Dynamic Configuration of nesC Components in Scala

Peter Chapin
The University of Vermont
Burlington, Vermont
Email: pchapin@cs.uvm.edu

Christian Skalka
The University of Vermont
Burlington, Vermont
Email: skalka@cs.uvm.edu

Michael Watson
The University of Vermont
Burlington, Vermont
Email: mpwatson@uvm.edu

***Abstract*—Building programs for constrained embedded devices is challenging due to severe limitations on processing speed, memory, and network bandwidth. Staged programming can help bridge the gap between high level code refinement techniques and device level programs by allowing a first stage program to programmatically specialize device level code. Here we introduce *Scalaness*, a two stage programming system for wireless sensor networks. With Scalaness the first stage program is written in an extended dialect of Scala where components written in a reduced dialect of nesC are composed and specialized. This allows dynamic configuration of nesC components in Scala in a well-typed manner. We give an overview of Scalaness and focus particularly on how it allows second stage programs to be constructed from a mixture of generated components and library components.**

***Keywords*-staged programming, wireless sensor networks, scala;**

I. Introduction

It is well known that wireless sensor network (WSN) programming is challenging because of the severe resource constraints imposed by sensor network nodes. Often, WSN software benefits from iterative refinement over time as it adapts to environmental and network conditions. *Staged programming* [1], [2], [3], [4] is a technique that supports programmatic specification of code deployment cycles and code refinement. In staged programming, code is a datatype that can be dynamically modified, composed, and specialized. Thus, one program "stage" can specify how to generate code for "later" stages. This idea has been explored previously for embedded systems programming, since it allows to obtain highly efficient code based on dynamically-determined conditions [5], [6].

We visualize a *two* stage programming system for wireless sensor networks where the first stage runs possibly *in the field* on a relatively powerful base station or hand-held device. The first stage could make use of inputs known only during deployment or by querying the network after it has been deployed. For example, using information about the actual the number of neighbors each node has, the first stage program could optimize routing tables, or other neighbor specific data structures, to the smallest possible size. The resulting specialized node-level program could then be (re)deployed to the network. This process could even be automated and executed periodically to allow the network to adapt to changing conditions while remaining highly optimized. In this way our system differs from macro-programming systems such as Kairos [7], and Regiment [8] which support full network programming but no facility for iteratively generating code.

In our envisioned scenario it is critical that in all cases the first stage program generates a sensible, type correct residual program. The user of the first stage program will not be in a position to fix, or perhaps even understand, error messages produced by the second stage compiler. This is particularly true in the case where the first stage executes automatically without direct human oversight.

Existing staged programming systems are not suitable for this application domain. Many systems assume that the first and second stage languages are the same or that both stages execute in the same address space. In our scenario the two stages are widely separated with the first stage executing on a powerful machine under the control of a conventional operating system and the second stage executing on small embedded devices. The dramatic difference between those platforms requires that significantly different programming languages be used for the two stages.

Also in existing staged programming environments the type correctness of second stage programs is verified either during the running of the first stage program or, even later, during the compilation of the second stage program [3]. This is too late for our envisioned sensor network applications. Instead the type correctness of all possible residual programs must be verified during the compilation of the first stage program. Previous work on $\langle \mathrm{ML} \rangle$, a staged language calculus, provides this guarantee in the context of stages executing as separate processes [4]. $\langle \mathrm{ML} \rangle$ was developed explicitly to provide a static type safety argument and theoretical foundation for the more practical programming system introduced here.

In our system the programmer writes first stage programs in a dialect of Scala, a powerful functional and object oriented hybrid language for the Java Virtual Machine [9]. The first stage program specializes and composes modules of code written in a simplified dialect of nesC, a language commonly used on wireless sensor network nodes [10]. We call our system *Scalaness*.

The remainder of this paper is organized as follows. In

Sec. II we give an overview of Mininess, the language in which node-level module code is written. In Sec. III we describe the extensions to the Scala programming language that allow Mininess modules to be specialized and composed. In Sec. IV we focus on how our system allows programs to be constructed from a mixture of staged components and prebuilt components. In Sec. V we discuss our implementation. Finally in Sec. VI we conclude.

II. Mininess

To simplify our language design and to support strong type safety not present in the full nesC programming language, we have defined a reduced dialect of nesC we call *Mininess*. While Mininess retains much of the expression and statement structure of nesC, it simplifies nesC primarily in the way in which interfaces to modules are defined and used.

In Mininess a module's uses-provides list can contain only bare command declarations. Commands that are provided form the *exports* of the module, and commands that are used form the *imports* of the module. NesC-style events are not permitted but can be encoded with suitably defined commands (e.g. a used interface event is semantically equivalent to a provided command). In addition each module has a module type which is the signature of that module's imports and exports taken together. Generic components are supported at the Scalaness level as described in Sec. III. In particular, Mininess components support a form of bounded parametric polymorphism [4].

As an example, the modules in Fig. 1 constitute part of a secure messaging system. The `MessageBuilderC` module builds a message consisting of a header and a block of message data. It then calls a `computeToken` command to obtain an authorization token over the completed message. The type `HeaderType` and value `messageSize` are not declared in the Mininess code but instead are provided during the execution of the first stage program using a mechanism described in Sec. III. Notice that `AuthorizeC` does not require parameterization.

Mininess also defines a subtype relation between primitive integer types in an obvious way and a restricted form of record width subtyping. To make programming less error prone Mininess disallows implicit conversions from wide integers to narrow integers or between signed and unsigned integers.

III. Scalaness

In this section we describe the first stage language as an extension of Scala. We do not intend to restrict Scala in any way; the first stage programmer has full access to the entire language and supporting libraries. We extend the language by providing a way for the first stage programmer to declare, specialize, and manipulate Mininess components.

```
module MessageBuilderC {
  uses command computeToken(
    void *data, size_t size);
}
implementation {
  struct Message {
    HeaderType header;
    uint8_t    message[messageSize];
  };
  struct Message mes;
  ...
  call computeToken(&mes, sizeof(mes));
  ...
}


module AuthorizeC {
  provides command computeToken(
    void *data, size_t size);
}
implementation { ... }
```

Figure 1. Mininess Message Building Modules

A. nesC Mappable Types

A Scalaness program manipulates values that will ultimately be used as parameters to generic Mininess modules. These values cross the boundary between Scalaness and Mininess and are subject to special handling by the system. We define a *nesC mappable type* as a Scala type that has a corresponding nesC type. In a Scalaness program all nesC mappable types must be subtypes of the marker trait `NesCType`. NesC mappable types are also first class values in Scalaness programs as further discussed in section Sec. III-D.

The primitive nesC types are mapped from Scalaness by classes with names such as, for example, `Int32`, `UInt16`, and `Void`. Instances of these classes can be operated on in the expected way, allowing the Scalaness program to compute values that will be passed to the second stage Mininess program.

Structure types in Mininess are mapped from Scala classes and can be defined by the programmer. A user defined class C is nesC mappable if it obeys the following inductive rules.

1) C is not generic.
2) C is a subtype of `NesCType`.
3) All of C's fields are nesC mappable.
4) All of C's supertypes (except for `AnyRef` and `ScalaObject`) are nesC mappable.

There are otherwise no restrictions on the definition or use of nesC mappable classes in the Scalaness program. In particular, they are able to have convenience methods, although the methods of a nesC mappable class have no manifestation in the generated nesC code.

Fig. 2 shows two nesC mappable classes that correspond to Mininess structures used by the secure messaging

system introduced in Fig. 1. The `Header` class contains a TinyOS node ID and the identifier of a component on that node where the message will be delivered. The `TimeStampedHeader` subclass includes a time stamp on the header.

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

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

Figure 2. nesC Mappable Class Types

The type used for the node ID, `nodeIDType`, is computed during the execution of the first stage program as described in Sec. III-D.

B. nesC Components

A Scalaness program treats Mininess components as entire units. Each component is parameterized by upper bounded types or by typed values, which allows module genericity to be expressed. The types used with these parameters must be nesC mappable types.

A Mininess module is represented in a Scalaness program as a Scala class or object that extends the marker trait `NesCComponet`. Type and value parameters of the module are provided using the usual Scala syntax; a Scala object is only permitted when the represented Mininess module is not generic.

The last expression in the class's (or object's) primary constructor must be a string literal that names a file containing the Mininess code of the represented module. Fig. 3 shows the Scalaness representation of the Mininess modules in Fig. 1.

```
class MessageBuilderC[HeaderType <: Header]
  (messageSize: UInt8)
    extends NesCComponent
{ "MessageBuilderC.nc" }

object AuthorizeC extends NesCComponent
{ "AuthorizeC.nc" }
```

Figure 3. Scalaness Message Building Modules

The `HeaderType` type parameter and `messageSize` value parameter are used in `MessageBuilderC`'s Mininess code. Each instance of the class corresponds to a separate instance of the module, appropriately specialized. The `AuthorizeC` module, in contrast, is not generic and can be represented by a singleton Scala object directly.

Fig. 4 illustrates how generic Mininess modules can be instantiated using Scala's ordinary syntax for doing so.

```
val basicBuilder =
  new MessageBuilderC[Header](payloadSize)

val timeStampedBuilder =
  new MessageBuilderC[TimeStampedHeader](16)
```

Figure 4. Scalaness Module Instantiation

For illustrative purposes the payload size is dynamically specified in one case and statically in the other.

The Scalaness compiler infers the module type for the instances. Note that there is currently no syntax for Scalaness module types in Scalaness programs. This imposes some limitations on how modules can be used since there are places in Scala where type annotations are necessary.

C. Component Composition

NesC programs consist of components that are wired together in *configurations*. Like a module a configuration also has imports and exports, and thus a Scalaness module type, but it is implemented by connecting together other components. Scalaness allows two components with module types to be composed with the `+` operator. This creates a new configuration from the components and joins exports of one component to the compatible imports of the other. It is important to note that it is a type error if such a composition can't be done.

Fig. 5 shows the basic builder module being connected to the authorizer component. The resulting configuration is then composed with the timestamped builder module. Note that the exports of `AuthorizeC` are still available for connection after composition allowing both message builders to use the same authorizer.

```
val configuration =
  timeStampedBuilder +
    (basicBuilder + AuthorizeC)

validate: configuration
```

Figure 5. Component Composition

The `validate:` method applied to a component causes the specialized Mininess program to be generated, along with any supporting nesC configurations required. Following the theory developed in [4] if the `validate` expression type checks then the generated nesC program will be type correct.

D. Mappable Type Computation

An important feature of Scalaness is its ability to treat nesC mappable types as first class values dung the execution of the first stage. This allows a Scalaness program to dynamically compute which types are most appropriate to use in the second stage code. Type computation in the first

stage is done with the help of instances of the `Typedef` class defined in Fig. 6.

```
class Typedef[+T <: NesCType]
  (t: CovariantClass[T]) extends NesCType

class CovariantClass[+T](t: Class[T])
```

Figure 6. Representation of a nesC Type

Each `Typedef` instance wraps a normal Java `Class` instance representing a nesC mappable type. Its type parameter is an upper bound on the types that a particular instance may wrap. A `Typedef` instance may then be used syntactically as a type in Scalaness to, for example, specify the type of a field in a nesC mappable class as in Fig. 2, or passed as a type argument to a Mininess module.

Fig. 7 shows an example of a node ID type being selected based on dynamic information about the size of the network. Note that there is an implicit conversion from `Class` to `CovariantClass`.

```
val nodeIDType = new Typedef[UInt16](
  if (topology.size >= 256)
    Class[UInt16]
  else
    Class[UInt8]
)
```

Figure 7. Dynamic Computation with nesC Types

IV. External Libraries

Our preliminary experiments with Mininess 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 that we call *external libraries*. It is not our intention to require the whole program be generated by Scalaness. One external library of critical importance is the TinyOS operating system itself.

We provide for such interaction using an approach we call *interface unwrapping*. For example, suppose a component contains the following uses-provides list.

```
uses interface U;
provides interface P;
```

Although not strictly legal in Mininess we nevertheless allow specification elements indicating interfaces in Mininess programs. These interfaces are unwrapped according to the following rules:

1) When an interface is used, its commands become bare commands that are used by the component. Its events become bare commands that are provided by the component.
2) When an interface is provided, its commands become bare commands that are provided by the component. Its events become bare commands that are used by the component.

In all cases Mininess only deals with commands. For example, suppose `U` and `P` above are defined as shown in Fig. 8. These interfaces are unwrapped as also shown in the figure.

```
interface U {
  command void Uc(int);
  event   void Ue(int);
}

interface P {
  command void Pc(int);
  event   void Pe(int);
}

Unwraps to...

// Unwrapped from "uses interface U"
uses     command void Uc(int);
provides command void Ue(int);

// Unwrapped from "provides interface P"
provides command void Pc(int);
uses     command void Pe(int);
```

Figure 8. Example Interfaces with Unwrappings

Notice that when using lower level libraries, such as TinyOS, interfaces would only be used. However, we wish to allow for the possibility of using Scalaness to create libraries that would be combined with pre-existing high level code. In that case Mininess components may wish to provide a previously defined interface to that code.

It is also necessary to give the Scalaness programmer a way of specifying which library components will back the used (or provided) interfaces. The precise components to be incorporated into the final program are dynamically selected during the execution of the first stage program. Thus they are declared in the Scalaness code in a manner similar to the way Mininess components are declared, as described in Sec. III-B. For example an external library component named `LibraryC` could be declared in Scalaness as:

```
object LibraryC extends NesCComponent {
  external("LibraryC.nc")
}
```

Generic external components are represented by parameterized Scalaness classes as previously described.

The Scalaness compiler parses the component specification of the external component and unwraps the interfaces used and provided by that component. For each such interface the Scalaness compiler generates a shim module that forwards Mininess commands to the full nesC interface. When generic interfaces are used by the external component these shim modules are themselves generic.

Fig. 9 shows the shims generated for a library component `LibraryC` that uses interface `U` and provides interface `P`.

```
module Sclnss_U1 {
  provides interface U;
  uses     command void Uc(int);
  provides command void Ue(int);
}
implementation {
  command void Ue(int i)
    { signal U.Ue(i); }

  command void U.Uc(int i)
    { call Uc(i); }
}

module Sclnss_P1 {
  uses     interface P;
  provides command void Pc(int i);
  uses     command void Pe(int i);
}
implementation {
  event void P.Pe(int i)
    { call Pe(i); }

  command void Pc(int i)
    { call P.Pc(i); }
}
```

Figure 9. Shim Modules

The precise names used for these components are internal identifiers generated by Scalaness and are not intended to be used by the programmer. Instead Scalaness wraps the original library component and the shims into a configuration as shown in Fig. 10. If the original component was generic or used generic interfaces, then the wrapping configuration is also generic so that it properly corresponds to its representation in Scalaness.

```
configuration Sclnss_LibraryC {
  uses     command void Uc(int);
  provides command void Ue(int);
  provides command void Pc(int);
  uses     command void Pe(int);
}
implementation {
  components Sclnss_P1, Sclnss_U1, LibraryC;
  LibraryC.U  -> Sclnss_U1;
  Sclnss_P1.P -> LibraryC;

  Sclnss_P1.Pe -> Pe;
  Pc -> Sclnss_P1.Pc;

  Sclnss_U1.Uc -> Uc;
  Ue -> Sclnss_U1.Ue;
}
```

Figure 10. Configuration Wrapping Library Component

In effect the Scalaness compiler automatically converts an external library component into a Mininess component that can be manipulated by the first stage program. The Scalaness compiler extracts a module type for the component, based on its imports and exports, as usual. The component can then participate in the process of component composition allowed by the Scalaness language.

In the current implementation the Scalaness compiler does not attempt to type check the external library component. Presumably external components have been previously well tested. In effect, Scalaness treats them as entirely static; staging is only applied to a portion of the final application and to its overall configuration.

As we've described so far each external library component is wrapped in a separate configuration. However, a programmer could create a larger configuration of library components manually and then treat that entire configuration as a single entity inside the Scalaness program. The choice depends on the amount of control the programmer wants the first stage program to have on how library components are specified and composed. In some applications it may make sense for an entire subsystem of library components to be configured into a single entity ahead of time. In other applications the programmer may wish to control library configuration as part of the first stage execution.

V. Implementation

In this section we outline our implementation of Scalaness.

The standard Scala compiler is organized as a pipeline of *phases* that progressively lower Scala source code to JVM bytecode. The compiler has an architecture that allows plugins to inject new phases into that pipeline. The new phases have access to internal compiler artifacts, such as the abstract syntax tree, generated by the standard phases.

Fig. 11 shows the phase structure of the Scala compiler with the Scalaness plugin activated.

The Scalaness to Scala conversion phase exists to handle `Typedef` instances that are used as types and type parameters to Mininess modules. The conversion phase replaces such instances with a token type that will be acceptable to the Scala type checker. Additional information about the `Typedef` instance is recorded for later use by the Scalaness type checker.

The Scalaness type checker also parses and type checks the Mininess code associated with Scala `NesCComponent` classes. For each Mininess module the initial type environment includes the parameters of the module declared in the Scalaness program. The type checking is done before specialization; if it succeeds all instances of the module will be type correct [4].

The Scalaness type checker phase is also responsible for interface unwrapping, as described in section Sec. IV, and performs additional type checking of the Scalaness code based on the module types it infers. Notice that Scalaness type checking is done after Scala type checking. This

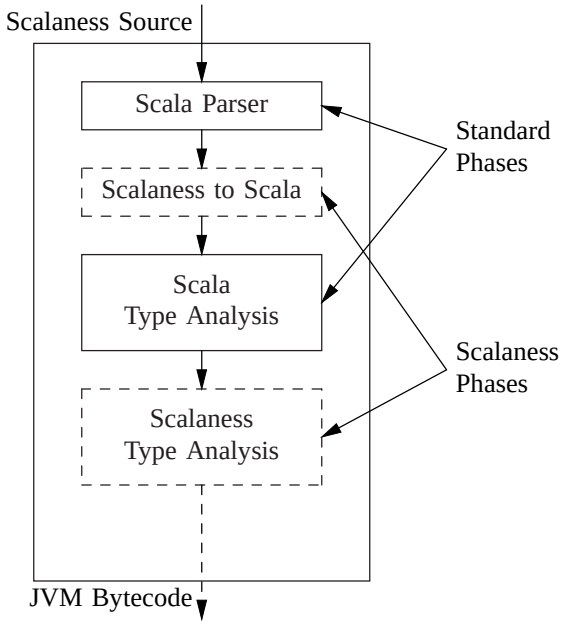

Figure 11. Scalaness Compiler Phases

simplifies the implementation by allowing the pure Scala parts of the program to be type checked using the existing, well tested Scala type checker. Scalaness typing is strictly more precise than Scala typing; a program that is Scalaness-typable is guaranteed to be Scala-typable.

Finally a runtime environment, made available through a library jar file, provides support for component composition. The runtime environment also performs the actual work of specializing the nesC code and outputting the residual program.

Progress on the implementation is ongoing. Currently the type checking of Mininess components, using type and term parameters provided by the Scalaness program, is complete. Mininess module types are inferred and are made available to the Scalaness type checker. Interface unwrapping as described in Sec. IV is partially complete. Type checking at the Scalaness level is currently being implemented.

VI. Conclusion

We have introduced Scalaness, a two stage programming system for wireless sensor networks. Scalaness provides a powerful programming environment for specializing and composing nesC modules in a type safe way; any type correct Scalaness program will generate only type correct residual programs.

Scalaness also provides a way to combine generated code with library code. It is not necessary to generate an entire program with Scalaness; instead only the parts of the final program that can benefit from specialization need be processed. The system is intended to support practical development of robust and efficient sensor network applications, especially those applications that can benefit from dynamic reconfiguration after deployment.

References

[1] W. Taha and T. Sheard, "MetaML: Multi-stage programming with explicit annotations," in *Proceedings of the 1997 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, ser. PEPM '97. New York, NY, USA: ACM, 1997, pp. 203–217. [Online]. Available: http://doi.acm.org/10.1145/258993.259019

[2] T. Sheard and S. P. Jones, "Template meta-programming for haskell," *SIGPLAN Not.*, vol. 37, pp. 60–75, December 2002. [Online]. Available: http://doi.acm.org/10.1145/636517.636528

[3] G. Mainland, G. Morrisett, and M. Welsh, "Flask: staged functional programming for sensor networks," in *Proceeding of the 13th ACM SIGPLAN international conference on functional programming*, ser. ICFP '08. New York, NY, USA: ACM, 2008, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/1411204.1411251

[4] Y. D. Liu, C. Skalka, and S. Smith, "Type-specialized staged programming with process separation," *Journal of Higher Order and Symbolic Computation*, 2011, accepted for Publication.

[5] W. Taha, S. Ellner, and H. Xi, "Generating heap-bounded programs in a functional setting," in *EMSOFT*. Springer, 2003, pp. 340–355.

[6] W. Taha, "Resource-aware programming," in *ICESS*, 2004, pp. 38–43.

[7] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, V. Prasanna, S. Iyengar, P. Spirakis, and M. Welsh, Eds. Springer Berlin / Heidelberg, 2005, vol. 3560, pp. 466–466. [Online]. Available: http://dx.doi.org/10.1007/11502593_12

[8] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th international conference on Information processing in sensor networks*, ser. IPSN '07. New York, NY, USA: ACM, 2007, pp. 489–498. [Online]. Available: http://doi.acm.org/10.1145/1236360.1236422

[9] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala, second edition*. Artima, Inc, 2011.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/781131.781133