# TRUST MANAGEMENT IN DISTRIBUTED RESOURCE CONSTRAINED EMBEDDED SYSTEMS

A Dissertation Presented

by

Peter C. Chapin

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fullfillment of the Requirements
for the Degree of Doctor of Philosophy
Specializing in Computer Science

January, 2014

Accepted by the Faculty of the Graduate College, The University of Vermont, in partial fulfillment of the requirements for the degree of Doctor of Philosophy, specializing in Computer Science.

Thesis Examination Committee:

Christian Skalka, Ph.D. — Advisor

Alan Ling, Ph.D.

Margaret Eppstein, Ph.D.

Jeffrey Frolik, Ph.D. — Chairperson

Cynthia J. Forehand, Ph.D. — Dean, Graduate College

Date: October 25, 2013

# Abstract

Many embedded systems, such as wireless sensor networks, make use of highly resource constrained devices. Security goals for such systems tend to focus on keeping data confidential from outsiders or ensuring data integrity during communication. However as embedded systems from different administrative domains increasingly come into contact, for example via short hop radio links, a need arises for one system to allow partial access to its resources from adjoining systems. This dissertation explores two approaches for providing distributed trust management facilities to resource constrained embedded systems, in particular wireless sensor networks. The first is a direct approach using a secure remote procedure call mechanism called *SpartanRPC*. The second is a staged approach using a two stage programming system called *Scalaness/nesT*. In addition to describing these two approaches this dissertation also presents the results of evaluating them both in test environments and with a realistic application. Both approaches are feasible but the staged approach is far more flexible and, depending on application requirements, more efficient.

# Dedication

***To my wife Sharon for her unwavering support, continuous encouragement, and patient tolerance, and to my parents for showing me the value of education.***

# Acknowledgments

This dissertation would not have been possible without the assistance and guidance of many people. I would especially like to thank my adviser Christian Skalka for many years of valuable feedback. I would also like to thank my collaborators Sean Wang, Scott Smith, and especially Simone Willet and Michael Watson for their invaluable assistance in making the work I describe here a reality. Finally I'd like to thank the faculty and staff of the Department of Computer Science at the University of Vermont for creating an environment that allowed me to flourish.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Embedded systems present difficult programming challenges (Mottola and Picco 2011). For reasons of size, power consumption, disposability, or some combination of these things, embedded devices are often highly resource constrained. For example, a typical device might have only 48 KiB of program ROM, 10 KiB of RAM, and use a small, 16 bit microcontroller running at 8 MHz (moteiv 2006). Yet embedded applications are increasing in complexity and often provide mission-critical or even safety-critical services. Such systems need to be both efficient and correct.

This dissertation specifically looks at the problem of providing distributed trust management in resource constrained embedded systems. Here *trust management* refers to a general approach for authorizing access to resources in an environment where the identity of requesting principals is not known to the authorizer. A trust management system provides a way for the authorizer to define an access policy in terms of arbitrary certified attributes that the requester must possess. Many trust management systems have been described in the literature (Chapin, Skalka, and Wang 2008), and they vary in complexity, expressivity, and mathematical foundations. However, they all attempt to provide a well structured approach to the problem of access control in widely distributed and dynamic

environments.

Trust management systems are typically designed for use by authorizers with resource rich machines such as commercial web servers. Yet there are embedded applications that could also benefit from trust management. For example, "smart cars" that communicate with each other about road conditions (Seepold, Madrid, Gómez-Escalonilla, and Nieves 2009), or body area networks that provide medical monitoring features (Shnayder, Chen, Lorincz, Jones, and Welsh 2005; Chen, Gonzalez, Vasilakos, Cao, and Leung 2011), may encounter many unknown principals during their operation. The security and safety of these applications, and many others, will depend on their ability to distinguish trustworthy principals from unreliable or malicious ones.

For reasons of space and time efficiency, many embedded systems are programmed in low level languages such as C. Programming at that level is complicated and error prone. It is desirable, therefore, to provide programmers with convenient abstractions to shield them from low level complexities. These abstractions should be in the programming language itself, and this dissertation is about providing enriched languages that can address the needs of modern embedded systems in general and the embedded trust management problem in particular. This *language based* approach moves some of the work of producing correct programs to the language compiler and runtime system. Language features can be formally analyzed and rigorously tested once and then applied to many applications. This is in contrast to each application being an ad-hoc construction of customized components with limited use beyond the application for which they were created.

The value of formal foundations cannot be overstated. In critical systems where safety or security is at stake, a rigorous understanding of the mechanisms being used is essential. For example, trust management systems that provide a precisely defined policy language are preferable to systems that use informal methods.

The focus of this dissertation is on a kind of embedded system called a *wireless sensor*

*network* (WSN). Such systems consist of a network of small sensors or actuators that are connected by way of short hop radio links. Commonly such networks include one or more base stations, or "hubs," with wider connectivity that serve as an interface between the sensor network and external systems. Wireless sensor networks are an area of intense study with many envisioned applications ranging from environment, asset, and structural monitoring to emergency response (Culler, Estrin, and Srivastava 2004; Lorincz, Malan, Fulford-Jones, Nawoj, Clavel, Shnayder, Mainland, Welsh, and Moulton 2004). Yet despite the use of sensor networks to demonstrate the systems described herein, the techniques can be used with a wide range of embedded applications.

Two approaches to solving the problem of providing trust management-style distributed authorization in resource constrained embedded systems are discussed here. The first approach is based on a new remote procedure call (RPC) discipline named *SpartanRPC* (Chapin and Skalka 2010; Chapin and Skalka 2013). In this method all trust management computations are done directly on the embedded devices. However, the complexity of the system is hidden from the programmer behind a simple extension to the widely used nesC programming language (Gay, Levis, von Behren, Welsh, Brewer, and Culler 2003). In order to implement this *direct* approach, a compiler called *Sprocket* has been created. Sprocket takes an extended dialect of the nesC language as input and outputs an equivalent program in ordinary nesC. In addition Sprocket outputs the necessary runtime support to process authorization requests and policy statements in the $RT_0$ trust management language (Li, Mitchell, and Winsborough 2002; Li and Mitchell 2003b).

The second approach presented is based on *staged programming* (Taha and Sheard 1997; Sheard and Jones 2002; Mainland, Morrisett, and Welsh 2008; Liu, Skalka, and Smith 2012). In a staged environment, a first stage program is used to compose and specialize a lower level, second stage program. Specialized code can often be considerably optimized. However, flexibility is retained because the first stage program can be re-executed

at a later time to re-specialize the second stage program as needed.

Unlike with many staging systems, the work described here uses stages with different programming languages and that execute on different machines, i.e., in different address spaces. When applied to embedded systems the second (and final) stage must be in an embedded systems language running on the embedded hardware, whereas the first stage need not be as restricted.

This dissertation also describes *Scalaness* (Chapin, Skalka, Smith, and Watson 2013), an extension of Scala (Odersky, Spoon, and Venners 2011) with features that allow the programmer to compose and specialize components written in a reduced dialect of nesC called *nesT*. An important and novel feature of Scalaness is that it extends Scala's type system, so that a well-typed Scalaness program will always generate a well-typed nesT program. This *cross-stage type safety* property means the type correctness of the program that ultimately runs on the embedded device is guaranteed by the first stage Scalaness compiler.

Scala was chosen as the basis for the first stage language largely for pragmatic reasons, primarily to build a system that could be used for real applications. Scala is a rich language that runs on the Java Virtual Machine (JVM) and has access to the Java ecosystem. Also the Scala compiler has a plugin architecture, and it was originally intended to implement Scalaness as a compiler plugin. Unfortunately, as described in chapter 5 that proved difficult and Scalaness was instead implemented as a direct modification to the Scala compiler itself.

## 1.1 Motivation

As an example of an application that illustrates the concepts of trust management in embedded systems, consider a first responder situation in which multiple social entities must

interact and cooperate. Recent work has shown the effectiveness of wireless sensor networks in such scenarios (Gao, Pesto, Selavo, Chen, Ko, Lim, Terzis, Watt, Jeng, Chen, Lorincz, and Welsh 2008; Lorincz, Malan, Fulford-Jones, Nawoj, Clavel, Shnayder, Mainland, Welsh, and Moulton 2004) in their ability to coordinate multiple data collection and communication devices in an ad-hoc, easily deployable fashion. However, data collection and communication in this scenario (and other similar ones) must be a secured resource, due to, e.g., HIPA requirements in the case of medical response. Furthermore, security must be coordinated on-site in a sensor network comprising subnetworks administered separately (police, medical units from different hospitals, etc.), and no prior coordination between administrations can generally be assumed. Trust management authorization is well suited for this kind of scenario.

For instance, if an EMT team deploys a sensor network to monitor patient locations and vital signs, a security policy can be imposed whereby responding police departments can deploy their own sensor network, and through it access patient identity and location data but *not* medical data directly from the EMT network. This direct data access will often be necessary due to real-time constraints and lack of Internet connectivity in emergency situations.

SpartanRPC's ability to do trust management on the network nodes themselves would be invaluable in this scenario. However, Scalaness may also be useful. In the staged case, powerful base stations could communicate perhaps by way of shared files manually carried from one machine to the next. Since the first stage program does not need to execute frequently such sharing could be done while each service provider is setting up at the location of the emergency. Other environmental and security factors could be provided to the first stage program at that time, allowing the node software to be quickly and easily customized for the particular disaster at hand.

More generally Figure 1.1 shows two wireless sensor networks owned by separate ad-

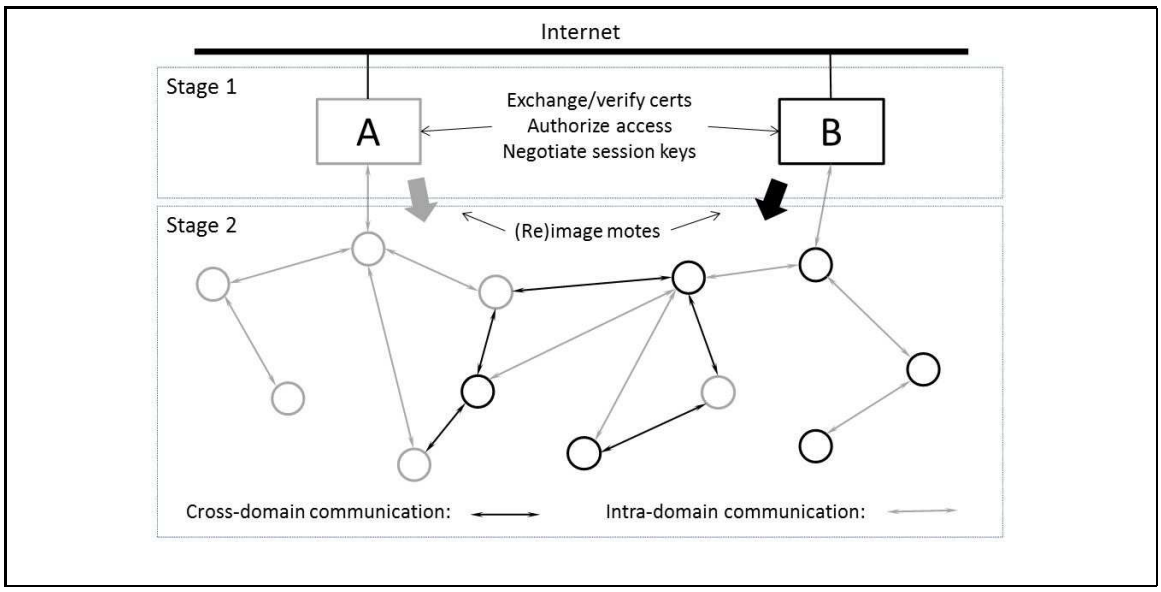


**Figure 1.1:** Motivational Scenario

ministrative domains, $A$ and $B$. The lower part of the figure shows the networks as consisting of multiple sensor nodes. Each node in the networks is an example of a resource constrained embedded system. The two networks overlap in space so that nodes from the two networks can potentially communicate with each other.

In some applications it may be desirable for the networks to share certain information while keeping other information private. As one example, $A$ and $B$ may agree to use each other's nodes for accurate time synchronization to their mutual best interest without wanting to share any other functionality. Alternatively, perhaps the networks are willing to carry data from foreign isolated nodes thus increasing each other's connectivity and enhancing their useful lifetimes, all without being able to access each other's primary functions.

In other scenarios one of the networks, say $B$, may be reduced to a single mobile node that wanders into the field of an established network $A$. In that case $B$ may wish to query $A$ or otherwise interact with it, yet $A$ and $B$ may have no prior association. section 6.1 describes a specific scenario of this type used during the evaluation of the work presented here.

Trust management systems provide exactly the kind of flexible, policy-driven authorization control needed to address these situations. The ability to define access policy for unknown principals, the hallmark of trust management, is particularly important in the case of mobile embedded systems where encountering new principals is routine.

SpartanRPC addresses this problem directly by providing a way for the embedded devices themselves to execute trust management logic. In that case no additional supporting infrastructure is needed but the nodes are required to do extensive computations.

Scalaness, as a staged programming system, requires support beyond the nodes where the first stage program can execute. This additional support is shown on top of Figure 1.1 where Scalaness programs execute on the base stations of $A$ and $B$ to compute node programs for deployment that are specialized with appropriate session keys. The Scalaness programs can communicate over the Internet to share credentials or other security tokens as required.

## 1.2 Security Model

Although many security properties may be of interest to embedded systems applications, only one is the focus of this work: *a system is said to be secure if only authorized users of a resource can access it*. In this context a *resource* could be a physical device on a node, e.g., a sensor or an actuator, or it could be a pure software component providing, e.g., a computation, communication, or storage service. This work is only concerned with access to physical devices via application level software; access via physical attacks or attacks against low level device drivers is not considered in the model used here.

Each resource is presumed to have an *authorizer* who controls access to that resource. A user is authorized for a resource if and only if the authorizer's access policy for that resource grants access.

In the manner of many trust management systems, and in the $RT_0$ system specifically, each principal, also called an *entity*, is represented directly by a public/private key pair. Consequently the requester of a resource does not need to authenticate to the authorizer to prove her identity, nor provide identity-to-key binding certificates. She only needs to prove that she has access to the private key of an authorized public/private key pair. Accordingly authorizers define policies in terms of the (public) keys themselves. While keys can be given names, such names are purely for local convenience; they have no significance to the security of the system and need not be shared.

An important consequence of the lack of identity-to-key bindings is that impostor keys can not be created. If an attacker generates a new public/private key pair, it would be regarded as an entirely new entity. Access would be evaluated based on that entity's certified attributes (if any). It is not possible for a "bogus" principal to pose as a legitimate principal. As is typical for trust management systems, this moves the problem of associating a specific attribute with the correct key to those who create the attribute certificates. However, those certificates are created off line before resource access is requested and evaluated; requesters and authorizers are not concerned with *how* certificates are created and the means used to produce them is outside the scope of this security model.

In order to improve the efficiency of normal message transfers between a requester and an authorizer, symmetric session keys are computed using a simple Diffie-Hellman key agreement protocol (Diffie and Hellman 2006) as described in section 3.6.1. In this protocol each of two communicating entities compute a common shared secret by combining the public key of its peer with its own private key. No secret information is transmitted over the network and an eavesdropping third party is unable to compute the same shared secret without access to either private key.

Yet for the reasons given earlier, this simple approach is nevertheless not vulnerable to a man in the middle attack. The key used by the authorizer to authorize access is the same

as that used to compute the session key. Consequently only a legitimate user will be able to compute the same session key. Either the man in the middle will not be authorized or else the man in the middle is a legitimate user of the resource anyway.

A man in the middle would be able to pose as a legitimate service from the point of view of the requester. However, requester messages are not considered secret so an eavesdropper could read them in any case. Since the systems described here use only unidirectional communication, requesters that wish to receive results from an authorizer must provide a suitable service of their own for authorizers to use. In that case the roles of requester and authorizer are reversed, and the service provided by the original requester could be protected by an appropriate policy to prevent unauthorized nodes from returning fake results.

Ultimately the session key is used to compute a message authentication code (MAC) on requester messages. Verification of this MAC proves that the requester is in possession of a session key that was previously computed using an authorized public key. The MAC on request messages then serves to verify authorization; *no unauthorized user can compute a valid MAC.*

Other security properties are not directly supported by this work. Notably, neither SpartanRPC nor Scalaness address the issue of node tampering or denial of service attacks. Both systems as described here are also vulnerable to various kinds of replay attacks. Issues of data confidentiality are also outside the immediate scope of this work.

However, SpartanRPC and Scalaness do not interfere with the addition of other security services to an application. For example, an application specific protocol that adds a monotonic counter to messages could be layered on top of either system to protect against replay attacks. Scalaness, in particular, could be used to support such an approach by letting a first stage program compose and specialize the mechanism, choosing appropriate parameters at first stage execution time. Confidentiality of messages could also be added by encrypting message contents with the previously negotiated session key.

As usual it is assumed that the cryptographic primitives used by both systems are secure in the sense that it is computationally infeasible for any attacker of interest to defeat the cryptographic protections directly.

## 1.3 Related Work and Contributions

The first trust management systems were inspired by early foundational work in authentication logics such as BAN (Burrows, Abadi, and Needham 1990) and authorization logics such as ABLP (Abadi, Burrows, Lampson, and Plotkin 1993). However, the concept of trust management as an independent area of study was first introduced with PolicyMaker (Blaze, Feigenbaum, and Lacy 1996; Blaze, Feigenbaum, and Strauss 1998). PolicyMaker policies are implemented as arbitrary programs in a suitable "safe" programming language. This gives the system great flexibility but also introduces intractability.

KeyNote (Blaze, Feigenbaum, Ioannidis, and Keromytis 1999) is a direct descendant of PolicyMaker. KeyNote restricts PolicyMaker by specifying a limited language for creating policies. However, a full analysis of KeyNote's policy language (Li and Mitchell 2003a) shows that certain authorization problems nevertheless remain undecidable. KeyNote has been used to enforce IPsec security requirements (Blaze, Ioannidis, and Keromytis 2002; Blaze, Ioannidis, and Keromytis 2003).

SDSI/SPKI (Rivest and Lampson 1996; Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonen 1999) provides a relatively simple, yet expressively interesting trust management language that is a precursor to the $RT_0$ system used here. The semantics of SDSI/SPKI have been analyzed by several authors (Abadi 1998; Halpern and van der Meyden 1999; Howell and Kotz 2000; Li 2000; Clarke, Elien, Ellison, Fredette, Morcos, and Rivest 2001) making it one of the best studied trust management systems. SDSI/SPKI has been used to provide security in component based programming language design (Liu and Smith 2002).

QCM (Gunter and Jim 1997; Gunter and Jim 2000a) and its successor SD3 (Jim 2001; Jim and Suciu 2001) cast distributed authorization as a kind of distributed database problem. As a result, these systems are able to leverage well-studied database techniques and abstractions. These systems reveal a deep and interesting connection between authorization logics and database theory that inspired later work with database query languages such as Datalog and Datalog$_C$ (Li and Mitchell 2003a).

Other notable examples of trust management systems include Cassandra (Becker and Sewell 2004), a system that has been studied in the context of the United Kingdom's proposed nationwide electronic health records database. Also the Extensible Access Control Markup Language (XACML) (OASIS 2006a) and the Security Assertion Markup Language (SAML) (OASIS 2006b), define XML policy and assertion languages that make use of many trust management concepts.

While there has been a great deal of research on security in sensor networks, much of that work has focused on low level concerns such as link layer security, key distribution (Çamtepe and Yener 2005), and secure network protocols (Gupta, Millard, Fung, Zhu, Gura, Eberle, and Shantz 2005; Fouladgar, Mainaud, Masmoudi, and Afifi 2006). Systems such as TinySec (Karlof, Sastry, and Wagner 2004) and MiniSec (Luk, Mezzour, Perrig, and Gligor 2007) are based on shared secrets and generally assume that an entire network comprises a single security domain. Furthermore, these systems support confidentiality and integrity properties, but not access control.

Extending sensor network software platforms with support for secure interactions between domains has been studied in previous research on SSL for sensor networks (Jung, Hong, Ha, Kim, and Kim 2009). However, that work was focused on extending the Internet to sensor networks (aka "IP for WSNs"), whereas SpartanRPC is a more general system for enhancing secure communications *within* a sensor network. Research on sensor network security has also addressed secure routing (Karlof and Wagner 2003), cryptogra-

phy (Bertoni, Breveglieri, and Venturi 2006), and hardware issues (Perrig, Stankovic, and Wagner 2004). In contrast to these low-level systems, SpartanRPC provides language-level abstractions for secure RPC services.

More closely related is a system for establishing fine-grained, "node-level" policies in sensor networks (Claycomb and Shin 2011). However, this work is more focused on group-based key negotiation and distribution, and while it does offer a policy language, it is rooted in implementation details and not as a separable specification. Also, that work does not provide a language API for integrating their system into secure applications as does SpartanRPC.

Previous related work also illustrates interest in and useful applications of RPC in embedded networks. For example, the Marionette system uses network layer RPC for remote (PC-based) analysis and debugging of sensor networks (Whitehouse, Tolle, Taneja, Sharp, Kim, Jeong, Hui, Dutta, and Culler 2006). The Fleck operating system provides a small pre-defined set of RPC services for sensor network applications, while the trustedFleck system extends this with a form a secure RPC (Hu, Corke, Shih, and Overs 2009; Hu, Tan, Corke, Shih, and Jha 2010). S-RPC provides an RPC facility for sensor networks that allows remote services to be added to the system dynamically (Reinhardt, Mogre, and Steinmetz 2011). SpartanRPC differs from these systems in that it extends the nesC programming language (unlike trustedFleck) to allow programmer definition of secure RPC services (unlike S-RPC) that can be accessed by nodes within the network itself (unlike Marionette). SpartanRPC is similar to, and inspired by, TinyRPC (May, Dunning, Dowding, and Hallstrom 2007). TinyRPC, however, does not provide security and has different semantics that are not as expressive as SpartanRPC's approach. In particular, SpartanRPC allows asynchronous invocations to be sent to a dynamically selected subset of neighbors.

TeenyLIME allows application programs to access an abstract "tuple space" that is the union of tuple spaces on the local node and the immediately neighboring nodes (Costa,

Mottola, Murphy, and Picco 2007). This provides an alternative to RPC for uniformly accessing remote and local data. However, interaction with the middleware is by way of a dedicated API; there is no attempt to provide a true RPC mechanism. Also TeenyLIME does not address issues of access control.

Secure Middleware for Embedded Peer to Peer systems (SMEPP) is a general framework for creating security sensitive applications from a distributed network of embedded peers (Brogi, Popescu, Gutiérrez, López, and Pimentel 2008). SMEPP Light (Vairo, Albano, and Chessa 2008) is a reduced version of SMEPP to address the resource constraints of wireless sensor networks. SMEPP Light provides a publish/subscribe communication model using directed diffusion (Intanagonwiwat, Govindan, Estrin, Heidemann, and Silva 2003) to distribute "events" to all subscribers and symmetric key cryptography to provide confidentiality and data integrity within a group of nodes. However, SMEPP Light is not integrated into a programming language and does not provide a remote procedure call mechanism. Furthermore, SMEPP Light only supports a simple model of access control based on group membership.

High level macroprogramming languages such as Kairos (Gummadi, Gnawali, and Govindan 2005), and Regiment (Newton, Morrisett, and Welsh 2007) provide a way to program the entire network as a single entity. These systems attempt to hide not only the inter-node communication from the programmer, but also the entire node level programs. SpartanRPC operates at a much lower level making it potentially more flexible and also, unlike these macroprogramming systems, SpartanRPC addresses access control issues in networks containing multiple security domains.

Whole network programming of wireless sensor networks has also been investigated using mobile agents in systems such as Agilla (Fok, Roman, and Lu 2009) and Wiseman (González-Valenzuela, Chen, and Leung 2010). However, like the macroprogramming systems mentioned previously, neither of these systems address issues related to access

control in the presence of multiple security domains.

The potential of applying staged metaprogramming techniques to sensor networks was explored in the functional sensor language Flask (Mainland, Morrisett, and Welsh 2008). Flask allows functional reactive programming (FRP)-based stream combinators to be precomputed before network deployment, but it is possible to generate ill-typed Flask object code since cross-stage static type checking is not performed. Hume (Hammond and Michaelson 2003) is a domain specific language for real-time embedded device programming. It includes a metaprogramming layer but that layer is more like nesC's configuration files in that there is a very restricted syntax for a few special metaprogramming operations including component wiring, macros, and code templating.

MetaML (Taha and Sheard 1997; Taha 2004) and MetaHaskell (Mainland 2012) are foundations the work described here builds on. MetaHaskell does support heterogeneous language staging where the lower stage language is defined by a plug-in and several instantiations have been defined including one for a low-level C-like language. Like this dissertation's approach, they guarantee type safety of all lower stage code produced. They use a more traditional metaprogramming model, however, not the *process separation* model needed for embedded systems metaprogramming where different stages execute on different machine architectures and in different address spaces. Also neither MetaML nor MetaHaskell address the issues of metaprogramming module composition and type specialization. In contrast, Scalaness follows the foundational work on Framed ML ($\langle$ML$\rangle$) (Liu, Skalka, and Smith 2012); subsection 4.3.5 discusses how it serves as the theoretical underpinning of the Scalaness system.

Lightweight Modular Staging (Rompf and Odersky 2010) describes a method of expressing staged computations using a Scala host framework without any compiler modifications. The approach allows cross-stage type safety but does not support *dynamic type construction*, a method by which second stage types can be manipulated as first stage val-

ues. This feature provided by Scalaness is important for optimizing the layout of data structures by tuning the types used for their members.

Actor based sensor metaprogramming has been studied in (Cheong 2007); this work also focuses on high level dynamic reprogrammability but is untyped. More broadly, metaprogramming is known to be useful for increasing the efficiency of systems applications. One example is Tempo (Consel, Hornof, Marlet, Muller, Thibault, Volanschi, Lawall, and Noyé 1998), a system that integrates partial evaluation and type specialization for increasing efficiency of systems applications. Ur (Chlipala 2010) allows for type safe metaprogramming for web applications.

The units of staged code composition in nesT are *modules*. Countless different module systems exist, but they are primarily designed to achieve separate compilation and sound linking (Cardelli 1997).The different design goals of nesT lead to different design choices in nesT modules. For example, data crossing nesT module boundaries needs to conform to the property of process separation, a non-issue in standard module system designs. In addition, nesT modules allow values/types across the boundary of modules to be flexibly constructed, including dynamic construction of types. Module systems such as ML modules (MacQueen 1984) and Units (Flatt and Felleisen 1998) allow types to be imported/exported as Scalaness supports. However, there are several features of ML modules including type hiding that Scalaness does not aim to support.

NesT modules are more expressive in their support of first class modules as values and the possibility of dynamic construction of "type exports." That said, first class modules are not new (Mitchell, Meldal, and Madhav 1991; Ancona and Zucca 2002). The novelty of nesT arises in its application to program staging and the incorporation of dynamic type construction.

The type parametricity of System F and $F_{\leq}$ (Cardelli and Wegner 1985), and the practical type systems it inspired such as Java's generics, do not treat types as first class values.

C++ templates support types as meta values in template expansion, but type safety of generated code is not guaranteed without full template expansion. Concepts (Gregor, Järvi, Siek, Reis, Stroustrup, and Lumsdaine 2006) improves on this, but types are still not first class values.

### 1.3.1 Summary of Contributions

The main contributions of this work include a demonstration, for the first time, of the feasibility of using trust management in resource constrained embedded systems such as sensor networks. No previous work has attempted to implement support for such a general and flexible authorization system on such small devices. Previous work on sensor network security has tended, instead, to focus on low level issues, e. g., providing confidentiality on links, or on relatively ad-hoc solutions to specific problems, e. g., key distribution.

In contrast, the SpartanRPC language, together with its implementation in Sprocket, provide language based support for general purpose distributed authorization in sensor networks. Trust management authorization can now be used as a primitive for building more elaborate security services in complex embedded networks spanning multiple security domains where fine grained access control is required. SpartanRPC also provides a new remote procedure call discipline for nesC featuring asynchronous invocations and an ability to dynamically control the subset of neighbors to which each remote invocation is directed.

This work also introduces Scalaness/nesT as a well founded, two stage programming system for developing general embedded applications. The foundations of Scalaness are presented in the distilled languages DScalaness and DnesT. An implementation of the system has been created by modifying the open source Scala compiler.

Scalaness offers a unique combination of staging with cross-stage type safety, process separation, and dynamic type construction that make it a powerful tool for creating flexible

and efficient embedded applications. Scalaness treats the trust management problem as simply one application of many, and this work demonstrates Scalaness by using it to re-implement a solution to embedded trust management in sensor networks.

This work also describes the use of both SpartanRPC and Scalaness on a realistic field example with non-trivial application requirements. This example shows that both systems can be used to solve real-world problems and are not merely toy systems of theoretical interest only.

## 1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Trust management systems are described, in general, in chapter 2, outlining the different features provided by common trust management systems and motivating their use. Special focus is given to the $RT$ family of trust management systems used in this work. The design of SpartanRPC is described, as well as the details of its implementation, in chapter 3. Of particular note is the description of the added support for $RT_0$ trust management to a general RPC mechanism. Scalaness and nesT are introduced in more detail in chapter 4, and then the syntax and semantics of both languages are described, using simplified "distilled" versions of those languages called *DScalaness* and *DnesT*. The implementation of the practical Scalaness/nesT system is described in chapter 5, relating the features of the implementation to the earlier foundational presentation. An evaluation of both systems is provided in chapter 6 using simple test programs in the context of a realistic field example. The conclusion is documented in chapter 7. Finally, in Appendix A the full source code of a simple Scalaness/nesT sample is demonstrated, with commentary.

# Chapter 2

# Trust Management

Distributed applications that span administrative domains have become commonplace in today's computing environment. Electronic commerce, high performance scientific computing, groupware, and multimedia applications all require collaborations between distinct social entities. In such systems each administrative domain, also called a security domain, controls access to its own resources and operates independently of other administrative domains. The problem of how to best specify and implement access control in such an environment has been a topic of considerable research. To address this problem the concept of trust management was introduced (Blaze, Feigenbaum, and Lacy 1996).

Most existing embedded applications entail only a single administrative domain that owns the embedded devices. Security in that context is mostly concerned with preventing access by outsiders. However some applications have been described, such as scenarios involving emergency response (Lorincz, Malan, Fulford-Jones, Nawoj, Clavel, Shnayder, Mainland, Welsh, and Moulton 2004), that could easily benefit from a facility that allowed multiple domains to interact in a controlled manner. As embedded systems in general, and sensor networks in particular, become more pervasive, situations where multiple domains interact will become more common. Devices in several domains will then be motivated to

use each other's resources in an effort to increase their efficiency, functionality, or lifetime. Hence the need for fine-grained application level access control will increase.

At the heart of all trust management systems is the *authorization procedure*, which determines whether access to a resource should or should not be granted based on a number of conditions. While a number of techniques have been proposed to characterize authorization in trust management systems, the most promising are those based on rigorous formal foundations. This argument is not new, in fact it has motivated trust management research since its inception (Woo and Lam 1993). When security is at stake it must be possible to specify policies in a precise, unambiguous way, and to have confidence that those policies are correctly enforced. Formally well founded trust management systems achieve this, providing a setting in which reliability can be rigorously established by mathematical proof. In particular, various logics have served as the foundation for trust management (Abadi 2003; Bertino, Catania, Ferrari, and Perlasca 2003).

It is important to clearly distinguish between *authorization* and *authentication*. The latter addresses how to determine or verify the identity of principals in a transaction. Authorization, on the other hand, is about what the principals are permitted to do once their identities are known. Although any real implementation of an authorization system will rely on authentication to establish identities, and key-to-identity bindings may even have an abstract representation in the system, authorization generally treats authentication and (public) key infrastructure as orthogonal issues.

Authorization in trust management systems is more expressive than in traditional access control systems such as role based access control (RBAC) (Sandhu, Coyne, Feinstein, and Youman 1996). In those simpler models, an assumption is made that all principals are known to the authorization procedure a priori. Access is based on the identities of authenticated principals. But in a distributed environment creating a single, local database of all potential requesters is untenable. Where there are multiple domains of administra-

tive control, no single authorizer can be expected to have direct knowledge of all users of the system. For example, a sensor network owned by a university might want to provide access not only to the university's students, but also to visiting professors, guest lecturers, and other entities known to cooperating institutions.

Finally, basing authorization purely on identity is not a sufficiently expressive or flexible approach, since security in modern distributed systems often utilizes more sophisticated features (e.g., delegation) and policies (e.g., separation of duty (Simon and Zurko 1997)). These issues are addressed by the use of trust management systems.

## 2.1 Components of Trust Management Systems

Trust management systems in practice comprise a number of functions and subsystems, which can be divided into three major components: *the authorization decision*, *certificate storage and retrieval*, and *trust negotiation*. The authorization decision is where the semantics of the trust management system are made manifest by way of some core logical structure. Certificate storage and retrieval is relevant to the physical location of certificates that are representations of access control elements such as credentials and policies. For example, systems have been proposed for storing SPKI certificates using DNS (Nikander and Viljanen 1998) and for storing SDSI certificates using a peer-to-peer file server (Ajmani, Clarke, Moh, and Richman 2002). Trust negotiation (Winsborough, Seamons, and Jones 2000; Yu, Ma, and Winslett 2000; Seamons, Winslett, and Yu 2001; Yu, Winslett, and Seamons 2001; Winsborough and Li 2002; Winsborough and Li 2004) is necessary for access control decisions where some elements of access policies or the credentials used to prove authorization with those polices should not be arbitrarily disclosed. For example, in (Winsborough, Seamons, and Jones 2000) a scheme is proposed whereby access rights held by requesters are protected by their own policies, and both authorizers and requesters

must show compliance with policies during authorization, i.e., they must negotiate.

The importance of these other components notwithstanding, the discussion in this chapter focuses on authorization decisions. This is because the authorization decision is the basis of any trust management system. Furthermore, not all the systems proposed in the literature have been developed sufficiently to include certificate storage implementations or trust negotiation strategies. Finally, the applications of trust management described in this dissertation do not use a formal approach for certificate handling nor any trust negotiation.

### 2.1.1 Structure of an Authorization Decision

The authorization decision component of a trust management system includes more than just a core authorization semantics. The word *system* here is defined as the set of components that provide an implementation, not just an abstract specification of the authorization semantics. In this section, the components of a generic authorization decision are identified and its structure is characterized.

Figure 2.1 illustrates the structure of an authorization decision. This graphic is meant as a rough sketch, not a formal specification, and not all trust management systems contain all the components described. The graphic is read from top to bottom, and shows the flow of information through a particular authorization process, with output computed in response to an authorization request. The diagram is intentionally vague about the nature of the output: in the simplest case, the output is a simple "yes" or "no" decision as to whether or not to grant resource access, but in systems that support trust negotiation, the output could be a partial answer that provides direction for additional input. Within the scope of this dissertation, focus is concentrated on the case where the output is a boolean value, hence the terminology authorization *decision*. The core authorization semantics $L$ implement the authorization decision, and may be a specialized inference system, or a proof search in a

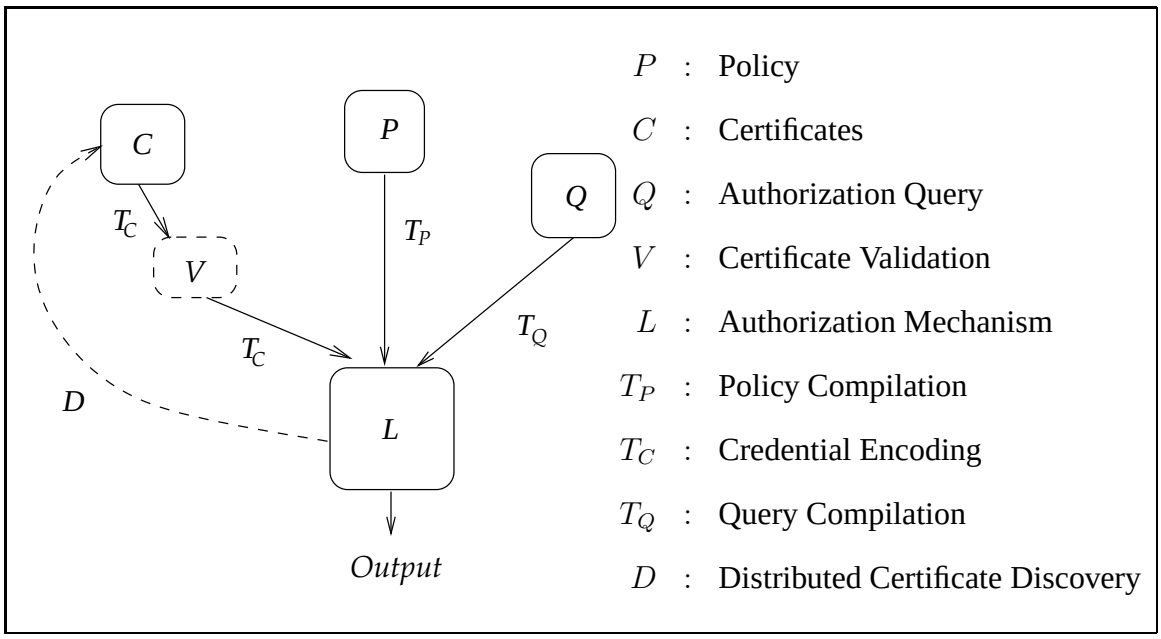


**Figure 2.1:** Structure of an Authorization Decision

generic programming logic such as Prolog, for example. The authorization semantics takes as input parameters from $C$, $P$, and $Q$, which are now described in detail.

Local policy $P$ is defined in some specification language, that is transformed into terms understood by the core semantics by the transformation function $T_P$. This translation may just consist of parsing from concrete to abstract syntax, or $T_P$ may compile statements in a high-level policy language into lower level terms for the core semantics. For example, TPL (Herzberg, Mass, Michaeli, Naor, and Ravid 2000) provides an XML-based "trust policy language" that is compiled into Prolog.

Credentials for a particular requester may be defined as part of local policy. An earmark of trust management systems, however, is their ability to extend local policies with credentials conferred by non-local authorities. This is realized as a set of available certificates $C$ that are transformed by a function $T_C$ into credentials defined in terms understood by the core semantics. The transformation $T_C$ provides a level of indirection allowing systems to choose between various certificate wire formats and PKIs such as X.509 (International

Telecommunications Union 2000) or WS-Security (OASIS 2006c). When working with resource constrained embedded systems it is desirable to use certificate formats that are as compact as possible, as described in section 3.6.1, but that does not affect the behavior of the trust management system.

The transformation $T_C$ also has special significance for the semantics of trust management systems, since it is often not a straight parsing or compilation procedure. Rather, certificates may be rejected, or their credential representations enhanced, by certificate validity information. Validity information is external to the authorization semantics in some systems, but internal to it in others, so the certificate validation component of the authorization decision $V$ is represented as a dashed box.

For example, any given certificate $c \in C$ almost always defines a finite lifetime for the certification, also called a validity interval (Winslett, Ching, Jones, and Slepchin 1997). Some trust management systems such as PCA (Bauer, Schneider, and Felten 2002) support lifetime information in the authorization semantics, and in such a case $T_C$ can map the lifetime information in $c$ to its credential representation. However, other systems do not represent lifetimes in the authorization semantics per se (that is, in $L$), and in such cases the onus is on $T_C$ to filter out expired certificates. For example, SPKI provides a mechanism for certificates to be checked on-line to see if they have been revoked (Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonen 1999), but this mechanism is not part of SPKI's formal structure. This means on the one hand SPKI's revocation policy cannot be expressed in the SPKI policy language itself, nor enforced by its authorization semantics. On the other hand it allows a SPKI implementation to apply a different revocation policy without changing their underlying logical structure, and in general the difficulties associated with formalizing certificate revocation (Stubblebine 1995; Stubblebine and Wright 1996; Rivest 1998a) can be avoided, while a means for certificate revocation in the system is still available.

In addition to policy $P$ and certificates $C$, the authorization decision takes as input a

question or goal $Q$ that is specialized for a particular access request. As an example, some trust management systems, such as SDSI and $RT_0$ (Li, Mitchell, and Winsborough 2002; Li and Mitchell 2003b), define roles. These systems allow one to prove that a particular principal is in a particular role. Resources are associated with roles, and the authorization decision is based on whether the requester is a member of the relevant role. The transformation $T_Q$ translates the goal into terms understood by the core semantics. Finally, the core semantics combines policies and credentials established by input certificates to determine whether the authorization goal is satisfied, and outputs "yes" or "no" based on this determination.

However, as denoted by the dotted line, some systems also provide a "feedback" mechanism $D$ between the semantics of authorization and certificate collection. Rather than merely answering "no" outright in case an authorization goal cannot be reached, the system might identify credentials that are missing and attempt to collect them. This functionality is sometimes called *distributed certificate chain discovery* (Li, Winsborough, and Mitchell 2003) or *policy directed certificate retrieval* (Gunter and Jim 2000b).

Whatever the specifics, it is clear that this functionality makes for a more flexible system in terms of certificate distribution and storage, but presents a significant challenge to system designers, particularly in the embedded case where access to Internet resources may be severely limited.

## 2.2 Features of Trust Management Systems

This section both describes and discusses a number of features relevant to many trust management systems and comments on their potential applicability to embedded systems. This is not intended to be an exhaustive listing, but rather to provide a focus on features that are generally considered important for trust management applications.

### 2.2.1 Formal Foundation

Since authorization systems are used in security-sensitive contexts, mathematically precise descriptions of their behavior and formal assurances of their correctness is essential. A variety of formalisms serve as effective foundations for the definition of trust management authorization semantics. The formalisms used can be divided into three main categories: logics, database formalisms, and graph theory.

In the case of trust management systems based on logic, the authorization problem is expressed in terms of finding a proof of a particular formula representing successful resource access, with the policy represented as a collection of suitable axioms. Credentials relevant to a particular decision become additional hypotheses to be used in the proof. Trust management systems based on database formalisms (e.g., relational algebra) see the authorization decision as a query against a distributed database. The certificates issued by a principal contain, in effect, tuples from relations that a principal controls. Trust management systems based on graph theory define the authorization decision in terms of finding a path through a graph. The request is represented by a particular node in the graph. Principals are also graph nodes and the certificates they issue denote edges.

It is not unusual for a particular trust management system to be described by more than one formalism. In fact, some aspects of trust management are more naturally expressed using one formalism or another. Also, Datalog serves as both a database formalism and a programming logic, and several trust management systems, including $RT_0$ that is used in this work, have been specified in Datalog (Li and Mitchell 2003a).

### 2.2.2 Authorization Procedure. Authorization Complexity

Trust management systems differ in exactly how the authorization decision is implemented. In a broad sense this is due to differences in the way the systems are described; systems

using the same style of formalization tend to use similar authorization procedures. This is particularly evident among the systems using programming logics such as Datalog as both their formal foundation and implementation. However, some variations between systems result in significant differences in how authorization is computed even when the underlying formalism is the same, for example, if certificate revocation is present in one system but not another. In some cases no authorization procedure is given; the details of computing authorization is entirely left to the implementers.

The computational complexity of the authorization decision is clearly of practical interest, especially to developers of resource constrained systems. Ideally, authorization should be decidable and tractable, but there is a trade off between the expressiveness of the certificate and policy language and the complexity of the authorization decision. For example, the systems that use Datalog with constraints (Datalog$_{\mathcal{C}}$) can have various levels of computational complexity depending on the constraint domain used (Li and Mitchell 2003a). Yet even trust management systems with undecidable decision procedures can be potentially useful; realistic policies may be decidable even if the general policy language is not. Furthermore, practical implementations can time-out authorization decisions and return a failed access indication in order to avoid problems with non-termination.

For constrained systems the resources required to make an authorization decision is a matter of critical importance. Implementing a full Prolog or Datalog$_{\mathcal{C}}$ interpreter in a small embedded device would seem to be prohibitively difficult. However, choosing a system that is sufficiently limited (while still allowing for sufficiently rich access policies) enables various optimizations that can bring the implementation cost into a reasonable range. This was a major factor in choosing $RT_0$ for the work described here.

### 2.2.3 Public Key Infrastructure (PKI)

It is common for trust management systems to treat keys directly as principals. This creates a conceptually clean design. In contrast, some systems regard the human or machine participants as the principals and encode a relationship between principals and the keys that identify them. In the former case, key bindings are not represented in the authorization semantics, whereas in the latter case they are. Although PKIs underpin the implementation of trust management systems, the question here is: to what extent does a particular trust management system directly concern itself with the details of key management?

### 2.2.4 Threshold and Separation of Duty Policies

Many systems support threshold policies, where at least $k$ out of a set of $n$ entities must agree on some point in order to grant access. Threshold policies are appealing since agreement provides confidence in situations wherein no single authority is trusted by itself. The concept of separation of duty is related to threshold policies. In the case of a separation of duty policy, entities from different sets must agree before access is granted.

As an example, a bank might require that two different cashiers approve a withdrawal (same set—threshold policy). The bank might also require that a cashier and a manager, who are not the same person, approve a loan (different sets—separation of duty policy). In general threshold policies and separation of duty policies cannot be implemented in terms of each other, although some trust management systems provide support for both (Li, Mitchell, and Winsborough 2002).

### 2.2.5 Local Name Spaces

It is desirable for trust management systems to allow each administrative domain to manage its own name space independently. Requiring that names be globally unique is problematic

and, in general, infeasible. Although there have been attempts at creating a global name space (International Telecommunications Union 2001), these attempts have at best only been partially successful. The ability to reference non-local name spaces is also a keystone of modern trust management, in that it allows local policy to consider requesters that may not be directly known to the local system.

### 2.2.6 Role-Based Access Control

In a large system with many principals it is often convenient to use role based access control (RBAC) (Ferraiolo and Kuhn 1992; Sandhu, Coyne, Feinstein, and Youman 1996). In such a system *roles* are used to associate a group of principals to a set of permissions. The use of roles simplifies administration since the permissions granted to a potentially large group of principals are defined in a single place. RBAC is a conceptual foundation of many modern authorization technologies.

Some trust management systems support RBAC by casting the access control decision as a role membership decision. Access will be granted if the requester is a member of an appropriate role but the precise meaning of the roles, in terms of the permissions that are connected to them, is defined outside the trust management environment. In contrast some trust management systems include a mechanism in their policy language to define permissions explicitly. In these systems the access control decision is directly rendered for a particular permission. Finally, in some cases roles are not provided directly but can be simulated by assigning an appropriate interpretation to suitable objects within the system.

### 2.2.7 Delegation of Rights

All trust management systems allow an authorizer to delegate authority. In other words, an authorizer can specify third parties that have the authority to certify particular attributes.

This is one of the defining characteristics of a trust management system. However, in many applications a requester will also want to delegate some or all of his or her rights to an intermediary who will act on that requester's behalf.

Delegation of rights is important in a distributed environment. For example, a request may be made to an organization's front end system that accesses internal servers where the request is ultimately processed. The classic three-tier architecture of web applications follows this approach. In many environments the back end servers may have their own access control requirements, in which case the requester will need to delegate his or her rights to the front end system for use when making requests to the internal servers.

Trust management systems differ in their support for rights delegation. Delegation credentials may be formally provided, or delegation can be simulated via more primitive forms. Also, delegation *depth* can be modulated in some systems—rather than being purely transitive, delegation of rights may only be allowed to be transferred between fixed $n$ principals. In some cases rights can be delegated arbitrarily or not at all. A system that has this latter feature is said to support boolean delegation depth.

### 2.2.8 Certificate Validity

Since an authorizer receives certificates from unknown and potentially untrustworthy entities, the validity of those certificates must be checked. Usually, signatures must be verified and the certificate must not have expired, since in practice certificates will almost always have a finite lifetime in order to ensure that obsolete information cannot circulate indefinitely. In some systems certificate validity is explicitly treated as part of the structure of the trust management authorization semantics—the component $L$ described in subsection 2.1.1. In such cases certificate lifetimes can be directly represented in credentials and taken into account in policy (Bauer, Schneider, and Felten 2002; Li and Feigen-

baum 2002; Skalka, Wang, and Chapin 2007).

In other systems, certificate validity is defined externally and checked as part of the translation of certificates into credentials—the component $T_C$—and not formally reflected in the authorization semantics (Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonen 1999). Note that it is a topic of debate whether authorizers (Rivest 1998a) or certificate authorities (McDaniel and Rubin 2001) should determine validity intervals for authorization decisions.

### 2.2.9 Credential Negation

Policy languages sometimes allow policy makers to specify that a credential *not* be held. For example, access to a resource may require that requesters not possess a credential endowing them with a felon role. In systems using logic as a foundation for the semantics of authorization, this is expressed as credential negation. That is, authorization is predicated on the negation of a role attribute expressed as a credential. Note that this makes the semantics nonmonotonic—as more credentials (facts) are added to the system, it is possible that fewer authorizations succeed. As noted in (Seamons, Winslett, Yu, Smith, Child, Jacobson, Mills, and Yu 2002), this makes credential negation a generally undesirable feature, since nonmonotonic systems are potentially unsound in practice. For instance, if a certificate is not discovered due to a network failure, access might be granted that would otherwise have been denied. In the embedded environments considered here, this is a major concern.

Monotonicity also allows undecidable (or intractable) authorization logics to be used safely. An authorizer could simply abort an excessively long running computation and deny access. While this approach might prevent some legitimate requests from succeeding, in a monotonic system it remains sound since it would never grant access inappropriately.

### 2.2.10 Certificate Revocation

Certificate revocation is similar to credential negation, but allows previously granted access rights to be explicitly eliminated (Rivest 1998a). Like certificate validity, this can be implemented in the translation $T_C$ from certificates to credentials. For example, in SPKI/SDSI (Ellison, Frantz, Lampson, Rivest, Thomas, and Ylonen 1999) online revocation lists can be defined that filter out revoked certificates prior to conversion to credentials for the authorization decision. At first glance it may appear that certificate revocation entails nonmonotonicity. However, it has been demonstrated that certificate revocation can be encoded monotonically in both the Proof Carrying Authorization framework (Bauer, Schneider, and Felten 2002) and a logic-based PKI infrastructure (Li and Feigenbaum 2002). The technique points out a relation between certificate revocation and certificate validity, in that monotonic revocation can be based on lifetimes and the requirement to renew certificates. Various high-level approaches to, and nuances of, certificate revocation are discussed in (Rivest 1998a).

### 2.2.11 Distributed Certificate Chain Discovery

Where do certificates for a particular access request come from? In common scenarios the requester presents all relevant certificates when requesting access. It is also easy to imagine settings in which authorizers maintain local databases of certificates. In fact, these are the scenarios that are assumed in this work.

More generally, however, certificates could be stored anywhere in the network, as long as the local system has some way of finding them. Of course, given the potentially enormous number of certificates on the network, it is necessary to define some means of selectively retrieving only certificates that might pertain to a particular authorization decision. Formally well founded techniques for doing distributed certificate chain discovery have

been described in (Li, Winsborough, and Mitchell 2003; Gunter and Jim 2000b).

## 2.3 Foundations of Authorization

This dissertation's focus is specifically on trust management systems that use a programming logic as their formal foundation. This approach has the advantage that a specification of the system's semantics can also serve, in principle, as its implementation. In addition, programming logics such as Prolog and Datalog are a well studied and well understood formalism.

Programming logics provide useful abstractions for authorization semantics. They have served as target languages for the compilation of higher-level authorization languages (Li and Mitchell 2003a; Woo and Lam 1993), as well as the foundation for enriched authorization languages (Li and Mitchell 2006; Jim 2001; DeTreville 2002; Li, Mitchell, and Winsborough 2002; Li, Grosof, and Feigenbaum 2003), and have been used for the formalization and study of trust management systems (Li and Mitchell 2006; Polakow and Skalka 2006).

Both Prolog and Datalog are *Horn-Clause* logics, in which all formulae are restricted to the form *head* $\leftarrow$ *body*, where $\leftarrow$ is a right-to-left implication symbol, *head* is a proposition, and *body* is a conjunction of propositions. If variables $X$ appear in a rule, the rule is implicitly universally quantified over those variables. The head of each rule is the consequent of the body. If *body* is empty then the rule is a *fact*.

As a simple example of how logics can apply in a trust management framework, imagine that delegation should be transitive. Suppose that $delegation(X, Y)$ is defined to mean that the rights of $X$ have been delegated to $Y$. Suppose also that $cert(X, Y)$ represents a delegation certificate passing rights directly from $X$ to $Y$. The following Horn clauses

obtain transitivity of delegation:

$$delegation(X, Y) \leftarrow cert(X, Y) \qquad delegation(X, Y) \leftarrow cert(X, Z),\ delegation(Z, Y)$$

Letting $a, b, c, ...$ denote constants, the following represents a collection of delegation certificates:

$$cert(a, b) \qquad cert(b, c) \qquad cert(b, d) \qquad cert(c, e)$$

From these facts and the definition of $delegation$, the query $delegation(a, e)$ will succeed while $delegation(d, e)$ fails.

Datalog was developed as a query language for databases. It is not a full programming language. In contrast Prolog is Turing complete and thus more expressive than Datalog. This extra expressivity is useful in certain contexts. For example, a full-featured authorization logic called Delegation Logic has been defined as a strict extension of Datalog at a high level, that is ultimately compiled to Prolog for practical implementation (Li, Grosof, and Feigenbaum 2003).

Nevertheless, Datalog has certain advantages in the authorization setting: the combination of monotonicity, a bottom-up proof strategy, and Datalog's *safety condition* (any variable appearing in the head of a rule must also appear in the body) guarantee program termination in polynomial time. In contrast, Prolog's top-down proof search can cause non-termination in the presence of cyclic dependencies. For example, if we added the certificate $cert(e, b)$ to the above fact set, some queries would not terminate. This problem is resolved by *tabling* as in XSB (XSB Inc. 2006), but it has been argued that this solution adds too much size and complexity to the implementation for authorization decisions in general (Li, Mitchell, and Winsborough 2002), and for embedded systems particularly. And while Datalog is not capable of expressing structured data, Datalog with constraints (Datalog$_C$), a restricted form of constraint logic programming (Jaffar and Maher 1994), has

been shown sufficiently expressive for a wide range of trust management idioms (Li and Mitchell 2003a).

## 2.4 The RT Trust Management System

This section provides a detailed review of the $RT$ family of trust management systems (Li, Mitchell, and Winsborough 2002). Focus is on $RT$ because it is the trust management system used in the demonstration applications. Although the choice of $RT$ was largely arbitrary, it offers an effective combination of expressivity, ease of use, and efficient implementability. Also $RT$ has a strong formal foundation based on Datalog and its variants.

$RT$ is actually a trust management framework and not a single trust management system. The systems in the $RT$ framework have varying expressiveness and complexity (Li, Mitchell, and Winsborough 2002; Li, Winsborough, and Mitchell 2003; Li and Mitchell 2003b). $RT$ stands for "role based trust management" because it uses policy and credential statements to associate principals, called *entities* in the $RT$ literature, to roles. The significance of the roles is defined externally. The base system, $RT_0$, provides credential forms for simple role membership as well as indirection roles and intersection roles as described below.

$RT_1$ is an extension of $RT_0$ providing parameterized roles. $RT_1^C$ further extends $RT_1$ to allow for the description of structured resources (Li and Mitchell 2003a; Li and Mitchell 2003b). The system $RT^D$ provides a mechanism to describe the delegation of rights and role activations, and $RT^T$ provides support for threshold and separation of duty policies. $RT^T$ and $RT^D$ can be used in combination with $RT_0$, $RT_1$, or $RT_1^C$ to create trust management systems such as $RT_0^T$, $RT_1^{TD}$, and so forth. A rich complexity analysis has also been developed for the $RT$ framework for problems beyond simple authorization, e.g., role inclusion and role membership bounds (Li, Mitchell, and Winsborough 2005).

### 2.4.1 Features

The $RT$ framework represents entities as public keys and does not attempt to formalize the connection between a key and an identity. The $RT$ framework allows each entity to define roles in a name space that is local to that entity. An authorizer is expected to associate permissions with a particular role; to access a resource a requester must prove membership in the role. In this way the $RT$ framework provides role based access control, but it does not deal with permissions directly in the trust management logic.

To define a role, an entity issues credentials specifying the role's membership. Some of these credentials may be a part of private policy, others may be signed by the issuer and made publicly available as certificates. The overall membership of a role is taken as the union of the memberships specified by all the defining credentials.

Let $A, B, C, \ldots$ range over entities and let $r, s, t, \ldots$ range over role names. A role $r$ local to an entity $A$ is denoted by $A.r$. $RT_0$ credentials are of the form $A.r \longleftarrow f$, where $f$ can take on one of four forms to obtain one of four credential types:

1. $A.r \longleftarrow E$

   This form asserts that entity $E$ is a member of role $A.r$.

2. $A.r \longleftarrow B.s$

   This form asserts that all members of role $B.s$ are members of role $A.r$. Credentials of this form can be used to delegate authority over the membership of a role to another entity.

3. $A.r \longleftarrow B.s.t$

   This form asserts that for each member $E$ of $B.s$, all members of role $E.t$ are members of role $A.r$. Credentials of this form can be used to delegate authority over the

membership of a role to all entities that have the attribute represented by $B.s$. The expression $B.s.t$ is called a *linked role*.

4. $A.r \longleftarrow f_1 \cap \cdots \cap f_n$

   This form asserts that each entity that is a member of all roles $f_1, \ldots, f_n$ is also a member of role $A.r$. The expression $f_1 \cap \cdots \cap f_n$ is called an *intersection role*.

For all credential forms $A.r \longleftarrow f$, the principal $A$ is called the *issuer* of the credential. A credential is transformed into a certificate when it is signed by the issuer's private key. Recall that the entities are represented by their public keys directly, i.e., the $A$ and $E$ in $A.r \longleftarrow E$ are public keys.

$RT_1$ enhances $RT_0$ by allowing roles to be parameterized. For example, the second credential form above is extended to $A.r(h_1, h_2, \ldots, h_n) \longleftarrow B.s(k_1, k_2, \ldots, k_m)$ where the $h_i$ and $k_j$ are parameters. Role parameters are typed and can be integers, floating point values, dates and times, enumerations, or finite sets or ranges of these datatypes. An $RT_1$ credential is *well formed* if the parameters given to the roles have the right type and if each variable in the credential appears in the body of that credential.

As an example of an $RT_1$ credential (Li, Mitchell, and Winsborough 2002), suppose company $A$ has a policy that the manager of an entity also evaluates that entity. This can be expressed in $RT_1$ using a policy statement such as

$$A.evaluatorOf(?Y) \longleftarrow A.managerOf(?Y)$$

This policy can't be feasibly expressed in $RT_0$ because the role parameters might take on an arbitrarily large number of values. In $RT_0$ individual credentials would be needed for each possible value of the role parameter.

$RT_1^C$ further enhances the expressive power of $RT_1$ by allowing structured constraints to be applied to role parameters. In addition the restriction on variables only appearing in

the body of a rule is lifted (Li and Mitchell 2003a; Li and Mitchell 2003b). For example, suppose a host $H$ wishes to grant access to a particular range of TCP ports to those entities that are employed by the information technology department. The host might have as its local policy:

$$Host.p(port \in [1024..2048]) \longleftarrow IT.employee$$

This example assumes that an entity is granted access to a particular TCP port if that entity is a member of the $Host.p$ role with the port specified as a parameter.

To accommodate threshold structures, representing agreement between a group of principals, the system $RT^T$ interprets roles as sets of sets of entities, called *principal sets*. These principal sets can be combined with role product operators $\odot$ and $\otimes$.

New credential forms are as follows:

1. $A.r \longleftarrow B_1.r_1 \odot B_2.r_2 \odot \cdots \odot B_k.r_k$

   Each principal set $p \in A.r$ is formed by $p = p_1 \cup \cdots \cup p_k$ where each $p_i \in B_i.r_i$ for $1 \leq i \leq k$.

2. $A.r \longleftarrow B_1.r_1 \otimes B_2.r_2 \otimes \cdots \otimes B_k.r_k$

   Each principal set $p \in A.r$ is formed by $p = p_1 \cup \cdots \cup p_k$ where $p_i \cap p_j = \emptyset$ for all $i \neq j$ and $p_i \in B_i.r_i$ for $1 \leq i \leq k$.

The features introduced by $RT^T$ allow threshold policies and separation of duty policies to be written (Li, Mitchell, and Winsborough 2002).

$RT^D$ adds the concepts of role activations and delegations to $RT_0$, via the delegation credential form $A \stackrel{C\ as\ D.r}{\longrightarrow} B$. In this case $A$ delegates to $B$ the *role activation* of $C\ as\ D.r$. Empowered with this role activation $B$ can then access whatever facilities $C$ can access from role $D.r$. This presupposes that $A$ has been delegated the activation $C\ as\ D.r$, which

holds when $A = C$ and $A$ is a member of role $D.r$ in the basic case. Hence, delegated activations don't carry any authority unless there is a chain of delegation credentials where the credential at the head of the chain was issued by the entity mentioned in the role activation.

While the original $RT$ framework does not support revocation in its policy language, it is proposed to incorporate revocation (Li, Mitchell, and Winsborough 2002) by leveraging a monotonic approach developed in (Li and Feigenbaum 2002) based on certificate lifetimes. While lifetimes and the requirement for freshness are encoded logically, the proposal suggests the use of external certificate revocation lists to implement verification; this is an interesting example of the possible interplay between the semantics of authorization per se and components external to them.

A variant of the $RT$ framework has been developed that associates risk values with credentials (Skalka, Wang, and Chapin 2007). These risks are tracked through the authorization process so that the role membership is parameterized by the total membership risk. The set of risks and their ordering is left abstract, and can be specialized to a number of applications, e.g., risk can be defined as remaining certificate lifetime, so that role membership is parameterized by the minimal lifetime of certificates used for authorization.

Finally $RT+_0$ extends $RT_0$ by adding an integer delegation depth control to most credential forms (Hong, Zhu, and Wang 2005), a capability that $RT_0$ lacks. $RT+_0$ delegation depths limit the delegation of authority by tracking the number of namespaces (administrative domains) such delegations cross. Delegation depth is also allowed to be unlimited, in which case $RT+_0$ degenerates to $RT_0$.

Although the work presented in this dissertation made use of $RT_0$ exclusively, supporting more advanced variations of the $RT$ framework, or indeed other trust management systems entirely, would be an interesting avenue for future development.

### 2.4.2 Example

Suppose Alice is a cancer patient at a hospital being treated by Bob, a doctor. Alice grants Bob access to her medical records and also allows Bob to delegate such access to others as he sees fit.

Bob defines his team as a particular collection of individuals together with the people supporting them. A person supporting one of Bob's team members becomes a team member herself, so Bob's definition is open ended and can potentially refer to a large number of people he does not know directly. Here we assume that Bob's team includes both medical and non-medical personnel, e.g., other doctors as well as receptionists. Once his team is defined, Bob then delegates his access to Alice's medical records to only the medical staff on his team and not the administrative staff.

Suppose further that Bob consults with another doctor, Carol, on Alice's condition. Bob modifies his policy to add Carol temporarily to his team. Carol orders some blood tests that are then analyzed by Dave, a lab technician and one of Carol's support people. The policy is intended to allow Dave to access Alice's medical records so that he may, for instance, input the blood test results.

Dave signs the test results when he uploads them to the hospital database. He also includes appropriate credentials so that the database will authorize his access. These credentials must include

- Bob has delegated his access to Alice's medical records to people on his team who are members of the medical staff.
- Carol is on Bob's team.
- If someone is on Bob's team, than any person on their support staff is also on Bob's team.

- Dave is one of Carol's support people.
- Dave is a member of the hospital's medical staff.

On the basis of these relations, one may deduce that Dave has access to Alice's medical records.

Complex access control scenarios such as this are difficult to express using traditional methods. Neither Alice nor Bob realize that Dave needs to be granted access to Alice's medical records. Although Dave's role as one of Carol's support people might be enough to grant him access to the records of Carol's patients, Dave's relationship to Bob, and hence to Alice, is indirect; it is Bob's act of adding Carol to his team that causes Dave to gain access to Alice's records. Observe also that Bob's team policy is recursive. A primary purpose of trust management systems is to provide language features and authorization semantics that support such complex policies.

To express this example using $RT$ only the facilities of $RT_0$ are necessary. This shows that even the simplest member of the $RT$ family can be used to express interesting policy statements. Alice defines a role `records` whose members are able to access her medical records. She creates the policy

- `Alice.records` $\leftarrow$ `Bob`
- `Alice.records` $\leftarrow$ `Bob.alice_delegates`

The first rule grants her doctor, Bob, access to her records. The second rule allows Bob to further delegate that access by defining the membership of an `alice_delegates` role.

Bob's standing policy is

- `Bob.team` $\leftarrow$ `Bob.team.support`
- `Bob.alice_delegates` $\leftarrow$ `Hospital.medical_staff` $\cap$ `Bob.team`

The first rule defines Bob's team as including all the support personnel specified by the members of his team. In the second rule, Bob uses an intersection role to specify that only the medical personnel on his team should have access to Alice's medical records.

When Bob consults with Carol he adds `Bob.team` $\leftarrow$ `Carol` to his policy to add Carol, and indirectly all of Carol's support people, to his team.

The only part of Carol's policy relevant to this example places Dave in her `support` role: `Carol.support` $\leftarrow$ `Dave`. Finally Dave has a credential from the hospital asserting his membership in the `medical_staff` role. $RT_0$ can use these credentials to prove that Dave is a member of `Alice.records` and thus able to access Alice's medical records.

### 2.4.3 Semantics

The original formal semantics of $RT$ is based on Datalog (Li, Mitchell, and Winsborough 2002). Specifically each $RT$ credential is translated into a Datalog rule. The meaning of a collection of $RT$ credentials is defined in terms of the minimum model of the corresponding Datalog program. In the case of the $RT_1^C$, Datalog with constraints is used (Li and Mitchell 2003a).

The translation from $RT_0$ to Datalog requires only a single predicate *isMember* to assert when a particular entity is a member of a particular role. The translation rules are shown below where Datalog variables are shown prefixed with '*?*' to distinguish them from constants.

1. $A.r \longleftarrow E$

   *isMember*$(E, A, r)$.

2. $A.r \longleftarrow B.s$

*isMember*$(?x, A, r) \leftarrow$ *isMember*$(?x, B, s)$.

3. $A.r \longleftarrow B.s.t$

   *isMember*$(?x, A, r) \leftarrow$ *isMember*$(?y, B, s)$, *isMember*$(?x, ?y, t)$.

4. $A.r \longleftarrow B_1.s_1 \cap \cdots \cap B_n.s_n$

   *isMember*$(?x, A, r) \leftarrow$ *isMember*$(?x, B_1, s_1), \ldots,$ *isMember*$(?x, B_n, s_n)$.

The authorizer associates a permission with a particular role, say $A.g$, named the *governing role*. Access is granted to $E$ if and only if the Datalog query *isMember*$(E, A, g)$ succeeds.

An alternative set-theory semantics has also been defined for $RT_0$ (Li, Winsborough, and Mitchell 2003). In this semantics each role $A.r$ is represented as a set of entities rmem$(A.r)$ that are members of that role. For a given set of credentials $\mathcal{C}$ these sets are the least sets satisfying the set of inequalities

$$\{\text{rmem}(A.r) \supseteq \text{expr[rmem]}(e) \mid A.r \longleftarrow e \in \mathcal{C}\}$$

where expr[rmem]$(e)$ is the set of entities in a particular role expression $e$. A *role expression* includes both linked roles and intersection roles. In particular:

$$\begin{aligned}
\text{expr[rmem]}(B) &= \{B\} \\
\text{expr[rmem]}(A.r) &= \text{rmem}(A.r) \\
\text{expr[rmem]}(A.r_1.r_2) &= \bigcup_{B \in \text{rmem}(A.r_1)} \text{rmem}(B.r_2) \\
\text{expr[rmem]}(f_1 \cap \cdots \cap f_k) &= \bigcap_{1 \le j \le k} \text{expr[rmem]}(f_j)
\end{aligned}$$

The set-theory semantics for $RT_0$ was developed primarily to provide theoretical support for a distributed credential chain discovery algorithm (Li, Winsborough, and Mitchell 2003). The set-theory semantics facilitate proving soundness and completeness of that algorithm.

### 2.4.4 Implementation

Li et al. describe an implementation strategy for $RT_0$ in terms of a construct called a credential graph $\mathcal{G}_\mathcal{C}$ (Li, Winsborough, and Mitchell 2003). Each node in $\mathcal{G}_\mathcal{C}$ represents a role expression with directed edges corresponding to each credential. In addition, *derived edges* are added to represent the indirect relationships between roles that are introduced by linked roles and intersections. An entity is a member of a role if, and only if, there exists a path from the entity to the role in $\mathcal{G}_\mathcal{C}$. Li et al. prove that credential graphs are sound and complete with respect to the set-theory semantics of $RT_0$.

In addition Li et al. describe a distributed credential chain discovery algorithm that finds a path in $\mathcal{G}_\mathcal{C}$ given initially incomplete credentials (Li, Winsborough, and Mitchell 2003). The algorithm assumes that either the issuer or subject of a credential can be contacted on-line and queried for more credentials on demand, an assumption that may not be true in an embedded systems context.

The most straightforward implementation of $RT_0$ is to simply compute the minimum model of the Datalog program implied by the union of policy statements and credentials provided by the requester. This can be done by updating role memberships repeatedly until a fixed point is reached, a process that is guaranteed to terminate in time polynomial in the total number of credentials (Li and Mitchell 2003a).

# Chapter 3

# SpartanRPC and Sprocket

This chapter describes SpartanRPC (Chapin and Skalka 2010; Chapin and Skalka 2013) and its implementation in the Sprocket compiler (Chapin 2013b).

SpartanRPC is a dialect of nesC that provides built-in support for authorized remote procedure calls. SpartanRPC as a language allows potentially many different forms of access control to be used, however Sprocket currently only supports the use of the $RT_0$ trust management system. Sprocket also uses radio links to implement *dynamic wires* (as described in section 3.4) and thus targets TinyOS and wireless sensor networks. However, there is nothing in the design of SpartanRPC that would preclude the use of other communications channels.

The use of a trust management system allows embedded developers to specify high level authorization policies that permit different security domains to interact without prior introduction. In a sensor network context this might arise when, for example, the wearer of a body area network enters a region of space covered by a metropolitan network. These networks may have never encountered each other, yet wish to access sensitive functions, such as for medical monitoring and control.

Trust management systems use public key cryptography and require some mechanism

for evaluating authorization requests in the light of an access policy (the $L$ in Figure 2.1). Although the feasibility of using public key cryptography on sensor nodes has been shown by several authors (Gupta, Millard, Fung, Zhu, Gura, Eberle, and Shantz 2005; Malan, Welsh, and Smith 2008; Bertoni, Breveglieri, and Venturi 2006; Kumar and Paar 2006; Lee, Sakiyama, Batina, and Verbauwhede 2008; Liu and Ning 2008; Szczechowiak, Oliveira, Scott, Collier, and Dahab 2008), combining this with the necessary authorization decision to support a full trust management system, and showing the feasibility and practicality of doing so on resource constrained devices, has not been previously demonstrated. As will be shown in chapter 6 the Sprocket runtime system exacts a significant performance penalty on the nodes, particularly with respect to system startup time. Yet despite this problem useful work can still be accomplished.

## 3.1 Overview and Applications

The SpartanRPC language allows network administrators to define $RT_0$ policies that mediate access to specified resources on network nodes. In SpartanRPC a resource is user-defined functionality programmed in an extension of nesC, and accessible in RPC style by client code programmed in the same extension of nesC. Thus, while previous systems have explored the problem of establishing multiple security domains in a wireless sensor network (Claycomb and Shin 2011), and others have considered RPCs in sensor networks (May, Dunning, Dowding, and Hallstrom 2007; Bergstrom and Pandey 2007; Reinhardt, Mogre, and Steinmetz 2011), SpartanRPC provides a readily-accessible mechanism that combines these features. Furthermore, SpartanRPC's use of $RT_0$ allows specification of fine-grained, decentralized security policies.

In addition to the first responder application described in chapter 1, various other potential applications of SpartanRPC exist. For example, time synchronization is another

important sensor network function that is security sensitive, since many higher-level protocols rely on it. A number of previous authors have considered secure time synchronization in the presence of "insider" attacks (Manzo, Roosta, and Sastry 2005; Ganeriwal, Pöpper, Čapkun, and Srivastava 2008), whereby nodes within the network may be compromised and function as malicious actors capable of corrupting the protocol.

In particular, the FTSP protocol can be attacked by a single compromised "root" node injecting false timing information into the network (Manzo, Roosta, and Sastry 2005), even when symmetric keys are used for secure information exchange. However, the threat model in this work treats all nodes in a network as equally compromisable. In cases where a connected sub-component of a network running an FTSP protocol is more resistant to compromise, due to, e.g., the use of tamper-proof hardware, a policy can be established whereby only nodes in the most tamper-resistant sub-component of the network may function as roots. FTSP time sync updates on any given node can be defined to require a root authorization level. This implies that nodes requiring secure time synchronization must be at most a single radio hop from a root node, but nodes willing to accept possibly corrupted time sync data can extend the network indefinitely. Note that in this scenario, SpartanRPC policies adapt to heterogeneity in network device hardware, vs. network administration as in the first responder example in chapter 1.

Other potential applications of this system include secure routing protocols in heterogeneous trust environments (Karlof and Wagner 2003), transport and network layer protocols (Perillo and Heinzelman 2005), tracking protocols (Brooks, Ramanathan, and Sayeed 2003), and even node-based web servers supporting secure channels (Gupta, Millard, Fung, Zhu, Gura, Eberle, and Shantz 2005).

## 3.2 Technical Foundations

**Language-Based Security.** SpartanRPC provides language-level abstractions for defining remote services and associated security policies. Programmers are presented with an extension of nesC, with new features for defining remote access controlled services, and for invoking those services securely. This hides the implementation details of the underlying security protocols and only requires mastery of $RT_0$, a simple authorization logic. SpartanRPC programs are compiled in the same manner as nesC programs, in fact the SpartanRPC compiler rewrites SpartanRPC programs to ordinary nesC code.

**Asynchronous Remote Procedure Call.** As other authors have observed (May, Dunning, Dowding, and Hallstrom 2007), RPC is an appropriate abstraction for node services on the network and supports whole-network (vs. node-specific) programming. Secure RPC is well-studied in a traditional networking environment, and is a natural means of layering security over a distributed communication abstraction.

It is necessary for RPC invocation in a wireless sensor network to be asynchronous, since synchronous call-and-return to a remote node would significantly impede performance in the best case and cause deadlock in the worst. In order to minimally impact the nesC programming model, SpartanRPC defines RPC invocation as a form of *remote task*. Local tasks are units of programmer-defined asynchronous computation in nesC, so treating remote computational services as remote tasks fits this paradigm. Remote tasks can be invoked on one-hop neighbors, providing a link layer service on which network layer services can be built.

**PK-Based Authorization Policies.** SpartanRPC provides language-level abstractions for specifying RPC authorization policies. The $RT_0$ trust management system allows network entities to communicate credentials for authorizing service invocations. In SpartanRPC

these credentials are implemented with ECC public keys (Bertoni, Breveglieri, and Venturi 2006), which are validated during the initial authorization phase. ECC is significantly more tractable than RSA in a resource constrained setting. Furthermore, following an initial authorization phase SpartanRPC protocol establishes a shared AES key for subsequent invocations of a given service by the same node. Since hardware AES is available on common radio chipsets, highly efficient performance for secure invocations is obtained following authorization. This is demonstrated with empirical results reported in chapter 6.

## 3.3 Duties and Remotability

Because of the slow, unreliable nature of wireless communications it is unrealistic for RPC services in wireless sensor networks to be synchronous. Instead, the semantics of tasks are considered a more appropriate abstraction. They are not quite right however, as RPC services will typically require arguments to be passed—a feature not provided by nesC tasks—and while the poster of a task defines it, an RPC service invokes remotely defined functionality. SpartanRPC therefore defines a new RPC abstraction called a *duty*.

### 3.3.1 Syntax and Semantics

Duties are declared in nesC interfaces and syntactically resemble nesC command declarations. Instead of using the reserved word **`command`** the new reserved word **`duty`** is used. Duties are allowed to take parameters (with restrictions as discussed below) but must return the type **`void`**. For example, the following interface describes an RPC service for remotely controlling a collection of LEDs:

```
interface LEDControl {
    duty void setLeds( uint8_t ctl );
}
```

```
module LEDControllerC {
  provides remote interface LEDControl;
}
implementation {
  duty void LEDControl.setLeds(uint8_t ctl) { ... }
}

module LoggerC {
  uses interface LEDControl;
}
implementation {
  void f() { ... post LEDControl.setLeds(42); }
}
```

**Figure 3.1:** Duty Implementation and Invocation Examples

Duties are defined in modules in a manner similar to the way tasks, commands, or events are defined. The reserved word **`duty`** is again used on the definition. Similar to commands and events the name of the duty is qualified by the name of the interface in which it is declared. Including a duty in an interface definition automatically implies that the interface can be remotely invoked, or is *remotable* in the sense formalized in subsection 3.3.2. Any remotable interface provided by a component must be specified as **`remote`** in its provides specification. The first code sample in Figure 3.1 shows an `LEDControllerC` component that provides the `LEDControl` interface remotely, i.e., that allows remote nodes to control LED status lights on a board.

A module on the client node that wishes to use a remote interface simply posts the duty in the same manner as tasks are posted. The use of **`post`** emphasizes the asynchronous nature of the invocation. An example duty posting is illustrated in Figure 3.1. The standard component semantics of nesC provide a natural abstraction of "where" the RPC call goes, just as a normal command invocation will go through a component interface that is disconnected from its implementation. Like a normal command invocation, configuration wirings determine where duty control flows. However, in SpartanRPC duty invocation flows to a

component residing on a different node. The invoking module must be connected to the remote modules by way of a dynamic wire as described in section 3.4.

When a duty is posted by a client it may run at some time in the future on the server node. The client node continues at once without waiting for the duty to start, i.e., duty postings are asynchronous in the same manner that tasks are. Once posted the client has no direct way to determine the status of the duty. Also, due to the unreliability of the network a posted duty may not run at all. The success or failure of a duty posting is not signaled to the client in the implementation just as, for example, the receipt or non-receipt of a message send is not signaled in the `AMSend` protocol in TinyOS. Hence any error semantics for duty postings must be implemented by the application developer.

### 3.3.2 Remotable Interfaces

SpartanRPC imposes certain requirements on RPC service definitions for ease of implementation. First, since sensor network nodes do not share state passing nesC pointers to duties is disallowed—such a reference would be meaningless on the receiving node. Thus remotable types are defined as follows:

**Definition 3.3.1** *A type is* remotable *if and only if it satisfies the following inductive definition: The nesC built-in arithmetic types, including enumeration types, are remotable, and structures containing remotable types are remotable.*

Since a remotable interface describes RPC services, such interfaces are required to declare duties taking only arguments of remotable type; also, remotable interfaces can only contain duties, to ensure meaningful remote usage.

**Definition 3.3.2** *An interface is* remotable *if and only if it contains only duties, and those duties have argument types and return types that are remotable.*

## 3.4 Dynamic Wires

In an ordinary nesC program the "wiring" between components as defined by configurations is entirely static. The nesC compiler arranges for all connections and at run time the code invoked by each called command or signaled event is predetermined.

In a remote procedure call system for distributed embedded environments, especially those communicating using radio links, this static arrangement is insufficient. A node cannot, in general, know its neighbors at compilation time but rather must discover this information after deployment. In addition, the volatility of wireless links, and of the nodes themselves, means that a given node's set of neighbors will change over time. This section discusses the facility in SpartanRPC to allow *dynamic wirings* for control flow from duty invocation via remotable interfaces to duty implementation, wherein the programmer has control over wiring endpoints and how they may change during program execution.

### 3.4.1 Component IDs, Component Managers

The discussion begins with how remote components are identified for wiring. In order to uniquely identify components on a network of devices, remotable components are specified via a two-element structure called a `component_id` defined on the left side of Figure 3.2. The `node_id` member is the same node ID used by TinyOS and is set when the node is programmed during deployment. The local ID member is an arbitrary value defined by the programmer of the server node. Only components that are visible remotely need to have ID values assigned, however, the ID values must be unique *on the node*. The `component_set` structure defined on the right side of Figure 3.2 wraps an arbitrary array of `component_id` values.

A *component manager* is a component that provides the `ComponentManager` interface defined at the bottom of Figure 3.2. It dynamically specifies a set of component IDs

```
typedef struct {
  uint16_t node_id;
  uint8_t  local_id;
} component_id;
```

```
typedef struct {
  int count;
  component_id *ids;
} component_set;
```

```
interface ComponentManager {
  command component_set elements( );
}
```

**Figure 3.2:** Component Manager Interface and Type Definitions

that ultimately serve as dynamic wiring endpoints.

As a simple example, consider the component manager `RemoteSelectorC` as shown in Figure 3.3. This example component manager always returns a component set containing a single component. However, in general, multiple components on neighboring (one-hop) nodes could be selected. Hence dynamic wires are inherently a multi-cast communication channel. In a more complex example the component manager would compute the component set each time the dynamic wire is used, filling in an array of component IDs based on information gathered earlier in the node's lifetime.

### 3.4.2 Syntax and Semantics

In SpartanRPC the syntax and semantics of nesC is extended to allow the target of a connection to be dynamically specified by a component manager. The syntax of wirings, or connections, is extended as follows:

```
      connection ::= endpoint '->' dynamic_endpoint
dynamic_endpoint ::= '[' IDENTIFIER ']' ('.' IDENTIFIER)
```

Given a dynamic wiring of the form `C.I ->[RC].I`, its semantics are informally summarized as follows. First, `RC` is statically required to be a component manager, and

```
module RemoteSelectorC {
  provides interface ComponentManager;
}
implementation {
  // 0xFFFF is the special broadcast address.
  // Local component #1 on each node selected.
  component_id  broadcast  = { 0xFFFF, 1 };
  component_set broadcast_set = { 1, &broadcast };

  command component_set ComponentManager.elements() {
    return broadcast_set;
  }
}
```

**Figure 3.3:** Example Component Manager

`I` must be remotable. At run time, if control flows across this wire via posting of some duty `I.d` within `C`, the command `elements` in `RC` is called to obtain a set of component IDs. The duties `I.d` provided by those remote components will then be posted on the host nodes via an underlying link layer communication, the details of which are hidden from the SpartanRPC programmer. Thus, duties can only be posted on neighbors. Note that since this call to `elements` may return more than one component ID, this is a sort of fan-out wiring.

For example, the `LoggerC` component mentioned in Figure 3.1 could be wired by the programmer to LED controller components on a dynamically changing subset of neighbors using a configuration such as:

```
LoggerC.LEDControl -> [RemoteSelectorC].LEDControl;
```

The server's configuration does not need to wire anything to the remote interface explicitly.

### 3.4.3 Callbacks and First-Class IDs

It is assumed that the component IDs for well known services will be agreed upon ahead of time by a social process outside of SpartanRPC. By broadcasting to a "well known" component ID, a node can use services on neighboring nodes without knowing their node IDs. The use of well known ID values is analogous to the use of well known TCP port numbers to provide easily accessible Internet services.

If a node expects a reply from a service that it invokes, the invoking node must set up a component with a suitable remote interface to receive the service's result. In SpartanRPC remote invocations can only transmit information in one direction. Bidirectional data flow requires separate dynamic wires. This design provides a natural "split-phase" semantic wherein the invoker of a service can continue executing while waiting for the result of that service, a common idiom for nesC programming. For instance, a service might require the client to provide the node ID and component ID of the component that will receive the service result as arguments to the service invocation. The server could store those values for use by a server-side component manager. It is permitted for a component to be its own component manager making it easy for a service to return a result by posting the appropriate duty.

For example, assume that the LED controller on the server returns the old state of the LEDs whenever the LED value is changed. The server configuration would include an appropriate dynamic wire as follows

```
LEDControllerC.LEDResult -> [LEDControllerC].LEDResult;
```

The client must provide the LEDResult interface remotely to receive this result. In this example the `LEDControllerC` component is its own component manager. This makes it easy for the `elements` command to access global data that was recorded inside `LEDControllerC` when the service it provides was previously invoked. This is a

common SpartanRPC idiom.

## 3.5 Security Policy Specification

This section discusses how to extend the language setting described previously with security features. The goal is a language framework where RPC services require authorization for use, and where authorization policies support collaboration between multiple social domains. The authorization model can be viewed as a client-server interaction; respective sides of the interaction protocol are summarized separately as follows.

### 3.5.1 RPC Server Side Logic

RPC service providers establish policy by assigning governing roles $A.g$ to remote interface implementations. Service providers also possess a set of assumed credentials $\mathcal{C}$, which establish an authorization environment including, perhaps (but not necessarily), the access policy. As will be described in detail, the set $\mathcal{C}$ may grow as additional credentials are communicated to servers. Finally, in the presence of security, client invocations of any RPC service are not anonymous, but are performed on behalf of some entity $B$, which must be a member of the governing role $A.g$ to use the protected service.

In summary, access to an RPC level is allowed if and only if the property $\mathcal{C} \vdash B \in A.g$ holds, where:

- $B$ is the identity of the RPC client.
- $A.g$ is the governing role of the RPC service.
- $\mathcal{C}$ are the credentials known to the RPC server.

RPC service programmers specify governing roles as part of module definitions, specifically at remote interface **`provides`** clauses. Hence, governing roles are associated with interface *implementations*, not interfaces themselves. This allows application flexibility, in that the same interface can be implemented with various authorization levels within the same network. Syntax is as follows:

**`provides remote interface`** $I$ **`requires`** $A.g$

Note the minor modification to previously introduced syntax for remote module definitions, via the **`requires`** keyword.

### 3.5.2 RPC Client Side Logic

In order to use a secure remote module, RPC clients wire to it as for unsecured modules (see subsection 3.4.2), but with two additional capabilities: (1) the client specifies under what $RT$ entity the invocations will be performed, and (2) the client may also specify credentials in their possession which are to be activated for use in the invocation. Syntax is as follows:

**`activate`** "$C_1, \ldots, C_n$" **`as`** "$B$" **`for C.I -> [M].J`**

For any invocation made through this wiring the credentials $C_1, \ldots, C_n$ will be remotely added to the RPC server's authorization environment for the authorization decision, via a process detailed in section 3.6. Note that these credentials *need not establish authorization entirely by themselves*, rather they will be *added* to the server's existing credentials, all of which will be used in the authorization decision. A special form of the **`enable`** clause using **`"*"`** for the list of credentials is also supported. This form indicates that all credentials known to the client should be communicated to the server.

Each node is deployed with a collection of ECC key pairs, one for each entity the node represents. When an invocation is made the entity $B$ mentioned in the **`as`** clause

of the dynamic wire is used in the request. The **as** clause is optional; if it is omitted a distinguished *default entity* is used for the invocation.

### 3.5.3 Example

Suppose that an existing network deployment NetA is imaged with a component called SamplingRateC which provides a means to control sampling rates through an interface such as SamplingRate. Further, since sampling rate modification is a sensitive operation, the network administrators require NetA.control authorization to use this component:

```
module SamplingRateC {
  provides remote
    interface SamplingRate requires "NetA.control";
}
```

Any node supporting this component will transparently receive $RT$ credentials from neighboring nodes and attempt to use those credentials to establish that each client entity is a member of the NetA.control role in the formal sense described above.

Suppose also that nodes in NetA are deployed with the credential

$$\texttt{NetA.control} \leftarrow \texttt{WSNAdmin.control}$$

Here the role WSNAdmin.control is administered by some overarching network authority. However, this authority need not be physically "present" in the network during operation. Instead the credential above represents NetA's access control policy: any entity blessed by WSNAdmin as a controller can control NetA.

Suppose further that another subnet, called NetB, wishes to modify the sampling rate of NetA. A node in NetB might be imaged with the following credentials, among possibly others:

```
activate
  "WSNAdmin.control <- NetB.control,
   NetB.control     <- NetB" as "NetB"
for
  ClientC.SamplingRate -> [RemoteSelectorC].SamplingRate;
```

**Figure 3.4:** Security Enabled Dynamic Wire

$$\texttt{WSNAdmin.control} \leftarrow \texttt{NetB.control} \quad (3.1)$$

$$\texttt{NetB.control} \leftarrow \texttt{NetB} \quad (3.2)$$

Note that credential (1) is issued by the `WSNAdmin` authority, while credential (2) is issued by `NetB`. Critically, direct communication with `NetA` authorities to obtain these credentials is unnecessary.

In order to invoke this service the wiring as shown in Figure 3.4 could be made on the client side. Note the activation of the necessary credentials, as well as the specification of client identity as `NetB`.

## 3.6 The SpartanRPC Implementation

This section describes the Sprocket implementation of the SpartanRPC system using $RT_0$ trust management for authorization. Sprocket rewrites a SpartanRPC program into a pure nesC program and provides a supporting runtime system. Program rewriting converts remote duty postings into a nesC messaging protocol. The main task of the runtime system is to implement the encapsulated, underlying security protocols for authorization of remote duty postings.

### 3.6.1 Authorization and Security Protocols

Sprocket implements SpartanRPC authorization using a combination of public and symmetric key cryptography. The TinyECC library (Liu and Ning 2008) was used for public key functionality, and AES encryption for symmetric key functionality. TinyECC uses elliptic curve cryptography for more efficient public key operations in sensor networks. Using AES has the benefit of hardware support on many current embedded platforms, e.g., those employing the Chipcon CC2420 radio.

There are three security protocols for authorized duty postings, illustrated in Figure 3.5, each operating asynchronously. First, a credential exchange protocol, wherein $RT$ credentials are communicated between nodes and authorization for various entities are computed, i.e., the *minimum model* as described in section 3.5. Second, a session key negotiation protocol, where symmetric keys for multiple authorized service invocations between a duty client and server are computed. And third, an authorized service invocation protocol, wherein duty posting requests are checked to ensure the appropriate authorizations. This decomposition of authorized service invocation into three protocols supports efficiency especially through the use of symmetric keys for multiple service invocations. Its asynchronous nature is also appropriate in an asynchronous TinyOS setting.

#### Credential Exchange

SpartanRPC credentials are implemented as signed certificates. All SpartanRPC-enabled nodes contain a certificate sender component and a certificate receiver component, to transfer certificates between nodes and to verify them and interpret the credentials they represent. Both components run as background daemons. A SpartanRPC-enabled node is deployed with a collection of certificates in read-only storage representing that node's credentials, which are determined by some external means. Once the node is booted, the

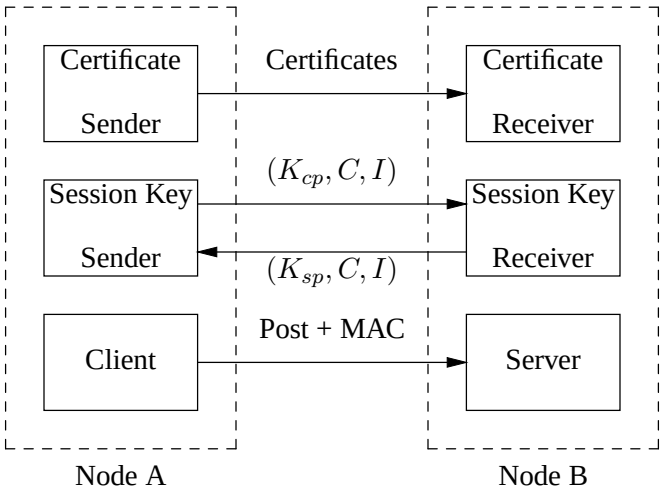


**Figure 3.5:** SpartanRPC Security Protocol Elements

certificate sender starts a periodic timer. When the timer fires, the node link-layer broadcasts (i.e., only to neighbors) all certificates in its certificate storage that are mentioned in the **`enable`** clauses in its program. To prevent adjacent node certificate broadcasts from colliding, the certificate broadcast interval is modulated randomly by $\pm 10\%$. For example if the nominal broadcast interval is one minute, the actual time varies randomly between 54 and 66 seconds.

The certificate distribution strategy is robust in the face of new nodes being added to the network or intermittent radio connectivity. If a node fails to receive certain certificates from its neighbors it will have another opportunity to do so when those neighbors rebroadcast their certificates. There is a trade off between the broadcast interval, responsiveness, and network energy consumption. A short broadcast interval allows authorizations to succeed "quickly" since neighbors become aware of the necessary credentials early, but at the cost of increased radio traffic and power consumption.

Once a newly received certificate has been verified, the credential it represents is extracted and stored. The credential storage also contains the $RT_0$ minimum model implied by the currently known collection of credentials. Each time a new credential is added to

$A.r \leftarrow B.s \cap C.t$

| 4 | $A$ (40) | $r$ | $B$ (40) | $s$ | $C$ (40) | $t$ | sig (42) | chk (2) |
|---|---|---|---|---|---|---|---|---|

**Figure 3.6:** Intersection Certificate Format (parenthesized numbers indicate byte counts)

storage, the minimum model is updated. This is done by repeatedly applying authorization logic inference rules implied by the credentials to the current model until a fixed point is reached, i.e., a logical closure (Li and Mitchell 2003a). Thus, each node maintains a local view of authorization levels for network entities based on received credentials.

The certificate representation of an $RT$ credential contains the public keys denoting entities mentioned in the credential. Roles are identified by one byte numeric codes and are scoped by the entity defining the role. Credential forms are distinguished by numbers $\{1, 2, 3, 4\}$. Certificates are also signed by their issuing authority. Conveniently, the issuing authority is always mentioned in a credential (e.g., the issuing authority of $A.r \longleftarrow B$ is $A$) so the public key required to verify the certificate is always included with it for free. This does not introduce a security problem. Since entities are identified directly by their keys, an attacker who creates a new key is simply creating a new entity.

The over-the-air format for the intersection certificate is illustrated in Figure 3.6. The other certificate forms are organized in a similar way. Certificates range in size from 124 bytes for the membership credential to 166 bytes for the intersection credential. This is larger than the maximum payload size limit of TinyOS T-Frame Active Message packets as transported by IEEE 802.15.4 (Society 2003; Hui, Levis, and Moss 2008). It is much larger than the default maximum payload size of 28 bytes used by TinyOS (Levis ). Consequently the certificates are fragmented into four messages requiring a maximum payload size of 43 bytes. Notice that SpartanRPC limits intersection roles to just two subroles and does not allow an arbitrary number of subroles as described in section 2.4. This does not limit expressivity because intermediate roles can be defined if necessary.

Message fragments are sent back to back with a 200 ms delay between each to allow the receiver time to assemble them. Fragments are sent in order with no fragment identifiers. To stay synchronized with the sender, receivers expect to receive all the fragments in a timely manner. If a fragment is not received within 750 ms of the previous fragment, the partial certificate is discarded on the assumption that the expected fragment was lost.

Verification of $RT$ certificates is the most computationally expensive component of the system as discussed in subsection 6.2.2. Thus, it is important to minimize the amount of effort spent on verification. To this end, a 16-bit Fletcher checksum is appended to each certificate to ensure integrity over unreliable channels. Also, nodes maintain a database of certificate checksums, to quickly check whether a certificate has already been received and verified. Fletcher checksums are commonly used in sensor networks and other embedded systems since their error detection properties are almost as good as CRCs with significantly reduced software computational cost (Fletcher 1982).

Currently certificates carry no lifetime information and are considered to be valid forever. This is not ideal since a certificate issuer may eventually change his/her policy but currently has no way to revoke old certificates. However, adding a feature for certificate revocation introduces non-monotonicity into the semantics of the authorization logic (Li and Feigenbaum 2001; Rivest 1998b). Adding an expiration time to the certificates is more logically appealing but would require nodes to support real time services and some degree of time synchronization. This is a non-trivial extension of the basic system that was beyond the scope of this work.

**Session Key Negotiation**

Public key cryptography is much too computationally expensive to use for authorizing routine duty postings. Sprocket's run time system addresses this by negotiating session keys between the client and server nodes. Figure 3.7 shows the session key processing

architecture of a node.

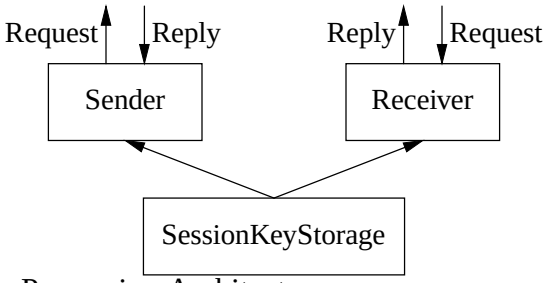


**Figure 3.7:** Session Key Processing Architecture

The client maintains a session key storage that is indexed by the triple $(N, C, I)$ where $N$ is the remote node ID, $C$ is the remote component ID, and $I$ is the remote interface ID. A session key is thus created for each combination of these IDs. The server also maintains a session key storage indexed by $(N, C, I)$. In this case $N$ is the node ID of the client and $C$, $I$ are the component and interface IDs on the server to which that client is communicating. Since any given node can be both server and client, each session key storage entry has a flag to indicate the nature (client-side or server-side) of the session key.

The first time a client attempts to access a service on a particular server, it will send a session key negotiation request as shown in the middle portion of Figure 3.5. When a server receives a session key negotiation request message from a client node $N$ containing the public key $K_{cp}$ of the requesting entity (as mentioned in the **`as`** clause of the dynamic wire) and the $(C, I)$ address of the desired service, the following steps are taken:

1. Authorization of $K_{cp}$ for service $(C, I)$ is checked using the $RT$ minimum model computed by the certificate receiver. If authorization fails nothing more is done.

2. A session key is computed using elliptic curve Diffie-Hellman key agreement and added to the session key storage under the proper $(N, C, I)$ value. The key is stored as a remote client key.

3. A message is returned to the client containing the server's public key $K_{sp}$ and the

original $(N, C, I)$ values used by the client. This is so the client is able to compute the same session key and associate it with the proper endpoint from its perspective.

The session key negotiation protocol is a simple Diffie-Hellman key agreement protocol that combines the public key of the peer entity with the private key of the local entity. The implementation does not include any nonces as would be done, for example, with the ECMQV protocol (ISO 2008). As a result any renegotiated session keys between the same entities would be identical. However, this is not a serious problem because the implementation does not currently renegotiate session keys anyway. Furthermore the ECMQV protocol entails three exchanged messages and additional computations and so would further increase the burden on the nodes.

A potentially more serious concern is that the simple protocol described here would normally be vulnerable to a man in the middle attack whereby an active attacker negotiates independent session keys with each peer and is then able to modify messages sent between those peers. However, in an $RT$ trust management context this is not a concern because authorization is entirely based on key rather than on any identity bound to that key. An active man in the middle who creates a "bogus" key would simply be creating a new and presummably unauthorized $RT$ entity.

The session key negotiation protocol described here always computes the same session key between nodes $N_A$ and $N_B$ for the same requesting entity. This is also not a problem since the server node uses $(C, I)$ to look up the session key in its storage. If $N_A$ previously negotiated a session key with $N_B$ for service $(C_1, I_1)$, an attempt by $N_A$ to use that session key to access an unauthorized service $(C_2, I_2)$ will fail because the server has no entry for $(N_A, C_2, I_2)$ in its session key storage. In fact, this design creates an optimization opportunity called *session key stealing* where, in the case of a successful authorization for $(C_2, I_2)$, a previously computed session key for $(C_1, I_1)$ can simply be copied by the server

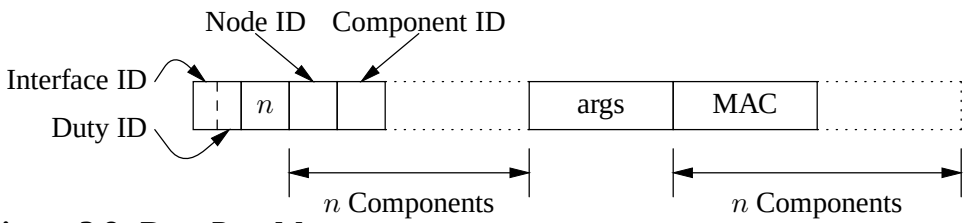

**Figure 3.8:** Duty Post Message

from $(N_A, C_1, I_1)$ to $(N_A, C_2, I_2)$ without being recomputed. However, at the time of this writing Sprocket does not implement session key stealing.

### Authorized RPC Invocations

Authorized RPC invocations are made using message authentication codes (MACs) on invocation messages, created with AES session keys. Verification of a MAC for a particular service on the receiver side constitutes authorization, since a session key for a particular client and service is negotiated only after client credentials have been collected and verified that establish the appropriate authorization for the service. Figure 3.8 shows the format of authorized invocation request messages.

Since invocation of an RPC service on multiple hosts can be made at once in a fan-out wiring (see section 3.4), a single invocation request message may apply to multiple servers in the neighborhood of the client. To conserve bandwidth, fan-out invocation messages include multiple MACs, since separate session keys are negotiated with each of $n$ servers, allowing a single message to invoke the same service on the servers. If the duty arguments consume $d$ bytes of data, then invocation messages consume $2 + n + d + 4n$ bytes. In practice this puts significant restrictions on the amount of data that can be passed to duties.

As described above this implementation uses a 43 byte message payload for sending certificate fragments. Experience suggests that using the same payload size for invocation messages allows for reasonable values of both $d$ and $n$.

Alternatively an implementation could send multiple invocation messages with one for

each server, reducing the number of MACs required on each message to one. However, that greatly increases radio traffic since the duty arguments and active message overhead must be duplicated for each message.

To conserve space in the invocation messages, only a 32 bit MAC is used. Such a small MAC would not normally be considered secure. However, wireless sensor networks generate data so slowly that attacking even such a short MAC is not considered feasible (Karlof, Sastry, and Wagner 2004; Luk, Mezzour, Perrig, and Gligor 2007). Nevertheless, in other environments a larger MAC may be necessary further increasing message size.

**Security Properties**

It must be stressed that this scheme is intended to enforce authorization, which is achieved via the protocols described above. Integrity is a side effect of this, since MACs are used to enforce authorization, which are computed over complete message payloads and are verified by the receiver. Although confidentiality is not directly supported by the current system, it could be easily added. In particular payloads could be encrypted using negotiated session keys (payloads are currently sent as plaintext).

Sprocket does not provide any form of replay protection out of the box, but this can be added at the application level. For example an application could pass a counter as an additional duty argument. The server could verify that the count increases monotonically as a simple form of replay protection. Delegating replay protection to the application is appropriate since SpartanRPC is intended to be a low level infrastructure on which more complex systems can be built. Furthermore the need for replay protection is likely to be application specific.

Perhaps the most problematic vulnerability of this system is to denial of service attacks. It is not clear how these could be mitigated without significant changes to the underlying security protocols. For example, a constant flood of certificates over the correct active

message channel would place receiving nodes in a constant state of ECC digital signature verification, potentially consuming large amounts of CPU time and energy. Mitigation of such attacks is outside the scope of this work, but has been discussed in the literature (Raymond and Midkiff 2008).

**A note on multicast security.** Fan-out wirings are a common idiom, and provide a form of multicast communication. However, the use of MACs for security in a multicast setting presents well-known challenges. In particular, while $n$-way Diffie-Hellman can be used to negotiate secret keys between $n$ actors, such a scheme cannot be used in light of the possibly heterogeneous authorization requirements anticipated. For instance, suppose a node $A$ fan-out wires to service $s$ on distinct nodes $B$ and $C$, and suppose also that $A$ is authorized for $s$ on both nodes but that $B$ is not authorized for $s$ on $C$ and vice-versa. If a single session key were negotiated between $A$, $B$, and $C$ in this case, then $B$ could make unauthorized use of $C$'s version of $s$ and vice-versa. While a variety of techniques have been proposed to mitigate this problem (Canetti, Garay, Itkis, Micciancio, Naor, and Pinkas 1999), most typically rely on very large multicast groups and are not applicable in a wireless sensor network setting. Thus, fan-out wirings use multiple, independent MACs as described above.

### 3.6.2 Identifying Services Over the Air

RPC service endpoints are identified by the 4-tuple $(N, C, I, D)$ where $N$ is the TinyOS ID of the node on which a duty is implemented. $C$ is the local component ID assigned to each component that provides a remotable interface. $I$ is an interface ID, required since a component may provide more than one remotable interface. Interface IDs are component-level unique. Finally $D$ is a duty ID, which must be interface-level unique.

In the current version of Sprocket, $(C, I)$ values are assigned statically by an arbitrary

(automated or social) process. Sprocket accepts configuration files that define the association between $(C, I)$ values and the entities to which they refer. Duties are numbered in the order in which they appear in their enclosing interface definitions.

Some RPC systems, such as ONC RPC, allow each node to provide a registry of RPC services available on that node (Srinivasan 1995). When a large number of RPC services are provided by a node it becomes unreasonable to expect clients to have hard coded knowledge of the endpoint identifiers for all those services. Instead clients communicate with the single well known registry to obtain endpoint identifiers that were dynamically assigned. In contrast it is assumed this configuration information is known a priori to all interacting actors. It is unclear how many embedded systems could benefit from a more sophisticated technique for defining and communicating endpoint identifiers, but it would be an interesting topic for future work.

### 3.6.3 Rewriting SpartanRPC to nesC

Sprocket rewrites five major features from SpartanRPC to nesC: interface definitions, call sites where remote services are invoked, duty definitions, dynamic wires, and server components providing remote interfaces. Additionally, Sprocket generates a stub component for each dynamic wire, and a skeleton component for each remote interface. Finally, Sprocket generates configurations that wrap server components. The following summarizes rewriting strategies for these features.

#### Interfaces, Call Sites, and Duty Definitions

Duty declarations in interfaces are rewritten to command declarations by substituting the reserved word **`command`** for the reserved word **`duty`**. Since nesC commands are allowed to have arbitrary parameters, duties with parameters present no complications. Sprocket

verifies that if an interface contains a duty, then the only declarations in that interface are duties. Sprocket further verifies that the parameters of each duty, if any, conform to the restrictions described in subsection 3.3.2.

Call sites where duties are posted are rewritten to command invocations by substituting **`call`** for **`post`**. Only post operations applied to duties are rewritten in this way. Finally, duty definitions are rewritten to command definitions by also substituting **`command`** for **`duty`**.

### Authorization Interfaces

The rewriting process makes use of two interfaces that mediate the interaction between the Sprocket generated code and the security processing components of the run-time system. Figure 3.9 shows how a message, entering from the left, is extended with authorization information by the client and then passed to the server where the authorization information is checked.

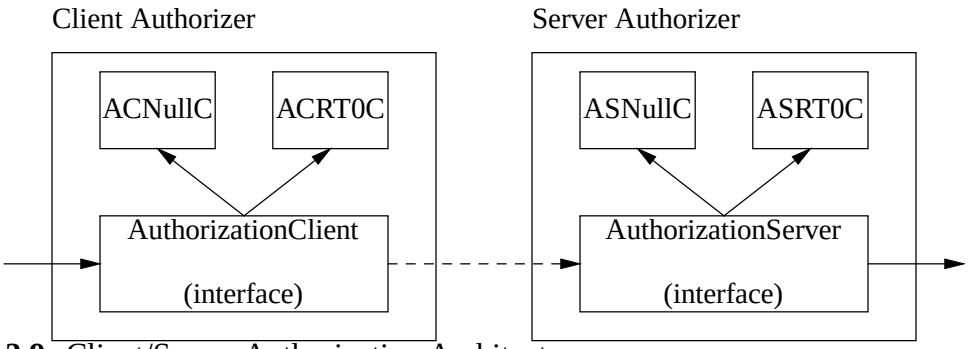


**Figure 3.9:** Client/Server Authorization Architecture

The `AuthorizationClient` interface abstracts the details of how an authorized message is prepared before being sent. The `AuthorizationServer` interface abstracts the details of how authorized messages are processed after they are received. This design allows for pluggable authorization mechanisms. Future versions of Sprocket could support, in a modular fashion, other authorization schemes than those described here.

The authorization interfaces provide their services in a split-phase manner so that potentially long-running authorization computations can be performed while allowing the node to continue other functions. In the current implementation, two kinds of authorization are supported. On the client side the precise method used depends on the dynamic wire over which a particular communication takes place. On the server side it depends on the presence of a **`requires`** clause on the remotely provided interface.

The full $RT_0$ mechanism is supported by client and server components `ACRT0C` and `ASRT0C` respectively. In addition a "null" authorization is supported by client and server components `ACNullC` and `ASNullC` respectively. The null authorization components perform no operation. They are used for dynamic wires that do not require authorization and remote interfaces provided publicly by servers.

**Dynamic Wires**

In the following, italics are employed for metavariables that range over arbitrary identifiers. The reader is referred to the rewriting schema defined in Figure 3.10. Configurations containing dynamic wires are rewritten to configurations that statically wire the using component *`ClientC`* to a stub `Spkt_`*`n`* that interacts with the appropriate component manager *`SelectorC`* and that handles the communication channel. Every stub generated by Sprocket is uniquely identified over the scope of the entire program by an arbitrary integer $n$. The *`AuthorizerC`* component is `ACNullC` in the case where no authorization is requested.

In contrast, a dynamic wire using either an **`enable`** or **`as`** clause is rewritten the same way except that the *`AuthorizerC`* component is `ACRT0C`. Furthermore, the list of enabled credentials is added to local certificate storage by Sprocket. Certificates in storage are periodically beaconed at run-time as described above. Finally, the entity on whose behalf the RPC invocation is performed is specified in the session key negotiation message sent to

*Dynamic Wire*

```
ClientC.I -> [SelectorC].I;
```

*Rewritten as...*

```
components Spkt_n;
ClientC.I -> Spkt_n;
Spkt_n.ComponentManager -> SelectorC;
Spkt_n.AuthorizationClient -> AuthorizerC;
Spkt_n.Packet -> AMSenderC;
Spkt_n.AMSend -> AMSenderC;
```

**Figure 3.10:** Dynamic Wire Rewriting

the server, also as described above.

The `Spkt_n` stub provides the same interface provided by `ClientC`. Wherever a duty is posted by `ClientC` in source code, the rewritten call invokes code in the stub that was specialized to handle that duty. The stub calls into the component manager at run time to obtain a list of the dynamic wire's endpoints and then prepares a data packet containing remote endpoint addresses and marshaled duty arguments. Finally, the stub calls through the `AuthorizationClient` interface to perform whatever authorization is needed.

### Remote Services

For nodes supporting RPC services, Sprocket generates a skeleton component for each remote interface provided. Figure 3.11 shows the form of a generated skeleton for an interface $I$ providing a single duty $d$ that takes a single integer parameter. This is for purposes of illustration; the scheme is generalized in an obvious manner. In general, the skeleton contains a task corresponding to each duty provided in the interface, and every generated skeleton is distinguished by a unique integer $n$ taken from the same numbering space as the generated stubs.

When messages are received on a node that provides RPC services, they are exam-

*Server Component*

```
module ServerC {
  provides remote interface I requires "A.g";
  other (local) uses/provides
}
```

*Skeleton generated as...*

```
module Spkt_n {
  uses interface I;
  uses interface Receive;
  uses interface AuthorizationServer;
}
implementation {

  int value_1;
  task void d() {
    call I.d(value_1);
  }

  event message_t *Receive.receive( ... ) {
    ...
  }
}
```

**Figure 3.11:** Server Skeleton Generation

ined to see if they are duty postings and thus to be handled by a skeleton. If so, the `AuthorizationServer` interface is used to authorize the message. If authorization succeeds, the task corresponding to the specified duty is posted. That task simply calls into *`ServerC`* through the original interface *`I`*. Thus the task-like behavior of duties is ultimately implemented using actual nesC tasks inside the server skeletons. Duty parameters are conveyed via module-level variables accessed by the duty tasks (since nesC tasks do not take formal arguments).

For each component that provides at least one remote interface, Sprocket creates a configuration as shown in Figure 3.12 that wires the corresponding skeleton(s) to that com-

```
configuration ServerC_SpktC {
    other (local) uses/provides
}
implementation {
    components ServerC, Spkt_n;
    Spkt_n.I -> ServerC;
    Spkt_n.Receive -> AMReceiverC;
    Spkt_n.AuthorizationServer -> AuthorizerC;
    pass local uses/provides directly to ServerC
}
```

**Figure 3.12:** Server Skeleton Wiring

ponent. This new configuration wraps the original component and replaces uses of the original component in other configurations in the program.

In this Figure, as is the case for client-side code, the *AuthorizerC* component is either ASRT0C or ASNullC depending on whether the original remote interface specified authorization or not.

# Chapter 4

# DScalaness/DnesT

This chapter describes a staged programming system supporting type safe dynamic code generation for resource constrained embedded devices. This system features programming abstractions for specializing device code and allowing on-the-fly adaptation to current device deployment conditions. The system has been implemented as an extension to Scala (Odersky, Spoon, and Venners 2011), through modification of the Scala compiler.

Specific consideration is given to scenarios where a relatively powerful hub device can automatically combine dynamically specialized libraries and deploy them to a wireless sensor network using some over-the-air deployment method such as Deluge (Hui and Culler 2004). To this end a restricted form of staging (Taha and Sheard 1997; Taha 2004; Consel, Hornof, Marlet, Muller, Thibault, Volanschi, Lawall, and Noyé 1998) is used to achieve well founded dynamic program generation. *First stage* code is written in an extended version of Scala, called Scalaness which is programmer friendly and suitable for running on powerful hubs. The execution of a Scalaness program yields a residual *second stage* node program written in nesT, a type safe variant of the nesC programming language. The second stage program is constructed from module components treated as first class values, which may be type and value specialized during the course of first stage computation.

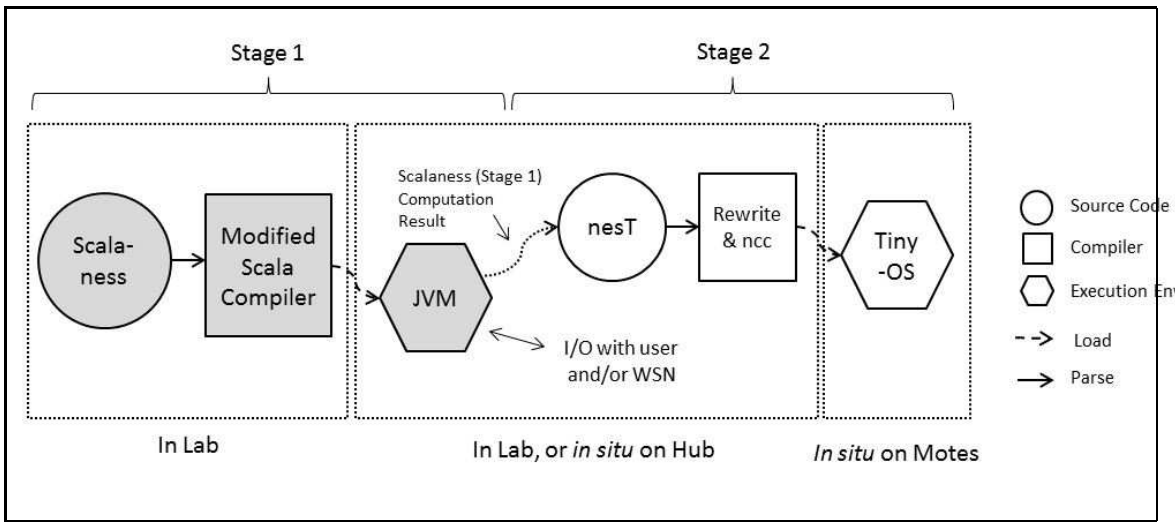


**Figure 4.1:** Scalaness/nesT Compilation and Execution Model

Figure 4.1 provides an overview of the Scalaness/nesT language architecture. Scalaness source code is compiled in a modified Scala compiler to Java bytecode, and run in a standard JVM. At runtime this Scalaness program may generate nesT code, which is subsequently rewritten to nesC and compiled using the standard TinyOS compiler. The resulting image can then be installed on sensor network nodes.

Since the Scalaness program has at its disposal the resources and features of the full Scala environment, including the JVM and its associated libraries, there are few limitations imposed on the first stage program. It could, at the programmer's option, generate separate images for each node on the sensor network or regenerate the node images at a later time to account for evolving network behavior.

Another interesting feature of the architecture, captured in Figure 4.1, is the physical platform on which different elements of the Scalaness/nesT "workflow" may be executed. Scalaness source code will typically be compiled in the lab, prior to deployment but execution of the Scalaness program may be done on a separate device in the field where the user will not be in a position to fix type errors in the generated images.

Consequently a central contribution of Scalaness is static type safety. In particular, the

Scalaness type system ensures that typeable Scalaness programs will always generate type safe residual nesT program. Since type generalization is allowed to be cross-stage, the technology supports a novel form of cross-stage type specialization. In existing strongly typed staged sensor programming environments the type correctness of second stage programs must be verified after execution of first stage code (Mainland, Morrisett, and Welsh 2008), and could in fact produce an error which would invalidate the deployment. Such type errors are always caught at first stage compilation time with Scalaness. Previous work on the staged programming calculus ⟨ML⟩ (Liu, Skalka, and Smith 2012) provides a theoretical foundation for Scalaness/nesT type safety.

Both Scala and nesC are complex, industrial strength languages. Neither of them are fully formalized. Thus in order to effectively study the Scalaness/nesT staged programming system, it is useful to consider simplified or "distilled" versions of those languages. Here, the names Scalaness and nesT refer to the languages as implemented, while *DScalaness* and *DnesT* refer to the distilled languages studied theoretically. The implementation is described in more detail in chapter 5. The distilled languages are described in this chapter.

## 4.1 Overview of DScalaness/DnesT Design

The goal of DScalaness/DnesT is to describe a practical programming system for writing arbitrary embedded applications. In that respect it is more general than SpartanRPC, which focuses on the specific problems of providing a convenient RPC discpline together with language level features for controlling resource access. In contrast, authorization is only one application of many to which Scalaness/nesT could be applied.

Scala is an appropriate choice as a basis for the first stage language because its compiler is open source and easy to modify and maintain. Also Scala offers a rich, flexible, and user friendly set of features familiar to application programmers working in traditional

(desktop, server) environments. Finally, Scala has an active user community with a growing collection of tools and other supporting resources.

However, Scala is not appropriate in the extremely resource constrained environment of a sensor network node or other small embedded system. In contrast nesT is implemented by translation into nesC, which can in turn be compiled for TinyOS platforms. The nesT language is roughly (although not exactly) a subset of nesC and shares with nesC many features appropriate for embedded systems programming.

DScalaness and DnesT will be demonstrated via an example that illustrates both type and value specialization of DnesT modules. Although the example is small it nevertheless demonstrates the essential features of the system in a suggestive manner.

It is well known that minimizing the number of bits used to represent a sensor node address can produce significant energy savings. Each bit of transmission consumes energy similar to 800 instructions (Madden, Franklin, Hellerstein, and Hong 2002) so the fewer bits transmitted the better. However, sensor networks are "ad hoc" in the sense that the distribution of the nodes is often unpredictable until deployment, so the optimal data type used to represent node addresses is an environmental property that may need to be determined *in situ*.

The example also value specializes DnesT modules with specific session keys for use during secure communication. In particular, as with SpartanRPC, communication between security domains in a sensor network can be mediated by credentials implemented as keys, with nodes lying at domain frontiers using different keys to send (to the other domain) and receive (from the other domain) over secured links. Since it is unpredictable where nodes will be physically distributed in space, appropriate keys for each node need to be established *in situ*. Defining node functionality using generic code that must be instantiated with specific keys allows adaptation to a deployment environment, and allows expensive computations for establishing session keys to be offloaded from the network nodes to a

higher powered device.

Figure 4.2 shows the complete example. The definition begins with a parameterized type `mesgT(t)` using the DScalaness `abbrvt` binder, where an instance `mesgT`($\tau$) denotes the ground type obtained by substituting $\tau$ for `t` in the definition of `mesgT`.

Next, on line 3 defines a type `radioT`, which is the type of DnesT modules that provide an API to the radio. The DnesT module language is a simplified version of the nesC component language. In this example, any module of type `radioT` exports a `radio_x` function for sending messages, and imports a `handle_radio_r` function that allows received messages to be handled in a user-defined manner. Both functions take message references as arguments. Furthermore, the module is parameterized by the type of node addresses `at`, upper-bounded by 32-bit unsigned integer. Thus, any module of type `radioT` can be dynamically specialized to an 8, 16, or 32 bit address space. Module type parameters are always defined with brackets $< ... >$.

Now on line 7 another type `commT` is defined which is the type of modules providing a QOS layer over a specialized radio. Although this type is also parameterized by a bounded address type `at`, as is `radioT`, the parameterization is subtly different syntactically and semantically, since `commT` expects a program context where the radio has been specialized. Thus, in `commT`, `at` is understood as being "some" type with an upper bound of `uint` which occurs in the module signature, whereas the module itself has no parameters to be instantiated. This sort of type is needed in the presence of *dynamic type construction*.

Next on lines 12 and 17 modules are defined for sending and receiving messages that provide a layer of authentication security over the radio. Observe that in the implementation of `send` in module `authSend`, messages are signed with a key `sendk`, whereas when messages are received they must be signed with a possibly different key `recvk` before being passed on to the user's receive handler, as specified in module `authRecv`. These modules are parameterized by an address type `at`, and also the `sendk` and `recvk` key values.

```
abbrvt mesgT(t) =
 { src : t; dest : t; data : uint8[] };
abbrvt radioT =
 < at ≼ uint >
 { export error_t radio_x(mesgT(at)*);
   import error_t handle_radio_r(mesgT(at)*); };
abbrvt commT =
 (at ≼ uint) ∘ < >
 { export error_t send(mesgT(at)*);
   import error_t handle_receive(mesgT(at)*); };

val authSend =
 < at ≼ uint; sendk : uint8[] >
 { import error_t radio_x(mesgT(at)*);
   export error_t send(m : mesgT(at)*)
       { radio_x(AES_sign(m, sendk)); } };
val authRecv =
 < at ≼ uint; recvk : uint8[] >
 { import error_t handle_recv(mesgT(at)*);
   export error_t handle_radio_r(m : mesgT(at)*)
       { if (AES_signed(m, recvk))
             handle_recv(m); } };

def authSpecialize
 (nmax   : uint16,
  radioM : radioT,
  keys   : Array[Array[uint8]]) : commT
    typedef adt ≼ uint =
      if (nmax <= 256) uint8 else uint16;
    val sendM = authSend⟨mesgT(adt); keys(0)⟩;
    val recvM = authRecv⟨mesgT(adt); keys(1)⟩;
    sendM ⋉ radioM⟨mesgT(adt)⟩ ⋉ recvM;
   }

val appMR =
 < >{ export handle_recv(m : mesgT(uint8)*) {...} };
val appM =
 < >{ import send(mesgT(uint8)*); export main() {...} };
image(appM ⋉
        authSpecialize(nmax, radioM, keys) ⋉ appMR); }
```

**Figure 4.2:** DScalaness/DnesT Example

To generalize a technique for composing these modules with a radio to yield a module of type `commT`, that is abstract with respect to neighborhood sizes, radio implementations, and session key material, the DScalaness `authSpecialize` function is defined on line 24. The first-class status of DnesT modules in DScalaness is apparent here. Starting at line 25 the method is specified to take a module parameter `radioM` of type `radioT` among its arguments, and to return a module of type `commT` as a result. It also takes an array of keys as an argument, and on lines 30 and 31 it instantiates `sendMesg` and `recvMesg` with the keys in the array. Finally it uses the type `adt` in the instantiations, which in line 28 is dynamically constructed on the basis of the input variable `nmax`.

This illustrates a key novelty of the system, the ability to *dynamically* set a type to use on a node based on a decision made during the execution of the DScalaness program. Since the value of `nmax` cannot be statically determined, the type analysis only knows that `adt` is some subtype of `uint`. Finally, on line 32 the instantiated radio module is composed with the instantiated send and receive modules via the DScalaness $\ltimes$ operator. The semantics of module composition here is standard (Cardelli 1997); in a composition aka wiring $\mu_1 \ltimes \mu_2$, the exports of $\mu_2$ are connected to imports of $\mu_1$. The function result is a module of type `commT`.

To obtain a module defining a node image in a program context where neighborhood size is known, a radio implementation has been provided, and session keys have been computed. The results can then be composed of an `authSpecialize` function with modules specifying top-level message send and receive behaviors, and a `main` application entry point (here knowledge that address sizes can be limited to 8 bits is assumed, so `nmax` $<$ 256). At line 40 a closed module is defined and a binary mote image can be produced by a call to `image`.

In DScalaness, `image` is an assertion that its argument is a *runnable* module, with no unresolved parameters or imports. In the Scalaness implementation, this is the point

where nesT source code is actually generated. Successful DScalaness/DnesT type checking (which occurs during stage 1 compilation as per Figure 4.1) statically guarantees that specialized code generated at the point of `image` will run in a type-safe manner when it is eventually loaded and run on a node.

### 4.1.1 Modules as Staging Elements

In DScalaness, DnesT modules can be treated as data to be composed, following traditional staged programming languages (Taha and Sheard 1997). The so-called "runnable" modules—ones without imports or generic parameters—define an initial machine configuration. This supports a TinyOS node reprogramming model where the entire OS is recompiled and target nodes are reimaged and rebooted. The `image` operation (invoked in line 40 of the example) asserts its argument to be runnable and dumps the module code for subsequent deployment.

DnesT modules specify a list of imported function signatures, and a list of exported functions implemented by the module. Module genericity is obtained via a sequence of type and value parameter definitions. For example, as specified in Figure 4.2, any module `radioC` satisfying `radioT` has an address type parameter `adt` which specializes the message type declared in the exported `radio` function.

All generic type parameters are assigned an upper *type bound* via the subtyping symbol $\preccurlyeq$. The `nodeC` module additionally has value parameters `self` and `neighbor`, which are type cast during the call to the imported `send` function. The concrete syntax used in Figure 4.2 precedes each import/export definition with keyword `import`/`export` for a more readable presentation; this is not part of the formal DnesT syntax.

The other DScalaness operations on DnesT modules are instantiation and composition. In lines 30 and 31, the modules `authSend` and `authRecv` are instantiated with

arguments specifying the type of message to be used, parameterized by a dynamically constructed type, and by the value of the desired key. In lines 32 and 40, modules are composed using the $\ltimes$ operator. The semantics of module composition $\mu_1 \ltimes \mu_2$ is standard (Cardelli 1997); imports of one module are connected to exports of the other. DnesT module composition is analogous to nesC *configuration wiring*.

### 4.1.2 Typing

The most novel feature of the DScalaness type system is dynamic type construction. Dynamic construction of DnesT types is allowed at the DScalaness level for module instantiation and specialization. On line 28 in Figure 4.2, the address type `adt` is dynamically constructed via a conditional expression.

Scala type checking has been formally studied and shown to be decidable (Cremet, Garillot, Lenglet, and Odersky 2006). DScalaness type checking is an extension of Scala type checking; a new module type form is introduced to the Scala type language, type checking cases have been added for the three module operations: instantiation, composition, and imaging. No other part of Scala type checking needs to be modified. DnesT type checking is defined as a standalone type system, and yields first stage module types.

### 4.1.3 Cross-Stage Migration of Types and Values.

A crucial feature of the DScalaness/DnesT programming model is *process separation* between stages (Liu, Skalka, and Smith 2012). Since first and second stage code are to be run on separate devices, state is not shared between these stages. Thus, serialization may be required when modules are instantiated. Furthermore, types and base values may be represented differently on the first and second stages, requiring some sort of transformation during module instantiation. An example transformation is discussed in subsection 4.3.3.

## 4.2 The DnesT language

The DnesT design aims to distill a production language, nesC, into its fundamental elements, yielding a language that is amenable to formal analysis but also practical. However, since the focus here is on type safety, DnesT enhances, and in some instances restricts, those fundamental elements to obtain a type safe language, as discussed below.

**Notation**

*Sequences* are notated $x_1, \ldots, x_n$, and are abbreviated $\overline{x}$; $\overline{x}_{(i)}$ is the $i$-th element, $\emptyset$ denotes the empty sequence, and $|\overline{x}|$ is the size. The notation $x \in \overline{x}$ denotes membership in sequences, and $x\overline{x}$ denotes a sequence with head $x$ and tail $\overline{x}$. Append is denoted as $\overline{x}@\overline{y}$. For relational symbols $R \in \{\preccurlyeq, =, :\}$, the abbreviation: $\overline{x}\, R\, \overline{y} = x_1\, R\, y_1, \ldots, x_n\, R\, y_n$ is used. So for example, $\overline{x} : \overline{\tau} = x_1 : \tau_1, \ldots, x_n : \tau_n$.

The following naming conventions are also used for various language constructs. Meta-variable $\mathsf{f}$ (of set $\mathcal{F}$) is used for function names, $\mathsf{l}$ (of set $\mathcal{L}$) for field names, *x* (of set $\mathcal{V}$) for term variables, *t* (of set $\mathcal{T}$) for type variables, $m$ (of set $\mathcal{M}$) for memory locations, $n^8$ (of set $\mathbb{Z}_{2^8}$) for 8-bit unsigned integers, and $n^{16}$ (of set $\mathbb{Z}_{2^{16}}$) for 16-bit unsigned integers. Finally, use of $n$ to range over both types of integers when their type is irrelevant.

### 4.2.1 Syntax and Features of DnesT

The syntax of DnesT is presented in Figure 4.3. It comprises a core language of expressions for defining computations, a language of declarations for defining variables and functions, and a language for defining modules.

$$
\begin{array}{lll}
\varsigma, \tau & ::= t \mid \top \mid \texttt{uint8} \mid \texttt{uint16} \mid \texttt{uint} \mid \texttt{uninit} \mid & \textit{types} \\
 & \quad \{\overline{\mathsf{l}} : \overline{\tau}\} \mid \tau[] \mid \tau\texttt{*} & \\
e & ::= v \mid \ell e \mid e \; op \; e \mid (\tau)\, e \mid \mathsf{f}\texttt{()} \mid \ell e = e \mid\mid \&\ell e \mid e \rhd e \mid & \textit{expressions} \\
 & \quad \texttt{if}\ (e)\ e\ \texttt{else}\ e \mid \texttt{while}\ (e)\ e \mid e; e \mid \texttt{post}\ \mathsf{f}\texttt{()} & \\
\ell e & ::= x \mid e[e] \mid e.\mathsf{l} \mid \texttt{*}e & \textit{l-values} \\
v & ::= n^8 \mid n^{16} \mid \texttt{uninit} & \textit{base values} \\
op & ::= + \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{==} \mid \ldots & \textit{operations} \\
id & ::= \mathsf{f} \mid x & \textit{identifiers} \\
c & ::= \mathsf{f}(V) : \tau = \{e\} & \textit{command definition} \\
s & ::= \mathsf{f}(V) : \tau & \textit{command signature} \\
d & ::= \tau\ x\ =\ e \mid \tau\ x\ =\ [\overline{e}\,] \mid \tau\ x\ =\ \{\overline{\mathsf{l}} = \overline{e}\} \mid c & \textit{declarations} \\
T & ::= \overline{t} \preccurlyeq \overline{\tau} & \textit{type parameters} \\
V & ::= \overline{x} : \overline{\tau} & \textit{value parameters} \\
\iota & ::= \overline{s} & \textit{imports} \\
\xi & ::= \overline{c} & \textit{exports} \\
\varepsilon & ::= \overline{s} & \textit{export signatures} \\
\mu & ::= <T; V>\{\iota; \overline{d}; \xi\} & \textit{module definitions} \\
\mu\varpi & ::= <T; V>\{\iota; \varepsilon\} & \textit{module signatures}
\end{array}
$$

**Figure 4.3:** Program Syntax of nesT

### Expressions

The DnesT language includes standard C-like constructs, such as conditional branching, looping, sequencing of expressions, and function calls, arrays, structs, numeric base data-types (and operations on them). Familiar syntax $e[e]$ and $e.\mathsf{l}$ is used for array indexing and structure field selection, respectively.

A "null" value **`uninit`** is also provided. Function definitions and calls allow multiple parameters as is usual in C-like languages. Memory locations $m$ are program values (*á la* pointers). As in all C dialects, assignment can only be performed on so-called *l-values* $\ell e$, a restricted subset of expression syntax.

As in nesC, no dynamic memory allocation is possible; all memory layout is established by static variable declarations. However, unlike nesC full pointer arithmetic is not supported for the sake of type safety. Instead an array increment operator $\triangleright$ allows the suffix of an array to be computed based on an integral shift distance. Type casting and array operations have run time checks imposed, as explained in subsection 4.2.2.

Also as in nesC, a **`post`** operation is provided for posting tasks. The semantics of tasks follow the "run-to-completion" model of TinyOS. Interrupts are omitted from the language to avoid concurrency issues in the semantics. Typical sensor network applications do not use interrupts directly; they are needed instead in low level libraries.

### Declarations

Programs in DnesT may refer to declarations of values and functions which are scoped at the module level and establish statically the memory layout of a DnesT module. A convenient form for explicit initialization of array and struct values is provided, though there are no base values for either arrays or structures. Besides convenience, declarations are useful to support serialization of program objects passed in from DScalaness, as discussed

in subsection 4.3.3.

In the formal syntax and semantics of DnesT presented here, functions are nullary (i.e., parameterless) as a simplification. This does not limit expressivity since DnesT does not permit recursion. Thus function parameters can be simulated in principle by way of module level declarations. The actual implementation of nesT allows functions to be parameterized as in nesC. Accordingly, syntactic liberties are taken in the examples in this chapter and make use of non-nullary functions.

#### Modules

DnesT Modules are written $<T; V>\{\iota; \overline{d}; \xi\}$ with *T* and *V* being generic type and term parameters, $\overline{d}$ being module scope identifier declarations, including function definitions, and $\iota$ and $\xi$ being imports and exports. Exports are explicitly defined in the module.

A "runnable" module—one without imports or parameters—is the DnesT model of a device image. The declarations in the module defines an initial machine configuration, and the application entry point is defined in a required command `main`.

**Definition 4.2.1** *A module of the form* $<\varnothing; \varnothing>\{; \overline{d}; \xi\}$*, where* `main()` : **`uninit`** $\in \xi$*, is called* runnable.

### 4.2.2 Semantics of DnesT

The operational semantics for DnesT is defined as a small step relation $\rightsquigarrow$ on dynamic configurations in Figure 4.4. The semantics are decomposed into several distinct $\rightsquigarrow$ relations; each computation "sub-relation" can be distinguished by the arity of the relevant configurations. The semantics for the non-module portions follows standard C-like language formalizations (Leroy 2006; Grossman 2003).

$$
\begin{array}{rcll}
\pi & ::= & m \mid \circ & \textit{l/r tags} \\
\kappa & ::= & \lfloor \pi, \nu \rfloor & \textit{dynamic objects} \\
\ell e & ::= & \cdots \mid \lfloor m, \nu \rfloor & \\
e & ::= & \cdots \mid \kappa & \\
\nu & ::= & v \mid m \mid [\overline{\kappa}] \mid \{\overline{\mathsf{l}} = \overline{\kappa}\} & \textit{dynamic values} \\
F & ::= & \overline{\mathsf{f}} : \overline{\tau} = \overline{(e)} & \textit{codebase} \\
M & ::= & \overline{m} : \overline{\tau} = \overline{\nu} & \textit{memory} \\
E & ::= & \{\} \mid E; e \mid \kappa[E] \mid E[e] \mid E.\mathsf{l} \mid \kappa = E \mid E = e \mid a(\tau)\, E \mid & \textit{evaluation contexts} \\
 & & \texttt{*}E \mid \&E \mid E \; \textit{op} \; e \mid \kappa \; \textit{op} \; E \mid \texttt{if}\ (E)\ e\ \texttt{else}\ e \mid \texttt{while}\ (E)\ e & \\
\ell & ::= & \mathbf{boot}(\overline{d}) \mid \mathbf{run}(e) & \textit{run levels}
\end{array}
$$

**Figure 4.4:** Syntactic Definitions for Dynamic Configurations

**Semantics of Expressions**

At the heart of DnesT is a C-like language of expressions built from l- and r-values. An l-value is an object in memory that can be the target of an assignment, in particular a variable, a structure field, an array member, or a dereferenced pointer. An r-value is a value resulting from expression computation and may be a value that is not necessarily in memory.

The DnesT syntax for necessary dynamic entities is given in Figure 4.4. In DnesT, computed values are represented as pairs $\lfloor \pi, \nu \rfloor$, where $\nu$ is a base, pointer, array, or structure value, and $\pi$ is a tag indicating whether or not the value is in memory. Computed r-values not in memory are denoted $\lfloor \circ, \nu \rfloor$, e.g., $\lfloor \circ, 2 \rfloor$ is the result of computing 1 + 1. The l-value object $\lfloor m, \nu \rfloor$ indicates that the value $\nu$ is in memory at location $m$.

Memories are modeled as sequences of definitions $m : \tau = \nu$; observe that each memory location $m$ is typed at $\tau$ and assigned a value $\nu$. Memories are interpreted as mappings, writing $M(m) = \nu$ when there exists some $\tau$ such that $m : \tau = \nu$ is the leftmost definition

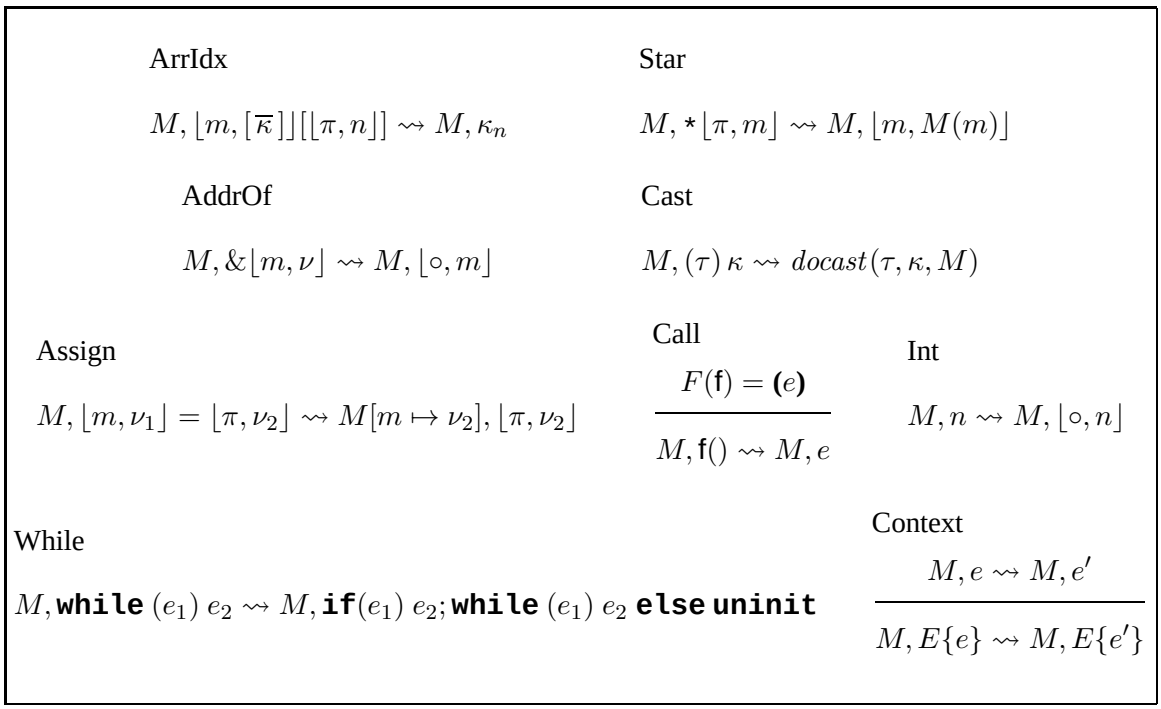


**Figure 4.5:** Dynamic Semantics of Selected Expressions

of $m$ in $M$, and writing $M[m \mapsto \nu]$ to denote $(m : \tau = \nu)$ where $m : \tau = \nu'$ is the leftmost definition of $m$ in $M$.

Evaluation rules for selected expressions are given in Figure 4.5. Here, computation is on pairs of memories and expressions. The existence of a function $\mathit{docast}$ is assumed, which performs a casting conversion. This function is allowed to be defined by users, and in certain cases may be a no-op (e.g., casting pointers to arrays when the latter are contiguous in memory). In any case, if $\mathit{docast}(\tau, \nu, M)$ is defined, the cast conversion is required to be type safe, in that the result must be of type $\tau$. This is discussed more in subsection 4.2.3.

Note that a pointer is modeled by an object of the form $\lfloor \pi, m\rfloor$. The operation $*\lfloor \pi, m\rfloor$ looks up the value at address $m$ in memory. The operation $\&\lfloor m, \nu\rfloor$ returns the address $m$ of the object in memory as an r-value. Functions are defined in an assumed-given codebase $F$ with a lookup semantics defined similarly to that for memories $M$.

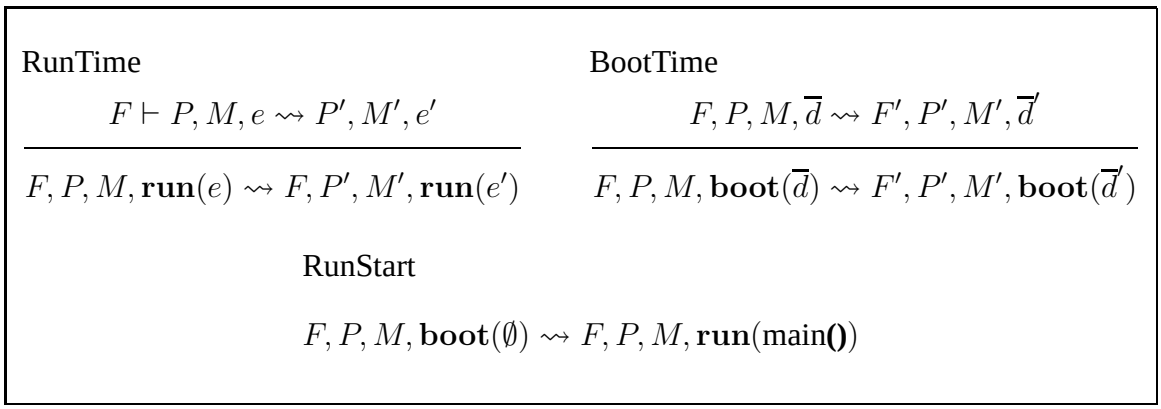


**Figure 4.6:** Boot and Runtime Semantics

### Semantics of Tasks

NesC uses a simple scheduling model of serial, run-to-completion execution of queued *tasks* where each task is defined by a parameterless function. The base semantics of DnesT are thus supplemented with a corresponding *task collection* $P$ of the tasks yet to run, and defined with a single step transition relation on configurations extended with task collections. The definition of task collections is left undetermined, and also how tasks are added and retrieved—this because it is unspecified how tasks are treated by the scheduler. The notation $add(P, \mathsf{f()})$ denotes $P'$ which is $P$ plus the task consisting of the function call **f()**, and $next(P)$ denotes a pair $P', \mathsf{f()}$ which comprises the "next" task **f()** in $P$, and $P'$ which is $P$ with **f()** removed. The task semantics, integrated with the expression semantics defined previously, are defined in Figure 4.7. As for expressions, the existence of a given codebase $F$ is assumed. When it is necessary to be explicit about which codebase is given for a computation, $F \vdash P, M, e \rightsquigarrow P', M', e'$ will be written.

### Semantics of Declarations

The operational behavior of declarations is fairly straightforward and is shown in Figure 4.8. Functions and first class mutable variables may be declared and initialized. At run time mutable variables are bound (via substitution) to an l-value $\lfloor m, \nu \rfloor$, where $m$ is the address

CoreStep

$$\frac{M, e \rightsquigarrow M', e'}{P, M, e \rightsquigarrow P, M', e'}$$

Post

$$P, M, E\{\texttt{post f()}\} \rightsquigarrow add(P, \texttt{f()}), M, E\{\texttt{uninit}\}$$

TaskStart

$$\frac{next(P) = P', \texttt{f()}}{P, M, m \rightsquigarrow P', M, \texttt{f()}}$$

**Figure 4.7:** Semantics of Tasks and Configurations

of the variable. Thus, for base and function type declarations the following rules apply, respectively:

FDecl

$$F, P, M, (\texttt{f} : \tau = \textbf{(}e\textbf{)})\overline{d} \rightsquigarrow (\texttt{f} : \tau = \textbf{(}e\textbf{)})F, P, M, \overline{d}$$

BaseInit

$$\frac{\kappa = \lfloor m, \nu \rfloor \qquad m \notin \mathrm{Dom}(M)}{F, P, M, (\tau\ x\ =\ \lfloor \pi, \nu \rfloor)\overline{d} \rightsquigarrow F, P, (m : \tau = \nu)M, \overline{d}[\kappa/x]}$$

A contextual evaluation rule for declarations allows variables to be initialized with arbitrary expressions. This is omitted for brevity but is similar to the expression Context rule, using a notion of declaration evaluation contexts denoted $D$.

### Semantics of Boot and Run Time

In the DnesT machine model, a top level program execution is obtained by loading and running a fully instantiated module. The codebase, memory layout, and initial machine configuration is generated at load time by evaluating the declarations in the module. The top level program is then started at the `main` entry point.

To differentiate load/boot and run segments of a computation, $\mathbf{boot}()$ and $\mathbf{run}()$ constructors are defined to inject declarations and expressions into a uniform datatype. Top

DeclContext

$$\frac{F \vdash P, M, e \rightsquigarrow P', M', e'}{F, P, M, D[e]\overline{d} \rightsquigarrow F, P', M', D[e']\overline{d}}$$

BaseInit

$$\frac{\kappa = \lfloor m, \nu \rfloor \qquad m \notin \mathrm{Dom}(M)}{F, P, M, (\tau\ x\ =\ \lfloor \pi, \nu \rfloor)\overline{d} \rightsquigarrow F, P, (m : \tau = \nu)M, \overline{d}[\kappa/x]}$$

StructInit

$$\frac{\kappa = \lfloor m, \nu \rfloor \qquad m \notin \mathrm{Dom}(M)}{F, P, M, (\tau\ x\ =\ \nu)\overline{d} \rightsquigarrow F, P, (m : \tau = \nu)M, \overline{d}[\kappa/x]}$$

FDecl

$$F, P, M, (\mathsf{f} : \tau = \textbf{(}e\textbf{)})\overline{d} \rightsquigarrow (\mathsf{f} : \tau = \textbf{(}e\textbf{)})F, P, M, \overline{d}$$

**Figure 4.8:** Semantics of Declarations

level computation is then defined as a single step reduction relation $\rightsquigarrow$ on configurations $F, P, M, X$, where $X$ is of the form $\mathbf{boot}(\overline{d})$ or $\mathbf{run}(e)$ depending on whether the machine is booting or running.

**Definition 4.2.2** *For a runnable module of the form* $<\varnothing; \varnothing>\{; \overline{d}; \xi\}$ *the following is defined:*

$$bootload(<\varnothing; \varnothing>\{; \overline{d}; \xi\}) = \xi, \varnothing, \varnothing, \mathbf{boot}(\overline{d})$$

Now, for all computation relations $\rightsquigarrow^*$ is defined to be the reflexive, transitive closure of $\rightsquigarrow$. The concern for type analysis is to rule out modules which, when bootloaded, will evaluate to semantically ill-formed configurations. In the context of DnesT this is defined as follows. Notice that failing casts and out-of-bound array access are not stuck cases, since run time checks enable graceful failure behavior.

**Definition 4.2.3** *A configuration* $M, e$ fails a run time check *if and only if* $e$ *is of the form* $(\tau)\,\kappa$ *and* $docast(\tau, \kappa, M)$ *is undefined, or* $e$ *is of the form* $\lfloor m, \nu \rfloor[\lfloor \pi, n \rfloor]$ *and* $n \geq |\overline{\nu}|$.

**Definition 4.2.4** *A configuration* $F, P, M, \ell$ *is* stuck *if and only if it is irreducible and* $\ell$ *is neither of the form* $\mathbf{run}(E\{e\})$ *nor* $\mathbf{boot}(D\{e\})$ *where* $M, e$ *fails a run time check. A runnable module* $\mu$ goes wrong *iff* $bootload(\mu) \rightsquigarrow^{\star} F, P, M, \ell$ *where* $F, P, M, \ell$ *is stuck.*

### 4.2.3 DnesT Type Checking

The typing rules for DnesT combine a standard procedural language typing approach with subtyping techniques adapted from previous foundational work (Liu, Skalka, and Smith 2012; Ghelli and Pierce 1998). The goal here is to specify the typing algorithm used in the DScalaness implementation.

#### Subtyping

At the heart of the system is a decidable subtyping judgement $T \vdash \tau_1 \preccurlyeq \tau_2$, where $T$ in the context of typing is called a *coercion* and defines a system of upper bounds for type variables. Recursive type bounds definitions are not allowed.

The implementation of the subtyping algorithm is based on a classic technique (Ghelli and Pierce 1998), with straightforward extensions to accommodate structures and arrays as defined in Figure 4.9.

A subtyping relation typically called *promotion* is also central to the approach; given a set of subtyping coercions $T$ and a type variable $t$, promotion will return the least upper bound of $t$ which is also a structured type, i.e., not a type variable.

**Definition 4.2.5** *The relation* $\ll$ promotes *a type variable:*

$$\frac{T \vdash T(t) \ll \tau}{T \vdash t \ll \tau} \qquad \frac{\neg \exists t.\tau = t}{T \vdash \tau \ll \tau}$$

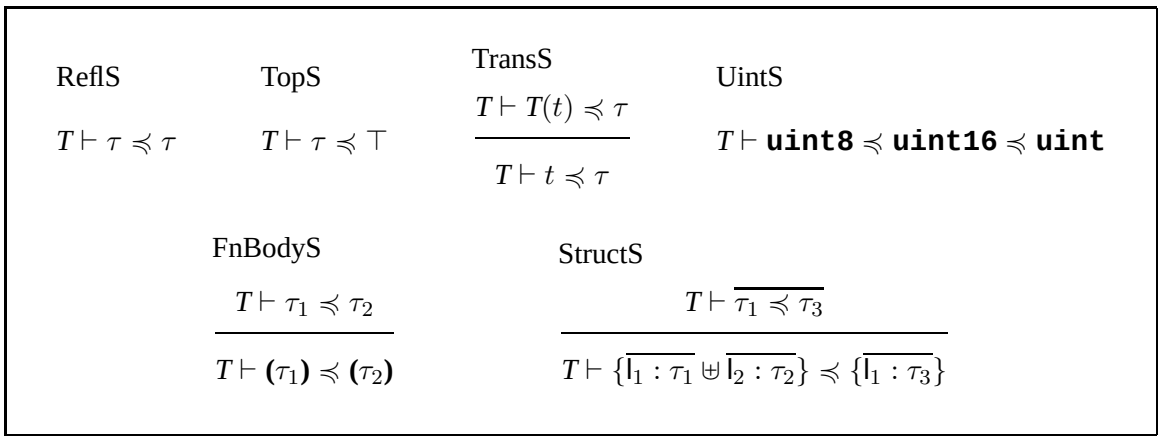


**Figure 4.9:** Subtyping Rules

It is important to observe how promotion and subtyping are used differently. Since any sort of l-value can be written to via assignment, subtyping invariance *must* be imposed on l-values occuring in write positions to maintain type soundness. Therefore type subsumption is allowed only at program points where read-only control flow occurs—for example when an r-value is directly assigned to an l-value.

### Type Environments and Checking

The typing algorithm for source code expressions is based on judgements $G, T \vdash e : \tau$, where $G$ is an environment of free term variable typings, syntactically defined equivalent to value parameters $V$ and imports $\iota$. Type environments are also endowed with the same lookup semantics as memories and codebases. Representative typing rules for selected expressions are given in Figure 4.10. The derivation of any judgement $G, T \vdash e : \tau$ can be interpreted as an algorithm where both $G$ and $T$ are given as arguments and $\tau$ is returned as a result.

Note that type casting is only statically allowed if the types involved are *compatible* as specified in rule CastT. This relation, formalized as $T \vdash compatible(\tau_1, \tau_2)$, is left abstract and user defined. Recall that the semantics of DnesT relies on a $docast$ function that im-

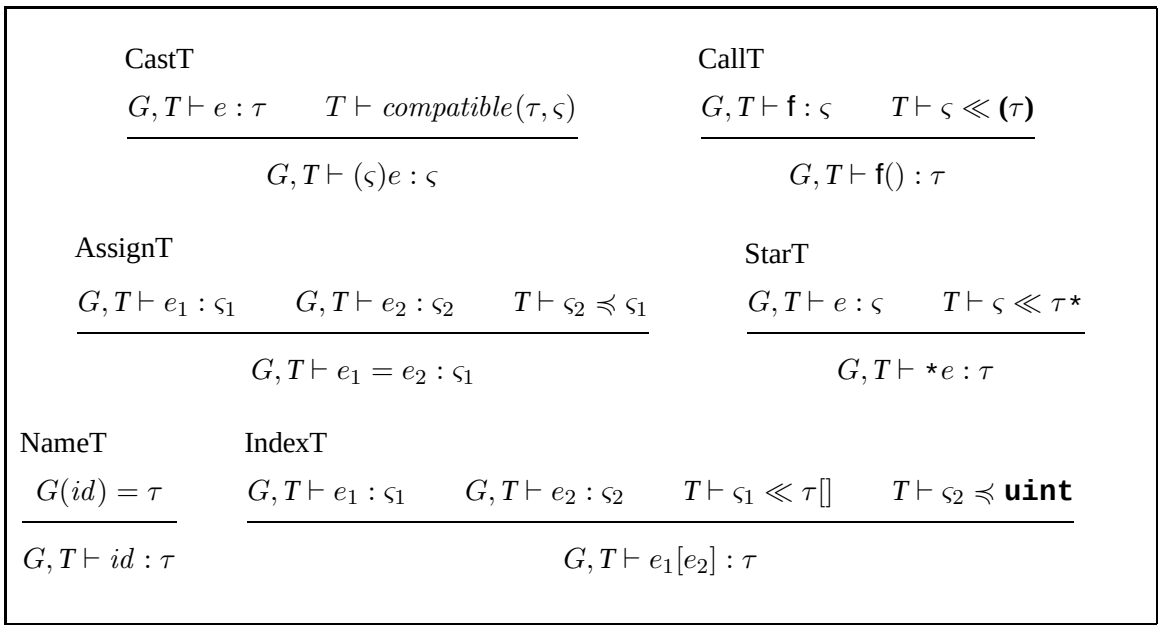


**Figure 4.10:** Typing Rules for Selected DnesT Expressions

plements cast conversions. Any implementation of $docast$ must be type safe, which allows the ruling out of run time cast failures in well typed programs. Informally, $docast$ is type safe if and only if the resulting expression has the type of the cast. Refer to (Liu, Skalka, and Smith 2012) for a thorough formal discussion of type safety for this style of casting.

### Declaration and Module Typings

At the module level, it is necessary to first type check and generate typing environments from declarations, as specified in Figure 4.11 (rules for array and struct declarations omitted for brevity). Given this, a module typing is obtained by type checking module exports, using a coercion obtained from the module type parameters and a typing environment obtained from a combination of module value parameters, imports, and variable type declarations. Module type checking is also specified in Figure 4.11. A type safety conjecture for DnesT can then be stated as follows.

**Conjecture 4.2.1 (DnesT Type Safety)** *If $\mu : \mu\pi$ is valid and $\mu$ is runnable, then $\mu$ does*

DeclsNoneT

$$\frac{}{G, T \vdash \varnothing : \varnothing}$$

DeclsSomeT

$$\frac{G, T \vdash d \Rightarrow (x : \tau) \qquad (x : \tau)G, T \vdash \overline{d} : G'}{G, T \vdash d\overline{d} : (x : \tau)G'}$$

DeclBaseT

$$\frac{G, T \vdash e : \tau}{G, T \vdash \tau\ x\ =\ e \Rightarrow x : \tau}$$

DeclFunT

$$\frac{G, T \vdash e : \tau}{G, T \vdash \mathsf{f} : (\tau) = (e) \Rightarrow \mathsf{f} : (\tau)}$$

ModuleT

$$\frac{\iota@V, T \vdash \overline{d} : G \qquad G@\iota@V, T \vdash \xi : \varepsilon}{<T, V>\{\iota; \overline{d}; \xi\} : <T, V>\{\iota; \varepsilon\}}$$

**Figure 4.11:** Selected Declaration and Module Typing Rules

*not go wrong.*

## 4.3 The DScalaness Language

DScalaness serves as the language for DnesT module composition in the same manner as nesC configurations serve to compose nesC modules, but DScalaness is a more powerful metalanguage since modules are treated as a new category of first class values in DScalaness. Instantiation, composition ("wiring"), and imaging of modules are defined as operations on module values. Because instantiation of modules with both types and values is allowed, values and types may migrate from the DScalaness level to the DnesT level, realizing a disciplined form of code specialization.

The goal of this section is to describe the DScalaness syntax and semantics realized in the implementation, and justify the prior claims of type safety. Since Scala as implemented is too large to easily formalize, the formalization here is of distilled subset of Scala,

| | | | |
|---|---|---|---|
| L | ::= | $\texttt{class C}\langle\bar{\texttt{X}} <: \bar{\texttt{N}}\rangle\ \texttt{extends N}\ \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \texttt{K}\ \bar{\texttt{M}}\}$ | *class definitions* |
| K | ::= | $\texttt{C}(\bar{\texttt{T}}\ \bar{\texttt{f}})\{\texttt{super}(\bar{\texttt{f}});\ \texttt{this.}\bar{\texttt{f}} = \bar{\texttt{f}};\}$ | *constructors* |
| M | ::= | $\texttt{T m}(\bar{\texttt{T}}\ \bar{\texttt{x}})\{\texttt{return e};\}$ | *methods* |
| e | ::= | $\texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\bar{\texttt{e}}) \mid \texttt{new C}\langle\bar{\texttt{T}}\rangle(\bar{\texttt{e}}) \mid (\texttt{N})\texttt{e} \mid \texttt{e.f} = \texttt{e} \mid \texttt{l} \mid$ | *expressions* |
| | | $\texttt{def x} : \texttt{T} = \texttt{e in e} \mid \mu \mid \texttt{e} \ltimes \texttt{e} \mid \texttt{e}\langle\bar{\texttt{e}}; \bar{\texttt{e}}\rangle \mid \texttt{image e} \mid$ | |
| | | $\texttt{abbrvt X}(\bar{\texttt{X}}) = \texttt{T in e}$ | |
| T | ::= | $\texttt{X} \mid \texttt{N} \mid T \circ \mu\pi$ | *scala level types* |
| N | ::= | $\texttt{C}\langle\bar{\texttt{T}}\rangle$ | *class types* |
| l | ::= | $(\texttt{p}, \texttt{N})$ | *references* |

**Figure 4.12:** The Syntax of DScalaness

*DScalaness*, that has also been extended to include syntax and semantics for definining and composing DnesT modules. A formalized core calculus and type analysis for Scala exists (Cremet, Garillot, Lenglet, and Odersky 2006), but the formalization presented here is adequate and simpler. Many features of Scala have been elided in DScalaness, but all Scala features are adopted unchanged in the Scalaness implementation. Here the focus is primarily on the module metaprogramming operations that have been added. This presentation "cleans up" some implementation details, but is otherwise an accurate description of the module operation semantics and especially the module operation typing rules.

### 4.3.1 Syntax of DScalaness

The DScalaness language syntax is presented in Figure 4.12. To represent an adequate core calculus of Scala, it subsumes two Featherweight Java variants: Featherweight Generic

Java (FGJ) (Igarashi, Pierce, and Wadler 2001) and Assignment Featherweight Java (AFJ) (Molhave and Petersen 2005). The generic class types of FGJ are needed to model type construction, and the mutation in AFJ is essential to consider since one main concern is DnesT code specialization; DnesT programs are run in a separate process space, so specialization with stateful values, a likely common idiom in a Scala setting, presents a challenge.

Refer to (Igarashi, Pierce, and Wadler 2001; Molhave and Petersen 2005) for details on the FGJ and AFJ object oriented calculi, which are represented in the languages of class definitions, constructors, methods, and the first line of expression forms defined in Figure 4.12. DScalaness extends these features with a typed variable declaration form $\texttt{def x : T} = \texttt{e}_1 \texttt{ in e}_2$ where the scope of $\texttt{x}$ is $\texttt{e}_2$, and a dynamic type construction form $\texttt{typedef x} <: \texttt{T} = \texttt{e}_1 \texttt{ in e}_2$ (defined as syntactic sugar in Definition 4.3.1) with similar scoping rules. For programming convenience a simple parameterized type abbreviation binder `abbrvt` is also provided.

DnesT modules $\mu$ are included in the DScalaness expression and value spaces: instantiation is obtained via the form $\texttt{e}_1\langle\bar{\texttt{e}}_1; \bar{\texttt{e}}_2\rangle$, where $\bar{\texttt{e}}_1$ are type parameters and $\bar{\texttt{e}}_2$ are value parameters. Wiring of modules is denoted $\texttt{e}_1 \ltimes \texttt{e}_2$. Imaging of modules, denoted `image` $\texttt{e}$, ensures that $\texttt{e}$ computes to a runnable module, in the sense of 4.2.1.

### 4.3.2 Semantics of DScalaness

The semantics of DScalaness is an extension of the semantics of AFJ and FGJ to incorporate DnesT modules and operations. Computations assume a fixed class table $CT$ allowing access to class definitions via class names, which always decorate an object's type. A *store* $ST$ is a function from memory locations $\texttt{p}$ to object representations. Objects are represented in memory by lists of object references $\bar{\texttt{l}}$, which refer to the locations of the objects stored in mutable field values. A reference $\texttt{l}$ is a pair $(\texttt{p}, \texttt{N})$ where $\texttt{p}$ is the memory location

of an object representation and $\mathtt{N}$ is the nominal type of the object, including its class name. Hence, given an object reference $(\mathsf{p}, \mathtt{C}\langle\bar{\mathtt{T}}\rangle)$, one can access and mutate its fields $\bar{\mathtt{l}} = ST(\mathsf{p})$, and access and use its methods via the definition $CT(\mathtt{C})$.

Following AFJ, the semantics of DScalaness is defined as a *labeled transition system*, where transitions are of the form $\mathsf{e} - \{s = ST, s' = ST'\} \rightarrow \mathsf{e}'$. Intuitively, this denotes that given an initial store $ST$ and expression $\mathsf{e}$, one step of evaluation results in a modified store $ST'$ and contractum $\mathsf{e}'$. As an abbreviation, $\mathsf{e} \rightarrow \mathsf{e}'$ is written when the store is not altered.

The primary novelty of DScalaness over FGJ/AFJ is the formal semantics of type and module construction. Type construction is provided to allow programmers to dynamically construct module type instances. The appropriate behavior is obtained by treating dynamically constructed types as extensions of a basic class of objects, and declarations of DnesT level types via a `typedef` construct as syntactic sugar for ordinary object construction. A `LiftableType` class is defined as the supertype of all types of objects which can be used to instantiate a module, and dynamically constructed types are defined as instances of a generic `MetaType` class.

**Definition 4.3.1** *Any DScalaness class table $CT$ comprises the following definitions:*

$$CT(\texttt{LiftableType}) = \texttt{class}\ \texttt{LiftableType}\langle\rangle\ \texttt{extends}\ \texttt{Object}\ \{\ldots\}$$
$$CT(\texttt{MetaType}) = \texttt{class}\ \texttt{MetaType}\langle\texttt{X <:}\quad \texttt{LiftableType}\rangle\ \texttt{extends}\ \texttt{Object}\ \{\ldots\}$$

*Then assumed as given the following syntactic sugar:*

$$\texttt{typedef}\ \texttt{x} <: \texttt{T} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 \triangleq \texttt{def}\ \texttt{x} : \texttt{MetaType}\langle\texttt{T}\rangle = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2$$

*Class type* `MetaType` *is generalized on a single type variable. For brevity of notation, define:*

$$\texttt{MetaType}\langle\bar{\texttt{T}}\rangle \quad \triangleq \quad \overline{\texttt{MetaType}\langle\texttt{T}\rangle}$$

A crucial fact of DScalaness type construction is that any dynamically constructed type cannot be treated as a type at the DScalaness level. This is a more restrictive mechanism than envisioned in the foundational model (Liu, Skalka, and Smith 2012; Liu, Skalka, and Smith 2009), however it allows DScalaness to be defined as a straightforward extension to Scala, especially in terms of type checking.

Module instantiation, shown in Figure 4.13, is the only point where specialization of DnesT modules is allowed. Since DScalaness and DnesT are two different language spaces, some sort of transformation must occur when values migrate from DScalaness to DnesT via module instantiation. This *lifting* transformation involves both data mapping and serialization since the process spaces also differ. The aim is to be flexible and allow the user to specify how values are lifted and how types are transformed. The only requirement is that lifting and type transformation are coherent, in the sense that the lifting of an object should be typeable at the object's type transformation. This is formalized in the following definition.

**Definition 4.3.2** *Assume a relation* $\overset{lift}{\hookrightarrow}$ *which transforms a DScalaness reference* $l$ *into DnesT declarations* $\overline{d}$ *and expression* $e$ *is provided. Also assume a DScalaness-to-DnesT transformation of types* $[\![\cdot]\!]$ *is provided. To preserve type safety, it is required that in all cases* $(p, \mathtt{N}) \overset{lift}{\hookrightarrow} \overline{d}, e$ *implies both of the following for some type environment* $G$*:*

$$\varnothing, \varnothing \vdash \overline{d} : G \qquad \textit{and} \qquad G, \varnothing \vdash e : [\![N]\!]$$

The full definition of serialization, along with an example, are given and discussed below in subsection 4.3.3. In brief, when a module $\mu$ is instantiated, serialization will bind the value parameters of $\mu$ to the lifted values of their instances in a series of declarations that are added to its own. This is specified in the ModInst rule in Figure 4.13. Another important detail of the ModInst rule is that only type information in type parameters is used, and migrates into the module via type transformation and ordinary substitution.

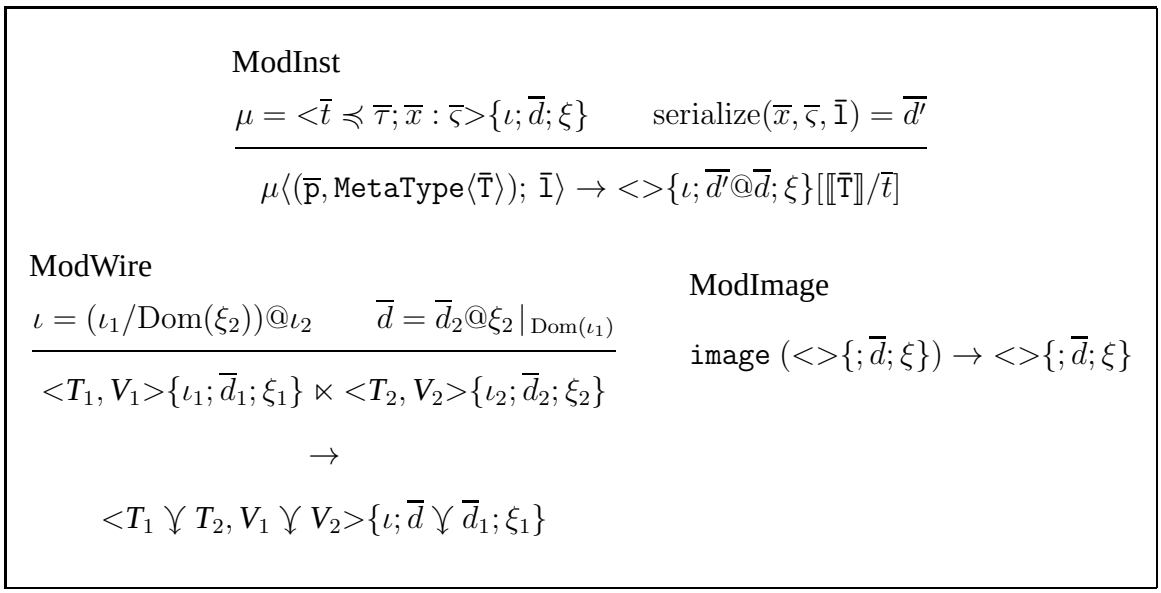


**Figure 4.13:** DScalaness Module Semantics

Module wiring is given a standard component composition semantics. Only the wiring of instantiated modules is allowed, which is consistent with nesC and simpler to implement. In a wiring $\mathsf{e}_1 \ltimes \mathsf{e}_2$, the imports of $\mathsf{e}_1$ are wired to the exports of $\mathsf{e}_2$. This is specified in the ModWire rule in Figure 4.13, which relies on the following auxiliary definition of operations for combining mappings.

**Definition 4.3.3 (Special Mapping Operations)** *Let $m$ range over vectors with mapping interpretations, in particular T, V, $\iota$, and $\xi$. Binary operator $m_1 \curlyvee m_2$ represents (non-exclusive) map merge, i.e., $m_1 \curlyvee m_2 = m_1@m_2$ with the requirement that $id \in \mathrm{Dom}(m_1) \cap \mathrm{Dom}(m_2)$ implies $m_1(id) = m_2(id)$. The mapping $m/S$ is the same as $m$ except undefined on domain elements in set $S$, and the mapping $m\,|_S$ is the same as $m$ except undefined on elements not in $S$.*

Finally, the ModImage rule in Figure 4.13 shows that imaging it is an assertion requiring its arguments to be a runnable module.

### 4.3.3 Serialization and Lifting

Serialization generates a flattened DnesT source code version of a DScalaness object in memory. At the top level, serialization binds the value parameters of a module to the results of flattening, aka lifting, via a sequence of declarations. Here is the precise definition.

**Definition 4.3.4 (Serialization)** *Assume given a store $ST$ which is implicit in the following definitions. The serialization of DScalaness references is defined as follows, along with an extension of the user defined lifting relation to sequences of references:*

$$\frac{\overline{\mathtt{l}} \overset{\mathit{lift}}{\hookrightarrow} \overline{d}, \overline{e}}{\mathrm{serialize}(\overline{x}, \overline{\tau}, \overline{\mathtt{l}}) = \overline{d}@\,\overline{\tau}\,\overline{x} = \overline{e}} \qquad \varnothing \overset{\mathit{lift}}{\hookrightarrow} \varnothing, \varnothing \qquad \frac{\mathtt{l} \overset{\mathit{lift}}{\hookrightarrow} \overline{d}, e \qquad \bar{\mathtt{l}} \overset{\mathit{lift}}{\hookrightarrow} \overline{d'}, \overline{e}}{\mathtt{l}\,\bar{\mathtt{l}} \overset{\mathit{lift}}{\hookrightarrow} \overline{d}@\overline{d'}, e\overline{e}}$$

Although lifting is user defined, a standard strategy is to introduce a new declared variable for each memory reference in the lifted object, and bind the variable to the lifted referent. Hence, lifting will typically be defined recursively. In this implementation, a "default" lifting has been adapted which follows this strategy, and also transforms objects by just transforming the fields into a representative structure, and ignoring methods. This is illustrated with an example in section 3.6.

The essence of this transformation can be formally captured with the following definitions. It is easy to see that these definitions will satisfy the requirements of Definition 4.3.2.

**Example 4.3.1** *In this example lifting of any object references is allowed, and transform the object $o$ into a structure containing the transformed fields of $o$. Methods are disregarded by the transformation. Here is the specification of the type transformation:*

*ChapinT*

$$\frac{CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\bar{\mathtt{X}} <: \bar{\mathtt{S}}\rangle\ \mathtt{extends}\ \mathtt{N}\ \{\bar{\mathtt{R}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\}}{[\![\mathtt{C}\langle\bar{\mathtt{T}}\rangle]\!] = \{\overline{\mathtt{f}} : [\![\bar{\mathtt{R}}[\bar{\mathtt{T}}/\bar{\mathtt{X}}]]\!]\}}$$

ModT

$$\frac{\mu : \mu\tau \text{ in nesT type checking}}{\Gamma \vdash \mu : \varnothing \circ \mu\tau}$$

ModInstT

$$\frac{\Gamma \vdash \mathtt{e} : \varnothing \circ <\overline{t} \preccurlyeq \overline{\tau}_1; \overline{x} \preccurlyeq \overline{\tau}_2 > \{\iota; \varepsilon\} \qquad \Gamma \vdash \overline{\mathtt{s}} : \mathtt{MetaType}\langle \overline{\mathtt{T}}_1 \rangle \qquad \Gamma \vdash \overline{\mathtt{e}}_2 : \overline{\mathtt{T}}_2 \qquad \vdash [\![\overline{\mathtt{T}}_1]\!] \preccurlyeq \overline{\tau}_1 \qquad \vdash [\![\overline{\mathtt{T}}_2]\!] \preccurlyeq \overline{\tau}_2}{\Gamma \vdash \mathtt{e}\langle \overline{\mathtt{s}}; \overline{\mathtt{e}}_2 \rangle : \overline{s} \preccurlyeq [\![\overline{\mathtt{T}}_1]\!] \circ <> \{\iota[\overline{s}/\overline{t}]; \varepsilon[\overline{s}/\overline{t}]\}}$$

ModWireT

$$\frac{\Gamma \vdash \mathtt{e}_1 : T_1 \circ <> \{\iota_1; \varepsilon_1\} \qquad \Gamma \vdash \mathtt{e}_2 : T_2 \circ <> \{\iota_2; \varepsilon_2\} \qquad \iota = (\iota_1 / \mathrm{Dom}(\varepsilon_2)) @ \iota_2}{\Gamma \vdash \mathtt{e}_1 \ltimes \mathtt{e}_2 : T_1 \curlyvee T_2 \circ <> \{\iota; \varepsilon_1\}}$$

ModImageT

$$\frac{\Gamma \vdash \mathtt{e} : T \circ <> \{\iota; \varepsilon\} \qquad \mathrm{main}() : \tau \in \varepsilon}{\Gamma \vdash \mathtt{image}\ \mathtt{e} : T \circ <> \{\iota; \varepsilon\}}$$

**Figure 4.14:** DScalaness Module Typing Rules

*and here is the specification of lifting.*

*Chapin*

$$\frac{ST(\mathtt{p}) = \overline{\mathtt{l}} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \overline{\mathtt{l}} \overset{\mathit{lift}}{\hookrightarrow} \overline{d}, \overline{e} \qquad x \ \textit{fresh}}{(p, \mathtt{C}\langle \overline{\mathtt{R}} \rangle) \overset{\mathit{lift}}{\hookrightarrow} \overline{d} @ ([\![\mathtt{C}\langle \overline{\mathtt{R}} \rangle]\!]\ x\ =\ \{\overline{\mathtt{f}} = \overline{e}\}), x}$$

### 4.3.4 DScalaness Type Checking

The DScalaness type checking rules adapt the typing rules of FGJ in their entirety. Refer to (Igarashi, Pierce, and Wadler 2001) for relevant details. Since type construction via `typedef` is syntactic sugar for normal object construction, that is covered by those rules as well. It remains to define typing rules for DnesT modules and module operations.

The DnesT module type form at the DScalaness level is $T \circ \mu\pi$, where $\mu\pi$ is a DnesT module type. The $T$ in this form represents the type bounds of dynamically constructed types that have been used to instantiate the module; this part of the type is referred to as the *instance coercion*. Because these types are dynamically constructed, their identity is not known statically, hence the need to treat them as upper-bounded type names in the static type analysis. It is important to note that the type names in $T$ will be fully resolved at run time, so that any module generated by a DScalaness program execution will have a fully reified DnesT type.

This is reflected in the ModT rule in Figure 4.14, which connects the DnesT typing system with the DScalaness type system. Since in this case an uninstantiated module definition is being typed, its instance coercion is empty. An instance coercion in a module type is directly populated when a module is instantiated, as in the ModInstT rule. Here, the type instances $\bar{s}$ are all dynamically constructed, so they define the upper bounds of the instantiated module's instance coercion. All type and value parameters are expected to respect the typing bounds specified in the module definition.

A subtle but significant detail in this rule is the consequence of dynamically constructed types having no meaning "as types" at the DScalaness level. This means that no DScalaness value of that type can be constructed, so dynamically constructed type names do not occur in the typings of value parameters.

However, dynamically constructed type names do need to be substituted for module type parameters in the import and export signatures. This ensures that wirings will be consistent regardless of the actual types eventually computed by the DScalaness program. A consequence of this is that modules can't be instantiated with anonymous expressions; only named type definitions can be used. Those names become part of the module's type.

The ModWireT typing rule for module wiring is a straightforward reflection of the operational rule for module wiring, as is the ModImageT rule for module runnability imaging.

### 4.3.5 Foundational Insights and Type Safety

Type checking of modules and operations is inspired by the type theory and metatheory developed for the language $\langle\text{ML}\rangle$ (Liu, Skalka, and Smith 2012). DScalaness module instantiation in particular can be decomposed into a set of $\langle\text{ML}\rangle$ operations, and typeablity of module instantiation follows from the typeablity of their composition. The language $\langle\text{ML}\rangle$ is obtained by extending system $F_{\leq}$ with state, dynamic type construction, and staging features. The expression $\langle e\rangle$ is a code value, and the $\text{lift}$ operation takes a value at one stage and "lifts" it to the next, by turning it into code and performing any necessary serialization.

Given this, a DScalaness module with a value and type parameter can be modeled in $\langle\text{ML}\rangle$ as a term:

$$\lambda x : \varsigma_1.\Lambda t \preccurlyeq \varsigma_2.\langle e\rangle$$

where $x$ and $t$ are value and type parameters for the block of code $\langle e\rangle$. Then, module instantiation can be modeled as the application of this term to a type and value parameter, where the latter must be lifted into the next stage:

$$(\lambda x : \varsigma_1.\Lambda t \preccurlyeq \varsigma_2.\langle e\rangle)\ (\text{lift}\ e)\ \tau$$

This interpretation of modules and module operations for the purposes of typing is evidenced by the DScalaness type form $T \circ \mu\pi$, where $T$ defines the type bounds for dynamically constructed types used to instantiate a module. This is directly analogous to $\exists$ type bindings in $\langle\text{ML}\rangle$ types, which statically define the upper bounds of dynamically constructed types.

Observing that AFJ, FGJ, and $\langle\text{ML}\rangle$ are all proven type safe, and that DScalaness is in essence an orthogonal composition of these three languages, the following conjecture states that type safety is maintained in this composition.

**Conjecture 4.3.1 (DScalaness Type Safety)** *If* $\varnothing \vdash e : T$ *and* $e \rightarrow^* \texttt{image}\ \mu$*, then* $\mu$ *is runnable and does not go wrong.*

# Chapter 5

# Scalaness/nesT

This chapter covers the specific details of Scalaness and nesT, the practical realization of DScalaness and DnesT. First described is nesT along with the details of how nesT programs are transformed into nesC programs. Additionally, included is a description of how the Scala compiler was modified to provide Scalaness type checking with a minimum of disruption to the the compiler's existing functionality. The full source code of a simple example Scalaness/nesT program is discussed in Appendix A. The use of Scalaness on a larger example is discussed in section 6.3.

## 5.1 NesT

NesT is the name give to the second stage language used by the implementation. Roughly speaking, nesT is the practical realization of DnesT. In particular, nesT uses the syntax of nesC to the greatest extent possible in order to simplify the compiler and to minimize the learning burden placed on existing nesC programmers. For example, a nesT module is specified exactly as a nesC module except that it can only `use` (import) and `provide` (export) nesC commands. In particular, neither nesC events nor nesC interfaces can appear

in a nesT component specification.

NesT also supports DnesT's notion of subtyping and features, for safe memory access and safe casting. Thus, while nesT programs are syntactically identical (aside from the new array increment operator), and semantically similar to programs written in nesC, nesT programs are more robust than equivalent nesC programs.

NesT type checking implements the rules described in subsection 4.2.3 and is largely straight forward. The intention is to follow nesC's type system to the greatest extent possible, with changes to account for stricter rules disallowing implicit conversions. However, nesT does support the subtyping rules of DnesT. The details of how nesT type checking was implemented is described by Watson (Watson 2013).

NesT is implemented as a rewriting to nesC. Because of nesT's special relationship with nesC, this rewriting is largely trivial. However, the implementation of the special features of nesT are described in more detail in this section.

The description that follows assumes the reader is familiar with nesC.

### 5.1.1 Component Specifications

In nesT components (after specialization) present interfaces that are sequences of imports and exports. The imports are implemented as nesC commands that are "used" by the component and the exports are implemented as nesC commands that are "provided" by the component. NesC-style events are not part of nesT but can be simulated using commands; an event used in nesC becomes a command provided in nesT and vice versa.

NesT does not provide separate interfaces as nesC does. Instead all interaction with other components is done by way of separately declared commands. These bare commands can be wired together in the usual way by the nesC compiler (Gay, Levis, von Behren, Welsh, Brewer, and Culler 2003). Figure 5.1 shows a simple example of a nesT module

that interacts with a timer. Instead of using an interface with an event `fired`, the module provides a callback command of the same name.

```
module ExampleC {
    uses command void setPeriodic( uint32_t period );
    provides command void fired( );
}
implementation {
    // Written in the nesT subset of nesC.

    void f( T param )
    {
        uint16_t value = x;
    }
}
```

**Figure 5.1:** Example nesT Module

The example in Figure 5.1 also shows the use of an undeclared type `T` and value `x`. Such types and values are instead declared as parameters of the module in the Scalaness program. The Scalaness compiler first adds these parameters to the appropriate environments before it type checks the nesT module (Watson 2013).

A nesT code base consists of a collection of unspecialized nesT modules. These modules do not by themselves constitute a complete program. It is the job of the first stage Scalaness program to specialize and compose the nesT modules, along with supporting components written in full nesC, into full applications.

NesT currently does not support nesC-style configuration components. Such support could reasonably be added since the implementation of a component is not important to the first stage code that manipulates it. The runnable module built by the Scalaness program as it specializes and composes the constituent nesT modules is transformed by the Scalaness runtime system into a nesC configuration reflecting the result of composition. However, this transformation is transparent to the Scalaness programmer. Libraries written in full

nesC must be wrapped in components with nesT interfaces as described below in order to become part of a nesT application.

Each nesT module has a nesT module type implied by its specification element list. This module type is extracted from the specification element list when the nesT component is type checked. In most cases, except as described below, it is compared against a module type annotation used in the Scalaness program for the module (see subsection 5.2.5). Any discrepancy is flagged by the Scalaness compiler as a type error.

### 5.1.2 External Libraries

Experiments with nesT show that it is expressive enough to write useful program components. However, any realistic application will need to interact with various libraries written in full nesC, named *external libraries*. It is not intended here to require the whole program be written in nesT, for such a requirement would not be practical. Instead external libraries could represent low level code such as the TinyOS operating system or high level application code that wishes to use Scalaness generated nesT modules.

At the time of this writing, neither nesT nor Scalaness provide any direct support for interfacing to external libraries, although such support might be useful future work. However, a programming technique can be used whereby shim components are manually created that wrap library interfaces. The following is an illustration of that technique using a small example.

Consider first the TinyOS `Boot` interface. This interface is used to indicate when a node is started; all useful nesC programs must interact with it. Yet nesT does not support interfaces at all, much less some of the entities, such as events, that are commonly declared in interfaces. Instead the programmer creates a shim component such as `BootShimC` as shown below

```
module BootShimC {
  uses command void booted( );
  uses interface Boot;
}
implementation {
  event void Boot.booted( )
  {
    call booted( );
  }
}
```

The shim component is legal nesC but not legal nesT. Its purpose is to expose all the commands and events in an external library interface as bare commands. To this end wrapper command and event implementations must be manually created.

Although creating shim components is a burden their form is highly stylized. A future version of the Scalaness compiler might generate them automatically. However, some shim components are complex and must do additional transformations on command arguments to interface with the non-nesT external library commands/events. In any case, shim components can be reused across nesT applications. Thus it is reasonable to expect programmers to accumulate a library of shims.

The shim components must be wired to the external library components they wrap. This is done by producing two nesC *wrapping configurations*. The first, conventionally called `LibraryIC` encapsulates all nesC components that have imports. The second, conventionally called `LibraryEC`, encapsulates all nesC components that have exports. Normally these are the only two configurations an application needs. If the programmer has full control over the entire application he or she can add the necessary external library components (via their shims) to either `LibraryIC` and/or `LibraryEC` as appropriate.

For example, Figure 5.2 shows an example `LibraryIC` component and an example `LibraryEC` component for a hypothetical application that uses the external `MainC` component and a specific instance of the generic timer module, both from the TinyOS library.

Notice that the SpecficTimerC component appears in both wrappers since it both provides and uses at least one command.

```
configuration LibraryIC {
  uses command void booted( );
  uses command void fired( );
}
implementation {
  components MainC, BootShimC, SpecificTimerC;

  BootShimC.booted = booted;
  BootShimC.Boot   -> MainC;

  SpecificTimerC.fired = fired;
}


configuration LibraryEC {
  provides command void startPeriodic( uint32_t period );
}
implementation {
  components SpecificTimerC;

  startPeriodic = SpecificTimerC.startPeriodic;
}
```

**Figure 5.2:** Example LibraryIC/EC configurations

NesC generic components must be instantiated in nesC configurations and each instance wrapped in its own shim. In Figure 5.2 the SpecificTimerC component is a shim that wraps a specific instance of the TinyOS generic timer. This limitation may seem restrictive, but nesT has its own support for genericity although the two mechanisms are independent.

NesT does not support nesC configurations but Scalaness does allow the components such as shown in Figure 5.2 to be declared and manipulated in Scalaness code. Such components are represented as Scala objects that extend the NesTComponent trait and that specify the source file of the nesC configuration using an external method as shown

in Figure 5.3.

```
@ModuleType(
  """{}<;>
     { booted(): Void,
       fired() : Void; }""")
object LibraryIC extends NesTComponent {
  external("LibraryIC.nc")
}


@ModuleType(
  """{}<;>
     { ; startPeriodic(period: UInt32): Void }""")
object LibraryEC extends NesTComponent {
  external("LibraryEC.nc")
}
```

**Figure 5.3:** Representation of External Components

The module type of external components cannot be determined by examining their definitions since they are not in nesT. However, as with all nesT modules they must be annotated with their module type in the Scalaness program as shown in Figure 5.3 and discussed further in subsection 5.2.5. For external modules this annotation is accepted without question by the compiler.

The `LibraryIC` and `LibraryEC` objects are then manipulated in the usual way by the Scalaness program. Because `LibraryIC` has only imports and `LibraryEC` has only exports it is normal for these components to appear at the ends of a wiring chain. Figure 5.4 shows an example where the result module is runnable. In Figure 5.4 the +> symbol is the Scalaness wiring operator.

Although it is not possible to use Scalaness to compose external library components directly, the programmer is free to create several different wrapping configurations, if desired, and represent each of them separately in the Scalaness program. The Scalaness

```
@ModuleType("""{ checksumType <: UInt32 }
               <;>
               { ; }""")
val resultModule =
  LibraryIC +>
    formattingModule +> checkingModule +>
      LibraryEC
```

**Figure 5.4:** Wiring nesT Components

program could then dynamically select which wrapping configuration is to be used in the final generated code. In any case the type system will ensure that illegal wirings can never be made.

### 5.1.3 Structure Subtyping

DnesT supports width subtyping of structures as shown in Figure 4.9. To implement this, nesT supports covariant subtyping of pointers to structure types. If $\tau_1$ and $\tau_2$ are structure types and $\tau_1 \preccurlyeq \tau_2$ according to DnesT subtype rules, then $\tau_1* \preccurlyeq \tau_2*$. To implement the important case of passing a pointer to a structure into a function, the Scalaness compiler need only add an appropriate cast as it rewrites the nesT to nesC. For example, consider the nesT code below

```
struct X {
  int a;
};

struct Y {
  int a;
  int b;
};

void f( struct X * );
struct Y object;
f( &object );
```

The call to `f` is rewritten to `f( (struct X*)&object )`. The structure layout rules of nesC guarantee this is safe and `f` will only manipulate the `X` subobject of its parameter.

Like DnesT, nesT has no notation to indicate a subtype relation between structures. Instead, the judgment is entirely based on structural considerations.

### 5.1.4 Safe Casts

Since one of the goals of nesT is to promote type safety, no implicit type conversions, aside from subtype conversions, are provided. Explicit conversions are permitted only when configured by the programmer. This allows the programmer to define certain casts that are logical even if they require non-trivial user defined code to execute.

The Scalaness compiler accepts a configuration file that defines a relation on types *isCompatible*. If *isCompatible(*$T_1$, $T_2$*)* is true then it is permitted to cast an expression of type $T_1$ into an expression of type $T_2$. There are no restrictions on the types $T_1$ and $T_2$. However, all such conversions require explicit cast expressions; they are never applied implicitly. The *isCompatible* relation is the implementation of *compatible* in the CastT rule of Figure 4.10.

To illustrate the way these casts are implemented in nesT programs, consider as an example the following two structure definitions.

```
struct UserInfo {
  char name[25];
  int  age;
  int  id;
};

struct UserToken {
  int id;
  int hash;
};
```

The programmer may wish to allow an object of type `UserInfo` to be explicitly cast into an object of type `UserToken`. Assuming the Scalaness configuration file has been edited to allow this, the Scalaness compiler rewrites each cast expression into a call of a conventionally named conversion command. These conversion commands exist in a nesC interface `DoCast`. For example

```
token = (struct UserToken)user;
// ... rewritten to ...
token = (call DoCast.UserInfo_UserToken(user));
```

The programmer is required to provide the `DoCast` interface and a component called `DoCastC` that provides that interface and contains an implementation of the various conversion commands needed. The Scalaness compiler wires to `DoCastC` automatically without any further programmer intervention.

### 5.1.5 Array Operations

Each nesT expression `a` of array type `Array(T)` for some element type `T` has a corresponding hidden dynamic value representing the size of the array. Array increment expressions of the form `a |> e` can nominally be rewritten to nesC using pointer arithmetic as `( (a) + (e) )`. Let $n$ be the dynamic size of expression `a`, then $n_e$, the dynamic size of `a |> e`, is $n_e = n - e$. This size might be negative but any use of an array expression with a negative size results in a run time error *at the point of use*.

A statement containing one or more array increment expressions or array indexing expressions is rewritten as a block enclosed sequence of statements containing Scalaness compiler generated local variables for the dynamic sizes of the temporary arrays along with appropriate run time checks.

For each array increment operation $a_i \rhd e_i$ in a statement a variable to hold the value of $e_i$ is declared and initialized. This is done so that $e_i$ will only be evaluated once; an

important consideration in a language, such as nesT, with side effects. Also the dynamic size of the result of each array operation $d_i$ is declared and initialized appropriately. For example

```
... (a |> e) ...
```

Is rewritten without regard to any possible optimizations as:

```
{
  int __e_1 = e;
  int __d_1 = __d_0 - __e_1;

  ... ((a) + __e_1) ...
}
```

Here `__d_0` is the dynamic size associated with the array expression `a`. In the common case where `a` is a declared array the size will be known statically and an appropriate constant can be used instead of a reference to a dynamic size variable.

For each array indexing operation $a_i[e_i]$ in a statement a variable to hold the value of $e_i$ is declared and initialized, as before. A run time check is inserted to ensure that the value of $e_i$ is inside the dynamic size of $a_i$. For example

```
... a[n] ...
```

Is rewritten without regard to any possible optimizations as

```
{
  int __e_1 = n;
  if (__e_1 >= __d_0) call boundsCheckFailed( );
  ... a[__e_1] ...
}
```

As before `__d_0` is the dynamic size associated with the array expression `a`.

In a statement involving multiple array operations, each operation is rewritten as described above one at a time. After the first operation is rewritten, the enclosed modified

statement is further expanded with the second rewriting. The checks are issued in the order they are encountered during a depth first left to right traversal of the nesT abstract syntax tree. For example a statement such as

```
x = ((a |> e1) |> e2)[b[i]];
```

Is first rewritten as

```
{
  int __e_1 = e1;
  int __d_1 = __d_0 - __e_1;
  x = (((a) + __e_1) |> e2)[b[i]];
}
```

The resulting statement still contains three array operations. The second stage of rewriting yields

```
{
  int __e_1 = e1;
  int __d_1 = __d_0 - __e_1;
  {
    int __e_2 = e2;
    int __d_2 = __d_1 - __e_2;
    x = ( ((a) + __e_1) + __e_2 )[b[i]];
  }
}
```

The inner indexing operation is then rewritten

```
{
  int __e_1 = e1;
  int __d_1 = __d_0 - __e_1;
  {
    int __e_2 = e2;
    int __d_2 = __d_1 - __e_2;
    {
      int __e_3 = i;
      if (__e_3 >= __d_b) call boundsCheckFailed( );
      x = ( ((a) + __e_1) + __e_2 )[b[__e_3]];
    }
```

```
    }
  }
```

Finally, the outer indexing operation is rewritten in a similar manner, including an additional call to `boundsCheckFailed`.

Functions declared to take an array as a parameter are rewritten so that the dynamic size of the array is passed as an additional parameter. This parameter becomes the $d$ of array expressions involving only the parameter. Calls to such functions are rewritten to pass the additional dynamic size information as appropriate.

The command `boundsCheckFailed` must be provided by the programmer in a component named `BoundsCheckC`. The behavior of this command is unspecified but it should not return. The expectation is that in most cases it will restart the node after, perhaps, attempting to log the problem. As with `DoCastC` the Scalaness compiler automatically wires to `BoundsCheckC` as appropriate.

The user defined handling of bounds check failure and of explicit casts as described previously is where the runtime failure semantics of Definition 4.2.3 are implemented.

## 5.2 Scalaness

Scalaness is implemented as a modified Scala compiler (Chapin 2013a) based on the open source development Scala compiler. The Scala compiler has a plug-in architecture and it had originally been anticipated that Scalaness could be implemented as a compiler plug-in. That would have made Scalaness easier to use and maintain and, thus, enhanced the systems practicability.

Unfortunately, implementing Scalaness as a plug-in met with difficulties. The main problem was with extending the type checker of Scala to accommodate the Scalaness type system. The plug-in approach required a complete reimplementation of Scala typing inside

the plug-in. This is because plug-ins can only gain control either before Scala typing has occurred or after it has completed. Consequently, the implementation of the Scalaness typing rules couldn't easily benefit from the logic in the existing type checker.

In contrast, building Scalaness as a modified compiler allowed Scalaness type information to "piggyback" on the existing type checker infrastructure. In particular, Scalaness type information was added to the singleton types already created and maintained by the Scala compiler for each declared value. This information could then be queried at critical points during the type checking process where Scalaness rules, as shown in Figure 4.14, were applied (Watson 2013).

Nevertheless, in order to facilitate keeping Scalaness synchronized with future developments of the main Scala compiler, every attempt was made to implement Scalaness in the least invasive way possible. Much of the logic, including the new typing rules themselves, are implemented in separate packages away from the main body of the compiler code base. The instances where it was necessary to insert Scalaness specific code into, for example, the existing type checker, have been kept to a minimum.

Making radical changes to Scala syntax was not seriously considered. For reasons of simplicity, it was deemed undesirable to modify *both* the parser and the type checker. Fortunately Scala has a general mechanism for adding arbitrary information to declarations, namely *annotations*. Scala annotations were used to express nesT module types as strings using an arbitrarily chosen syntax designed to be palatable to Scala programmers.

This work's foundation utilizes Scala 2.10, which also provides an extensive reflection API and experimental support for expression macros. These facilities allow one to do abstract syntax tree (AST) transformations on Scala programs using ordinary Scala code. Macros are described by the Scala community as a kind of "lightweight" plug-in mechanism. Unfortunately, at the time of this writing, type macros are not available so it is not yet possible to write a macro that outputs a class definition. However, in the future when

type macros become available it might be possible to implement some, or all, of Scalaness as a macro library.

### 5.2.1 Scala Compiler Organization

The Scala compiler is organized as a number of *phases* that rewrite the input in successive steps lowering it to JVM bytecode. The precise phases used can be listed with the command `scalac -Xshow-phases`. Of primary significance to Scalaness are the first four phases used by the stock Scala compiler as shown below.

```
parser
namer
packageobjects
typer
...
```

The bulk of the modifications made by Scalaness are in the typer phase. Hooks were added at critical points in the Scala type checker that call into Scalaness-specific code in package `edu.uvm.scalaness`. Furthermore, a new phase was added between the parser and namer phases. This new phase is responsible for augmenting certain Scalaness constructs with their necessary runtime support. This is done by inserting material in the AST produced by the parser. In principle, that material could have been manually written by the programmer but instead is automatically generated as a convenience. It is this "post-parser" phase that could potentially be eliminated by type macros when they become available.

### 5.2.2 Liftable Types

Certain types and their corresponding values that appear in a Scalaness program are liftable to types and values in the nesT modules manipulated by that program as described in

subsection 4.3.3. Values of these types need to be transformed as they cross the boundary between the two programming languages due to differences in the way a liftable type and its nesT counterpart are represented. This section describes which Scalaness types are liftable and how their values are handled when used to specialize a nesT module.

**Primitive Types**

All liftable types except arrays are subtypes of a special marker trait `Liftable`. The primitive types in nesT have liftable counterparts in Scalaness that are classes extending `Liftable`. For example, the type `uint16_t` in nesT corresponds with class `UInt16` in Scalaness. In this implementation there are six primitive, liftable integer types: three unsigned types `UInt8`, `UInt16`, `UInt32`, and three corresponding signed integer types. All of these types have specific sizes; the implementation does not provide a simple integer type. This avoids issues associated with the machine dependent size of `int` in nesC. Finally, two other liftable primitive types are also provided: `Char` and `Uninit` (which lifts to `void`).

The nesT subtype relations for primtive integers are preserved in Scalaness. In Scalaness the primtive types types are defined in the object `LiftableTypes` so that they don't conflict with any normal (non-liftable) types defined by the programmer or the language, such as `Char`. Furthermore, the integer liftable types are endowed with the usual arithmetic operations so they can be manipulated in the Scalaness program in a natural way.

No conversions are provided between the liftable types and their ordinary Scala analogs. This means existing libraries that, for example, manipulate Scala `Char` objects won't work with `LiftableTypes.Char`. This is not regarded as a problem for two reasons.

1. Since values of liftable type will eventually be written into nesT components, they will likely be put to very different uses than values of ordinary Scala types. In fact,

letting the Scala type system catch inadvertent mixing of ordinary primitives and liftable primitives could be seen as a desirable feature.

2. Implicit conversions can be easily added by the Scalaness programmer, if desired, using the normal facilities of Scala.

To facilitate the second point, *explicit* conversion methods from each liftable type to its corresponding non-liftable counterpart are provided as a convenience.

The Scala type system is used to ensure compile-time type safety of the primitive liftable types in a Scalaness program. For example, the type `Int16` can only be constructed using a value of Scala's type `Short`. Consequently, normal Scala type checking prevents a potentially out of range value from being used.

Unfortunately, Scala does not support unsigned types natively. In the current implementation a sufficiently wide signed type is used to initialize objects of unsigned liftable type. This makes it possible to use an out-of-range value during the execution of the Scalaness program resulting in a runtime exception. However, Scala programs are subject to runtime exceptions for a variety of reasons. It is well outside the scope of this work to address the problem of how to ensure a Scalaness program never exits by way of an exception.

**Arrays**

*An ordinary Scala array type is liftable if, and only if, its element type is liftable.* This is an exception to the rule stated in section 5.2.2 that says all liftable types must extend `Liftable` and, thus, arrays are handled in a special way. Yet it is a significant convenience to the programmer to be able to use ordinary Scala arrays, and not some special "liftable array" class, to hold liftable arrays. This need does not arise for the other containers in the Scala collections library since those containers have no counterpart in nesT anyway.

For example, the programmer may wish to create and manipulate a `List[UInt8]` during the execution of a Scalaness program, but the list itself won't be liftable. In contrast the programmer may wish to lift an `Array[UInt8]` into nesT.

**Classes**

A Scala class $C$ that extends the `Liftable` trait is liftable to a nesT structure type provided it additionally obeys the following inductive rules.

1. $C$ is not generic.
2. All of $C$'s fields have liftable type.
3. All of $C$'s supertypes (except `AnyRef` and `ScalaObject`) are liftable types.

In this case $C$ is said to be a *liftable class*. Except for the rules mentioned here there are no restrictions on the definition or use of liftable classes. In particular, they are able to have methods, although the methods of a liftable class have no manifestation in the generated nesT code and would exist only as a convenience to the Scalaness programmer.

For example, consider the following Scalaness code:

```
class Header
 (val nodeID      : nodeIDType,
  val componentID : UInt8 ) extends Liftable

class TimeStampedHeader
 (val timeStamp   : UInt16) extends Header
```

Here `nodeIDType` is a previously defined liftable class type. Consequently, both of these classes are liftable and have representations as nesT structure types.

### 5.2.3 Lifting

When nesT modules are specialized by values, the Scala values used to make those specializations are lifted into the nesT code. For primitive types, occurrences of the value parameter in the nesT code is simply replaced by a constant representing the actual value used to specialize the module.

Values with array or structure (class) types are handled differently. In that case the Scalaness compiler writes a global declaration into the nesT module that defines the value along with an initializer constructed from the Scala value used to specialize the module. This follows the semantics described in subsection 4.3.2.

As an example, consider the following Scalaness class representing a nesT module that does encryption. The module is parameterized by a key value.

```
@ModuleType(
  """{}
     < ; key: Array[UInt8] >
     { ; encrypt(data: Array[UInt8]): Void }""")
class EncryptorC extends NesTComponent {
  "EncryptorC.nt"
}
```

When this module is instantiated as described in section 5.2.7 a Scala array of `UInt8` values is provided. The Scalaness compiler will output, at second stage generation time, nesT code such as

```
module EncryptorC {
  provides void encrypt(uint8_t data[]);
}
implementation {
  // Added by the Scalaness compiler.
  uint8_t key[] = { 1, 2, 3, 4 };
  // Uses of 'key' as before.
}
```

Here `{ 1, 2, 3, 4}` is a sample value of the key parameter used to instantiate the

module. The nesT programmer does not declare the global variable `key` in the nesT module but nevertheless uses the name `key` freely in the module. In effect, the nesT programmer is using the parameter declaration (in the Scalaness code) to guide his/her work. This follows the behavior of the nesT type checker.

The example above shows the result before the final translation to nesC. During that translation the Scalaness compiler will also augment the parameter list of `encrypt` to include an additional size parameter for the array as described in subsection 5.1.5.

### 5.2.4 MetaType

Scalaness allows types to be dynamically constructed. However, the Scala type system does not directly support using types as values. To work around this limitation, a wrapper generic class `MetaType[T]` was explicitly introduced to represent any liftable type that is a subtype of `T`.

```
class MetaType[+Tau <: Liftable]
  (val wrappedType: TypeRepresentation) extends Liftable
```

Values in the Scalaness program that are intended to hold nesT types that are a subtype of $\tau$ have a Scala type of `MetaType[Tau]`. A variance annotation is used to ensure that `MetaType` is covariant in its type parameter. This allows flexibility since, for example, a `MetaType[UInt16]` value should be usable where a `MetaType[UInt32]` is expected. This is sound since the subtype relation is transitive and, for example, any type that is a subtype of `UInt16` is also a subtype of `UInt32`.

Objects of type `MetaType` contain a *representation* of a nesC type. While `MetaType`'s type parameter is a Scala type that is liftable to nesT, the value it wraps is a representation of the already lifted type. Thus `MetaType` objects form a bridge between the Scalaness and nesT type systems.

### 5.2.5 Module Type Annotations

Values definitions, method parameters, and method results that are intended to manipulate nesT component values must be explicitly decorated with a module type annotation. In this way nesT type information can be made known to the Scalaness compiler without modifying the Scala parser to understand an extended type language directly. Module type annotations are string literals that obey the abbreviated syntax in Figure 5.5.

```
module-type ::=
    '{' existential-binders? '}'
    '<' type-parameters? ';' value-parameters? '>'
    '{' imports? ';' exports? '>'

existential-binder ::= IDENTIFIER '<:' type
type-parameter     ::= IDENTIFIER '<:' type
value-parameter    ::= IDENTIFIER  ':' type
```

**Figure 5.5:** Module Type Syntax

The imports and exports in Figure 5.5 are nesT declarations written in a Scala-like syntax using `Array` and `PointerTo` type constructors to define array and pointer types. Structure types are specified using $\{ \ldots \}$ syntax to enclose the declarations of structure members and are prefixed by the structure name.

The following shows a sample of a `SendC` component parameterized by an integer type suitable for use as a network address. The component imports a command `radio` that takes a parameter `message` of structure type. The component exports a command `send` that returns the TinyOS standard error type `error_t` which has built in support in module type annotations.

```
@ModuleType(
  """{}
     < addrT <: UInt32; >
     { radio(message:
         MessageType{src : addrT,
```

```
                    dest: addrT,
                    data: Array[UInt8,64]}): ErrorT;
        send(s: addrT,
             d: addrT,
             data: Array[UInt8]): ErrorT }""" )
class SendC extends NesTComponent { ...
```

Annotations, such as above, that are placed on class or object definitions are checked against the nesT code that implements that component—except for external library components as described in subsection 5.1.2. Method parameters, method results, and **`val`** and **`var`** definitions also need to be explicitly annotated; Scalaness does not support type inference of nesT types. This does place a considerable burden on the programmer. However, a type abbreviation scheme has been developed to alleviate this burden (Watson 2013).

In places where module types are required to be annotated, the annotated type is checked against the actual type derived by the Scalaness type rules in Figure 4.14. Type errors are reported as necessary.

### 5.2.6 Component Declarations

Components in nesT can be parameterized by types and values and instantiated multiple times. These properties are closely modeled by Scala classes. Thus, the representation of a nesT component in Scalaness is by way of a class that extends a special `NesTComponent` marker trait.

One might be tempted to allow a syntax such as

```
class SendC
  [Adt <: UInt32, MessageT <: AbstractMessage[Adt]]
  (self: Adt) extends NesTComponent {

  import error_t radio( MessageT * );

  export error_t send( Adt addr, uint8_t *data ) {
    MessageT packet = { self, addr, data };
```

```
    radio( &packet );
    return SUCCESS;
  }
}
```

This intends to define a Scalaness component using Scala syntax for representing type and value parameters with the body of the component written in nesT. Unfortunately this cannot be supported without modifying the Scala parser to accept nesT as well as Scala.

One way to work around the problem of mixed language syntax is presented by Garcia (Garcia, Izmaylova, and Schupp 2010) where the "alien" language is included as a string literal. The Scala type checker would treat the nesT program as having type String but the additional Scalaness type checking could parse the string literal's contents and impose additional typing rules on those contents. However, this approach leads to a rather ungainly programming style:

```
class SendC
  [Adt <: UInt32, MessageT <: AbstractMessage[Adt]]
  (self: Adt) extends NesTComponent {

 """import error_t radio( MessageT * );

    export error_t send( Adt addr, uint8_t *data ) {
      MessageT packet = { self, addr, data };
      radio( &packet );
      return SUCCESS;
    }"""
}
```

Since the nesT code implementing a component is often long and complex it makes sense to allow the programmer to edit and manage that code in tools that are nesC-aware such as nesC syntax highlighting editors. It is anticipated that in many cases different programmers with very different kinds of expertise will be editing the Scalaness and nesT code bases. Thus, this implementation uses a string literal to name an external file containing

the nesT contents of a component as shown below.

```
class SendC
  [Adt <: UInt32, MessageT <: AbstractMessage[Adt]]
  (self: Adt) extends NesTComponent {

  "SendC.nt"
}
```

Conceptually the contents of the named file replace the literal name in the Scalaness program much as #include directives work in C programs. This approach allows the normal Scala parser and type checker to process the program successfully. During compilation the Scalaness extension locates the specified nesT file (`SendC.nt` above), parses it as nesT and does nesT type checking on that file using type and value parameters as provided to the Scalaness class.

Using Scala's syntax for specifying type and value parameters as shown above is attractive but unfortunately it does not work for Scalaness. There are two problems:

1. It is the intent to allow nesT components to be passed around in a Scalaness program in an uninstantiated state. In contrast Scala classes are not first class values in Scala.

2. NesT components can have parameters involving dynamically constructed types. In contrast Scala class parameters must involve only types that are fully specified statically.

Modifying the Scala compiler to allow dynamically constructed types to appear in declarations and as type parameters was considered, but this required extensive modifications to the existing Scala type checker and so was rejected as an option. Instead the Scalaness representation of a nesT component includes a special method `instantiate`, automatically generated by the compiler, that is used to create instantiated nesT components. That method accepts the value parameters as ordinary Scala parameters using liftable types, and

it accepts the type parameters as ordinary Scala parameters of type `MetaType[T]` where `T` is liftable. The example above becomes:

```
class SendC extends NesTComponent {

  def instantiate(
    Adt     : MetaType[UInt32],
    MessageT: MetaType[AbstractMessage],
    self    : UInt32);

  "SendC.nt"
}
```

### 5.2.7 Runtime Support

In addition to compile-time analysis and type checking, Scalaness programs require support for nesT module composition (aka "wiring") at runtime. In this section, an overview of how this runtime support works is provided. Full details can be found in the documented source code of Scalaness (Chapin 2013a). Note that the current implementation of Scalaness composes nesT modules and rewrites those modules to nesC in a single, integrated processing step. The runtime system currently emits nesC directly without the need for any explicit nesT-to-nesC rewriting.

There are two operations to consider: component composition and component instantiation.

#### Composition

Each Scalaness class that represents a nesT module is augmented by the Scalaness compiler to contain a hidden field that represents a nesC configuration wrapping that single component. Figure 5.6 shows an example of a Scalaness class representing a nesT module that provides a command for computing checksums on a given array of bytes. This module

is parameterized by the type used for the checksum and by the size of the arrays that it processes.

The code marked in Figure 5.6 as being generated by the compiler is not legal Scala as shown but is presented as an aid to understanding. The compiler actually inserts, during compilation, a type-correct AST of the necessary code into the AST of the enclosing class. The Scalaness runtime library, specifically the methods in the inherited `NesTComponent` trait, makes use of this generated code during module composition.

In particular, the generated field `configuration` holds the reference to an object representing a nesC configuration that wraps the nesT module. The imports and exports of that module are extracted from the abstract syntax tree of the nesT code which is parsed by the runtime system and is represented by `abstractSyntax` in Figure 5.6. Note that the nesT code has already been syntax checked and type checked by the Scalaness compiler during the compilation of the Scalaness program. The reparsing done at runtime is thus guaranteed to succeed.

The named component wrapped by the configuration is made aware of the names of the type and value parameters. Furthermore, the Scalaness compiler augments the class with methods `getTypeMap` and `getValueMap` that return maps associating those names with additional hidden fields (the definitions of which are not shown in Figure 5.6) holding the runtime representation of the type and value arguments actually used. This information is used during module instantiation as described in section 5.2.7.

The method `+>` inherited from `NesTComponent` combines the configurations in its operands to return a new program component representing the overall nesC configuration. It is in the `+>` method where the operational semantics of wiring is implemented (see Figure 4.13). The program component returned from `+>` is flattened in the sense that it is a single configuration that wires all the named program components (nesT modules) it contains; a hierarchy of configurations is not created.

```
@ModuleType(
  """{}
     < checksumType <: UInt32; size: UInt16 >
     { ; compute_checksum(
           data: Array[UInt8]): checksumType }""")
class ChecksumC extends NesTComponent {

  /////////
  // Code generated by the Scalaness compiler.
  /////////
  val configuration =
    new ProgramComponentWrapper(
     new NamedProgramComponent(
      name            = "ChecksumC",
      enclosingObject = this,
      typeParameters  = Set("checksumType"),
      valueParameters = Set("size"),
      imports         = extractImports(abstractSyntax),
      exports         = extractExports(abstractSyntax),
      abstractSyntax  = abstractSyntax))

  def getTypeMap =
    Map("checksumType" -> sclnsChecksumType)

  def getValueMap =
    Map("size" -> sclnsSize)
  ////////
  // END of Scalaness generated code.
  ////////

  "ChecksumC.nt"
}
```

**Figure 5.6:** Generated Runtime Support for Composition

An `image` method in the `ProgramComponent` class writes the nesC configuration implied by the current program component and then iterates over all the named components serializing their abstract syntax trees to nesC. During this serialization the nesT to nesC transformations, for example array bounds checks, etc., as described in section 5.1 are also done.

### Instantiation

A class representing a nesT component can be instantiated using Scala's operator **`new`** like any other class. However, such an instance still has an uninstantiated module type. Instantiation of a component at the nesT level is done in Scalaness by invoking a compiler generated `instantiate` method, an example of which is shown in Figure 5.7.

As for Figure 5.6 the code shown as generated by the compiler is for illustration only. The compiler inserts during compilation the AST of the appropriate code into the AST of the enclosing class.

The `instantiate` method is provided module type parameters as Scala `MetaType` values, and module value parameters, all of which must have liftable types. The method creates a fresh Scala instance of the class and stores the module parameters into hidden fields where they are subsequently used (during imaging) to specialize the module's body. A single instance of the Scala class could thus create many different specializations of the nesT module by way of separate invocations of `instantiate` with potentially different parameters.

```
@ModuleType(
  """{}
     < checksumType <: UInt32; size: UInt16 >
     { ; compute_checksum(
           data: Array[UInt8]): checksumType }""")
class ChecksumC extends NesTComponent {

  /////////
  // Code generated by the Scalaness compiler.
  /////////
  private var sclnsChecksumType: MetaType[UInt32] = null
  private var sclnsSize: UInt16 = null

  def instantiate(
    checksumType: MetaType[UInt32], size: UInt16) = {

    val result = new ChecksumC
    result.sclnsChecksumType = checksumType
    result.sclnsSize = size
    result
  }
  ////////
  // END of Scalaness generated code.
  ////////

  "ChecksumC.nt"
}
```

**Figure 5.7:** Generated Instantiate Method

# Chapter 6

# Evaluation

This chapter presents the results of evaluating the performance of Sprocket and Scalaness/nesT. The results are presented in terms of both simple "toy" programs that explore specific issues in isolation, and on a larger, realistic application employing trust management authorization that demonstrates the applicability of the systems in real world scenarios.

## 6.1 Field Example

To evaluate the performance of SpartanRPC and Scalaness in a real application setting, both systems are used to implement secure versions of data collection and sampling control protocols in an environmental monitoring system. The Snowcloud system (Frolik and Skalka 2013; Moeser, Walker, Skalka, and Frolik 2011) is a wireless sensor network developed at the University of Vermont for snow hydrology research applications. It is based on the MEMSIC TelosB mote platform running TinyOS, and has seen multiple field deployments. Typical deployed systems comprise 4-8 sensor nodes but the technology is scalable to arbitrary numbers of nodes. For data collection and sampling rate control, the system also includes a handheld "Harvester" device. This device incorporates a TelosB mote to

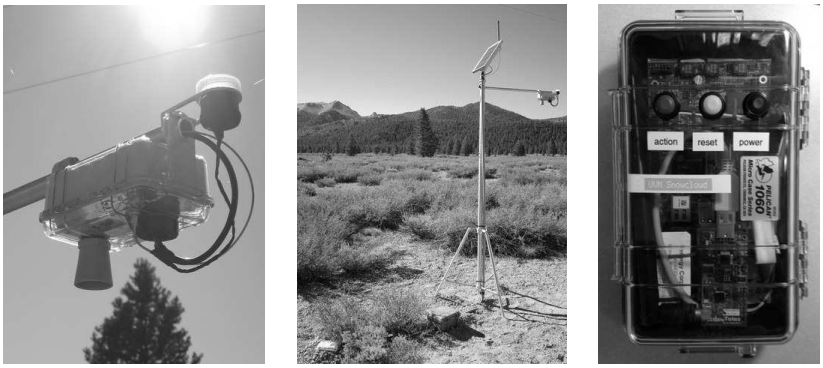


**Figure 6.1:** A Snowcloud Sensor Node (L,C) and Harvester Device (R).

establish a network connection when in radio communication with the deployment. Users transport the device to and from deployment sites, and interact with the sensor node network by issuing commands from a simple push-button interface. A Harvester device and a deployed Snowcloud sensor tower are pictured in Figure 6.1. The scheme described here has been implemented and tested in the UVM test network, which uses the same software and hardware platforms as in the active deployments.

In the secured version of the Snowcloud system, the goal is to treat data collection and sampling rate control as protected resources requiring authorization. Furthermore, sampling rate modifications should require a higher, "administrator" level of authorization than data collection. That is, only system engineers should be able to perform control operations, whereas data end-users making field visits should be able to collect data. Snowcloud sensor node code in particular makes use of nearly every resource available on the mote—including timing, sensor I/O, radio messaging, and flash memory, not to mention CPU and main memory. Thus, it is a robust example of a realistically scaled application.

The system described here is also informative since it can be easily ported to other similar application settings. That is, sensor network application settings wherein multiple users of various authorization levels need to interact with the same network in control or

collection capacities, as mediated by security policy.

**A note on secure deployment.** In the Sprocket version of the field example, the nodes are deployed with private keys embedded in ROM. The system is thus as secure as the tamper-resistence of the nodes permits. However, one scenario visualized with Scalaness is for the specialized program to be deployed using an over-the-air reprogramming system such as Deluge. In that case session keys negotiated by the first stage program could be exposed to eavesdroppers. However, this problem can be mitigated using secure over-the-air deployment (Dutta, Hui, Chu, and Culler 2006). In any case, Scalaness does not *require* over-the-air deployment; programming nodes in the lab prior to being physically deployed would also be a common scenario.

## 6.2 Sprocket

This section discusses the performance of the programs generated by Sprocket in terms of both space and time. It begins by evaluating Sprocket using "toy" programs that focus on specific aspects of the system's performance. Next the performance of Sprocket on the field example is discussed. The combined use of public and private key cryptography in the underlying security protocol is shown to impose a low amortized cost over time, despite high costs for initial authorizations.

Since many communication chips now support hardware AES encryption, this evaluation demonstrates performance using that feature. In particular, the popular Tmote Sky wireless sensor mote (moteiv 2006) uses a Chipcon CC2420 transceiver with hardware encryption. Unfortunately, the standard TOSSIM simulation environment does not model hardware encryption for TinyOS 2.1 so all tests were performed on real hardware, limiting the tests to small scale configurations. Tmote Sky nodes were used, with 10 KiB

of RAM, 48 KiB of ROM and an 8 MHz MSP430 microcontroller running TinyOS 2.1.2 (Community ).

The system was exercised using several small test programs. These programs consisted of a client/server pair where the client repeatedly sent a message containing a 16 bit value to the server. The purpose of these tests was to explore the overhead induced by the system with minimal obscuring effects from application logic. The percentage overhead observed with the small programs is thus a worst case overhead.

A demonstration program was also created that implemented the directed diffusion algorithm (Intanagonwiwat, Govindan, Estrin, Heidemann, and Silva 2003) over several nodes. This program allowed testing of the behavior of the system in a long-running setting, and exercised the system in a multi-mote, multi-hop network environment. Although the demonstration program did not perform any significant function, it did show that useful higher level services can be built on top of SpartanRPC.

### 6.2.1 Memory Overhead

The Sprocket run time system uses several memory caches to hold key material, credential information, and the minimum model implied by the set of known credentials. These caches are statically allocated but must be stored in RAM since their contents are dynamic. Table 6.1 summarizes the RAM consumption of the various storage areas used by the current implementation.

The number of items in each cache are tunable parameters. The optimum settings depend on the intended application. The values in Table 6.1 attempt to strike a balance between usability and flexibility on one hand and excessive memory consumption on the other. In applications where these needs are more clearly known a priori, the sizes of the caches can be adjusted to potentially result in lower memory consumption.

**Table 6.1:** RAM consumed by various storage areas

| *Storage Area* | *# Items* | *Bytes/Item* | *Total Bytes* |
|---|---|---|---|
| Session Keys ($n_k$) | 10 | 22 | 220 |
| Public Keys ($n_p$) | 12 | 40 | 480 |
| Credentials ($n_c$ ) | 12 | 16 | 192 |
| Model ($n_m$) | 16 | 6 | 96 |
| **Total** | | | **988** |

The justification for the choice of the number of items in each storage area is as follows. Assume a node $N_i$ offers $n_{si}$ services and has $m_i$ neighbors. In the worst case a session key is needed for each service on all of $N_i$'s neighbors and for every neighbor connecting to $N_i$'s services. The number of session keys $n_k$ is given by

$$n_k = \left( \sum_{j=1}^{m_i} n_{sj} \right) + m_i n_{si}$$

where $n_{sj}$ represents the number of services on neighbor $j$. For example if $N_i$ had five neighbors each offering one service and if $N_i$ offered one service, the total number of session keys required would be 10. Sprocket presumes a small number of neighbors with a small number of services on each node. Notice, however, that this does not preclude using the system in a large network; SpartanRPC is a link-layer protocol and is only concerned with the immediate neighbors of a node.

The number of public keys is related to the complexity of the access policies used by the services. The intersection credential mentions three public keys so in the worst case the number of public keys $n_p = 3n_c$ where $n_c$ is the number of credentials in the credential storage. However, the intersection credential is rare and all other credentials only mention two public keys. This suggests an upper bound closer to $n_p = 2n_c$.

In real policies, however, it is necessary for the same public key to be mentioned in

more than one credential. For example consider a simple credential chain such as $E_1.r_1 \leftarrow E_2.r_2, \ldots, E_i.r_i \leftarrow E_{i+1}.r_{i+1}, \ldots, E_n.r_n \leftarrow E_{n+1}$. In this case the number of credentials is $n$ and the number of unique public keys is $n + 1$. It appears reasonable to suppose that in realistic policies the number of credentials and the number of public keys are about the same. For this reason Sprocket sets $n_p = n_c$.

More difficult to judge is the number of credentials involved in real-world authorization scenarios. Clearly this will be application specific and will vary widely. However, two or three credentials needed to establish authorization is a reasonable assumption, since most likely application designers will avoid complicated policies in a resource constrained setting. Thus $n_c \approx 3n_d$ where $n_d$ is the number of interacting domains, assuming each domain provides a single protected service. Assumming that only two other domains will be in the immediate vicinity of a node $N_i$, then $n_d = 3$. Sprocket sets $n_c = 12$ to provide some space for the case when a neighboring domain offers more than once service.

If every entity defines the same roles and if the policies are such that every entity is in every role, then the number of model tuples required is $n_m = n_r n_p^2$ where $n_r$ is the total number of roles involved. This value is unrealistically large, however. In a system where access is widely granted (cooperating domains) a value $n_m = n_r n_d$ would be more appropriate. Sprocket assumes that $n_d$ is about three and that $n_r$ is about four or five, thus $n_m = 16$ is used.

Table 6.2 shows the overall memory consumption of two small client/server pairs. The baseline pair handle all communication through normal Active Message packets that are explicitly programmed by the user. The SpartanRPC pair uses Sprocket which includes support for certificate distribution and verification, session key management, authorization logic, and MAC computations. The Directed Diffusion entry shows the memory consumption of the demonstration program that implements that algorithm.

Although the overhead incurred by the Sprocket runtime system is significant on this

**Table 6.2:** Memory consumption of test programs

| *Test Program* | *RAM Bytes* | *ROM Bytes* |
|---|---|---|
| Baseline Client | 349 | 10982 |
| Baseline Server | 283 | 10490 |
| SpartanRPC Client | 2222 | 23108 |
| SpartanRPC Server | 2126 | 23394 |
| Directed Diffusion | 3105 | 27826 |

test platform, nearly 80% of RAM and 50% of ROM resources are still available. Furthermore, these memory usage numbers scale well to denser neighborhoods and extended RPC services because many aspects of the runtime system, in particular the RAM reserved for SpartanRPC, are independent of the number of RPC services in use.

### 6.2.2 Transient and Steady State Processor Overhead

The execution performance of Sprocket generated programs displays two distinct behaviors. The first is a transient behavior that occurs after a node boots when certificates are exchanged and session keys are negotiated, on demand, between the new node and its neighbors. The second is a steady-state behavior that occurs during normal operation. The transient overhead of Sprocket is large but the steady state overhead is not. In a quasi-static environment, where new nodes enter the network infrequently, the transient costs are amortized and it is the small, steady state overhead that dominates.

To explore the steady state overhead three tests were conducted.

1. A baseline test where the message handling was done explicitly using traditional Active Message interfaces.

2. A duties test where Sprocket was used but no authorization was requested. This

**Table 6.3:** Maximum message transfer rate

| *Test* | *messages/s* | *% Reduction* |
|---|---|---|
| Baseline | 128 | – |
| Duties | 119 | 7.0 |
| MAC | 87 | 32.0 |

is equivalent to using the authorization components `ACNullC` and `ASNullC` in Figure 3.9.

3. A MAC test where authorization was requested but where the session key storage areas were preloaded with appropriate session keys.

Table 6.3 shows the maximum rate at which messages could be sent and received by the test programs mentioned above. Note that the MAC test made use of the hardware assisted AES support provided by the CC2420 radio chip. These results show that maximum message send rates decrease by a factor of 7% due to the addition of the duties program logic, and further decreases by a factor of 25% due to MAC calculations. It is noted that the latter overhead would be incurred in any system using CC2420 MAC calculations.

The transient runtime overhead of this system can be subdivided into three primitive operations: the time required to transmit and verify a certificate, the time required to build the minimum model, and the time required to negotiate a session key. Two of these operations require lengthy public key computations and dominate the transient behavior. Thus the performance in this regard is closely tied to the performance provided by TinyECC.

TinyECC provides a number of tunable parameters that can be used to optimize performance by trading off space and time (Liu and Ning 2008). Since the tests on this system had no particular application constraints in mind, the TinyECC "out of the box." was used. However, TinyECC's optimizations can be used to tune the performance of the system to better match a particular application. For example, activating the Shamir Trick cut cer-

tificate verification time in half at the expense of increasing RAM usage by nearly 700 bytes.

Table 6.4 shows the times required for each of the primitive transient operations. The time required to build the minimum model is directly related to the number and nature of the credentials involved. In this test a collection of five representative credentials that included at least one of each type was used. In any case this time is entirely negligible compared to the other transient operations.

**Table 6.4:** Processing time for transient operations

| *Operation* | *Time* |
|---|---|
| Certificate Verification | 82s |
| Minimum Model Construction | 370$\mu$s |
| Session Key Negotiation | 80s |

The time quoted for session key negotiation represents the time required for both negotiating partners to compute the session key. In the current implementation the two negotiating nodes do this sequentially with the server node computing the session key before responding to the client node. This was done in case the session key computation failed on the server to ensure that the client does not falsely believe a session key was successfully negotiated.

### 6.2.3 Transient Times for Directed Diffusion

As argued above, the overhead imposed by Sprocket is primarily the time the network spends in an initial transient state when credentials are verified and session keys are negotiated. Subsequently, the network enters a steady state during which the main cost is a 32% reduction in *maximal* message send rates due to hardware AES encryption. In order to evaluate the performance of Sprocket in a realistic application, therefore, the transient

times of the demonstration directed diffusion application were quantified. The experiments elected a single node to repeatedly express an interest and observe how long was required for that interest to flood the network. This time depends on three major factors:

1. The number of certificates transferred.

2. The number of neighbors for each node.

3. The number of hops to the "far" edge of the network.

Two experiments were conducted, one on a single hop (star) network and another on a multi-hop (mesh) network.

In the single hop case, transient time $T$ can be described by the following equation:

$$T = n_c B + V + n_n K$$

where $B$ is the certificate broadcast interval, $V$ is the certificate verification time, $K$ is the session key negotiation time, $n_c$ is the number of certificates and $n_n$ is the number of neighbors. Since $B$ was set to 90 seconds, which is greater than $V$, certificate verification for $n_c$ certificates takes time $n_c B + V$ given a 90 second system initialization period. And since session keys need to be negotiated with $n_k$ neighbors in turn, $T$ also comprises a $n_n K$ delay. Table 6.5 shows the transient time required to flood a network where all nodes are one-hop neighbors of the root node. Values are given for three different policies with different numbers of certificates transferred from the root to the neighbors.

The behavior of the system was explored in a multi-hop environment by creating a linear mesh network. Each node (except the root) had a single downstream neighbor. All nodes were booted simultaneously and the time required for interest information to reach each node was observed. The policy used required only a single certificate to be transferred between nodes. Table 6.6 shows the results of several runs.

**Table 6.5:** Transient time in single hop directed diffusion

| *# neighbors* | *1 Cert* | *2 Certs* | *3 Certs* |
|---|---|---|---|
| 1 | 4m03s | 5m27s | 6m52s |
| 2 | 5m16s | 6m50s | 8m24s |
| 3 | 6m32s | 7m57s | 9m30s |
| 4 | 7m50s | 9m22s | 10m51s |

**Table 6.6:** Transient time in multi-hop directed diffusion

| *Run* | *1 hop* | *2 hops* | *3 hops* |
|---|---|---|---|
| 1 | 4m05s | 7m24s | 9m10s |
| 2 | 3m12s | 5m12s | 6m30s |
| 3 | 3m57s | 7m37s | 9m15s |
| 4 | 4m09s | 7m15s | 8m49s |
| *Average* | 3m51s | 6m52s | 8m23s |

The reason for variations in transient times over each run was due to a randomized element in the protocol, specifically a randomized $\pm 10\%$ interval in certificate broadcast times to avoid collisions. In these results it is essential to note that for hops $> 2$, extra transient time is comprised solely of session key negotiation times (80s per session key, see Table 6.4) that are forced by duty postings as interests propagate through the network. Certificates are broadcast and verified in parallel throughout the network upon system bootup, during the same time period required for the root's interest to propagate through the first and second hops.

### 6.2.4 Snowcloud with Sprocket

To explore the real-world feasibility of using SpartanRPC and Sprocket, the unsecured versions of the Harvester and sensor node programs described in section 6.1 were enhanced to use SpartanRPC for access control.

To specify and implement the security policies informally described previously, the sensor network and the Harvester single node "network" were considered as separate security domains, each with its own set of credentials. The sensor network is always endowed with administrator-level credentials. If a Harvester is to be used by a system engineer, it is also endowed with administrator-level credentials, whereas a Harvester to be used by a data end-user is only endowed with user-level credentials. When a Harvester is introduced to the sensor network, its resource accesses are mediated by its authorization level. Since credentials are unforgeable, a user-level Harvester can never be used for sensor network control even if it is reprogrammed.

Sensor nodes within the network possess four credentials, as follows. In these credentials the Snowcloud domain is abbreviated $SC$. Authority to collect data and control sensors in the network are governed by the roles $SC.Col$ and $SC.Con$, respectively. Cre-

dential (1), below, says that any node with control authority also has collection authority. (2) says that nodes in the Snowcloud domain have control authority. (3) says that any entity in a Snowcloud collaborator's $Usr$ role has collection authority. (4) says that the node identified by $Nid$ is in the Snowcloud domain.

$$(1) \quad SC.Col \longleftarrow SC.Con \qquad (2) \quad SC.Con \longleftarrow SC.Node$$

$$(3) \quad SC.Col \longleftarrow SC.Collab.Usr \qquad (4) \quad SC.Node \longleftarrow NId$$

When invoking remote services, the node will do so on behalf of the entity $NId$. It will also be imaged with the $NId$ private key for session key negotiation.

Any Harvester within the Snowcloud domain is given the credential $SC.Node \longleftarrow HId$ and the $HId$ private key issued by Snowcloud domain administration. This will provide that Harvester with collection and control authority in the domain. For Harvesters to be provided to collaborators, the Snowcloud administrators issue a credential establishing the institution as a collaborator, while the institution itself may define and manage policy for their $Usr$ role membership. For example, the University of New Hampshire, represented by $RT$ entity $UNH$, can be established as a collaborator with credential (5), below, issued by Snowcloud domain administration, and may specify role membership with the credential (6) issued by UNH domain administration:

$$(5) \quad SC.Collab \longleftarrow UNH \qquad (6) \quad UNH.Usr \longleftarrow UsrID$$

These two credentials, along with the $UsrID$ private key, are imaged on Harvesters used by UNH collaborators for data collection, but which remain unauthorized for control. Significantly UNH could program their own Harvester nodes without the Snowcloud domain being involved aside from providing credential (5) above. The policy set by UNH to decide who, exactly, is in the $UNH.Usr$ role is of no concern to the Snowcloud domain administrators.

**Implementation**

Resources themselves are accessed through a secure command dissemination protocol, that is modeled upon the TinyOS Dissemination protocol (as described in TEP 118). In short, protected RPC services establish network level broadcast channels requiring authorization for use. Commands are communicated to the network over these channels, and different channels are used for different sorts of commands.

In more detail, command broadcast services can be specified as a duty in a remotable interface:

```
interface SpDissemUpdate {
    duty void change( command_t new_value );
}
```

To implement, e.g., the control command channel, the following module can be defined and included on sensor nodes in the Snowcloud domain:

```
module ControlDissemC {
    provides remote
      interface SpDissemUpdate requires "SC.Con";

    uses interface SpDissemUpdate as NeighborUpdate;
    provides interface ComponentManager;
}
implementation { ... }
```

In the implementation, the provided `SpDissemUpdate` interface accepts command invocations from neighbors, but requires them to be authorized for the $SC.Con$ role. Commands are relayed to all other neighbors (i.e., disseminated) via the used `NeighborUpdate` interface; those neighbors are identified by the provided `ComponentManager`.

To use this component, both sensor and Harvester nodes can configure it through the following component instantiation and wiring, where the component's `NeighborUpdate` interface is wired remotely to neighbors:

```
components ControlDissemC as ControlChan;
activate "*" for
    ControlChan.NeighborUpdate ->
        [ControlChan].SpDissemUpdate;
```

Note that a node must be endowed with the appropriate credentials for this wiring to be useful.

This same code pattern can be used to implement a data collection request channel, protected by the $SC.Col$ role instead of $SC.Con$. In response to an authorized control command invocation, sensor nodes will modify their behavior appropriately, whereas in response to authorized data collection requests sensor nodes will report their data using collection tree protocol (TEP 123) to the Harvester.

**Results**

Results can be characterized according to both the user experience and to quantitative aspects. As detailed in subsection 6.2.2, a one-time transient overhead is imposed for initial credential exchange and session key negotiation when a Harvester is first introduced to the network. However, since data collection for a network after several months of deployment can take up to an hour, this overhead is relatively insignificant. And steady-state overhead is small, and does not significantly affect data collection rates. Thus, authorized user experience is not negatively impacted by the addition of security.

From a quantitative perspective, the most important measurements to consider for this application are RAM and ROM consumption of the unsecure and secured versions of the Harvester collection protocol. It must be considered whether layering SpartanRPC security over a realistic application will overrun the resources available to a node. Relevant measurements are shown in Table 6.7.

Both RAM and ROM consumption are significantly increased by the addition of Spar-

**Table 6.7:** RAM and ROM comparison for SpartanRPC Snowcloud

| *Program* | *RAM Bytes* | *ROM Bytes* |
|---|---|---|
| Unsecure Harvester | 2274 | 24316 |
| Secure Harvester | 4771 | 35834 |
| Unsecure Sensor Node | 2868 | 36254 |
| Secure Sensor Node | 5417 | 48616 |

tanRPC security to this application. However, these numbers are within operating parameters. Also Sprocket has not yet been optimized so additional improvements could likely be made.

## 6.3 Scalaness/nesT

The generality of Scalaness makes a full evaluation of the system difficult to interpret. However, in keeping with the aim to demonstrate trust management in embedded systems, Scalaness was applied to the problem of supporting trust management in the Snowcloud application in a manner similar to that described in subsection 6.2.4. It should be noted, however, that as a general staged programming system, Scalaness can be used for many purposes; building authorization systems is only one application. Furthermore Scalaness could be used to support authorization in various ways depending on the trade offs needed between node efficiency, deployment frequent, and system functionality.

### 6.3.1 Snowcloud with Scalaness

To demonstrate a staged solution to providing trust management in Snowcloud, a Scalaness program called *Snowstorm* was developed. Snowstorm is intended to be run by each security domain participating in a deployment. It targets a conventional machine with Internet

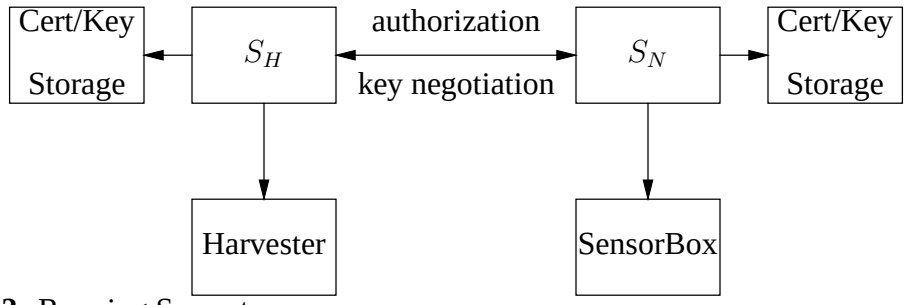


**Figure 6.2:** Running Snowstorm

connectivity and arbitrary resources.

Figure 6.2 shows two instances of Snowstorm running, $S_H$ and $S_N$, one by each of two administrative domains. $S_N$ is run by the sensor network administrators and is only interested in generating the sensor node application. $S_H$ is run by the collaborating domain and is only interested in generating the Harvester application. Normally, the two domains would probably run completely independent Scalaness programs, perhaps using a common library, but as a convenience during development a single program was created to serve the needs of both domains.

$S_N$ reads the access policy from suitable configuration files (or as entered by the user) consisting of $RT_0$ credentials in a convenient syntax. $S_N$ and $S_H$ run continuously and communicate via the Internet. Both programs provide an interactive user interface with features for generating and managing keys, issuing credentials, and storing policy statements and credentials from its peers. As directed by its user $S_H$ requests access to node collection or control resources, causing authorization and session key negotiation to all take place automatically. Once session keys are available, the user can direct Snowstorm to generate the appropriate node level program, with $S_N$ generating the sensor node program and $S_H$ generating the Harvester program. The nesT modules that will communicate during stage two execution are specialized with the previously computed session key values.

The development of Snowstorm was a straight forward exercise in software engineering; most of the program is ordinary Scala. Snowstorm makes use of widely used third

party Java libraries for Internet communication and ECC cryptographic operations. Thanks to the expressive power of the Scala language, it was possible to implement the core $RT_0$ authorization decision in just 90 lines. Furthermore, although Snowstorm has only a text-mode interface it would have been a simple matter to endow it with a fully fledged graphical interface if desired. A majority of Snowstorm development was done without any specialized knowledge of embedded systems development, a point of significance since embedded systems programming often requires different training and experience from that used by general application developers.

When asked to generate their node level programs, Snowstorm specializes a few key nesT modules with key information and then composes those modules to form fully functioning node programs. When deployed to the nodes, these programs behaved as did the original implementation. Anecdotally the Scalaness type system proved its worth several times during the development of Snowstorm. The compiler detected improper wirings as type errors, thus preventing nonsense compositions of nesT modules.

Snowstorm's implementation also made extensive use of external nesC libraries. In fact, the bulk of the original, tested sensor node and Harvester programs were wrapped as external libraries in the manner described in subsection 5.1.2. NesT modules were created primarily to hold key material and to interact with the AES encryption hardware on the CC2420. No significant changes were needed to the existing code base.

### 6.3.2 Memory Usage

To explore the efficiency of Scalaness generated programs, the memory consumption of the generated code was measured. Table 6.8 shows the results with the memory values of the Sprocket version shown in Table 6.7 duplicated in the "Unstaged" column as a convenience.

The "Savings" are the percent reduction from unstaged to staged secure implemen-

**Table 6.8:** RAM and ROM comparison for Scalaness Snowcloud

| | Unsecured | Unstaged | Staged | Savings |
|---|---|---|---|---|
| Sensor ROM | 36254 | 48616 | 36596 | 25% |
| Sensor RAM | 2868 | 5417 | 3038 | 44% |
| Harvester ROM | 24316 | 35834 | 24436 | 32% |
| Harvester RAM | 2274 | 4771 | 2402 | 50% |

tation, and these numbers demonstrate that the potential for saving both RAM and ROM space is significant. Unsurprisingly the memory consumed by the Scalaness generated code is virtually identical to that used by the unsecured programs. The only overhead injected into the staged node programs is that required to interact with the AES encryption hardware and, of course, to hold the negotiated session key material.

From the perspective of user experience, the staged version of this application is more convenient, since no initial authorization period is needed when the harvester is first introduced to the network. The staged version also exposes the system to fewer bugs and failures that would be obstacles to the primary goal of data collection. On the other hand the staged version requires the presence, somewhere in the deployment cycle, of a powerful machine on which the first stage program can be executed.

# Chapter 7

# Conclusion

This dissertation has described two language-level approaches for providing, for the first time, trust management style authorization to resource constrained embedded systems. One approach, SpartanRPC, is based on a remote procedure call discipline with primitives for specifying authorization requests and requirements. The other approach, Scalaness, makes use of staged programming to off-load complicated security computations to a higher powered machine.

As a method for providing distributed trust management to resource constrained systems, both approaches are feasible. SpartanRPC demands considerable resources on the devices, limiting the amount of memory and processor time available for application logic. In particular, SpartanRPC enabled applications exhibit transient start-up times measured in minutes, although maximum steady-state message transfer rates exhibit a degradation of only about 30%. In addition, the SpartanRPC runtime system consumes approximately 13 KiB of ROM and 2 KiB of RAM. Despite these significant overheads, realistic applications can nevertheless use the system as evidenced by the SpartanRPC-enabled version of the Snowcloud application.

SpartanRPC is fundamentally a link-level protocol. Since the number of neighbors in

a typical sensor network remains small as the network grows, the applicability of SpartanRPC is only weakly impacted by the total size of the network. The main issue is in the relatively long session key negotiation time; the first time a message floods the network an extremely long time may pass before the message reaches the network frontier since session key negotiations must occur sequentially at each hop.

Scalaness has the potential of greatly reducing the load on the embedded devices. In a trust management context, with a Scalaness program pre-computing session keys, the long transient start-up time and large memory overheads of SpartanRPC are all but eliminated. The very slow network flooding time experienced by SpartanRPC applications is also removed. However, The 30% reduction in maximum message transfer rate remains.

Scalaness does require a deployment scenario where a more powerful machine is available to specialize the device programs. In some scenarios the time required to generate and deploy the specialized node programs might be significant, negating somewhat the advantage in transient start-up time Scalaness has relative to SpartanRPC. However, Scalaness is a far more flexible system, admitting other kinds of deployment scenarios and application use-cases besides those available to the more limited SpartanRPC.

Indeed, Scalaness represents a more principled approach to generating efficient embedded systems software in general, as evidenced by the formal description of the system in chapter 4 and in the foundational ⟨ML⟩ work (Liu, Skalka, and Smith 2012). Scalaness provides a unique combination of staging with process separation, dynamic type construction, and a cross-stage type safety conjecture that enable the robust and efficient generation of many embedded systems applications. In a Scalaness context, the embedded trust management problem is nothing more than a demonstration application.

Both SpartanRPC and Scalaness are tied to the nesC programming language, either by extending nesC in the case of SpartanRPC or by translating a specialized language into nesC in the case of Scalaness. However, the systems described here are not specific to

sensor networks and would be applicable in any environment where nesC is used. Furthermore, although nesC was developed for sensor networks, it could be used as a general purpose embedded systems language.

## 7.1 Future Work

Possible future directions of this work can be divided into two broad categories: generalizing the systems and providing additional safety guarantees.

The Sprocket implementation of SpartanRPC is already modular enough to support alternate (and even multiple, simultaneous) authorization mechanisms. It would be interesting to experiment with richer trust management languages such as $RT_1$ and its variations to see how expressive a trust management language could be supported on constrained devices. Currently the $RT_0$ authorization logic uses minimal time and space so conceivably fairly complex trust management languages could be supported without significantly increasing the overall overhead of the system. Notice that the current version of Scalaness already supports arbitrary trust management languages because the first stage program runs in an environment with relatively infinite resources.

Sprocket currently supposes that neighboring nodes communicate over a radio link. However, this assumption is only reflected in the code generated by Sprocket for the stubs and skeletons. It would be a simple engineering matter to modify Sprocket to generate stubs and skeletons for some other communication technology such as TCP/IP or the Controller Area Network (CAN) bus widely used in automotive embedded systems (Pazul 1999).

SpartanRPC is, however, closely tied to nesC because of the way it defines and uses dynamic wires. In contrast, the current implementation of Scalaness formally translates nesT to nesC as it generates the second stage program. This final translation step could be modified to produce a different language, such as C, with no change to the founda-

tional semantics. This would make the system applicable to a larger group of embedded developers.

The type safety guarantee provided by Scalaness is valuable but embedded systems have other correctness needs as well. Many embedded systems are used in safety critical applications where assurance of freedom from runtime errors, such as array bounds overflow, is essential. Systems exist that can analyze Ada or C programs to prove freedom from such errors (Barnes 2000; Cuoq, Kirchner, Kosmatov, Prevosto, Signoles, and Yakobowski 2012) and those systems could conceivably be applied to the output of Scalaness now. However, it would be an interesting and challenging direction for future work to extend Scalaness so the programmer could be assured that *all possible* generated programs were free of important classes of runtime errors.

# Appendix A

# Scalaness/nesT Sample

This appendix shows a simple Scalaness/nesT sample. The sample composes two nesT modules being used as part of a hypothetical communication protocol. One module formats messages for transmission and the other module computes checksums.

It is reasonable for the checksum module to be separate since different applications may wish to use different checksum algorithms. In fact, the Scalaness program could dynamically select one of several candidate checksum modules as it composes the overall application, although that feature is not demonstrated here.

Beginning with the Scalaness program itself: In this simple example the entire program is contained in a single Scala object holding the main method. The program begins by declaring objects representing the wrapped nesC libraries needed. It then declares classes representing the nesT components to be used, defines some helper methods, and executes the main body. Notice that the type abbreviation binders (Watson 2013) are used to introduce convenience objects holding module type information.

This program demonstrates dynamic type construction, nesT module type and value parameters, module instantiation, and wiring. The program also demonstrates returning a dynamically constructed type from a Scala method.

The full source code is below.

```
//-----------------------------------------------------
// FILE   : Main.scala
// SUBJECT: Scalaness checksum sample.
//-----------------------------------------------------

object Main {

  import edu.uvm.nest._
  import edu.uvm.scalaness._
  import LiftableTypes._

  // The Scalaness representation of the library imports.
  @ModuleType(
    """{}
       <;>
       { booted(): Void,
         fired(): Void; }""")
  object LibraryIC extends NesTComponent {
    external("LibraryIC.nc")
  }


  // The Scalaness representation of the library exports.
  @ModuleType(
    """{}
       <;>
       { ; startPeriodic(period: UInt32): Void }""")
  object LibraryEC extends NesTComponent {
    external("LibraryEC.nc")
  }


  // A component for computing checksums.
  @ModuleType(
    """{}
       < checksumType <: UInt32; size: UInt16 >
       { ; compute_checksum(
             data: Array[UInt8]): checksumType }""")
  class ChecksumC extends NesTComponent {
    "ChecksumC.nc"
```

```
}


// A component for creating messages.
@ModuleType(
  """{}
     < checksumType <: UInt32; size: UInt16 >
     { compute_checksum(
         data: Array[UInt8]): checksumType,
       startPeriodic(period: UInt32): Void;
       booted(): Void,
       fired(): Void }""")
class MessageFormatterC extends NesTComponent {
  "MessageFormatterC.nc"
}


/**
 * The following method returns a fully instantiated
 * nesT module for computing checksums. The precise
 * module created depends on runtime information.
 */
def getChecksummer(
  size: UInt16, checksumType: MetaType[UInt32]) = {

  @ModuleType(
    """{}
       < checksumType <: UInt32; size: UInt16 >
       { ; compute_checksum(
             data: Array[UInt8]): checksumType }""")
  val CheckSummer = new ChecksumC

  @ModuleType(
    """{ checksumType <: UInt32 }
         <;>
         { ; compute_checksum(
               data: Array[UInt8]): checksumType }""")
  val instCheckSummer =
    CheckSummer.instantiate(size, checksumType)

  instCheckSummer
}
```

```
/**
 * The main method obtains configuration information
 * from the command line and composes the final
 * program.
 */
def main(args: Array[String]) {

  // Create type abbreviations for convenience.

  // An uninstantiated module type.
  val MesgT = new TypeAbbreviation(
    """{}
       < checksumType <: UInt32; size: UInt16 >
       { startPeriodic(period: UInt32): Void,
         compute_checksum(
           data: Array[UInt8]): checksumType;
         fired(): Void,
         booted(): Void }""", List())

  // An instantiated module type.
  val FormT = new TypeAbbreviation(
    """{ checksumType <: UInt32 }
       <;>
       { compute_checksum(
           data: Array[UInt8]): checksumType,
         startPeriodic(period: UInt32): Void;
         booted(): Void,
         fired(): Void }""", List())

  // An instantiated module type.
  val CheckT = new TypeAbbreviation(
    """{ checksumType <: UInt32 }
       <;>
       { ;
         compute_checksum(
           data: Array[UInt8]): checksumType }""",
    List())

  // A runnable module type.
  val ResultT = new TypeAbbreviation(
```

```
  """{ checksumType <: UInt32 }
     <;>
     { ; }""", List())

// Return a MetaType based on command line argument.
def getChecksumType(args: Array[String]) = {
  args(0).toInt match {
    case  8 =>
      println("Selecting 8 bit checksums")
      new MetaType[UInt32](NesTTypes.UInt8)

    case 16 =>
      println("Selecting 16 bit checksums")
      new MetaType[UInt32](NesTTypes.UInt16)

    case 32 =>
      println("Selecting 32 bit checksums")
      new MetaType[UInt32](NesTTypes.UInt32)
  }
}

// Method that returns a liftable value.
def getSize(args: Array[String]) = {
  val size = args(1).toInt
  println(s"Selecting $size byte message blocks")
  new UInt16(size)
}


if (args.length != 2)
  println("Usage: Main bit_length block_size")
else {

  // Get run time information about types/values.
  val desiredChecksumType = getChecksumType(args)
  val desiredSize = getSize(args)

  // Uninstantiated message formatter.
  @TypeAbbr(MesgT)
  val MessageFormatter =
    new MessageFormatterC
```

```
      // Instantiated message formatter.
      @TypeAbbr(FormT)
      val formattingModule =
          MessageFormatter.instantiate(
            desiredSize, desiredChecksumType)

      // Compute appropriate checking module.
      @TypeAbbr(CheckT)
      val checkingModule =
          getChecksummer(
            desiredSize, desiredChecksumType)

      // Wire things together.
      @TypeAbbr(ResultT)
      val resultModule =
          LibraryIC +>
            formattingModule +> checkingModule +>
              LibraryEC

      // Generate the nesT/nesC.
      resultModule.image()
    }
  }

}
```

Below is the nesT implementation of the checksum module. This version uses a simple arithmetic summation. It is parameterized by the type used to hold the checksum and by a value representing the size of the array to be processed. Thus specializations of this module can only operate on fixed sized arrays, presummably the size of some standard message format.

```
// Type : checksumType: Type used to hold a checksum.
// Value: size: Size of the data array to process.
module ChecksumC {
  provides command
    checksumType compute_checksum(uint8_t data[]);
}
implementation {
```

```
  // Computes a simple checksum over the data array.
  command checksumType compute_checksum(uint8_t data[])
  {
    checksumType sum = 0;
    int16_t i;

    // Casting from uint16_t to int16_t is explicitly
    // enabled in type compatibility relation. The
    // compiler uses a built-in implementation of this
    // conversion.
    //
    for( i = 0; i < (int16_t)size; ++i ) {
      sum += data[i];
    }
    return sum;
  }

}
```

The listing below illustrates how the Scalaness compiler rewrites the checksum module to pure nesC in the example program given. This listing also shows the result of type and value specialization, in this instance implemented by way of simple substitution. This version of the module was created for 8 bit checksums on eight element data arrays.

Notice the addition of compiler generated variables to hold dynamic size information for the array expressions. These variables are checked to ensure memory safety as described in subsection 5.1.5.

```
module ChecksumC  {
  provides {
    command uint8_t compute_checksum(
      uint8_t data[], uint16_t _sc_data_SIZE );
  }
  uses {
    command void boundsCheckFailed( );
  }
}
implementation {
```

```
  command uint8_t compute_checksum(
    uint8_t data[], uint16_t _sc_data_SIZE)
  {
    uint8_t sum = 0;
    int16_t i;
    for( i = 0; i < (int16_t)(8); ++i )
    {
    {
      int _sc_2 = i;
      if( _sc_2 >= _sc_data_SIZE )
        call boundsCheckFailed( );
      sum += data[_sc_2];
    }
    }
    return sum;
  }

}
```

The message formatting module constructs a “message” consisting of ascending byte values and then computes a checksum over that message. Nothing more is done with the message in this simple demonstration. A more realistic program would then send the message to an underlying communication module for transmission.

```
// Type : checksumType: Type used to hold a checksum.
// Value: size: Size of the data array to process.
//
// Main program of the node.
module MessageFormatterC {

  uses command
    checksumType compute_checksum(uint8_t data[]);
  uses command void startPeriodic( uint32_t period );
  provides command void booted( );
  provides command void fired( );
}
implementation {

  command void booted( )
  {
```

```
    // Casting from int16_t to uint32_t is explicitly
    // enabled in type compatibility relation.
    //
    call startPeriodic( (uint32_t)1000 );
  }

  // Called once per second.
  command void fired( )
  {
    uint8_t  raw[size];
    uint16_t i;
    checksumType checksum;

    // Construct message.
    for( i = 0U; i < size; ++i ) {
      raw[i] = (i & 0x00FF);
    }
    checksum = call compute_checksum( raw );

    // Other program components are used to send
    // the message with checksum.
  }

}
```

The final sample fragment below shows the specialized and rewritten nesC with the dynamic size of an array expression being passed to `compute_checksum`.

```
module MessageFormatterC  {
  provides {
    command void fired( );
    command void booted( );
  }
  uses {
    command void startPeriodic( uint32_t period );
    command uint8_t compute_checksum(
      uint8_t data[], uint16_t _sc_data_SIZE );
    command void boundsCheckFailed( );
  }
}
implementation {
```

```
  command void booted( )
  {
    call startPeriodic( (uint32_t)( 1000 ) );
  }

  command void fired( )
  {
    uint8_t  raw[8 ];
    uint16_t i;
    uint8_t  checksum;
    for( i = 0U; i < 8; ++i )
    {
    {
      int _sc_1 = i;
      if( _sc_1 >= 8 )
        call boundsCheckFailed( );
      raw[_sc_1] = ( i & 0x00FF );
    }
    }
    checksum = call compute_checksum( raw, 8 );
  }

}
```

# Bibliography

Abadi, M. (1998). On SDSI's linked local name spaces. *Journal of Computer Security 6*(1–2), 3–21.

Abadi, M. (2003, June). Logic in access control. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Sciennce*.

Abadi, M., M. Burrows, B. Lampson, and G. Plotkin (1993, September). A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems 15*(4), 706–734.

Ajmani, S., D. E. Clarke, C.-H. Moh, and S. Richman (2002, January). ConChord: Cooperative SDSI certificate storage and name resolution. In *International Workshop on Peer-to-Peer Systems*.

Ancona, D. and E. Zucca (2002). A calculus of module systems. *Journal of functional programming 11*, 91–132.

Barnes, J. (2000, December). The spark way to correctness is via abstraction. *Ada Lett. XX*(4), 69–79.

Bauer, L., M. A. Schneider, and E. W. Felten (2002, August). A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, pp. 93–108.

Becker, M. Y. and P. Sewell (2004, June). Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*.

Bergstrom, E. and R. Pandey (2007). Anycast-RPC for wireless sensor networks. In *Proceedings of the IEEE international conference on mobil adhoc and sensor systems*, pp. 1–8.

Bertino, E., B. Catania, E. Ferrari, and P. Perlasca (2003, February). A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security 6*(1), 71–127.

Bertoni, G., L. Breveglieri, and M. Venturi (2006). ECC hardware coprocessors for 8-bit systems and power consumption considerations. *itng 00*, 573–574.

Blaze, M., J. Feigenbaum, J. Ioannidis, and A. D. Keromytis (1999, September). *RFC-2704: The KeyNote Trust-Management System Version 2*. Internet Engineering Task Force.

Blaze, M., J. Feigenbaum, and J. Lacy (1996, May). Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 164–173. IEEE Computer Society Press.

Blaze, M., J. Feigenbaum, and M. Strauss (1998). Compliance checking in the policymaker trust management system. In *Proceedings of the 2nd International Conference on Financial Cryptography*, pp. 254–274. Springer-Verlag.

Blaze, M., J. Ioannidis, and A. D. Keromytis (2002, May). Trust management for IPsec. *ACM Transactions on Information and System Security 5*(2), 95–118.

Blaze, M., J. Ioannidis, and A. D. Keromytis (2003, May). Experience with the keynote trust management system: Applications and future directions. In *Proceedings of the*

*1st International Conference on Trust Management*, Keraklion, Crete, Greece, pp. 284–300. Springer-Verlag.

Brogi, A., R. Popescu, F. Gutiérrez, P. López, and E. Pimentel (2008, April). A service-oriented model for embedded peer-to-peer systems. *Electron. Notes Theor. Comput. Sci. 194*, 5–22.

Brooks, R. R., P. Ramanathan, and A. M. Sayeed (2003, August). Distributed target classification and tracking in sensor networks. *Proceedings of the IEEE 91*(8), 1163–1171.

Burrows, M., M. Abadi, and R. M. Needham (1990, February). A logic of authentication. *ACM Transactions on Computer Systems 8*(1), 18–36.

Canetti, R., J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas (1999, mar). Multicast security: a taxonomy and some efficient constructions. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Volume 2, pp. 708 –716 vol.2.

Cardelli, L. (1997). Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, POPL '97, New York, NY, USA, pp. 266–277. ACM.

Cardelli, L. and P. Wegner (1985, December). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*(4), 471–523.

Çamtepe, S. A. and B. Yener (2005). Key distribution mechanisms for wireless sensor networks: a survey. Technical Report TR-05-07, Rensselaer Polytechnic Institute.

Chapin, P. (2013a, October). Scalaness home page. https://github.com/pchapin/scala. Accessed October 2013.

Chapin, P. (2013b, October). Sprocket home page. https://github.com/pchapin/sprocket.

Accessed October 2013.

Chapin, P. and C. Skalka (2010, November). SpartanRPC: Secure WSN middleware for cooperating domains. In *Proceedings of the Seventh IEEE International Conference on Mobile Ad-hoc and Sensor Systems*.

Chapin, P. and C. Skalka (2013). Spartan RPC. Technical report, University of Vermont. Submitted. `http://www.cs.uvm.edu/~skalka/skalka-pubs/chapin-skalka-spartanrpc`

Chapin, P., C. Skalka, S. Smith, and M. Watson (2013, October). Scalaness/nesT. type specialized staged programming for sensor networks. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences (GPCE '13)*.

Chapin, P. C., C. Skalka, and X. S. Wang (2008, August). Authorization in trust management: Features and foundations. *ACM Computing Surveys 40*, 9:1–9:48.

Chen, M., S. Gonzalez, A. Vasilakos, H. Cao, and V. C. Leung (2011, April). Body area networks: A survey. *Mob. Netw. Appl. 16*(2), 171–193.

Cheong, E. (2007). *Actor-Oriented Programming for Wireless Sensor Networks*. Ph. D. thesis, University of California, Berkeley.

Chlipala, A. (2010). Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI*.

Clarke, D., J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest (2001). Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security 9*(4), 285–322.

Claycomb, W. R. and D. Shin (2011, January). A novel node level security policy framework for wireless sensor networks. *J. Netw. Comput. Appl. 34*(1), 418–428.

Community, T. TinyOS community forum. http://www.tinyos.net/. Accessed November 2013.

Consel, C., L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé (1998, September). Tempo: specializing systems applications and beyond. *ACM Comput. Surv. 30*(3es).

Costa, P., L. Mottola, A. L. Murphy, and G. P. Picco (2007). Programming wireless sensor networks with the teenylime middleware. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, New York, NY, USA, pp. 429–449. Springer-Verlag New York, Inc.

Cremet, V., F. Garillot, S. Lenglet, and M. Odersky (2006). A core calculus for scala type checking. In *Proceedings of the 31st international conference on Mathematical Foundations of Computer Science*, MFCS'06, Berlin, Heidelberg, pp. 1–23. Springer-Verlag.

Culler, D., D. Estrin, and M. Srivastava (2004, August). Guest editors' introduction: Overview of sensor networks. *Computer 37*(8), 41–49.

Cuoq, P., F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski (2012). Frama-c: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, Berlin, Heidelberg, pp. 233–247. Springer-Verlag.

DeTreville, J. (2002). Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Diffie, W. and M. Hellman (2006, September). New directions in cryptography. *IEEE Trans. Inf. Theor. 22*(6), 644–654.

Dutta, P. K., J. W. Hui, D. C. Chu, and D. E. Culler (2006). Securing the deluge network

programming system. In *IPSN*, pp. 326–333.

Ellison, C., B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen (1999, September). *RFC-2693: SPKI Certificate Theory*. Internet Engineering Task Force.

Ferraiolo, D. and R. Kuhn (1992). Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pp. 554–563.

Flatt, M. and M. Felleisen (1998). Units: Cool modules for HOT languages. In *PLDI*.

Fletcher, J. G. (1982, jan). An arithmetic checksum for serial transmissions. *Communications, IEEE Transactions on 30*(1), 247 – 252.

Fok, C.-L., G.-C. Roman, and C. Lu (2009, July). Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Trans. Auton. Adapt. Syst. 4*, 16:1–16:26.

Fouladgar, S., B. Mainaud, K. Masmoudi, and H. Afifi (2006). Tiny 3-tls: a trust delegation protocol for wireless sensor networks. In *Proceedings of the Third European conference on Security and Privacy in Ad-Hoc and Sensor Networks*, ESAS'06, Berlin, Heidelberg, pp. 32–42. Springer-Verlag.

Frolik, J. and C. Skalka (2013). Snowcloud. Technical report, University of Vermont. Submitted. `http://www.cs.uvm.edu/~skalka/skalka-pubs/frolik-skalka-snowcloudt`

Ganeriwal, S., C. Pöpper, S. Čapkun, and M. B. Srivastava (2008, July). Secure time synchronization in sensor networks. *ACM Trans. Inf. Syst. Secur. 11*(4), 23:1–23:35.

Gao, T., C. Pesto, L. Selavo, Y. Chen, J. G. Ko, J. H. Lim, A. Terzis, A. Watt, J. Jeng, B.-R. Chen, K. Lorincz, and M. Welsh (2008, may). Wireless medical sensor networks in emergency response: Implementation and pilot results. In *Technologies for Homeland Security, 2008 IEEE Conference on*, pp. 187–192.

Garcia, M., A. Izmaylova, and S. Schupp (2010, July). Extending scala with database query capability. *The Journal of Object Technology 9*(4), 45–68.

Gay, D., P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler (2003). The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, New York, NY, USA, pp. 1–11. ACM.

Ghelli, G. and B. Pierce (1998). Bounded existentials and minimal typing. *Theoretical Computer Science 193*(1-2), 75 – 96.

González-Valenzuela, S., M. Chen, and V. C. Leung (2010, December). Programmable middleware for wireless sensor networks applications using mobile agents. *Mob. Netw. Appl. 15*, 853–865.

Gregor, D., J. Järvi, J. G. Siek, G. D. Reis, B. Stroustrup, and A. Lumsdaine (2006). Concepts: Linguistic support for generic programming in C++. In *OOPSLA*.

Grossman, D. J. (2003). *Safe Programming at the C Level of Abstraction*. Ph. D. thesis, Cornell University.

Gummadi, R., O. Gnawali, and R. Govindan (2005). Macro-programming wireless sensor networks using kairos. In V. Prasanna, S. Iyengar, P. Spirakis, and M. Welsh (Eds.), *Distributed Computing in Sensor Systems*, Volume 3560 of *Lecture Notes in Computer Science*, pp. 466–466. Springer Berlin / Heidelberg.

Gunter, C. A. and T. Jim (1997, September). Design of an application-level security infrastructure. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*.

Gunter, C. A. and T. Jim (2000a, January). Generalized certificate revocation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

*ming Languages*, pp. 316–329.

Gunter, C. A. and T. Jim (2000b). Policy-directed certificate retrieval. *Software: Practice & Experience 30*(15), 1609–1640.

Gupta, V., M. Millard, S. Fung, Y. Zhu, N. Gura, H. Eberle, and S. C. Shantz (2005). Sizzle: A standards-based end-to-end security architecture for the embedded internet (best paper). In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, Washington, DC, USA, pp. 247–256. IEEE Computer Society.

Halpern, J. and R. van der Meyden (1999). A logic for SDSI's linked local name spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pp. 111–122.

Hammond, K. and G. Michaelson (2003). Hume: A domain-specific language for real-time embedded systems. In *GPCE*, pp. 37–56. Springer-Verlag.

Herzberg, A., Y. Mass, J. Michaeli, D. Naor, and Y. Ravid (2000, May). Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Hong, F., X. Zhu, and S. Wang (2005). Delegation depth control in trust-management system. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pp. 411–414. IEEE Computer Society Press.

Howell, J. and D. Kotz (2000). A formal semantics for SPKI. Technical Report 2000-363, Dartmouth College.

Hu, W., P. Corke, W. C. Shih, and L. Overs (2009). secFleck: A public key technology platform for wireless sensor networks. In *EWSN '09: Proceedings of the 6th European Conference on Wireless Sensor Networks*, Berlin, Heidelberg, pp. 296–311.

Springer-Verlag.

Hu, W., H. Tan, P. Corke, W. C. Shih, and S. Jha (2010, August). Toward trusted wireless sensor networks. *ACM Trans. Sen. Netw. 7*, 5:1–5:25.

Hui, J., P. Levis, and D. Moss (2008, June). TinyOS 802.15.4 frames. http://www.tinyos.net/tinyos-2.x/doc/html/tep125.html. Accessed December 2011.

Hui, J. W. and D. Culler (2004). The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA, pp. 81–94. ACM.

Igarashi, A., B. C. Pierce, and P. Wadler (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23*(3), 396–450.

Intanagonwiwat, C., R. Govindan, D. Estrin, J. Heidemann, and F. Silva (2003, feb). Directed diffusion for wireless sensor networking. *Networking, IEEE/ACM Transactions on 11*(1), 2–16.

International Telecommunications Union (2000). *Information Technology - Open Systems Interconnection - The Directory: Public Key and Attribute Certificate Frameworks*. International Telecommunications Union.

International Telecommunications Union (2001). *Information Technology - Open Systems Interconnection - The Directory: Overview of Concepts, Models, and Services*. International Telecommunications Union.

ISO (2008). Iso/iec 1170-3:2008 information technology – security techniques – key management – part 3: Mechanisms using asymmetric techniques.

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Jim, T. (2001). SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE Computer Soceity.

Jim, T. and D. Suciu (2001). Dynamically distributed query evaluation. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, pp. 28–39. ACM Press.

Jung, W., S. Hong, M. Ha, Y.-J. Kim, and D. Kim (2009). Ssl-based lightweight security of ip-based wireless sensor networks. *Advanced Information Networking and Applications Workshops, International Conference on 0*, 1112–1117.

Karlof, C., N. Sastry, and D. Wagner (2004). TinySec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA, pp. 162–175. ACM.

Karlof, C. and D. Wagner (2003, September). Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols 1*(2–3), 293–315.

Kumar, S. S. and C. Paar (2006, July). Are standards compliant elliptic curve cryptosystems feasible on rfid? In *Proceedings of the 2006 Workshop on RFID security*.

Lee, Y. K., K. Sakiyama, L. Batina, and I. Verbauwhede (2008, nov.). Elliptic-curve-based security processor for RFID. *Computers, IEEE Transactions on 57*(11), 1514 –1527.

Leroy, X. (2006). Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pp. 42–54. ACM Press.

Levis, P. TEP-111: message_t. http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html.

Accessed August 2011.

Li, N. (2000, July). Local names in SPKI/SDSI. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, UK, pp. 2–15. IEEE Computer Society Press.

Li, N. and J. Feigenbaum (2001). Nonmonotonicity, user interfaces, and risk assessment in certificate revocation. In *Proceedings of the 5th International Conference on Financial Cryptography (FC01*, pp. 166–177. Springer-Verlag.

Li, N. and J. Feigenbaum (2002). Nonmonotonicity, user interfaces, and risk assessment in certificate revocation. In *Proceedings of the 5th International Conference on Financial Cryptography*, London, UK, pp. 166–177. Springer-Verlag.

Li, N., B. N. Grosof, and J. Feigenbaum (2003, February). Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security 6*(1), 128–171.

Li, N. and C. Mitchell (2006). Understanding spki/sdsi using first-order logic. *International Journal of Information Security 5*(1), 48–64.

Li, N. and J. C. Mitchell (2003a, January). Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*.

Li, N. and J. C. Mitchell (2003b, Apr). RT: A role-based trust-management framework. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*, pp. 201–212. IEEE Computer Society Press.

Li, N., J. C. Mitchell, and W. H. Winsborough (2002, May). Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 114–130. IEEE Computer Society Press.

Li, N., J. C. Mitchell, and W. H. Winsborough (2005, May). Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM 52*(3), 474–514.

Li, N., W. H. Winsborough, and J. C. Mitchell (2003, Feb). Distributed chain discovery in trust management. *Journal of Computer Security 11*(1), 35–86.

Liu, A. and P. Ning (2008). Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, Washington, DC, USA, pp. 245–256. IEEE Computer Society.

Liu, Y., C. Skalka, and S. Smith (2012). Type-specialized staged programming with process separation. *Higher-Order and Symbolic Computation 24*(4), 341–385.

Liu, Y. D., C. Skalka, and S. Smith (2009). Type-specialized staged programming with process separation. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, WGP '09, New York, NY, USA, pp. 49–60. ACM.

Liu, Y. D. and S. Smith (2002, July). A component security infrastructure. In *Proceedings of the 2002 Foundations of Computer Security Workshop*.

Lorincz, K., D. J. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton (2004). Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing 3*(4), 16–23.

Luk, M., G. Mezzour, A. Perrig, and V. Gligor (2007). MiniSec: a secure sensor network communication architecture. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, New York, NY, USA, pp. 479–488. ACM.

MacQueen, D. (1984). Modules for Standard ML. In *Proceedings of ACM Conference*

*on Lisp and Functional Programming*.

Madden, S., M. J. Franklin, J. M. Hellerstein, and W. Hong (2002). TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev. 36*(SI), 131–146.

Mainland, G. (2012). Explicitly heterogeneous metaprogramming with MetaHaskell. In *ICFP*.

Mainland, G., G. Morrisett, and M. Welsh (2008). Flask: staged functional programming for sensor networks. In *Proceeding of the 13th ACM SIGPLAN international conference on functional programming*, ICFP '08, New York, NY, USA, pp. 335–346. ACM.

Malan, D. J., M. Welsh, and M. D. Smith (2008, September). Implementing public-key infrastructure for sensor networks. *ACM Trans. Sen. Netw. 4*, 22:1–22:23.

Manzo, M., T. Roosta, and S. Sastry (2005). Time synchronization attacks in sensor networks. In *Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*, SASN '05, New York, NY, USA, pp. 107–116. ACM.

May, T. D., S. H. Dunning, G. A. Dowding, and J. O. Hallstrom (2007, March). An RPC design for wireless sensor networks. *International Journal of Pervasive Computing and Communications 2*(4), 384–397.

McDaniel, P. and A. D. Rubin (2001). A response to "can we eliminate certificate revocation lists?". In *Proceedings of the 4th International Conference on Financial Cryptography*, London, UK, pp. 245–258. Springer-Verlag.

Mitchell, J., S. Meldal, and N. Madhav (1991). An extension of standard ML modules with subtyping and inheritance. In *POPL*.

Moeser, C. D., M. Walker, C. Skalka, and J. Frolik (2011). Application of a wireless sensor network for distributed snow water equivalence estimation. In *Western Snow*

*Conference*.

Molhave, T. and L. H. Petersen (2005). Assignment Featherweight Java: Bringing mutable state to Featherweight Java. Master's thesis, University of Aarhus.

moteiv (2006, November). Tmote sky low power wireless sensor module. Datasheet.

Mottola, L. and G. P. Picco (2011, April). Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys 43*, 19:1–19:51.

Newton, R., G. Morrisett, and M. Welsh (2007). The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, New York, NY, USA, pp. 489–498. ACM.

Nikander, P. and L. Viljanen (1998). Storing and retrieving internet certificates. In *Proceedings of the Third Nordic Workshop on Secure IT Systems*.

OASIS (2006a). OASIS eXtensible Access Control Markup Language Technical Committee at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.

OASIS (2006b). OASIS Security Services Technical Committee at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=securit`

OASIS (2006c). OASIS Web Services Security Technical Committee at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.

Odersky, M., L. Spoon, and B. Venners (2011). *Programming in Scala, second edition*. Artima, Inc.

Pazul, K. (1999). Controller area network (can) basics. *Microchip Technology Inc. Preliminary DS00713A-page 1*.

Perillo, M. and W. Heinzelman (2005). *Fundamental Algorithms and Protocols for Wireless and Mobile Networks*, Chapter Wireless Sensor Network Protocols, pp. 813–

842. CRC Hall.

Perrig, A., J. Stankovic, and D. Wagner (2004). Security in wireless sensor networks. *Communications of the ACM 47*(6), 53–57.

Polakow, J. and C. Skalka (2006, June). Specifying distributed trust management in LolliMon. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*.

Raymond, D. and S. Midkiff (2008, jan.-march). Denial-of-service in wireless sensor networks: Attacks and defenses. *Pervasive Computing, IEEE 7*(1), 74 –81.

Reinhardt, A., P. Mogre, and R. Steinmetz (2011, march). Lightweight remote procedure calls for wireless sensor and actuator networks. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, pp. 172 –177.

Rivest, R. L. (1998a). Can we eliminate certificate revocation lists? In *Proceedings of the 2nd International Conference on Financial Cryptography*, London, UK, pp. 178–183. Springer-Verlag.

Rivest, R. L. (1998b). Can we eliminate certificate revocations lists? In *Proceedings of the Second International Conference on Financial Cryptography*, London, UK, pp. 178–183. Springer-Verlag.

Rivest, R. L. and B. Lampson (1996, October). SDSI — A Simple Distributed Security Infrastructure. Version 1.1, at `http://theory.lcs.mit.edu/~rivest/sdsi11.html`, October 2, 1996.

Rompf, T. and M. Odersky (2010). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth interna-*

*tional conference on Generative programming and component engineering*, GPCE '10, New York, NY, USA, pp. 127–136. ACM.

Sandhu, R. S., E. J. Coyne, H. L. Feinstein, and C. E. Youman (1996). Role-based access control models. *Computer 29*(2), 38–47.

Seamons, K., M. Winslett, and T. Yu (2001, February). Limiting the disclosure of access control policies during automated trust negotiation. In *Proceedings of the Symposium on Network and Distributed System Security*.

Seamons, K., M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu (2002). Requirements for policy languages for trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, Washington, DC, USA, pp. 68. IEEE Computer Society.

Seepold, R., N. M. Madrid, J. S. Gómez-Escalonilla, and A. R. Nieves (2009, January). An embedded software platform for distributed automotive environment management. *EURASIP J. Embedded Syst. 2009*, 5:1–5:10.

Sheard, T. and S. P. Jones (2002, December). Template meta-programming for haskell. *SIGPLAN Not. 37*, 60–75.

Shnayder, V., B.-r. Chen, K. Lorincz, T. R. F. F. Jones, and M. Welsh (2005). Sensor networks for medical care. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, New York, NY, USA, pp. 314–314. ACM.

Simon, R. T. and M. E. Zurko (1997, June). Separation of duty in role-based environments. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pp. 183–194. IEEE Computer Society Press.

Skalka, C., X. S. Wang, and P. Chapin (2007). Risk management for distributed autho-

rization. *Journal of Computer Security 15*(4), 447–489.

Society, I. C. (2003, October). IEEE std. 802.15.4 - 2003: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs). Standard.

Srinivasan, R. (1995, August). *RFC-1833: Binding Protocols for ONC RPC Version 2*. Internet Engineering Task Force.

Stubblebine, S. (1995). Recent-secure authentication: Enforcing revocation in distributed systems. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pp. 224–235. IEEE Computer Society.

Stubblebine, S. G. and R. N. Wright (1996). An authentication logic supporting synchronization, revocation, and recency. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, New York, NY, USA, pp. 95–105. ACM Press.

Szczechowiak, P., L. B. Oliveira, M. Scott, M. Collier, and R. Dahab (2008). Nanoecc: testing the limits of elliptic curve cryptography in sensor networks. In *Proceedings of the 5th European conference on Wireless sensor networks*, EWSN'08, Berlin, Heidelberg, pp. 305–320. Springer-Verlag.

Taha, W. (2004). Resource-aware programming. In *ICESS*, pp. 38–43.

Taha, W. and T. Sheard (1997). MetaML: Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, PEPM '97, New York, NY, USA, pp. 203–217. ACM.

Vairo, C., M. Albano, and S. Chessa (2008). A secure middleware for wireless sensor networks. In *Proceedings of the 5th Annual International Conference on Mobile and*

*Ubiquitous Systems: Computing, Networking, and Services*, Mobiquitous '08, ICST, Brussels, Belgium, Belgium, pp. 59:1–59:6. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Watson, M. (2013). Type checking implementation in scalaness/nest. Master's thesis, University of Vermont.

Whitehouse, K., G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler (2006). Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, New York, NY, USA, pp. 416–423. ACM.

Winsborough, W. H. and N. Li (2002, June). Towards practical automated trust negotiation. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*. IEEE Press.

Winsborough, W. H. and N. Li (2004). Safety in automated trust negotiation. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, pp. 147. IEEE Computer Society.

Winsborough, W. H., K. E. Seamons, and V. E. Jones (2000). Automated trust negotiation. In *Procedings of the DARPA Information Survivability Conference and Exposition. Volume 1*, pp. 88–102. IEEE Computer Society.

Winslett, M., N. Ching, V. Jones, and I. Slepchin (1997). Assuring security and privacy for digital library transactions on the web: Client and server security policies. In *Proceedings of the IEEE International Forum on Research and Technology Advances in Digital Libraries*, Washington, DC, USA, pp. 140–151. IEEE Computer Society.

Woo, T. Y. C. and S. S. Lam (1993). Authorizations in distributed systems: A new

approach. *Journal of Computer Security 2*(2-3), 107–136.

XSB Inc. (2006). XSB home page. `http://xsb.sourceforge.net`.

Yu, T., X. Ma, and M. Winslett (2000). PRUNES: An efficient and complete strategy for automated trust negotiation over the internet. In *Proceedings of the 7th ACM conference on Computer and communications security*, New York, NY, USA, pp. 210–219. ACM Press.

Yu, T., M. Winslett, and K. E. Seamons (2001). Interoperable strategies in automated trust negotiation. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, New York, NY, USA, pp. 146–155. ACM Press.