational resource bounds (*i.e.*, time and space), but we also derive and solve data and resource sizing constraints (*e.g.*, Image resolution, camera capabilities) from the implicit constraints embedded in program instances. In fact, the static methods presented here have benefit beyond their application to Image data, and may be extended

to other data types that require tracking multiple dimensions (*e.g.*, image “quality”, video frame-rate or aspect ratio, audio sampling rate). In this chapter we present the syntax and semantics of our functional language, our type system that builds costs and resource/data constraints, and (through both formalism and specific details of our implementation) provide concrete examples of how the constraints and sizing information are used in practice. As far as we are aware, we are the first such project to actually provide a system that employs such static verification techniques in this domain. Additionally, unlike other attempts to provide static verification to the lowest level language (in other domains), we apply our techniques to the narrow-waist tasking language, and thus are able to provide the benefits of static verification independent of the source language or target platform.

5.1 Introduction

Motivation

The Sensor Network WorkBench (SNBENCH) is a collection of compile-time tools and runtime components that enable the painless development and deployment of Sense and Response services that run on a shared infrastructure. Toward SNBENCH’s goal of enabling novice users to compose these services we provide our users with a functional-style Domain Specific Language (DSL) for specification, called STEP (Sensor Task Execution Plan).¹ STEP is resource agnostic insofar as service logic may refer to particular types of resources (*e.g.*, an Image sensor) without indicating which *specific* resources should be utilized within the service.

Our ability to allocate resources on which to deploy STEP services is contingent upon our ability to verify the safety of new services and to derive resource requirements from new service instances. In this chapter we present the static analysis techniques that we have developed to provide safety and resource constraint extraction on our sensing-centric STEP language. We base our type system on sized (*a.k.a.* static-dependent) type systems, wherein

¹We actually provide other, high-level languages that are compiled down to STEP as our common Instruction Set Architecture.

upper bound size annotations on types coupled with cost functions are used to determine memory (storage) and processing (worst case execution) bounds.

We expand our size tracking to multiple dimensions (*i.e.*, multiple dimensions of size annotations) toward the goals of (1) supporting Images as a first-class data type and (2) enabling the static inference of required image (and image sensor) resolutions from implicit constraints.

Unlike traditional scalar data, both size bounds of an Image (*i.e.*, upper and lower, where the lower bound is the potential minimum image resolution) may have an impact on functional correctness. For example, attempting to recognize a face in a low-resolution image may never succeed, or worse, might diverge depending on the implementation. While one could consider adding additional types and subtyping relations to the type system to support awareness of image resolutions (*e.g.*, `LowResolutionImage`, `MediumResolutionImage`, `HighResolutionImage`) it should be obvious that this sort of solution does not scale.

Our sized type system cannot only bound costs for memory and computation, it can also produce sensing domain specific resource constraints. In tracking both upper size bounds and lower size bounds we are able to make statements that bound a worst-case execution time and also provide bounds for Image resolution; the latter property ultimately leads to establishing the correctness of Image processing expressions.

A Motivating Example

Our goal is to be able to leverage the size annotations in the type system to provide both an upper-bound for computational requirements of services (as prior works have done), while additionally (1) maintaining minimum size aspects to verify correctness in the presence of functions that require a minimum size to ensure correctness and (2) determining and maintaining implicit constraints on resources and data sizes as extracted from contextual usage in a given service instance.

For example consider the code fragment below:

```
(* every 100 milliseconds, take an image from "any" camera and
try to detect motion. If motion is detected, send an e-mail. *)
letonce img = get(sensor(IMAGE,ANY)) in
  period(100ms,
    trigger(facedetect(img),email("MOCEAN",img))
  )
```

In the code given, the variable `img` represents an image captured from “any” image sensor. However not all image sensors (*i.e.*, cameras) have the same capabilities with respect to image resolution (*e.g.*, a webcam might capture images at a resolution several times lower than that of an embedded Pan-Tilt-Zoom camera). In this program instance there are *implicit* constraints on the image that indicate that not just *any* sensor will do. The function (or as we call it, Opcode) `facedetect` constrains the size of `img`, as it requires a minimum resolution to correctly detect a face in an image. While the explicit periodicity function (or as we call it, Flowtype) `period` indicates that this expression must run every 100 milliseconds. Thus there is another constraint on the resolution; the resolution must also be low enough to allow computation every 100 milliseconds. These constraints on the resolution of the image must propagate back to the image sensor from which the image will be acquired to ensure that the sensor reserved for this program can support the required resolution (or range of resolutions).

Our size constraint set (when solved) can be used to (1) guide task assignment (*e.g.*, do not split computation over the network where the data size will incur a steep networking overhead), (2) guide resource allocation (*e.g.*, reserve the correct sensor determined from a resolution range derived from use in context), and (3) determine if a program is fundamentally or temporarily infeasible (*e.g.*, some specific resolution too low to perform computations, required periodicity cannot be met given current available resources).

In this chapter we present our type system as applied to a subset of our domain specific language, give details of its implementation, and work through concrete examples of the system in use. The organization of the rest of this chapter is as follows: Section 5.2 provides

the syntax of our DSL and the static and dynamic semantics of our type system, and Section 5.3 applies this formalism to some examples to show the system in action as well as giving an overview of our implementation. In Section 5.4 we indicate some of the many possible future directions for this work.

Related Work

Bounding the execution time of programs and program fragments is a well-established problem in computing. Our work has been largely inspired by existing works that aim to solve this problem by providing a formal type system that has been annotated with upper size bounds on data types to estimate an upper bound on execution time and memory requirements given input size. These works are largely known as Dependent Type Systems.

We have made the conscious decision to apply static verification to our narrow-waist tasking language, unlike other attempts to provide verification to the lowest level language in other domains, and thus are able to provide these verification benefits independent of the source language or target platform. Examples of applying static verification at the lowest (*i.e.*, Assembly language) level include the Typed Assembly Language [MWCG99] and a closer comparison can be made to the Dependently Typed Assembly Language [XH01] or its current incarnation as ATS (Applied Type System) [Xi04]. Ultimately these works, insofar as they are focused on the lowest level language, are naturally used to verify properties at the lowest level that are relatively outside the scope of the Sense and Respond environments that we target (*e.g.*, array size bounds and valid memory references are considerably different than determining image sensor constraints for resource allocation).

Works that define Dependant Type Systems on higher-level functional languages include (but are not limited to) Static Dependent Costs [RG94], Sized Type Systems [HPS96], and Sized Time Systems [LH96]. Indeed, there was a large interest in applying custom type systems to domain specific languages in the late nineties (*e.g.*, the USENIX Conference on Domain-Specific Languages (DSL) in 1997 and 1999). We intended to use the Dependent Typing techniques (with small adjustments to support STEP) in order to verify basic type

safety and extract execution and memory bounds to guide resource allocation and scheduling within SNBENCH. The operating environment of SNBENCH is intrinsically distributed and time dependent, yet it seemed that these existing works should have been able to be ported more or less directly ([HFH06], for example, is the current incarnation of the Sized Time System, and is aimed directly at the real-time embedded sensing community). However, this was not the case.

Our language (and infrastructure) supports the direct modification of Image data which, unlike traditional scalar data, has an overloaded notion of size (*i.e.*, resolution) that has a direct impact on functional correctness. In this environment the type signature of a function must include explicit resolution (size) bounds to convey what size *ranges* of data a function can *correctly* process. Thus our needs began to diverge almost immediately. In the existing works size annotation has nothing to do with functional correctness, moreover we recognized a need to track a lower size bound (annotation) in addition to the upper size bound. From the need to add the additional lower size bound we have established a system in which the size annotations are *multi-dimensional*; while the formalism described in section 5.2 only includes a lower and upper size bound, Section 5.4 discusses how easily more dimensions could be added and gives examples.

Additionally our system uses size constraints to solve for data size when it is not explicitly specified by the programmer (*i.e.*, size annotation variables). Our constraint set is explicit within the typing rules yet constraints are derived implicitly from program code, using our constrained size type signature for primitive operators. In solving the constraint set we can deduce feasible (and/or optimal) data sizes which directly map to image resolutions, resource constraints and sensor capabilities for image manipulation programs. Finally we allow primitive operators that directly manipulate the constraint set, allowing the programmer to explicitly constrain types without influencing the execution (so called, Flowtypes). We are unaware of any other work that treats images as first class datatypes or that uses a type system to statically create such size constraint relationships to deduce required image sizes and sensor capabilities.

It is also worth noting that our language includes a slightly unusual let semantic which is advantageous in time dependent programming environments. While several functional languages include this same let behavior in their implementation, in time-independent operating environments implementation of the let discipline in this way does not influence the result of the computation, but rather is provided as an optimization. In our environment, this is not the case. Thus, we define our let semantic formally and include it in our proofs for soundness and completeness.

5.2 Formalism for “Core” STEP

In this section we present the formal logic that underlies the verification component described above. The type system we present in this Section supports a subset of the complete STEP programming language; we call this subset “Core STEP.” We discuss the challenges of supporting the few remaining STEP components in Section 5.4.

Readers who are actively familiar with the project may note that what is presented as STEP appears to be STEP’s high-level, functional sibling SNAFU. In fact, SNAFU is a largely a convenience wrapper for STEP, including some additional syntactic sugar (which we do not address in this chapter).

In Core STEP we formally present only one let form. As we show later, this let behaves in a way that is temporally interesting. The Complete STEP supports other, traditional let forms (*i.e.*, lazy and eager), yet we omit them from the Core formalism as they are well-known and relatively less interesting in our domain.

5.2.1 Syntax of Expressions

Below we define the syntax of the valid expression forms of Core STEP.

$e ::=$	expressions
v	<i>value</i>
x	<i>variable</i>
$\text{cond } e \ e$	<i>conditional (if-then-else)</i>
$\text{let } \{x_i=e_i\}^{i \in 1..n} \text{ in } e$	<i>let binding</i>
$\text{get } e$	<i>read a sensor</i>
$op \ e$	<i>opcode/primitive operation</i>
$\text{trigger } \{x_i=e_i\}^{i \in 1..n} \ e \ e$	<i>construct for repetition</i>
$v ::=$	values
$0 \mid 1 \mid 2 \mid \dots$	<i>integer</i>
$\text{true} \mid \text{false}$	<i>boolean</i>
i	<i>image</i>
$\text{time} \mid \text{image}$	<i>sensor</i>
$op ::= op_1 \mid op_2 \mid op_3 \mid \dots$	opcodes

5.2.2 Preliminary Definitions

The evaluation (dynamic semantics) of expressions take the general form:

$$e \mid \nu \mid t \rightarrow e' \mid \nu' \mid t + 1$$

Where ν is a special variable store required for our let semantic, and t is a discretized time that increments per evaluation step (*e.g.*, computation count, clock tick). We read this in English as “the expression e and its variable store ν take an evaluative step to the new expression e' with new variable store ν' .”

Definition: The Variable Store ν :

The evaluation of the STEP language is time dependent insofar as values that are read from the sensing environment may change over time. Thus, *when* a function is evaluated will directly effect the value retrieved (specifically *via* the sensor reading function, `get`). Put differently, the let semantic we provide has a direct impact on the values retrieved for a variable (*e.g.*, an eager let retrieves a reading at time 0, a typical lazy or by-need let retrieves a new reading at every time the substituted variable is encountered). To best accommodate the needs of STEP and this temporal dependence, we define our default let-binding to be a hybrid approach; we offer a deferred evaluation (*i.e.*, by need) that also offers the reuse provided by eager evaluation (*i.e.*, with caching).

To achieve the desired result our let-binding construct manipulates a variable store ν that stores the mapping of variables to unreduced expressions which will be reduced within the environment itself when the variable is encountered (by need). Should the same variable be encountered again further in the expression, the mapping in ν will later point to the fully reduced value and will therefore return that previously computed value (with caching).

We define the store, ν below, whose domain ranges over variable symbols, each pointing to valid STEP expression (either an unreduced expression or a value).

$$\begin{aligned}\nu &= \{x_1 \mapsto \mathbf{e}_1, x_2 \mapsto \mathbf{e}_2, \dots, x_n \mapsto \mathbf{e}_n\} \\ \text{Domain}(\nu) &= \{x_1, \dots, x_n\} \\ \nu(x_i) &= \mathbf{e}_i\end{aligned}$$

Definition: The free-variable function FV

FV calculates the free-variables in an expression (e) or expression and store pairing ($e \mid \nu$).

$$\begin{aligned}
FV(v) &= \emptyset \\
FV(x) &= \{x\} \\
FV(\text{cond } e_1 \ e_2 \ e_3) &= FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(\text{let } \{x_i = e_i\}^{i \in 1..n} \text{ in } e) &= \left(\bigcup_{i=1}^n FV(e_i) \cup FV(e) \right) - \{x_1 \dots x_n\} \\
FV(\text{get } e) &= FV(e) \\
FV(\text{op } e) &= FV(e) \\
FV(\text{trigger } \{x_i = e_i\}^{i \in 1..n} \ e_{n+1} \ e_{n+2}) &= \left(\bigcup_{i=1}^n FV(e_i) \cup FV(e_{n+1}) \cup FV(e_{n+2}) \right) - \{x_1 \dots x_n\} \\
FV(e \mid \nu) &= \left(\bigcup \{FV(\nu(x)) \mid x \in \text{Domain}(\nu)\} \cup FV(e) \right) - \text{Domain}(\nu)
\end{aligned}$$

Definition: expression and store pair closure

We say $e \mid \nu$ is **closed** if and only if $FV(e \mid \nu) = \emptyset$.

5.2.3 Dynamic Semantics

In this section we present the evaluation rules (dynamic semantics) for the various constructs (*i.e.*, syntactic forms) of the language.

Conditional

$$(E\text{-IFTRUE}) \quad \text{cond true } e_2 \ e_3 \mid \nu \mid t \rightarrow e_2 \mid \nu \mid t + 1$$

$$(E\text{-IFFALSE}) \quad \text{cond false } e_2 \ e_3 \mid \nu \mid t \rightarrow e_3 \mid \nu \mid t + 1$$

$$(E\text{-IF}) \quad \frac{e_1 \mid \nu \mid t \rightarrow e'_1 \mid \nu' \mid t + 1}{\text{cond } e_1 \ e_2 \ e_3 \mid \nu \mid t \rightarrow \text{cond } e'_1 \ e_2 \ e_3 \mid \nu' \mid t + 1}$$

Opcodes

We refer to primitive operators in STEP as Opcodes. Evaluation of opcodes is strict (arguments must be reduced to values before the opcode may be evaluated). Evaluation and typing rules for specific instances of opcodes (*e.g.*, `facetedetect`, `resample`) are given in Section 5.3. The general form is presenter here.

$$(E-OP1) \frac{e_1 \mid \nu \mid t \rightarrow e'_1 \mid \nu' \mid t + 1}{op \ e_1 \mid \nu \mid t \rightarrow op \ e'_1 \mid \nu' \mid t + 1}$$

where $op \in \{op_1 \mid op_2 \mid op_3 \mid \dots\}$

$$(E-OPAPPLY) \quad op \ v_1 \mid \nu \mid t \rightarrow v_2 \mid \nu \mid t + 1$$

where $v_2 = \mathit{Apply}(op \ v_1)$

and $op \in \{op_1 \mid op_2 \mid op_3 \mid \dots\}$

Sensor reads (and the physical sensor environment \mathcal{E})

As STEP is a sensing centric DSL, it is essential that we have the ability to read values from sensors that are embedded in the physical environment. We imagine a logical array that contains all the data that a sensor will produce at every discretized time interval. Reading a value from a sensor is analogous to extracting a value from this array indexed at the current time (t). We define an abstract matrix \mathcal{E} to correspond to the physical sensing environment such that $\mathcal{E}_{i,j}$ is the reading (value) from sensor i at discretized time j . We define the function $f_{\mathcal{E}}(i, j)$ to extract the j^{th} value (corresponding with time j) from stream (sensor) i from \mathcal{E} .

We define the (strict) function `get` to extract a value from the sensing environment. In the simplified Core STEP we support only two types of sensors, a time sensor (`time`) and Image sensors (`image`).

$$(E\text{-GET1}) \frac{e_1 \mid \nu \mid t \rightarrow e'_1 \mid \nu' \mid t + 1}{\text{get } e_1 \mid \nu \mid t \rightarrow \text{get } e'_1 \mid \nu' \mid t + 1}$$

$$(E\text{-GETAPPLY}) \text{get } v_1 \mid \nu \mid t \rightarrow v_2 \mid \nu \mid t + 1$$

$$\text{where } v_2 = f_E(v, t)$$

$$\text{and } f_E : \text{Sensor } \tau \times \text{Int} \rightarrow \tau$$

The `time` sensor returns the number of evaluations (computations) since the beginning of the evaluation (*i.e.*, returns t).

$$(E\text{-GETTIME}) (\text{get time} \mid \nu \mid t) \rightarrow (t \mid \nu \mid t + 1)$$

Let binding (by-need with caching)

Again, our let semantic is a hybrid approach that is deferred evaluation (*ala* lazy) coupled with evaluation re-use (*ala* by-value/eager). To facilitate this, when a let is encountered, its assignments are added directly to the store ν (*via* E-LETN) without evaluation. Expressions and instead are evaluated within ν when their variable is encountered elsewhere (*via* E-VAR1 and E-VAR2) including variables that point to other variables. Our let expression allows for simultaneous assignment; in E-LETN we assume that all terms x_i are assigned simultaneously and may have interdependencies.²

Variable terms are subject to alpha renaming to avoid variable capture, *etc.*

$$(E\text{-LETN}) \frac{\text{let } \{x_1 = e_1, \dots, x_n = e_n\} \text{ in } e_{n+1} \mid \nu \mid t}{\rightarrow e_{n+1} \mid \nu \cup \{x_1 \mapsto e_1, \dots, x_n \mapsto e_n\} \mid t + 1}$$

$$(E\text{-VAR1}) \frac{e_1 \mid \nu \mid t \rightarrow e'_1 \mid \nu' \mid t + 1}{(x \mid \nu \cup \{x \mapsto e_1\} \mid t) \rightarrow (x \mid \nu' \cup \{x \mapsto e'_1\} \mid t + 1)}$$

$$x \notin \text{domain}(\nu) \cup \text{domain}(\nu')$$

²Put differently, the multiple assignments in E-LETN are *not* syntactic sugar for nested lets.

$$(E\text{-VAR2}) \quad (x \mid \nu \cup \{x \mapsto v\} \mid t) \rightarrow (v \mid \nu \cup \{x \mapsto v\} \mid t + 1)$$

Triggers

STEP provides a `trigger` construct to specify repetitive conditional evaluation. It repeatedly evaluates a boolean expression until it is true and then evaluates a second expression (the first expression “triggers” the second). We also provide the ability to provide sequential let-bindings that behave as though they are within the scope of each trigger expression evaluation (*i.e.*, the let-term is recomputed on every expansion of the trigger) are available to both branches of the trigger (a single let-binding result spans both trigger arguments).

We define the trigger’s repetition recursively, via the conditional. For completeness we also define functionally degenerate trigger forms; expressions that specify a constant value for the trigger predicate produce a superfluous trigger expression (*i.e.*, `trigger true e1` always reduces to `e1` while `trigger false e1` would never proceed).

$$(E\text{-TRIGGER1}) \quad \begin{array}{l} \text{trigger } e_1 \ e_2 \mid \nu \mid t \\ \rightarrow \text{cond } e_1 \ e_2 \ (\text{trigger } e_1 \ e_2) \mid \nu \mid t + 1 \end{array}$$

$$(E\text{-TRIGGERDEG}) \quad \text{trigger } v \ e_2 \mid \nu \mid t \rightarrow \text{cond } v \ e_2 \ (\text{trigger } v \ e_2) \mid \nu \mid t + 1$$

The `letonce` expression shown is syntactic sugar for the expanded trigger expression, which builds a sequential set of variable assignments for which there is one one expansion per trigger iteration (alpha renaming ensures we are not using the same variable in every iteration). The binding is intentionally placed at the scope of both the predicate and the conclusion (*i.e.*, consequent) of the trigger term.

$$\begin{aligned} & \text{letonce } \{x_1=e_1, \dots, x_n=e_n\} \text{ in trigger } e_{n+1} \ e_{n+2} \\ & \equiv \text{trigger } \{x_1=e_1, \dots, x_n=e_n\} \ e_{n+1} \ e_{n+2} \end{aligned}$$

$$\begin{array}{l}
\text{trigger } \{x_1=e_1, \dots, x_n=e_n\} \ e_{n+1} \ e_{n+2} \\
(\text{E-TRIGGERLET}) \ \rightarrow \ \text{let } \{x_1=e_1, \dots, x_n=e_n\} \ \text{in} \\
\qquad \text{cond } e_{n+1} \ e_{n+2} \ (\text{trigger } \{x_1=e_1, \dots, x_n=e_n\} \ e_{n+1} \ e_{n+2})
\end{array}$$

5.2.4 Syntax of Types

The syntax of types in Core STEP are given below.

$t ::=$	base types
	<i>Int</i> <i>Bool</i> <i>Image</i>
$\tau_p ::=$	primitive types
	$t^{\{s,s\}}$ <i>w/ size annotation</i>
$\tau ::=$	types
	τ_p <i>primitive type</i>
	<i>Sensor</i> τ_b <i>sensor</i>
	$\tau \rightarrow \tau$ <i>opcode</i>
$s ::=$	size annotation
	n r

5.2.5 Static Semantics and Sizing of Types

The typing (static semantic) of an expression takes the general form:

$$\Gamma \vdash e : t^{\{s_{min}, s_{max}\}} \ \$c, \kappa$$

Where t is base type (*e.g.*, *Int*, *Bool*, *Image*), $\{s_{min}, s_{max}\}$ is the *size* annotation for the type (s_{min} is the lower size bound, s_{max} is the upper size bound)³, c is the worst-case approximation of computational cost of the expression and κ is a size constraint set (s_{max} and s_{min} are size annotations constrained by the simple equations stored in κ , as we will see later). In English we read this as: “Expression e has a worst-case computational cost of c and is of type t under the typing environment Γ , where t has a minimum size s_{min} , a maximum size s_{max} , and is subject to size constraints κ .”

³This pair can be expanded to a tuple to track more dimensions/aspects, as described in Section 5.4

For clarity of presentation we denote size annotations with different symbols depending on the base type they annotate. We use n_i to represent a size annotation for an Integer, and r_i for a size annotation on an Image. In our presentation we omit the implicit constraint $s_1 \leq s_2$ present for every size vector $\{s_1, s_2\}$,

Primitive Types

The typing rules for values are given below. The computational cost (c) for a value is always 0.

$$(T\text{-INT}) \quad \frac{}{\Gamma \vdash \mathbf{n} : \mathit{Int}^{\{n,n\}} \ \$ 0, \{n = \mathbf{n}\}}$$

A constant integer value has both size annotation variables constrained to the integer's actual value.

$$(T\text{-IMAGE}) \quad \frac{}{\Gamma \vdash \mathbf{i} : \mathit{Image}^{\{r,r\}} \ \$ 0, \{r = \mathit{resolutionof}(\mathbf{i})\}}$$

The size of an image is given by its resolution, taken logically to be the number of pixels in the Image, however to simplify our presentation we use only the width of the image. The pair specifies the range of possible resolutions for the image; As with integers, both values will be the same for a specific, concrete instantiation of an Image.

$$(T\text{-BOOL}) \quad \frac{}{\Gamma \vdash \mathbf{b} : \mathit{Bool} \ \$ 0, \emptyset}$$

A boolean (`true` | `false`) has a negligible fixed size and thus its size annotation is omitted (*i.e.*, $\{1,1\}$).

We define some convenience functions to manipulate the type and its size annotation :

$$\begin{aligned} \mathit{minsize}(t^{\{n_1, n_2\}}) &= n_1 & \mathit{maxsize}(t^{\{n_1, n_2\}}) &= n_2 \\ \mathit{base}(t^{\{n_1, n_2\}}) &= t \end{aligned}$$

For example: $\mathit{minsize}(\mathit{Int}^{\{4,8\}}) = 4$ $\mathit{base}(\mathit{Int}^{\{4,8\}}) = \mathit{Int}$

$$\mathit{maxsize}(\mathit{Image}^{\{2,n\}}) = n$$

Subtyping (bounding sizes)

$$\begin{array}{l}
\text{(S-REFL)} \quad \tau <: \tau \qquad \text{(S-TRANS)} \quad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \\
\text{(S-ARROW)} \quad \frac{\tau_2 <: \tau_1 \quad \tau'_1 <: \tau'_2}{\tau_1 \rightarrow \tau'_1 <: \tau_2 \rightarrow \tau'_2} \qquad \text{(S-PAIR)} \quad \frac{\tau_1 <: \tau_2 \quad \tau'_1 <: \tau'_2}{\{\tau_1 \times \tau'_1\} <: \{\tau_2 \times \tau'_2\}}
\end{array}$$

$$\text{(S-SENSOR)} \quad \frac{\tau_1 <: \tau_2}{\text{Sensor } \tau_1 <: \text{Sensor } \tau_2}$$

We define a relation similar to [LH96]’s subtyping relation (\trianglelefteq) which allows a weakening of the type to increase a size bound, in order to provide an upper bound of the size of the input relative to work to be completed. In our environment we notice that the correctness of Image processing functions may be impacted by the size of the input (*i.e.*, resolution of the image); as such we cannot arbitrarily increase the logical size (resolution) of this data without adverse consequences to functional correctness.

The need to track the lower bound extends into all aspects of the sized typing system, so we use a general sizing/weaken rule (S-SIZED) to describe the subtype relationship for specific sized types.

$$\text{(S-SIZED)} \quad \frac{(s_{min} \geq s'_{min}) \quad (s_{max} \leq s'_{max})}{t\{s_{min}, s_{max}\} <: t\{s'_{min}, s'_{max}\}}$$

Finally, we give the rule for weakening via the subtype relation, which should be clear when considered with S-SIZED, above. Notice the constraint set κ grows to include the sizing relationship between τ_1 and τ_2 . If one were to expand the sizing pair to include more dimensions/aspects of type size, then the S-SIZED and T-WEAKEN constraints would be augmented to support the new sizing logic (Section 5.4 has more on this).

$$\text{(T-WEAKEN)} \quad \frac{\Gamma \vdash e : \tau_1 \ \$ \ c, \kappa \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2 \ \$ \ c, \kappa \cup \kappa_2}$$

where $\kappa_2 = \{ \text{minsize}(\tau_2) \leq \text{minsize}(\tau_1), \text{maxsize}(\tau_2) \geq \text{maxsize}(\tau_1) \}$

Conditional

$$(T\text{-COND}) \frac{\Gamma \vdash \mathbf{e}_1 : Bool \$ c_1, \kappa_1 \quad \Gamma \vdash \mathbf{e}_2 : \tau \$ c_2, \kappa_2 \quad \Gamma \vdash \mathbf{e}_3 : \tau \$ c_3, \kappa_3}{\Gamma \vdash \text{cond } \mathbf{e}_1 \ \mathbf{e}_2 \ \mathbf{e}_3 : \tau \$ 1 + c_1 + \max(c_2 \ c_3), \kappa_1 \cup \kappa_2 \cup \kappa_3}$$

In the (common) event where terms of the conditional branches are different sizes, the application of weaken can be used to relax the bounds on either side to meet at the lower minimum size and larger maximum size.

In the event that either branch's type contains a size variable, each branch may be weakened to a new, common size variable for the conditional. The example that follows portrays exactly this, in which two images (or expressions of type image) that have different, yet unknown sizes that are supplied as the branches of a conditional only after applying T-WEAKEN to each to arrive at the new size variables n_5 and n_6 :

$$\frac{\begin{array}{c} \vdots \\ \mathbf{b} : Bool \$ c_0, \kappa_0 \end{array} \quad \frac{\mathbf{i}_1 : Image \{r_1, r_2\} \$ c_1, \kappa_1}{\mathbf{i}_1 : Image \{r_5, r_6\} \$ c_1, \kappa_1 \cup \kappa'_1} (T\text{-WEAKEN}) \quad \frac{\mathbf{i}_2 : Image \{r_3, r_4\} \$ c_2, \kappa}{\mathbf{i}_2 : Image \{r_5, r_6\} \$ c_2, \kappa \cup \kappa'_2} (T\text{-WEAKEN})}{\text{cond } \mathbf{b} \ \mathbf{i}_1 \ \mathbf{i}_2 : Image \{r_5, r_6\} \$ c_0 + \max(c_1, c_2), \kappa_0 \cup \kappa_1 \cup \kappa'_1 \cup \kappa_2 \cup \kappa'_2} (T\text{-COND})$$

where:

$$\kappa'_1 = \{r_5 \leq r_1, r_6 \geq r_2\}$$

$$\kappa'_2 = \{r_5 \leq r_3, r_6 \geq r_4\}$$

Sensors

A Sensor is a “container” type; a sensor of type *Sensor* τ will return values of type τ when “read”. The Sensor type has an negligible, omitted static size, however the inner (contained) type can be annotated with size bounds to indicate the capabilities of the sensor (*e.g.*, the range of resolutions a camera can support). The rule below indicates that **image** is an image sensor that can return images ranging in size from \mathbf{r}_{min} to \mathbf{r}_{max} (that should correlate with the maximum and minimum resolution capabilities of physical hardware). As this judgment has no premise and introduces the size variables r_1 and r_2 into the derivation tree, we will solve our constraint set for these variables once type derivation is complete.

$$(T\text{-IMAGESENSOR}) \frac{}{\Gamma \vdash \mathbf{image} : \text{Sensor Image}^{\{r_1, r_2\}} \$ 0, \{r_1 \geq \mathbf{r}_{min}\} \cup \{x_2 \leq \mathbf{r}_{max}\}}$$

The `time` sensor returns the number of evaluations (computations) since the beginning of the evaluation.

$$(T\text{-TIMESENSOR}) \frac{}{\Gamma \vdash \mathbf{time} : \text{Sensor Int}^{\{n_1, n_2\}} \$ 0, \{n_1 \geq 0\}}$$

$$(T\text{-SENSORREAD}) \frac{\text{type}(\mathbf{get}) = \text{Sensor } \tau_1 \rightarrow \tau_1 \quad \Gamma \vdash \mathbf{e}_1 : \text{Sensor } \tau_1 \$ c, \kappa_1}{\Gamma \vdash \mathbf{get } \mathbf{e}_1 : \tau_1 \$ c_1 + 1 + \text{latentcost}(\mathbf{get}, \text{maxsize}(\tau_1)), \kappa}$$

Opcodes

$$(T\text{-OP}) \frac{\text{type}(op) = \tau_1 \rightarrow \tau_2 \quad \text{constr}(op) = \kappa}{\Gamma \vdash op : \tau_1 \rightarrow \tau_2 \$ 0, \kappa}$$

$$(T\text{-OPAPPLY}) \frac{\Gamma \vdash op : \tau_1 \rightarrow \tau_2 \$ 0, \kappa_1 \quad \Gamma \vdash \mathbf{e}_2 : \tau_1 \$ c, \kappa_2}{\Gamma \vdash op \ \mathbf{e}_2 : \tau_2 \$ 1 + c + \text{latentcost}(op, \text{maxsize}(\tau_1)), \kappa_1 \cup \kappa_2}$$

where $op \in \{op_1 \mid op_2 \mid op_3 \mid \dots\}$

The $\text{latentcost}()$ function [RG94] returns the discretized computational cost of each opcode as an equation of the size of its input. For example, the complexity of finding a face in an image is a function of the total number of pixels in the image.

Let binding

An instance of a variable has whatever type is assumed for it in the typing environment Γ . A variable has no computational cost or constraints associated with it, rather the costs and constraints are assigned when variables are bound (*i.e.*, in the `let` term).

$$(T\text{-VAR}) \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \$ 0, \emptyset}$$

In a **let** term, we take the sum of the costs of the let-bound expressions as well as the union of all of their associated constraints.

$$(T\text{-LETN}) \frac{\Gamma \vdash \mathbf{e}_1 : \tau_1 \ \$ \ c_1, \kappa_1 \ \dots \ \Gamma \vdash \mathbf{e}_n : \tau_n \ \$ \ c_n, \kappa_n \quad \Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash \mathbf{e}_{n+1} : \tau_{n+1} \ \$ \ c_{n+1}, \kappa_{n+1}}{\Gamma \vdash \mathbf{let} \ \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{in} \ \mathbf{e}_{n+1} : \tau_{n+1} \ \$ \ c_{n+1} + \sum_{i=1}^n c_i, \kappa_{n+1} \cup \bigcup_{i=1}^n \kappa_i}$$

The cost defined in this rule is an over-estimate of the total cost as the let-bound symbols (some x_i) may not occur in the evaluation path of \mathbf{e}_{n+1} . These costs will not, however, be “charged” twice, as a variable itself has cost zero when computing the cost of \mathbf{e}_{n+1} .

Triggers

A trigger’s cost and constraint takes much the same form as the **cond** and **let** instances from which the trigger is derived.

(T-TRIG)

$$\frac{\Gamma \vdash \mathbf{e}_i : \tau_i \ \$ \ c_i, \kappa_i \ (\text{for } i \in 1..n) \quad \Gamma, \{x_i : \tau_i\}^{1..n} \vdash \mathbf{e}_{n+1} : \text{Bool} \ \$ \ c_{n+1}, \kappa_{n+1} \quad \Gamma, \{x_i : \tau_i\}^{1..n} \vdash \mathbf{e}_{n+2} : \tau \ \$ \ c_{n+2}, \kappa_{n+2}}{\Gamma \vdash \mathbf{trigger} \ \{x_i = \mathbf{e}_i\}^{1..n} \ \mathbf{e}_{n+1} \ \mathbf{e}_{n+2} : \tau \ \$ \ \mathcal{T} * (c_{n+1} + \sum_{i=1}^n c_i) + c_{n+2}, \kappa_{n+1} \cup \kappa_{n+2} \cup \bigcup_{i=1}^n \kappa_i}$$

where \mathcal{T} is a new cost variable from the set of unused cost variables, \mathcal{C}

Specific values for \mathcal{T} may be provided by explicit user bounds or other means.

Typing of the Let-store, ν

We can assign a type (more accurately sequence of types) to a variable store ν if we have assumed types for all of the variables contained within the store. Similarly we associate a cost and constraint set with the store (for use in interim steps of typing derivations).

$$(T\text{-NU}) \frac{\Gamma, \Gamma' \vdash \nu(x_i) : \Gamma'(x_i) \ \$ \ c_i, \kappa_i \ (\text{for every } x_i \in \{x_1, \dots, x_n\} = \text{Domain}(\nu))}{\Gamma \vdash \nu : \Gamma' \ \$ \ \sum_{i=1}^n c_i, \bigcup_{i=1}^n \kappa_i}$$

Finally we can combine an expression *and* a typed variable store into a typed pair by discharging the typing assumptions of the variable store, as shown below.

$$(T\text{-COMPLETE}) \frac{\Gamma \vdash \mathbf{e} : \tau \ \$ \ c_1, \kappa_1 \quad \emptyset \vdash \nu : \Gamma \ \$ \ c_2, \kappa_2}{\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c_1 + c_2, \kappa_1 \cup \kappa_2}$$

5.2.6 Soundness of Core STEP

This Section proves *Soundness* for the Core STEP (*Soundness = Progress+Preservation*). *Progress* means that every well-typed expression is either a value or can take an evaluative step (*i.e.*, expressions don't get stuck), while *Preservation* means that every typed expression that takes an evaluative step results in another typed expression (*i.e.*, evaluation is type preserving). We begin by proving some Lemmas that will be useful for our proofs of Progress and Preservation.

Lemma: L-Var

Suppose $\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathbf{e} \mid \nu$ is closed.

If $\mathbf{e} = x_i$, then for some c' and κ' :

$$\emptyset \vdash \nu(x_i) \mid \nu : \tau \ \$ \ c', \kappa'$$

Proof: By the structure of the derivation of $\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ where $\mathbf{e} = x_i$.

$$\frac{\frac{\Gamma(x_i) = \tau}{\Gamma \vdash x_i : \tau \ \$ \ 0, \emptyset} \text{(T-VAR)} \quad \frac{\Gamma \vdash \nu(x_j) : \Gamma(x_j) \ \$ \ c_j, \kappa_j \quad (\text{for every } x_j \in \{x_1, \dots, x_n\} = \text{Domain}(\nu))}{\emptyset \vdash \nu : \Gamma \ \$ \ c, \kappa} \text{(T-NU)}}{\emptyset \vdash x_i \mid \nu : \Gamma(x_i) \ \$ \ c, \kappa} \text{(T-COMPLETE)}$$

From the premise of T-NU: $\Gamma \vdash \nu(x_i) : \Gamma(x_i) \ \$ \ c_i, \kappa_i$

From the premise of T-VAR: $\Gamma(x_i) = \tau$

Combining these we have: $\Gamma \vdash \nu(x_i) : \tau \ \$ \ c_i, \kappa_i$

$$\frac{\Gamma \vdash \nu(x_i) : \tau \ \$ \ c_i, \kappa_i \quad \emptyset \vdash \nu : \Gamma \ \$ \ c, \kappa}{\emptyset \vdash \nu(x_i) \mid \nu : \tau \ \$ \ c', \kappa'} \text{(T-COMPLETE)}$$

Lemma(s): Inversion $+\nu$

Suppose $\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ is the last judgment \mathcal{J}_0 in a typing derivation tree \mathcal{D} . Then:

- (1) The left premise of this judgment in \mathcal{D} is $\Gamma \vdash \mathbf{e} : \tau \ \$ \ c''', \kappa'''$ (say \mathcal{J}_1) for some c''' and κ'''
- (2) If \mathcal{J}_2 is a premise of \mathcal{J}_1 of the form $\Gamma' \vdash \mathbf{e}' : \tau' \ \$ \ c', \kappa'$ for some c', κ' then $\Gamma' \vdash \mathbf{e}' \mid \nu : \tau' \ \$ \ c', \kappa'$ can be derived for all such \mathcal{J}_2 .

Proof: (Generic) By the structure of the derivation of $\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ (below). In the general form, to have reached $\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$, we must have an application of T-COMPLETE with right-side premise $\emptyset \vdash \nu : \Gamma \ \$ \ c'', \kappa''$ and on the left side of the derivation we find the individual sub-premisses to type the expression \mathbf{e} . We call this rule T-*Rule* as a placeholder for specific instances of \mathbf{e} and similarly denote the sub-premisses as \mathcal{J}_2 and the consequent (in which \mathbf{e} is given a type) as \mathcal{J}_1 . We can apply T-COMPLETE with the right hand side premise of the existing T-COMPLETE to each sub-premise of \mathcal{J}_2 individually to arrive at our conclusion. We detail the specific individual cases below.

$$\frac{\frac{\mathcal{J}_2}{\mathcal{J}_1 = \Gamma \vdash \mathbf{e} : \tau \ \$ \ c''', \kappa'''}{\text{(T-Rule)}} \quad \frac{\Gamma \vdash \nu(x_j) : \Gamma(x_j) \ \$ \ c_j, \kappa_j \text{ (for every } x_j \in \{x_1, \dots, x_n\} = \text{Domain}(\nu))}{\emptyset \vdash \nu : \Gamma \ \$ \ c'', \kappa''} \text{(T-NU)}}{\mathcal{J}_0 = \emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa} \text{(T-COMPLETE)}$$

Case L-COND:

Suppose $\Gamma \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathbf{e} \mid \nu$ is closed.

If $\mathbf{e} = \text{cond } \mathbf{e}_1 \ \mathbf{e}_2 \ \mathbf{e}_3$ then for some c'_1, c'_2, c'_3 and $\kappa'_1, \kappa'_2, \kappa'_3$:

$$\begin{aligned} \emptyset \vdash \mathbf{e}_1 \mid \nu : \text{Bool} \ \$ \ c'_1, \kappa'_1 \\ \emptyset \vdash \mathbf{e}_2 \mid \nu : \tau \ \$ \ c'_2, \kappa'_2 \\ \emptyset \vdash \mathbf{e}_3 \mid \nu : \tau \ \$ \ c'_3, \kappa'_3 \end{aligned}$$

Proof: Using the generic proof above, where:

$$\begin{aligned} \mathcal{J}_2 &= \Gamma \vdash \mathbf{e}_1 : \mathit{Bool} \ \$ \ c_1, \kappa_1 \quad \Gamma \vdash \mathbf{e}_2 : \tau \ \$ \ c_2, \kappa_2 \quad \Gamma \vdash \mathbf{e}_3 : \tau \ \$ \ c_3, \kappa_3 \\ \mathcal{J}_1 &= \Gamma \vdash \mathbf{cond} \ \mathbf{e}_1 \ \mathbf{e}_2 \ \mathbf{e}_3 : \tau \ \$ \ (1 + c_1 + (\mathit{max})(c_2, c_3)), (\kappa_1 \cup \kappa_2 \cup \kappa_3) \\ \mathit{T-Rule} &= \mathit{T-COND}. \end{aligned}$$

Case L-GET:

Suppose $\Gamma \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathbf{e} \mid \nu$ is closed.

If $\mathbf{e} = \mathbf{get} \ \mathbf{e}_1$ then for some c'_1 and κ'_1 :

$$\emptyset \vdash \mathbf{e}_1 \mid \nu : \mathit{Sensor} \ \tau_1 \ \$ \ c'_1, \kappa'_1$$

Proof: Using the generic proof above, where:

$$\begin{aligned} \mathcal{J}_2 &= \Gamma \vdash \mathbf{e}_1 : \mathit{Sensor} \ \tau_1 \ \$ \ c_1, \kappa_1 \\ \mathcal{J}_1 &= \Gamma \vdash \mathbf{get} \ \mathbf{e}_1 : \tau_1 \ \$ \ c'_1, \kappa'_1 \\ \mathit{T-Rule} &= \mathit{T-SENSORREAD}. \end{aligned}$$

Case L-OPAPPLY:

Suppose $\Gamma \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathbf{e} \mid \nu$ is closed.

If $\mathbf{e} = \mathbf{op} \ \mathbf{e}_2$ then for some c'_1, c'_2 and κ'_1, κ'_2 :

$$\begin{aligned} \emptyset \vdash \mathbf{op} \mid \nu : \tau_1 \rightarrow \tau_2 \ \$ \ c'_1, \kappa'_1 \\ \emptyset \vdash \mathbf{e}_2 \mid \nu : \tau_1 \ \$ \ c'_2, \kappa'_2 \end{aligned}$$

Proof: Using the generic proof above, where:

$$\begin{aligned} \mathcal{J}_2 &= \Gamma \vdash \mathbf{op} : \tau_1 \rightarrow \tau_2 \ \$ \ 0, \kappa_1 \quad \Gamma \vdash \mathbf{e}_2 : \tau_1 \ \$ \ c, \kappa_2 \\ \mathcal{J}_1 &= \Gamma \vdash \mathbf{op} \ \mathbf{e}_2 : \tau \ \$ \ c, \kappa_1 \cup \kappa_2 \\ \mathit{T-Rule} &= \mathit{T-OPAPPLY}. \end{aligned}$$

Case L-LETN:

Suppose $\Gamma \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathbf{e} \mid \nu$ is closed.

If $\mathbf{e} = \mathbf{let} \ \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{in} \ \mathbf{e}_{n+1}$ then for some $c'_1 \dots c'_{n+1}$ and $\kappa'_1 \dots \kappa'_{n+1}$:

$$\begin{aligned}
& \emptyset \vdash \mathbf{e}_1 \mid \nu : \tau_1 \ \$ \ c'_1, \kappa'_1 \\
& \vdots \\
& \emptyset \vdash \mathbf{e}_n \mid \nu : \tau_n \ \$ \ c'_n, \kappa'_n \\
& \emptyset, x_1 : \tau_1, \dots, x_n : \tau_n \vdash \mathbf{e}_{n+1} \mid \nu : \tau_{n+1} \ \$ \ c'_{n+1}, \kappa'_{n+1}
\end{aligned}$$

Proof: Using the generic proof above, where:

$$\begin{aligned}
\mathcal{J}_2 &= \Gamma \vdash \mathbf{e}_1 : \tau_1 \ \$ \ c_1, \kappa_1 \ \dots \ \Gamma \vdash \mathbf{e}_n : \tau_n \ \$ \ c_n, \kappa_n \ \Gamma, x_1 : \tau_1, \dots, x_n : \\
& \tau_n \vdash \mathbf{e}_{n+1} : \tau_{n+1} \ \$ \ c_{n+1}, \kappa_{n+1} \\
\mathcal{J}_1 &= \Gamma \vdash \mathbf{let} \ \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{in} \ \mathbf{e}_{n+1} : \tau \ \$ \ c_{n+1} + \sum_{i=1}^n c_i, \kappa_{n+1} \cup \\
& \bigcup_{i=1}^n \kappa_i \\
& \text{T-}Rule = \text{T-LETN}.
\end{aligned}$$

Case L-TRIGGER:

Suppose $\Gamma \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c, \kappa$ and $\mathbf{e} \mid \nu$ is closed.

If $\mathbf{e} = \mathbf{trigger} \ \{x_i = \mathbf{e}_i\}^{i \in 1..n} \mathbf{e}_{n+1} \ \mathbf{e}_{n+2}$ then for some $c'_1 \dots c'_{n+2}$ and $\kappa'_1 \dots \kappa'_{n+2}$:

$$\begin{aligned}
& \emptyset \vdash \mathbf{e}_1 \mid \nu : \tau_1 \ \$ \ c'_1, \kappa'_1 \\
& \dots \\
& \emptyset \vdash \mathbf{e}_n \mid \nu : \tau_n \ \$ \ c'_n, \kappa'_n \\
& \emptyset, \{x_i : \tau_i\}^{1..n} \vdash \mathbf{e}_{n+1} \mid \nu : Bool \ \$ \ c'_{n+1}, \kappa'_{n+1} \\
& \emptyset, \{x_i : \tau_i\}^{1..n} \vdash \mathbf{e}_{n+2} \mid \nu : \tau \ \$ \ c'_{n+2}, \kappa'_{n+2}
\end{aligned}$$

Proof: Using the generic proof above, where:

$$\begin{aligned}
\mathcal{J}_2 &= \Gamma \vdash \mathbf{e}_1 : \tau_1 \ \$ \ c_1, \kappa_1 \ \dots \ \Gamma \vdash \mathbf{e}_n : \tau_n \ \$ \ c_n, \kappa_n \ \Gamma, \{x_i : \tau_i\}^{1..n} \vdash \\
& \mathbf{e}_{n+1} : Bool \ \$ \ c_{n+1}, \kappa_{n+1} \ \Gamma, \{x_i : \tau_i\}^{1..n} \vdash \mathbf{e}_{n+2} : \tau \ \$ \ c_{n+2}, \kappa_{n+2} \\
\mathcal{J}_1 &= \Gamma \vdash \mathbf{trigger} \ \{x_i = \mathbf{e}_i\}^{i \in 1..n} \mathbf{e}_{n+1} \ \mathbf{e}_{n+2} : \tau \ \$ \ \mathcal{T} * (c_{n+1} + \sum_{i=1}^n c_i) + \\
& c_{n+2}, \kappa_{n+1} \cup \kappa_{n+2} \cup \bigcup_{i=1}^n \kappa_i \\
& \text{T-}Rule = \text{T-TRIG}.
\end{aligned}$$

Theorem [Progress]

Suppose $e \mid \nu$ is closed and $\emptyset \vdash e \mid \nu : \tau \text{ \$ } c, \kappa$ (for some cost c and constraint set κ)

Either: (1) e is a value or (2) there exists some e' and store ν' such that for every t :

$$(e \mid \nu \mid t) \rightarrow (e' \mid \nu' \mid t + 1).$$

Proof: By induction on the number of *unique* sub-derivations of a typing derivation of $\emptyset \vdash e \mid \nu : \tau \text{ \$ } c, \kappa$.

We proceed by analysis of the shape of e to show that the property holds for the larger derivation.

Case $e = v$:

Satisfied trivially as these terms are values.

Case $e = \text{cond } e_1 \ e_2 \ e_3$:

Applying the Induction Hypothesis to a derivation whose final judgment is $\Gamma \vdash e_1 \mid \nu : \text{Bool} \text{ \$ } c_1, \kappa_1$ (via L-COND) we can verify that progress holds for $e_1 \mid \nu$ (*i.e.*, e_1 is a value or $e_1 \mid \nu$ can take an evaluative step). So either (1) e_1 is a value (specifically, a Boolean) and E-IFTRUE or E-IFFALSE can be applied or $e_1 \mid \nu$ can take an evaluative step such that E-IF applies and $(e \mid \nu \mid t) \rightarrow (\text{cond } e'_1 \ e_2 \ e_3 \mid \nu' \mid t + 1)$.

Case $e = x_i$ with $x_i : \tau_i$ and $x_i \in \text{Domain}(\nu)$ (as $e \mid \nu$ is closed):

By the definition of ν (and the fact that $e \mid \nu$ is closed) we re-write $\emptyset \vdash x_i \mid \nu : \tau$ as $\emptyset \vdash x_i \mid \{x_i \mapsto e_i\} \cup \nu_+ : \tau$. By the definition of ν either $e_i = v_i$ and E-VAR applies without premise or e_i is a composite expression. To apply E-VAR1 we require progress for $e_i \mid \nu$. By the definition of ν we can re-write e_i as $\nu(x_i)$ for which progress holds via L-VAR (to which we apply the Inductive Hypothesis). Hence E-VAR or E-VAR1 may be applied and progress holds for this term.

Case $e = \text{let } \{x_1 = e_1, \dots, x_n = e_n\} \text{ in } e_{n+1}$:

E-LETN takes an evaluative step without premise such that $(e \mid \nu \mid t) \rightarrow (e_{n+1} \mid \nu \cup \{x_1 \mapsto e_1, \dots, x_n \mapsto e_n\} \mid t + 1)$.

Case $e = \text{trigger } \{x_1 = e_1, \dots, x_n = e_n\} e_{n+1} e_{n+2}$:

E-TRIGGERLET applies without premise for a single step of evaluation.

In the event that the let assignment set is empty (*i.e.*, $e = \text{trigger } e_{n+1} e_{n+2}$) then E-TRIGGER1 applies without premise.

Case $e = \text{get } e_1$:

Either e_1 is a value and E-GETAPPLY applies (or E-GETTIME if $e_1 = \text{time}$), or by L-GET and the Inductive Hypothesis, we show progress holds on $(e_1 \mid \nu)$ so $e_1 \mid \nu \mid t \rightarrow e'_1 \mid \nu' \mid t + 1$ and E-GET1 applies such that $(e \mid \nu \mid t) \rightarrow (\text{get } e'_1 \mid \nu' \mid t + 1)$

Case $e = \text{op } e_2$:

Either e_2 is a value and E-OPAPPLY applies, or by L-OPAPPLY and the Inductive Hypothesis, progress holds on $e_2 \mid \nu$ so $e_2 \mid \nu \mid t \rightarrow e'_2 \mid \nu' \mid t + 1$ and E-OP1 applies such that $(e \mid \nu \mid t) \rightarrow (\text{op } e'_2 \mid \nu' \mid t + 1)$

Theorem [Preservation]

Suppose $e \mid \nu$ is closed and $\emptyset \vdash e \mid \nu : \tau \ \$ \ c, \kappa$.

If $(e \mid \nu \mid t) \rightarrow (e' \mid \nu' \mid t + 1)$, then for some cost c' and constraint set κ' :

$$\emptyset \vdash e' \mid \nu' : \tau \ \$ \ c', \kappa'$$

Proof: By induction on the number of *distinct* sub-derivations of a typing derivation of $\emptyset \vdash e \mid \nu : \tau \ \$ \ c, \kappa$.

We proceed by analysis of the shape of e to show that the property holds for the larger derivation.

Case $e = v$:

These cases are satisfied trivially, as these terms are values and do not make a further evaluative step.

Case $e = \text{cond } e_1 \ e_2 \ e_3$:

We consider each possible evaluative case, individually.

Case E-IFTRUE:

$$(\text{cond true } e_2 \ e_3 \mid \nu \mid t) \rightarrow (e_2 \mid \nu \mid t + 1)$$

We consider the typing derivation of $e \mid \nu$ in which we have $\Gamma \vdash e_2 : \tau \ \$ \ c_2, \kappa_2$ (from the premise T-COND) and $\nu : \Gamma \ \$ \ c'', \kappa''$ (from the premise T-COMPLETE). We combine them in a new application of T-COMPLETE to verify $\emptyset \vdash e_2 \mid \nu : \tau \ \$ \ c', \kappa'$.

Case E-IFFALSE:

$$(\text{cond false } e_2 \ e_3 \mid \nu \mid t) \rightarrow (e_3 \mid \nu \mid t + 1)$$

We consider the typing derivation of $e \mid \nu$ in which we have $\Gamma \vdash e_3 : \tau \ \$ \ c_3, \kappa_3$ (from the premise T-COND) and $\nu : \Gamma \ \$ \ c'', \kappa''$ (from the premise T-COMPLETE). We combine them in a new application of T-COMPLETE to verify $\emptyset \vdash e_3 \mid \nu : \tau \ \$ \ c', \kappa'$.

Case E-IF:

$$(\text{cond } e_1 \ e_2 \ e_3 \mid \nu \mid t) \rightarrow (\text{cond } e'_1 \ e_2 \ e_3 \mid \nu' \mid t + 1)$$

By the premise of T-COND $e_1 : \text{Bool}$ and by L-COND $e_1 \mid \nu : \text{Bool}$.

By applying the Inductive Hypothesis to a judgment with this as its last term, we obtain $e'_1 \mid \nu' : \text{Bool}$.

Looking at the typing derivation of $e'_1 \mid \nu' : \text{Bool}$:

$$\frac{\frac{g = (\Gamma \vdash \mathbf{e}'_1 : Bool \ \$ \ c_1, \kappa_1) \quad \frac{\Gamma \vdash \nu(x_j) : \Gamma(x_j) \ \$ \ c_j, \kappa_j \text{ (for every } x_j \in \{x_1, \dots, x_n\} = Domain(\nu))}{h = (\emptyset \vdash \nu' : \Gamma \ \$ \ c'', \kappa'')}_{(T-COMPLETE)}}{(\emptyset \vdash \mathbf{e}'_1 \mid \nu' : Bool \ \$ \ c'_1, \kappa'_1)}_{(T-NU)}$$

Now we apply T-COND to g and the terms from the derivation of $\text{cond } \mathbf{e}_1 \ \mathbf{e}_2 \ \mathbf{e}_3 \mid \nu$ (if needed, see α in L-COND for clarity):

$$\frac{\Gamma \vdash \mathbf{e}'_1 : Bool \ \$ \ c_1, \kappa_1 \quad \Gamma \vdash \mathbf{e}_2 : \tau \ \$ \ c_2, \kappa_2 \quad \Gamma \vdash \mathbf{e}_3 : \tau \ \$ \ c_3, \kappa_3}{i = (\Gamma \vdash \text{cond } \mathbf{e}'_1 \ \mathbf{e}_2 \ \mathbf{e}_3 : \tau \ \$ \ c'_4, \kappa'_4)}_{(T-COND)}$$

Now we apply T-COMPLETE to h and i to verify: $\emptyset \vdash \text{cond } \mathbf{e}'_1 \ \mathbf{e}_2 \ \mathbf{e}_3 \mid \nu' : \tau \ \$ \ c', \kappa'$

Case $\mathbf{e} = x_i$ with $x_i : \tau_i$ and $x_i \in Domain(\nu)$ (as $\mathbf{e} \mid \nu$ is closed):

By the definition of ν (and closure of the pair $\mathbf{e} \mid \nu$) we rewrite $\vdash x_i \mid \nu : \tau \ \$ \ c', \kappa'$ as $\vdash x_i \mid \{x_i \mapsto \nu(x_i)\} \cup \nu_+ : \tau \ \$ \ c', \kappa'$. We now consider the possible evaluation cases individually:

Case E-VAR1 ($\nu(x_i) = \mathbf{v}_i$):

$$(x_i \mid \{x_i \mapsto \mathbf{v}_i\} \cup \nu_+ \mid t) \rightarrow (\mathbf{v}_i \mid \{x_i \mapsto \mathbf{v}_i\} \cup \nu_+ \mid t + 1)$$

We know from the derivation of $\mathbf{e} \mid \nu$, the premise of T-NU gives $\Gamma \vdash \nu(x_i) : \tau_i \ \$ \ c_i, \kappa_i$. As $\nu(x_i) = \mathbf{v}_i$ and $\Gamma(x_i) = \tau_i = \tau$ we can re-write this as $g = (\Gamma \vdash \mathbf{v}_i : \tau \ \$ \ c_i, \kappa_i)$.

Again from the derivation we also have the premise of T-COMPLETE: $h = (\emptyset \vdash \nu : \Gamma \ \$ \ c'', \kappa'')$. Combining g and h under T-COMPLETE we can verify $\vdash \mathbf{v}_i \mid \nu : \tau \ \$ \ c', \kappa' = \vdash \mathbf{e}' \mid \nu : \tau \ \$ \ c', \kappa'$.

Case E-VAR2 ($\nu(x_i) = \mathbf{e}_i$):

$$(x_i \mid \{x_i \mapsto \mathbf{e}_i\} \cup \nu_+ \mid t) \rightarrow (x_i \mid \{x_i \mapsto \mathbf{e}'_i\} \cup \nu'_+ \mid t + 1)$$

L-VAR gives us $(\emptyset \vdash \nu(x_i) \mid \nu : \tau \ \$ \ c', \kappa')$, which we rewrite by

the definition of ν as $(\emptyset \vdash \mathbf{e}_i \mid \nu : \tau \ \$ \ c', \kappa')$. Applying the Inductive Hypothesis we have that $(\emptyset \vdash \mathbf{e}'_i \mid \nu' : \tau \ \$ \ c''', \kappa''')$. From the typing derivation of this expression we have a right side premise for T-COMPLETE: $h = (\emptyset \vdash \nu' : \Gamma \ \$ \ c'', \kappa'')$. Combining h with our prior left hand premise of T-COMPLETE, the T-VAR's $\Gamma \vdash x_i : \tau \ \$ \ 0, \emptyset$, we have:

$$\vdash x_i \mid \nu' : \tau \ \$ \ c', \kappa' = \vdash \mathbf{e}'_i \mid \nu' : \tau \ \$ \ c', \kappa'.$$

Case $\mathbf{e} = \mathbf{let} \ \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{in} \ \mathbf{e}_{n+1}$:

Case E-LETN:

$$(\mathbf{let} \ \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{in} \ \mathbf{e}_{n+1} \mid \nu \mid t) \rightarrow (\mathbf{e}_{n+1} \mid \nu \cup \{x_1 \mapsto \mathbf{e}_1, \dots, x_n \mapsto \mathbf{e}_n\} \mid t + 1)$$

Thus our goal is to type: $\mathbf{e}_{n+1} \mid \nu \cup \{x_1 \mapsto \mathbf{e}_1, \dots, x_n \mapsto \mathbf{e}_n\}$

We first refer to the typing derivation of $\mathbf{e} \mid \nu$ (and define aliases to save space):

$$g = (\Gamma \vdash \mathbf{e}_1 : \tau_1 \ \$ \ c_1, \kappa_1 \ \dots \ \Gamma \vdash \mathbf{e}_n : \tau_n \ \$ \ c_n, \kappa_n)$$

$$h = (\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash \mathbf{e}_{n+1} : \tau_{n+1} \ \$ \ c_{n+1}, \kappa_{n+1})$$

$$\begin{array}{c} \mathcal{G} = \frac{g \quad h}{\Gamma \vdash \mathbf{let}\{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{in} \ \mathbf{e}_{n+1} : \tau_{n+1} \ \$ \ c_{n+1} + \sum_{i=1}^n c_i, \kappa_{n+1} \cup \bigcup_{i=1}^n \kappa_i} \text{(T-LETN)} \\ \\ \frac{\mathcal{G} \quad \frac{i = (\Gamma \vdash \nu(x_j) : \Gamma(x_j) \ \$ \ c_j, \kappa_j \ \text{for every } x_j \in \{x_1, \dots, x_n\} = \mathbf{Domain}(\nu))}{\emptyset \vdash \nu : \Gamma \ \$ \ c'', \kappa''} \text{(T-NU)}}{\emptyset \vdash \mathbf{e} \mid \nu : \tau \ \$ \ c_1, \kappa_1} \text{(T-COMPLETE)} \end{array}$$

We call $\nu_1 = \{x_1 \mapsto \mathbf{e}_1, \dots, x_n \mapsto \mathbf{e}_n\}$ and call to g and h show that each $\nu_1(x_i) : \Gamma(x_i)$ for $x_i \in \mathbf{Domain}(\nu_1)$. We see $\nu' = \nu \cup \nu_1$ and now we have what we need to apply T-COMPLETE and show our desired result.

$$\frac{\frac{\Gamma \vdash \nu'(x_j) : \Gamma(x_j) \$ c_j, \kappa_j \quad \text{for every } x_j \in \{x_1, \dots, x_n\} = \text{Domain}(\nu')}{\text{h}}}{\frac{\quad}{\emptyset \vdash \nu' : \Gamma \$ c'', \kappa''} \text{(T-COMPLETE)}}{\emptyset \vdash \mathbf{e}_{n+1} \mid \nu' : \tau \$ c', \kappa'} \text{(T-NU)}$$

Case $\mathbf{e} = \text{trigger } \mathbf{e}_1 \ \mathbf{e}_2$:

Case E-TRIGGER1:

$$(\text{trigger } \mathbf{e}_1 \ \mathbf{e}_2 \mid \nu \mid t) \rightarrow (\text{cond } \mathbf{e}_1 \ \mathbf{e}_2 (\text{trigger } \mathbf{e}_1 \ \mathbf{e}_2) \mid \nu \mid t + 1)$$

From the typing derivation of $\mathbf{e} \mid \nu$ we have $\mathbf{e}_1 : \text{Bool}$, $\mathbf{e}_2 : \tau$, $\mathbf{e} : \tau$, and $\emptyset \vdash \nu : \Gamma \$ c'', \kappa''$. We combine the first three under T-COND and the result with the last under T-COMPLETE to obtain our desired result.

Case E-TRIGGERDEG:

$$(\text{trigger } \mathbf{v}_1 \ \mathbf{e}_2 \mid \nu \mid t) \rightarrow (\text{cond } \mathbf{v}_1 \ \mathbf{e}_2 (\text{trigger } \mathbf{e}_1 \ \mathbf{e}_2) \mid \nu \mid t + 1)$$

Identical to the above.

Case $\mathbf{e} = \text{trigger } \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{e}_{n+1} \ \mathbf{e}_{n+2}$:

Case E-TRIGGERLET:

$$\begin{aligned} & (\text{trigger } \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \mathbf{e}_{n+1} \ \mathbf{e}_{n+2} \mid \nu \mid t) \\ & \rightarrow (\text{let } \{x_1 = \mathbf{e}_1, \dots, x_n = \mathbf{e}_n\} \ \text{in } (\text{cond } \mathbf{e}_{n+1} \ \mathbf{e}_{n+2} \ \mathbf{e}) \mid \nu \mid t + 1) \end{aligned}$$

From the typing derivation of $\mathbf{e} \mid \nu$ we have $\mathbf{e}_{n+1} : \text{Bool}$, $\mathbf{e}_{n+2} : \tau$, and $\mathbf{e} : \tau$. We combine these under T-COND, the result with the variable premises under T-LETN and finally this with $\emptyset \vdash \nu : \Gamma \$ c'', \kappa''$ under T-COMPLETE to obtain our desired result.

Case $e = op\ e_1$:

Case E-OPAPPLY:

$(op\ v_1 \mid \nu \mid t) \rightarrow (v_2 \mid \nu \mid t + 1)$ The function *Apply* by definition, returns a value (which is inherently typed). By the typing derivation of $op\ v_1 \mid \nu$ we find $\emptyset \vdash \nu : \Gamma \$ c'', \kappa''$ and can apply T-COMPLETE to obtain our desired result.

Case E-OP1:

$(op\ e_1 \mid \nu \mid t) \rightarrow (op\ e'_1 \mid \nu' \mid t + 1)$

By the premise of T-OPAPPLY $e_1 : \tau_1$ and by L-OPAPPLY $e_1 \mid \nu : \tau_1$.

By the Inductive Hypothesis we have $e'_1 \mid \nu' : \tau_1$.

Looking at the typing derivation of $e'_1 \mid \nu' : \tau_1$:

$$\frac{g = (\Gamma \vdash e'_1 : \tau_1 \$ c_1, \kappa_1) \quad h = (\emptyset \vdash \nu' : \Gamma \$ c'', \kappa'')}{\emptyset \vdash e'_1 \mid \nu' : \tau_1 \$ c', \kappa'} \text{(T-COMPLETE)}$$

Now we apply T-OPAPPLY to g and the terms from the derivation of $op\ e_1 \mid \nu$ (if needed, see α in L-OPAPPLY for clarity):

$$\frac{\Gamma \vdash op : \tau_1 \rightarrow \tau_2 \$ c_1, \kappa_1 \quad \Gamma \vdash e'_1 : \tau_1 \$ c_2, \kappa_2}{i = (\Gamma \vdash op\ e'_1 : \tau_2 \$ c', \kappa')} \text{(T-OPAPPLY)}$$

Now we apply T-COMPLETE to h and i to verify: $\emptyset \vdash op\ e'_1 \mid \nu' : \tau \$ c''', \kappa'''$

Case $e = get\ e_1$:

Case E-GETAPPLY:

$(get\ v_1 \mid \nu \mid t) \rightarrow (v_2 \mid \nu \mid t + 1)$

The function *get* by definition, returns a value (which is inherently typed). By the typing derivation of $get\ v_1 \mid \nu$ we find $\emptyset \vdash \nu : \Gamma \$ c'', \kappa''$ and can apply T-COMPLETE to obtain our desired result.

Case E-GET1:

$$(\text{get } \mathbf{e}_1 \mid \nu \mid t) \rightarrow (\text{get } \mathbf{e}'_1 \mid \nu' \mid t + 1)$$

By the premise of T-SENSORREAD $\mathbf{e}_1 : \text{Sensor } \tau_1$ and by L-GET $\mathbf{e}_1 \mid \nu : \text{Sensor } \tau_1$. By the Inductive Hypothesis we have $\mathbf{e}'_1 \mid \nu' : \text{Sensor } \tau_1$.

Looking at the typing derivation of $\mathbf{e}'_1 \mid \nu' : \text{Sensor } \tau_1$:

$$\frac{g = (\Gamma \vdash \mathbf{e}'_1 : \text{Sensor } \tau_1 \$ c_1, \kappa_1) \quad h = (\emptyset \vdash \nu' : \Gamma \$ c'', \kappa'')}{\emptyset \vdash \mathbf{e}'_1 \mid \nu' : \text{Sensor } \tau_1 \$ c', \kappa'} \text{(T-COMPLETE)}$$

Now we apply T-SENSORREAD to g and the terms from the derivation of $\text{get } \mathbf{e}_1 \mid \nu$ (if needed, see α in L-GET for clarity):

$$\frac{\Gamma \vdash \text{get} : \text{Sensor } \tau_1 \rightarrow \tau_1 \$ c_1, \kappa_1 \quad \Gamma \vdash \mathbf{e}'_1 : \text{Sensor } \tau_1 \$ c_2, \kappa_2}{i = (\Gamma \vdash \text{get } \mathbf{e}'_1 : \tau_1 \$ c', \kappa')} \text{(T-SENSORREAD)}$$

Now we apply T-COMPLETE to h and i to verify: $\emptyset \vdash \text{get } \mathbf{e}'_1 \mid \nu' : \tau_1 \$ c''', \kappa'''$

5.3 Applications of the formalism

In this section we present concrete instantiations of the formalism presented in the previous section, in order to illustrate its benefit.

5.3.1 Additional Syntax

In addition to defining specific primitive operators (Opcodes) we also define the notion of a pair to allow these operators to accept multiple inputs.

e ::=	expressions
{e,e}	<i>pair</i>
ft	<i>flowtypes</i>
v ::=	values
{v,v}	<i>pair</i>
op ::=	opcodes
fst	<i>first projection of a pair</i>
snd	<i>second projection of a pair</i>
sizedimage	<i>allocate an image sensor (supporting specific sizes)</i>
add	<i>add integers</i>
facect	<i>count faces in an image</i>
resample	<i>change the resolution of an image</i>
ft ::=	flowtypes
deadline n e	<i>define a timing constraint</i>
τ ::=	types
$\tau \times \tau$	<i>pair</i>

5.3.2 Typing and Evaluation Rules for Pairs

Evaluation (strict) and typing rules for pairs are handled in a standard way.

$$(T\text{-PAIRCONS}) \frac{\Gamma \vdash e_1 : \tau_1 \ \$ \ c_1, \kappa_1 \quad \Gamma \vdash e_2 : \tau_2 \ \$ \ c_2, \kappa_2}{\Gamma \vdash \{e_1, e_2\} : \tau_1 \times \tau_2 \ \$ \ 1 + c_1 + c_2, \kappa_1 \cup \kappa_2}$$

$$(T\text{-PAIRFIRST}) \frac{\Gamma \vdash \mathbf{fst} : \tau_1 \times \tau_2 \rightarrow \tau_1 \ \$ 0, \emptyset \quad \Gamma \vdash \mathbf{e} : \tau_1 \times \tau_2 \ \$ c, \kappa}{\Gamma \vdash \mathbf{fst} \ \mathbf{e} : \tau_1 \ \$ 1 + c, \kappa}$$

$$(T\text{-PAIRSECOND}) \frac{\Gamma \vdash \mathbf{snd} : \tau_1 \times \tau_2 \rightarrow \tau_2 \ \$ 0, \emptyset \quad \Gamma \vdash \mathbf{e} : \tau_1 \times \tau_2 \ \$ c, \kappa}{\Gamma \vdash \mathbf{snd} \ \mathbf{e} : \tau_2 \ \$ 1 + c, \kappa}$$

$$(E\text{-PAIRFST}) \ \mathbf{fst} \ \{v_1, v_2\} \mid \nu \mid t \rightarrow v_1 \mid \nu \mid t$$

$$(E\text{-PAIRSND}) \ \mathbf{snd} \ \{v_1, v_2\} \mid \nu \mid t \rightarrow v_2 \mid \nu \mid t$$

$$(E\text{-PAIR1}) \frac{e_1 \mid \nu \mid t \rightarrow e'_1 \mid \nu' \mid t + 1}{\{e_1, e_2\} \mid \nu \mid t \rightarrow \{e'_1, e_2\} \mid \nu' \mid t + 1}$$

$$(E\text{-PAIR2}) \frac{e_2 \mid \nu \mid t \rightarrow e'_2 \mid \nu' \mid t + 1}{\{v_1, e_2\} \mid \nu \mid t \rightarrow \{v_1, e'_2\} \mid \nu' \mid t + 1}$$

5.3.3 Additional Typing Rules for Image Manipulation

Operations that manipulate images may explicitly specify a range of valid input sizes (*i.e.*, size constraints) on their input to ensure correct processing. In the example below, the face count opcode requires that its input be in the size range of 320 to 1024 (using manageable image widths as a size rather than actual total numbers of pixels which fall in the millions of pixels). The definitions of these opcodes implicitly combine an aspect of T-WEAKEN, by using size ranges (rather than single values) in their constrained size variables. This alternate approach has a distinct advantage over using T-WEAKEN prior to T-OPAPPLY (as is done in the Conditional in Section 5.2.5); namely the size and cost bounds are more accurate if they are computed against the input's actual size bound instead of weakened size bounds. It should be obvious to the reader that the *latentcost* values for the functions in this section have also been contrived in a manner to ease the presentation and discussion.

$$(T\text{-FACECT}) \frac{\mathit{type}(\mathit{facect}) = \mathit{Image}^{\{r_1, r_2\}} \rightarrow \mathit{Int}^{\{n_1, n_2\}} \quad \mathit{constr}(\mathit{facect}) = \kappa_1}{\mathit{facect} : \mathit{Image}^{\{r_1, r_2\}} \rightarrow \mathit{Int}^{\{n_1, n_2\}} \text{ \$ } 0, \kappa_1}$$

$$\kappa_1 = \{r_1 \geq 320, r_2 \leq 1024\}$$

$$\mathit{latentcost}(\mathit{facect}, \tau) = (\mathit{maxsize}(\tau)/2)$$

A resampling operation is analogous to casting an image to be of a different size.⁴ It has no explicit values for the size variables of its input. Its only constraint is that the input be of some positive size (greater than zero).

$$(T\text{-RESAMPLE}) \frac{\mathit{type}(\mathit{resample}) = \mathit{Int}^{\{n_1, n_2\}} \times \mathit{Image}^{\{r_1, r_2\}} \rightarrow \mathit{Image}^{\{r_3, r_4\}} \quad \mathit{constr}(\mathit{resample}) = \kappa_1}{\mathit{resample} : \mathit{Int}^{\{n_1, n_2\}} \times \mathit{Image}^{\{r_1, r_2\}} \rightarrow \mathit{Image}^{\{r_3, r_4\}} \text{ \$ } 0, \kappa_1}$$

$$\kappa_1 = \{n_1 > 0, r_1 > 0, r_3 = r_1 * n_1, r_4 = r_2 * n_2\}$$

$$\mathit{latentcost}(\mathit{resample}, \tau) = (\mathit{maxsize}(\tau)/8)$$

Image sensing hardware (*e.g.*, a web camera) has the capability to capture images in a range of possible resolutions. While the sized annotation makes sizing explicit in the type system, our programming language lacks explicit type annotation within its syntax. Thus, we formally define a new primitive to allocate a new image sensor (*ala* `image`) that accepts a resolution range explicitly, as an argument.

$$(T\text{-SIZEDIMAGE}) \frac{\mathit{type}(\mathit{sizedimage}) = \tau \rightarrow \mathit{Sensor Image}^{\{r_1, r_2\}} \quad \mathit{constr}(\mathit{sizedimage}) = \kappa_1}{\mathit{sizedimage} : \tau \rightarrow \mathit{Sensor Image}^{\{r_1, r_2\}} \text{ \$ } 0, \kappa_1}$$

$$\tau = \mathit{Int}^{\{n_1, n_2\}} \times \mathit{Int}^{\{n_3, n_4\}}$$

$$\kappa_1 = \{n_1 > 0, n_3 > 0, r_1 \geq \mathit{min}(n_1, n_3), r_2 \leq \mathit{max}(n_2, n_4)\}$$

$$\mathit{latentcost}(\mathit{sizedimage}, \tau) = (\mathit{maxsize}(\tau)/8)$$

⁴While a resampling operation does increase the number of pixels in an image, it does not improve image *quality*. We will return to this issue in Section 5.4.

5.3.4 Flowtypes

Finally we introduce a new function whose sole purpose is to inject run-time constraints (a Flowtype in our nomenclature). This function annotates that its argument has an explicit deadline within the type system. For example, the example with function `period()` from the Introduction would be implemented as syntactic sugar using `deadline()`.

$$(T\text{-DEADLINE}) \frac{\Gamma \vdash n : \text{Int} \{n,n\} \$ 0, \kappa_1 \quad \Gamma \vdash e_2 : \tau \$ c_2, \kappa_2}{\Gamma \vdash \text{deadline } n \ e_2 : \tau \$ c_2, \kappa_1 \cup \kappa_2 \cup \kappa_3}$$

where $\kappa_3 = \{c_2 \leq n\}$

$$(E\text{-DEADLINE}) \ (\text{deadline } n \ e_2 \mid \nu \mid t) \rightarrow (e_2 \mid \nu \mid t)$$

5.3.5 In Practice

In this section we have expanded our core language as much as possible so as to reflect our real operating and tasking environment. We will now construct several examples on which we apply our type system.

Inferring Optimal Image Resolution

In practice, a user may not provide a specific resolution (size) bound for images that are used as part of a larger computation in which the image is not part of the desired output. For example, a user who wishes to determine whether or not a light is on in a particular office is interested in a boolean result, not the intermediate image used to generate this result. As image sensors are able to capture images in a range of possible resolutions, our type system can use its size constraint system to suggest an optimal resolution, a range of feasible resolutions, or indicate that there is no feasible solution for the program as specified.

Example 1:

Face counting within an image from an image sensor with no explicit size bounds:

`facect(get(image))`

$$\begin{array}{c}
 \frac{\text{type}(\text{get}) = \text{Sensor } \tau \rightarrow \tau \quad \text{image} : \text{Sensor Image } \{r_1, r_2\} \text{ \$ } 0, \kappa_1}{\text{get}(\text{image}) : \text{Image } \{r_1, r_2\} \text{ \$ } c_1, \kappa_1} \quad \begin{array}{l} \text{(T-IMAGESENSOR)} \\ \text{(T-SENSORREAD)} \end{array} \\
 \\
 \frac{\text{type}(\text{facect}) = \text{Image } \{r_1, r_2\} \rightarrow \text{Int } \{n_1, n_2\} \quad \text{constr}(\text{facect}) = \kappa_2}{\text{facect} : \text{Image } \{r_1, r_2\} \rightarrow \text{Int } \{n_1, n_2\} \text{ \$ } 0, \kappa_2} \quad \text{(T-FACECT)} \\
 \mathcal{D}_1 = \frac{\text{facect}(\text{get}(\text{image})) : \text{Int } \{n_1, n_2\} \text{ \$ } c_2, \kappa_1 \cup \kappa_2}{\text{facect}(\text{get}(\text{image})) : \text{Int } \{n_1, n_2\} \text{ \$ } c_2, \kappa_1 \cup \kappa_2} \quad \begin{array}{l} \mathcal{D}_0 \\ \text{(T-OPAPPLY)} \end{array}
 \end{array}$$

$$c_1 = 1 + \text{latentcost}(\text{get}, r_2) = 1 + (r_2/8)$$

$$c_2 = 1 + c_1 + \text{latentcost}(\text{facect}, r_2) = 1 + c_1 + r_2/2$$

$$\kappa_1 = \{r_1 \geq r_{min}, r_2 \leq r_{max}\}$$

$$\kappa_2 = \{r_1 \geq 320, r_2 \leq 1024\}$$

Solving the constraints for r_1 and r_2 (we minimize r_1 and maximize r_2 subject to the constraints given above, we determine the simple constraints that the resolution of the image to manipulate, and thus the sensor itself, fall in the range [320, 1024]. Thus the SSD may allocate any available sensor that can produce images within this range of resolutions. A camera that can capture images at resolutions ranging from 1024 to 4096 is valid for this service fragment (provided it samples at 1024), as is a camera that can capture images at the range from 320 to 512.

Bear in mind that the minimum resolution in a range of resolutions is not always the most desirable value; while the minimum will consume the least computational resources, it may do so at the expense of computation confidence (*e.g.*, it may not be possible to detect all the faces in an image if the resolution is too small). As a result we advocate the use of

the maximum size, provided there are resources available to accommodate the additional processing overhead. This processing overhead is easily measured by the cost function (c_2 , in the above).

Example 2:

Face counting within an image from an image sensor with no explicit size bounds *with an explicit deadline*: `deadline 322 facect (get(image))`

$$\frac{\text{322 : Int } \{n,n\} \text{ \$ } 0, \kappa_0 \quad \mathcal{D}_1}{\text{deadline 322 facect(get(image)) : Int } \{n_1,n_2\} \text{ \$ } c_2, \kappa_0 \cup \kappa_1 \cup \kappa_2 \cup \kappa_3} \text{(T-DEADLINE)}$$

$$c_1 = 1 + \text{latentcost}(\text{get}, r_2) = 1 + (r_2/8)$$

$$c_2 = 1 + c_1 + \text{latentcost}(\text{facect}, r_2) = 1 + c_1 + r_2/2$$

$$\kappa_1 = \{r_1 \geq \mathbf{r}_{min}, r_2 \leq \mathbf{r}_{max}\}$$

$$\kappa_2 = \{r_1 \geq 320, r_2 \leq 1024\}$$

$$\kappa_0 = \{n = 322\}$$

$$\kappa_3 = \{c_2 \leq 322\}$$

In this example the valid range for capture is no longer $[320, 1024]$, when we solve for r_1 and r_2 we are limited to the range $[320, 512]$ as larger values of r_2 would exceed the constraint imposed by κ_3 .

It is worth noting that, despite slight differences in the typing derivation, this example has a size bound result that is identical to an example with explicit (user-specified) size bounds that are the same as those imposed by the `facect` opcode:

`(deadline 322 facect(get(sizedimage({320, 1024}))))`.

Example 3:

Face counting within an image from an image sensor with no explicit size bounds *as the response in a trigger expression:*

trigger e_0 facect(get(image))

We assume the presence of a Boolean typed expression e_0 with cost c_0 and constraint set κ_0 .

$$\frac{e_0 : Bool \ \$ \ c_0, \kappa_0 \quad \mathcal{D}_1}{\text{trigger } e_0 \ \text{facect}(\text{get}(\text{image})) : Int^{\{n_1, n_2\}} \ \$ \ c_3, \kappa_0 \cup \kappa_1 \cup \kappa_2} \text{(T-TRIG)}$$

$$c_1 = 1 + \text{latentcost}(\text{get}, r_2) = 1 + (r_2/8)$$

$$c_2 = 1 + c_1 + \text{latentcost}(\text{facect}, r_2) = 1 + c_1 + r_2/2$$

$$\kappa_1 = \{r_1 \geq r_{min}, r_2 \leq r_{max}\}$$

$$\kappa_2 = \{r_1 \geq 320, r_2 \leq 1024\}$$

$$c_3 = (\mathcal{T} * c_0) + c_2$$

If expression e_0 starts running at time t_{start} , transitions to **true** at time t_{true} , and $\text{delay}(e_0)$ is the time to evaluate e_0 then we can write: $\mathcal{T} = \lceil \frac{t_{true} - t_{start}}{\text{delay}(e_0)} \rceil$. Recognizing that c_0 is a delay (bound) for e_0 , we can express this as $\mathcal{T} = \lceil \frac{t_{true} - c_2}{c_0} \rceil$. This example underscores the correctly bounding (or estimating) the “weight” of \mathcal{T} is dependent on estimating the factors required to cause e_0 to transition to true, which may include information about the sensing environment \mathcal{E} and costs determined elsewhere in the derivation (*i.e.*, to supply t_{start}).

Example 4:

Face counting within a *resampled* image, originally captured from an image sensor with no explicit size bounds:

`facect (resample {4, get(image) })`

$$\begin{aligned}
\mathcal{D}_2 &= \frac{4 : Int \{n,n\} \$0, \{n = 4\} \quad \mathcal{D}_0}{\{4, \text{get}(\text{image})\} : Int \{n,n\} \times Image \{r_1,r_2\} \$ c_1, \kappa_1 \cup \{n = 4\}} \text{(T-PAIRCONS)} \\
\mathcal{D}_3 &= \frac{\text{type}(\text{resample}) = Int \{n,n\} \times Image \{r_1,r_2\} \rightarrow Image \{r_3,r_4\} \quad \text{constr}(\text{resample}) = \kappa_2}{\text{resample} : Int \{n,n\} \times Image \{r_1,r_2\} \rightarrow Image \{r_3,r_4\} \$ 0, \kappa_2} \text{(T-RESAMPLE)} \\
\mathcal{D}_4 &= \frac{\mathcal{D}_2 \quad \mathcal{D}_3}{\text{resample } \{4, \text{get}(\text{image})\} : Image \{r_3,r_4\} \$ c_2, \kappa_1 \cup \kappa_2 \cup \{n = 4\}} \text{(T-OPAPPLY)} \\
&\frac{\text{type}(\text{facect}) = Image \{r_3,r_4\} \rightarrow Int \{n_3,n_4\} \quad \text{constr}(\text{facect}) = \kappa_1}{\text{facect} : Image \{r_3,r_4\} \rightarrow Int \{n_3,n_4\} \$ 0, \kappa_3} \text{(T-FACECT)} \\
&\frac{\text{facect} : Image \{r_3,r_4\} \rightarrow Int \{n_3,n_4\} \$ 0, \kappa_3 \quad \mathcal{D}_4}{\text{facect } (\text{resample } \{4, \text{get}(\text{image})\}) : Int \{n_3,n_4\} \$ c_3, \kappa_4} \text{(T-OPAPPLY)}
\end{aligned}$$

$$c_1 = 1 + \text{latentcost}(\text{get}, r_2) = 1 + (r_2/8)$$

$$c_2 = \text{latentcost}(\text{resample}) + c_1$$

$$c_3 = \text{latentcost}(\text{facect}) + c_2$$

$$\kappa_1 = \{r_1 \geq r_{min}, r_2 \leq r_{max}\}$$

$$\kappa_2 = \{n > 0, r_1 > 0, r_3 = r_1 * 4, r_4 = r_2 * 4\}$$

$$\kappa_3 = \{r_3 \geq 320, r_4 \leq 1024\}$$

$$\kappa_4 = \{n = 4\} \cup \kappa_1 \cup \kappa_2 \cup \kappa_3$$

We obtain that $r_1 * 4 \geq 320 \Rightarrow r_1 \geq 80$ and $r_2 * 4 \leq 1024 \Rightarrow r_2 \leq 256$. Minimizing for r_1 and maximizing for r_2 gives the resolution range [80,256] required of the sensor to be allocated for this fragment.

Worst-Case Computational Cost and Expression Size Bounds

The worst case computational cost is given as a result of our type system as the first argument after the \$ in a typing judgment. This is a scalar value and could be adjusted or calibrated as a function to estimate actual execution times on various physical resources. The worst-case computational bounds provided by our system provide one static-time validation mechanism to determine if explicit deadlines (or other run-time constraints given other Flowtypes) cannot be met as specified. As presented, the `deadline` opcode relies on the unadjusted (discretized) computation bound.

In addition to static verification, the Service Dispatcher component of `SNBENCH` often must partition a task across physical resources if there is insufficient computing resources available on a single node to accommodate the entire task. The worst case computational cost, as presented, provides us with an initial metric to guide the allocation of physical computing nodes and sensory resources for a given task (and its constituent subtasks).

Similarly, for any given task or sub-task we can look at the upper bound size annotation on its type information to determine the potential network overhead associated with partitioning the larger task at that point.

5.3.6 Implementation Details

The type system described in this document has been implemented in Java and utilizes the open-source JavaCC project [Sri]. It is presented to the user as part of the `SNBENCH` development tool chain, and is automatically invoked when compiling a high-level programming language (*i.e.*, `SNAFU`) to `STEP`. As the implementation checks `STEP` code rather than `SNAFU` code, there is no technical challenge to separating the use of the type checker *apart from* the `SNAFU` compiler, however (1) `SNAFU` is eminently more readable than `STEP` and (2) the implementation includes several hooks to trace `STEP` typing errors back to line numbers within the `SNAFU` code as a convenience to its users. While the type system can exist without `SNAFU` (and inevitably will when other high-level languages are provided), the benefits of `SNAFU` are diminished without type checking.

Our implementation of the type checking engine includes some degree of support for all STEP constructs, rather than only the Core STEP subset that is presented in this Chapter. Naturally, there are some significant limitations to that support; for STEP constructs for which we lack complete formalism, we use heuristics to bound cost and, for opcodes which lack size constraint annotation, we are forced to default back to basic, unsized type checking.

To solve the constraint sets that are built as a result of the sized type checking, we invoke the GNU Linear Programming Toolkit (GLPK) [And], which can be used to solve a system of constraints for linear programming, and mixed integer programming. The decision to use GLPK is based on project maturity, community support and API availability. At present we emit our sizing constraints in the GNU MathProg language and invoke the GLPK via external script, though nothing (other than time) precludes finer integration via the GLPK Java Interface [Bjo].

Figures 5.1 and 5.2 are screen shots taken from the SNAFU development environment that feature results generated from the implementation of the type checker.

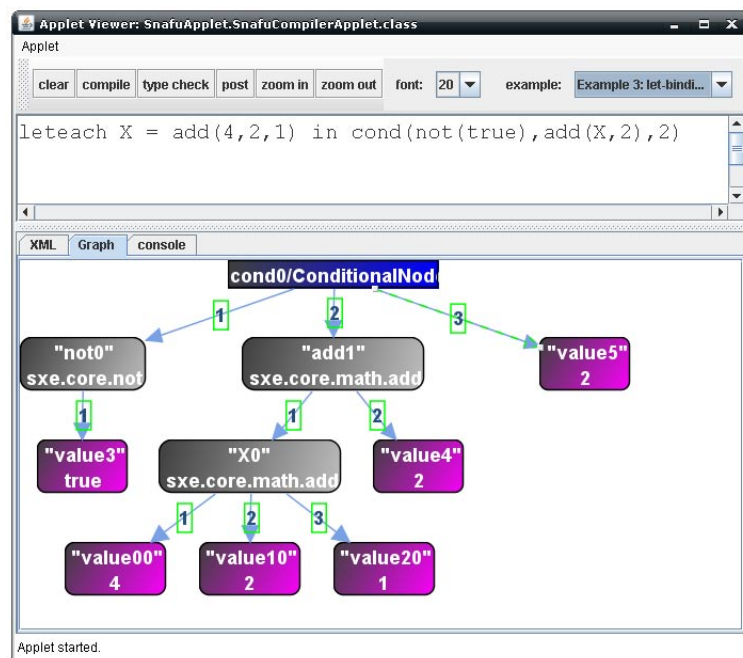


Figure 5.1: A screen shot from the SNAFU compiler showing the successful type checking of a STEP program. By default, the feedback is given graphically to the user.

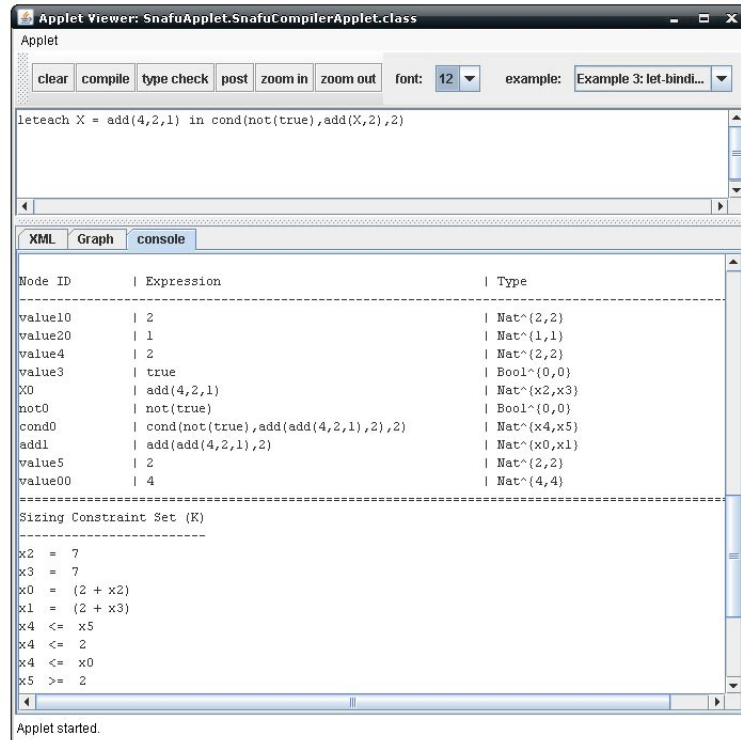


Figure 5.2: A screen shot from the SNAFU compiler showing size annotation constraints that result from type checking a STEP program.

5.4 Future Work

5.4.1 Additional Type Annotations

In this chapter we have presented upper and lower size bound data type annotation, yet other useful annotations exist and can be easily integrated into this type system by extending the existing annotation pair to a tuple or larger ordered set and defining the desired subtyping relation.

One such example is the notion of image quality, which is different from data size. Data *size* is used bound the operational requirements of a function (including those that manipulate images), whereas image *quality* speaks to the valid data in the image. As far as operational/functional correctness is concerned a resized image is operationally valid, however with respect to the desired programmatic output, a smaller image that has been resized to a larger resolution is not truly interchangeable with an image captured at a

larger resolution. When an image is resized (*e.g.*, via scaling or resampling, say) the size (resolution) of the image changes no longer reflects the number of data points it contained originally (we are calling this “quality”).

We could easily support a notion of an image’s quality within our type annotations, by adding a dimension for the “lowest” value that we have ever seen for an image’s lower size bound.⁵ This value could distinguish between a true high resolution image and data that has been up-cast or coerced to satisfy a function’s size constraints. Quality also has a well established meaning with respect to numerical data as well, and might be defined to reflect potential for rounding errors, data accuracy, *etc.* Certainly other image and video related aspects could be tracked as well, including color-depth for images, frame rates for video, and so on.

5.4.2 Applications to Image Pyramids

Image Pyramids [Kro91],[AH91] are a well established technique in the field of image processing. The technique involves maintaining multiple copies of the same image at different resolutions (the a hierarchy of resolutions form a logical pyramid) such that the appropriate resolution can be selected for manipulation depending on the needs of the manipulation function. The applications of this formalism to this community is potentially two-fold. (1) We can extend the type system to include an image pyramid as a first class type and extend the annotations to support the list of resolutions available in the pyramid. (2) Static analysis of the image processing flow can tell us precisely which resolutions need to be kept in the pyramid and which can be removed. In the latter case, the potential benefit of `SNBENCH` and sized typing is quite significant as it might be possible to remove the pyramid entirely. For distributed computations, we can split the pyramid into individual image instances based on the analysis of size/use and ensure that we are passing as few copies of the image (inside the pyramid) and as few copies of the pyramid itself, as possible.

⁵Recall the definition of `T-RESAMPLE` increases both the upper and the lower size bound.

5.4.3 Cost and Size Signatures For New Opcodes

The operational correctness of the type system is contingent on the presence of accurate size, cost and constraint data for Opcodes (primitive operators). `SNBENCH` provides a facility by which new Opcodes may be added to a service library quickly and easily, through the implementation of a simple Java interface. At present the type system maintains an embedded definition for the latent costs and size constraints of the current Opcode library, however for the sustainability of the type system, it is essential that these definitions are provided by the Opcode authors and automatically extracted directly from the Opcode definitions. Beside changing the Opcode implementation interface, the type system must also change to import the rules from the Opcode library directly. There is also a concern that Opcode authors might specify very weak size constraints and very high costs (possibly lacking the knowledge to do so correctly) and that the end result is a type system that is only as strong as its weakest link. Any future work that would automatically extract this information from an Opcode implementation would be a fantastic solution to this problem (and others).

5.4.4 Different Models of Cost

At present our type system provides a single model of computational cost, a worst-case upper bound. As there is the possibility for multiple size annotations (dimensions) of types, there is an opportunity to provide multiple cost metrics based on these other dimensions. Defining a so-called “minimum” computational cost (or optimal case) would be trivial, and other cost models including non-computation costs models (*e.g.*, defining some financial cost, or total memory utilization cost) would be possible, if not trivial. The approach that seems the most straight forward to enabling multiple cost models would be to provide functional costs, such that all costs computed on the right hand side of the \$ would be user/programmer customizable and configurable as a function of the data available in the typing rules.

5.4.5 Expanding to “Complete” STEP

At present we support only a subset of the STEP programming language. The remaining STEP constructs (*e.g.*, streaming/persistent triggers and their associated read semantics) are difficult to represent as a result of their asynchronous execution and complex internal state. Despite these challenges, we have worked to establish a typing formalisms for these constructs however they are presently incomplete. Despite this, the present limited form is useful for non-persistent, “straight-shot” programs (which actually capture a significant amount of image processing tasks). For persistent/streaming trigger constructs we apply a simple heuristic: we solve the computable portion and multiply by estimates to derive the approximate costs for services that include non-Core portions of STEP. Regardless of this makeshift solution, it would be advantageous (and naturally appealing) to have a type system that includes the complete STEP language.

5.5 Conclusions

In this chapter, we have presented our formal type system for multi-dimensional sized types. Unlike other sized type systems our work tracks both an upper and lower size bound for data, defines a logical subtype relation for images capable of bounding computation, maintaining functional correctness, and deduce feasible data sizes from implicit and explicit constraints within program fragments. We presented this system and have provided examples that illustrate the use of the type system. We are confident in the many potential uses for this formalism to the Image processing and Sense and Respond communities.

Chapter 6

A Native Execution Environment for Embedded Devices

Embedded sensing and actuation environments require language abstractions that enable temporal execution constraints and safe resource sharing. Unfortunately, current real-time language offerings are either too cumbersome for lightweight devices or too simplistic to facilitate any guarantees of functionally or temporally correct behavior. `SNBENCH` provides a high-level programming language, compilers and run-time support to manage an embedded sensing and actuation environment with attention toward this challenge. In this chapter we detail a compiler, virtual machine, and small ISA that naturally extend the applicability of the `SNBENCH` infrastructure to light-weight, resource constrained devices. The simple instruction set (`snCode`) can be executed by a lightweight, stack-based virtual machine *without garbage collection*. This work opens the door to a native Sensor eXecution Environment (`nSXE`) that, in its reduced CPU and memory overhead, is appropriate for embedded devices and well provisioned hosts alike. In addition to the reduced processing and memory requirements, this alternate execution environment provides fine grained resource control, memory protection and predictable performance with which we can extend soft real-time performance guarantees to the programs executed on top of the `SNBENCH`.

6.1 Overview

The particular (embedded) SN infrastructure targeted by SNBENCH is, in part, characterized by the presence of resources of varied capabilities. The standard run-time execution environment of SNBENCH is appropriate for well apportioned devices that do not require fine grained resource control. It has always been our intent to provide a “lightweight” execution environment for highly constrained sensing and actuation devices that, by their very nature, are subject to legitimate real-time constraints.

In this chapter we describe the design and technical challenges of the “native” Sensor eXecution Environment (nSXE). The Sensor eXecution Environment’s primary task, the interpretation and execution of remotely dispatched Sensorium Task Execution Plan (STEP) XML programs, is achieved on the nSXE by compiling STEP programs into a functionally equivalent lightweight half-byte code representation (snCode). snCode features a very small set of opcodes (4 bits worth, naturally), and thus the execution environment for this instruction set can be quickly and easily ported to a myriad of devices. Translation of the functional-style STEP language into an imperative, stack-based snCode representation requires unwinding the functional program; and the compilation rules are included herein. While the STEP-to-snCode compiler and corresponding nSXE could eventually replace the STEP-based SXE for all devices, the snCode is not a replacement for STEP (*i.e.*, our static analysis techniques target STEP). We also detail the design and requirements of our proof of concept nSXE (virtual machine) and provide benchmarks that compare the performance of the native SXE with the original SXE.

6.2 Related Work

Clearly our work is not the first to provide lightweight code to task embedded, sensing and actuation devices. Although we briefly mention non-VM based, approaches we believe the advantages afforded by Virtual Machines in embedded deployments are difficult to dispute. Separating performance concerns, VMs extend safety to embedded applications including

memory protection and separate, protected services that monitor the health of embedded applications and perform dynamic updates. Such functionality is ideal in embedded devices, particularly when devices must be updated with new code and deployed in inaccessible locations (*e.g.*, Mars). We would readily trade a performance/computational throughput penalty for interpretation, provided we can obtain predictable performance.

The Lego Robotics platform (“Mindstorm” and its successor “NXT”)[KL99] has several programmatic interfaces to tasking the robot devices. The user community’s Not Quite C [Bau] offers a low-level C-like interface to programming the Mindstorm while Not eXactly C [Han] provides a much richer, yet still low-level C interface for the NXT devices (each of which is compiled into a bytecode for the devices). Much closer to high-level interface of STEP is the graphical tasking language of NXT-G [LN] which allows users to specify their programs graphically, in a functional manner. While the interfaces are somewhat similar, STEP provides the capability to task devices beside robotics, to specify programs at a scope of an overall distributed computation (not just role local device behaviors) and finally to be inherently extensible to support new functionalities and devices by the Opcode and GenericSensor abstractions, respectively.

As we offer a bytecode and virtual machine, the natural comparison is to Java Standard Edition (Java) and its JVM or the Java2 Micro Edition (J2ME) and its KVM. While these comparisons are apt there are several clear ways in which our works differ. Java suffers from a tension between supporting many target device platforms while simultaneously offering an extremely large and well provisioned core runtime API. A JVM implementation must support over 200 bytecodes (admittedly, many operations are virtually identical, *i.e.*, perform the same operation, but on different types of data). While the KVM for J2ME aims to be a smaller, yet “complete” JVM, even the smallest “complete” JVM is still overblown for our needs. For example, snCode is generated from the compilation of STEP, which is a functional language. Thus snCode does not manipulate explicit state and does not require a heap or GarbageCollector. Additionally neither the Java or J2ME address real-time concerns.

The Real Time Specification for Java (RTSJ) does extend real-time concerns and constraints to Java, however they are largely centered around the implementation and extensions of the core API and do not restrict the bytecode in any way. To extend predictable execution to the RTSJ we must have worst- case execution time (WCET) for the bytecode and a hard rule about how threads will be implemented in the JVM (*i.e.*, native or user level). Several Java “bytecode-interpreter in hardware” solutions exist, however we are interested in maximum portability and device independence. Related, the Squawk VM [SCC⁺06] enables Java bytecode to be executed on embedded devices, however they target a particular platform, the SunSPOT device, and thus the applicability is sorely limited.

Programming sense and respond services from a high level language is akin to the tasking of small embedded devices and is thus related to Active Sensor Networks [LGC05], which leverages the TinyScript scripting language (among others) to task sensors. TinyScript is compiled into code that runs on the Mate Virtual Machine for Sensors and, as Mate requires the presence of TinyOS, the result in a rather limited target architecture and no memory safety or real-time performance guarantees.

Finally, we address why we do not use the GNU Compiler for Java [Lic] to produce native code from our Java SXE as suggested in previous publications. Unfortunately, compiling Java with GCJ requires a fair amount of tweaking for relatively complicated code (GCJ does not support all of Java) and we would need to port libgcj to every target device (the present version does not support threads, file I/O or networking). Even if GCJ could compile the SXE for the target device, the result would be too heavyweight and maintaining the binary output would require intimate knowledge of GCJ.

6.3 The STEP Virtual Machine

The requirements of the STEPvm and the snCode language itself were established in tandem, driven by the needs of the STEP language. As STEP programs are functional, there is no explicit state manipulation possible in user code. As a result the STEP Virtual Machine

(STEPvm), unlike the Java VM, does not provide dynamic memory allocation or garbage collection, however does utilize a variable stack. The STEPvm must maintain some shared state to support the runtime semantic of the STEP language constructs, however these memory needs are known at compilation time. There is no dynamic generation supported in STEP.

The snCode mnemonics, their function, and their effects on the stack are given in Table 6.1. The STEPvm is arguably a descendant of Landin’s Stack Environment Code Dump (SECD) abstract machine [Lan64], however STEPvm is considerably more simple insofar as the source language (STEP) lacks traditional functional recursion.

A central concept to STEP programming is the persistent trigger abstraction, which is used to specify asynchronous STEP code fragments that produce data streams. To support the simultaneous and concurrent execution of persistent triggers, the STEPvm requires multi-threaded execution. Each thread has its own context frame consisting of its own code segment, program counter, stack, small memory segment to pass arguments to library functions, and relative priority for scheduling. The STEPvm maintains a shared, global “environment memory”, a critical section lock pool and all thread context frames. New threads (frames) are created by the compilation process either from a program root or the presence of persistent trigger expressions.

In the STEPvm each thread contains its own code segment such that, when a thread expires or a new thread is added dynamically (*i.e.*, dispatched to the nSXE), instructions can be easily removed or added on a thread-by-thread basis. In STEP, threads do not generally have code in common, unless explicitly indicated by the presence of a `splice` STEP node; the splice node indicates that the value of a STEP node (or STEP sub-graph) should be shared by multiple parent nodes in the graph, without re-computation. The STEPvm’s shared environment memory is used to facilitate the semantic of splice nodes. In a snCode translation of the STEP splice node, the code first checks to see if a value has already been stored in the environment and will compute the value only if it does not already exist.

When a splice node enables the sharing of values across two (or more) threads of execution (*i.e.*, subtrees that are shared by different root nodes), there is a potential for a race condition: multiple threads could simultaneously see the value of the environment variable as non-initialized and each start computing the value in their own thread (computing the value multiple times). The STEPvm provides a critical section construct and it used in the splice node scenario to prevent the thread scheduler from causing a spliced subexpression to be executed by multiple threads simultaneously. The critical sections are identified by the shared environment memory values they ultimately compute, and therefore logically translate to locks on the specific shared memory variable. As STEP is defined, the splice nodes are the only environment memory variables that may exist in common across separate threads of execution in the STEPvm; this is the scenario in which protected access to the environment memory is required.

In addition to the features already specified, the STEPvm must also support the execution of the functional library operations (so called “STEP *Opcodes*”), invoked by the Expression nodes of the STEP language. The STEPvm is agnostic to the particular implementation of the *Opcodes* library, provided the individual functions/subroutines can produce valid data.¹

6.4 STEP Compilation

A STEP graph is a functional-style Sensor Network program, in which the STEP nodes represent sensors, values, and manipulation functions required to perform a given task. There are nearly a dozen STEP node types in the larger STEP programming language. Each node in a STEP graph is an instance of one of the STEP node classes; the node’s class defines its interpreted behavior and how it is connected to other nodes. For example, a conditional node has three children: a predicate node (or subgraph) that produces a boolean value, and two other children, one to be evaluated if the predicate is true and one to be evaluated if the predicate is false.

¹The STEP Opcode library is similar to Java’s core runtime API, `rt.java`.

PUSH	Push the arg on to the stack
POP	Pop a value off the stack
DUPE	Replicate the top value on the stack
JUMP	Unconditional jump to the line specified
JUMPF	Jump to the line specified if Pop() == <i>false</i>
JUMPV	Jump to the line specified if Pop() != <i>ERR</i>
ESTORE	Pop a value off the stack and store it into the shared environment memory specified as an operand
ELOAD	Push a value onto the stack loaded from the shared environment memory location (operand), pushes <i>ERR</i> if not found
NOP	No op (for label/line number consistency)
ARGLIST	Build an arglist from the operand many popped values from the stack
OP	Jump to a subroutine passing in the ARGLIST as arguments
MUTEXLOCK	Prevent any other thread from entering the critical section defined by the shared variable
MUTEXUNLOCK	Release the lock on the critical section
EBUFADD	Pop a value off the stack and store it into the shared environment FIFO <i>buffer</i> specified as an operand
EBUFRMV	Remove the oldest element from the named shared environment buffer and push this value onto the stack (blocks if buffer is empty)

Table 6.1: Opcodes for snCODE

In unwinding the functional style of the original STEP program graph, we translate STEP constructs to a directed flow graph and determine basic blocks [AU77]. In compilation, we apply dynamically generated labels to basic blocks to enable jumps/branching required for the `trigger` and `cond` constructs of the STEP programming language. The compilation rules presented take a STEP node (class) and an optional label as arguments, and produce a functionally equivalent snCode representation. When compilation is complete we replace the logical label references with line numbers.

Below we present the compilation strategy for each of the STEP node classes, including the node's representation in STEP (as XML) and its compilation rule. The XML examples have been abbreviated to call attention to the attributes most relevant to the compilation process. A more detailed discussion of the STEP nodes is presented in Chapter 3.

Value nodes

Value nodes convey a constant/literal value. In the example STEP value node below is of type integer with value 45 and has the unique ID `intnode`.

```
<value id="INTNODE">
  <snoject type="INTEGER">
    45
  </snoject>
</value>
```

The compilation rule is given below, which simply pushes the value of the node onto the stack. The label argument, which is used for other STEP constructs to reconcile line numbers as jump targets, will be clear in further examples.

$$\mathcal{C}(\text{label}, \text{value}) \Rightarrow \left| \begin{array}{l} \text{label: PUSH value} \end{array} \right.$$

Conditional nodes

Conditional nodes convey an “if-then-else” semantic. It has three child nodes/subtrees, the predicate (P), the subtree to be executed if true (T) or the subtree to be executed if false (F).

```
<cond id="CONDNODE">
  <P/><T/><F/>
</cond>
```

The compilation of conditional nodes utilize labels to ensure the correct jump target. The new labels, `cond.f` and `cond.r` are derived from the `cond` node’s id attribute.

$$\mathcal{C}(\text{label}, \text{cond}(\text{P}, \text{T}, \text{F})) \Rightarrow \left| \begin{array}{l} \mathcal{C}(\text{label}, \text{P}); \\ \text{JUMPF } \text{cond.f}; \\ \mathcal{C}(\text{nil}, \text{T}); \\ \text{JUMP } \text{cond.r}; \\ \mathcal{C}(\text{cond.f}, \text{F}); \\ \text{cond.r: NOP}; \end{array} \right.$$

Expression nodes

Expressions convey a particular computation (STEP *Opcode*) to be performed by the SX-E/nSXE host, with the child nodes of this node passed as arguments of the operation. We assume Opcode library implementations are available and, in some way trusted, not entirely unlike the implementation of the host-specific Java runtime classes (rt.java).

```
<expnode opcode="SXE.CORE.MATH.ADD" id="OTHER">
  <A/>
  <B/>
</expnode>
```

Invocation of a STEP *Opcode* involves two operations in snCode; First, arguments are removed from the stack and placed in the thread’s argument memory via a call to ARGLIST. Second, a call to OP passes control (and the argument memory) to the library function outside the STEPvm (*i.e.*, a library call). The result of the OP is passed back to the STEPvm and placed at the top of the variable stack.

$$\mathcal{C}(\text{label}, \text{exp}(\mathbf{A}, \mathbf{B})) \Rightarrow \left| \begin{array}{l} \mathcal{C}(\text{label}, \mathbf{A}); \\ \mathcal{C}(\text{nil}, \mathbf{B}); \\ \text{ARGLIST } 2; \\ \text{OP } \text{exp}; \end{array} \right.$$

Transient Trigger nodes

Triggers enable user specified reactions that are “triggered” by the evaluation of a user specified predicate. In STEP, triggers are either of two forms; transient or “fire-once” triggers and persistent or “streaming”. Persistent triggers execute their predicates in perpetuity, concurrent with their parent expressions. Transient triggers run on demand of their parent nodes and are thus, much easier to understand.

“Basic” Trigger nodes: The basic transient trigger node executes the predicate expression repeatedly, until it evaluates to true. At that point the response is executed and returned as the value of the trigger expression.

```

<trigger id="TRIGGER">
  <P/><R/>
</trigger>

```

Compilation into snCODE is straightforward, utilizing the JUMPF opcode to jump back to the predicate expression if the value on the top of the stack is false.

$$\mathcal{C}(\text{label}, \text{trigger}(\text{P}, \text{R})) \Rightarrow \left\{ \begin{array}{l} \mathcal{C}(\text{label}, \text{P}); \\ \text{JUMPF } \text{label}; \\ \mathcal{C}(\text{nil}, \text{R}); \end{array} \right.$$

While-Trigger nodes: The transient while trigger node executes the predicate expression repeatedly, and evaluates the response while the predicate is true. When the predicate evaluates to false, the last value of the response is returned (which may be NIL).

```

<while_trigger id="WHILETRIGGER">
  <P/><R/>
</while_trigger>

```

The compilation of the while_trigger to snCode entails tracking the previous evaluation of the response (R), which we initially place in the stack as *ERR*, indicating to the “caller” that P was initially false (*i.e.*, the response has never executed). Should P be initially true, we repeatedly execute and maintain the prior value of R on the stack. When the execution is complete (*i.e.*, P is false) the prior value of R is the only value left on the stack at completion.

Additionally the while_trigger must also store its prior value of R in the environment to support the LAST_TRIGGER_EVAL construct (the compilation of which is detailed in Section 6.4).

$$\mathcal{C}(\text{label}, \text{while_trigger}(\mathbf{P}, \mathbf{R})) \Rightarrow \left. \begin{array}{l} l: \text{PUSH } ERR \\ \mathcal{C}(\alpha, \mathbf{P}) \\ \text{JUMPF } \beta \\ \text{POP} \\ \mathcal{C}(\text{nil}, \mathbf{R}) \\ \text{DUPE} \\ \text{ESTORE } \textit{trigger.id} \\ \text{JUMP } \alpha \\ \beta: \text{NOP} \end{array} \right|$$

Persistent Trigger nodes

The persistent trigger constructs supported by STEP associate a STEP (sub) program “response” (R) to be executed (*i.e.*, “triggered”) when the persistently evaluated “predicate” expression (P) evaluates to true. In effect, a persistent trigger runs in a separate thread of control and has a return type of a *stream* of values.

To support repeated evaluation in the SXE’s STEP interpreter, these triggers are always in the “ready” node queue for scheduling.² To achieve a similar effect in the nSXE’s imperative version, persistent triggers run in a separate thread in the VM.

Discretization from the stream of values into individual reads are handled by the STEP read node construct, which is detailed below. Value exchange between the trigger thread and the reader occurs using the environment memory which is shared across all threads. Values from the trigger are placed into the memory named for the calling read node and the read node encapsulates the specific read semantic. By default, a stream value is a single memory location; only the buffered read requires utilizing more than one memory location for its trigger/read semantic.

²This is an over-simplification, but conveys the spirit of the persistent trigger.

Level Trigger nodes: Level trigger nodes fire their response expression every time the predicate evaluates to true.

```
<level_trigger id="LEVEL_TRIGGER">
  <P/><R/>
</level_trigger>
```

$\mathcal{C}(l, \text{level_trigger}(P, R)) \Rightarrow$	<pre>// New VM Thread C(l, P); JUMPF l; C(nil, R); DUPE ESTORE read.id ESTORE trigger.id JUMP l;</pre>
---	--

Edge Trigger nodes: Edge trigger nodes fire the response expression every time the predicate evaluation transitions from false to true (and on the first true evaluation).

```
<edge_trigger id="EDGE_TRIGGER">
  <P/><R/>
</edge_trigger>
```

The snCode version must track the previous value of the predicate on the stack and the current value. The branches each ensure that the current predicate value becomes the new prior predicate value. We place FALSE on the stack as the first prior evaluation to facilitate the desired behavior that the first evaluation of P that produces TRUE should also result in the execution of R.

$\mathcal{C}(l, \text{edge_trigger}(P, R)) \Rightarrow$	<pre> // New VM Thread l: PUSH FALSE; C(α, P); JUMPF l; JUMPF β; PUSH TRUE; JUMP α; β: PUSH TRUE; C(<i>nil</i>, R); DUPE ESTORE <i>read.id</i> ESTORE <i>trigger.id</i> JUMP α; </pre>
--	---

LAST_TRIGGER_EVAL nodes

LTEs allow the predicate or response expressions of a trigger expression to refer to the previous value generated from a trigger response. As the LTE will only occur within the expression of a containing trigger the LTE can directly read the value of the trigger without threat of a race condition. The example below assumes the LTE is embedded in the some trigger expression (as defined above).

```

<level_trigger id="LEVEL_TRIGGER">
  ...
  <last_trigger_eval id="LTE" target="LEVEL_TRIGGER" />
  ...
</level_trigger>

```

As the STEP contains an attribute which specifies to which trigger the LTE is attached, the emitted snCode uses that target attribute as the label for access.

$$\mathcal{C}(l, \text{LTE}) \Rightarrow \left| \begin{array}{l} l: \text{ ELOAD } target \end{array} \right.$$

Read nodes

Read nodes are required to access to the value stream of a persistent trigger expression; a STEP graph must always have a read node parenting a persistent trigger unless the trigger is the root of the graph. We consider the specific read semantics: **block**, **nonblock**, **fresh**, and **buffered**.

Non-blocking read: A non-blocking read returns immediately with whatever value (or NIL) is currently available to the consumer.

```
<read id="READNODE" type="NONBLOCK">
  <level_trigger id="LEVEL_TRIGGER">
    <P/><R/>
  </level_trigger>
</read>
```

As a non-blocking read may legally return NIL, we compile the read to a load of the environment variable that contain the value for the trigger expression. If the value had not been set, the load will immediately return NIL, which is in keeping with the non-blocking read semantic.

$$\mathcal{C}(l, \text{read_nonblocking}(T)) \Rightarrow \begin{cases} l: \text{ELOAD } \textit{read.id} \\ \mathcal{C}(\textit{nil}, T) \end{cases}$$

Blocking read: The blocking read must ensure that a value is available before returning (*i.e.*, will not return NIL).

```
<read id="READNODE" type="BLOCK">
  <level_trigger id="LEVEL_TRIGGER">
    <P/><R/>
  </level_trigger>
</read>
```

For illustrative purposes, the implementation shown below is a busy wait. A better performing solution would be to require a new Opcode, EWAIT, which would wait until the shared environment variable is available before returning.

$$\mathcal{C}(l, \text{read_blocking}(T)) \Rightarrow \left. \begin{array}{l} l: \text{ ELOAD } read.id \\ \text{ DUPE} \\ \text{ JUMPV } \alpha \\ \text{ POP} \\ \text{ JUMP } l \\ \alpha: \text{ NOP} \\ \mathcal{C}(nil, T) \end{array} \right|$$

Fresh read: The fresh read must ensure that the returned value has not been previously read (*i.e.*, is a fresh computation).

```
<read id="READNODE" type="FRESH">
  <level_trigger id="LEVEL_TRIGGER">
    <P/><R/>
  </level_trigger>
</read>
```

The implementation shown clears the cached value (ensuring that the value is *freshly* computed for this consumption) and continues with a blocking read implementation to ensure the value is available before proceeding.

$$\mathcal{C}(l, \text{read_fresh}(T)) \Rightarrow \left. \begin{array}{l} l: \text{ PUSH } ERR \\ \text{ ESTORE } read.id \\ \beta: \text{ ELOAD } read.id \\ \text{ DUPE} \\ \text{ JUMPV } \alpha \\ \text{ POP} \\ \text{ JUMP } \beta \\ \alpha: \text{ NOP} \\ \mathcal{C}(nil, T) \end{array} \right|$$

Buffered read: The buffered read semantic implies the presence of a buffer on the values produced by the trigger, and that the reader will (eventually) retrieve every value produced.

```

<read id="READNODE" type="BUFFERED">
  <level_trigger id="LEVEL_TRIGGER">
    <P/×R/>
  </level_trigger>
</read>

```

The buffered read must not lose values; a concern not present in the previous read semantics. To facilitate this, we utilize the Opcodes EBUFADD and EBUFRMV to add an entry to a buffer and remove an entry from the buffer. We *require* that the trigger expression be compiled to use EBUFADD for the line `ESTORE read.id`. This is handled during compilation by the maintenance of internal state that indicates the type of read that encloses this trigger.

$$\mathcal{C}(l, \text{read_buffered}(T)) \Rightarrow \left| \begin{array}{l} l: \text{EBUFRMV } \text{read.id} \\ \mathcal{C}(\text{nil}, T) \end{array} \right.$$

Const nodes

Const nodes are used to prevent the re-evaluation of its child node. Its use is transparent to a service composer using the higher-level SNAFU language, rather it is emitted by the SNAFU compiler. For example the SNAFU `letconst` variable binding is syntactic sugar that allows a symbol to be used as a macro for a sub-expression that is computed once (*i.e.*, the compilation parents the subexpression with a const node). Similarly the `letonce` SNAFU binding is used to ensure that a symbol standing in for a sub-expression evaluates once per trigger evaluation.

```

<const id="CONSTNODE">
  <A/>
</const>

```

The const node is supported by using the environment memory store to store a computation and retrieving that value rather than recomputing the expression³.

³A node can be parented by a const node and be a splice target. In this scenario the value can be recomputed “out from under” the const node. In fact, this is the desired behavior and is the way in which the `letonce` binding is supported in SNAFU.

$$\mathcal{C}(l, \text{const}(\mathbf{A})) \Rightarrow \left. \begin{array}{l} l: \text{ ELOAD } \textit{const.id} \\ \text{ DUPE} \\ \text{ JUMPV } \alpha \\ \text{ POP} \\ \mathcal{C}(\textit{nil}, \mathbf{A}) \\ \text{ DUPE} \\ \text{ ESTORE } \textit{const.id} \\ \alpha: \text{ NOP} \end{array} \right|$$

Splice nodes

Splice nodes are used to encode a STEP *graph* into a tree for serialization in XML. Typically, the presence of a splice node indicates that a given STEP node has multiple parents. Splices are used for value re-use (not re-computation) and as such the call to a splice node should return the value of the evaluation of the target without necessarily re-evaluating the target.

One possible solution has any node that is the target of a splice node record its value in global state and the splice node itself return the value from the global target variable. A problem arises, however, should the splice node occur in the execution path before the target node. It is not possible to statically check that the target will be evaluated before the splice and swap the ordering (*e.g.*, a splice is on one branch of a conditional clause and the splice target on the other).

Instead we resolve splices as a special case where in both parents will jump to the same code block that first checks the environment to see if this value has already been computed and, if not, computes the value and stores it in the environment. To facilitate this, the compiler records all splice targets before compilation and, when such a node is encountered, an entry is placed in a table mapping targets to their corresponding label (line number).

```
<splice id="SPLICENODE" target="OTHER"/>
```

The first instance of either the splice or the target is compiled as indicated below and further splices jump into that code block. To protect against erroneous behavior from an

expression that is shared between nodes in different subtrees (*e.g.*, a subexpression shared between two graphs with distinct roots) we designate a critical section to ensure only one thread will evaluate the target.

$$\mathcal{C}(l, \text{splice}) \Rightarrow \left. \begin{array}{l} l: \text{MUTEXLOCK } target.id \\ \text{ELOAD } target.id \\ \text{DUPE} \\ \text{JUMPV } \beta \\ \text{POP} \\ \mathcal{C}(nil, target) \\ \text{DUPE} \\ \text{ESTORE } const.id \\ \beta: \text{MUTEXUNLOCK } target.id \end{array} \right|$$

Sensor nodes

Sensor nodes refer to physical resources to be manipulated by opcodes.

```

<sensor id="IMAGE-ANY0">
  <device id="ANY" type="IMAGE" />
</sensor>
```

Like value nodes, sensor nodes are compiled into literals.

$$\mathcal{C}(label, value) \Rightarrow \left. \begin{array}{l} label: \text{PUSH sensor } type@id \end{array} \right|$$

Socket nodes

Socket nodes send or receive serialized `SNBENCH` typed values to another `SXE` across the network. The socket node has an attribute “role” which defines whether it is a sender (sending the value of its child node across the network) or a receiver (accepting a value from the network and passes it up the `STEP` graph) and the “protocol” attribute specifies the type of transfer (*e.g.*, `HTTP/1.1 push` `HTTP/1.1 pull`, `UDP client`, `TCP server`). Structurally they

are compiled exactly as Expression nodes that invoke network operations. The compilation of socket nodes is still incomplete at this time, as its implementation on the SXE is tied to its HTTP server; we have not provided an HTTP server to the light-weight nSXE as of yet.

6.5 Benchmarks

We can clearly measure the performance gains of moving to snCODE programs from the original interpreted STEP SXE; specifically faster execution times and decreased memory usage. These findings confirm our expectations. To level the comparison, we use a modified SXE STEP interpreter that removes all tasks that are not needed for our execution time benchmarking purposes (the HTTP server, SSD heartbeat communication, *etc*). Similarly our STEPvm has been implemented in Java and utilizes the same Opcode library (to remove any bias for faster opcode implementations). The latter implies that further memory and execution time improvements could be achieved.

In our benchmarks we compare execution time and peak memory usage between program execution in the STEP interpreter and STEP compiled to snCODE. We do compare only the execution/evaluation time for both the traditional SXE evaluator and the STEPvm. We are not including the time to load the STEP graph, or the time to compile the STEP graph in these benchmarks.

The first benchmark (Figure 6.1) attempts to clearly isolate the computational overhead of evaluating the STEP Graph compared to the linear, snCode representation. In our first test use a `while_trigger` to count up to n , and show the execution times of both the snCode Evaluator (STEPvm) and the traditional STEP Evaluator (the SXE). These benchmarks are measured against the Sun 1.6_03 Java Virtual Machine running in server mode with parallel garbage collection, assertions and debugging messages enabled (to avoid dead code elimination effects). Additionally, to that end, we are using a *cold* JVM (*i.e.*, a JVM instance is created per execution trial); doing so ensures that we limit hot spot (dynamic recompilation) effects and dead code elimination (*i.e.*, performance times improve drastically

over successive runs within the same VM). These benchmarks have been executed under Linux 2.6 running on an Intel Pentium-M 1.7GHz processor.

This benchmark shows a clear performance improvement in the snCode evaluation model.

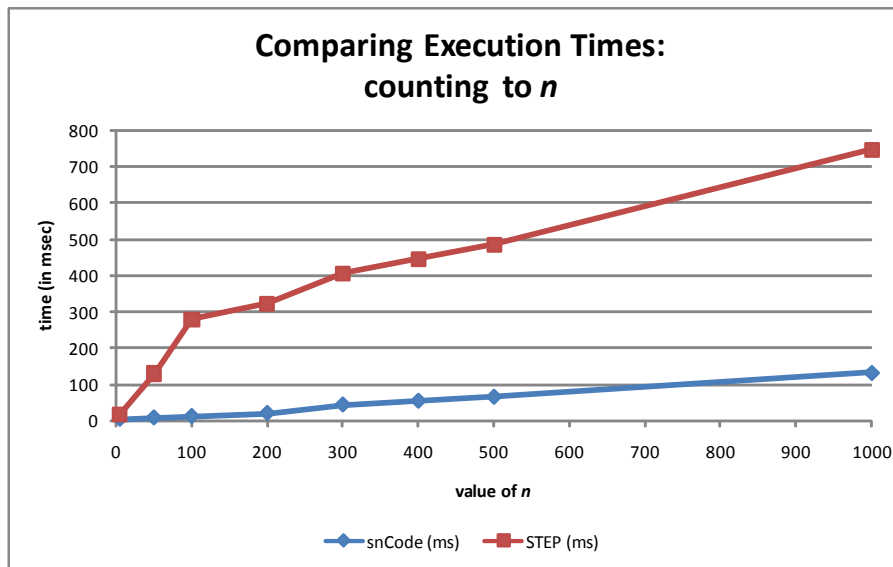


Figure 6.1: Comparing execution times to evaluate snCode versus STEP. The common source program is a while_trigger counting up to n . Table 6.2 contains the precise values illustrated here.

$n =$	snCode (ms)	STEP (ms)
5	5.12	17.65
50	8.82	130.47
100	12.813	279
200	21.27	322.17
300	45.07	405.66
400	55.37	444.1
500	66.54	485.6
1000	133.07	746.25

Table 6.2: Performance measurements for Figure 6.1.

The next two figures show the peak memory pool usage and garbage collection overhead of the STEPvm versus the traditional SXE STEP evaluator for this same counting application. Again, we are using the same Opcode implementations and “sn-Typed” data

objects. Despite this, we observe a significant memory usage decrease in the STEP_{vm} evaluator. In these scenarios we have chosen to limit the upper bound of heap memory to the realistic (indeed, default) JVM client setting of two megabytes (allowing for one megabyte for use by the Eden or new allocation memory pool). In Figure 6.2, we see that the snCode evaluator is *significantly* slower to reach the maximum allocation than the STEP Evaluator. The cost of reaching the maximum allocation bound is paid in execution time, where (as reflected in Figure 6.3) the STEP Evaluator suffers through significantly more garbage collector invocations.

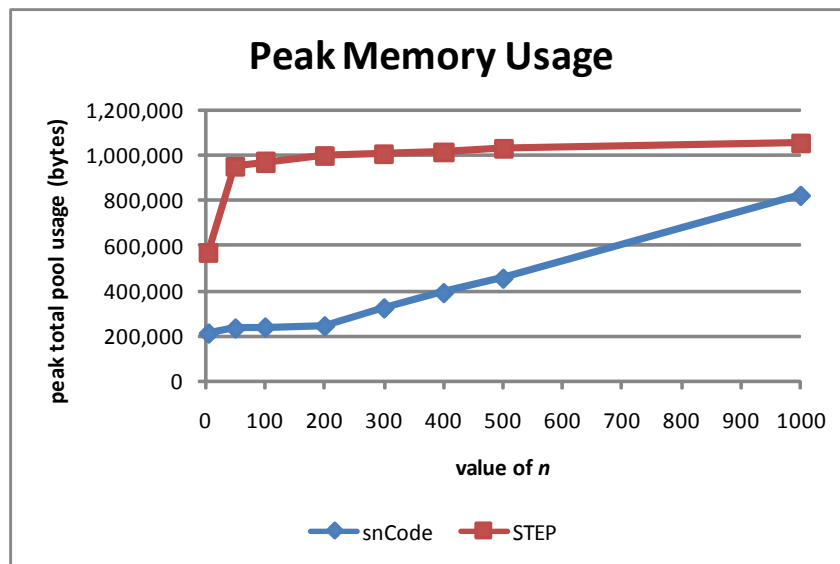


Figure 6.2: Comparing peak memory pool usage between the evaluation of snCode versus that of STEP. The common source program is a while_trigger counting up to n.

6.6 Conclusions and Future Work

Initially we considered trivially compiling STEP into an intermediate representation of either C or Java and then a second compiler pass to produce native executable code or Java bytecode. Other functional languages have been compiled into C as an intermediate target, *e.g.* [JHH⁺93]. Trivially compiling the functional-style STEP language into a procedural representation is achieved by traversing the STEP graph to produce a function declaration

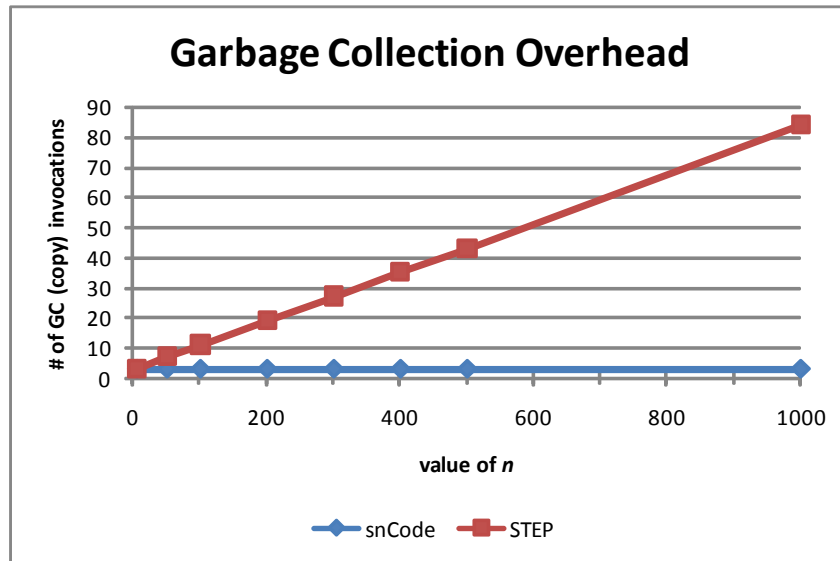


Figure 6.3: Comparing the number of times the Copy Garbage Collector is invoked between the evaluation of snCode versus STEP. The common source program is a while_trigger counting up to n .

for each node (named by its corresponding node’s unique ID) and inserting a call to the root node’s function in the `main()` of the program. The separate Java or C compiler unravels the functional execution flow at run time, weighing on the call stack. While creating separate functions for each node would yield more detail from profiling tools that support stack trace inspection, we would rather unravel a STEP graph without resorting to excessive function definitions, incur penalties on the call stack which are not strictly necessary, or rely on a third party compiler which removes the granularity of control we desire over the program’s ultimate execution.

In fact, translating demand and value propagation into function calls required a fair amount of uninteresting book-keeping; *e.g.*, ensuring that function names are unique (guaranteed by the STEP node ID), that function names are valid and do not clash with any reserved words in the target language. Intermediate function definition also required complete type annotation and although STEP is strongly typed, it lacks complete type annotation in its serialized XML form. Thus we wrote a type inference module for STEP itself that, given STEP’s restrictions, uses an ad-hoc type inference/checking algorithm that runs in

at most three passes over the STEP graph.

In STEP interpretation, a fair amount of node state is maintained to support proper evaluation of the graph (*e.g.*, checking whether a node is wanted by its parents, has fresh input, has not evaluated) and in converting the program into a procedural execution context this is unnecessary – only those nodes whose values are needed will be called in the body of the `main()` method. However some of the state information that was previously stored within nodes needed to be moved into the compiled program’s global state to maintain the semantics of the particular node types (not entirely unlike the shared memory environment we employ in the STEPvm). This intermediate translation from STEP to C, although ultimately successful, served only to motivate us to proceed toward the interpreted approach described herein.

Our ultimate goals for this work require that we migrate the current implementation to run in either a Real-Time Java Virtual Machine to gain greater control over the VM’s overhead, scheduling and garbage collection policies. Additionally we may consider a lighter-weight representation of `snObjects` to reduce the memory overhead of the STEPvm. If we do so, however, we run the risk of breaking compatibility with the existing STEP Opcode library. It is unclear, at this time, if the expected performance benefit would out-weight the cost of maintaining multiple Opcode implementations. Regardless of whether the STEPvm Opcode library has the same specification and underlying implementation as its heavier-weight sibling (the SXE), there are several specific Opcodes we intend to develop for this environment. Specifically, we have designed a series of Opcodes that facilitate mobility and remote way-point tasking for a robotics platform with self locomotion (we discuss this further in Chapter 9).

Chapter 7

Case Study: Combined Physical and Wireless Network Security

Wireless Intrusion Detection Systems (WIDS) monitor 802.11 wireless frames (Layer-2) in an attempt to detect misuse. What distinguishes a WIDS from a traditional Network IDS is the ability to utilize the broadcast nature of the medium to reconstruct the physical location of the offending party, as opposed to its possibly spoofed (MAC addresses) identity in cyber space. Traditional Wireless Network Security Systems tools are still heavily anchored in the digital plane of “cyber space” and hence cannot be used reliably or effectively to derive the *physical identity* of an intruder in order to prevent further malicious wireless broadcasts, for example by escorting an intruder off the premises based on physical evidence.

In this chapter, we show that Embedded Sensor Networks can be used effectively to bridge the gap between digital and physical security planes, and thus be leveraged to provide reciprocal benefit to surveillance and security tasks on both planes. Toward that end, we present our recent experience integrating wireless networking security services into the snBENCH. The snBENCH provides a high-level programmatic interface to the resources of a Sensor Network (SN) and thus the inclusion of wireless network sensors enables intrusion detection and response services to be written quickly and easily. The snBENCH has been designed with extensibility and modularity as a central tenet and therefore the changes

required to include these new sensing modalities is quite low. Moreover, the framework's modular nature allows a user to swap in any improved emergent wireless surveillance tool or technology (be it algorithmic or a physical turn-key device) with nominal effort and such changes would be transparent to their dependent services. We submit that our programmable, adaptable SN framework is the ideal foundation on which to compose Wireless Network Security services and physical security services alike, providing reciprocal benefit to each. The example programs given provide some insight into the highly customized, cross-modal Wireless Security behaviors that are possible in this context.

7.1 Motivation

Wireless Network Security is a non-trivial problem and as such a variety of Wireless Intrusion Detection Systems (WIDS) have been created. WIDS deploy wireless probes/sensors to passively or actively monitor the MAC frames transmitted on the wireless medium and identify misuse by observing either suspicious characteristics of individual frames (*e.g.*, exhibiting characteristics imprinted by standard hacking tools) or a particular pattern in a sequence of frames (*e.g.*, sequences in violation of protocol standards). Wireless misuse includes illegitimate users attempting to gain access to the network (intrusion), man-in-the-middle attacks (*e.g.*, luring legitimate users into communication with a rogue access point), and various Denial of Service (DoS) attacks [BS03] (*e.g.* spoofing a legitimate wireless Access Point (AP) and sending a disauthenticate beacon to legitimate users).

While it is generally advantageous to secure a network at the lowest layer possible, the appropriate layer for incorporating security functionalities is highly dependent on the nature of the threat and whether such threat could be dealt with (*i.e.*, identified and managed) at that layer. For example, the threat of wireless intrusion is often dealt with using Layer-3 mechanisms (*e.g.*, content based packet filtering, IP address isolation), essentially ignoring the option of Layer-2 detection and prevention. Layer-3 IDSs are likely popular because there is far more data available at Layer-3, making it straightforward to respond to attacks,

and because detection and response at Layer-3 is independent of the Layer-2 connection medium. On the other hand, wireless Denial of Service (DoS) attacks are much more difficult to deal with at Layer-3 as these attacks occur at Layer-2 and, even if detected, response is limited given that attackers will likely utilize fictitious or spoofed MAC addresses and may not have an IP address to retaliate against. Ultimately the only way to respond to these types of attack is to utilize information derived from the wireless medium (*e.g.*, received signal strength) to reconstruct physical location toward the goal of preventing further wireless transmissions from that user [Far05].

Wireless Intrusion Detection Systems provide mechanisms to identify, detect and locate DoS attacks, yet these systems are generally limited to logging or email alert response mechanisms. Many works ultimately recommend dispatching administration personnel to further analyze and respond to a detected attack – a costly and impractical solution in most situations. Instead, once the physical area of an attack has been derived it is possible to utilize automated responses from a variety of actuation hardware, if available; *e.g.*, embedded pan-tilt-zoom video cameras to gather an image, wireless detectors on pan-tilt motors to pin-point a signal, programmable robots to triangulate signal, a common message display (virtual bulletin board) in the environment informing users why their service has been interrupted and who is responsible. Additionally, there would be a clear benefit from including other, non-network centric *inputs* to the Wireless Network Security System (*e.g.*, driving a MAC whitelist from Bluetooth/RFID tracking, security camera images, passcard logs).

To achieve such cross-modal interaction within the context of a Network Intrusion Detection tool would require the generation of extensive, package and deployment specific software (modules, scripts, *etc*) that are, by their very nature, cumbersome to maintain. Indeed, such an approach is wrong headed. We observe that Wireless Network Security Services are specific, narrowly focused instantiations of an Embedded Sensor Network wherein sensory data includes the output of such monitoring tools. Rather than “hack” a Wireless Security System to include Sensor Network functionality, we advocate the inclusion of

Wireless Security within a Sensor Network. Thinking differently about Network Security, the integration of new sensory data (*e.g.*, motion detection, face detection) and actuation responses expand Network Security beyond the digital plane and into the physical plane.

In this paper we detail our work to include wireless network monitoring devices as sensors in our Sensor Network infrastructure, the SNBENCH (Sensor Network Workbench). The SNBENCH provides a high-level programmatic interface to the resources of a Sensor Network (SN) and thus the inclusion of wireless network sensors enables intrusion detection and response services to be written quickly and easily. The SNBENCH has been designed with extensibility and modularity as a central tenet and therefore the changes required to include these new sensing modalities are quite modest. Moreover, the framework's modular nature allows a user to swap in any improved emergent wireless surveillance tool or technology (be it algorithmic or a physical turn-key device) with nominal effort and such changes would be transparent to their dependent services. We submit that our programmable, adaptable SN framework is the ideal foundation on which to compose Wireless Network Security services and physical security services alike, providing reciprocal benefit to each. The example programs given provide some insight into the highly customized, cross-modal Wireless Security behaviors that are possible in this context.

7.2 Related Work

While many Network Intrusion Detection (Security) Systems exist (both commercial and open-source), we are presently unaware of any other work that leverages a programmable Sensor Network framework toward joint physical and Wireless Network Security, and thus believe we are unique in this regard. We present works that are related in three major thrusts; We distinguish between works that provide detection on a single wireless source (probe) as *Wireless Intrusion Detectors* (WIDs), those works that detect events across multiple detectors simultaneously as *Intrusion Detection Systems* (IDSs) and finally those that determine attack location as *Wireless Intrusion Detection Systems* (WIDSs). Although

WIDSs contain a WID component, these works are not necessarily proper subsets of each other, as IDSs may not provide wireless detection.

Wireless Intrusion Detection

Kismet [Kerb] is the *de facto* open-source Layer-2 Wireless Intrusion Detector. Kismet passively scans 802.11 channels for activity and can be used for a variety of uses including finding hidden access points, mapping access points in a geographic region via GPS, or generating alert events when suspicious frames are detected. A Kismet deployment may consist of three distinct components, (1) a light-weight Kismet Drone that passively captures the wireless frames from its local interface and sends them to (2) a Kismet Server that processes the frames from one or more drones to detect either fingerprint or trend based suspicious activity and (3) an optional remote Kismet client that connects to the Server to receive notifications and render the results. The Kismet server may drive external wireless event notification by providing custom clients that communicate using the published Kismet protocol. Kismet may be configured as an Intrusion Detection System by associating several drones with a single server process to build a single, central wireless event log file. Kismet is not considered a Wireless Intrusion Detection System by our definition however, as the Kismet server does not indicate which physical drone is responsible for an alert which prevents spatial intrusion tracking. Kismet-newcore, a re-write of the Kismet project, does preserve which drone generated a wireless alert event yet lacks a stable build at this time. Other tools have existed in the WID space prior to Kismet (*e.g.*, WIDZ [lfb]) but have been largely unmaintained in recent years. Similarly, AirIDS [Lyn] set out to be the first open-source Intrusion Detection System aimed at 802.11 attacks, however the project never reached a stable release, is no longer available for download, and appears to have been abandoned.

Intrusion Detection Systems

While Kismet is the *de facto* Layer-2 WID and IDS, Snort [Roe99] is the *de facto* standard IDS for Layer-3 (IP traffic analysis). Snort is a mature IDS with a large user base and comprehensive set of detection rules for detecting malicious content in IP packets for a wide range of attacks. Snort also offers very basic response mechanisms (*e.g.*, logging or email alert mechanisms) however the Barnyard project not only aims to increase performance of logging output but also claims to enable the creation and use of custom output plug-ins. As Snort is aimed at Layer-3, it offers no support for wireless monitoring, however the Snort-wireless [Loc] project adapts the Snort rule engine for Layer-2 wireless use by adding wireless frame capture and replacing IP addresses with MAC addresses in rule processing. Unfortunately the Snort-wireless project has not been updated since late 2005 and plans for integrating Snort-wireless into Snort appear to have been abandoned.

In many ways, our vision is similar to that of modular IDSs (*e.g.*, [Yoa], [VVK03]). These modular (or so-called “Hybrid”) systems are designed to allow various Intrusion Detection Software packages to be integrated as “Sensors” in the IDS. These works share the modular approach which is similar in spirit to the cross tool integration that we hope to provide to the Network Security community, yet these works are narrowly focused on issues of traditional Network Security. Our work enables the composition of sense and respond programs that manipulate both network data and physical sensory data (*e.g.*, image processing on embedded video cameras) in a manner that would be impossible on these IDS platforms without significant changes.

Wireless Intrusion Detection Systems

Tracking MAC addresses alone is insufficient for a WIDS as they may be easily spoofed by malicious tools [BS03]. The requirement that a WIDS must determine attack location is sensible, considering that Layer-2 DoS attack response generally requires physical intervention [Far05].

Nominally we expect we know the Cell of Origin (COO) of a detected wireless transmission (*i.e.*, the user's distance must be within the detection range of the physical location of the detection point), however while this might be useful for short range media (*e.g.*, Bluetooth, RFID) we'd like to obtain higher accuracy than the range of 802.11 (ranging from 200 to 25meters, depending on the physical layout). Many approaches to derive location from Signal Strength Information (SSI) of RF transmissions have been undertaken, including Microsoft Research's RADAR [BP00] which uses readings from multiple sensors to perform on-line triangulation, compared against off-line training data. Many works since have tried to loosen the off-line training needs of this work attempting to dynamically overcome issues of transmission reflection, diffraction and interference (*e.g.*, [YAS03]). Work presented in [TRLW03] frames these goals directly in terms of reconstructing wireless entity location particularly for security purposes and a similarly goaled architecture is described in [LRG03].

The work in [AAJI04] offers an architecture for a Wireless Intrusion Detection System that breaks from the norm slightly, in so far as it establishes wireless sensors that form a perimeter around an access point (or area) to be protected and uses directional antennae (opposed to the typical, omni-directional antenna found on WAPs) that would sweep the region to better pin-point the location of a particular wireless user. This work provides detailed analysis of particular directional antennas and is able to pinpoint wireless intruders accurately. Our work is compatible with the use of sweeping directional antenna; and perhaps more so than what is envisioned in [AAJI04] as the SNBENCH can address and direct the servos that control antenna movement explicitly, enabling on-demand target tracking. In particular, we envision either making the directionality of the wireless sensors explicit to be controlled within the service logic, or mounting the wireless antennae to the pan-tilt-zoom camera network that sweeps the perimeter of our SN testbed. Additionally we note that requiring a perimeter defense may be useful in some installations, but perhaps impractical in most. The ability to use modalities other than directional antennae for identification (*e.g.*, the use of cameras) extends the applicability of our approach.

In the commercial spectrum, the Wireless Intrusion Detection System AirDefense Enterprise [Air] integrates the industry standard signal strength positioning system, Ekahau [Eka] to provide location tracking of wireless intruders¹. IBM's Internet Security Systems' Wireless Products [IBM] are also popular, but lack location tracking. Both tools attempt to provide complete, turn-key detection and response systems for corporate wireless networks. Given their single-solution nature, these tools do not provide integration with third-party Intrusion Detectors or tools. Responses to wireless attack detection in these systems are more pro-active, for example, disauthenticating a malicious user to effectively kick him or her off the network, yet they do not offer an accessible programming interface to adjust the sense and respond behavior. While these solutions represent the upper echelon of commercial sense and respond WIDSs, their lack of extensibility makes cross-modal monitoring solutions (*e.g.*, utilizing video frames) unattainable.

7.3 Enabling Wireless Monitoring

SNBENCH is extensible by design insofar as support for new sensing devices may be added to the Sensor eXecution Environment (SXE) by providing implementations of two relatively small interfaces; a *SensorHandler* translates SNBENCH requests to interact with a specific device and a *SensorDetector* module must provide facility to detect new devices of this type and inspect their state. The *SensorHandler* is akin to a device driver, abstracting away the specific idiosyncrasies of the particular device's interface and enabling the device to be accessed by higher-level programming constructs. As far as the SNBENCH framework is concerned the abstracted device becomes just another managed input device/event generator only different from a video camera or motion sensor insofar as the datatype of its output.

To enable wireless network security service composition on SNBENCH, we have added two new sensors and a new actuator; The *WifiAlertSensor* reports wireless alert detection events,

¹Popular turn-key hardware solutions to wireless device tracking also exist, *e.g.*, Cisco's 2700 Series Wireless Location Appliance [Sys06].

the `WifiActivitySensor` reports MAC addresses and Received Signal Strength Indication (RSSI) for any passively observed wireless activity, and the `WifiResponder` actuator sends a disauthenticate flood to a particular MAC address. Rather than implement wireless Layer-2 tools from scratch we opted to leverage several existing open-source software packages.

WifiAlertSensor

The `WifiAlertSensor` is a `SensorHandler` implementation that leverages the Kismet [Kerb] wireless intrusion detector via a self-contained customized Kismet client. The Java based `WifiAlertSensor` class is hosted by a “non-lightweight” SXE and translates the proprietary Kismet client-server protocol into structured, typed `SNBENCH` objects (tagged XML) that encapsulate notifications from the Kismet server. The decision to use Kismet stems from its passive scanning ability, wide range of hardware support, and modular design (described in Section 7.2). While the decision to use this package in particular may be debated, the inclusion of any another functionally-equivalent Wireless Intrusion Detector would be equally straightforward.

A Kismet client may request to receive several types of Kismet messages from a Kismet server/drone pair (client traffic, AP detection, suspicious activity alerts, *etc.*). In the case of the alert sensor, we request notification of all wireless alerts supported by the current stable build of Kismet (detailed in Table 7.1). Whenever the Kismet server detects an Alert condition from its corresponding drone’s data feed, an alert is sent to the `WifiAlertSensor` client which translates and buffers the alert message. In addition to translating the Kismet protocol, the `WifiAlertSensor` adds additional fields to alert message; a local timestamp to measure buffer service delay, a sensor source to identify the physical sensor (drone) that produced the message, and a severity field that indicates the relative threat of the particular attack (this corresponds to the values we have specified as the danger column in Table 7.1).

Name	Danger	Type	Description
PROBENOJOIN	none	Trend	A user is probing periodically without joining any AP Windows XP clients cause this alert as normal behavior.
LUCENTTEST	none	Fingerprint	A site survey package is in use
NETSTUMBLER	low	Fingerprint	An attempt to discover the SSID of a hidden AP
WELLENREITER	low	Fingerprint	Dictionary based attempt to discover hidden AP's SSID
DISASSOCTRAFFIC	medium	Trend	A user is transmitting data shortly after being disassociated (likely an indication that this user is victim of an attack)
BSSTIMESTAMP	medium	Trend	An AP's timestamps are out of sequence and thus may indicate a spoofing attempt
AIRJACKSSID	high	Fingerprint	An AP is broadcasting an SSID that is the default of the hijacking too Air-Jack
DEAUTHFLOOD	high	Trend	Disassociate or deauthenticate are being repeatedly sent (flooded) from a non-AP node
CHANCHANGE	high	Trend	An AP is now advertising a different channel than previously detected (likely man-in-the-middle attack)
BCASTDISCON	high	Fingerprint	A disassociate or deauthenticate message has been broadcast. May be used to disclose a hidden SSID, perform a man-in-the-middle attack, or denial service
NOPROBERESP	high	Fingerprint	A response to a probe containing a 0 length SSID, which is an attempt to exploit a bug in some AP firmware
MSFBCOMSSID MSFDLINKRATE MSFNETGEARBEACON	high	Fingerprint	A packet crafted to exploit a particular Windows driver fault that allows arbitrary code injection
DISCONCODEINVALID DEAUTHCODEINVALID LONGSSID	high	Fingerprint	A packet crafted to exploit a driver or AP firmware fault that might allow arbitrary code injection

Table 7.1: Alert events detected by Kismet and reported to SNBENCH

The WifiAlertSensor's message buffer is configurable in length (where length is measured in either size or time) and alert messages are retrieved from the buffer by Opcodes requesting data from this sensor. Implementation of the retrieval Opcode may impose a blocking or nonblocking semantic, as needed. In our experimentation we implemented a single Alert-centric Opcode, `sxe.core.wifi.get`, that performs a non-blocking read from the alert sensor's buffer to populate and return a WifiAlert. The WifiAlert data-type is a subtype of `snStruct`, with tagged fields corresponding to the fields populated by the WifiAlertSensor and thus accessing the data within a WifiAlert reuses the existing `snStruct` manipulation opcodes. A Service Developer retrieves WifiAlerts via the high-level function

`DetectWifiAlert()` that is compiled into a call to the Opcode `sxe.core.wifi.get` with a `WifiAlertSensor` (or set of sensors) as a parameter. Complete examples of high-level service logic are given in Section 7.5.

WifiActivitySensor

The Activity sensor provides data regarding wireless transmissions that have been detected by a passive, promiscuous-mode wireless sensor. In particular we are interested in the MAC address of a transmission, the observed signal strength (RSSI) and the mode of the transmission (*i.e.*, Access Points, Clients, Ad-hoc participants). While determining physical location from RSSI is imperfect (as RSSI readings themselves may not be entirely accurate depending on the driver implementation and other physical factors) the use of RSSI readings can better pinpoint a MAC address' physical location than the simple cell-of-origin data alone. `WifiActivitySensor` maintains a hash-table of the detected wireless activity (keyed by MAC address), which can be used to either report new/updated wireless activity similar to the Alert sensor or to query the activity log to find information about a particular MAC address. Like the alert sensor the activity sensor also communicates with a remote sensor “server” process responsible for gathering data.

In practice we actually support two different physical implementations for the activity sensor server as the Kismet drone/Server does not retrieve RSSI on our preferred target platform. For Kismet RSSI supported hardware we use a client derived from the `WifiAlertSensor` implementation that requests and parses `NETWORK` and `CLIENT` messages from the Kismet server rather than `ALERT` messages. For the OpenWRT platform, we use custom code that sends `ioctl`'s to the wireless device to put the device in passive, monitor mode, accept frames, and retrieve data from the frames and device (including the RSSI). Our program is based on code from the open source `WiViz` [Tru] package for OpenWRT, which contains the `ioctl` codes needed to achieve the proper device state and interaction. Like the Kismet server, this program provides notifications of activity messages which are received and hashed by the `WifiActivitySensor`.

The high-level Opcode `DetectWifiActivity()` is compiled into `sxe.core.wifi.get` with a `WifiActivitySensor` as a parameter and blocks until a new activity message is available from that sensor. Additionally `QueryWifiActivity()` (compiled into `sxe.core.wifi.find`) searches the `WifiActivitySensor`'s hash table for the latest reading associated with the specified MAC address. As with the `WifiAlertSensor`, returned data is structured as a `snStruct` derivative.

WifiResponder

In addition to the wireless network sensing described above, we have also implemented a Layer-2 wireless actuator (*i.e.*, output device) `WifiResponder` that may be used as a retaliatory action against a detected attacker. The `WifiResponder` invokes a script on a trusted (whitelisted) device running Linux with a compatible 802.11 interface and the `airreplay-ng` [Dev] tool. The Opcode `APDeauth()` takes as arguments a `WifiResponder` that will send a flood of deauthenticate messages to a particular MAC address (the second argument) from a particular MAC address (the third argument).² An actuator is nearly identical to a Sensor in its implementation within the `SNBENCH`. The Handler for `WifiResponder` invokes the remote common gateway interface (CGI) script to initiate the deauthenticate “attack” against the specified host.

7.4 Deployment Environment

Our test-bed deployment contains several Linksys WRT54GL Wireless Access Points (APs), running the OpenWRT [The] Linux platform. On top of OpenWRT, the `kismet-drone`, `airreplay-ng`, and `signal strength monitor` packages are installed. The APs are configured to use their wireless interface in promiscuous client mode. To communicate the results of their wireless monitoring, the APs are tethered to our Gigabit research LAN by their on board 100Mbit Ethernet port.

²Readers may readily note that this opcode is a loaded weapon and may gasp or recoil in horror. In fact, this is not the first Opcode that requires special user privileges to ensure correct use.

To support the WifiAlertSensor, we run a Kismet drone process on each of the access points while the Kismet Server process runs on the same host as the SXE. Although the Kismet Server process could also be run directly on the AP, the RAM and CPU limitations of these devices lead to a less responsive system than if the Kismet Server process is hosted on a separate host. As the Kismet Server does not presently distinguish the results from different Kismet Drones [Kera], we run one Kismet server per drone and each WifiAlertSensor connects to a unique Kismet Server process thus allowing snBENCH to distinguish which drone generated a wireless event. Running one Kismet Server per Drone also carries the advantage of minimizing the impact of a Kismet server process hanging, or failing to process updates from its drones (admittedly a fairly uncommon occurrence).

In our tests of the WifiAlertSensor we were able to simulate and detect all relevant attacks detected by Kismet (Table 7.1) and were unable to measure any significant induced delay on event detection in the snBENCH infrastructure. Analysis confirmed the expectation that the amount of time a single Kismet message spent in the Sensor buffer was directly related to the computation load on the SXE host and the alert generation rate. In general the observed buffer service delay oscillated between 0 and 15ms per alert under moderate load with unrealistically high message flooding arrival rates (in practice, Kismet can and will throttle alert notification rates, however this was disabled for our performance tests). Under heavy load conditions with alert message flooding we experienced queuing delay as long as 300ms. This gives us a good indication as to the maximum acceptable workload for an individual SXE before it is no longer a viable host for wireless sensing tasks. Ultimately we believe that any response detection under 1 second is reasonable as it is unlikely that the attacker would, say, flee the premises (or video frame) within that amount of time.

7.5 Wireless Security Services

Simple Detection

An example SNAFU program that provides simple logging is given in Program 7.1. Its representation in STEP is given in the appendix. A `level_trigger` is used to assign an event handler to the detection of a high severity wireless alert. The `storage.append` Opcode modifies a named storage entity (*i.e.*, table) specified by the first argument by inserting a data object and its corresponding unique key. The storage table is keyed by timestamp and includes entries for each detected violation containing the recorded MAC address, the sensor from which the alert was detected, and the type of alert. Unlike the logging provided by Kismet as an IDS, this service records which sensor has detected the event and is backed by an SQL server. The logged data is available programmatically via storage access Opcodes or direct SQL queries, or through a standard web browser via the SXE host's web service that performs XSL translations to render the local data storage.

Listing 7.1: Add an entry to a central log when a wireless alert is detected.

```
let_once ALERT = DetectWifiAlert(sensor(WifiAlert,ALL)) in
  level_trigger(
    equals(ALERT."SEVERITY","HIGH"),
    storageappend("ALERTLOG",concat(ALERT."TIMESTAMP",ALERT."SOURCE"),
      ALERT)
  )
```

This sample SNAFU program could easily be extended to establish a log of all observed wireless activity (not just attacks) by adjusting the predicate of the trigger from `DetectWifiAlert` to `DetectWifiActivity` and removing the severity check. Another simple example service logic is given in Program 7.2; This program will automatically email an administrator when a specific wireless attack is detected.

Attack Response

The previous examples are essentially the status quo for a response to the detection of a breach in a Wireless Network – an entry into a log file or an email alert. The advantage of employing the SNBENCH in the wireless security domain is the wider range of responses possible. Nominally, the email operation in Program 7.2 could be replaced with any number of response mechanisms including sending an explicit deauthorization to the detected MAC address³ using the WifiResponder and APDeauth opcode described in Section 7.3. Instead we explore the unique cross section of the network plane (*e.g.*, wireless data frames) with the physical plane (*e.g.*, signal strength and signal loss of signal over distances). For example, an embedded, cross-modal Sensor Network such as the Sensorium can utilize both wireless network sensors (*i.e.* network plane sensors) and a pan-tilt-zoom video camera network (*i.e.* physical plane sensors) to catch an image of the attacker “in the act.”

Listing 7.2: E-mail an administrator whenever a specific wireless alert is detected.

```
let_once ALERT = DetectWifiAlert(sensor(WifiAlert,ALL)) in
  level_trigger(
    equals(ALERT."TYPE", "DEAUTHFLOOD"),
    email("MOCEAN@CS.BU.EDU",
      concat("$NOW$",
        " :_DEAUTH_FLOOD_DETECTED_FROM_MAC_", ALERT."MAC",
        "_AT_TIME_", ALERT."TIMESTAMP",
        "_BY_SENSOR_", ALERT."SOURCE"
      )
    )
  )
)
```

Logging Physical Evidence

Any user detected engaged in wireless network intrusion is clearly within a bounded distance from the detecting sensor. This coarse, cell of origin based physical location of wireless users

³A MAC address is far from the best way to uniquely identify an attacker as the attacker will likely use a fictitious MAC address or worse, clone a legitimate user’s MAC during an attack.

is available, imprinted in all wireless data returned from the WifiSensors (determined by which sensor has detected the user). A very simple wireless cell of origin location example is specified in Program 7.3. The program's content is very similar to the previous examples and introduces some Pan-Tilt-Zoom sensor (PTZCamera) specific opcodes, the function of which should be clear from context. This sample streams images of a region where an attack has been detected. The location logic is explicit in the service logic, selecting an image from the camera that best covers the physical space within the signal coverage of the relevant Wifi sensor, using "hard coded" location logic that is specific to the sensor configuration of the particular deployment. The `case` expression is used for readability as syntactic sugar (shorthand) for nested conditionals and takes the same syntax as in StandardML. Connecting this program fragment to either of the previous examples would enable the logging or emailing of images that correspond to the attack location.

Listing 7.3: Point a PTZ camera and return an image when a wireless alert is detected.

```
def BestPTZForViewOf(alert) =
  case APName(alert.SOURCE) of
    ``CS Grad Lab West`` => List(45,0,0,sensor(PTZCamera,"PTZ1")),
  | ``CS Grad Lab East`` => List(15,0,0,sensor(PTZCamera,"PTZ1")),
  | ``CS Grad Lab Lounge`` => List(0,0,0,sensor(PTZCamera,"PTZ3")),
  | ``CS UGrad Lab`` => List(0,0,0,sensor(PTZCamera,"PTZ4"))

let_each ALERTSENSORS = sensor(WifiAlert,"ALL") in
  let_once ALERT = DetectWifiAlert(ALERTSENSORS) in
    level_trigger( not(isNull(ALERT)),
      PTZSnapshot(BestPTZForViewOf(ALERT))
    )
```

Alternatively user location reconstruction could be implemented within an Opcode, resulting in intrusion detection service logic that is agnostic to the particular location resolution mechanism used. Such an approach makes sense if the deployment environment already contains a wireless location infrastructure (*e.g.*, [Sys06], [Eka], all knowing ora-

cle) that could be accessed from within an Opcode call. An example of this approach is given in Program 7.4. `WifiLocateMac` encapsulates the physical location of MAC addresses and (`PTZLocate`) determines the best PTZ Camera (and corresponding angle) to capture an image of that location. The implementation of `WifiLocateMac` is functionally similar to `BestPTZForViewOf` in the example in Program 7.3, yet uses a received signal strength from multiple sensors to estimate the target's location between the sensors.

Recall the `SNBENCH` not only eases the composition of such alert services, it also eases deployment by automated re-use of existing computation/deployments to improve resource utility. All the examples given thus far share the same predicate logic and could share a single instantiation of that portion of the logic.

Listing 7.4: Functionally equivalent to Program 7.3, but uses black-box opcodes.

```
let_each ACTSENSORS = sensor(WifiActivity, ALL) in
  let_each PTZSENSORS = sensor(ptz_image, ALL) in
    let_once ALERT = DetectWifiAlert(sensor(WifiAlert, ALL)) in
      level_trigger( not(isNull(ALERT)),
        PTZSnapshot(PTZLocate(QueryWifiAlert(ALERT."MAC", ACTSENSORS)),
          PTZSENSORS))
```

7.6 Future Work

As briefly touched on in the previous Section 7.5, computational complexity can either be placed explicitly in the service logic or pushed into the Opcodes on which the service relies. Naturally, there is a resource flexibility and performance trade-off in this decision. In the most degenerate case any program can be implemented as one giant Opcode that eliminates any performance penalties incurred by the STEP interpreter. Such an approach yields a complex Opcode implementation that eliminates `SNBENCH`'s inherent resource management benefits (*i.e.*, computation and resource sharing becomes unlikely as there will be few common subgraphs between STEP programs, individual opcodes cannot be split across multiple SXEs, and there may be few SXEs available to accommodate such a large computation).

On the other extreme a program composed entirely of very basic STEP Opcodes may pay a high overhead for the SXE interpreter, yet has maximum flexibility to be split across any compatible partitioning of the SXE space. Finding the optimal balance between STEP and Opcode complexity largely depends on the particular needs of the given service.

Advanced Location Reconstruction

Program 7.5 tries to find a balance between the extremes of the last two examples in the prior section. This program also produces an image whenever an attack is detected, but specifies much of the location logic within the STEP program while still leaving the significant computation to Opcode implementations (*e.g.*, `rssiToDist`, `PTZFovCover`).

Listing 7.5: Get an image from the camera with the best coverage of a wireless alert.

```
def BetterView(T,X,Y) =
  let L = Pair(locate(T.SOURCE),rssiToDist(T.RSSI))
      in cond(greater(PTZFovCover(X,L),PTZFovCover(Y,L)), X, Y)

let_once ALERT = DetectWifiAlert(sensor(WifiAlert,"ALL")) in
  let_once ALERTINFO = QueryWifiActivity(WifiActivity,ALERT.SOURCE) in
    level_trigger(
      not(or(isNull(ALERT),isNull(ALERTINFO))),
      PTZSnapshot(
        Fold(
          BetterView(ALERTINFO,-),
          sensor(PTZCamera,"ALL")
        )
      )
    )
  )
```

The function `Fold`, commonly referred to as `reduce` in some languages, is a higher-order function that applies a given function accumulatively across a given list. In this application, the function `BetterView` is applied across the list of all Pan-Tilt-Zoom sensors to determine which sensor has the best coverage of the area in which an attack was detected in order to

take a snapshot from that sensor. In compilation Fold is syntactic sugar that is expanded by the compiler via substitution; its implementation is an expression that uses a `WhileTrigger`, the `LAST_TRIGGER_EVAL` token, lists and pairs to provide iterative application over the list of elements.

Wireless Access Lists from Physical Data

The last example program (Program 7.6) also leverages the natural connection between wireless network security and physical site surveillance, however does so in the other direction, using information detected on the physical plane to (re-)configure the wireless network. An embedded camera network and face detection Opcodes are used to detect the identities of individuals entering or leaving a secured space as a trigger to enable the detected user's wireless MAC address for service in that physical area. Put simply, when we see Jane enter the lab we want to allow Jane's MAC address to be used in the lab (placed in the whitelist), and we want to remove her MAC address from the whitelist when she leaves the lab. The goal is to make it slightly more difficult for a malicious user to find an unused, authorized wireless MAC address to abuse for great lengths of time.

Listing 7.6: Enable only the MAC addresses of user's whose faces have been recognized.

```
let_once SNAP = snapshot(sensor(Camera, "LAB_DOOR_IN", "LAB_DOOR_OUT")) in
  let_once WLAN_MAC_ADDR = storage.lookup("MACMAP", FaceDetect(SNAP)) in
    level_trigger( not(isNull(WLAN_MAC_ADDR)),
      case SNAP.SOURCE of
        "LAB_DOOR_IN" => WifiWhitelist(WLAN_MAC_ADDR, sensor(AP, "CS_LAB
          *")),
        | "LAB_DOOR_OUT" => WifiBlacklist(WLAN_MAC_ADDR, sensor(AP, "CS_LAB
          *"))
      )
```

Modification of the WLAN's access control list in this example is performed by assuming the presence of a `WifiWhitelist`; This implementation would be straightforward on OpenWRT enabled Access Points, requiring a CGI script to modify the device's `maclist` con-

figuration file. One may easily imagine other sensors that may be used in tandem with face detection as the trigger predicate in this expression (*e.g.*, magnetic card or RFID reader).

snBench as a Turn-Key Network Security Solution

There is no reason to limit SNBENCH's Network Security to Wireless attacks. Other Intrusion Detection Tools and Network interfaces could easily be added to further improve the SNBENCH as a complete, cross-layer Network Intrusion Detection System. Integrating Layer-3 detection (*e.g.*, Snort) as a sensor would enable the detection of misuse from IP contents that could be used to drive isolation or removal responses at Layer-2. Additionally including port scanning and other fingerprinting tools would greatly increase confidence in user identification enabling more confident automated response.

In our vision of SNBENCH we view the Sensor Task Execution Plan (STEP) as a thin-waist language that may be a compilation target from other domain specific languages (*e.g.*, a Structured Query Language). Ideally we would like to move our own campus IT department to the use of SNBENCH over their current Network Security and Intrusion tools. As such we might consider the development of a STEP compiler for a declarative/rule-oriented language that is similar to existing network rule specification languages to ease the staff's transition to SNBENCH.

Related to this goal, our work on a lightweight SXE for embedded devices will be used to explore deployment of the Sensor eXecution Environment directly on OpenWRT enabled access points to provide SNBENCH as a turn-key solution for Wireless Network Security services.

7.7 Conclusions

Many excellent Wireless Intrusion Detection tools exist and they achieve their specialized goals well. These tools should not (and rarely attempt to) provide complete Network Security Systems as they are generally focused on detecting a particular kind of attack (denial-

of-service, intrusion, *etc*) at a particular scope (Layer-2 or Layer-3). A complete Network Security solution should integrate multiple tools to cover a superset of possible attacks at all possible layers and should bridge the divide between cyber and physical identities. In practice this allows NID tools to focus on what they do best (*i.e.*, detection) and yet be woven into a comprehensive Network Security solution. Network Security (specifically, wireless security) is not a problem that exists in a vacuum detached from the physical space in which the network is deployed. We promote an approach to unify physical site surveillance and network security security under the umbrella of SNBENCH — a general purpose sensing infrastructure we have developed. In that regard, we have demonstrated how SNBENCH enables the rapid development and deployment of cross-modal security services. We have shown that with SNBENCH (1) detection of wireless anomalies can be correlated with other sensory inputs providing reciprocal benefit to merging security on the physical and cyber planes, (2) detection and response services may be easily composed and modified without technical knowledge of the specific protocols or implementations of the underlying sensory tools, and (3) adding additional intrusion detection tools as input or other devices for response is straightforward given SNBENCH’s modular architecture. The illustrative example programs provided in this paper range from the status quo (simple logging and email alerts) to beyond (enabling MAC addresses based on face detection) in a hope to spark the reader’s imagination to more elaborate services and responses that are currently possible with the SNBENCH platform (*e.g.*, locking doors, turning on a siren, ...).

Listing 7.7: The STEP representation of SNAFU Program 7.1.

```
<?xml version="1.0"?>
<!-- generated with SNAFU compiler 0.8 -->
<stepgraph id="WIFLEXAMPLE_1">
  <level_trigger id="LEVEL_TRIGGER">
    <exp id="EXP_3" opcode="SXE.CORE.EQUALS">
      <exp id="STRUCT.GET" opcode="SXE.CORE.STRUCT.GET">
        <exp id="ALERT" opcode="SXE.CORE.WIFI.GET">
          <sensor id="WIFIALERT-ALL">
            <device id="ALL" type="WIFIALERT"/>
          </sensor>
        </exp>
      </exp>
    </level_trigger>
  </stepgraph>
```

```

    </exp>
    <value id="VALUE">
      <sobject type="SNBENCH/STRING"><![CDATA["SEVERITY" ]]></sobject>
    </value>
  </exp>
  <value id="VALUE_7">
    <sobject type="SNBENCH/STRING"><![CDATA["HIGH" ]]></sobject>
  </value>
</exp>
<exp id="EXP_6" opcode="SXE.CORE.STORAGE.APPEND">
  <value id="VALUE_1">
    <sobject type="SNBENCH/STRING"><![CDATA["ALERTLOG" ]]></sobject>
  </value>
  <exp id="EXP_1" opcode="SXE.CORE.STRING.CONCAT">
    <exp id="STRUCT.GET_1" opcode="SXE.CORE.STRUCT.GET">
      <const id="CONST_3">
        <splice id="SPLICE_3" target="ALERT" />
      </const>
      <value id="VALUE_12">
        <sobject type="SNBENCH/STRING"><![CDATA["TIMESTAMP" ]]></sobject>
      </value>
    </exp>
    <exp id="STRUCT.GET_0" opcode="SXE.CORE.STRUCT.GET">
      <const id="CONST_0">
        <splice id="SPLICE_6" target="ALERT" />
      </const>
      <value id="VALUE_8">
        <sobject type="SNBENCH/STRING"><![CDATA["SOURCE" ]]></sobject>
      </value>
    </exp>
  </exp>
  <const id="CONST_7">
    <splice id="SPLICE_2" target="ALERT" />
  </const>
</exp>
</level_trigger>
</stepgraph>

```

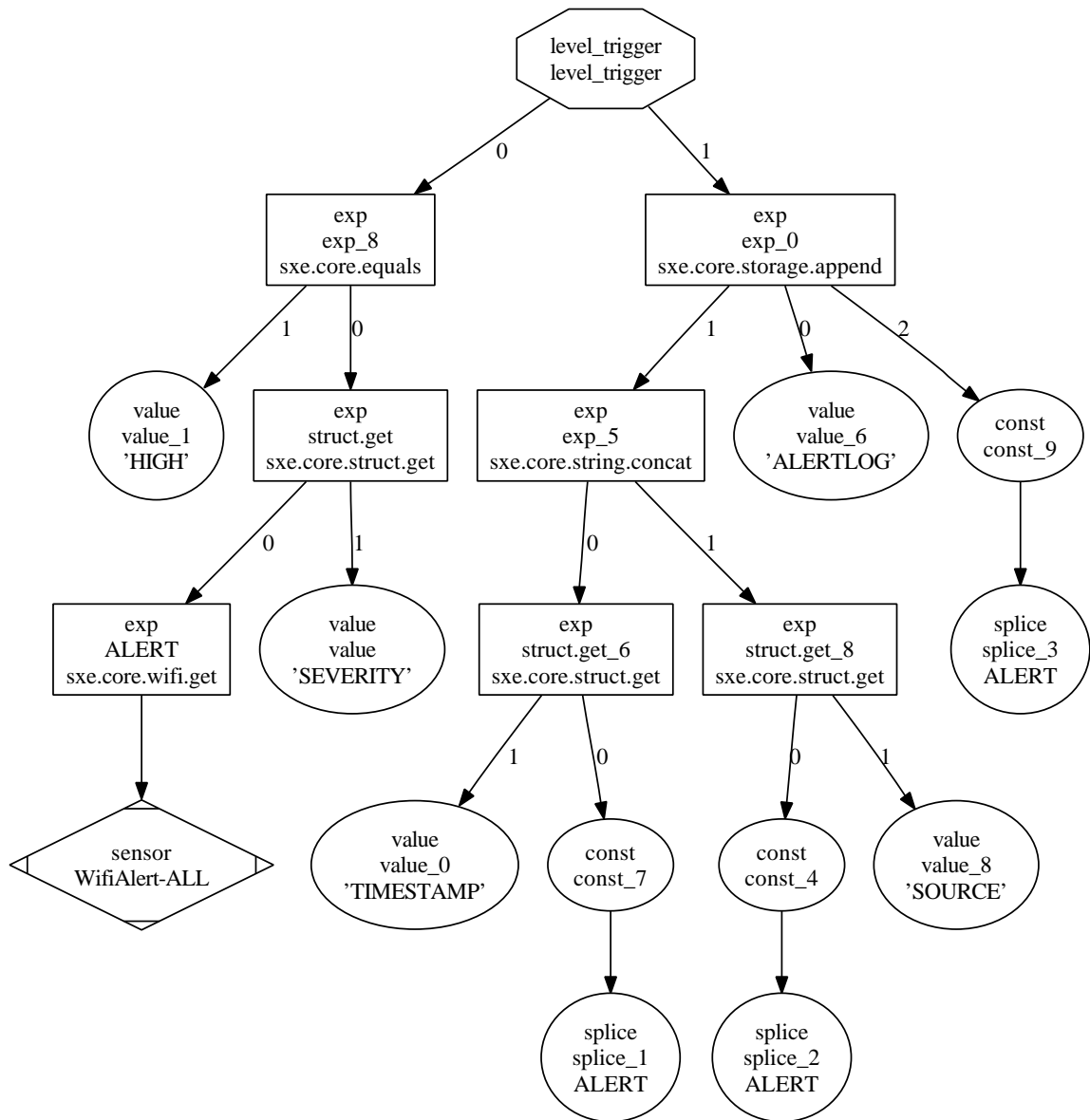


Figure 7.1: The graphical representation of STEP Program 7.7

Listing 7.8: The STEP representation of SNAFU Program 7.2.

```

<?xml version="1.0"?>
<!-- generated with SNAFU compiler 0.8 -->
<stepgraph id="WIFILEXAMPLE_2">
  <level_trigger id="LEVEL_TRIGGER">
    <exp id="EXP_3" opcode="SXE.CORE.EQUALS">
      <exp id="STRUCT.GET" opcode="SXE.CORE.STRUCT.GET">
        <exp id="ALERT" opcode="SXE.CORE.WIFI.GET">
          <sensor id="WIFIALERT-ALL">
            <device id="ALL" type="WIFIALERT"/>
          </sensor>
        </exp>
      </exp>
      <value id="VALUE">
        <sobject type="SNBENCH/STRING"><![CDATA["TYPE"]]</sobject>
      </value>
    </exp>
    <value id="VALUE_5">
      <sobject type="SNBENCH/STRING"><![CDATA["DEAUTHFLOOD"]]</sobject>
    </value>
  </exp>
  <exp id="EXP_8" opcode="SXE.CORE.CONTACT.EMAIL">
    <value id="VALUE_2">
      <sobject type="SNBENCH/STRING"><![CDATA["mocean@cs.bu.edu"]]</sobject>
    </value>
    <exp id="EXP_7" opcode="SXE.CORE.STRING.CONCAT">
      <value id="VALUE_4">
        <sobject type="SNBENCH/STRING"><![CDATA["$NOW$"]]</sobject>
      </value>
      <value id="VALUE_7">
        <sobject type="SNBENCH/STRING">
          <![CDATA[" : Deauth flood detected from MAC "]]>
        </sobject>
      </value>
    </exp>
    <exp id="STRUCT.GET_8" opcode="SXE.CORE.STRUCT.GET">
      <const id="CONST_9">
        <splice id="SPLICE_9" target="ALERT"/>
      </const>
      <value id="VALUE_0">
        <sobject type="SNBENCH/STRING"><![CDATA["MAC"]]</sobject>
      </value>
    </exp>
  </level_trigger>
</stepgraph>

```

```
</exp>
<value id="VALUE_6">
  <snoject type="SNBENCH/STRING"><![CDATA[" at time " ]]></snoject>
</value>
<exp id="STRUCT.GET_5" opcode="SXE.CORE.STRUCT.GET">
  <const id="CONST_8">
    <splice id="SPLICE_5" target="ALERT" />
  </const>
  <value id="VALUE_9">
    <snoject type="SNBENCH/STRING"><![CDATA[" TIMESTAMP" ]]></snoject>
  </value>
</exp>
<value id="VALUE_18">
  <snoject type="SNBENCH/STRING"><![CDATA[" by sensor " ]]></snoject>
</value>
<exp id="STRUCT.GET_1" opcode="SXE.CORE.STRUCT.GET">
  <const id="CONST_16">
    <splice id="SPLICE_4" target="ALERT" />
  </const>
  <value id="VALUE_11">
    <snoject type="SNBENCH/STRING"><![CDATA[" SOURCE" ]]></snoject>
  </value>
</exp>
</exp>
</level_trigger>
</stepgraph>
```

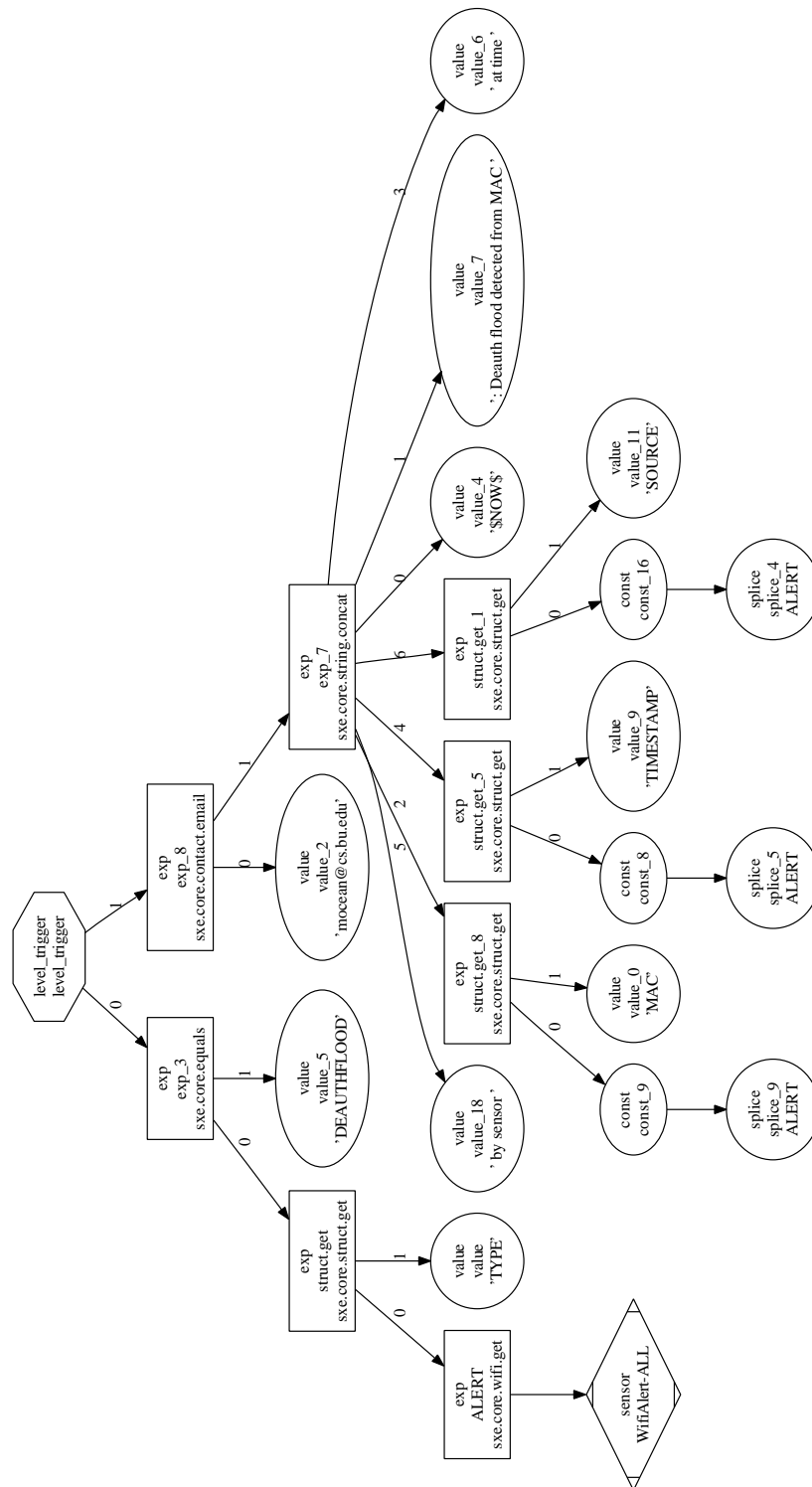


Figure 7.2: The graphical representation of STEP Program 7.8

Listing 7.9: The STEP representation of SNAFU Program 7.3.

```

<?xml version="1.0"?>
<!-- generated with SNAFU compiler 0.8 -->
<stepgraph id="WIFILEXAMPLE_4">
  <level_trigger id="LEVEL_TRIGGER">
    <exp id="EXP_5" opcode="SXE.CORE.NOT">
      <exp id="EXP_0" opcode="SXE.CORE.ISNIL">
        <exp id="ALERT" opcode="SXE.CORE.WIFI.GET">
          <sensor id="WIFIALERT-ALL">
            <device id="ALL" type="WIFIALERT"/>
          </sensor>
        </exp>
      </exp>
    </exp>
    <exp id="EXP_2" opcode="SXE.CORE.PTZ.GET">
      <exp id="EXP_1" opcode="SXE.CORE.PTZ.LOCATE">
        <exp id="EXP_3" opcode="SXE.CORE.WIFI.FIND">
          <exp id="STRUCT.GET" opcode="SXE.CORE.STRUCT.GET">
            <const id="CONST_8">
              <splice id="SPLICE_7" target="ALERT"/>
            </const>
            <value id="VALUE">
              <sobject type="SNBENCH/STRING"><![CDATA["MAC" ]]></sobject>
            </value>
          </exp>
        </exp>
      <sensor id="ACTSENSORS">
        <device id="ALL" type="WIFIACTIVITY"/>
      </exp>
    </exp>
    <sensor id="PTZSENSORS">
      <device id="ALL" type="PTZ_IMAGE"/>
    </exp>
  </level_trigger>
</stepgraph>

```

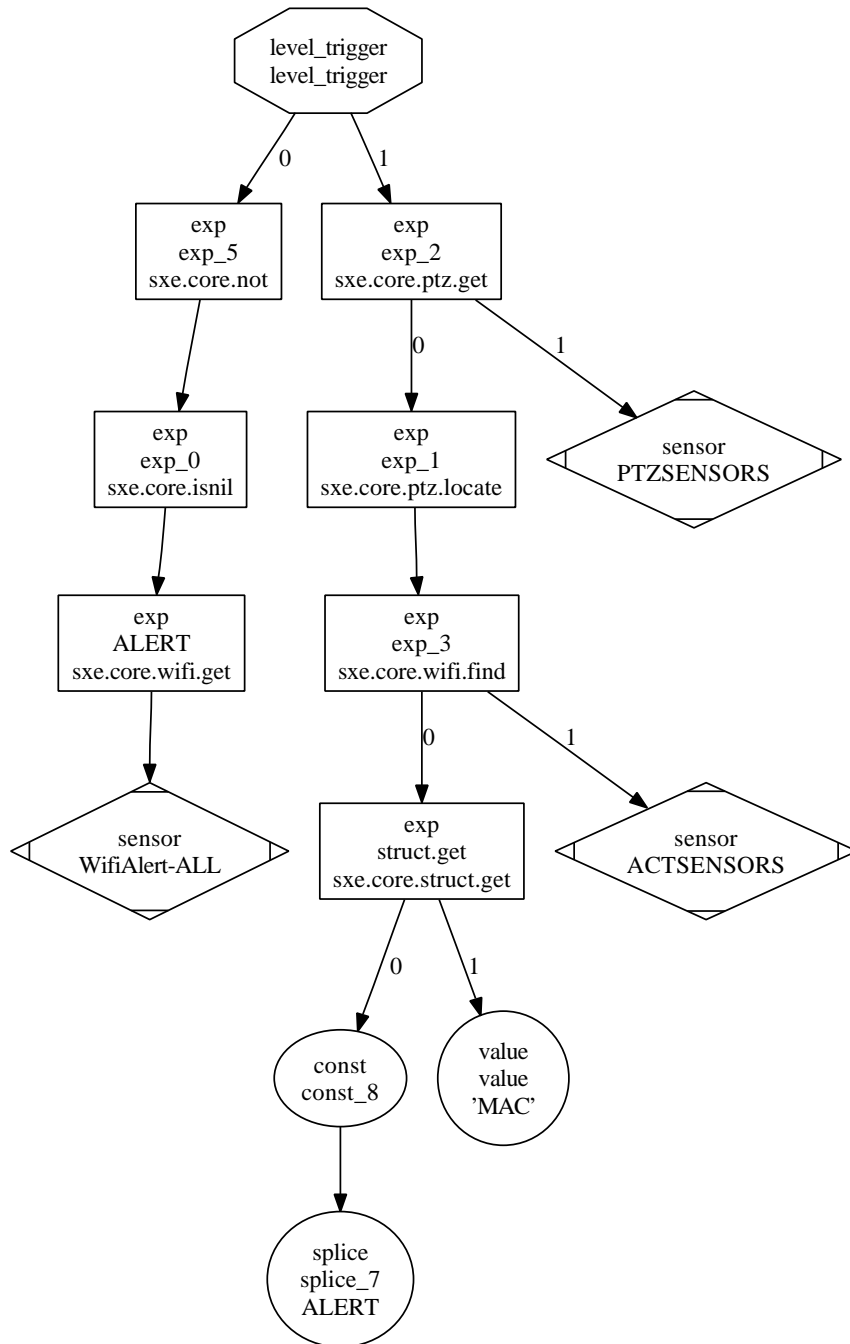


Figure 7.3: The graphical representation of STEP Program 7.9

Chapter 8

Case Study: SNBENCH as a Teaching Aide

Software and safety risks are greatly magnified in a society in which Sensor Networks (SNs) are common place and nearly all computers are remotely accessible. Sensor Network applications are more distributed, concurrent, modular and interactive – in very significant degrees – than the traditional execution environment of stand-alone applications. Development environments for such applications are also undergoing great changes, frequently calling for new compatible ways to develop software to accommodate these new needs, yet typically lagging behind the enormous and relentless hardware innovations and emergent networking concepts. The development challenges inherent to distributed sensory programming place stringent demands on the software developer who must often approach novel and unique problems ill-equipped with traditional tools and approaches.

Software Engineering (SE), in as much as it tries to be an established area of the CS curriculum with well defined topics, tends to be even more conservative. This conservatism is reflected in current SE books and in the organization of SE courses, which tend to tackle software-development problems of the traditional programming environment very well, but not so well those of the new networked and embedded environments.

At BU, we are trying to realize a SE course which can adapt to this fast-changing reality. There are limits, of course, the result of tension between the goal to adapt quickly and the need to teach well-established principles, unavoidably more suitable to the traditional pro-

programming environment than the new emerging one. Regardless, it is necessary to attempt to find such a balance, to the benefit of the students such that they may be best prepared for the future of software engineering work.

Toward this goal, this chapter details our experience piloting the use of the SNBENCH as the foundation for a semester-long final project for a 500-level Software Engineering class here at Boston University. There is obvious reciprocal benefit to such a collaboration; The curriculum benefits from the integration of the new paradigm of programming sensor networks, the research and development agenda of the SNBENCH benefit from both exposure and potential additions to the code base, and students benefit from exposure to an exciting domain in which their work is not merely intellectual exercise to be discarded.

In this Chapter we first provide an overview of the goals of the SNBENCH project and describe its architecture and relevant components. In section 8.1 we discuss in detail our goals for the use of SNBENCH within the Software Engineering course. In section 8.2 we discuss the projects offered to students as well as some of the project submissions. Finally we conclude in section 8.3 with closing thoughts and lessons learned for future efforts.

SNBENCH has been developed as a research project within Boston University to support and accelerate other research that will be developed either on top of the SNBENCH infrastructure (*i.e.*, research that leverages the SNBENCH to advance the state-of-the-art in other areas such as Computer Vision, Network Security, *etc*) or to improve future generations of the SNBENCH infrastructure (*e.g.*, resource management, allocation, negotiation, scheduling, *etc*). SNBENCH has already been used successfully in an extrinsic capacity in support of a Graduate course in Artificial Intelligence during the 2005 and 2007 Fall semesters. This report focuses on the integration of the SNBENCH into a graduate-level software engineering course during the Spring semesters from 2006 to 2008.

8.1 Software Engineering Goals

The Graduate Software Engineering course is offered by the Boston University Computer Science Department with an enrollment of roughly thirty students who are a mix of undergraduates, masters and PhD candidates. Throughout the semester students are taught software engineering principles including (but not limited to) design patterns, software life-cycle testing, type-safety, object-oriented programming principles and the use of concurrent versions system. Students are expected to complete a semester-long final project that aims to provide a realistic corporate development experience in which students work in small groups to develop modular software components that operate either within or on top of some larger work. Rather than use a contrived and static textbook project as the foundation for this work, beginning in the Spring 2006 semester we elected to leverage the Sensor Network WorkBench (SNBENCH) research project as a platform for these student projects.

Working within the SNBENCH students are given the opportunity to work in an active project in a new domain (SNs) with expertise and support readily available on campus. Our goal is to expose the students to engineering issues surrounding large-scale distributed sensory systems as well as the traditional topics associated with a SE curriculum. Within the domain of SNs students face concrete manifestations of what are largely theoretical concerns to most Undergraduate students (e.g., Security, Safety, Error Checking, Space and Time Complexity).

The student projects represent a significant part of their final grade for the course and easily comprise the bulk of the student's effort for the semester. Rather than a series of smaller assignments, the students are expected to work (as a group) on the same project throughout the semester. As this class integrates graduate and undergraduate students, expectations for undergraduates are adjusted appropriately, however the exposure to the graduate students helps give the undergrads a different perspective on their work. It should be stressed that students projects are non-competitive; Collaboration and discussion between groups is strongly encouraged.

To help orient students within SNBENCH students are provided a variety of documentation including high level motivation, functional descriptions and implementation details of existing SNBENCH software components and details of the relevant interfaces/specifications of SNBENCH modules required for the student projects. These resources are available in a single, publicly available, well organized web page for the students [OK]. Students have access to the SNBENCH codebase as well as the project submissions of students from previous semesters.

In addition to the static documentation, students are given access to the lead developer on the SNBENCH project; this access includes two lectures, one providing an overview of the SNBENCH vision and the other detailing each of the projects tasks in detail, meeting with each group individually at least once per semester, and continued email support throughout the semester.

Toward ensuring student efforts do not fall off track, each group gives a mid-semester project status presentation (also attended by the lead SNBENCH developer and other “guest” panelists) in which comments and criticisms are given to the students. In the following section we describe the individual recommended project assignments that we have offered to the students over the course of this integration.

8.2 Projects

To provide students with some initial direction, we provide up to five “suggested” project topics that ranged in complexity, effort, integration and concreteness. Suggested topics break down into two larger categories; those that enhance the SNBENCH infrastructure (*i.e.*, those that are internal to the SNBENCH) and those that are enhanced by the infrastructure (*i.e.*, those that rely on, run atop the SNBENCH). Given the multi-modal nature of the SNs targeted by SNBENCH, student projects are not limited to any single concentration/research area within CS. The breadth of project topics allow students to learn proper Software Engineering principles while implementing working in areas that they might be familiar

with or have some external interest in (*e.g.*, Computer Vision, Databases, Networking, Distributed Systems, Human Computer Interaction).

The project topic suggestions (and some notable results) from past semesters are listed below. They are presented in order of increasing complexity and effort.

When discussing results of student work, it must be emphasized that all groups have completed this work in very little time; project work occurs over approximately twelve weeks of the semester in addition to other work required for this course (in addition to a student's other course load and responsibilities). As such any omissions, flaws, or other negative assessments are largely assumed to be due to this severe time constraint.

Students are not permitted to work directly in the `SNBENCH` CVS repository. Instead, students received a CVS snapshot of the code-base, and (as part of their grade) their modifications are subject to an audit and review process compared against that snapshot. The submissions whose contribution are deemed most valuable are merged into the `SNBENCH` code-base. Naturally, not all projects are selected to be reused in the `SNBENCH` project. This does not speak to the quality of the work in all cases, as the selection also reflects the cost-benefit analysis (the amount of time and effort required to port the student work into the code base relative to the benefit of these modifications to the `SNBENCH` project overall).

8.2.1 STEP Programming GUI

This project tasks students with the goal of generating a graphical user interface (GUI) with which users can compose new `SNBENCH` programs. The Sensor Task Execution Plan (STEP) language is human readable and we (developers) are able to compose programs directly in STEP using a regular text editor. As STEP is a direct representation of the abstract syntax tree of a program, one can imagine a more accessible graphical programming interface in which a palette of nodes may be offered and wired into a flow of execution that is saved directly into STEP (XML). The details of this project asked the group to build a tool that allows a user to specify a STEP program graphically (from an extensible palette of functionalities) and save the generated program as a properly formatted STEP.

As a stand-alone component, this project was intentionally crafted to be external to the bulk of the SNBENCH code base and requires no deep integration into the existing SNBENCH components. It is, however, highly recommended that students reuse the existing internal data structures to save time and avoid redundant work. The only required integration was understanding the syntax and semantics of the STEP graph. Although a group may take this work to be as simple as developing any graphical interactive development environment and in no way special to the Sensor Network domain, students are expected to consider the needs of a novice SN developer and offer an easy and intuitive interface to the capabilities of the STEP programming language that would provide graphical access to task the resources of the Sensor Network.

In the first semester this project was offered, the work completed by [CSP06] was the best work submitted under this topic, offering a graphical STEP programming environment that supports the loading and saving STEP files, tabbed editing and an XML based pallet of capabilities. This group had clearly considered what user's may want from their editing tool as they offer tabbed editing, a (cleverly integrated) basic STEP validation engine and an option to post STEP programs for deployment directly from the editor. In addition, the layout of the editor they provided was straight-forward and effective.

This work stood out as a good basis for future development of a complete STEP graphical development environment, and this has ultimately been the case for successive semesters. The submitted work was somewhat incomplete from a usability point of view, unarguably due to time constraints. This included some operational flaws discovered during our own testing and a lack of descriptive messages/prompting leading to weak in-program communication with the user.

Although this project assignment is, by design, intended to be only nominally integrated to the existing code base (it's usage is outside of the run-time scope of the SNBENCH), a surprising problem occurred when attempting to integrate this work. This work separates the graphical construction and the STEP graph validation into two distinct operations. As such, during the composition of a STEP graph, the user is allowed to construct incomplete

or erroneous STEP graphs, presumably on the way to a final, complete STEP graph. Thus, many of the operations that this work allows the user to graphically perform on a STEP graph may be entirely safe/normal in this off-line and transient context, however some of those intermediate STEP graphs violate data structure invariants that, in an online (*e.g.*, interpretation and evaluation) context guard against data corruption or other undesirable behavior. Since the methods of the existing STEP data structures operate under the assumption that the graph is in a structurally complete or sane state, naturally there are “assertions” in the methods of the STEP data structure to ensure this is the case. As such, running this project “off-the-shelf” with assertions enabled results in premature termination due to otherwise irrelevant assertion failures.

As such the integration work for this project almost entirely entailed changes to allow the GUI to coexist with the existing STEP data structures. This included replacing and updating some of work’s interfaces to the SNBENCH objects, adding real-time checks in the GUI to prevent the construction of nonsensical STEP programs before they could be passed to the STEP data structures, and many other assorted small reliability and performance improvements.

There was little doubt that given more time this work could be polished into a superior interface. Unfortunately due to other constraints, the resources required to perfect this work were unavailable and the benefits of this work easily outweighed its flaws. Thus beyond the small corrections indicated above we had included this work otherwise “as-is”, repairing bugs that were discovered during integration but otherwise documenting its benign shortcomings with the intent that improvements will be added as an ongoing effort, by future semesters of Software Engineering students.

In successive semesters, students groups were given this submission as a basis for their work on this project topics and with this added help, the project was offered only to undergraduate students in the 2007 and 2008 semesters. The efforts of those students (in refining and extending this interface) have been extracted and reintegrated and have come to form the STEP IDE that is part of the project code-base today. Some of the most

notable additional contributions have included an XML Schema and document instance that defines the Opcode library and their corresponding type signatures, an interface that allows the user to search the Opcode library when composing a STEP graph, the ability to collapse STEP nodes into a black box, and a battery of UI refinements, bug fixes, and other niceties.

8.2.2 Expanding SXE Capabilities

This project suggestion is by far the most open-ended and domain specific, asking students to take the perspective of “SNBENCH users” and help build support infrastructure to enable interesting, useful or otherwise desirable applications on the SN framework. A successful project in our eyes is one in which the students enable some useful and well motivated application to run on the SNBENCH, adding any lacking sensor functionalities (opcodes) to support those applications. The SNBENCH Sensor eXecution Environment has been designed with opcode extensibility in mind such that the code to integrate the opcode is trivial, yet students must understand the big picture of the SNBENCH, the provided SXE Opcode interface, and the syntax and semantics of STEP such that they can build up STEP programs that utilize their new functionalities. The modularity of Opcode development also serves to aid the SNBENCH insofar as newly developed Opcodes are easily be added to the main code base without modification.

This project has been offered for all three semesters and has been fine-tuned over this time period. This project’s open-ended nature proved to be quite seductive; in the first semester offered more groups chose this project than any other. For the next two semesters we capped the number of student groups working on any single project to discourage this. Additionally the open-ended nature provided a wide range of submissions, running the gamut from barely any work at all, to those students who put their other work in jeopardy to work on their projects. A recurring theme in student submissions for this project has been a difficulty in keeping the larger vision of the SNBENCH in focus while working on “lower-level” functionalities (*i.e.*, Opcodes). Students are encouraged to implement any new opcodes they

are inspired to provide, however are recommended to “justify” the need for their opcodes by explaining their use in the overall system before proceeding with development.

In the first year the topic description gave suggestions for new opcodes from two different topics: Image Manipulation and Persistent Storage (distributed hash tables, database/SQL integration, *etc.*). We observed that, in general, undergraduate groups tended to submit only those opcodes that were directly suggested. Specifically most groups turned in little more than the suggested distributed storage opcodes. In that first year, few groups tackled the suggestions of image manipulation opcodes (*e.g.*, image differencing, motion detection, average image intensity, face detection).

One group that stood out was the work of three undergraduate students ([CFB06]) that was clearly the product of a group that had seriously considered the goals and domain of the SNBENCH. They clearly enjoyed the novelties of SN programming, as illustrated by their demonstration, which provided motion sensitive triggers to email notification (using their database connectivity to compute image differences).

This work is a clear asset to the SNBENCH; The submission includes over 30 useful new Opcodes for use within the STEP programming language including logical, mathematic, image manipulation, e-mail and distributed storage operations. To support distributed storage, this group implemented a stand-alone centralized hash server, a client for the individual SXEs, distributed hash-table Opcodes for the SXE client and finally a web interface for users/administrators to inspect the hash-table state.

Given SNBENCH’s modular Opcode architecture, after testing the new Opcodes were literally copied and pasted directly into the code-base. Integrating the hash-table opcodes was just as simple but their proper operation relies on the SXE-side hash client and hash server. Merging changes for the hash client into the SXE were straight forward due to some thoughtful design decisions and to properly fold the centralized hash server into the SNBENCH architecture it has been ported to a threaded task that is launched, controlled and monitored by the Sensorium Service Dispatcher. Closer inspection of the client and server code revealed some synchronization issues that were trivially corrected by wrapping

their data storage structure with reentrant locks. We discuss some general issues involving teaching synchronization details in a software engineering class in section 8.3.

In the second year, a group of graduate students who happened to be working in the Image and Video Computing group managed to provide much of the desired image processing functionality that is available in the `SNBENCH` opcode library today. In its third year, some interesting and novel opcodes were submitted that fell clearly outside of the suggested Opcodes; specifically Layer-2 network analysis opcodes.

8.2.3 STEP Compilation

This project tasked a group to re-develop, from scratch, the `SNAFU` (high-level SN programming language) compiler. We forgo the complete details of the project description as they are largely irrelevant. While this project was offered twice, it was only ever selected once, and never offered again for a number of reasons. First, this project offering is extremely challenging for a group with no compiler experience. Second, it is not obvious to students how work on a compiler for a small language can be broken up across group members and hence it is difficult for multiple students to work on in parallel. This latter challenge was evident in the group's submission and presentation which reflected a strong disparity between the contributions of the group's members. Unfortunately, the particular submission was so lacking in internal and external documentation that it could not be used. Despite this, the project was not a wasted effort; the updated `SNAFU` grammar (BNF) that was established during this project (between the `SNBENCH` development lead and the student group) was a very valuable outcome.

8.2.4 SXE Scheduler Modification

This project was only offered in the first semester of this integration work. This project topic recommended students make changes to the current task scheduler to improve the execution of `STEP` programs within the Sensor eXecution Environment (SXE). This is an inherently difficult task and had been considered to be the most involved project, by far.

Changes to a scheduler are difficult to test and synchronization issues are easily introduced. Thus, this project was available to graduate students only on a “by permission only” basis.

Students who selected this project (and hoped to achieve a positive result) needed to be familiar with scheduling and systems topics and must acquire a deep understanding of the existing SXE’s scheduling and evaluation operations as well as the runtime semantic of the Sensorium Task Execution Plan (STEP).

To better understand the work involved in this task, we illustrate the SXE’s basic mode of operation: Each SXE accepts a partial STEP graph which indicates the exact tasks (nodes) it is expected to execute and the dependency between the tasks (edges). As the STEP graph is directed (computations percolate upward) a STEP node higher in the graph cannot be evaluated until the lower nodes have been evaluated. Thus leaves are evaluated first and so on, upward, until a result reaches the root. Some branches of the graph may not need to be evaluated at all (*e.g.*, they are one branch on a conditional) such that checking if a node should be evaluated depends on state of both children and parents (*i.e.*, if this nodes’ children have produced “fresh” values and if the result of this node is wanted farther up the tree).

Rather than traverse the graph repeatedly, the initial SXE’s scheduler is a simple round robin scheduler that checks for enabled nodes “anywhere” in the graph. Enabled nodes are moved to a queue and are separately executed by a single thread of execution . Suggested optimizations for this project included using multiple threads to execute multiple enabled nodes simultaneously (*e.g.*, nodes of independent branches) and/or adding different scheduling algorithms that ensure the nodes get a either a fair or explicitly desired share of the SXE’s cycles. Both of the groups that selected this project topic achieved a fantastic result (which is why this project was only offered once).

The scheduling work of [PZB06] adds true hierarchical scheduling to the Sensor Execution Environment and does so with absolutely minimal changes required to the existing SXE code-base (only two methods of one class have been modified). The scheduling code is quite complete and offers the construction of a hierarchy of schedulers including Round

Robin, Fixed Priority and Proportional Share scheduling. The integration to the SXE is very clean and simple yet the integration is in somewhat incomplete and inefficient due to a lack of time. Although the work supports a variety of schedulers, code augmenting the SXE to enable reading scheduler Flow-types (*i.e.*, scheduler directives) from the STEP graph was not provided, causing much of the project's benefit to not be utilized from the SXE.

The scheduling work of [MLH06] on the other hand is considerably more ambitious with respect to integration with the existing SXE. Although the scheduling offered lacks a true hierarchical scheduler, this work offers a hybrid scheduler and the deep integration into the existing SXE implementation enables access to all their provided schedulers from within an active SXE. To support this, the group has added parsing for scheduler flow-types and modifies the scheduler to enable scheduler annotated STEP nodes to be enqueued to the correct queue. This work is exceptional in its completeness and scope (*e.g.*, excellent design, an automated regression test-suite is provided, and synchronization has been added around some existing STEP data structures.).

Unfortunately, this scheduler has an intricate enqueueing mechanism that makes its integration into the SXE more intrusive. This new approach annotates some nodes with new mechanisms for enqueueing based on partial STEP graph traversals, and flow types for unannotated nodes are rediscovered by each node re-traversing part of the graph every time this node is enabled for execution making flow-type annotations a necessity to avoid this penalty. In addition small issues surrounded this work's STEP synchronization approach, which uses a sporadic lock/release/repeat that potentially allows two simultaneous clients to manipulate data based on transient/expired data views. It is clear that these particular synchronization flaws result from an attempt to minimize changes to the admittedly complex and involved existing STEP Graph data structure. In our integration effort we have added proper synchronization that is based on our in-depth knowledge of the underpinnings of the STEP Graph data structure.

Given two excellent projects we were able to adopt the best aspects of both works into the SNBENCH code base. The scheduler of [PZB06] has been integrated as the scheduler,

adapted with [MLH06]’s flow-type parsing and support to support programmatic access to hierarchical schedules. The new resulting SXE scheduler gives every STEP program its own proportional share of the SXE and, by default, each STEP is evaluated with its own Round Robin scheduler within its share. Scheduler flow-type specifications within a STEP graph allow the nodes within a STEP program to override their default Round Robin scheduling behavior as needed, substituting Fixed Priority, Proportional Share or a true hierarchy of schedulers with the aforementioned schedulers. When a new STEP graph or fragment is added to the SXE, flow-types are discovered for the new nodes by traversing the entire graph exactly once. From every “root” node in this SXE’s STEP graph, we perform a breadth first traversal assigning new Opcodes (tasks) to their closest assigned scheduler and creating new schedulers within the hierarchy if flow-type directives are present.

8.3 Conclusions

The SNBENCH project has always been conceived as a foundation for accelerating various research initiatives and its use within a Software Engineering class stands as a testament to our progress toward that goal. Integrating the SNBENCH into a SE curriculum has provided the students with exposure to concrete instances of new and well established design and engineering challenges in the SN domain. We conclude this chapter with a discussion of our achievements and areas for improvement for future semesters.

From the point of view of the SNBENCH its successful use to support this Software Engineering class is an important milestone on the path to maturity. In addition to the intrinsic stability benchmark associated with having a large experimental user base, it is clearly beneficial to have a large group of students fluent in SNBENCH programming and use. Those students are now more likely to use the SNBENCH in the future or to recommend its use to others. The project submissions described in this report have provided important and beneficial contributions to the SNBENCH framework and the effort of their authors is greatly appreciated.

From the student's perspective, this collaborative effort was well received. Students reported interest and excitement being able to work on an active project in a still evolving domain. Indeed the participation in the project became quite consuming to many of the students, some of whom petitioned for the project to be worth a greater portion of their grade and a larger part of the class over all. Those students who were informed that their work would be reused in the SNBENCH project expressed pleasure in the idea that their efforts were legitimately useful. Although not every project that was turned in was able to be absorbed into the code base, all of the students have provided useful contributions to the SNBENCH framework. Every student project has taught us valuable lessons as to how we might adjust the project assignments and student development in future semesters.

Lessons Learned and Future Adjustments

Although the collaboration was a clear success, we view this effort as ongoing and strive to improve future iterations and detail some lessons learned in this section.

Material

Adding the SNBENCH to the curriculum of the class brought many systems-centric engineering issues (*e.g.*, synchronization and resource allocation) to the forefront of the students' SE projects. Generally this material is not covered deeply in SE classes despite the clear increasing trend toward shared and distributed architectures in modern software. This trend must be reflected in the SE curriculum and as such design principles for concurrency and synchronization should become a larger part of this course.

Some students had expressed a preference for the project work as opposed to the other requirements associated with the class (*e.g.*, textbook readings, quizzes and exams). Although the textbook readings and exams cannot be replaced, it may be beneficial to integrate the project material into some of the lessons more closely or to replace quizzes with small SNBENCH integrated assignments (*e.g.*, a selection of objects from the existing source may be selected to illustrate use of various design patterns).

Student Guidance

Although each group met with the lead SNBENCH developer in person once, it seems beneficial for students to discuss their projects and plans more often throughout the semester. The best work done in this class tended to be that of the groups who met with the lead developer more than once during the semester. Although such meetings may be time consuming, there is little doubt that the benefit of such meetings far exceed the time investment. These meetings may also help groups stay on track and keep the big (SN) picture in mind during their development cycle.

Similarly, students present a mid-semester progress report as well as an end of semester progress report. The mid-semester report is useful to determine if groups are headed in the right direction. These presentations have a length that balances consumed lecture time and the time required to be useful. In this semester the 12 minutes per group does not shed enough light on what a group is doing and as the semester is already half over by the time of the mid-semester report, it is generally too late to recommend any drastic changes if they are needed. It may make sense to require groups to meet with faculty individually outside of class both one-quarter and half-way into the semester for a more in-depth review (in addition to the mid-semester presentation).

Student Collaboration

Despite urgings from the faculty, groups did not collaborate across groups as much as we had hoped. Although there is certainly a sense of competition that is natural when multiple groups are working on the same topic/project, there is room for groups to discuss their work with one another for help. While student groups maintained public web pages that were visible by all, very few students looked at each other's web pages, and had little idea what the other groups were doing or how they were doing it. This problem is difficult to fix, but it may be the case that providing incentives for collaboration may solve this problem.

Time Limitations

Another difficulty is that once the semester ends students' work on projects ends as well. Given the short semester and the ambition of some groups this has the undesirable consequence of incomplete work. Although this may be corrected with better advise regarding expectation and time allotments, for particularly promising work we may wish to offer an arrangement that would allow select groups to continue, polish and integrate their work into the SNBENCH code base for credit. It might also be nice to be able to give students detailed feedback about their work after the semester has ended and the code has been thoroughly reviewed. Even if this information is too late to benefit them for the current semester, such feedback may be useful for the rest of their academic careers and beyond.

Suggested Projects

We found that while providing an assortment of projects was beneficial to the groups, a careful balance must be struck to maintain requirements of a proper academic exercise. Allowing more students to work particular project resulted in greater discussion of the issues and challenges in that project and ultimately better submissions. This effect is strengthened even when some students are disinclined to ask for help, as they will still benefit from the answers to questions asked by others. Additionally time/effort spent answering questions for a project that has a large number of students in turn benefits a large number of students. Offering too many projects has lead to a situation in which some projects are selected by only one group, thus increasing the overhead and decreasing the utility of support and documentation efforts. Reducing the total number of projects limits the options provided to students, however it guarantees that there will be multiple groups working on the same project which leads to multiple submissions on the same project, and an increased likelihood of a strong result being adopted into the code base (either from either a single work or a composite of submissions).

Chapter 9

Conclusion and Future Work

In this chapter we summarize the contributions of the work, reflect on several design decisions that would be made differently with the knowledge that we have presently, and suggest several possible areas for additional research in this domain.

9.1 Summary of Contributions

The SNBENCH project offers the a Sensor Network programming and execution framework that supports the *entire* life-cycle of Sense-and-Respond Services. The framework revolves around a single (*i.e.*, common) tasking language that is the target of multiple high-level programming languages and may be re-translated for execution on severely constrained devices. The use of this common tasking language enables the use of a variety of high-level programming languages (and interfaces) such that users are not bound to a programmatic single interface. All run-time components provided (including resource allocation, task assignment and task execution) are capable of processing this common tasking language. In providing sufficiently insular constructs within these high-level languages, SNBENCH achieves its goal of enabling novice users to task and ultimately deploy SN services. As the library of functionalities and datatypes available for manipulation by the common tasking language are extensible, SNBENCH can be updated to support new sensors, actuators, and computations for new deployment scenarios, as needed.

SNBENCH includes static verification of newly submitted services toward the goals of safety verification and providing static-time resource bounds. This contribution is particularly valuable in the context of image manipulation services, wherein treating images as first class datatypes yields the ability to bound the image resolution range required for specific computations. Tracking these resolution bounds provides constraints on physical sensory resources (*i.e.*, image sensors) to preserve functional correctness, while simultaneously considering impact to execution time or other run-time constraints (*i.e.*, Flowtypes).

9.2 Future Directions

9.2.1 Resource Allocation

Expressive Naming and Name Resolution

At present we support two extremes of sensor naming: completely agnostic (*i.e.*, ANY or ALL sensor of a specific type) or specific (*e.g.*, give the URI of the physical SXE (host) and the ID of the sensor). The use of URIs requires the Resource Manager to maintain knowledge of all sensors connected to each host and perform computation at the time of task submission to resolve resources to compute and reserve physical sensor resources. We wish to generalize sensor naming to support a middle ground of naming based on potentially dynamic attributes (the type of the sensor is a static attribute). Examples include location (*e.g.* “The webcam in Azer’s Office”), naming by property (*e.g.* “Any two cameras aimed at Michael’s chair by 90 degrees apart”), naming by performance characteristics (*e.g.* “Any processing element within 2msec from WebCam1 and WebCam2”), and naming by content (*e.g.* “Any webcam which sees Assaf right now”). Such naming conventions will require persistent, prioritized STEP queries to be running as the basis for these results, however it is unknown which such persistent queries should be instantiated, the resource cost of allocating sensors for these tasks, and how we can express these tasks as more commonly used expressions such that we produce the highest odds of potential computational reuse in tandem with the most valuable property detection.

Performance Profiling/Benchmarking

Our present performance monitoring uses stub code to represent the free computational resources an SXE has and the computational cost of each opcode. It is clear that an accurate characterization of the computational availability of resources and each opcodes computational requirements will be needed to enable the SSD to *accurately* allocate resources and dispatch programs. We envision a solution in which SXEs generate simple performance statistics about each opcode as it is run, and these statistics are reported to the local SRM to build opcode performance profiles. Such an approach allows new opcodes to be developed with their profiles dynamically built and probabilistically refined.

This is extremely important to ensure operational correctness of the type system, which is contingent on the presence of accurate size, cost and constraint data for Opcodes (primitive operators). SNBENCH provides a facility by which new Opcodes may be added to a service library quickly and easily, through the implementation of a simple Java interface. At present the type system maintains an embedded definition for the latent costs and size constraints of the current Opcode library, however for the sustainability of the type system, it is essential that these definitions are provided by the Opcode authors and automatically extracted directly from the Opcode definitions. Beside changing the Opcode implementation interface, the type system must also change to import the rules from the Opcode library directly. There is also a concern that Opcode authors might specify very weak size constraints and very high costs (possibly lacking the knowledge to do so correctly) and that the end result is a type system that is only as strong as its weakest link. Any future work that would automatically extract this information from an Opcode implementation would be a fantastic solution to this problem (and others).

Fairness and Resource Cost

With multiple users tasking SN resources, some notion of fairness must be provided to prevent any user(s) from monopolizing the system. While several approaches exist in the Distributed Computing domain (*e.g.*, fair share, virtual currency, auctions) the solution and

infrastructure to provide fairness must minimize bandwidth, CPU overhead and the latency between job submission and execution. In keeping with the modularity of SNBENCH this solution should be as modular as possible, allowing different solutions to be “swapped-in” depending on the needs of the deployment.

Graph Partitioning and Optimality

From the perspective of communication cost, there is performance pressure to generate STEP schedules (STEP graph partitions) in which contiguous regions are scheduled to the same SXE, to minimize communication between SXEs. Although we can use the size-annotated datatypes as an indication of the communication cost, it may be the case that those expressions that receive large amounts of data as input may have computation costs which dwarf their communication cost (*e.g.*, pattern matching, face-finding, *etc.*). The deployment of such resource intensive expressions may generate graphs in which we have many small regions and high communication cost. As the task assignment strategy is highly dependent on the particular SN deployment (and to the extent to which we intend to leverage SNBENCH as a framework for ongoing SN research) the SSD provides a simple interface that allows SN engineers to use their own, custom task allocation implementation.

That said, unlike Grid computing environments (*e.g.*, Emulab), a SN user may have a short, one-time SN service that, given some external deadline, requires immediate deployment and thus cannot wait for an optimal resource allocation (*e.g.*, observing an event that is going to occur imminently, but only once). Quick and dirty resource allocation algorithms may exhibit extreme service fragmentation (excessive communication due to low availability on every node) or resource affinity (most services are assigned to the same nodes); these limitations may be easily overlooked if the cost (*e.g.*, time required) of “optimal” resource allocation exceeds the total cost of the new service itself.

By providing resource allocation as a customizable module, it would be beneficial to have a way to automatically characterize a particular resource allocation (*i.e.*, graph partitioning) algorithm across a variety of metrics (*e.g.*, response time, fragmentation, fairness).

9.2.2 Extending snBench to Other Domains

Applications to Cloud Computing

The distributed sensing and responding systems targeted by SNBENCH are, in principle, distributed systems. Extending SNBENCH to additionally support a more traditional distributed system architecture (*e.g.*, the Google Cloud, PlanetLab) requires more than ensuring that the Sensor eXecution Environment will run on the nodes in this space (they will). Rather, supporting execution in these environments requires adjustments the Opcode library to (1) better support (or essentially, *wrap*) traditional and potentially external “batch” processing jobs with an opcode and (2) to enable arbitrary fragmentation/partitioning of these jobs within SNBENCH to enjoy the modularity and resource assignment benefits that SNBENCH provides in the large-scale, cloud computing domain. Additionally, the high-level languages to compose such large-scale computations should be designed particularly for this target deployment.

Distributed, Dynamic Robotics Applications

The benefits of extending SNBENCH to include robotic devices are significant. The tasks executed on a robotic device may be adjusted from local sensory input (*e.g.*, proximity sensors) as well as from sensors that are connected to other, remote devices (*e.g.*, neighboring devices). The ability to access both local and remote sensory inputs, logically, extends the “sensory perception” of a robotic device to cover the entire space monitored by a VSN.

As a concrete example, consider a multi-modal application that detects a spill on an embedded video network, and dispatches the closest robot to the spill location to clean it up. A robot will use its local proximity sensors to ensure that a collision is not imminent, but it may also use input from external video sensors to indicate that a pathway has been blocked, a new destination is required, or that its services are no longer required.

Additionally the actuation (response) capabilities of these devices motivate the need for new abstractions and constructs to deal with the challenges inherent to sharing mobile

devices; a careful balance must be reached when defining the atomicity of actuation tasks to ensure that multiple users can “simultaneously” use the device, without actively disrupting the tasks of other users.

The infrastructure provided by SNBENCH is well suited to address these challenges in a number of ways: (1) the native SXE is ideal for such constrained target devices, (2) our static analysis techniques can be used to bound the processing delay (*i.e.*, responsiveness) of a robotics task, and this delay can be used to determine the likelihood of collision in the event of sudden changes in the environment, and (3) the extensible task allocation module of the Service Dispatcher makes it easy to “drop in” new task allocation approaches that include custom logic for task assignment to robotic devices.

Scalability of Networked SSDs

As mentioned earlier, the SSD and SRM maintain a resource view for a “local-area” Sensorium. Although this hierarchal division seems rather natural, the number of resources to be monitored by an SSD must be “within reason”. As more Sensoria come on-line, there will inevitably be demand for computations that involve resources of disjoint Sensoria (*e.g.*, nodes on the periphery between two SSD regions). Such “*elastic* SNs” require potentially specialized addressing, resource allocation, and program composition approaches; these approaches must be implemented, verified, and evaluated for scalability.

Comparing Programming Granularities

A multitude of programming languages exist today with different aims (*e.g.*, scripting, printing and reporting) and all of which leverage the API provided by the Operating System. Similarly we expect multiple, high-level SN languages to emerge targeting different types of applications that are all compiled to the SNBP’s API. As defining a “best” API is impossible given the inherent qualitative nature of such a discussion, some metrics must be defined to quantitatively compare SN APIs.

9.2.3 Security and Privacy

Security and Safety

The emergence of embedded SNs in public spaces produces a clear and urgent need for well-planned, safe and secure infrastructure as security and safety risks are magnified. For example, a hacker gaining access to private emails or crashing a mail server is certainly bad, however it is clearly worse if that same hacker can virtually case an office via stolen video feed, disable the security system, remotely unlock the door, and steal both the physical mail server and the data it contains. In any remotely programmable system it is imperative that only authorized users task resources and that data cannot be intercepted by unauthorized third-parties. Similarly there is a need to ensure that the SN computation correctness is not compromised by malicious entities spoofing legitimate SN components.

SNBENCH provides an ideal infrastructure to experiment with the inherent security issues in this novel domain; easing the incorporation of mechanisms that provide authentication support for privacy, constraints, and trust. Traditional security approaches may prove too heavy weight for the highly-constrained SN domain. At present, most works underscore the importance of security concerns yet fail to address them in any concrete way.

9.2.4 Safety and Verification

Additional Type Annotations

Other useful type annotations (beyond upper and lower size bounds) and can be integrated into the type system of the SNBENCH. In Chapter 5 we gave the examples of image quality, numerical quality, color-depth for images, frame rates for video, and so on.

Verification of Other Properties

Determining SN program faults statically (*i.e.*, static type-checking) has established value in the domain of desktop machines, and our work has begun to illustrate the value of extending static verification techniques to SN programs. In addition to the existing static

type-checking provided by `SNBENCH`, one could easily imagine extending further safety and analysis techniques into this domain (*e.g.*, run-time verification, deadlock detection/prevention, race condition detection on shared state).

Software/Sensor Fault Isolation

The Sensor eXecution Environments (SXE) of `SNBENCH` may be executing portions of multiple unrelated services simultaneously. As such it is essential that any a single erroneous computation or task does not expose the data of other private or privileged computations that happen to be located on the same resource.

Bibliography

- [AAJI04] Frank Adelstein, Prasanth Alla, Rob Joyce, and Golden G. Richard III. Physically locating wireless intruders. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing, Volume 2*, page 482, Washington, DC, USA, 2004. IEEE Computer Society.
- [ABC⁺04] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS '04: Proceedings of the International Conference on Distributed Computing Systems*, 2004.
- [AH91] F. Ackermann and M. Hahn. Image pyramids for digital photogrammetry. *Digital Photogrammetric Systems*, pages 43–59, 1991.
- [Air] AirDefense, Inc. AirDefense Enterprise Product Homepage. <http://www.airdefense.net/products/enterprise.php>.
- [And] Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia. GLPK (GNU Linear Programming Kit). <http://www.gnu.org/software/glpk/>.
- [AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

- [Bau] Dave Baum. Not Quite C (NQC). <http://www.enteract.com/~dbaum/nqc/index.html>.
- [BBH05] Azer Bestavros, John Byers, and Khaled Harfoush. Inference and Labeling of Metric-Induced Network Topologies. *TPDS: IEEE Transactions on Parallel and Distributed Systems*, 2005.
- [Bes97] Azer Bestavros. Load Profiling: A Methodology for Scheduling Real-Time Tasks in a Distributed System. In *ICDCS'97: Proceedings of the IEEE International Conference on Distributed Computing Systems, Baltimore, Maryland, May 1997*.
- [BHH⁺05] Phil Buonadonna, Joseph Hellerstein, Wei Hong, David Gay, and Samuel Madden. Task: Sensor network in a box. In *EWSN 2005: Proceedings of the Second European Workshop on Wireless Sensor Networks*, 2005.
- [BHS03] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, pages 187–200, New York, NY, USA, 2003. ACM Press.
- [Bjo] Bjoern Frank. GLPK 4.8 Java Interface. <http://bjoern.dapnet.de/glpk/index.htm>.
- [BP00] Paramvir Bahl and Venkata N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *INFOCOM 2000: Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Volume 2*, pages 775–784, 2000.
- [BPP05] Amol Bakshi, A. Pathak, and Viktor K. Prasanna. System-level Support for Macroprogramming of Networked Sensing Applications. In *PSC '05: Proceed-*

ings of the International Conference on Pervasive Systems and Computing, June 2005.

- [BPRL05] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *EESR '05: Proceedings of the 2005 Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services*, pages 19–24, Berkeley, CA, USA, 2005. USENIX Association.
- [BS03] John Bellardo and Stefan Savage. 802.11 denial-of-service attacks: real vulnerabilities and practical solutions. In *SSYM'03: Proceedings of the 12th Conference on USENIX Security Symposium*, pages 2–2, Berkeley, CA, USA, 2003. USENIX Association.
- [CBMP03] Jeffrey Considine, John W. Byers, and Ketan Mayer-Patel. A case for testbed embedding services. In *HotNets-II: Proceedings of the Second Workshop on Hot Topics in Networks*, November 2003.
- [CFB06] Dave Cecere, Ben Freiberg, and Dustin Burke. Extending snBench to Support Distributed Storage Across Sensors. Technical Report BUCS-TR-2006-015, CS Department, Boston University, September 2006.
- [CG05] Kevin K. Chang and David Gay. Language support for interoperable messaging in sensor networks. In *SCOPES '05: Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems*, pages 1–9, New York, NY, USA, 2005. ACM Press.
- [CGG⁺05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Tinylime: Bridging mobile and sensor networks through middleware. In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 61–72, Washington, DC, USA, 2005. IEEE Computer Society.

- [CLKB04] J. Considine, F. Li, G. Kollios, and J. W. Byers. Approximate Aggregation Techniques for Sensor Databases. In *ICDE '04: Proceedings of the 20th IEEE International Conference on Data Engineering*, Boston, April 2004.
- [CSP06] Ching Chang, Raymond Sweha, and Panagiotis Papapetrou. Extending snBench to Support a Graphical Programming Interface for a Sensor Network Tasking Language (STEP). Technical Report BUCS-TR-2006-014, CS Department, Boston University, July 14 2006.
- [CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL: Proceedings of the Symposium on Principles of Programming Languages*, pages 184–198, 2000.
- [Dev] Christophe Devine. Aircrack-ng homepage. <http://www.aircrack-ng.org/>.
- [Eka] Ekahau, Inc. Ekahau Positioning Engine 4.0 Product Homepage. <http://www.ekahau.com/products/positioningengine/>.
- [Elo] Jarno Elonen. NanoHTTPD homepage. <http://elonen.iki.fi/code/nanohttpd/>.
- [ENRS95] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Divide-and-conquer approximation algorithms via spreading metrics (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 62–71, 1995.
- [Far05] Jamil Farshchi. Wireless intrusion detection systems. <http://www.securityfocus.com/infocus/1742>, 2003-11-05.
- [GBM⁺04] Mina Guirguis, Azer Bestavros, Ibrahim Matta, Niky Riga, Gali Diamant, and Yuting Zhang. Providing Soft Bandwidth Guarantees Using Elastic TCP-based Tunnels. In *ISCC'04: Proceedings of the IEEE Symposium on Computer and Communications*, Alexandria, Egypt, 2004.

- [GGG] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. <http://citeseer.ist.psu.edu/gummadi05macroprogramming.html>.
- [GKE04] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 69–80, New York, NY, USA, 2004. ACM Press.
- [GOKL06] Yarom Gabay, Michael Ocean, Assaf Kfoury, and Likai Liu. Computational Properties of SNAFU. Technical Report BUCS-TR-2006-001, CS Department, Boston University, February 6 2006.
- [Han] John Hansen. Not eXactly C (NXC) Programmer's Guide. http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf.
- [HFH06] Kevin Hammond, Christian Ferdinand, and Reinhold Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *SIGBED Review: ACM Special Interest Group on Embedded Systems*, 3(4):27–36, 2006.
- [HM06] Salem Hadim and Nader Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3):1, 2006.
- [HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL: Proceedings of the Symposium on Principles of Programming Languages*, pages 410–423, 1996.
- [HR06] Karen Henricksen and Ricky Robinson. A survey of middleware for sensor networks: state-of-the-art and future directions. In *MidSens '06: Proceedings of the International Workshop on Middleware for Sensor Networks*, pages 60–65, New York, NY, USA, 2006. ACM Press.

- [HSW⁺00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ASPLOS: Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.
- [IBM] IBM Internet Security Systems. Wireless products homepage. http://www.iss.net/documents/whitepapers/wireless_LAN_security.pdf.
- [JHH⁺93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *JFIT: Proceedings of the UK Joint Framework for Information Technology Technical Conference*, 1993.
- [JSS00] Chaiporn Jaikaeo, Chavalit Srisathapornphat, and Chien-Chung Shen. Querying and Tasking in Sensor Networks. In *SPIE: Proceedings of the 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V)*, Orlando, Florida, April 24–28 2000.
- [Kera] Mike Kershaw. Kismet user forum. <http://www.kismetwireless.net/Forum/General/Messages/1142522037.4893529>.
- [Kerb] Mike Kershaw. Kismet (version 2007-01-r1b). <http://www.kismetwireless.net/documentation.shtml>.
- [KGMG07] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI'07: Proceedings of the Special Interest Group on Programming Languages (SIGPLAN) Conference on Programming Language Design and Implementation*, 2007.
- [KL99] Jonathan B. Knudsen and Mike Loukides. *Unofficial Guide to LEGO MIND-STORMS Robots*. O'Reilly and Associates, Inc., Sebastopol, CA, 1999.

- [Kro91] W. Kropatsch. Image pyramids and curves. an overview. Technical report, Department for Pattern Recognition and Image Processing of the Institute of Automation, University of Technology, Vienna, Austria, 1991.
- [KWA⁺03] Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin, Phillip Hutto, Arnab Paul, and Umakishore Ramachandran. DFuse: a framework for distributed data fusion. In *SenSys '03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 114–125, New York, NY, USA, 2003. ACM Press.
- [LAHS06] Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. Envisuite: An environmentally immersive programming framework for sensor networks. *Transactions on Embedded Computing Systems*, 5(3):543–576, 2006.
- [Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1 1964.
- [LC02] Philip Levis and David Culler. Mate: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.
- [lfb] “loud-fat bloke”. WIDZ (Wireless Intrusion Detection System) Homepage. <http://freshmeat.net/projects/widz/>.
- [LGC05] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI'05: Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation*, May 2005.
- [LH96] Hans-Wolfgang Loidl and Kevin Hammond. A sized time system for a parallel functional language. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1996.

- [Lic] GNU Public Licence. The GNU Compiler for the Java Programming Language. <http://gcc.gnu.org/java/>.
- [LM03] Ting Liu and Margaret Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *PPoPP '03: Proceedings of the Ninth ACM Special Interest Group on Programming Languages (SIGPLAN) Symposium on Principles and Practice of Parallel Programming*, New York, NY, 2003. ACM Press.
- [LN] LEGO and NI LabVIEW. LEGO MINDSTORMS NXT software. http://mindstorms.lego.com/Overview/NXT_Software.aspx.
- [Loc] Andrew Lockhart. Snort-wireless homepage. <http://snort-wireless.org/>.
- [LPCS04] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI: Proceedings of the First Symposium on Network Systems Design and Implementation*, March 2004.
- [LRG03] Joshua Lackey, Andrew Roths, and James Goddard. Wireless intrusion detection. http://www-935.ibm.com/services/us/bcrs/pdf/wp_wireless-intrusion-detection.pdf, 2003.
- [LRW⁺05] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, pages 149–162, New York, NY, USA, 2005. ACM Press.
- [Lyn] Michael Lynn. AirIDS Project Homepage. <http://airids.sourceforge.net/>.

- [LZ05] Jie Liu and Feng Zhao. Towards semantic services for sensor-rich information systems. In *BROADNETS: Proceedings of the 2nd International Conference on Broadband Networks*, pages 44–51, 2005.
- [MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review*, 36(SI), 2002.
- [MFHH05] 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*, 30(1):122–173, 2005.
- [MLH06] Sowmya Manjanatha, Jorge Londono, and Zhinan Han. An Extension to the Sensorium Execution Environment (SXE) to Provide Concurrency Support in the snBench framework. Technical Report BUCS-TR-2006-013, CS Department, Boston University, July 14 2006.
- [MLM⁺05a] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Matthias Gauger, Olga Saukh, and Kurt Rothermel. Management and configuration issues for sensor networks. *International Journal of Network Management – Special Issue: Wireless Sensor Networks*, 15(4):235–253, 2005.
- [MLM⁺05b] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *EWSN 2005: Proceedings of the Second European Workshop on Wireless Sensor Networks*, pages 278–289, January 2005.
- [MSWB04] J. J. Magee, M. R. Scott, B. N. Waber, and M. Betke. EyeKeys: A Real-time Vision Interface Based on Gaze Detection from a Low-grade Video Camera.

In *RTV4HCI: In Proceedings of the IEEE Workshop on Real-Time Vision for Human-Computer Interaction*, July 2004.

- [MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 85–97, 1999.
- [NAW05] Ryan Newton, Arvind, and Matt Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. In *IPSN: Proceedings of the International Symposium on Information Processing in Sensor Networks*, 2005.
- [NW04] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.
- [OBK06] Michael J. Ocean, Azer Bestavros, and Assaf J. Kfoury. snbench: programming and virtualization framework for distributed multitasking sensor networks. In *VEE '06: Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments*, pages 89–99, New York, NY, USA, 2006. ACM Press.
- [OK] Michael J. Ocean and Assaf Kfoury. snBench documentation for the Spring '08 offering of cs511. <http://cs-people.bu.edu/mocean/cs511>.
- [PZB06] Gabriel Parmer, Georgios Zervas, and Angshuman Bagchi. Extending snBench to Support Hierarchical and Configurable Scheduling. Technical Report BUCS-TR-2006-012, CS Department, Boston University, July 14 2006.
- [RG94] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LISP and Functional Programming*, pages 65–78, 1994.

- [RKM02] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. *SIGMOBILE: Mobile Computing and Communications Review*, 6(4):59–61, 2002.
- [Roe99] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA '99: Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.
- [SCC⁺06] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.
- [Smi07] Randall B. Smith. Spotworld and the sun spot. In *IPSN: Proceedings of the International Symposium on Information Processing in Sensor Networks*, pages 565–566, 2007.
- [SN04] Jonathan M. Smith and Scott Nettles. Active networking: One view of the past, present, and future. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 34(1):4–18, 2004.
- [Sri] Sriram Sankar, Metameta. Java Compiler CompilerTM(JavaCCTM) - The Java Parser Generator. <https://javacc.dev.java.net/>.
- [SSJ01] Chien-Chung Shen, Chavalit Srisathapornphat, and Chaiporn Jaikaeo. Sensor Information Networking Architecture and Applications. *IEEE Personnel Communication Magazine*, 8(4):52–59, August 2001.
- [Sys06] Cisco Systems. Wi-Fi Based Real-Time Location Tracking: Solutions and Technology. http://www.cisco.com/application/pdf/en/us/guest/products/ps6386/c1244/cdcont_0900aecd80477957.pdf, 2006.

- [The] The OpenWrt Team. OpenWRT Project Homepage. <http://openwrt.org/>.
- [TLS05] Tai-Peng Tian, Rui Li, and Stan Sclaroff. Tracking Human Body Pose on a Learned Smooth Space. In *IEEE Workshop on Learning in Computer Vision and Pattern Recognition*, 2005.
- [TRLW03] P. Tao, A. Rudys, A. Ladd, and D. Wallach. Wireless LAN location sensing for security application. In *WISE'03: Proceedings of the ACM Workshop on Wireless Security*, 2003.
- [Tru] Nathan True. Wi-viz: Wireless Network Environment Visualization. <http://devices.natetrue.com/wiviz/>.
- [VVK03] Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. *Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes*, 28(5):88–97, 2003.
- [WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, pages 99–110, New York, NY, USA, 2004. ACM Press.
- [WZL06] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *EWSN 2006: Proceedings of the Second European Workshop on Wireless Sensor Networks*, pages 5–20, 2006.
- [WZSP04] Richard West, Yuting Zhang, Karsten Schwan, and Christian Poellabauer. Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers. *IEEE Transactions on Computers*, 53(6), 2004.

- [XH01] Hongwei Xi and Robert Harper. A dependently typed assembly language. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, 36(10):169–180, 2001.
- [Xi04] Hongwei Xi. Applied type system (extended abstract). In *Post-Workshop Proceedings of TYPES 2003: Computer-Assisted Reasoning based on Type Theory*, pages 394–408. Springer-Verlag LNCS, 2004.
- [YAS03] Moustafa Youssef, Ashok Agrawala, and Udaya Shankar. WLAN Location Determination via Clustering and Probability Distributions. In *PerCom 2003: Proceedings of the IEEE Annual Conference on Pervasive Computing and Communications*, March 2003.
- [YG02] Yong Yao and Johannes Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM Special Interest Group on Management Of Data (SIGMOD), SIGMOD Record*, 31(3), 2002.
- [Yoa] Yoann Vandoorselaere, et. el. Prelude Hybrid IDS. <http://www.prelude-ids.org/>.
- [YRBL06] Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 139–152, New York, NY, USA, 2006. ACM Press.
- [ZBG⁺05] Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West. Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation. In *VEE'05: Proceedings of the 1st ACM/USENIX Conference on Virtual Execution Environments*, June 2005.

Curriculum Vitae

Michael James Ocean
mocean@cs.bu.edu

Education

- Ph.D.** Computer Science, Boston University, (Expected) January 2009.
*The Sensor Network Workbench: Towards Functional Specification,
Verification and Deployment of Constrained Distributed Systems*
Advisor: Azer Bestavros
- B.S.** Computer Science, Rutgers College, May 1998.

Academic Experience

Research Fellow, Boston University 2003 - *current*
Department of Computer Science

System Design, Distributed Systems, Safety and Assurance Systems.

My research at BU revolves around the intersection of Systems, Languages and Networks culminating in the design, development, and refinement of the SNBENCH: a platform for research on constrained, sense and respond distributed systems. My research on and within the SNBENCH reflects these interests: specifically, enabling virtualization of resources, combining security on the physical and wireless planes, as well as my current work to establish a sized type system toward static safety verification and resource constraints for distributed applications.

Teaching Assistant, Boston University 2006 - *current*
Department of Computer Science

Object Oriented Software Principles; Spr06, Spr07, Spr08.

My role in supporting the mixed graduate and undergraduate Software Engineering class includes designing a set of student projects that run on the SNBENCH, and working closely with students to shepherd their projects toward successful completion. I maintain support documents, give guest lectures, and assist with project grading. For undergraduate groups my help extends to editing their semester project reports into departmental Technical Reports published in their names.

Lecturer, Boston University
Department of Computer Science

2003 - 2006

*Introduction to Computer Science for non-majors;
F03, Spr04, Su04, F04, Spr05, Su05, F05 (two sections), Spr06 (two sections).*

I redesigned this large undergraduate introductory class as an introduction to the *science* of computers, making the material accessible and engaging for potentially disinterested students taking the course as a University prerequisite. I developed new course materials including lecture slides, notes, homework, and exams and also coordinated the lab sessions, teaching fellows and graders. While enrollment varied depending on semester (the largest section was 130 students, and the smallest was six) student reviews were consistently overwhelmingly positive.

Professional Experience

Research Scientist, Telcordia Technologies
Multimedia Research Group

1998 - 2003
Morristown, NJ

Working in Applied Research I pursued a wide range of research projects including automated software testing, distributed workplace collaboration and video conferencing, wireless applications and technology, video multicasting, and data over cable networks.

Software Engineer, Bellcore
Rapid Applications Group

1998
Piscataway, NJ

As lead engineer for subsystem enhancement and product support of a large-scale, *actively deployed* service provisioning system for regional Bell operating carriers, my responsibilities included drafting requirements and documentation, development and build management, and acting as client liaison for an assortment of support and enhancement issues.

System Staff, Rutgers University
NASA Grant for AXAF Educational Outreach

1996 - 1998
New Brunswick, NJ

As per a grant awarded to the Physics department at Rutgers University, I helped create an educational website providing information about NASA's Advanced X-Ray Astrophysics Facility (AXAF) targeting students grades K-12.

Honors / Awards

- National Science Foundation Award – Co-authored proposal that builds on my research.
Title: Leveraging Type Systems for High-Assurance Cyber-Physical Systems # 0720604
Program: CISE/CNS Computing Systems Research, 2007.
Award: \$99,999
- Research fellowship, Boston University, 2006 - 2008.
- US DoEd Graduate Assistance in Areas of National Need Fellowship, awarded (declined), 2006.
- High Honors in Computer Science, Rutgers College, 1994 - 1998.
- General High Honors, Rutgers College, 1994 - 1998.
- Dean's List, Rutgers College, 1994 - 1998.
- Golden Key National Honor Society, 1998.

Patents

- Michael Long. *Methods and Systems for Monitoring Quality Assurance.*(Patent #6,754,847; Issued June 22, 2004), Co-Inventors: S.R. Dalal, A. Jain, G. Patton, M. Rathi, J. Appenzeller.
- Michael Long. *A Method and System for Providing Secure, Instantaneous, Directory Integrated, Multiparty, Communications Services.* (Provisional filed 2001), Co-Inventors: S.R. Dalal, G. Patton, R. Graveman, C. Chung, G. Di Crescenzo, H. Shim.

Publications

Book Chapters

[HRTES07] Azer Bestavros and Michael Ocean. *Programming and Virtualization of Distributed Multitasking Sensor Networks.* Insup Lee, Joseph Leung, and Sang Son, editors, Handbook of Real-Time and Embedded Systems, chapter 23. CRC Press, 2007.

Refereed Conference Proceedings

[WISEC08] Michael Ocean and Azer Bestavros. *Wireless and Physical Security via Embedded Sensor Networks.* In Proceedings of the First ACM Conference on Wireless Network Security (WiSec 2008), Pages 131-139, *Best Paper Award*, Alexandria, VA, April 2008.

- [VEE06] Michael Ocean, Azer Bestavros, and Assaf Kfoury. *SNBENCH: Programming and Virtualization Framework for Distributed Multitasking Sensor Networks*. In Proceedings of the 2nd ACM International Conference on Virtual Execution Environments (VEE 2006), pages 89-99, New York, NY, USA, June 2006.
- [BN05] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Michael Ocean. *SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications*. In Proceedings of the IEEE International Workshop on Broadband Advanced Sensor Networks (Basenets 2005), Boston, MA, October 2005.
- [IPTEL01] Hyong Sop Shim, Chit Chung, Michael Long, Gardner Patton, and Sidhartha Dalal. *An Example of Using Presence and Availability in an Enterprise for Spontaneous, Multiparty, Multimedia Communications*. 2nd IP-Telephony Workshop (IPTEL2001), pages. 138-148, April 2001.

Technical Reports

- [TR0802] Michael Ocean, and Azer Bestavros. *Extending the SNBENCH to Support Wireless Network Intrusion Detection*. Technical Report BUCS-TR-2008-002, CS Department, Boston University, January 15 2008.
- [TR0616] Michael Ocean, Assaf Kfoury, and Azer Bestavros. *Integrating Sensor-Network Research and Development into a Software Engineering Curriculum*. Technical Report BUCS-TR-2006-016, CS Department, Boston University, July 14 2006.
- [TR0601] Yarom Gabay, Michael Ocean, Assaf Kfoury, and Likai Liu. *Computational Properties of SNAFU*. Technical Report BUCS-TR-2006-001, CS Department, Boston University, February 6 2006.

Presentations

- ACM Wireless Network Security (WiSec), April 2008.
- ACM Virtual Execution Environments (VEE), June 2006.
- BU Operating Systems and Services (BOSS), January 2006.
- BU Sensorium Research Group, May 2005.

Posters and Abstracts

- National Science Foundation (NSF), Computer and Information Science and Engineering (CISE) Directorate, Computing Research Infrastructure (CRI) program meeting (NSF CRI'07 PI meeting), June 2007.
- BU Center for Information and Systems Engineering (CISE) Sensor Network Consortium (SNC), November 2006.
- Boston University Industrial Affiliates Program (IAP) Research Open House, March 2006, March 2007.

Editorial Services

Reviewer for:

- ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) 2006, 2007, 2008.
- ACM Virtual Execution Environments (VEE) 2006.
- IEEE International Conference on Distributed Computing Systems (ICDCS) 2007.

Research Interests

Systems and Networks

Operating Systems, Distributed Systems, Sensor Networks/Embedded Systems, Real-Time Systems; Architecture and Implementation.

Applied Programming Languages

Type Theory, Domain Specific Languages, Software Design, Automated Verification.

Teaching / Advising Activities

- *Software Engineering 2006 - 2008*; Advising of undergraduate students for research projects and Technical Report publication in their names.
- *Image and Video Computing 2005, 2007*; Provided and supported my research project as a platform for students to implement Computer Vision projects.

Extracurricular Activities

Outside the office I enjoy snowboarding, rock-climbing, hiking and collecting/repairing pin-ball machines.

References

Teaching & Research

Azer Bestavros - Professor, Boston University
(617) 353.9726 – best@bu.edu

Assaf Kfoury - Professor, Boston University
(617) 353-8911 – kfoury@cs.bu.edu

Teaching

Wayne Snyder - Associate Dean for Students, Boston University
(617) 358-2739 – dfs@bu.edu

Margrit Betke - Director of Undergraduate Studies, Computer Science, Boston University
(617) 353-6412 – betke@cs.bu.edu

Research

Ibrahim Matta - Associate Professor, Boston University
(617) 358-1062 – matta@cs.bu.edu

Professional *(contact information available on request)*

Siddhartha Dalal - Vice President and Manager of the Xerox Imaging and Services Technology Center, (Former Chief Scientist, Executive Director, Telcordia Technologies)

Ashish Jain - Director, Advanced Technology Solutions, Telcordia Technologies